Università di Pisa
Dipartimento di Informatica

Dottorato di Ricerca in Informatica

Ph.D. Thesis: TD-09/03

Dynamic shared data in structured parallel programming frameworks

Marco Aldinucci

December 2003

Address: Via F. Buonarroti 2, 56127 Pisa – Italy
Tel: +39-050-2212728 — Fax: +39-050-2212726
E-mail: aldinuc@di.unipi.it — Web page: http://www.di.unipi.it/∼aldinuc

Thesis Supervisors:

Prof. Marco Vanneschi, Prof. Marco Danelutto

Abstract

This work originates from the wish to simplify the coding of irregular applications within structured parallel programming environments. In these environments parallelism is exploited by composing "skeletons", i.e. parallelism exploitation patterns. The skeletal approach has been proved to be effective, at least if application algorithms can be somehow expressed in terms of skeleton composition. However, in some cases our skeletal frameworks fail in providing the application programmer with convincing solutions both from ease of programming and performance viewpoints. Major lacks of expressivity have emerged in dynamic/irregular algorithms and applications that oddly access to large data sets. The first part of the thesis moves along this path, and reports all attempts we made to improve the effectiveness of environments' compiler, static optimizer, and run-time support.

The main goal of the thesis is to take a step further with respect to the achieved results. In particular we aim to defeat expressivity lacks emerged in skeletal languages approaching irregular problems and dealing with dynamic data structures. The basic idea consists in providing the application designer with a shared address space and a skeletal framework that enables and enforces the co-design of (shared) dynamic data structures and (parallel) algorithms.

At this aim, a new skeletal programming environment based on shared address programming is proposed (i.e. eskimo). The language is as an extension of a "host" language (i.e. the C language). eskimo is conceived to be a framework to experiment how to support dynamic data structures in a skeletal framework. Its run-time support is based on a software distributed shared memory, and allows the programmer to freely access data items in the shared memory. eskimo is designed to match the hooks offered by ASSIST, thus to be experimented within. Notably eskimo is not yet another DSM, rather it relies on DSM already known technologies to experiment the co-design of dynamic data structures and parallel programming patterns enforcing locality in the distributed memory access.

eskimo has been designed and developed from scratch. eskimo run-time support exploits multithreading, dynamic data-driven scheduling, and is very tolerant with respect to standard POSIX programming framework. It will be released as open source package.

To Rosalia

Oscuramente, credette di intuire che il passato è la sostanza di cui è fatto il tempo; perciò questo diviene subito passato.

Jorge Luis Borges, *El Aleph*, 1952.

'You were quite entitled to make any suggestion or protest at the appropriate time, you known ... the plans have been available in the local planning office for the last nine months.'
'Oh yes, well as soon as I heard I went straight round to see them, yesterday afternoon. You hadn't exactly gone out of your way to call attention to them, had you? I mean like actually telling anybody or anything.'
'But the plans were on display ...'
'On display? I eventually had to go down to the cellar to find them.'
'That's the display department.'
'With a torch.'
'Ah, well the lights had probably gone.'
'So had the stairs.'
'But look, you found the notice, didn't you?'
'Yes, I did. It was on display in the bottom of a locked filing cabinet stuck in a disused lavatory with a sign on the door saying *Beware of the Leopard*.'

Douglas Adams, *The Hitchhiker's Guide to the Galaxy*, 1979.

Acknowledgments

A huge word of gratitude is owed to my supervisors Marco Vanneschi and Marco Danelutto. Both of them supported me far beyond I have expected, and actually they was the *deus ex machina* of my scientific experience. I hope to acquire their competence, and in particular their capacity to discern the real core of problems by throwing away in real time all misleading details.

I also took full advantage from my participation to several projects, which offered me the opportunity of meeting and discussing with many people. In particular I would remember Susanna Pelagatti, Sergei Gorlatch and Christian Lengauer.

I would like to acknowledge many of the persons I met in these years. My housemate Valentina *Von* Vivaldi who stayed awake up to improbable hours of the night trying to improve my English (almost uselessly). My room-mates at the department Nadia and Valentina who kindly tolerated my loudly arguing against the monitor while using the debugging tools (gdb, screwdriver, hammer, ...). Stefano from the *Aguaraja* kayak club; he taught me the *eskimo* technique and the importance of being self-confident (it may happen to be useful when you are sit upside-down in the middle of a creek, and sometimes also in research).

I also deserve my gratitude to the many people who shared with me the work, the spare time and the aperitifs. Amongst them Andrea, with whom I have many discussion about computer science, life, universe and anything (proudly, we hardly agree on anything). Emilio and Francesco who are able to drink more aperitfs than me (not an easy task).

Non posso dimenticare di ringraziare la mia famiglia: babbo Silvano e mamma Laura che non hanno ancora ben capito – non per loro colpa – che lavoro io faccia *di preciso*; mio fratello Piero, che oltre ad essere diventato un brillante informatico, ha assolto anche ai miei doveri familiari. Non so come, ma siete riusciti a lasciarmi vivere la mia vita pur essendomi sempre vicini.

Infine un pensiero per Rosalia. La sua naturale empatia ha reso il lavoro meno pesante; la sua spensieratezza la vita più felice. Un pensiero per i nostri viaggi in moto con il caldo e con il freddo, per il blu del mare ed il rosso del tramonto di San Vito ... un pensiero per tutto quello che ancora deve venire.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Information Technology advances in supercomputing, simulation, and networks are creating a new window into the natural world, making high end computational experimentation a vital tool for path-breaking scientific discovery.

Supercomputing is one of the foremost technologies in computing domain. We can see the imprint of this technology in many vital areas of scientific concern, such as forecasting global climate changes, monitoring nuclear reactors, enhancing automotive efficiency, modeling the evolution of galaxies, forecasting the flow of air over surface of vehicles and the damage due to impacts, and so forth.

In these areas, computational modeling is used to simulate physical phenomena that are impossible or very costly to observe through empirical means. Computational modeling allows in-depth analyses to be performed cheaply on hypothetical designs through computer simulation. In the coming years, computer simulation, spurred by technology changes already underway, can and should play an even greater rôle in providing solutions to our most challenging problems. However, a direct correspondence can be drawn between levels of computational performance and the problems that can be studied through simulation: each science and engineering application has a proper threshold of computing capacity (and cost) at which it becomes viable. And each era has its own Grand Challenge problems, i.e. problems that require a computing power threshold that falls far beyond the current availability.

Commercial and industrial computing has also come to rely on high performance architectures for its high end. Although for industrial needs the scale of computational performance is typically not as large as in scientific computing, they require a very aggressive development and deployment time for both hardware solutions and applications. In the past decade our group has been particularly active in transferring results from the research environment to the application marketplace. The pioneering work on $\mathsf{P}^3\mathsf{L}$ parallel language [27], its industrial deployment SkIE [29, 167], and lately on the ASSIST [169, 13, 12, 11] programming environment indeed take care of critical industrial requirements such as: rapid prototyping, performance portability, software reuse, integration and interoperability of parallel applications

with the already developed standard tools.

The widespread diffusion of high performance computing depends also on its ability to satisfy the needs of industrial users, whose main goal is to exploit potentiality of parallel machines, in a more modest-scale with respect to scientific computing, but with a more aggressive requirement in time to development and deployment both for hardware and applications.

Advancements in several technologies over the past few years have had a major impact on the computing arena. Today's application professionals have far more computing power available to them, thanks to the fast-paced growth in hardware technology both in "raw technology performance" (e.g. clock cycle, transistor density) and in "architectural performance" (e.g. pipelining). In spite of this, the demand for performance, propelled by both challenging scientific and industrial problems, is still increasing. Even assuming a very optimistic pace of growth in processor performance, very large parallel architectures are needed to address current challenging scientific/industrial problems in a reasonable time-gap. The importance of parallelism meeting the application demand for ever greater performance can be brought into sharper focus by looking more closely at the advancements in the underlying technology and architecture: growth of the instruction level parallelism and processor level parallelism both in commodity and high-end computing marketplaces.

Moreover, the machine peak power is not the only issue. Lots of efforts has been made in the software side by the research community in order to tame parallel machine peak power and turn it into application performance. Our direct experience in the design of parallel compilers and software environments enforces this trend. Efficiently supporting challenging social/industrial applications, which algorithmic solutions are often irregular and dynamic (notably massive data mining, computational chemistry), requires both very high computational power and high flexibility of the programming model. In turn, this requires a powerful but clean computational model at the hardware/software boundary.

1.1 Contributes and motivations of the thesis

This work originates from the wish to simplify the coding of irregular applications within our group programming environments. In these environments parallelism is exploited by composing "skeletons", i.e. parallelism exploitation patterns. From language viewpoint, a skeleton is a higher-order function that behaves as a pure function (no side-effects). Several real world, complex applications have been developed using these environments.

The skeletal approach has been proved to be effective, at least if application algorithms can be somehow expressed in terms of skeleton composition. However, in some cases our skeletal frameworks fail in providing the application programmer with convincing solutions both from ease of programming and performance view-

points. Major lacks of expressivity have emerged in dynamic/irregular algorithms and applications that oddly access to large data sets. The first part of the thesis moves along this path (Section 2), and reports all attempts we made to improve the effectiveness of environments' compiler, static optimizer, and run-time support.

Stimulated by application requirements we eventually changed also the programming model, thus the role of skeletons in the language. In order to easily manage large data sets we designed a new environment (i.e. ASSIST) that explicitly admit a shared state among processing elements.

These research results as has been already presented in published papers. Some of those originated from the design, development and experimentation of software packages. In particular, I participated to the design or development of the following programming environments:

- SkIE (1998) programming environment and its compiler [7];
- FAN (2000), a functional skeletal parallel programming framework [18];
- Lithium (2001), a pure Java parallel programming environment [15, 16, 17];
- ASSIST (2002) programming environment [11, 12, 13];

also, I designed and developed the following programming platforms:

- Meta (1999) optimization tool and for skeleton-based languages [8, 9];
- Skel-BSP (2000), a skeletal language and its run-time on top of the Padeborn University BSP-library [9];
- eskimo (2002) language and its run-time support [10] (Chapters 4, 5 and 6).

All software packages except SkIE are available as open source.

The main goal of the thesis is to take a step further with respect to the achieved results. In particular we aim to defeat expressivity lacks emerged in skeletal languages approaching irregular problems and dealing with dynamic data structures. The basic idea consists in providing the application designer with a shared address space and a skeletal framework that enables and enforces the co-design of (shared) dynamic data structures and (parallel) algorithms. We shall go further in the discussion in Section 1.3.

At this aim, in the second part of the thesis (Chapters 4, 5 and 6) a new skeletal programming environment based on shared address programming is proposed (i.e. eskimo). The language is as an extension of a "host" language (i.e. the C language). eskimo is conceived to be a framework to experiment how to support dynamic data

structures in a skeletal framework. Its run-time support is based on a software distributed shared memory, and allows the programmer to freely access data items in the shared memory. eskimo is designed to match the hooks offered by ASSIST, thus to be experimented within. Notably eskimo is not yet another DSM, rather it relies on DSM already known technologies to experiment the co-design of dynamic data structures and parallel programming patterns enforcing locality in the distributed memory access.

eskimo has been designed and developed from scratch, and actually is a pretty young product. eskimo run-time support exploits multithreading, dynamic data-driven scheduling, and is very tolerant with respect to standard POSIX programming framework (notably, it does not use signal-handlers). It will be released as open source package.

In the next section we shall recap some technical background of the work. In Section 1.3 we shall return back on thesis contributes and motivations in the light of introduced concepts.

1.2 Technical background

Dealing with sequential machines, we generally take programs for granted: the field is mature, and there is a large base of programs that can be viewed as fixed. We optimize the machine design against the requirements of these programs. Although we recognize that programmers might further optimize their code, we usually evaluate new designs without anticipating such software changes. Compilers may evolve along with architecture, but the source program is still treated as fixed. In parallel architecture, there is a much stronger and more dynamic interaction between the evolution of machine designs and that of parallel software. Since parallel computing is all about performance, programming tends to be oriented towards taking advantage of what machines provide.

In the next section we shall briefly analyze trends in parallel architectures and their building blocks, on this basis we shall choose a parallel architecture class, and we shall highlight what the peculiarities of machines in the class are. Along the discussion we shall put aside a number of findings, that eventually we shall elaborate to distill our wish list on the programming model, and on what are the issues it should cope with.

VLSI evolution. The VLSI technology trends may help us to understand what architectural directions may be adopted. Historically, Dynamic RAM (DRAM) has been recognized as the technology drivers for the whole VLSI industry. Prior to the early 1990s, logic (e.g. processor) technology was developed at slower pace than DRAM technology. During the last few years, the development rate of new technolo-

Moore's Law: A historical observation by Intel executive, Gordon Moore, that the market demand (and semiconductor industry response) for functionality per chip (bits, transistors) doubles every 1.5 to 2 years. He also observed that MPU performance [clock frequency (MHz) x instructions per clock = MIPS] also doubles every 1.5 to 2 years. Although viewed by some as a "self-fulfilling" prophecy, "Moore's Law" has been a consistent macro trend, and a key indicator of successful leading-edge semiconductor products and companies for the past 30 years.

Table 1.1: Moore's Law

gies used to manufacture microprocessors has accelerated, closing the technology gap with DRAM. Currently, both DRAMs and microprocessors are increasing in functions per chip more than a factor of 30 per decade, accordingly with Moore's Law (see Table 1.1). As shown in Table 1.2, the Semiconductor Industry Association (SIA) foresees the same trend also for the next decade [152].

However, logic and DRAM technology, both under strong market pressure, have followed different evolutions: DRAM technology has moved towards the reduction of costs, the increase of storage room and the productivity of assembly lines, which are dominated by cell density and chip size. Microprocessor technology has also moved towards the reduction of costs, but with the additional target of maximizing the performance, which is dominated by the length of the transistor gate and by the number of interconnected layers.

The large availability of resources have allowed an ever increasing number of parallel functional units in microprocessor design leading an increasing number of instruction per clock cycle. This "architectural" improvement coupled with clock frequency improvement is leading microprocessors performance (ops/second) to increase by more than a factor of 15–30 per decade[1]. In the same time-gap, DRAM cycle times is improving much more slowly, roughly a factor of two per decade. Currently, hundreds to thousands clock cycles are needed to service an off-chip cache

[1]The factor is 100–200 if considered on the basis of floating point operations.

Year	1999	2001	2003	2005	2008	2011	2014
Logic Gate Length *(nm)*	180	150	120	100	45	30	20
Logic Cost-Perf. *(Mtrans/chip)*	24	48	95	190	539	1523	4308
Logic High-Perf. *(Mtrans/chip)*	110	220	441	882	2494	7053	19949
DRAM *(Gbits/chip)*	1.07	2.15	4.29	8.59	24.3	68.7	194

Table 1.2: Semiconductor Industry Association (SIA) density forecast for logic (processor + cache) and DRAM [152]. Logic Cost-Perf. includes a little L1 cache, Logic High-Perf. includes large L1+L2+L3 caches.

	Pentium III (600MHz)	**Athlon (800MHz)**
No. of transistor (L1 cache only)	$\sim$ 9.5M	$\sim$ 21M
Operation per clock cycle	5	9
Integer+Floating pipelines	2+1	3+3
L1 cache size	32KB	128KB
L2 cache size (up to)	2MB	8MB
cache coherence support	MESI	MOESI/MESI
multiprocessing support	shared bus	point-to-point

Table 1.3: Intel Pentium III and AMD Athlon: A lot of instruction level parallelism and large caches.

miss in an uniprocessor box; such gap between processor speed and memory speed is likely to continue to widen in the coming years.

Together with parallelism, the other way to translate such large volume of transistor into performance is locality. Caches, through locality, help to maintain data close to processor, thus reducing the frequency of accesses at deeper levels of the storage hierarchy. Indeed, as we can see in Figure 1.1, since caches migrated on chip in the mid-1980s, the fraction of the transistor on commercial processor that are devoted to caches has risen steadily. As a matter of fact, today's microprocessors are mostly memory.

Let us make concrete our arguments with a couple of examples: The Intel Pentium III and the AMD Athlon. These are last generation commodity processors with a good cost-performance trade-off (not high-performance), Table 1.3 briefly describes their features [2]:

It is worth pointing out that both processors, in order to issue up to 5–9 instruction per clock cycle, internally exploit at least three different paradigms of parallelism: pipeline, functional units replication and specialization. Both processors support also up to 2–8 MBytes of L2 cache. In addition, both processors already contain hooks (such as snoopy buses) to work in a cache-coherent shared memory multiprocessor framework.

Let us put aside a first finding: Modern microprocessors are complex and exploit substantial instruction level parallelism. Large caches are used to provide a great deal of local storage in order to satisfy bandwidth requirement of processor functional units with a reasonable latency. Moreover, in order to stay on the processor performance growth trend, the increase in the ratio of memory access time to processor cycle time will require that the processors employ better latency avoidance and *latency tolerance* techniques. In addition, the increase in processor instruction rates, due to the combination of cycle time and parallelism, will demand that the *bandwidth* delivered by the memory increases.

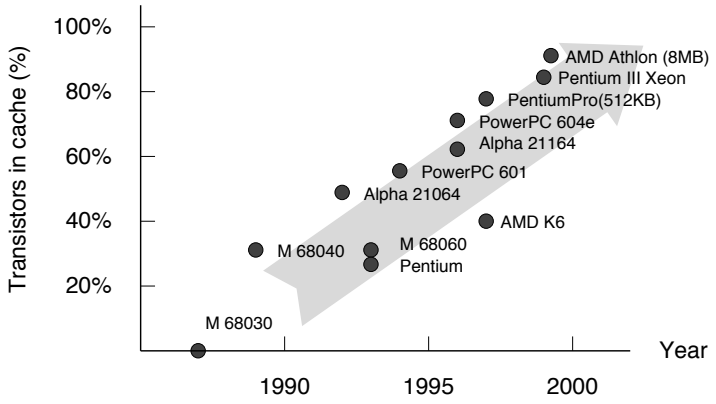


Figure 1.1: Fraction of transistors on microprocessor chip devoted to caches.

Parallel architectures. Current trends would suggest that, simply riding the commodity growth curve, we could look towards achieving petaflops-scale peak performance in a decade, or even earlier if the scale of parallelism is increased. It is less clear what level of communication performance these machines will provide.

As discussed, modern commodity microprocessors already includes hooks for implementing multiprocessor systems. Nevertheless, a large multiprocessor system needs a pretty efficient memory sub-system to feed many processors. Currently, the design and the implementation of memory sub-system is the main cost of large multiprocessors. As a matter of fact, the better price/performance trade-off is reached in low-scale multiprocessors. They may be in turn put together using high-speed networks, in facts turning every LAN into a potential parallel machine. These machines (namely Beowulf class clusters or simply clusters) are gaining more and more interest as low-cost parallel architectures, and actually more positions in the TOP500 [164] parallel architecture list. Currently ASCI clusters and SMP clusters occupy 112 positions of the list. Although these parallel machines cannot be classified as low-cost Beowulf clusters – mainly due to the adoption of high-end interconnection networks and to the number of nodes used – the architectural trend is clear [70].

In summary, microprocessors in a Beowulf machine may exchange data one each other using different means: the shared memory (within the single node) and the network (among the nodes). In general, these means support different data exchange protocols in native manner, i.e. a shared address space and message passing. Nevertheless, as we shall see in the next section, we can simulate one programming model using the other in order to supply the programmer with an uniform programming model. Therefore, let us assume a Beowulf machine as a big multiprocessor exploiting an extended (multi-level and distributed) memory hierarchy.

The two key factors in VLSI evolution discussed in previous section (i.e. the increase of single chip performance as well as on-chip storage capacity), and in parallel machines evolution (Beowulf) will cause the storage hierarchy to continue

becoming deeper and more complex.

The trend towards deeper hierarchies presents a problem for parallel architectures since communication, by its very nature, involves crossing out the lowest level of the memory hierarchy on the node, leading a growth of the actual latency on operations that crosses the processor chip boundary. The problem is further exacerbated in Beowulf class machines that in general exhibit a heavily unbalanced communication bandwidth/computing power ratio.

Let us put aside our second finding: *memory systems in parallel machines are constructed as a hierarchy of increasingly larger and slower memories: on an average, a large hierarchical memory is fast, as long as as the references exhibit good locality.*

1.2.1 Parallel architectures: programming model

Message passing and shared address space represent two clearly distinct programming models, each of them providing a well-defined paradigm for sharing, communication and synchronization. Historically, parallel architectures have been designed in such a way to naturally support a specific programming model.

A shared memory system makes a global physical memory equally accessible to all processors. These systems naturally offer a shared address space, i.e. a convenient programming model enabling simple data sharing through a uniform mechanism of reading and writing shared structures in the common memory (see Figure 1.2 a). However, such systems typically suffer from increased contention and latency in accessing common shared memory, which in general limits the scalability compared to other memory organizations.

Multicomputers consist of multiple independent processing nodes with local memory modules, connected by a general interconnection network. The scalable nature of the distributed memory makes them scalable and very powerful in computing peak power. The natural programming model is message passing, that is widely considered exacting from programmer viewpoint because of the explicit data distribution of data structures in the distributed memory (see Figure 1.2 b).

The evolution of the hardware and software has blurred the clear boundary between the multiprocessor and multicomputer machine organizations. The convergence has been driven by many factors. One is clearly that all of the approaches have common requirements. They require a fast, low latency and robust interconnect. They all benefit from hiding as much of the communication cost as possible: the goal is often pursed by equipping commodity processors (reducing also development cost and time) with specialized *communication co-processors* (e.g. smart communication boards like Myrinet and SCI).

On the software side, the convergence of multiprocessor and multicomputer machine schemes is enforced by the possibility of simulating one programming model using the other, and vice versa.

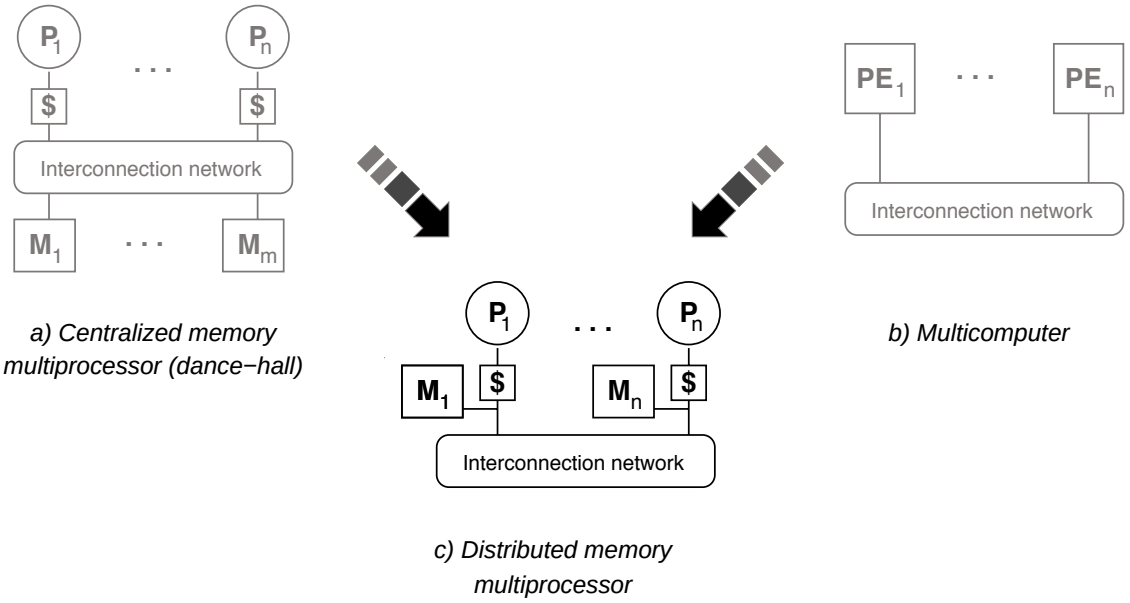


Figure 1.2: Typical parallel machine schemes.

- Classical message passing operations (send/receive) may be supported on shared memory machines through shared buffer storage. Send involves writing data into the buffer, while receive involves reading the data from the shared buffer. Flags into the buffer are used to enforce mutual exclusion (locks) and to indicate a category of events such as message arrival.

- User processes on a message passing machine may construct a global address space by carrying along pointers specifying the process and the local virtual address in that process. A logical read is implemented by sending a request to the process containing the object and receiving an answer. The read/write simulation may be hidden from the user by carrying out library or compiler generated code. As we will see later in Chapter 3, a shared virtual address space can be established at the page level. Since only local pages are directly accessible with such machines, pages instead of single words or cache lines are actually moved/copied among processing elements, in fact coarsening the granularity of messages.

The 1990s have exhibited the beginning of a convergence among these various factions. Propelled by both economical and technical issues, shared memory and multicomputer schemes have converged towards a common organization, represented by a collection of complete computers, augmented by a communication co-processor connecting each node to a scalable communication network. Focusing on the memory organization, we call them *distributed memory multiprocessors* (see Figure 1.2 c). Such systems, depending on co-processor functionality, may act as multiprocessor or

multicomputer, thus providing a shared address space or a message passing model at the assembler level [65]. Such co-processor may also be implemented in software. A software DSM may be used to provide a multicomputer with a shared address space; therefore it may turn a Beowulf class cluster into a multiprocessor. Since (Beowulf + software DSM) multiprocessors are both cheap and flexible, they are quite good platform to experiment solutions at hardware/software boundary like mapping, scheduling, caching strategies and memory consistency models.

Making the memory hierarchy efficient. As we shall see in Chapter 3, many methods have been proposed to alleviate latency/bandwidth problems in multiprocessor architectures. Almost all approaches rely on the following findings:

1. reduce communication and synchronization cost as seen by the processor;
2. reduce serialization at shared resources, reduce communication volume;
3. reduce non-inherent communication via data locality;

Almost all approaches to face the first issue consist in hiding long latency memory communications by overlapping them with computation or other communications. Actually, there are many ways to implement, both in hardware and software, this basic idea: prefetching, multithreading, non-blocking transactions and so forth. Some of these methods are currently supported in shipped multiprocessor and parallel applications with various mileage. However, many of the latency tolerance techniques increase the absolute amount of memory traffic by fetching more data than are needed, also creating contention in the memory system [46].

Relaxed memory consistency models essentially face up the first and the second issues by enabling the hiding of remote memory access latency and the reduction of coherence-oriented handshakes. The research community has proposed a plethora of relaxed memory consistency in the past decade, which are collected and discussed in Section 3.

Third issue is probably the foremost from our viewpoint. Memory systems in modern multiprocessor are constructed as a hierarchy of increasingly larger and slower memories: on an average, a large hierarchical memory is fast, *as long as as the references exhibit good locality.* This is particularly true in (Beowulf + software DSM) machines, where at least one layer of virtual memory is implemented through network transactions. At this end, software DSMs rely on relaxed memory consistency in order to reduce communication frequency and volume.

1.2.2 High-level parallel programming

Although building parallel computers has become easier, programming parallel computers can still be quite difficult. Most application programs are currently being written at the low level of C or Fortran, combined with a communication library

like MPI; moreover, they are often tuned towards one specific machine configuration. Since parallel computers are typically replaced within five years, parallel programs which live longer have to be re-tuned or redesigned. In addition, programming at this low level of abstraction is cumbersome and error-prone.

In sequential programming, coding for a specific machine also prevailed three decades ago. The software engineering solution to overcome it was to introduce levels of abstraction, effectively yielding a tree of refinements, from the problem specification to alternative target programs [137]. The derivation of a target program then follows a path down this tree. The transition from one node to the next can be described formally by a semantics-preserving program transformation or refinement. Conceptually, porting a program to a different machine configuration means backtracking to a previous node on the path and then following another path to a different target program.

In the parallel setting, high-level programming constructs and a refinement framework for them are necessary due to the inherent difficulties in maintaining the portability of low-level parallelism [61]. In the 1990s, the "skeletons" research community [59] has been working on high-level languages and methods for parallel programming [27, 29, 17, 18, 45, 77, 98]. Skeletons are higher-order functions which can be evaluated efficiently in parallel. They specify abstractly common patterns of parallelism which can be used as program building blocks. Typical skeletons model data and task parallel paradigms (see Chapter 2) as for example the pipeline, task farm, reduction, scan and "sequential". Sequential skeleton is a final skeleton embodying sequential chunks of user code. Such code is guaranteed to be executed in a sequential fashion. Sequential skeleton enables the reuse of already developed (and tested) application source code and libraries.

In the past decade our research group has been active in experimenting new technologies for high-level parallel programming. These are mainly targeted to simplify programming by raising the level of abstraction; to enhance portability by absolving the programmer of responsibility for detailed realization of the underlying parallel paradigms; to improve performance by providing access to carefully optimized implementations of the paradigms. These technologies have been used to design programming environments and languages and to implement their compilers. Skeletons have been present all along in programming environments, even if their role has been permanently changing (see Section 2.1). Cole's skeletons represent parallelism exploitation patterns that can be used (instantiated) to model common parallel applications. Later, different authors acknowledge that skeletons can be used as constructs of an explicitly parallel programming language, actually as the only way to express parallel computations in these languages [76, 27]. Recently, the skeleton concept evolved, and became the coordination layer of structured parallel programming environments [26, 29, 147, 169].

Current skeleton-based systems typically provide the user with a collection of high-level skeletal constructs and with a compiler for translating skeleton programs into low-level target code [27, 56, 138, 153]. Alternatively, high-level skeletal con-

structs are used to equip a sequential or already parallel language, in facts extending it; in this case skeletons' implementation code is collected in libraries [17, 72, 75, 120]. Typically, skeletons carry a large amount of information on program interaction structure, which can be used by the compiler/run-time support to exploit efficient code on different target machines. Compiler approaches mostly rely on static optimization of the generated code, while libraries follow a more dynamical way.

Static versus dynamic run-time support. It is an old story, and clearly we have nothing to add up to the general question. Let us restrict the context to skeleton-based languages.

Early approaches to skeletons implementation has followed the static way. Such approaches were based on the concept of *implementation template*, i.e. a parametric processes network. Given a program, i.e. a particular nesting of skeletons, the compiler turns it into a processes network. Together with processes network a mapping plan is generated. At the run time a program loader places processes on parallel machine nodes according to the mapping plan. Each process of the network has a fixed role, that is established by the compiler.

The basic idea under the approach is that the compiler may "compositionally" build the final processes network by associating a processes network (taken from a library) to each skeleton. In case a skeleton is nested into another the implementation template of the inner skeleton is somehow merged with the outer one. During the merge process several optimizations on the process network may be performed, basically by merging some of adjacent processes in a single one. Moreover, the processes network may be targeted to a given physical network topology. Eventually, a (flat) processes network is generated. This network is still parametric and may be instanced with typical architectural constants as for example computation and communication grain.

However, high performance is only reached if a composition of skeletons is found which matches both the application and the target machine requirements. One possible way to address the problem is to integrate skeletons with refinement framework. The basic idea follows the mainstream of code optimization for sequential languages: the skeleton-based program is optimized by means of source-to-source semantic-preserving refinements targeted to obtain a "better" source code. Furthermore, due to the fact that the skeletons have a clear functional and parallel semantics, different rewriting techniques have been developed that allow skeleton programs to be transformed/rewritten into equivalent ones achieving different performances when implemented on the target architecture [38, 94]. One further concept, not as crucial in sequential programming, has to be added: the program refinements must be adorned with a cost model, since "efficiency" is the main – often the single – reason for using parallelism. A cost model actually formalizes the comparison between two (semantically-equivalent) source codes. In the past decade this "efficiency" was primarily interpreted as "speedup". Nowadays – particularly in a distributed par-

allelism scenario – it assumes a larger meaning involving efficient use memory room and network bandwidth, etc.

Several cost models have been developed [158, 159, 14, 172] and have confirmed that porting a parallel program from one machine configuration to another may dramatically alter its performance [94]. Therefore, program design tools must apply transformations based on performance predictions made in a cost model.

At this end we designed and developed the Meta tool, i.e. an optimization tool for skeleton-based programs. Given a skeleton-based language and a set of semantic-preserving transformation rules, the tool locates applicable transformations and provides performance estimates, thereby helping the programmer in navigating through the program refinement space. Meta is described is Section 2.2. It has been used as optimization engine of several skeleton-based programming frameworks [18, 8, 9]. Meta has proved to be an effective optimization tool, provided that a rich set of rewriting rules exists and is equipped with a careful cost model. The cost model is crucial in order to make correct decision both during the compiling and optimization process.

Unfortunately, these assumptions have been proved hardly satisfied in practice. It is worth highlighting two major problems in that:

- It is pretty difficult to make assumptions about the computational cost of sequential parts of the application (i.e. parts within sequential skeletons). They may call functions that are not available in the source form; and even in case all sequential source code is available, its correct cost profiling is possible in limited cases only, i.e. cases in which sequential code behaves in a very predictable manner and its cost does not (heavily) depend on input data.

- Defining a compositional cost model for a generic target platform is almost impossible. One possible way to avoid the problem consists in constraining how the target platform behaves in respect of synchronizations. For example, fairly good predictions can be figured out assuming a BSP model [165] (see also Section 2.2.4 [9, 172]). However such constraints prevent the both run-time support designer and application programmer to exploit all performance capabilities of current technology.

Indeed, recently more dynamical ways to support skeletons have been explored. These are aimed at defeating the disadvantages relative to the need of a cost model both in compiling and optimizing a skeleton program. It has been observed that skeletons naturally impose a data flow relationship among the computations performed by the different processes in the process graph implementing the skeleton program on the target machine. Therefore it is possible to derive a graph of macro data flow instructions from the skeleton code in such a way that:

- each sequential portion of code in the skeleton program denotes a macro data flow instruction;

- and the parameter passing mechanism used in the skeleton program defines the arcs between such instructions.

That graph can be used to execute the parallel application by making each processing element in the target machine to behave as a data flow interpreter of the instructions of the graph [70]. The mechanism exploits some ideas from previous work on macro data flow developed in rather different contexts [114, 135].

The macro data flow implementation technique has been adopted in the run-time support of the Lithium parallel programming environment (described in Section 2.3 [17]) and in the SKIPPER project [153]. Macro data flow has demonstrated to be effective in design of the run-time support. The universality of macro data flow interpreter greatly simplifies the mapping and scheduling problems in respect of static approach. In addition, we demonstrated that some of the optimization techniques developed for the static case are still applicable and useful in the new approach (as for example the "normal form reduction" [15]). Overall, the macro data flow run-time support has demonstrated to overcome many of the problems of fully static skeletons' run-time support (introducing a limited overhead).

Lack of expressivity in classical skeleton approaches. In spite of the good results achieved in the run-time support design, the classical skeletal approach still has a major problem. Many parallel applications are not obviously expressible as instances of skeletons, whether existing or imagined. Some have phases which require the use of less structured interaction primitives. Some have conceptually layered parallelism, in which skeletal behavior at one layer controls the invocation of operations involving such ad-hoc parallelism within. Recently, Cole has written that *"It is clearly unrealistic to assume that skeletons can provide all the parallelism we need"* [60].

We believe the point a bit subtler. Programming with classical skeletons has strong affinities with functional programming. Really, we are convinced that some problems have not a straightforward solution in terms of skeleton nesting. The problem is further exacerbated by the fact we pretend to put ready-made code inside the sequential skeleton: C/C++ code relying on side-effects included. As a matter of fact, the problem has been emerged while developing several real world, complex applications using our group skeletal framework (SkIE [29]), especially in the parallel data-mining area [35, 24, 63]. During the development of such applications, the skeletal approach has been proved to be effective, at least whether application algorithms can be somehow expressed in terms of skeleton composition. However, in some cases we have been forced to adopt "bizarre" solutions to overcome the following problems:

- The parallel paradigm is not directly expressible as skeletons composition.
- The parallel paradigm is expressible as skeletons composition, but is not efficient. These cases regard mainly the irregular access to big data structures.

- The application requires a pro-active/reactive behavior in respect of the operating system or devices. Sequential parts of the application require a strong interaction with operating system mechanisms (threads management, GUI and devices management, etc.).

We pragmatically decided to change strategy. We moved to a different (probably lazier) interpretation of "what a skeleton is". Actually, such change of perspective has been involving the whole skeleton community. A complete discussion on this point can be found in Section 2.1 and Chapter 4, and we report here only the ending findings.

Structured parallel programming should build bridges to the programming standards of the day, refining or constraining only where strictly necessary. It should respect the conceptual models of these standards, offering skeletons as enhancements rather than as competition. We should construct our systems to allow the integration of skeletal and ad-hoc parallelism in a well defined way.

1.3 A deeper look at framework and motivations

In the light of previous section findings we can precisely frame the work. Our research group has been working for a long time in high-level parallel programming. We developed several tools and programming environment for structured parallel programming[2], yielding several good results. Such frameworks discipline parallelism within the application thus offering a high-performance implementation but seriously limiting the expressive power. The development of complex real world applications has been proved sometimes difficult due to expressivity constraints of our skeletal frameworks. The impossibility of accessing to very large data bases in a possibly odd and unpredictable way has been one of the foremost limitations of these frameworks.

Current and future high performance computing will be able to deal with very complex applications that exploit both dynamic and irregular communication patterns. The programming of such applications will involve thousands of interrelated threads, and possibly many odd interactions among them. Large and complex parallel applications will require to solve parallelization problems as well as efficiently integrate heterogeneous objects, such as existing software modules, application libraries and packages. These targets are viable only by means of software-engineering methodologies as well as high-level languages equipped with effective developing tools. In turn, such tools require a clean interface of the system implementation (hardware, operating system).

We eventually decided to design and develop a new programming environment (i.e. ASSIST, see Section 2.4). ASSIST accommodate a *external object space*, that may include objects/data structures in a (distributed) shared memory.

[2]among the others P^{3}L, SkIE, Lithium, ASSIST parallel programming frameworks and Meta optimization tool, see Section 2.1.

The shared memory abstraction well suited a for future systems for many reasons:

- It agrees the hardware evolution towards ever deeper memory hierarchies.
- It seems well-suited for the solution of highly irregular problems.
- It enables the development of latency tolerant techniques at the level of hardware/software interface, thus it supplies a clean and uniform view of the machine to software tools.
- It has been deeply discussed in the literature, we can take advantage of the already developed work in the field.

We imagined eskimo, Easy SKeleton Interface (Memory Oriented), as an experimenting platform for shared memory abstraction within the ASSIST environment.

1.3.1 eskimo motivations

eskimo is an extensions of the C language. eskimo run-time support implements a software DSM and presents it within an original skeletal framework. The motivation under eskimo is not to build yet another DSM or to propose yet another relaxed memory consistency model. eskimo takes advantage of already developed work in the field, possibly using it in an original manner. As far the main design philosophy concern, we would like to draw an analogy with DSM-PM2 system; we report here some of sentences motivating DSM-PM2 [19]:

Most approaches to DSM programming assume that the DSM library and the underlying architecture are fixed, and that it is up to the programmer to fit his program with them. We think that such a static vision fails to appreciate the possibilities of this area of programming. We believe that a better approach is to provide the application programmer with an implementation platform where both the application and the multithreaded DSM consistency protocol can possibly be co-designed and tuned for performance. This aspect is crucial if the platform is used as target for a compiler: the implementation of the consistency model through a specific protocol can then directly benefit from the specific properties of the code, enforced by the compiler in the code generation process.

During the design of eskimo we rely on a similar philosophy. We would like to build a implementation platform primarily targeted to exploit dynamic data structures in a software DSM framework. Such data structures, their allocation strategy, the consistency model they obey, the operations needed to tackle with them are not fixed. They have to be co-designed with them. eskimo is built on top of a hierarchy of two run-time layers, each of them providing mechanisms and policies to solve some of parallel programming run-time support issues. The lowest-level tier is designed to fall in the "high-level programming environment designer's hands", not in the casual parallel programmer's hands. The highest-level is thought to experiment what are

the typical parallel paradigms (or skeletons) we need to tackle with dynamic data structures and irregular applications.

eskimo language is an extension of the C language. The basic idea behind eskimo is that a programmer should concentrate on structuring his data structures and his algorithms. An eskimo program is not parallel ab initio. Just as in a serial program, a program starts as a single flow. The programmer may split the flow of control in potentially concurrent flows by means of language primitives. The run-time may turn concurrency into parallelism. Moreover, in order to obtain a high-performance application, the programmer ought to structure its application properly, and eventually suggest to run-time important information about algorithm data access patterns. eskimo run-time takes care of all other details like process scheduling and load balancing. In this setting, skeletons are not objects of the language, but rather "suggested" programming patterns (see Section 2.1). The programmers may decide to structure their applications according to them, but they are not forced to do it.

1.3.2 eskimo features

eskimo is conceived as a framework to experiment and prototype dynamic data structures (and their support) in a structured parallel programming environment. We mainly expect it to gain expertise from the work in order to apply it to more general and complete programming frameworks (as for example the ASSIST environment [169, 13, 12, 11]). In addition, we would take in account a loosely coupled parallel architecture in the perspective to port the experience in a GRID framework. For this reason we focused on a Beowulf architecture as target parallel architecture.

As previously mentioned such architectures, especially whether used as multiprocessors, have to be carefully programmed to be efficient. In particular we would like to exploit parallelism in a hierarchical fashion. We can recognize (at least) three different levels of parallelism in a typical architecture: parallelism among PEs, parallelism among processors within a PE, instruction-level parallelism within a processor both at the level of execution pipelines and processors SIMD extensions (e.g. Pentium MMX/SSE, PowerPC Velocity Engine, etc.). These levels have different peculiarities and become efficient at different computation grains. We should exploit all parallelism levels but we should provide the programmer with an uniform abstract view of the architecture.

In addition, since the architecture class exploits a complex memory hierarchy exposing a NUMA behavior, both spatial and temporal locality assume a key role. Despite of research efforts pushed in shared virtual memory technology, the performance gap between local and remote accesses is still pretty large both in latency and bandwidth. In this setting, even a gradual escape from a "local working set" towards a remote one leads to an abrupt slowdown in application performance. We expect the problem to exacerbate with ever increasing speed of computer boxes. In the large, we expect the problem will affect not only the DSM layer of the virtual

memory but all levels of it.

Trivially, the most effective "tool" to exploit locality is the programmer skill. Indeed, the primary aim of eskimo is to enable the programmer to enforce locality by raising the level of abstraction of programming language without confusing his insight of the algorithm. At this end, we need a high-level parallel language that would not hide completely the underlying architecture but rather gently abstract it. In the same way the support would provide the programmer with appropriate hooks in order to control its behavior. Let us take spatial locality as an example. Normally software DSMs cannot control the shared memory at the granularity of the machine word. Almost all of them[3] group data in blocks (typically in pages) in order to reach an acceptable working grain for the architecture. We should be sure that such process does not destroy space locality, on the contrary we would turn data blocking into group-locality exploitation.

Moreover, dynamic data structures are dynamically allocated and managed. In the general case the programmer is not able to bound neither the size nor the shape of data structure independently of input data. We should provide the programmer with dynamic allocation primitives for its dynamic data structures. We should consider the peculiarities of dynamic data structures. Let us take trees as an example. Trees are rarely accessed in random way (as we can expect for arrays). Trees are often used to describe a hierarchically organized data set. In many cases the programmer will follow the tree structure, from the root down to leaves and vice-versa. We can spread tree nodes among processing elements, let us say, using a hash function, but is this a good organization for a tree? We believe we should respect the nature of tree. More ambitiously, we would like the programmer to express his insight of the algorithm by co-allocating data exposing a good temporal/spatial locality.

eskimo provides the programmer with some ready-made global data types (trees, arrays, regions) and a method to build his global data structures. In both cases, the programmer can express spatial locality by means of language mechanisms. In addition, eskimo run-time implements a data-driven scheduling, thus it enhances application temporal locality.

1.4 Plan of the thesis

The thesis deals with two facets of parallel programming that at first glance seem pretty distant one each other: Structured programming (in particular in skeleton-based languages) and shared address programming (in particular in DSMs).

Our research started from the "classical" approach to skeleton-based languages. In this setting, a skeleton is seen basically as an higher-order function. The executable code is sorted out by compiling the high-level language into a lower-level language (e.g. C + MPI), thus it can be considered a static approach. Therefore, we

[3]Except object-based ones.

attacked the code optimization problem by means of (performance-driven) successive refinement of the initial specification [14, 15, 8, 9]. Later, we moved to a more dynamical run-time support for the same kind of skeletons in order to defeat problems related to the availability of faithful cost models for skeleton nesting [15, 17, 16]. Recently, we moved again toward a different interpretation of the skeleton concept in high-level parallel programming. The main motivation resides in the lack of expressivity of the previous approaches, especially in case of irregular/dynamic applications. The new approach relies on the shared address programming paradigm (onto DSMs). In particular we try to explore the pay-back of using a structured approach in the programming of irregular/dynamic problem using (distributed) dynamic data structures. The thesis is organized as follows:

Chapter 2: Structured parallel programming. In this chapter we take in account structured parallel programming, and in particular *skeletal* parallelism. After a general introduction, the topic is developed mainly along Pisa University group's goals in the research area, others groups' goals are reported as well as references to the literature. Section 2.1 presents a self-critical history of research activity along last decade; the section would provide the needed insights to cope with the rest of the chapter. Sections 2.2 and 2.3 present (in details) the Meta optimization tool and the Lithium parallel programming environment. Eventually, Section 2.4 presents ASSIST, i.e. the youngest programming environment designed by the group. The last three sections are quite self-contained and might be read in any order.

The author participates in the design of all programming environments presented (except P^3L). Meta [8, 9] has been designed and developed by the author himself and used as optimization tool of the FAN language (not presented here [18]). Lithium formal semantics (Section 2.3) has been designed by the author himself and prof. M. Danelutto (later on, it has been proposed as general technique to describe both functional and parallel semantics of skeleton-based languages [16]), the Lithium framework has been mainly developed by P. Teti during his Master's thesis [17, 161, 62, 15]. ASSIST [169, 12, 13] has been developed at computer architecture lab. of Pisa University leaded by Prof. M. Vanneschi. A further evolution of ASSIST is already underway [11]. Since ASSIST in not the main topic of this thesis we just sketch here the current status only, referring forward to Chapter 8 for a brief discussion about future work.

Chapter 3: DSM: the state of the art. In this chapter we present a survey of distributed shared memory architectures, that are part of the framework of the thesis. The core of the discussion is reached through a brief review of DSM basic concepts, namely cache coherence and memory consistency. In Section 3.1.3 DSMs are characterized accordingly several functional aspects: implementation level, consistency model, and behavior with respect data replication. Those aspect are discussed in Sections 3.2, 3.3 and 3.4 respectively. *Cilk* main features are also sketched within

consistency models section. In Section 3.5 some technical issues related to software implemented DSMs are presented. Eventually, the *Athapascan* language is presented in Section 3.6.

Chapter 4: eskimo: design principles In this chapter we introduce "eskimo" language, i.e. a *skeletal* extension of C language for parallel programming based on the shared address model. We introduce the topic by briefly analyzing the lacks of previous skeletal programming frameworks (in particular our group's ones), thus motivating (yet) another evolution of skeletal frameworks. In Section 4.1 we present eskimo basic design principles. These are developed along parallelism exploitation (Sections 4.1.1 and 4.1.2), and memory sharing (Sections 4.1.3 and 4.1.4). The expected pay-back of the skeletal approach is discussed in Section 4.2. Eventually, we conclude sketching the differences between eskimo and some related works (*Cilk* and *Athapascan*).

eskimo C language extension has been fully designed by the author himself. Part of this chapter will appear in [10].

Chapter 5: eskimo: language usage. In this chapter we take again eskimo concepts intuitively presented in Chapter 4. In particular, we describe and exemplify eskimo primitives syntax and pragmatics. In Section 5.1 we present the eskimo computational model. Then in Section 5.2 we present eskimo details: How to write a eskimo program; eskimo primitives dealing with data type abstraction; eskimo primitives dealing with parallelism exploitation. eskimo pragmatics is developed in Section 5.3 by means of a running example. eskimo is pretty young programming platform and it is explicitly targeted to dynamic data structure experimentation in a skeletal framework. Therefore it is subjected to continuous modifications and improvements. We present here the current assessed status only, avoiding to mention possible improvements (that are already underway).

eskimo languages have been designed and developed by the author himself. Part of this chapter will appear in [10].

Chapter 6: eskimo: implementation. In this chapter we present eskimo run-time support design principles. Section 6.1 describes how eskimo abstracts the parallel architecture (consistency model, cache). Section 6.2 briefly introduces the implementation of Shared Data Types. Some experimental results are presented in order to support design choices.

Chapter 7: eskimo: experiments. This chapter reports experimental result obtained by using eskimo on a small test-suite. The test-suite includes some micro-benchmarks in order to test efficiency of some significant features of the run-time support, as for example address translation overhead. Test-suite also includes a

significant application (a n-body simulation) in order to experiment both expressiveness and performance of the language on a dynamic application.

Chapter 8: Discussion and concluding remarks. The chapter summarizes the materials contained in the previous chapters and discusses the conclusions of the thesis. The extent to which the goals of the thesis have been met is discussed. Finally the future work related to the thesis is introduced.

Chapter 2

Structured parallel programming

Readers' road-map. In this chapter we take in account structured parallel programming, and in particular *skeletal* parallelism. After a general introduction, the topic is developed mainly along Pisa University group's goals in the research area, others groups' goals are reported as well as references to the literature. Section 2.1 presents a self-critical history of research activity along last decade; the section would provide the needed insights to cope with the rest of the chapter. Sections 2.2 and 2.3 present (in details) the Meta optimization tool and the Lithium parallel programming environment. Eventually, Section 2.4 presents ASSIST, i.e. the youngest programming environment designed by the group. The last three sections are quite self-contained and might be read in any order.
The author participates in the design of all programming environments presented (except P^3L). Meta [8, 9] has been designed and developed by the author himself and used as optimization tool of the FAN language (not presented here [18]). Lithium formal semantics (Section 2.3) has been designed by the author himself and prof. M. Danelutto (later on, it has been proposed as general technique to describe both functional and parallel semantics of skeleton-based languages [16]), the Lithium framework has been mainly developed by P. Teti during his Master's thesis [17, 161, 62, 15]. ASSIST [169, 12, 13] has been developed at computer architecture lab. of Pisa University leaded by Prof. M. Vanneschi. A further evolution of ASSIST is already underway [11]. Since ASSIST in not the main topic of this thesis we just sketch here the current status only, referring forward to Chapter 8 for a brief discussion about future work.

Researchers have been working for a long time to bring parallel hardware and software into widespread use. Recently there has been progress on the hardware front. Serial microprocessors have been used as cost-effective building blocks for medium and large scale parallel machines. Now many high-volume serial processors contain hooks, such as snoopy buses, for implementing multiprocessor systems. These hooks make it quite simple and cheap for commercial computer manufacturers to build inexpensive, entry-level, multiprocessor machines. This trend towards including multiprocessor support in standard microprocessors occurred first with processors used in workstations (e.g. Sparc, PowerPC 601) and more recently with processors for PCs (e.g. Intel's PentiumPro). As with any other commodity, as

parallel machines drop in price, they become cost-effective in new areas, leading to parallel machines being installed at more and more sites. If this trend wasn't enough, high-speed networks and lower-overhead software are threatening to turn every LAN into a potential parallel machine. These machines (namely Beowulf class clusters or simply clusters) are gaining more and more interest as low cost parallel architectures, and actually more positions in the Top500 [164] parallel architecture list. Currently ASCI clusters and SMP clusters occupy 112 positions of the list. Although these parallel machines cannot be classified as low-cost Beowulf clusters – mainly due to the adoption of high-end interconnection networks and to the number of nodes used – the architectural trend is clear [70]. We may finally be witnessing the move of parallel machines into the mainstream.

Although building parallel computers has become easier, programming parallel computers can still be quite difficult. Besides coding all the algorithm details, the programmer must also take care of the details involved in parallelism exploitation, among the others concurrent activity set up (either processes or threads), mapping and scheduling, communication/synchronization handling and data allocation. In unstructured, low level parallel programming approaches these activities are usually fully in charge of the programmer and constitute a difficult, error prone programming effort. The effort required to the programmer varies from moderate to high, depending on the programming language/environment chosen to develop parallel applications.

Structured parallel programming systems allow a parallel application to be constructed by composing a set of basic parallel patterns called *algorithmical skeletons*.

Skeletons have been originally conceived by Cole [59] and then used by different research groups to design high performance structured parallel programming environments [26, 29, 154, 153]. A skeleton (in its original formulation) is formally an higher order function taking one or more other skeletons or portions of sequential code as parameters, and modeling a parallel computation out of them.

Cole's skeletons represent parallelism exploitation patterns that can be used (instanced) to model common parallel applications. Later, different authors acknowledge that skeletons can be used as constructs of an explicitly parallel programming language, actually as the only way to express parallel computations in these languages [76, 27]. Recently, the skeleton concept evolved, and became the coordination layer of structured parallel programming environments (see Section 2.1 [169, 29, 26, 147]). In any case, we can consider a skeleton as *an abstraction modeling a common, reusable parallelism exploitation pattern.*

Skeletons can be provided to the programmer either as language constructs [27, 26, 29] or as libraries [17, 72, 75, 120]. Usually, the set of skeletons includes both data-parallel and task parallel patterns.

Owner computes rule: the processor that owns the left-hand side element of an expression, will perform the calculation. For example, in the HPF program

```
DO i = 1,n
  a(i-1) = b(i*7)/c(i+j)-a(i**i)
END DO
```

the processor that owns `a(i-1)` will perform the assignment. The components of the *rhs* expression may have to be communicated to this processor before the assignment is made. As this is a rule of thumb it is not always followed; for example, if all the *rhs* objects are co-distributed then, instead of all the *rhs* elements being sent to the owner of the *lhs* for computation, the computation of the result may take place on the home processor of the *rhs* elements and then be sent to the owner of the *lhs* for assignment. This would reduce the number of communications required [104].

Table 2.1: Concept recap: Owner computes rule.

Data parallelism

One of the most successful paradigm for parallelism exploitation is the data-parallel programming paradigm [103]. This paradigm is useful for taking advantage of the large amounts of data parallelism that is available in many scientific/numeric applications. The data parallelism is exploited by performing the same operation on a large amount of data, distributed across the processors of the machine. From the programmer viewpoint languages based on data-parallel paradigm (such as HPF [102] and CM Fortran [162]) are pretty similar to sequential languages. The main difference is that certain data types are defined to be parallel. Parallel data values consist of a collection of standard, scalar data values. These languages contain predefined operations on parallel variables that either operate on the parallel variable element-wise (e.g. multiplying every element by a scalar value), or operate on the parallel value as a whole (e.g. summing all elements of the parallel variable).

The data-parallel paradigm has three main virtues that have led to its success. The first virtue of this model is that data-parallel codes are fairly easy to write and debug. Just as in a serial program, the programmer sees a sequential flow of control The values making up a parallel value are automatically spread across the machine, although typically the programmer does have the option of influencing how data is placed. Parallel data types are typically static in size (e.g. arrays), their distribution across the machine is usually done at compile time. Any synchronization or communication that is needed to perform an operation on a parallel value is automatically added by the compiler/run-time system. Operations on parallel data values are collectively computed by the processors; computation load usually distributed directly linking (left) data values and computations through the *owner computes rule* (see Table 2.1). As data values, computation load is statically distributed across the processors of the system.

The second virtue of this model is that it is easy for a programmer to understand the performance of a program. Given the size of a parallel value to be operated on, the execution time for an operation is likely to be predictable. Since the execution of each operation is independent of the others, and there are is overhead due to the dynamic management of data values and computations, the execution time for the program as a whole is predictable as well. Faithful performance models can therefore been developed for this kind of languages.

The third major advantage of data parallelism derives from its scalability. Because operations may be applied identically to many data items in parallel, the amount of parallelism is dictated by the problem size. Higher amounts of parallelism may be exploited by simply solving larger problems with greater amounts of computation. Data parallelism is also simple and easy to exploit. Because data parallelism is highly uniform, it can usually be automatically detected by an advanced compiler, without forcing the user to manage explicitly processes, communication, or synchronization.

Many scientific applications may be naturally specified in a data-parallel manner. In this settings, program's data layout is often fixed; the most used data structures are large arrays. Operations on whole data structures, such as adding two arrays or taking the inner product of two vectors, are common, as are grid-based methods for solving partial differential equations (PDEs). Since data-parallel programs are relatively close to sequential programs, many compiler analysis and optimization techniques can be adapted to produce parallel programs automatically. The mapping of data and computation can affect performance significantly. With data-parallel programs, relatively simple data decomposition annotations are sufficient to achieve high performance on advanced parallel architectures. If communication and parallelism are implicit (as in HPF), the user may tune the program by small modifications to its data decomposition annotations.

In spite of this, data parallelism has a significant drawbacks: the limited range of applications for which data-parallel is well suited. Applications with data parallelism tend to be static in nature, the control flow of a data-parallel program is mostly data independent. Many applications are more dynamic in nature and do not have these characteristics. To run in parallel, these dynamic applications need to exploit control parallelism by performing independent operations at the same time. These applications, which may be as simple as recursively computing Fibonacci numbers or as complex as computer chess and n-body simulations, are nearly impossible to express in data-parallel languages. In addition, quite often recursive application needs dynamic data structures. Such data structures (e.g. trees and linked lists) cannot be always embedded in static arrays, and even when possible such embedding burdens the programmer with additional tasks such for example the even mapping of a dynamic structure in a static one. At this end, recent evolutions of data-parallel based languages (such as HPF version 2) provide to the programmer additional directives like DYNAMIC, REALIGN and REDISTRIBUTE [102].

Task Parallelism

In a task-parallel programming paradigm the program consists of a set of (potentially dissimilar) parallel tasks that interact through explicit communication and synchronization. Task parallelism may be both synchronous and asynchronous.

A major advantage of task parallelism is its flexibility. Because of its emphasis on explicit coordination of individual tasks (or processes, as they are often called), task parallelism can be used to exploit both structured and unstructured forms of parallelism. Many scientific applications contain task parallelism. For example, in a climate model application the atmospheric and ocean circulation may be computed in parallel as two separate tasks in a pipeline fashion. A task-parallel language can express this relationship easily, even if different methods are used for the two circulation models. Another natural application of task-parallel languages is reactive systems in which tasks must produce output in response to changing inputs, in a time-dependent manner. Tasks may also be organized as a pipeline to exploit pipeline parallelism [54, 55, 53].

Another common structured paradigm exploits parallelism on different data items through task/function replication. For example, the elaboration of a video stream may involve the filtering on each single frame. In a task-parallel language the filter may be *farm*ed out by spreading different frames on different worker processes, each of them computing the same function [28].

In unstructured task parallelism interactions between tasks are explicit, thus the programmer can write programs that exploit parallelism not detectable automatically by compiler techniques. The programmer may also carefully tune the application so that it includes only the communication and synchronization that is actually necessary or efficient, hence reducing reliance on compiler optimization. In general, unstructured task parallelism is less dependent on advanced compiler technology than is data parallelism; in many cases, all that is strictly necessary is the translation of task interactions into appropriate low-level primitives on the target architecture. However, compiler technology is still important as a means of guaranteeing correct execution and permitting representations of communication and synchronization that are convenient for the programmer.

A disadvantage of the unstructured task-parallel programming model is that it requires extra effort from the programmer to create explicit parallel tasks and manage their communication and synchronization. It is also often convenient to consider data owned by different tasks as being part of a single data structure; many task-parallel languages do not support this view directly. Because communication and synchronization are explicit, changing the manner a program is parallelized may require extensive modifications to the program text.

Structured version of task parallelism relieves the programmer effort by providing a set of predefined templates for processes relationship. Such templates model common organization schemes for processes and their communications/synchronizations.

2.1 Our skeletons (in the closet)

For more than a decade our research group has been active in experimenting new technologies in order to "simplify" parallel programming. These are mainly targeted to simplify programming by raising the level of abstraction; to enhance portability by absolving the programmer of responsibility for detailed realization of the underlying parallel paradigms; to improve performance by providing access to carefully optimized implementations of the paradigms. These technologies have been used to design programming environments and languages and implement their compilers. Skeletons have been present all along in programming environments, even if their role has been permanently changing (and maturing).

The P^{3}L (Pisa Parallel Programming Language, 1990) was the seminal project on structured parallel programming of the group. It was (initially) designed in collaboration with Hewlett Packard Laboratories. The language core includes programming paradigms like pipelines, task farms, iterative and data parallel skeletons. Skeletons in P^{3}L can be used as constructs of an explicitly parallel programming language, actually *as the only way to express parallel computations* [74, 139, 73, 140]. Several evolutions of the P^{3}L (shareware) compilers has been developed using C and OCaml as "host" languages [56, 72]. These compilers are based on the concept of *implementation template*, i.e. a parametric process network [27].

Later on all experiences assessed with P^{3}L have met into the SkIE language and its compiler. The SkIE (Skeleton Interface Environment, 1998) was designed and developed in collaboration with QSW ltd. [167] It improves P^{3}L features in many different ways [29, 166, 168], among others:

- enforces P^{3}L reuse feature. Existing sequential codes can be used to instance skeletons with with little or no amendment to the sources;
- it supports several guest sequential and parallel languages languages (C, C++, Fortran, Java, HPF) within the same application;
- a brand new design of implementation templates provides SkIE with a satisfactory absolute performance and performance portability for homogeneous platforms (Ethernet-connected clusters and QSW proprietary platforms running Linux/Solaris) [7].

Also, the skeleton concept has been enforced in SkIE : skeletons have been still the only way to express parallel computations, but they have been equipped with a compositional functional semantics, they become actually higher-order functions which can be evaluated efficiently in parallel. A SkIE program is basically a composition of skeletons.

Furthermore, due to the fact that the skeletons have a clear functional and parallel semantics, different rewriting techniques have been developed that allow skeleton programs to be transformed/rewritten into equivalent ones achieving different performances when implemented on the target architecture [14, 38, 94]. These

transformations can also be driven by some kind of analytical performance models, associated with the implementation templates of the skeletons, in such a way that only those rewritings leading to efficient implementations of the skeleton code are considered [14, 15].

The research community has been proposing several development frameworks based on the refinement of skeletons [18, 76, 158]. In such frameworks, the user starts by writing an initial skeletal program/specification. Afterwards, the initial specification may be subjected to a cost-driven transformation process with the aim of improving the performance of the parallel program. Such a transformation is done by means of semantic-preserving rewriting rules. A rich set of rewriting rules [14, 18, 95, 94] and cost models [18, 158, 172] for various skeletons have been developed recently. Conceptually, skeleton-based programs and semantic-preserving rewriting rules may be thought as terms and term-rewriting rules within a Term Rewriting System respectively. In this setting, the recursive application of all rules to a given term yields the set of all terms reachable through rules, i.e. the set of all (reachable) semantic-equivalent alternative programs to a given one. In particular for P^{3}L/SkIE programs (equipped with a standard set of rules) we define a program *normal form* (see also Chapter 4) representing the provably optimum[1] among all alternative programs reachable from the initial specification by means of rewriting rules in the standard set. Moreover, *normal form* can be algorithmically constructed from any given P^{3}L/SkIE program [15, 17].

Moreover, I designed and developed Meta, an interactive transformation tool for skeleton-based programs. The tool basically implements a term rewriting system that may be instantiated with a broad class of skeleton-based languages and skeleton rewriting rules. Given a language and a set of semantic-preserving transformations, the Meta tool assists the user in the transformation process. The transformation process may be also *performance driven*, provided each rewriting rule is equipped with a performance formula, which depends on the particular skeleton implementations for the target language. Alternatively, the tool may follow a global optimization strategy, as for example *normal form* reduction for SkIE programs. Overall Meta completes our path towards (structured) parallel programming by refinement. Meta is extensively described in Section 2.2 [8, 9].

Several real world, complex applications has been prototyped using SkIE , especially in the parallel data-mining area [35, 24, 63]. During the development of such applications, the skeletal approach has been proved to be effective, at least if application algorithms can be somehow expressed in terms of skeleton composition. Actually a lack of expressivity emerged, at least for complex applications. Cole effectively (but lately) expresses the problem as follows [60]:

> *Many parallel applications are not obviously expressible as instances of skeletons, whether existing or imagined. Some have phases which require*

[1] Both in terms of absolute performance and efficiency and provided some mild additional requirements.

> *the use of less structured interaction primitives. Some have conceptually layered parallelism, in which skeletal behavior at one layer controls the invocation of operations involving such ad-hoc parallelism within. It is clearly unrealistic to assume that skeletons can provide all the parallelism we need. We must construct our systems to allow the integration of skeletal and ad-hoc parallelism in a well defined way ...*
> *Skeletal programming is not functional programming, even though it may be concisely explained and expressed as such. Skeletal programming is not object oriented programming, even though this may be a similarly attractive vehicle. Instead, we should build bridges to the standards of the day, refining or constraining only where strictly necessary. We should respect the conceptual models of these standards, offering skeletons as enhancements rather than as competition. This need not be too difficult. For example, it is arguable that MPI already embodies simple skeletons in its collective operations.*

Eventually, another evolution of skeleton concept has been underway, both in our group and other research groups. As our group concern, we have taken a step back in respect of skeletons' role in parallel programming. In particular skeleton loose their "exclusiveness" on parallelism exploitation. The new skeletons' role has led to the exploration of several scenarios:

Skeletons as design patterns. A design pattern *per se* is not a programming construct, as happened for the skeletons. Rather, it can be viewed as "recipe" that can be used to achieve different solutions to common programming problems. The skeleton set may include (for example) usual pipeline, farm, data parallel and divide&conquer skeletons, plus a generic "parallel module" skeleton aimed at abstracting common, non-trivial data parallel patterns . The parallel skeleton support may be implemented using a layered, OO design [131]. Parallel skeletons can be declared as members of proper skeletons/patterns. Object code generation can be requested, leading to the automatic generation of a set of implementation classes, globally making up an implementation *framework*. Exploiting standard OO visibility mechanisms, part of the implementation framework classes may be made visible to the programmer in such a way he can perform different tasks: fine performance tuning (sub-classing existing implementation classes methods), introduction of new, more efficient implementation schemes (sub-classing existing implementation classes) or either introduction of new skeletons/patterns (introducing new skeleton classes, possibly sub-classes of existing ones, and using either existing or new implementation classes) [71, 144].

The new approach promises more flexibility in application programming by making skeletons objects that can be refined, both in the semantics and the implementation.

Skeletons as extension. Skeletons may be used to extend existing programming languages or programming frameworks (e.g. C + MPI) that are already able to exploit parallelism. Several recent programming frameworks may be numbered among this category, among the others:

`SKElib` extends C language with SkIE-like skeletons. Programs are written in SMPD style and skeletons behave as collective operations; the library allows the programmer to structure parallel computations whose patterns do not correspond to a skeleton by using standard Unix mechanisms [75].

Lithium is a *pure Java* structured parallel programming environment based on skeletons. Lithium is implemented as a Java package and represents both the first skeleton based programming environment in Java and the first complete skeleton based Java environment exploiting macro data flow implementation techniques [70]. Lithium supports a set of user code optimizations which are based on skeleton rewriting techniques. These optimizations improve both absolute performance and resource usage with respect to original user code. Parallel programs developed using the library run on any network of workstations provided the workstations support plain JRE. Lithium is extensively described in Section 2.3 [17].

`eSkel` is a library which adds skeletal programming features to the C/MPI parallel programming framework. It is a library of C functions and type definitions which extend the standard C binding to MPI with skeletal operations. Its underlying conceptual model is that of SPMD distributed memory parallelism, inherited from MPI, and its operations must be invoked from within a program which has already initialized an MPI environment. It is most readily understood as an extension to MPI's set of collective operations [60, 58].

Kuchen's skeleton library extends C++ language providing the programmer with task and data parallel skeletons, which can be combined on the *two-tier model* (see Section 2.2.1) taken from P^3L. Programs are written in SMPD style and skeletons behave as collective operations. Data parallelism is based on distributed data structures (arrays, actually) that can be manipulated by collective operations (like map and fold). Task parallelism is exploited through P^3L-like pipeline and farm skeletons [120, 45].

Skeletons as "good programming discipline". The approach consists in providing the programmer with "proto-skeletons" or constructs. Constructs extend an host language and represent skeletons' building blocks. Therefore, skeletons does not really exist in the program as language elements, rather they are particular programming idioms. In sequential programming setting we can recognize many of them: divide&conquer, exhaustive search in an array or in a linked data structure, etc. Actually, the approach have strong analogies with both previous approaches. In

one hand, the approach may be considered as a "low-level" extension of a language or better an extension of the programming model of a language. In the other hand, both OO and design patterns may be probably considered "an engineered" version of the approach.

Nevertheless, we would like maintain the approaches distinct (for a while). The idea beneath first two approaches is that general or important patterns have been already recognized. The programmer may refine them, but he cannot radically change their behavior. Constructs rely on another idea, these allows the programmer to build his own set of skeletons. We put eskimo language in this category (see Chapter 5).

Indeed, eskimo language is thought as programming layers of more abstract and complete environment designed by our group, i.e. the ASSIST environment. The ASSIST (A Software development System based upon Integrated Skeleton Technology, 2002) has been designed and developed at University of Pisa with the support of the *Italian Space Agency* and *National Research Council*[2]. [169, 66] We shall briefly present the ASSIST environment is Section 2.4, we anticipate here that ASSIST adopts two new concepts with respect to our group's previous environments:

- A new paradigm, called "parallel module" (`parmod`), is defined which, in addition to expressing the semantics of several skeletons as particular cases, is able to express more general parallel and distributed program structures, including both data-flow and nondeterministic reactive computations.
- `parmod`s are able to utilize *external objects*, including shared data structures and abstract objects, with standard interfacing mechanisms (e.g. IDL). The introduction of shared data structures aims to efficiently manipulate very large data sets, to simplify the programming of irregular and/or dynamic problems.

The eskimo language is designed to experiment how such shared objects may be defined and used in a high-level structured programming environment.

[2]With "ASI-PQE2000 project" and "Agenzia 2000 CNR project" respectively.

2.2 The Meta optimization tool

In this section we present Meta, an interactive transformation tool for skeleton-based programs. The tool basically implements a term rewriting system that may be instantiated with a broad class of skeleton-based languages and skeleton rewriting rules.

Given a skeleton-based language and a set of semantic-preserving transformation rules, the tool locates applicable transformations and provides performance estimates, thereby helping the programmer in navigating through the program refinement space. Meta is based on a novel program representation (called *dependence tree*) that allows to effectively implement a rewriting system via pattern-matching.

The Meta tool can be used as a building block in general transformational refinement environments for skeleton languages. Meta has already been used as transformation engine of the FAN skeleton framework [18, 95], that is a pure data parallel skeleton framework. Actually, Meta is more general and may be also used in a broad class of mixed task/data parallel skeleton languages [29, 147, 172].

The remaining of the section is organized as follows: Section 2.2.1 frames the kind of languages and transformations Meta can deal with. The Skel-BSP language, used as a test-bed for Meta, is presented. Section 2.2.2 describes the Meta transformation tool and its architecture. Then, Section 2.2.4 discusses a case study and the cost models for Skel-BSP, presenting some experimental results. An extended version of the results appearing in this section can be found in [8, 9, 18].

2.2.1 Skeletons and transformations

We consider a generic structured coordination language TL (for *target language*) where parallel programs are constructed by composing procedures in a conventional base language using a set of high-level pre-defined skeletons. We also assume that the skeletons set has three kinds of skeletons: *data parallel*, *task parallel* and *sequential* skeletons. Sequential skeletons encapsulate functions written in any sequential base language and are not considered for parallel execution. The others provide typical task and data parallel patterns. Finally, we constrain data parallel skeletons to call only sequential skeletons. This is usually the case in real applications and it is satisfied by the existing skeleton languages [76, 26, 29, 18, 172, 147]. Applications written in this way have the (up to) three-tier structure sketched out in Figure 2.1.

In order to preserve generality, Meta can be specialized with the TL syntax and its three skeleton sets. The only requirement we demand is that the above constraints on skeleton calls holds. This makes our work applicable to a variety of existing languages.

Besides a skeleton-based TL, the other ingredient of program refinement by transformation is a set of semantic-preserving rewriting rules. A rule for TL is a pair $L \rightarrow R$, where L and R are fragments of TL programs with variables $\nu_0, \nu_1 \ldots$ ranging over TL types, acting as place-holder for any piece of program. We require that

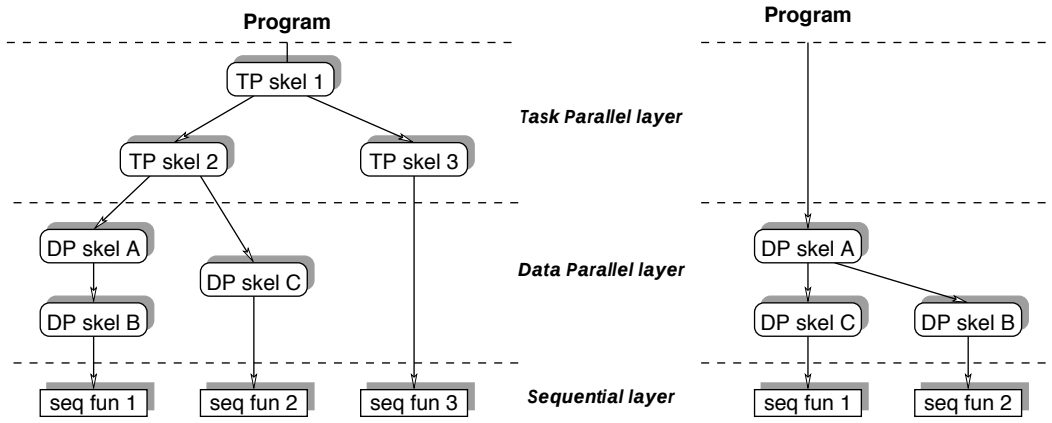


Figure 2.1: Three-tier applications: two correct skeleton calling schemes.

every variable occurring in R must occur also in L and that L is not a variable. Moreover, a variable may be constrained to assume a specified type or satisfy a specific property (e.g., we may require an operator to distribute over another operator). The left-hand side L of a rule is called a *pattern*.

In the rest of the section, we consider a simple concrete target language as a test-bed for the Meta transformation tool: Skel-BSP[172]. Skel-BSP has been defined as a subset of P^{3}L [27] on top of BSP (Bulk-Synchronous Parallel model [165]) and it can express both data and task parallelism. The following defines a simplified Skel-BSP syntax which is particularly suitable for expressing rules and programs in a compact way:

TL_prog ::= *TP* | *DP*
TP ::= farm "(" *TP* ")" | pipe "{" *TPlist* "}" | *DP*
TPlist ::= *TP* | *TP*, *TPlist*
DP ::= map *Seq* | map2 *Seq* | scanL *Seq* | reduce *Seq* | *Seq* |
comp "(" out *Var*, in *Varlist* ")" "{" *DPlist* "}"
DPlist ::= *Var* "=" *DP Varlist* | *Var* "=" *DP Varlist*, *DPlist*
Var ::= ⟨ *a string* ⟩
Varlist ::= *Var* | *Var*, *Varlist*
Seq ::= ⟨ *a sequential C function* ⟩

TL_prog can be formed with skeleton applications, constants, variables or function applications. Each skeleton instance may be further specified by its name just adding a dotted string after the keywords (e.g. comp.*mss*). Variables are specified by a name and by a type ranging over (all or some of) the base language types (e.g. all C types except pointers). The type of variables may be suppressed where no confusion can arise.

The pipe skeleton denotes functional composition where each function (stage) is executed in pipeline on a stream of data. Each stage of the pipe runs on different (sets of) processors. The farm skeleton denotes "stateless" functional replication on a stream of data. The map, scanL and reduce skeletons denote the namesake data parallel functions [38] and do not need any further comment. map2 is an extended version of map, which works on two arrays (of the same lengths) as follows: map2 f $[x_0, \ldots, x_n]$ $[y_0, \ldots, y_n] = [f\ x_0\ y_0, \ldots, f\ x_n\ y_n]$. The comp skeleton expresses the sequential composition of data parallel skeletons. The body of the comp skeleton is a sequence of equations defining variables via expressions. Such definitions follows the *single-assignment* rule: there is at most one equation defining each variable.

comp.*name* (out *outvar*, in *invars*){
  $outvar_1$ = dp.*1* Op_1 $invars_1$
    ⋮
  $outvar_n$ = dp.*n* Op_n $invars_n$}

where: $\forall k = 1..n, invars_k \subseteq \left(\bigcup_{i<k} outvar_i \cup invars\right), outvar \in \bigcup_{i \leq n} outvar_i$

The skeletons within the comp are executed in sequence on a single set of processors in a lock-step fashion, possibly with a (all-to-all) data re-distribution among steps. The cost estimate of Skel-BSP is based on the Valiant's Bulk-Synchronous Parallel model [165, 172]. The cost model for Skel-BSP is discussed in Section 2.2.4 along with some results on its accuracy. Results show that close estimate are possible on a fairly common parallel platform like a cluster of Pentium PCs.

Examples

In this section, we consider a pair of simple Skel-BSP programs: the maximum segment sum and the polynomial evaluation. Both programs are the Skel-BSP presentation of parallel algorithms appeared in [18, 95].

Maximum segment sum. Given a one-dimensional array of integers v, the maximum segment sum (MSS) is a contiguous array segment whose members have the largest sum among all segments in v. Suppose we would like to compute the MSS of a stream of arrays. The following code is a first parallel program for computing MSS following a simple strategy [18, 95]:

```
pipe.mss {
  map pair,          /* :  int [n]    →  int [n][2]  */
  scanL Op+,         /* :  int [n][2] →  int [n][2]  */
  map P1,            /* :  int [n][2] →  int [n]     */
  reduce max}        /* :  int [n]    →  int         */
```

The comments on the right hand side state the type of each skeleton instance; types are expressed using a C-like notation. The operator Op_+ is defined as follows:

$$[x_{i,1}, x_{i,2}]Op_+[x_{j,1}, x_{j,2}] = [max\{x_{i,1} + x_{j,2}, x_{j,1}\}, x_{i,2} + x_{j,2}]$$

while $pair\ x = [x, x]$ and $P_1\,[x_1, x_2] = x_1$. Intuitively, the purpose of scanL is to produce an array s whose ith element is the maximum sum of the segments of x ending at position i. Using a sequential program, this task can be accomplished simply by using scanL with operator $Op_1(a, b) = max(a + b, b)$. Unfortunately, such operator is not associative, thus this simple scanL cannot be parallelized. Op_+ uses an auxiliary variable to preserve the associativity. This variable is thrown away at the end of the scanL computation by the P_1 operator. Finally, reduce sorts out the maximum element of array s yielding to the desired maximum segment sum r.

Polynomial evaluation. Let us consider the problem of evaluating in parallel a polynomial $a_1x + a_2x^2 + \ldots a_nx^n$ at m points $y_1, \ldots y_m$. The most intuitive solution consists in parallelizing each basic step of the straightforward evaluation algorithm, i.e. first compute the vector of powers $ys^i = [y_1^i, \ldots, y_m^i], i = 1 \ldots n$, then multiply by the coefficients, and, finally, sum up the intermediate results. The algorithm can be coded in Skel-BSP as follows.

comp.*pol_eval* (out zs, in $ys,\ as$) {
  $ts =$ scanL $*\ ys,$ /* $ts[i] = ys^i$: float [n][m] */
  $ds =$ map2 $(*_{sa})\ as,\ ts,$ /* $ds[i] = [a_i * y_1^i, \ldots, a_i * y_m^i]$: float [n][m] */
  $zs =$ reduce $+\ ds\}$ /* $zs[i] = [\sum_{i=1}^n a_i * y_1^i, \ldots, \sum_{i=1}^n a_i * y_m^i]$: float [m] */

where $*_{sa}$ multiplies each element of a vector by a scalar value, $*$ and $+$ are overloaded to work both on scalars and (element-wise) on vectors. On the right side (in comments) we describe the variable values and types.

Transformation rules

When we design a transformation system a foremost step is the choice of the rewriting rules to be included and the definition of their costs. The goal of the system is to derive a skeletal program with the best performance estimate by successive (semantic-preserving) transformations (rewrites). Each transformation/rewrite correspond to the application of a rewriting rule. Here, we only collect the transformations needed to demonstrate the use of Meta on an example. We refer back to the literature for the proofs of the soundness of the rules [14, 15, 18, 38, 94]. For the sake of brevity, we use $L \rightleftarrows R$ to denote the pair of rules $L \rightarrow R$ and $R \rightarrow L$.

In the following, TSk_i can be any skeleton (task or data parallel, sequential), DSk_i can be any data parallel or sequential skeleton. $Op_1, Op_2, \ldots$ denote variables ranging over sequential functions. *pair* and P_1 are sequential auxiliary functions defined in the previous section. The labeled elision $< \cdots >_n$ represents an unspecified chunk of code that appears (unchanged) in both sides of the rules.

farm insertion/elimination. These rules state that farms can be removed or introduced on top of a TSk skeleton [15]. The rule preserves the constraint on layers since TSk cannot appear into a data parallel skeleton. A farm replicates TSk without changing the function it computes. Thus, it just increases task parallelism among different copies during execution.

$$\mathsf{TSk} \quad \begin{matrix}\rightarrow \\ \leftarrow\end{matrix} \quad \mathsf{farm}\ (\mathsf{TSk})$$

pipe → comp. The pipe skeleton represents the functional composition for both task and data parallel skeletons. The comp models a (possibly) more complex interaction among data parallel skeletons. If all the stages DSk_1, $\mathsf{DSk}_2 \ldots$ of the pipe are data parallel (or sequential) skeletons, then the pipe can be rewritten as a comp in which each DSk_i gets its input from DSk_{i-1} and outputs towards DSk_{i+1} only. Also in this case the two formulations differ primarily in the parallel execution model. When arranged in a pipe, the DSk_1, $\mathsf{DSk}_2 \ldots$ are supposed to run on different sets of processors, while arranged in a comp, they are supposed to run (in sequence) on a single set of processors.

$$\begin{array}{l}\mathsf{pipe}\ \{ \\ \quad \mathsf{DSk}_1\ Op_1, \\ \quad \mathsf{DSk}_2\ Op_2, \\ \quad <\cdots>_1 \\ \quad \mathsf{DSk}_n\ Op_n\}\end{array} \quad \rightarrow \quad \begin{array}{l}\mathsf{comp}\ (\mathsf{out}\ z, \mathsf{in}\ a)\ \{ \\ \quad b = \mathsf{DSk}_1\ Op_1\ a, \\ \quad c = \mathsf{DSk}_2\ Op_2\ b, \\ \quad <\cdots>_1 \\ \quad z = \mathsf{DSk}_n\ Op_n\ y\}\end{array}$$

map fusion/fission. This rule denotes the map (backwards) distribution through functional composition [38]. Notice that when we apply form left-to-right we do not require the two maps in the left hand side to be adjacent in the program code. We just require that the input to the second one (q) is the output from the first one.

$$\begin{array}{l}\mathsf{comp}\ (\mathsf{out}\ outvar, \mathsf{in}\ invars)\ \{ \\ \quad <\cdots>_1 \\ \quad q = \mathsf{map}\ Op_1\ p, \\ \quad <\cdots>_2 \\ \quad r = \mathsf{map}\ Op_2\ q, \\ \quad <\cdots>_3\ \}\end{array} \quad \begin{matrix}\rightarrow \\ \leftarrow\end{matrix} \quad \begin{array}{l}\mathsf{comp}\ (\mathsf{out}\ outvar, \mathsf{in}\ invars)\ \{ \\ \quad <\cdots>_1 \\ \quad q = \mathsf{map}\ Op_1\ p, \\ \quad r = \mathsf{map}\ (Op_2 \circ Op_1)\ p, \\ \quad <\cdots>_2 \\ \quad <\cdots>_3\ \}\end{array}$$

It is important to notice that, while rules are required to be *locally* correct, Meta ensures the *global* correctness of programs. For instance, using the rule from left-to-right (map fusion) the assignment in the grey box is not required to appear. Meta provides the program with the additional assignment (in the grey box) only if the intermediate result q is referenced in some expressions into $<\cdots>_2$ or $<\cdots>_3$.

SAR-ARA (Scan Arrange Reduce – Arrange Reduce Arrange). This rule (applied from left-to-right) aims to reduce the number of communications using the very complex operator Op_3. In general, the left-hand side is more communication

intensive and less computation intensive than the right-hand side. The exact trade-off for an advantageous application heavily depends on the cost calculus chosen (see [18, 95]).

comp (out *outvar*, in *invars*){
$< \cdots >_1$
q = scanL Op_1 p,
r = map P_1 q,
s = reduce Op_2 r,
$< \cdots >_2$ }

$\rightarrow$
$\leftarrow$

comp (out *outvar*, in *invars*){
$< \cdots >_1$
t = map *pair* p,
u = reduce Op_3 t,
v = map P_1 u,
x = map P_1 v,
$< \cdots >_2$ }

Op_1 must distribute forward over Op_{aux}. Op_{aux} and Op_3 are defined as follows:

$$[x_{i,1}, x_{i,2}]Op_3[x_{j,1}, x_{j,2}] = [x_{i,1}Op_{\text{aux}}(x_{i,2}Op_1x_{j,1}), x_{i,2}Op_1x_{j,2}]$$

$$[x_{i,1}, x_{i,2}]Op_{\text{aux}}[x_{j,1}, x_{j,2}] = [x_{i,1}Op_2x_{j,1}, x_{i,2}Op_2x_{j,2}]$$

Notice that, whereas operator Op_2 works on single elements, operators Op_1 and Op_{aux} are defined for pairs (arrays of length 2), and Op_3 works on pairs of pairs.

2.2.2 The transformation tool

In this section, we describe a transformation tool which allows the user to write, evaluate and transform TL programs, preserving their functional semantics, and possibly improving their performance. The tool is interactive. Given an initial TL algorithm, it proposes a set of transformation rules along with their expected performance impact. The programmer chooses a rule to be applied and successively (after the application) the tool looks for new matches. This process is iterated until the programmer deems the resulting program satisfactory, or there are no more applicable rules.

The strategy of program transformation is in charge of the programmer since, in general, the rewriting calculus of TL is not confluent: applying the same rules in a different order may lead to programs with different performance. The best transformation sequence may require a (potentially exponential) exhaustive search.

In the following, we define an abstract representation of TL programs and transformation rules, we describe the algorithm used for rule matching, and finally we sketch the structure of the tool.

Representing programs and rules

The Meta transformation system is basically a term-rewriting system. Both TL programs and transformation rules are represented by means of a novel data structure, so-called *dependence tree*. Dependence trees are basically labeled trees, thus the search for applicable rules reduces to the well established theory of subtree matching [106]. The tool attempts to annotate as many nodes of the tree representation

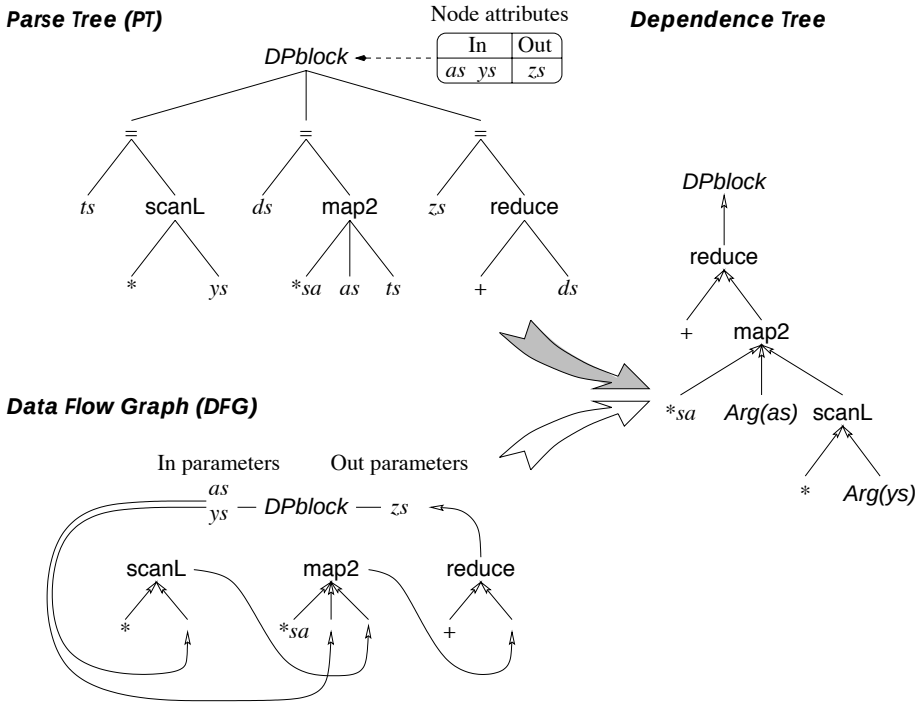


Figure 2.2: The parse tree, the data flow graph and the dependence tree of polynomial evaluation. Skel-BSP skeletons are in serif font. Special nodes are in *slanted serif* font. Sequential functions are in *italic* font.

Input: PT and DFG for a correct TL program. The starting node x is the root of PT. No nested *DPblock* are allowed (which can be easily flattened).
Output: The dependence tree DT.
Method:

1. Let x denote the current node, starting from the root of PT;
2. Copy x from PT on DT along with the arc joining it with its parent (if any), the arc is undirected as it comes from PT;
3. **if** not(x = *DPblock*)
4. **then** Recursively apply the algorithm to all sons of x in PT (in any order);
5. **else** Apply Procedure dpb(*DPblock*).

Procedure dpb($Node$):
a. From $Node$ follow backward the incoming edges in DFG;
b. **for each** node C_i reached in this way, **do**
c. Copy C_i from DFG to DT along with its out-coming edges;
d. Recursively apply dpb(C_i) until the starting node *DPblock* or a sink is reached; In the former case add a node *Arg* to represent the formal parameter name.

Table 2.2: Building up the dependence tree.

as possible with a *matching rule instance*, i.e., a structure describing which rule can be used to transform the subtree rooted at the node, together with the information describing how the rule has been instantiated, the performance improvement expected and the applicability conditions to be checked (e.g., the distributivity of one operator over another).

The dependence tree is essentially an abstract syntax tree in which each non-leaf node represents a skeleton, with sons representing the skeleton parameters that may in turn be skeletons or sequential functions. The leaves must be sequential functions, constants or the special node *Arg( )*. Unlike a parse tree, a dependence tree directly represents the data dependence among skeletons: if the skeleton Sk_1 directly uses data produced by another skeleton Sk_2, then they will appear as adjacent nodes in the dependence tree, irrespectively of their position in the parse tree. Each edge in the dependence tree represents the dependence of the head node from the data produced by the tail node. The dependence tree of a program is defined constructively, combining information held in the parse tree (PT) and in the data flow graph (DFG) of the program. The algorithm to build dependence trees is shown in table 2.2. The algorithm is illustrated in Figure 2.2, which shows the parse tree, the data flow graph and the correspondent dependence tree of the polynomial evaluation example (see Section 2.2.1). The nodes labeled with *DPblock* mark the minimum subtrees containing at least one data parallel skeleton, nodes *Arg(as)* and *Arg(ys)* represent the input data of a *DPblock*. In other words, *DPblock* nodes delimit the border between the task parallel and the data parallel layers.

It is important to understand why we need to introduce a new data structure instead of using the parse tree directly. The main reason lies in the nature of the class of languages we aim to deal with, i.e. mixed task/data parallel languages. Nested skeleton calls find a very natural representation as trees. On the contrary, data parallel blocks based on the single-assignment rule (e.g. Skel-BSP comp) need

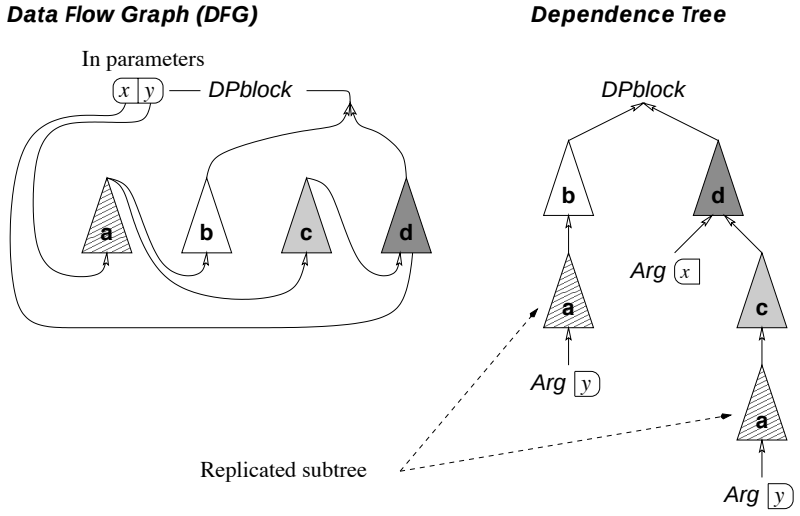


Figure 2.3: Replicating shared trees. Each triangle stands for a tree representing a TL expression.

a richer representation in order to catch the dependences among the skeletons (for example a data flow graph). The dependence tree enables us to compact all the information we need in a single tree, i.e. in a data structure on which we can do pattern-matching very efficiently.

There is one more point to address. The dependences shown in Figure 2.2 are rather simple. In general, as shown in Figure 2.3, a data structure produced by a single TL statement may be used by more than one statement in the rest of the program. We have two choices: (1) to keep a shared reference to the expression (tree), or (2) to replicate it. In option (1), the data flow can no longer be fully described by a tree. In addition, sharing the subtrees rules out the possibility of applying different transformations at the shared expression (tree) for different contexts. The Meta transformational engine adopts the second option, allowing us to map the data flow graph into a tree-shaped dependence structure. The drawback of replicating expressions is a possible explosion of the code size when we rebuild a TL program from the internal representation. To avoid this, the engine keeps track of all the replications made. This ensures a single copy of all replicated subtrees that have not been subject to an independent transformation.

Figure 2.4 depicts the internal representation of rule map fusion from Section 2.2.1. We represent the two sides of the rule as dependence trees, some leaves of which are variables represented by circled numbers. During the rule application, the instantiations of the left-hand side variables are substituted against their counterparts on the right-hand side. Figure 2.4 demonstrates how the conditions of applicability and the performance of the two sides of a rule are reported to the programmer. Notice in Figure 2.4 the "functional" fcomp, i.e. a special node used to specify rules in which

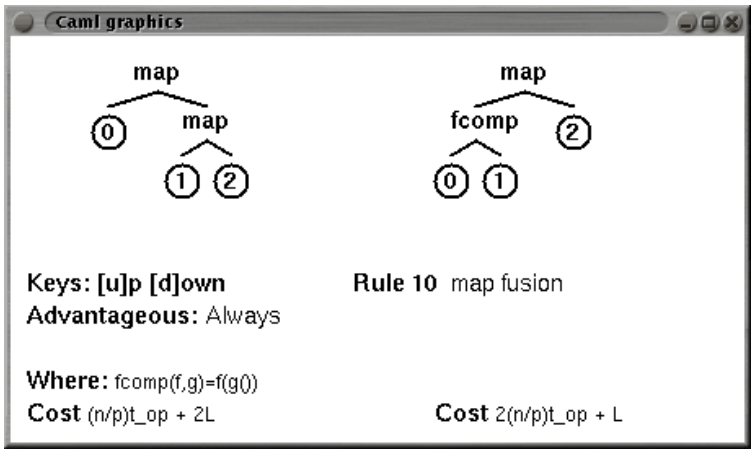


Figure 2.4: Internal representation of rule map fusion, conditions of its applicability and performance of the two sides of the rule.

two (or more) variables of the pattern are rewritten in the functional composition of them. Since variables have no sons, Meta first rewrites variables as sons of fcomp, then it makes the contractum $\{\nu_0 = f_0, \ldots \nu_n = f_n\}$ and, afterwards the result is equated using $\mathsf{fcomp}(f_0, \ldots f_n) = f_n \circ \cdots \circ f_0$.

Rule matching

Since programs and rules are represented by trees, we can state the problem of finding a candidate rule for transforming an expression as the well-known *subtree matching problem* [115, 133, 106]. In the most general case, given a pattern tree P and a subject tree T, all occurrences of P as a subtree of T can be determined in time $\mathcal{O}(|P| + |T|)$ by applying a fast string matching algorithm to a proper string representation [133]. Our problem is a bit more specific: the same patterns are matched against many subjects and the subject may be modified incrementally by the sequence of rule applications. Therefore, we distinguish a *preprocessing phase*, involving operations on patterns independent of any subject tree, and a *matching phase*, involving all operations dependent on some subject tree. Minimizing the matching time is our first priority.

The Hoffmann-O'Donnell bottom-up algorithm [106] fits our problem better than the string matching algorithm. With it, we can find all occurrences of a forest of patterns F as subtrees of T in time $\mathcal{O}(|T|)$, after suitable preprocessing of the pattern set. Moreover, the algorithm is efficient in practice: after the preprocessing, all the occurrences of elements in F can be found with a single traversal of T. The algorithm works in two steps: it constructs a *driving table*, which contains the patterns and their interrelations; then, the table is used to drive the matching algorithm.

The key idea of the Hoffmann-O'Donnell bottom-up matching algorithm is to find, at each point (node) in the subject tree, the set of all patterns and all parts of patterns which match at this point. Once we have assigned these sets to each node, we have essentially solved the matching problem, since each match is triggered by the presence of a complete pattern in some set. Notice that there can be only finitely many such sets, because both the kinds of nodes and the set of sub-patterns are finite. Thus we could precompute these sets, and code them by some enumeration to build driving tables. Given such tables, the matching algorithm becomes straightforward: traverse the subject tree in postorder and assign to each node the code of the set of partial matches. However, for certain pattern forest the number of such sets (thus the complexity of the generation of driving tables) grows exponentially with the cardinality of the pattern set. For an extensive description of the bottom-up matching we refer back to Hoffmann-O'Donnell paper [106] and Meta papers [8, 9].

Nevertheless, there is a broad class of pattern sets which can be preprocessed in polynomial time/space in the size of the set. The current set of Skel-BSP rules [14, 15, 172] and FAN rules [18] can be fully described by patterns that can be preprocessed in polynomial time/space. In addition, since the driving table depends only on the language and on the list of rules, it can be generated once and for all for a given set of rules and permanently stored for several subsequent match searches.

2.2.3 Tool architecture and implementation

The transformation engine applies the matching algorithm in an interactive cycle as follows:

1. Use the matching algorithm to annotate the dependence tree with the matching rules.
2. Check whether the rules found satisfy the type constraints and whether the side conditions hold (possibly interacting with the user).
3. Apply the performance estimates to establish the effect of each rule.
4. Ask the programmer to select one rule for application. In case no rule is applied, terminate; otherwise start again from Step 1.

We envision the Meta tool as a part of a general tool implementing a transformational refinement framework for a given target language TL. The global tool structure is depicted in Figure 2.5 (the part already implemented is highlighted with a dotted box). The whole system has two main capabilities: the conversion between TL programs and their internal representation (dependence tree) and the transformation engine working on dependence trees.

The system architecture is divided into five basic blocks:

1. The *Front End* converts a TL program into a parse tree and a data flow graph.

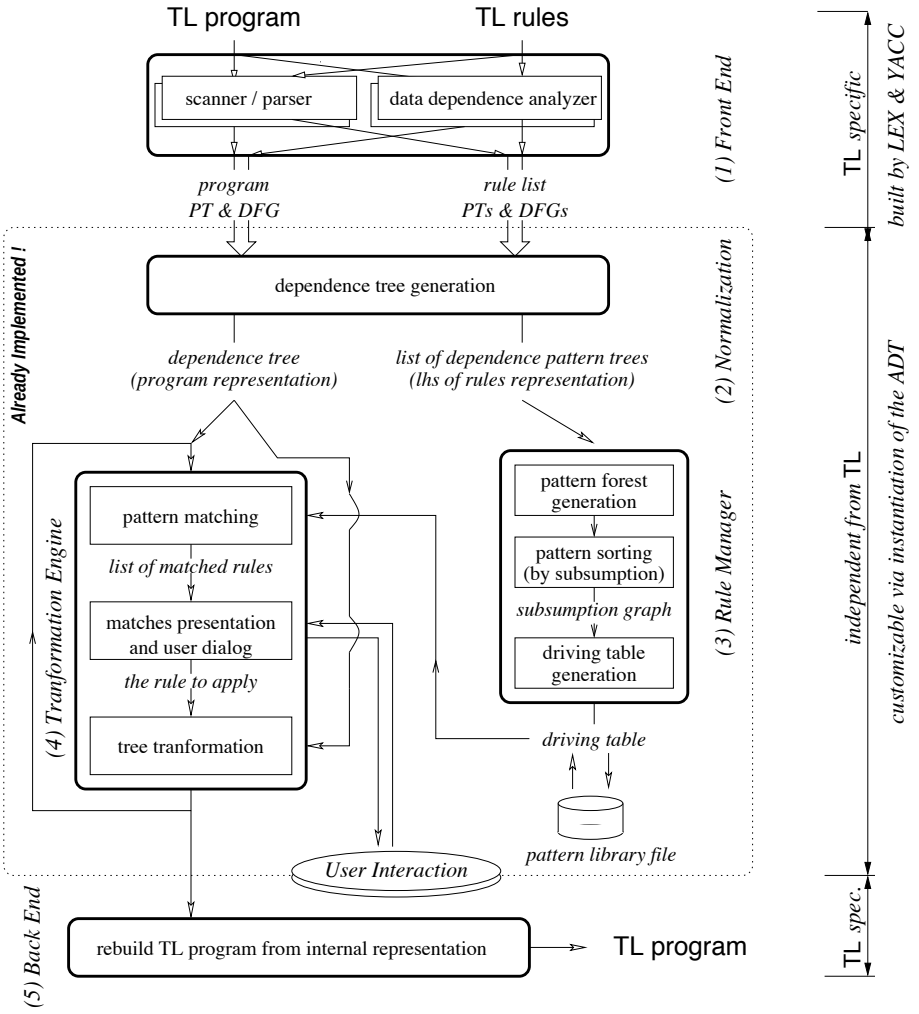


Figure 2.5: Global structure of the Meta transformation system.

2. The *Normalization* uses the PT and DFG to build the dependence tree both for the TL program and for the set of transformation rules.

3. The *Rule Manager* implements the preprocessing of rules (preprocessing phase, see Section 2.2.2); it delivers a matching table to drive the transformation engine. The driving table may be stored in a file.

4. The *Transformation Engine* interacts with the user and governs the transformation cycle.

5. The *Back End* generates a new TL program from the internal representation.

A prototype of the system kernel (highlighted in Figure 2.5 with a dotted box) has been implemented in Objective Caml 2.02. Our implementation is based on an

abstract data type (ADT) which describes the internal representation (dependence tree) and the functions working on it. The implementation is very general and can handle, via instantiation of the ADT, different languages with the requirement that rules and programs are written in the same language. Moreover, since several execution models and many cost calculi may be associated with the same language, any compositional way of describing program performance may be embedded in the tool by just instantiating the performance formulae of every construct. We call a cost calculus *compositional* if the performance of a language expression is either described by a function of its components or by a constant.

The Meta transformation tool prototype is currently working under both Linux and Microsoft Windows. A graphical interface is implemented using the embedded OCaml graphics library.

2.2.4 A case study: design by transformation

We discuss how Meta can be used in the program design process for the MSS algorithm, introduced in Section 2.2.1 and reported in the top-left corner of table 2.3.

First, the tool displays the internal representation of the program (Figure 2.6 (a)) and proposes 5 rules (Figure 2.6 (b)). The first one is pipe→comp rule, the others are instances of the farm introduction rule. The four stages of the pipe use exactly the same data distribution, but since each stage use a different set of processors each stage has to scatter and gather each data item. Transforming the pipe in a comp (that uses just one set of processors) would get rid of many unnecessary data re-distributions. Let us suppose the user chooses to apply the pipe→comp rule achieving the program version shown in Figure 2.6 (c). Next, Meta proposes a pair of rules (Figure 2.6 (d)): SAR-ARA to further reduce the number of communications into the comp, thus to optimize the program behavior on a single data item, and farm introduction to enhance the parallelism among different data items of the stream. Both rules may improve the performance of the program, let us suppose to choose the SAR-ARA (Figure 2.6 (e)).

Then, the transformation process continues choosing (in sequence) map fusion rule (2 times) and farm introduction rule. The resulting program is only one of the more than twenty different formulations Meta is able to find applying the transformation rules to the initial program. table 2.3 shows some of the semantic-equivalent formulations derivable.

In the rest of this section, we discuss a cost prediction model for Skel-BSP and we give some results of its accuracy on a concrete parallel architecture.

It is worth reminding that choosing in every step the transformation with the best performance gain does not guarantee to find the fastest program (optimum). Nevertheless, the knowledge of the performance gain/loss of each transformation is quite important to the programmer, since they can make decisions or build transformation strategies (e.g. greedy, tabu search, etc.) using such kind of information. An accurate prediction of transformations cost is quite important to this end.

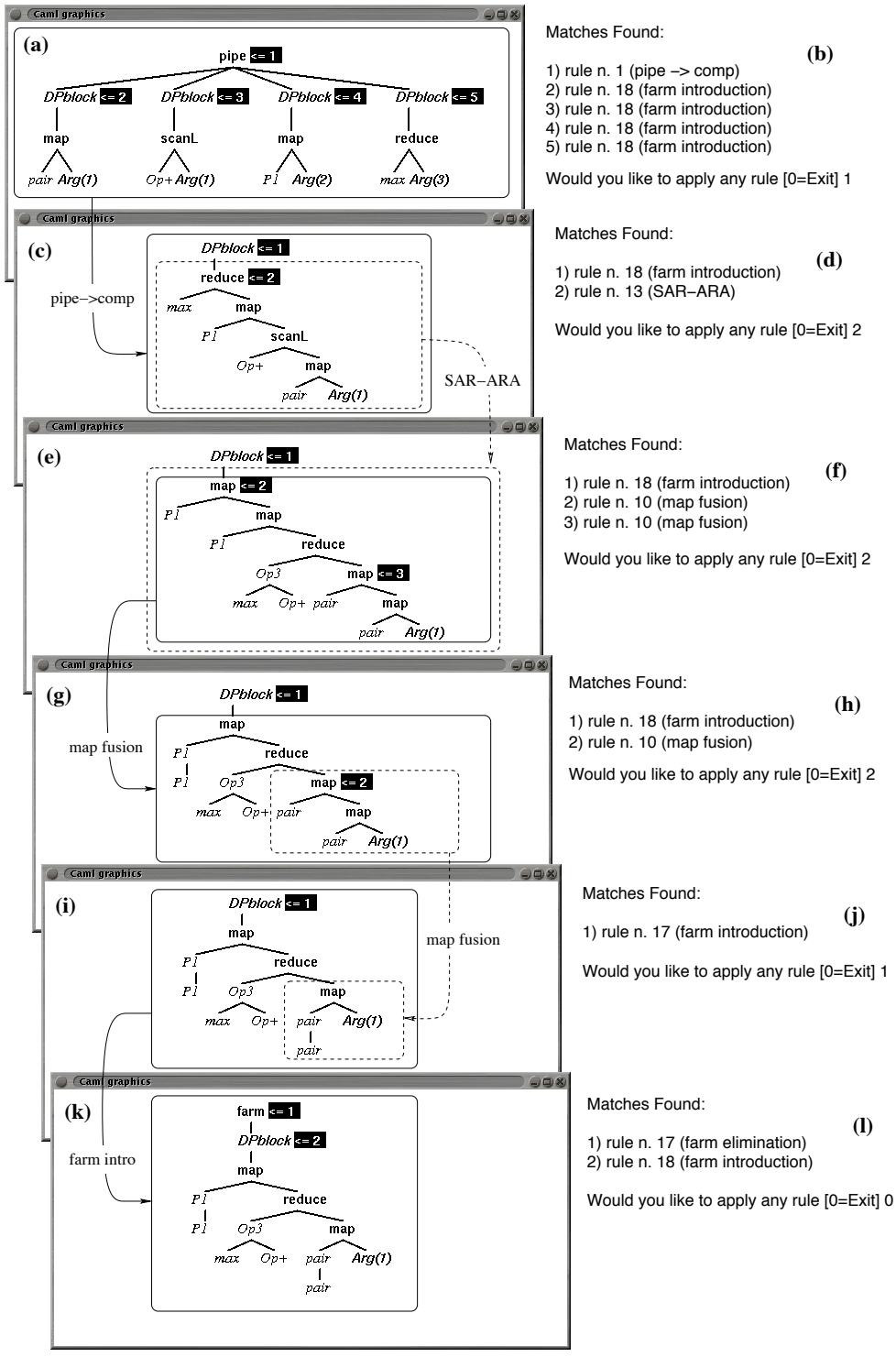


Figure 2.6: Transformation of the MSS program using the Meta tool. Skel-BSP skeletons are in serif font. Special nodes are in *slanted serif* font. Sequential functions are in *italic* font.

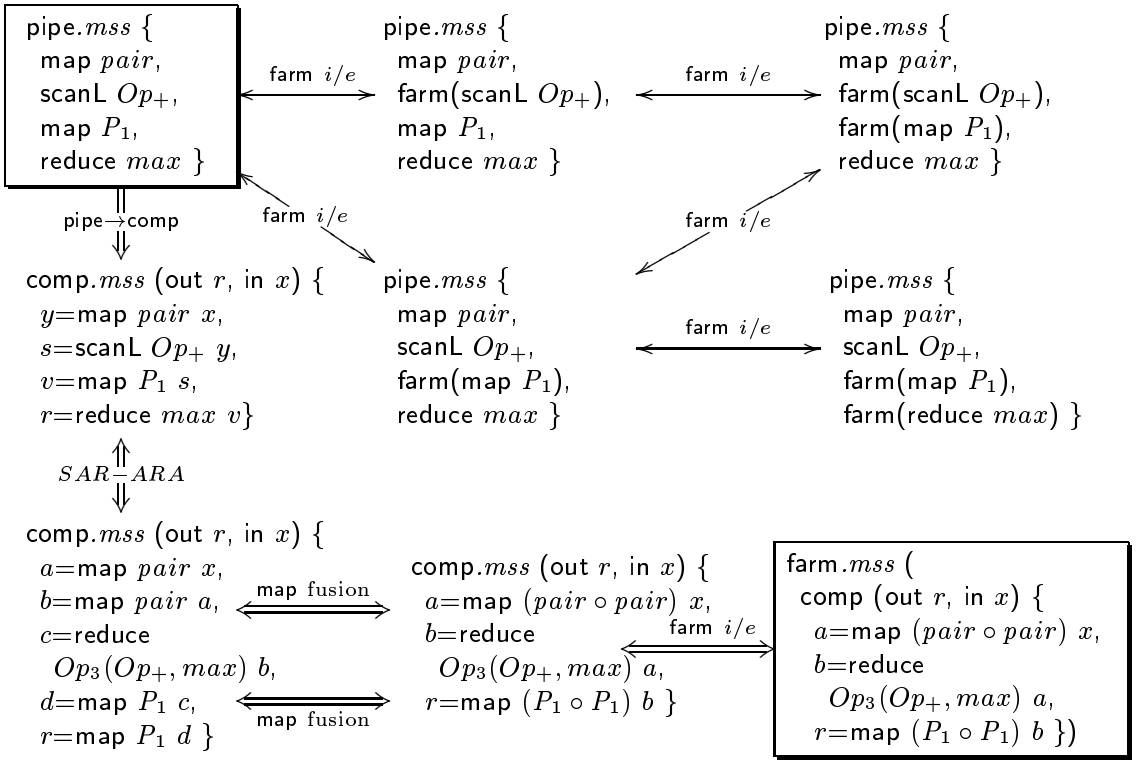


Table 2.3: Some of the transformations proposed by Meta for the MSS example. The double-arrow path denotes the derivation path followed in Figure 2.6.

In the case of Skel-BSP equipped with BSP costs, such prediction is pretty accurate. In the following section, we give evidence of this accuracy through the following steps. We first describe how Skel-BSP is implemented on a BSP abstract machine running on our concrete parallel architecture. Then we describe how programs and rules can be costed in this implementation. Finally, we compare the predicted and measured performance figures of two versions of our MSS example and compare the performance gain predicted by one transformation rule used by Meta with the real measured figures.

Prototyping Skel-BSP. Our Skel-BSP prototype is implemented using the C language and the PUB library (Padeborn University BSP-library [44]). The PUB library is a C-library of communication routines. These routines allow straightforward implementation of BSP algorithms.

The PUB library offers the implementation of a superset of the *BSP Worldwide Standard Interface* [96]. In addition, PUB offers some collective operations (`scan` and `reduce`), and it allows creating independent BSP objects each representing a virtual BSP computer. The last two features make PUB particularly suitable for prototyping Skel-BSP programs:

1. PUB collective operations may be used to implement Skel-BSP collective operations in a straightforward way. Unfortunately, PUB requires all operations used in `scan` and `reduce` to be commutative. Thus, the direct mapping from PUB to Skel-BSP collective operations may be done only if operations involved are commutative.

2. Independent (virtual) BSP computer may be used to implement effectively task parallel skeletons in Skel-BSP. Task parallel activities are often asynchronous on different pool of processors, and do not require all processing elements to synchronize at each superstep. PUB offers the possibility to divide a BSP computer in several subgroups each representing a virtual BSP computer. In this way, computations among processors belonging to different subgroups may proceed asynchronously since superstep barriers involve only processors belonging to the same subgroup.

Since global operations Op_+ and Op_3 we used in MSS programs are associative but not commutative, we extended PUB with a new parallel prefix operation (`TPscanL`) that requires global operations only to be associative. `TPscanL` is implemented using message passing primitives of PUB (send, receive, broadcast) following the two-phase BSP algorithm:

1. Each processor performs a (local) reduce on the local portion of the structure and broadcasts the result to all the processors with greater index.

2. Processor $i > 0$ computes the i^{th} segment of the prefix performing a local scan of the prefix array extended (on the left) with the i results received from all processors with index lower than i.

The two-phase parallel prefix algorithm is sketched in Figure 2.7 using + as global operation. Let p be the number of processors, n the length of prefix array (assumed multiple of p), t_{Op} the cost of the global operation, msg the size of a prefix array element and $\{g, l\}$ the usual BSP cost parameters. The BSP cost of `TPscanL` is:

$$T(\texttt{TPscanL}\ \ Op) = \underbrace{\left(\frac{n}{p} - 1\right)\ t_{Op}}_{\text{Phase1}} + \underbrace{g\ (p-1)\ msg + l}_{\text{barrier}} + \underbrace{\left(\frac{n}{p} + p - 2\right)\ t_{Op}}_{\text{Phase2}}$$

The two-phase algorithm is just one of the possible choices for the parallel prefix problem. We use the two-phase parallel prefix `TPscanL` to implement both Skel-BSP scanL and reduce. Notice that, to check the effectiveness of the transformation process, we only need to have an implementation with known cost, we do not need a particularly good implementation. For a comparison between two-phase algorithm and others BSP parallel prefix algorithms we refer back to [172].

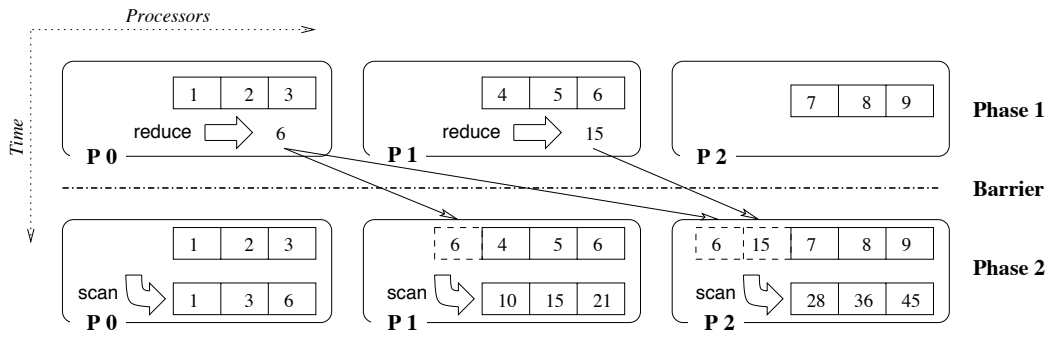


Figure 2.7: Two-phase BSP parallel prefix (`TPscanL`) using + as global operation.

Running and costing MSS programs. We focus on two different MSS Skel-BSP programs found using Meta and the proposed set of rules. Let us call *mss_c* and *mss_e* the programs in Figure 2.6 (c) and (e), respectively. *mss_e* is obtained from *mss_c* using the SAR-ARA rule, as follows:

comp.*mss_c* (out r, in x) {
y=map *pair* x,
s=scanL Op_+ y,
v=map P_1 s,
r=reduce max v}

$\Longrightarrow^{SAR-ARA}$

comp.*mss_e* (out r, in x) {
a=map *pair* x,
b=map *pair* a,
c=reduce
$Op_3(Op_+, max)$ b,
d=map P_1 c,
r=map P_1 d }

We describe the expected BSP cost of the two programs. Afterwards, we run a prototype of the two programs on a concrete parallel architecture, consisting in a cluster of Pentium II PCs (@266MHz) interconnected by a 100Mbit switched Ethernet. We instantiate cost formulae with BSP parameters collected during the experiments, miming the behavior of Meta. Finally, we discuss the accurateness of expected performance predicted by Meta using cost formulae with respect to experimental performance.

Let us assume each processor holds n/p elements of the input array. Since the comp skeleton executes its components in sequence, the cost of the *mss_c* program is figured out summing up the costs of each skeleton appearing into the comp. Notice we use the same primitive (`TPscanL`) to implement both scanL and reduce, thus the reduce will cost as much as scanL. All operations work on integers (4 bytes long). The *pair* operation consists in copying an integer, thus costs one BSP basic operation ($1 \cdot s$); the cost of the projection P_1 is zero. Messages sizes are two integers for the first `TPscanL` and one integer for the second one. The cost of Op_+ operation is assessed in $3 \cdot s$, while max costs just $1 \cdot s$. In total, we assess for the *mss_c*

program:

$$\begin{aligned} T(mss\_c) &= T(\mathsf{map}\ pair\ \mathsf{int}) + T(\texttt{TPscanL}\ \ Op_+) + T(\texttt{TPscanL}\ \ max) \\ &= s \cdot (9\ n/p + 4p - 12) + 12g(p-1) + 2l \end{aligned}$$

In the same way we evaluate the cost of the *mss_e* program. The first *pair* operation consists in copying one integer while the second *pair* in copying two integers, the total cost is $3 \cdot s$. The message size for `TPscanL` is four integers. The cost of $Op_3(Op_+, max)$ is $7 \cdot s$. In total we assess for the *mss_e* program:

$$\begin{aligned} T(mss\_e) &= T(\mathsf{map}\ pair\ \mathsf{int}) + T(\mathsf{map}\ pair\ \mathsf{int}[2]) + T(\texttt{TPscanL}\ \ Op_3(Op_+, max)) \\ &= s \cdot (17\ n/p + 7p - 21) + 16g(p-1) + l \end{aligned}$$

Each run consists in evaluating the MSS for 300 input arrays on several cluster configurations (2, 4, 8, 16 PCs). The length of input arrays ranges from 2^{13} to 2^{18} integers. All standard BSP parameters are profiled directly by the PUB library:

$$\begin{aligned} s &= 5.7 \cdot 10^{-8} \quad (17.54\ \text{M}\ BSP_{Ops}/\text{sec}) \\ g &= 0.2 \cdot 10^{-6} \quad (500\ \text{K Bytes/sec}) \\ l &= 3.2 \cdot 10^{-4} \cdot p \quad (640 - 5120\ \mu\text{secs, with } p = 2 - 16) \end{aligned} \tag{2.1}$$

Predicted performance and experimental performance of *mss_c* and *mss_e* programs are compared in Figure 2.8 a) and b), respectively. Considering all experiments, the average relative error of predicted performance with respect to experimental performance is 13% with 7% of standard deviation.

Performance driven transformations

Given a language and a set of semantic-preserving transformations, the Meta tool assists the user in the transformation process. The transformation process may be also *performance driven*, provided each rewriting rule is equipped with a performance formula, which depends on the particular skeleton implementations for the target language. In such case, proposing a transformation to the user, Meta suggests in which cases the transformation is advantageous, and what is the predicted performance for the transformed program. The prediction is figured out instantiating performance formulae with architecture parameters (e.g. BSP parameters) and basic operations cost (e.g. t_{Op_+}, t_{Op_3}).

Let us consider the application of SAR-ARA rule. Prototyping Skel-BSP as described in Section 2.2.4, thus supposing both scanL and reduce Skel-BSP skeletons are implemented using `TPscanL`, and BSP parameters are assigned as (2.1), the SAR-ARA rule is advantageous when:

$$n < \frac{p^2}{s}\left(-\frac{3\ s}{8} - \frac{g}{2}\right) + \frac{p}{s}\left(\frac{9\ s}{8} + \frac{g}{2} + l\right) = 1.6p + 654.9p^2 \tag{2.2}$$

Instantiating this formula with n and p, the Meta user may decide for each instance of the problem if the SAR-ARA application is advantageous, i.e. if *mss_e* perform better than *mss_c*. The same decision may be made on real data using Figure 2.9, which offers another view of data collected running *mss_c* and *mss_e* programs on several cluster configurations and array lengths. In the picture, given a point (p, n) in the (x, y)-grid, the best MSS program for that point is the one that belongs to the lower surface in the point. The (interpolated) intersection of the two surfaces is projected on the (x, y)-plane.

Finally, to give the flavor of accurateness in performance gain/loss prediction for rules, we compare the predicted and experimental behavior of Skel-BSP SAR-ARA rule. In Figure 2.10 the (p, n)-plane is partitioned by equation (2.2) and by the experimental performance of both *mss_c* and *mss_e*. The picture shows that (2.2) strikingly models the real behavior of the two programs in this case.

Notice that a more effective implementation of reduce would move the border in Fig 2.10 making greater the area where *mss_e* is faster.

Conclusions

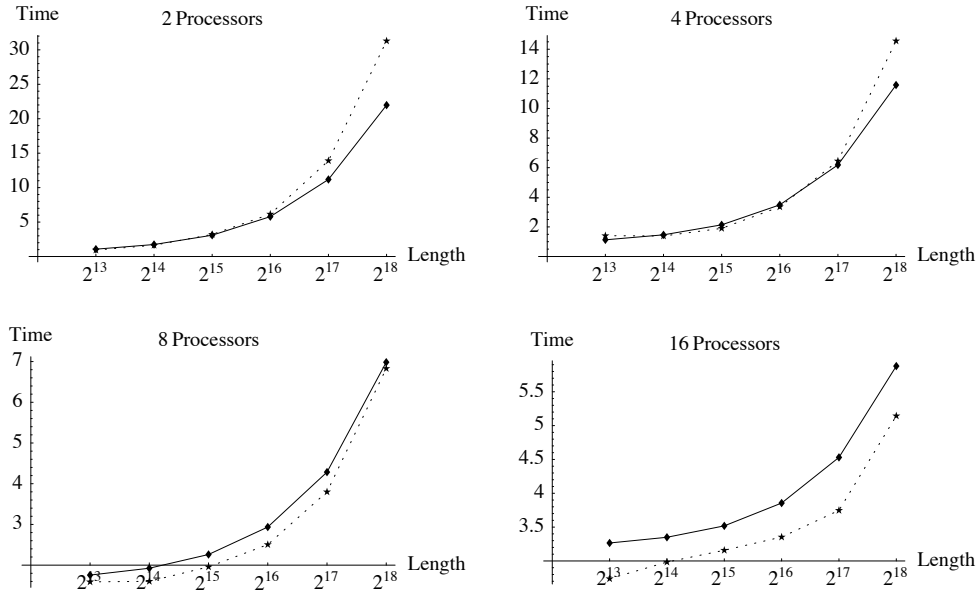
We have discussed the design and the implementation of an interactive, graphical transformation tool for skeleton-based languages. The Meta tool is (indeed) language-independent and is easily customizable with a broad class of languages, rewriting rules and cost calculi.

The design of our transformation engine Meta was influenced by the PARAMAT system [119]. However, our approach differs in many aspects. First, our goal is the optimization of high-level parallelism, rather than the parallelization of low-level sequential codes. Second, we do not define (as PARAMAT does) any a priori "good" parallel structure, we rather try to facilitate the exploration of the solution space towards the best parallel structure.

In addition to the described features, Meta may be instantiated with a set predefined heuristics to work as semi-automatic optimization tool. As an example Meta recognizes Skel-BSP data-parallel-free programs and optimizes them with a standard sequence of rewriting rules. Such program formulation (called *normal form*) is proved to be, under mild requirements, the fastest among the semantic-equivalent formulations that can be obtained using the rewriting rules [15].

Meta assists the user in the transformation process also driving it with performance predictions, even if, it is clear that the accurateness of prediction made by Meta primarily depends on the accurateness of the target language cost calculus. The use of Meta with FAN has proved that in many cases good parallel programs can be obtained via transformations [18]. Described experiments (Section 2.2.4) on Skel-BSP enforce the accurateness in the prediction of performance gain/loss due to a rule.

a) ***mss_c*** program (♦——♦ predicted performance, ★·····★ experimental performance)



b) ***mss_e*** program (♦——♦ predicted performance, ★·····★ experimental performance)

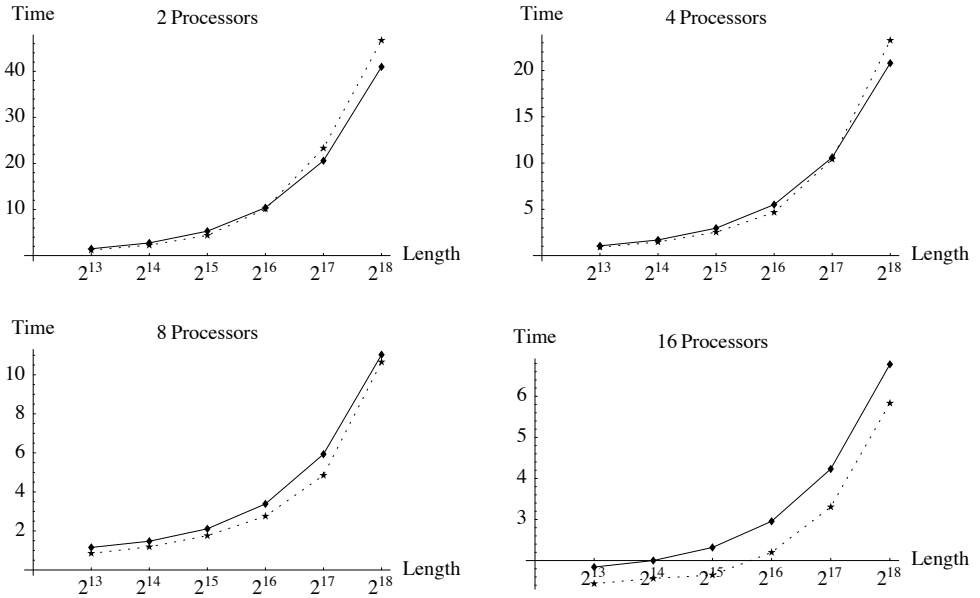


Figure 2.8: a) **mss_c** and b) **mss_e**: Comparing predicted performance (solid lines) with experimental performance (dotted lines). Each experiment is performed on several array lengths (x-axis). Four different cluster configurations are experimental (2,4,8,16 processors).

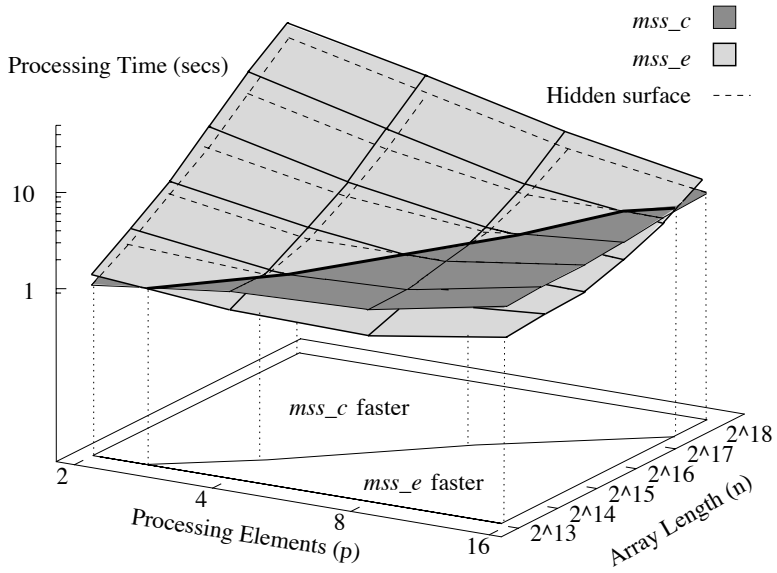


Figure 2.9: Experimental performance of **mss_c** and **mss_e** programs on several cluster configurations and several array lengths.

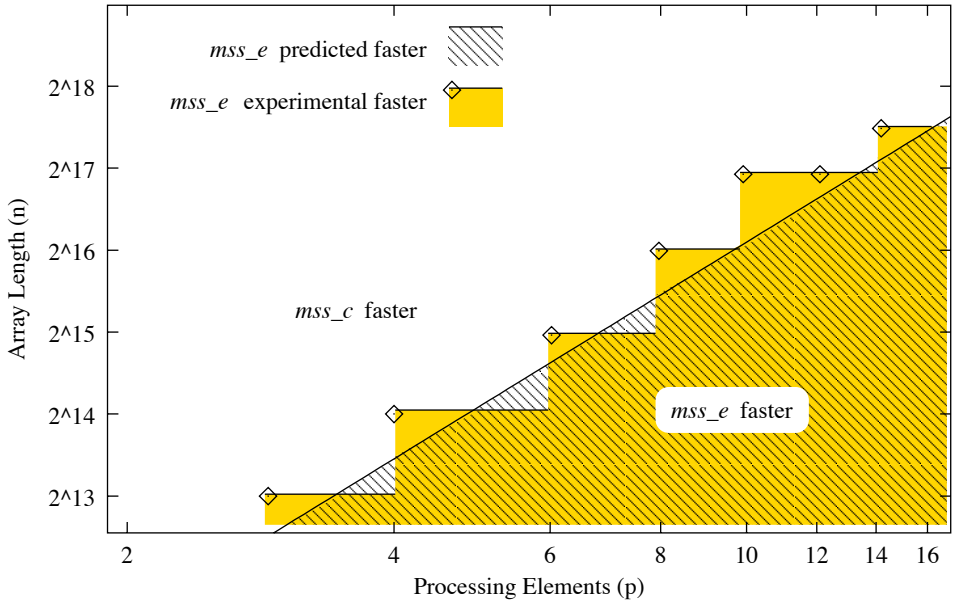


Figure 2.10: SAR-ARA rule: Predicted and experimental behavior (**mss_e** is faster than **mss_c** when SAR-ARA is advantageous).

2.3 Exploiting efficient skeletons in Java

The Java programming environment includes features that can be naturally used to address network and distributed computing: JVM and bytecode, multi-threading, remote method invocation, socket and security handling and, more recently, JINI, Java Spaces, Servlets, etc. [160]. Many parallel/distributed applications have been developed using these features [111]. Also, many efforts have been performed to make Java a more suitable programming environment for parallel computing. In particular, several projects have been started that aim at providing features that can be used to develop efficient parallel Java applications on a range of different parallel architectures [108, 155]. Such features are either provided as extensions to the base language or as class libraries. In the former case, ad hoc compilers and/or run-time environments have been developed and implemented. As an example extensions of the JVM have been designed that allow plain Java threads to be run in a seamless way on the different processors of a single SMP machine [20, 23]. In the latter, libraries are supplied to the programmer that simply uses them within his parallel code [134, 113].

In this section we discuss a new Java parallel programming environment, Lithium, which differs from the environments cited above and which is meant to be a further step in the direction of the design of user friendly, efficient, parallel programming environments based on Java. Lithium is a Java library that supports *structured* parallel programming. It is based on the *algorithmical skeleton* concept. Skeletons have been originally conceived by Cole [59] and then used by different research groups to design high performance structured parallel programming environments [26, 29, 154]. A skeleton is basically an abstraction modeling a common, reusable parallelism exploitation pattern. Skeletons can be provided to the programmer either as language constructs [26, 29] or as libraries [72, 75, 120]. They can be nested to build complex parallel applications. The compiling tools of the skeleton language or the skeleton libraries take care of automatically deriving/executing actual, efficient parallel code out of the skeleton application without any direct programmer intervention [140, 75].

In order to write a working parallel application using a skeleton based parallel programming environment, the programmer must usually perform the following steps:

- ⋆ first, the programmer must express the *parallel structure* of the application by using a proper skeleton nesting;
- ⋆ then, the programmer must write the *application specific sequential portions of code* used as skeleton parameters;
- ⋆ eventually the programmer must simply *compile and link* the resulting code to obtain running, parallel object code.

Lithium provides the programmer with a full Java, skeleton based parallel programming environment. The library supports common skeletons, including pipelines,

task farms, iterative and data parallel skeletons. Using Lithium, the programmer can set up a parallel application instantiating the skeletons provided, nesting them, providing tasks to be computed (input data), asking the parallel computation of the resulting program on a set of interconnected workstations and eventually get (and use) the results of such parallel computation.

As an example, the programmers can express their parallel application as a pipeline having stages that are either sequential or exploit data parallelism. Then, they can prepare the sequential portions of code implementing sequential stages and those implementing data decomposition, processing and recomposition relative to the data parallel stages. They can denote such portions of code as the proper pipeline, data parallel stages. Eventually they can compile and run the resulting program. When a working instance of the application has been obtained and tested, the programmer can start a *performance refinement* step. During this activity they can tune either the skeleton structure or the skeleton parameters in such a way that performance bottlenecks are removed/mitigated [140].

The implementation of Lithium fully exploits Java RMI to distribute computations across different processing elements of the target architecture. Java reflection features are also exploited to make the Lithium API simpler. Last but not least, the clean OO structure of Lithium code also allows the sequential execution (emulation, actually) of a parallel program onto a single machine. This is a very useful feature during functional application code debugging. Actually, Lithium represents a consistent refinement and development of a former work [69]. Main differences lay in the larger skeleton set implemented ([69] only handles embarrassingly parallel computations, while Lithium provides a complete skeleton set) and in the implementation of a set of optimization rules that may significantly enhance skeleton program performances. The optimization rules implemented in Lithium extend the ones discussed in [15] and [9]. They have not yet been used in any other skeleton based programming environment.

The Lithium library is presented as follows: in Section 2.3.1 we describe the skeletons provided by Lithium; in Section 2.3.2 we describe the optimization strategies implemented by Lithium (and available on user request); in Section 2.3.3 we outline the Lithium API. Eventually, in Section 2.3.4 we describe how the library is implemented exploiting macro data flow, and in Section 2.3.5 we report achieved experimental results.

2.3.1 Lithium skeletons

Lithium provides both task parallel and data parallel skeletons. All the skeletons work on streams, i.e. process a stream of input tasks to produce a stream of results. We denote streams[3] by angled braces (e.g. $\langle x_1, x_2, \ldots, \tau \rangle$) and we use a non strict stream constructor denoted by :: having type (Stream $\times$ Stream $\rightarrow$ Stream). Empty

[3]Both finite and infinite streams, i.e. lists and non-strict lists, respectively.

1. $\mathsf{seq}\ f\ \langle x,\tau\rangle_\ell \to \langle f\,x\rangle_\ell :: \mathsf{seq}\ f\ \langle\tau\rangle_\ell$
2. $\mathsf{farm}\ \Delta\ \langle x,\tau\rangle_\ell \to \Delta\ \langle x\rangle_{\mathcal{O}(\ell,\mathrm{x})} :: \mathsf{farm}\ \Delta\ \langle\tau\rangle_{\mathcal{O}(\ell,\mathrm{x})}$
3. $\mathsf{pipe}\ \Delta_1\ \Delta_2\ \langle x,\tau\rangle_\ell \to \Delta_2\ [\Delta_1\ \langle x\rangle_\ell]_{\mathcal{O}(\ell,\mathrm{x})} :: \mathsf{pipe}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell$
4. $\mathsf{comp}\ \Delta_1\ \Delta_2\ \langle x,\tau\rangle_\ell \to \Delta_2\ [\Delta_1\ \langle x\rangle_\ell]_\ell :: \mathsf{comp}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell$
5. $\mathsf{map}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle x,\tau\rangle_\ell \to f_\mathsf{c}\ (\alpha\ \Delta)\ f_\mathsf{d}\ \langle x\rangle_\ell :: \mathsf{map}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle\tau\rangle_\ell$
6. $\mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle x,\tau\rangle_\ell \to \mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle x\rangle_\ell :: \mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle\tau\rangle_\ell$
   $$\mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle y\rangle_\ell = \begin{cases} \Delta\ \langle y\rangle_\ell & \textit{iff}\ \ (f_\mathsf{tc}\ y) \\ f_\mathsf{c}\,(\alpha\,(\mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}))\ f_\mathsf{d}\ \langle y\rangle_\ell & \textit{otherwise} \end{cases}$$
7. $\mathsf{for}\ i\ \Delta\ \langle x,\tau\rangle_\ell \to \underbrace{\Delta(\Delta(\cdots(\Delta}_{i\ \text{times}}\langle x\rangle_\ell)\cdots)) :: \mathsf{for}\ i\ \Delta\ \langle\tau\rangle_\ell$
8. $\mathsf{while}\ f_\mathsf{tc}\ \Delta\ \langle x,\tau\rangle_\ell \to \begin{cases} \mathsf{while}\ f_\mathsf{tc}\ \Delta\ (\Delta\ \langle x\rangle_\ell :: \langle\tau\rangle_\ell) & \textit{iff}\ \ (f_\mathsf{tc}\ x) \\ \langle x\rangle_\ell :: \ \mathsf{while}\ f_\mathsf{tc}\ \Delta\ \langle\tau\rangle_\ell & \textit{otherwise} \end{cases}$
9. $\mathsf{if}\ f_\mathsf{tc}\ \Delta_1\ \Delta_2\ \langle x,\tau\rangle_\ell \to \begin{cases} \Delta_1\langle x\rangle_\ell :: \mathsf{if}\ f_\mathsf{tc}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell & \textit{iff}\ \ (f_\mathsf{tc}\ x) \\ \Delta_2\langle x\rangle_\ell :: \mathsf{if}\ f_\mathsf{tc}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell & \textit{otherwise} \end{cases}$

Figure 2.11: Lithium operational semantic. x, y $\in$ Value; $\sigma,\tau \in$ Value*; $\ell,\ell_i,\ldots \in$ Label = Strings $\cup\ \{\bot\}$; $\mathcal{O}$: Label $\times$ Value $\to$ Label.

$\mathsf{farm}\ (\mathsf{seq}\ f)\ \langle x_1,x_2,x_3,x_4,x_5,x_6\rangle_\bot \to^*$
$\mathsf{seq}\ f\ \langle x_1\rangle_1 :: \mathsf{seq}\ f\ \langle x_2\rangle_2 :: \mathsf{seq}\ f\ \langle x_3\rangle_3 :: \mathsf{farm}\ (\mathsf{seq}\ f\ )\ \langle x_4,x_5,x_6\rangle_\bot \to^*$
$\mathsf{seq}\ f\ \langle x_1\rangle_1 :: \mathsf{seq}\ f\ \langle x_2\rangle_2 :: \mathsf{seq}\ f\ \langle x_3\rangle_3 :: \mathsf{seq}\ f\ \langle x_4\rangle_4 :: \mathsf{seq}\ f\ \langle x_5\rangle_5 :: \mathsf{seq}\ f\ \langle x_6\rangle_6 \to^*$
$\langle f\,x_1\rangle_1 :: \langle f\,x_2\rangle_2 :: \langle f\,x_3\rangle_3 :: \langle f\,x_4\rangle_4 :: \langle f\,x_5\rangle_5 :: \langle f\,x_6\rangle_6 \to^*$
$\langle f\,x_1,\ f\,x_2,\ f\,x_3,\ f\,x_4,\ f\,x_5,\ f\,x_6\rangle_\bot$

$\mathsf{pipe}\ (\mathsf{seq}\ f)\ (\mathsf{seq}\ g)\langle x_1,x_2,x_3,x_4\rangle_\bot \to^*$
$(\mathsf{seq}\ g)\,[\,\mathsf{seq}\ f\ \langle x_1\rangle_\bot\,]_1 :: (\mathsf{seq}\ g)\,[\,\mathsf{seq}\ f\ \langle x_2\rangle_\bot\,]_1 :: \mathsf{pipe}\ (\mathsf{seq}\ f)\ (\mathsf{seq}\ g)\langle x_3,x_4\rangle_\bot \to^*$
$(\mathsf{seq}\ g)[\mathsf{seq}f\langle x_1\rangle_\bot]_1 :: (\mathsf{seq}\ g)[\mathsf{seq}f\langle x_2\rangle_\bot]_1 :: (\mathsf{seq}\ g)[\mathsf{seq}f\langle x_3\rangle_\bot]_1 :: (\mathsf{seq}\ g)[\mathsf{seq}f\langle x_4\rangle_\bot]_1 \to^*$
$\mathsf{seq}\ g\ \langle f\ x_1\rangle_1 :: (\mathsf{seq}\ g)\,[\,\mathsf{seq}\ f\ \langle x_2\rangle_\bot\,]_1 :: (\mathsf{seq}\ g)\,[\,\mathsf{seq}\ f\ \langle x_3\rangle_\bot\,]_1 :: (\mathsf{seq}\ g)\,[\,\mathsf{seq}\ f\ \langle x_4\rangle_\bot\,]_1 \to^*$
$\langle g\circ f\ x_1\rangle_1 :: \mathsf{seq}\ g\ \langle f\ x_2\rangle_1 :: (\mathsf{seq}\ g)\,[\,\mathsf{seq}\ f\ \langle x_3\rangle_\bot\,]_1 :: (\mathsf{seq}\ g)\,[\,\mathsf{seq}\ f\ \langle x_4\rangle_\bot\,]_1 \to^*$
$\langle g\circ f\ x_1\rangle_1 :: \langle g\circ f\ x_2\rangle_1 :: \mathsf{seq}\ g\ \langle f\ x_3\rangle_1 :: (\mathsf{seq}\ g)\,[\,\mathsf{seq}\ f\ \langle x_4\rangle_\bot\,]_1 \to^*$
$\langle g\circ f\ x_1\rangle_1 :: \langle g\circ f\ x_2\rangle_1 :: \langle g\circ f\ x_3\rangle_1 :: \langle g\circ f\ x_4\rangle_1 \to^*$
$\langle g\circ f\ x_1,\ g\circ f\ x_2,\ g\circ f\ x_3,\ g\circ f\ x_4\rangle_\bot$

Figure 2.12: Stream label usage examples.

streams are denoted by $\langle\rangle$.

All the skeletons are assumed to be stateless. No concept of "global state" is supported by the implementation, but the ones explicitly programmed by the user, possibly exploiting plain Java mechanisms (e.g. RMI servers encapsulating shared data structures, whose services/methods are invoked by code used to express computations within skeletons). In particular, static (class) variables cannot be used in the definition of Lithium code to share information across different concurrent/parallel entities.

The Lithium skeletons are fully nestable. Each skeleton has skeleton type parameters that model the computations encapsulated in the related parallelism exploitation pattern.

The skeletons (Δ) provided by Lithium are defined as follows:

$$
\begin{array}{rl}
\Delta ::= & \mathsf{seq}\ f \mid \\
& \mathsf{farm}\ \Delta \mid \mathsf{pipe}\ \Delta_1\ \Delta_2 \mid \mathsf{comp}\ \Delta_1\ \Delta_2 \mid \\
& \mathsf{map}\ f_\mathsf{d}\ \Delta\ f_\mathsf{c} \mid \mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{d}\ \Delta\ f_\mathsf{c} \mid \\
& \mathsf{for}\ i\ \Delta \mid \mathsf{while} f_\mathsf{tc}\ \Delta \mid \mathsf{if} f_\mathsf{tc}\ \Delta_1\ \Delta_2
\end{array}
$$

$f, f_\mathsf{c}, f_\mathsf{d}, f_\mathsf{tc} ::= \textit{Sequential Java functions}$[4].

and a Lithium program is a skeleton expression:

$$\textit{Lithium\_prog} ::= \Delta : \langle\sigma\rangle \rightarrow \langle\tau\rangle$$

processing a stream of input data $\langle\sigma\rangle$ and producing a stream of output results $\langle\tau\rangle$. Sequential functions $f_\mathsf{c}, f_\mathsf{d}$ represent families of functions that enable the splitting of a singleton stream in tuples of singleton streams and vice-versa: $f_\mathsf{c} : \langle\circ\rangle^* \rightarrow \langle\circ\rangle$ and $f_\mathsf{d} : \langle\circ\rangle \rightarrow \langle\circ\rangle^*$, being $\langle\circ\rangle$ the singleton stream. As an example $f_\mathsf{c}, f_\mathsf{d}$ may be used to split an array in their rows/columns and join them back to the original array shape.

Intuitively, seq skeleton just encapsulates sequential portions of code in such a way they can be used as parameters of other skeletons; farm and pipe skeletons model usual task farm (*alias* embarrassingly parallel) computations and computations organized in *stages*; comp models pipelines with stages executed serially (on the same processing element); map models data parallel computations: f_d decomposes the input data into a set of possibly overlapping data subsets, the inner skeleton computes a result out of each subset and the f_c function rebuilds a unique result out of these results; d&c models divide and conquer computations: input data is divided into subsets by f_d and each subset is computed recursively and concurrently until the f_{tc} condition does not hold true. At this point results of subcomputations are conquered via the f_c function. Last but not least, for, while and if skeletons model finite iteration, indefinite iteration and conditional.

More formally, the operational semantics of the Lithium skeletons is described in

[4]Namely the `run()` method call of an instance of `JSkeletons` class (see Section 2.3.3).

Figure 2.11 (a full version of these semantics appears in [16]. Here curly braces denote tuples (e.g. $\{x_1, \cdots, x_k\}$), α represents the apply-to-all on tuples ($\alpha\, \Delta\, \{x_1, \cdots, x_k\}$ $= \{\Delta\ x_1, \cdots, \Delta\ x_k\}$) and all the functions used are assumed to be strict, but the infix stream constructor :: which is only strict in his first argument[5]. Actually, in all those cases where a rule such as

$$\mathsf{Skel} \ldots \langle x, \tau \rangle_\ell \rightarrow \mathcal{F}(x) :: \mathsf{Skel} \ldots \langle \tau \rangle_\ell$$

holds[6], then another further rule holds, which is not summarized in Figure 2.11 to avoid clobbering the figure:

$$\mathsf{Skel} \ldots \langle x \rangle_\ell \rightarrow \mathcal{F}(x)$$

In the computation of a Lithium skeleton program parallelism comes from two distinct sources:

- *data parallelism*: all the computations in the α apply-to-all may be performed in parallel (in map and d&c skeletons)
- *task parallelism*: each item of a stream (e.g. a (strict) function application appearing within some :: operators) may be computed in parallel with any other function application appearing in the same stream provided that their labels differ;

In fact, in this operational semantics, streams are labeled and different skeletons behave differently with respect to labels. The idea here is that labels are used to distinguish computations that may be performed in parallel from those that may not. Function $\mathcal{O}(\ell, \mathrm{x})$ simply returns a "fresh", unused label built using information coming from the previously used label ℓ and from the current data item x.

Therefore, each data item processed by a farm is given a fresh label, modeling the fact that embarrassingly parallel task farm computations can all possibly happen concurrently. Pipeline keeps labels in such a way that first and second stage cannot be computed in parallel on the same data item.

As an example, consider the computations described in Figure 2.12. The operational semantics rewrites the farm computation relative to a six item stream by an expression involving seq skeletons (first half of Figure 2.12). All the seq skeletons work on singleton stream with *different* labels. Therefore all the seq $f\ \langle x_i \rangle_{l_j}$ can be possibly computed concurrently. When pipelines are used (second half of Figure 2.12) another seq based expression is derived, whose terms happen to have all the same label. Therefore, at the beginning only one seq skeleton can be computed, i.e. the rightmost of those with the $\perp$ label. From this step on, two expressions happen to have a different label: one relative to the computation of the first pipeline stage

[5] Therefore :: evaluates arguments left-to-right.

[6] Skel $\in$ [seq, farm, pipe, comp, map, d&c, for, while, if]

(with $\perp$ label) and one relative to the computation of the second pipeline stage (with 1 label): these skeleton expressions can actually be computed in parallel.

Lithium supports most of the skeletons discussed in previous skeleton/structured parallel programming works [26, 29, 154, 140, 132], and allows a reasonably large number of parallel applications to be implemented. The architectural design of Lithium also allows relatively easy extension of the skeleton set (see Sec. 2.3.4) in case it is needed.

2.3.2 Skeleton optimizations

We define a *stream parallel skeleton composition* as a skeleton expression only holding pipe, farm and seq skeletons. For such composition we inductively define the *fringe* (ϕ) as follows:

$$\phi(\Delta) = \begin{cases} \mathsf{seq}\ f & \Delta = \mathsf{seq}\ f \\ \mathsf{comp}\ \phi(\Delta_1)\ \phi(\Delta_2) & \Delta = \mathsf{pipe}\ \Delta_1\ \Delta_2 \\ \phi(\Delta_w) & \Delta_w = \mathsf{farm}\ \Delta_w \end{cases}$$

In [15] we demonstrated that for any stream parallel skeleton composition Δ a *normal form* $\overline{\Delta}$ exists (with $\overline{\Delta} = \mathsf{farm}\ \phi(\Delta)$) such that it computes the same program computed by Δ with a performance equal or better than the one of the original skeleton composition Δ (i.e. $T_s(\overline{\Delta}) \leq T_s(\Delta)$, with $T_s(\Delta)$ the service time of the skeleton program Δ). Equivalence of normal and non normal form is derived by using the skeleton "functional" semantics that can be easily derived from the operational semantics described in Figure 2.11. The relationship between performance of normal and non normal form is derived using a simple, ideal, logP-like performance model taking into account both sequential computation time and communication time. As an example, the performance model defines $T_s(\mathsf{farm}\ \Delta)$ as the $min\{max\{T_i(\Delta), T_o(\Delta)\}, T_s(\Delta)\}$. Here, T_i and T_o represent the time spent in delivering a new task and retrieving the computed result to and from the processing element computing the task, respectively. In other words, the time needed to accept a new input task in an embarrassingly parallel computation (the stateless farm) is determined by the minimum between the time spent in communicating data to and from the remote processing elements and the time spent in actually computing results out of the input tasks.

Starting from these results, while developing Lithium we also demonstrated two further results concerning skeleton tree (nesting) optimizations. The first one extends normal form to data parallel skeleton nesting with stream parallel only workers:

> *given a skeleton program* $\Delta = \mathsf{map}\ f_d\ \Delta_w\ f_c$, *with* Δ_w *being a stream parallel skeleton composition, a normal form exists* $\overline{\Delta} = \mathsf{map}\ \ f_d\ \overline{\Delta_w}\ f_c$ *such that* $T_s(\overline{\Delta}) \leq T_s(\Delta)$.

The second one concerns resources (processing elements) used by normal a non normal form programs. In theory, and according to the operational semantics of Figure 2.11, the execution of a skeleton program needs, at any time, a set of processing elements holding a distinct processing element for each concurrent activity[7]. The amount of resources needed to compute a skeleton program is the maximum number of elements in this set measured during the whole program execution. We denote such number by $\#(\Delta)$. Under this assumptions:

> *for any skeleton program Δ being either a stream parallel skeleton composition or a* map $f_d\ \Delta_w\ f_c$ *with Δ_w a stream parallel skeleton composition then* $\#(\overline{\Delta}) \leq \#(\Delta)$.

A full proof of these new results is described in [161]. The point we want to make here is that these results guarantee that Lithium can perform effective optimizations in the execution of skeleton code. Actually, Lithium performs automatical transformation of skeleton nestings into normal form, before proceeding to compute the programs. The user may explicitly ask to avoid such transformations.

2.3.3 Lithium API

Lithium provides the programmer with a set of classes implementing the skeletons described in Section 2.3.1. The classes can be used to instantiate objects that will populate the skeleton nesting modeling the parallel behavior of the applications. All the skeletons are provided as subclasses of a `JSkeletons` abstract class. This class defines two abstract methods: the first one is a `public Object run(Object)` method. It is used to encapsulate a sequential portion of code (in case of sequential skeleton) or the code that sequentially emulates the parallel skeleton behavior (in case of the other, non sequential skeletons). The second method defined is a protected `Object[] getSkeletonInfo()` method, which is basically used by Lithium to gather the information needed to build the application skeleton nesting.

Therefore, a Lithium sequential skeleton is nothing but a subclass of the Lithium abstract class `JSkeletons` providing and implementation of the abstract method `public Object run(Object)`, while a farm is modeled via the `Farm JSkeletons` subclass, the pipe via the `Pipe` one, etc. All the details relative to skeleton definition in Lithium can be found in [161]. Lithium source code is available as open source[8] with Javadoc documentation as well.

Beside defining skeleton nestings, Lithium API provides a way to execute such skeleton programs. This is accomplished through objects of the `Ske` class. This class provides the object actually taking a skeleton program, a set of input tasks and providing to compute the program in parallel. After creating a `Ske` object, a `setProgram(JSkeletons pgm)` method can be invoked to define which skeleton

[7] I.e. each one of the function applications labelled with different labels and each one of the computations happening within an apply-to-all.

[8] At `www.sourceforge.net/projects/massivejava` or Lithium home page [62].

```
import lithium.*;  ...
public class SkeletonApplication {
  public static void main(String [] args) {
    ...
    // define skel. program
    Worker w = new Worker();
    Farm f = new Farm(w);
    // setup evaluator
    Ske evaluator = new Ske();
    evaluator.setProgram(f);
    String [] hosts = {"alpha1","alpha2",
                       "131.119.5.91"};
    evaluator.addHosts(hosts);
    // prepare input tasks
    for(int i=0;i<ntasks;i++)
      evaluator.setupTaskPool(task[i]);
    evaluator.stopStream();
    evaluator.parDo();
    // ask parallel computation
    while(!evaluator.isResEmpty()) {
        Object res = evaluator.readTaskPool();
        ...      // consume results
     }
  }}
```

Figure 2.13: Sample Lithium code: parallel application exploiting task farm parallelism.

program is to be executed, an `addHosts(String [] hostlist)` method can be called to provide the names of the machines to be used for parallel execution, some `setupTaskPool(Object task)` can be invoked to provide the task items of the input stream, a `parDo()` method call can be issued to start parallel program execution and eventually `Object readTaskPool()` method can be invoked to read the results computed.

In summary, in order to write parallel applications using Lithium skeletons, the programmer should perform the following steps:

1. define the skeleton structure of the application;
2. declare a **Ske** object and define the program (the skeleton code defined in the previous step) to be executed by the evaluator as well as the list of hosts to be used to run the parallel code;
3. setup a task pool hosting the initial tasks;
4. start the parallel computation issuing a `parDo()` method call;
5. retrieve and process the final results.

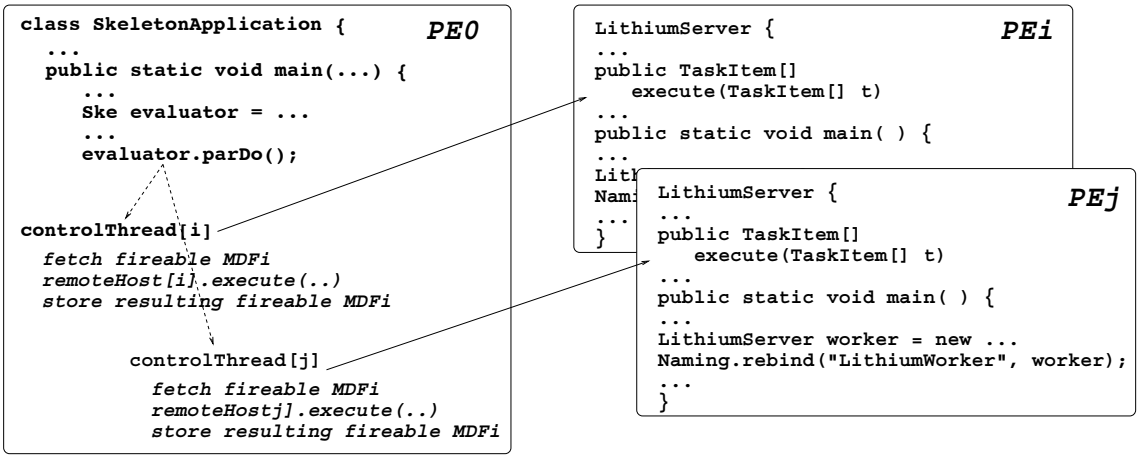


Figure 2.14: Lithium architecture.

Figure 2.13 outlines the code needed to setup a task farm parallel application processing a stream of input tasks by computing, on each task, the sequential code defined in the `Worker` `run` method. The application runs on three processors (the `hosts` ones). The programmers are neither required to write any (remote) process setup code nor any communication, synchronization and scheduling code. They issues an `evaluator.parDo()` call and the library automatically computes the `evaluator` program in parallel by forking suitable remote computations on the remote nodes. In case the user simply wants to execute the application sequentially (i.e. to functionally debug the sequential code), they can avoid to issue all the `Ske` evaluator calls. After the calls needed to build the `JSkeletons` program they can issue a `run()` method call on the `JSkeletons` object. In that case, the Lithium support performs a completely sequential computation returning the results that the parallel application would eventually return. We want to point out that, in case we understand that the computation performed by the farm workers of Figure 2.13 can be better expressed by a functional composition, we can arrange things in such a way that farm workers are two stage pipelines. This can be achieved by substituting the lines `Worker w = new Worker();` and `Farm f = new Farm(w);` with the lines:

```
Stage1 s1 = new Stage1();
Stage2 s2 = new Stage2();
Pipeline p = new Pipeline();
p.addWorker(s1);
p.addWorker(s2);
Farm f = new Farm(p);
```

and we get a perfectly running parallel program computing the results according to a farm of pipeline parallelism exploitation pattern. Therefore, a very small effort is needed to change the parallel structure of the application, provided that the suitable sequential portions of code needed to instantiate the skeletons are available.

2.3.4 Lithium implementation

Lithium exploits a macro data flow (MDF, for short) implementation schema for skeletons. The skeleton program is processed to obtain a MDF graph. MDF instructions (MDFi) in the graph represent sequential `JSkeletons` `run` methods. Data flow (i.e. the arcs of MDF graph) is derived by looking at the skeleton nesting structure [68, 70]. The resulting MDF graphs have a single MDFi getting input task (tokens) from the input stream and a single MDFi delivering data items (tokens) to the output stream.

The skeleton program is executed in Lithium by setting up a server process on each one of the processing elements available and a task pool manager on the local machine. The remote servers are able to compute any one of the fireable MDFi in the graph. A MDF graph can be sent to the servers in such a way that they get specialized to execute only the MDFi in that graph. The local task pool manager takes care of providing a MDFi repository (the *taskpool*) hosting fireable MDFi relative to the MDF graph at hand, and to feed the remote servers with fireable MDFi to be executed.

Logically, any available input task makes a new MDF graph to be instantiated and stored into the taskpool. Then, the input task is transformed into a data flow "token" and dispatched to the proper instruction (the first one) in the new copy of the MDF graph[9]. The instruction becomes fireable and it can be dispatched to one of the remote servers for execution. The remote server computes the MDFi and delivers the result token to one or more MDFi in the taskpool. Such MDFi may (in turn) become fireable and the process is repeated until some fireable MDFi exists in the task pool. Final MDFi (i.e. those dispatching final result tokens/data to the external world) are detected an removed from the taskpool upon `evaluator.readTaskPool()` calls.

Actually, only fireable MDFi are stored in the taskpool. The remote servers know the executing MDF graph and generate fireable complete MDFi to be stored in the taskpool rather than MDF tokens to be stored in already existing, non fireable, MDFi.

Remote servers are implemented as Java RMI servers. A remote server implements a `LithiumInterface`. This interface defines two main methods: a `TaskItem[] execute(TaskItem[] task)` method, actually computing a fireable MDFi, and a `void setRemoteWorker(Vector SkeletonList)` method, used to specialize the remote server with the MDF graph currently being executed[10]. RMI implementation has been claimed to demonstrate poor efficiency in the past [136] but recent improvements in JDK allowed us to achieve good efficiency and absolute performance in the execution of skeleton programs, as shown in Section 2.3.5. Remote RMI servers must be set up either by hand (via some `ssh hostname java Server &` command)

[9]Different instances of MDF graph are distinguished by a progressive task identifier.

[10]Therefore allowing the server to be run as daemon, serving the execution of different programs at different times.

or by using proper Perl scripts provided by the Lithium environment.

In the local task pool manager a thread is forked for each one of the remote hosts used to compute the skeleton program. Such thread obtains a local reference to a remote RMI server, first; then issues a `setRemoteWorker` remote method call in order to communicate to the server the MDF graph currently being executed, and eventually enters a loop. In the loop body the thread fetches a fireable instruction from the taskpool[11], asks the remote server to compute the MDFi by issuing a remote `execute` method call and deposits the result in the task pool (see Figure 2.14).

The MDF graph obtained from the `JSkeletons` object is used in the `evaluator.setProgram()` call can be processed unchanged or a set of optimization rules can be used to transform the MDF graph[12] methods of the `Ske` class. Such optimization rules implement the "normal form" concept outlined in Sec. 2.3.2.

As the skeleton program is provided by the programmer as a single (possibly nested) `JSkeletons` object, Java reflection features are used to derive the MDF graph out of it. In particular, reflection and `instanceOf` operators are used to understand the type of the skeleton (as well as the type of the nested skeletons). Furthermore, the `Object[] getSkeletonInfo private` method of the `JSkeletons` abstract class is used to gather the skeleton parameters (e.g., its "body" skeleton). Such method is implemented as a simple `return(null)` statement in the `JSkeletons` abstract class and it is overwritten by each subclass (i.e., by the classes `Farm`, `Pipeline`, etc.) in such a way that it returns in an `Object` vector all the relevant skeleton parameters. These parameters can therefore be inspected by the code building the MDF graph. Without reflection much more info must be supplied by the programmer when defining skeleton nestings in the application code [75].

2.3.5 Experiments

In order to assess Lithium performance, we performed a set of experiments on a Beowulf class Linux cluster operated at our department, as well as on a set of "production" workstations available at our department.

The cluster based experiments were aimed at demonstrating Lithium performance features, mainly. The cluster used for the experiments hosts 17 nodes: one node (`backus.di.unipi.it`) is devoted to cluster management, code development and user interface. The other 16 nodes (ten 266Mhz Pentium II and six 400Mhz Celeron nodes) are exclusively devoted to parallel program execution. All the nodes are interconnected by a (private, dedicated) switched Fast Ethernet network. `backus` is a dual hosted node and provides access to the rest of the cluster node from Internet hosts. All the experiments have been performed using Blackdown JDK ports version 1.2.2 and 1.3.

[11] Using proper `TaskPool synchronized` methods.

[12] Using the `setOptimizations()` and `resetOptimizations()`.

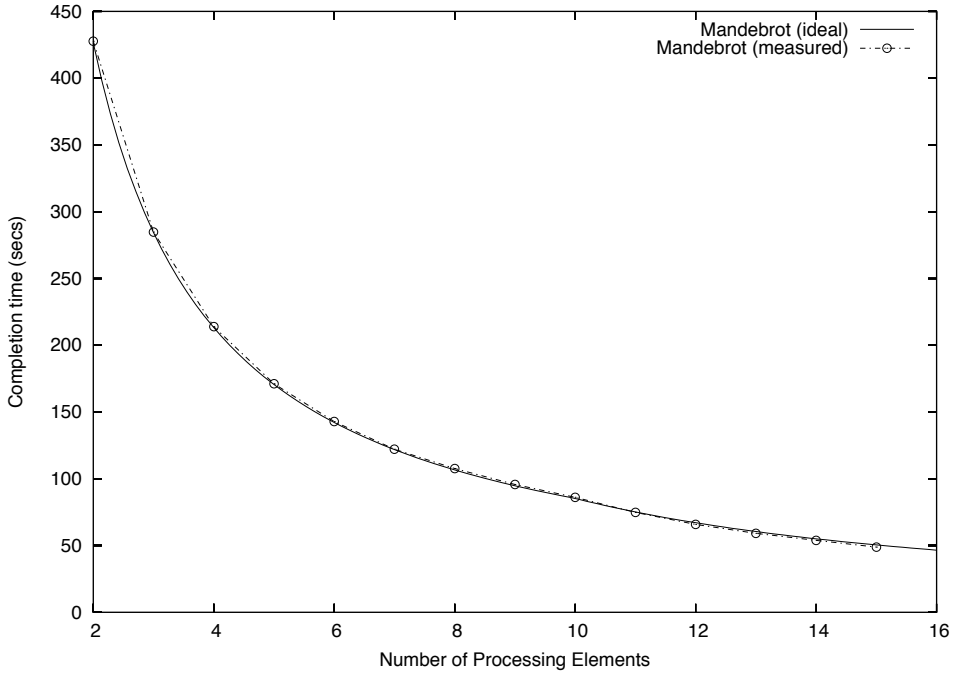


Figure 2.15: Mandelbrot application: ideal vs. measured completion time.

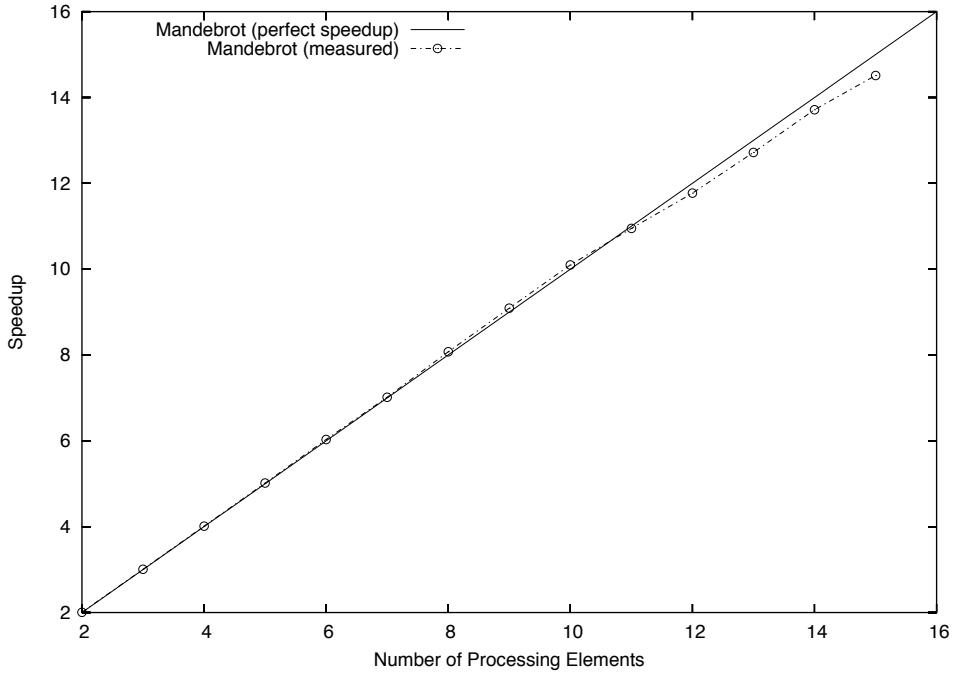


Figure 2.16: Mandelbrot application: ideal vs. measured speedup.

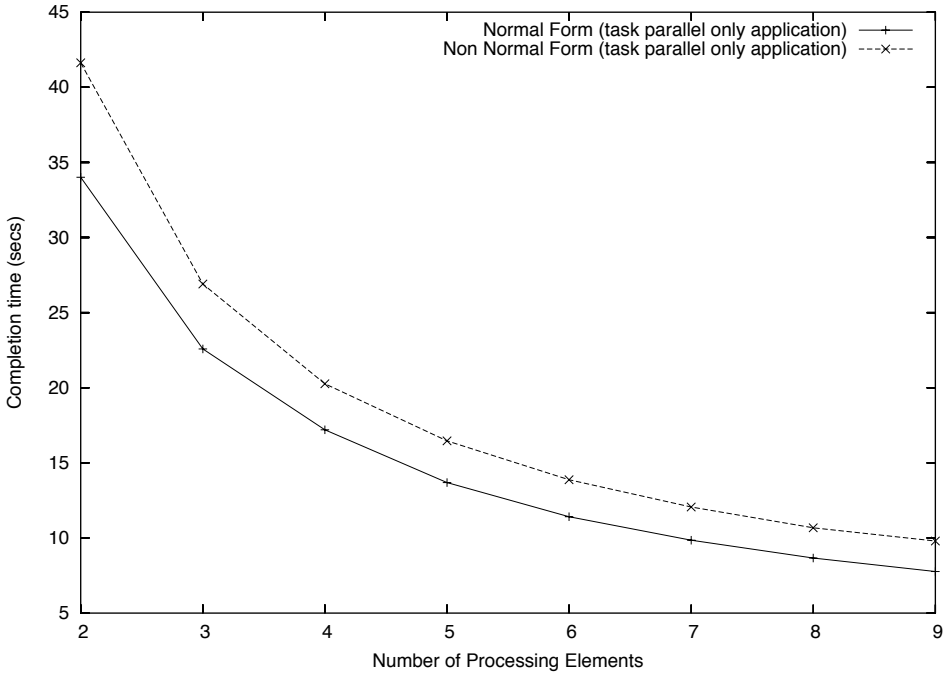


Figure 2.17: "Synthetic" task parallel application: Normal vs. non normal form completion times.

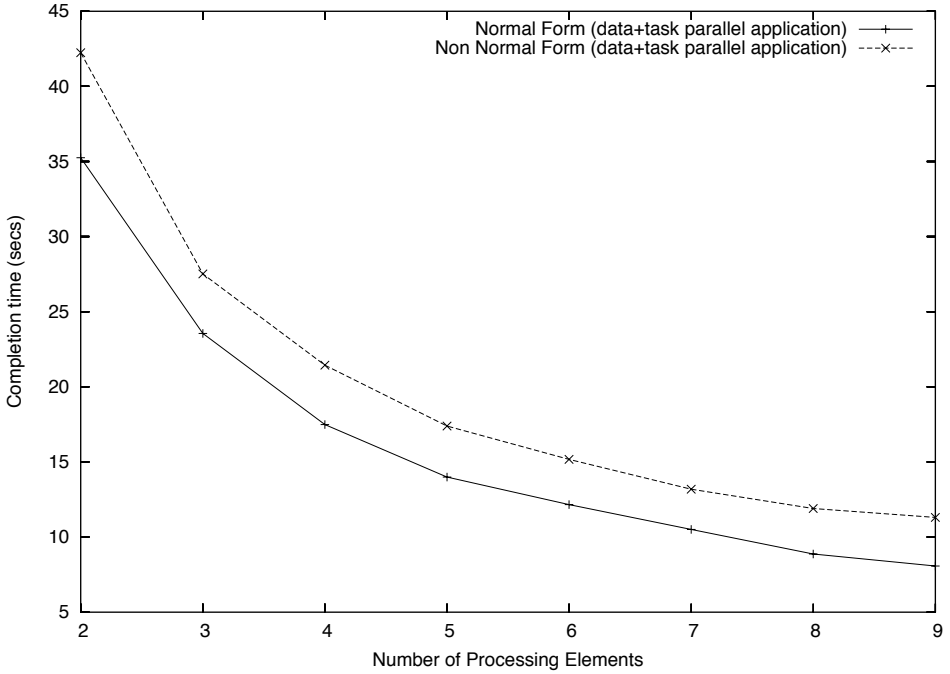


Figure 2.18: "Synthetic" task+data parallel application: Normal vs. non normal form completion times.

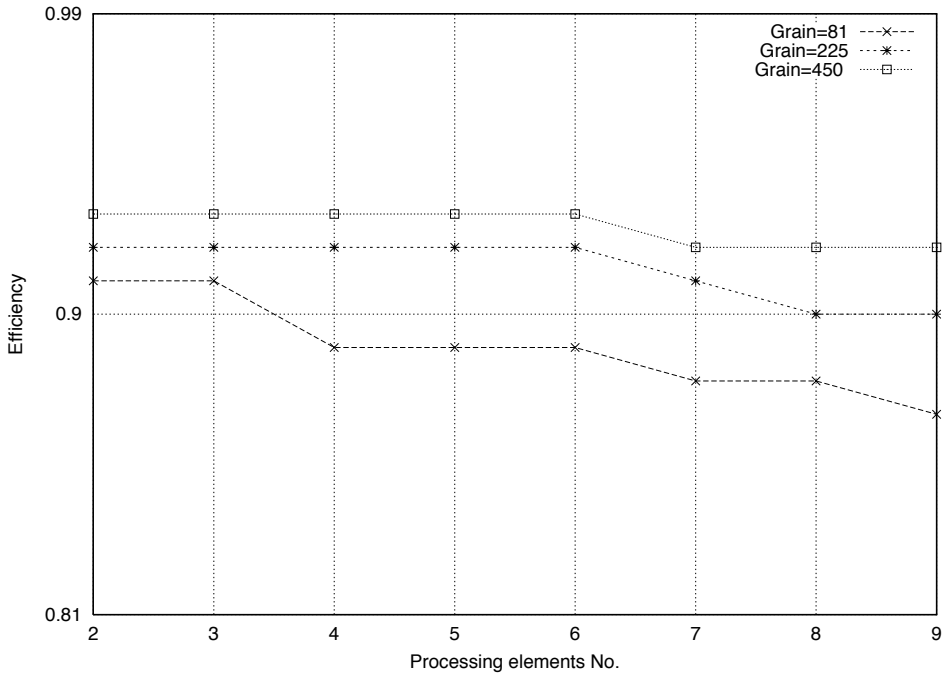


Figure 2.19: Effect of grain on efficiency.

First of all we considered the overhead introduced by serialization. As data flow tokens are dispatched to remote executor processes exploiting plain Java serialization, and as we use Java `Vector` objects to hold tokens, we measured the size overhead of the `Vector` class. The experiments showed that serialization does not add significant amounts of data to the real user data (less than 10%) and therefore serialization does not cause significant additional communication overhead.

Second, we measured the Lithium applications absolute completion time and speedup. Typical results are drawn in Figure 2.15 and 2.16. The plot shows that Lithium support scales. The graph actually refers to a skeleton version of a Mandelbrot benchmark, but we achieved similar results also with simple numerical applications. The completion times (ideal and measured) show an additional decrement from 10 nodes on, as the 11th to 16th nodes are more powerful that the first 10 nodes and therefore take a shorter time to execute sequential portions of Java code.

Third, we measured the impact of normal form optimizations, exploiting the possibility provided by Lithium of asking the execution of either the original program or the (automatically derived) normal form one. Figures 2.17 and 2.18 plots the differences in the completion time of different applications executed using normal and non normal form. As expected normal form always performs better that non normal form. The good news are that it performs significantly better and *scales* better (non normal form programs stops scaling before normal form ones).

In addition, we measured the effect of computational grain of MDFi on effi-

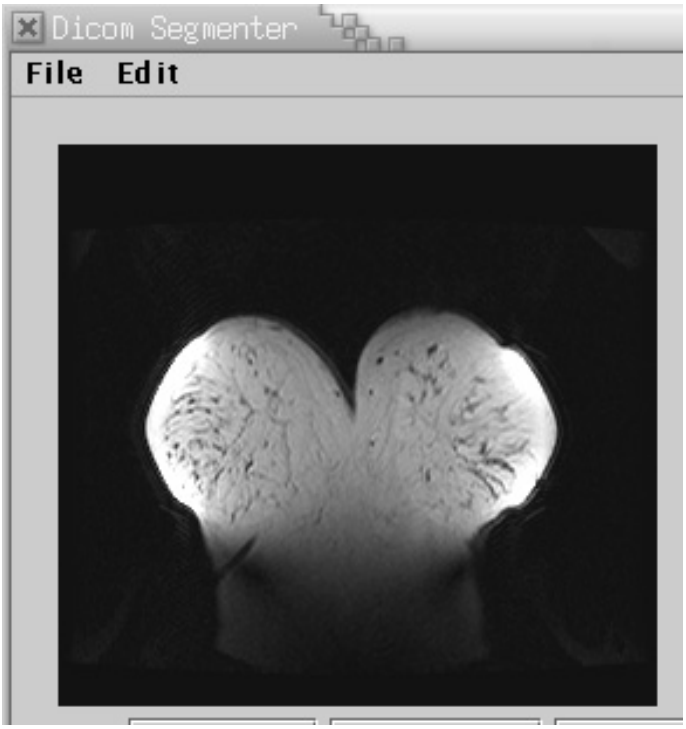


Figure 2.20: Medical image segmentation application: screen snapshot.

ciency[13]. Computational grain represents the average computational grain of MDFi. $grain = k$ means that the time spent in the computation of MDFi is k times the time spent in delivering such instructions to the remote servers plus the time spent in gathering results of MDFi execution from the remote servers (via plain Java RMI). Experimental results showed that efficiency is always more that 90% when grain is higher that 100 (roughly). When average grain is under 100 efficiency falls under 90% already when 3 processing elements are used for program execution and continues to decrease rapidly as more and more processing elements are used.

We also performed experiments using a real application. We considered a medical application rendering mammography images. Images come from patient analysis. First, a set of images is taken, each representing a breast slice, more and more distant from patient ribs. Then some series of similar images are taken after the ignition of a contrast liquid. Overall a single, complete examen consists of about one hundred images. Each image must be properly segmented in order to highlight the interest zones (i.e. zones were cancer may be discovered). Figure 2.20 shows one of such images in a video snapshot taken from our Java segmentation application.

[13]As usual, we define efficiency (ϵ) as $\epsilon = \frac{T_{seq}}{n \times T_{par}(n)}$, where T_{seq} represents the sequential execution time, n is the parallelism degree and $T_{par}(n)$ is the time spent in the execution of the parallel program with parallelism degree n.

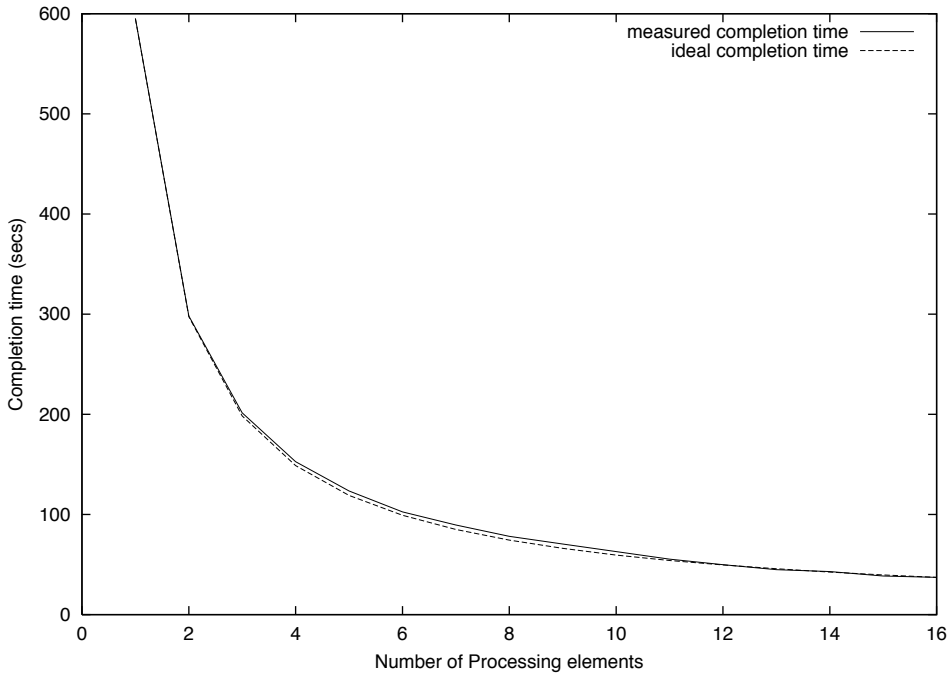


Figure 2.21: Results with the medical image segmentation application.

The rendering of an image set takes about 10 minutes on a 266Mhz Pentium II Linux box. Figure 2.21 shows the times achieved using Lithium on our cluster. The application perfectly scales[14]. Efficiency is constantly over 90% in this case. Figure 2.22 plots efficiency of segmentation application. Superscalar efficiency in the right part of the plot is due to the fact that sequential times are taken onto the Pentium processors (processors from 1 to 10) and processors 11 to 16 happen to be faster.

With the medical image segmentation code we also performed experiments on our Department production workstations, in order to assess the load balancing policies of the macro data flow execution mechanism. The production workstations used range from 233 Mhz PentiumII Linux boxes to dual 450 Mhz Pentium III and 1.6Ghz Pentium IV Linux workstations. All the machines are interconnected by means of a plain 10Mbit (not switched) Ethernet network that happens to be very busy all the time.

The first result is that using faster machines Lithium achieves better completion times. The processing of one hundred images took about 1.71 minutes on our 266 Mhz Pentium II cluster on 6 PEs, while using 6 faster production workstations (three 1.2 Ghz Pentium IV, two 450 Mhz Pentium III and a single 233 Mhz Pentium II machine) the same processing took only 0.85 minutes.

The second result concerns load balancing. Table 2.23 shows the number of

[14]The data is relative to normal form execution. Plain application is a farm with three stage workers.

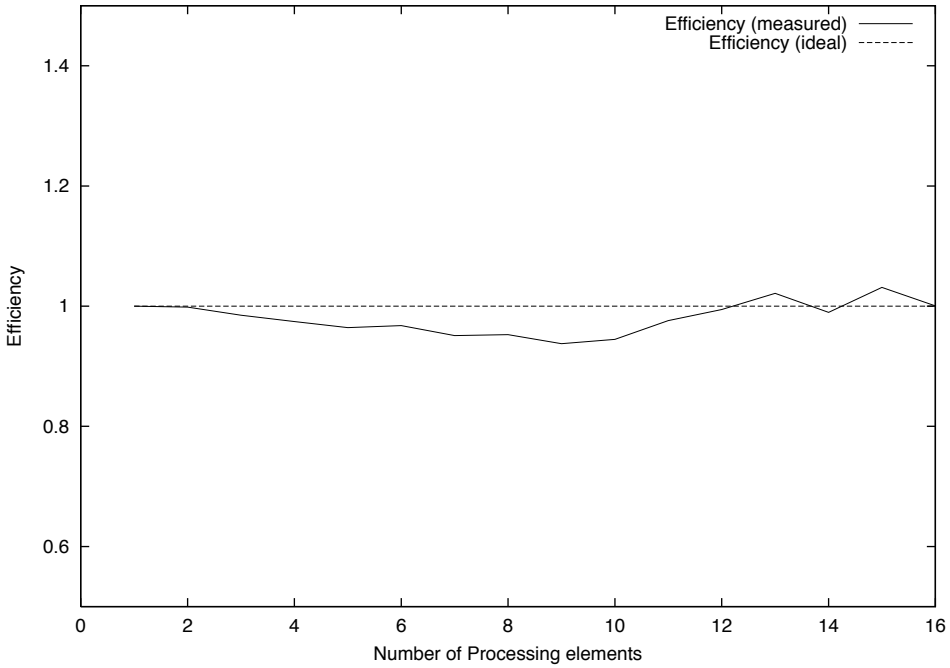


Figure 2.22: Efficiency of medical image segmentation application (The efficiency is figured out with respect to the sequential execution time computed on the slower processors (PE $\in [1, 10]$)).

task executed by each workstation along with workstation bogomips (the rough performance measure taken by Linux kernel at boot time) and load (measured with `uptime` during the experiments). It is clear that load balancing has been achieved in that slower, more loaded workstation participated in the computation by computing a smaller number of tasks with respect to faster, unloaded workstations.

		WS_1	WS_2	WS_3	WS_4	WS_5	WS_6
Bogomips		3204	3630	901	3204	466	897
6 PE	load	1.44	0.75	1.90	0.24	0.39	0.34
	tasks#	20	29	5	26	7	13
4 PE	load	1.44	0.88	0.46	0.37	–	–
	tasks#	24	36	6	34	–	–

Figure 2.23: Load balancing on heterogeneous processing elements (100 tasks).

```
import lithium.*;

public class Emitter extends JFrame {
    ...
    public static void main(String [] args) {
        ...
        File dir = new File(args[0]);
        String[] files = dir.list();
        ...
        JSkeletons stage1 = new DicomToImage();
        JSkeletons stage2 = new Segmenter();
        JSkeletons stage3 = new PostProcessing();
        JSkeletons worker = new Pipe();
        worker.addWorker(stage1);
        worker.addWorker(stage2);
        worker.addWorker(stage3);
        Farm theMain = new Farm(worker);
        Ske eval = new Ske();
        String[] hosts = {"izar","icaro","quanto",
                          "cassiopea","montecristo"};
        eval.addHosts(hosts);
        eval.setOptimizations();
        eval.skelProgram(theMain);
        for(int f=0; f<taskNo; f++) {
            ...
            b = f(DD.getImageToDisplay(files[f]));
            try {
                eval.setupTaskPool((Object)(b));
            } catch (Exception e) {
                System.out.println("TP setup: "+e);
                return;
            }
        }
        eval.stopStream();
        try {
            eval.parDo();
        } catch (Exception e) {
            System.out.println("Start: "+e);
            return;
        }
        while(!ske.isResEmpty()) {
          try {
              res = (byte[]) eval.readTaskPool();
          } catch (Exception e) {
              System.out.println("Result fetch: "+e);
              return;
          }
          ...
          e.doImage(g(res),w,h);  // display image
        }
        return;
    }

}
```

Figure 2.24: Medical image processing application skeleton.

Conclusions

We described a new Java parallel programming environment providing the programmer with simple tools suitable to develop efficient parallel programs on workstation networks/clusters. Lithium is the first skeleton based parallel programming environment written in Java and implementing skeleton parallel execution by using macro data flow techniques. We performed experiments that demonstrate that good scalability and efficiency figures can be achieved. Lithium it is currently available as open source [62].

Despite the large number of projects aimed at providing parallel programming environments based on Java, there is no existing project concerning skeletons but the $\mathbf{CO_2P_3S}$ one [132, 131]. Actually this project derives from the design pattern experience [88]. The users are provided with a graphic interface where they can combine different, predefined parallel computation patterns in order to design structured parallel applications that can be run on any parallel/distributed Java platform. In addition, the graphic interface can be used to enter the sequential portions of Java code needed to complete the patterns.

The overall environment is layered in such a way that the user designs the parallel application using the patterns, then those patterns are implemented exploiting a layered implementation framework. The framework gradually exposes features of the implementation code thus allowing the programmer to perform fine performance tuning of the resulting parallel application. The whole object adopts a quite different approach with respect to our one, especially in that it does not use any kind of macro data flow technique in the implementation framework. Instead, parallel patterns are implemented by process network templates directly coded in the implementation framework. However, the final result is essentially the same: the user is provided with a high level parallel programming environment that can be used to derive high performance parallel Java code running on parallel/distributed machines.

Macro data flow implementation techniques have also been used to implement skeleton based parallel programming environments by Serot in the SKiPPER project [154, 153]. SKiPPER is an environment supporting skeleton based, parallel image processing application development. The techniques used to implement SKiPPER are derived from the same results we started with to design Lithium, although used within a different programming environment (the whole SKiPPER environment is written using OCaml, the ML implementation from INRIA).

2.4 The ASSIST programming environment

ASSIST (A Software development System based upon Integrated Skeleton Technology) is a new programming environment oriented to the development of parallel and distributed high performance applications according to a unified approach. ASSIST has been designed and developed at University of Pisa[15], and actually is our group latest proposal in the area of structured parallel programming (see also Section 2.1).

ASSIST is a pretty complex environment framed in a currently ongoing project, although a running version already exist. ASSIST is actually the environment where eskimo class languages will live; at the least we expect to exploit eskimo design lessons learned in further releases of ASSIST. However, we believe that both eskimo design experiment and ASSIST are also meaningful per se. In the remaining of this section we shall sketch ASSIST main features, reminding to the literature for any further detail [169, 66, 13, 12, 11]. eskimo language is presented in Chapter 4.

2.4.1 Motivations and main goals

According to our previous experience in structured parallel programming, in ASSIST we wish to overcome some limitations of the classical skeletons approach to improve generality and flexibility, expressive power and efficiency for irregular, dynamic and interactive applications, as well as for complex combinations of task and data parallelism. A new paradigm, called "parallel module" (`parmod`), is defined which, in addition to expressing the semantics of several skeletons as particular cases, is able to express more general parallel and distributed program structures, including both data-flow and nondeterministic reactive computations. ASSIST allows the programmer to design the applications in the form of generic graphs of parallel components. Another distinguishing feature is that ASSIST modules are able to utilize external objects, including shared data structures and abstract objects (e.g. external objects accessed via CORBA), with standard interfacing mechanisms. In turn, an ASSIST application can be reused and exported as a component for other applications, possibly expressed in different formalisms.

Motivations. The motivations for our research on new programming environments for high-performance applications are mainly related to the requirements for:

1. a high-performance software component technology,
2. the development of high-performance applications on emerging Large-scale Platforms and Grids,
3. overcoming the limitations of structured parallel programming beyond the "classical" skeletons model.

[15]With the support of the *Italian Space Agency* (ASI-PQE2000 project) and *National Research Council* (Agenzia 2000 CNR project).

Software component technology is playing a central role in the development of complex and multidisciplinary applications, especially in distributed and loosely coupled systems. The main, strongly interrelated, goals are portability across different hardware-software platforms, reuse of existing components to create different more complex systems, easy evolution through specific versions of the application. These goals are fundamental for the high-performance computing field too, both in scientific and in industrial applications. Some interesting projects [89, 6, 25] are focused on the issue of characterizing the component technology for high-performance computing applications, and this is also one of the main goals of our research: a new programming environment oriented to the development of parallel and distributed high-performance applications according to a unified approach, that matches the features of component technology and the features of structured parallel programming technology.

Beyond the "classical" skeleton approach. Despite several advantages of skeletons, a strong evolution of structured parallel programming beyond such models is needed, at least for the following reasons:

a) in addition to the capability of expressing some typical parallel schemes, we need a larger degree of flexibility in expressing parallel and distributed program structures. In general generic graph structures are required for complex compositions in multidisciplinary applications;

b) essentially, skeleton-based programming models have functional and deterministic semantics that can be a serious obstacle in many complex applications. The concepts of internal state of parallel components and of nondeterminism in communications/interactions between components are fundamental, as well as dynamic interactivity in a client-server or in a peer-to-peer environment. In general, all such features are not stressed in skeletons models;

c) though in many skeletons programs the integration of task and data parallelism [33] can be expressed, there are many cases in which the composition of the available skeletons is not natural or it is inefficient. In our experience with SkIE, this occurs in some irregular and/or dynamic problems for which the integration of (task) stream parallelism and data parallelism is a source of inefficiency and produces codes that are more complex than the equivalent codes exploiting only data parallelism;

d) to develop a complex, multidisciplinary application we need to be able to utilize predefined objects in a modular and invisible way: they can be abstract objects according to commercial standards (e.g. CORBA), as well as abstractions of systems resources (devices, file, severs, and so on) and several kinds of libraries (scientific, image processing, data mining);

e) in many applications the adoption of a shared space of objects, or a Distributed Shared Memory space (independent of the distributed nature of the underlying platform), is fundamental to efficiently manipulate very large data sets, to simplify the programming of irregular and/or dynamic problems and, sometimes, to mask communication overheads. In our experience [63] the integration of shared objects into a structured parallel programming formalism increases the expressive power and the efficiency of parallel programs significantly;

f) at our knowledge, the skeletons model is not suitable to fully support the reuse of parallel applications written in different formalisms. This goal has to be achieved in the new context of parallel components.

One of the basic features of structured parallel programming that we wish to preserve is related to the efficiency of implementation, and of the run-time support in particular. On the one hand, compared to "classical" skeletons, it is more difficult to define a simple cost model for a generic construct like `parmod`, and this can render optimizations at compile time more difficult. On the other hand, at least with the current knowledge of software development for Large-scale platforms/Grids, we believe that run-time support optimizations are much more important than compile-time optimizations (which, anyway, remain a significant research issue).

2.4.2 Features of ASSIST

ASSIST is a new programming environment for parallel and distributed high-performance applications. The design of ASSIST applications is done by means of a coordination language, called ASSIST-CL.

Parallel/distributed programs can be expressed by generic graphs, without renouncing the possibility of structuring specific (skeletons-like) graphs for which some interesting cost models already exist [139]. The components can be parallel modules or sequential modules, with high flexibility of replacing components in order to modify the degree of granularity according to the application requirements of and/or to the underlying platform evolution. ASSIST introduces a construct, which is more flexible and powerful than the "classical" skeletons. It could be considered a sort of "generic" skeleton which can be programmed to emulate the most common "specific" skeletons, but which is also able to easily express new forms of parallelism (e.g. optimized forms of task + data parallelism, nondeterminism, interactivity), as well as the invariants and personalizations. When necessary, the parallelism forms that are expressed could beat a lower abstraction level in respect of the "classical" skeletons. We call this new construct parallel module, or `parmod`.

Generality and flexibility do not mean to abandon the idea of structured parallel programming. On the contrary, the proposal of a "generic" skeleton and of applications as graphs of parallel and sequential modules aims to reinforce such idea and, at the same time, to achieve a more satisfactory trade-off between high-level structuring, expressive power, reuse and efficiency.

The composition of parallel and sequential modules is expressed, primarily, by means of the very general mechanisms of streams, by which we can represent powerful interfaces simply and effectively. In addition, modules can share objects implemented by forms of Distributed Shared Memory, invoked through their original APIs or methods. While the stream-based composition is the basic mechanism for structuring applications and defining the component interfaces, shared memory objects are an additional mechanism for solving problems of memory space, programmability of highly dynamic structures, and communication of heavy datasets. Moreover, the parallel and the sequential modules have an internal state.

It is worth noticing that the modules of a parallel application can refer to any kind of existing external objects, like CORBA and other commercial standards objects, as well to system objects, though their interfaces. In the same way, an ASSIST parallel application can be exported as a component or as an object to other applications: the stream-based interface mechanisms is transformed at compile-time in a mechanism compliant to existing standards, i.e. CORBA IDL interfaces in the first version of ASSIST.

Finally, the methodology by which the environment is implemented is itself component-based and object-based, in order to be able to flexibly modify its implementation according to significant changes in the underlying technologies and/or in applications requirements [11].

2.4.3 Structure of ASSIST programs

The structure of an ASSIST program is a graph, whose nodes are components and the arcs are abstract interfaces that support streams, i.e. ordered sequences, possibly of unlimited length, of typed values. Any graph structure, possibly including cycles and merge (many-to-one) and multicast (one-to-many) streams, is permitted. Streams are the structured way to compose modules into an application. In addition, components can also interact by means of external (shared) objects, in general not expressed in ASSIST-CL. Components are expressed by ASSIST modules, which may be parallel modules (`parmod`, defined in the next section) or sequential modules. A sequential module is the simplest component expressed in ASSIST: it has an internal state and is activated by the input stream values according to a deterministic data-flow behavior (the nondeterministic behavior can be expressed only by parallel modules). Figure 2.25 shows the example graph of a simple program.
Modules *M1* and *M2* have no input stream and, in fact, are in charge of generating streams $S13, S23$ and $S24$. *M2* has two output streams $S23$ and $S24$; in general, during an activation, *M2* may send values onto both $S23$ and $S24$, or only onto one of them, or none of them. *M3* may have a deterministic or, if it is a parallel module, a nondeterministic behavior on the input streams $S23$. *M4* and *M5* have no output stream, thus are in charge of generating the result data structures of the program. A composition of modules, expressed by a graph $\mathcal{P}$, may be, in turn, reused as a component of a more complex program $\mathcal{Q}$. The composition is legal provided that $\mathcal{P}$

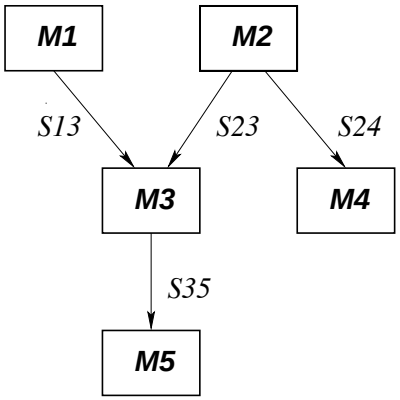


Figure 2.25: An ASSIST graph.

is correctly interfaced to the other modules of $\mathcal{Q}$, i.e. the types of input and output streams must be compatible.

2.4.4 Parallel module

The construct called parallel module (`parmod`) is the proposed solution for many parallel programming problems introduced in the previous section. The idea of `parmod` is shown graphically in Figure 2.26

A `parmod` is defined by the following elements: 1) Virtual processors; 2) Topology (naming scheme); 3) Internal state; 4) Input streams; 5) Output streams; 6) External objects.

A set of virtual processors (VPs), i.e. independent and cooperating entities delegated to perform the parallel computation, is defined as the "calculation engine" of a `parmod`. The support tools will map the set of VPs onto a suitable set of physical processing nodes, statically or dynamically.

Each VP has a unique identifier, which often is usefully expressed in a parametric way in order to implement collective forms of parallelism, e.g. data-parallelism. As a consequence, the naming scheme is often related to the way in which the internal state is assigned and distributed to VPs and to the way in which it is referred. The topology is in no way related to the cooperation scheme, or the communication stencil, of VPs. Currently ASSIST supports the following topologies:

multidimensional array to denote every VPs by the values of one or more indexes;

none to denote the name of VPs is not significant for the computation to be expressed, e.g. in several farm-like structures composed of fully independent workers;

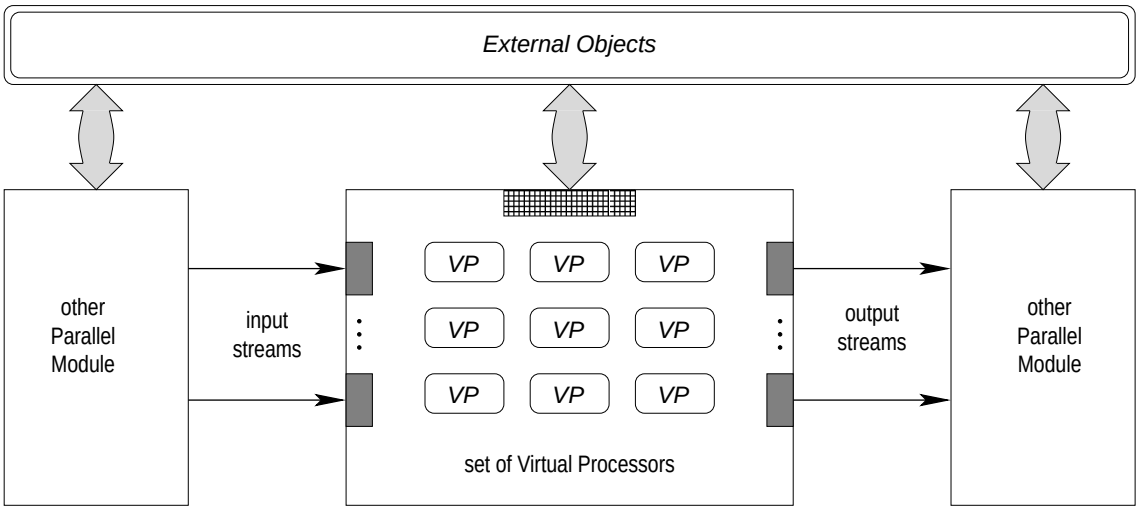


Figure 2.26: Graphical scheme of a Parallel Module.

one to denote that the `parmod` has only one VP. This is different from the sequential module, because a *parmod-one* exploits many of the general features of a `parmod`, in particular nondeterminism and stream control features.

Notice that more than *one* topology is possible for the same problem, according to the parallelization strategy and/or the granularity that the programmer wishes to express. The component-based structuring of an ASSIST program allows the designer to change easily the internal implementation, e.g. the granularity, according to new requirements. As a special case, the topology "one" is adopted when the programmer wishes to force a sequential (though nondeterministic) execution, being aware that a parallel implementation could be inefficient on the currently available platform; however, such a component is predisposed to be modified into a parallel one when the platform will be changed and/or more efficient parallelization strategies will be possibly defined.

Streams are the basic mechanism for the composition and interfacing of component modules, they may be thought as "channels" linking parallel modules one each others. The input streams may be used according to a data-flow scheme when, for each activation, the module waits for values from all the input streams. In the most general, case a subset of streams is selected at each activation: this expresses a nondeterministic behavior. The semantics of nondeterminism is the one of the CSP model [105] based on guarded commands, and in particular the semantics of ECSP concurrent language [32]. Data items flowing from input streams may be sorted into Virtual Processors according to several distribution strategies, namely *on demand, scatter, multicast*, and *broadcast*. ASSIST behaves as follows in respect of distribution strategies:

On demand: a received value is transmitted to a ready VP, chosen nondeterministically;

scatter: a received, structured value is partitioned to the VPs according to a rule expressed by program,

multicast: a received value is sent to all the VPs belonging to a certain subset expressed by program,

broadcast: a received value is sent to all VPs.

Similarly, the results of an activation may be sent onto one or more output streams, or not sent at all. This choice is controlled explicitly by program.

A `parmod` has a internal state that is logically partitioned and/or replicated into VPs. Some state variables will be also utilized to control the communication from the input streams and to the output streams.

A module (sequential or parallel) of an ASSIST program can refer to external objects according to the interfaces/methods or APIs of such object. As seen, this is a mechanism to exploit (import) the functionalities of possibly existing objects defined outside the application. External objects are also a mechanism to cooperate with other modules of the same application, in addition to the stream mechanism. External objects helps to overcome the limitations of the single node memory capacity in distributed architectures and make the management of dynamic and/or irregular program/data structures easier. Moreover, they provide a powerful mechanism to import/export abstract objects in commercial standards. Three kinds of external objects are distinguished at the moment:

Shared variables. A first kind of external object is defined just in terms of the same data types of ASSIST-CL. Any variable can be defined as shared by modules of an ASSIST application. This can be interpreted as an extension of the concept of internal state of modules: now the state can also be shared between distinct modules.

Distributed Shared Memory libraries. In many problems, the goals c), d), e) mentioned above can be met by means of external objects expressed by abstractions of shared memory. In particular, we consider the integration of libraries for both (software) DSM and abstract objects implemented on top of them (e.g. spread trees, see Chapter 5). While on shared variables we can only execute the operations corresponding to the ASSIST-CL types, on the shared memory objects the proper set of operations is defined for expressing powerful strategies of allocation, manipulation and synchronization. Currently, we are using DVSA [31] and SHared OBjects [48] libraries, all such tools having been developed at the Computer Science Department of Pisa in previous research projects. However, any existing library (e.g. JIAJIA [107], DSM-PM2 [19]) or imagined (eskimo indeed, see Chapter 5) can be integrated and utilized by

the ASSIST. Of course, it is responsibility of the programmer to correctly utilize the library according to its semantics (interfaces, consistency model, data types).

Remote Objects. The utilization of remote objects through CORBA (in the next versions of ASSIST, other commercial standards) is done according to modalities similar to the ones described for DSM libraries. ASSIST-CL defines an ORB interface containing APIs to connect and utilize remote objects in ASSIST applications. No constraints are imposed on the access and utilization of a CORBA object by an ASSIST program, except that the registration modality of the external object must be verified according to the implementation of the object itself [12].

Chapter 3

DSM: the state of the art

Readers' road-map. In this chapter we present a survey of distributed shared memory architectures, that are part of the framework of the thesis. The core of the discussion is reached through a brief review of DSM basic concepts, namely cache coherence and memory consistency. In Section 3.1.3 DSMs are characterized accordingly several functional aspects: implementation level, consistency model, and behavior with respect data replication. Those aspect are discussed in Sections 3.2, 3.3 and 3.4 respectively. *Cilk* main features are also sketched within consistency models section. In Section 3.5 some technical issues related to software implemented DSMs are presented. Eventually, the *Athapascan* language is presented in Section 3.6.

In order to satisfy the ever increasing demands of typical applications, a large progress was made in the research and development of systems with multiple processors capable of delivering high computing power. Distributed Shared Memory (DSM) systems tries to combine the advantage of two classes of systems: multiprocessors, having a single global physical memory (equally accessible to all processors) and multicomputers consisting of multiple processing nodes communicating by means of message passing. A DSM system logically implements the shared model on a physically distributed memory system.

3.1 Basic concepts

Realizing the shared address space abstraction basically requires a two-way request-response protocol. A global address is decomposed into a module number and a local address. For a read operation, a request is sent to the designated module requesting a load of the desired address and specifying enough information to allow the result to be returned to the requester through a response network transaction. A write is similar, except that data is conveyed with the address and command to the designated module, and the response is merely an acknowledgment to the requester that the write has been performed. In this case the response essentially informs the

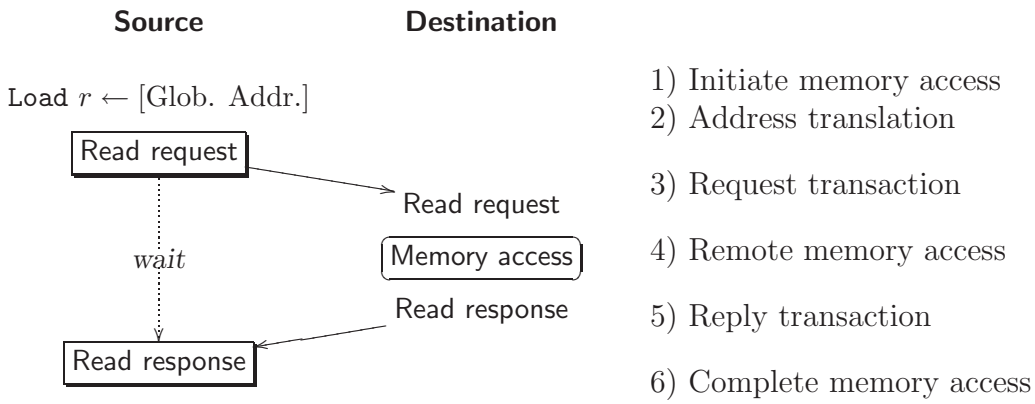


Figure 3.1: Shared address space abstraction:two-way request-response protocol.

source that the request has been serviced (see Figure 3.1).

In shared memory multiprocessor, reading and writing shared variables by different processors is expected to be a frequent event, since it is the way used by multiple processes (or threads) belonging to a parallel application to communicate with each other. Therefore, we want to allow caching of both private and shared data. As far as shared data concern, we assume to deal with *cache coherent*[1] shared memory multiprocessors. Coherence defines *what* value has to be assumed by a memory location (Section 3.1.1).

In order to formally define the memory system behavior we need also to define the *consistency* model, i.e. *when* a written value will be returned by a read.

Observe that the consistency model is orthogonal to the coherence model. Coherence defines the behavior of reads and writes only to the *same* memory location, all copies of a memory location (sooner or later) will be committed, afterwards all processors will observe a coherent view of the memory. Consistency defines in *what temporal order* two memory operations, both from a single thread and from different threads, will have their effect with respect to memory locations. For instance, all the following are consistency issues: In what order two memory operations, to different locations and from different threads, must complete? In what order two memory operations, to different locations and from the same thread, must complete? Is it possible that a read issued from a given processor complete while another processor is committing the same location? As we shall see, several answers are possible.

In the next two sections we shall discuss about cache coherence memory consistency models in very general terms.

[1]In this meaning we refer to "cache" as functional concept, i.e. a device holding a data copy, not necessarily a specialized hardware device or a particular type of memory.

3.1.1 Cache coherence

In sequential systems, following the intuition, memory provides a set of locations that holds values, and when a location is read it should return the latest value written in that location. We rely on such property writing sequential programs, and we expect the memory to behave in the same way in multiprocessor shared address space: a read would return the latest value written to the location regardless of which process wrote it. In sequential systems, caching does not change the behavior since all processes see the memory through the same cache/memory hierarchy. In multiprocessor systems, when two processes see the shared memory through different caches, a risk exists that one may see the new value in its cache while the others still see the old value. Cache coherence can be provided using both hardware and software techniques. Snooping and directory-based cache coherence are the most known techniques, we refer back to the literature in the field for any further detail [65, 100]. For our purpose is enough to know that coherence defines *what* value has to be assumed by a memory location.

3.1.2 Memory consistency

Often in writing a parallel program, we want to ensure that a read returns the value of a particular write; i.e. we want to establish an order between a write and a read. Typically, we use some form of event synchronization to convey this dependence, e.g. locks and barriers. Surrounding accesses to shared data with a pair of synchronization operations (lock-unlock) protects shared data from accesses by other threads, ensuring an order among different threads.

To put other irons in the fire, observe that we have said nothing yet about the order of memory operations from the same thread. (We are implicitly assuming that they are completed in program order!) Suppose the memory operation are generated where it naturally occurs in the program. The processor may be allowed to proceed past it and find other independent computation or memory access that would come later in the same thread of execution; in fact, making *non-blocking* (long latency) memory accesses. The subject is a key issue in design of a hardware level shared address space

For our purpose is more interesting to reason about the order of execution of memory operations belonging to different threads (and possibly different processors), assuming a safe order among memory operations belonging to the same thread has been guaranteed.

Anyway, since processors communicate through shared variables (both those for data values and those used for synchronization), it is of prime importance to establish the order in which a processor must observe the data writes of other processors, i.e. the *memory consistency model*.

The question is straightforward if there is just one data copy in the whole system and all memory operations are atomic: we expect from a memory system to return

"the last value written" for each location. The question becomes more complex if the system uses some kind of data replication (e.g. caches or local data copies) and an independent two-way transaction memory protocol, just because it is a bit more difficult to establish what is "the last value written". It is certainly possible to ensure the same behavior requiring that processors delay all memory accesses until the current one has been completed and all data copies has been kept coherent. But is this a good consistency model?

It depends on the viewpoint. The model we have implicitly defined is called *sequential consistency*, and is surely easy to understand. As we shall see, is neither the only one, nor the more efficient. In particular we shall see how, relaxing sequential consistency constraints, it is possible to address two major classes of performance optimizations in DSM systems:

- Relaxed memory consistency models enable the *hiding of remote memory access latency* in shared memory cache-coherent multiprocessor. The idea is hiding memory latency by finding something else useful for the processor to do, while remote memory accesses are ongoing. The memory operation may be generated where it naturally occurs in the program, but the processor is allowed to proceed past it and find other independent computation or memory access that would come later in the same thread of execution; i.e. long latency memory accesses are made *non-blocking*. The technique has been mainly used in hardware implementations of shared address space.

- Relaxed memory consistency models enable the *reduction of coherence oriented handshakes* in shared memory cache-coherent multiprocessor. The idea is delaying coherence oriented actions (invalidate, update) until certain synchronization points. This behavior may have a strong impact both on hardware required to realize the shared address space and on programming model.

We will consider both sequential and relaxed consistency models in Section 3.3, in particular we will examine how they work in abstract, how they may be characterized and what is their expected performance.

3.1.3 Characterizing DSMs

Since the early DSM systems developed in mid-eighties to nowadays, tens of different DSM systems have been proposed. From the first research prototypes, the interest of research community in such systems is progressively grown particularly because DSM allows the experimentation of innovative solutions (also) with low cost hardware. Actually this is one reason why we decided to use DSM as framework. In this contest, make no sense exhaustively list and discuss all DSM prototypes we have got to known. Conversely, it is important functionally classify solutions appeared in DSM technology.

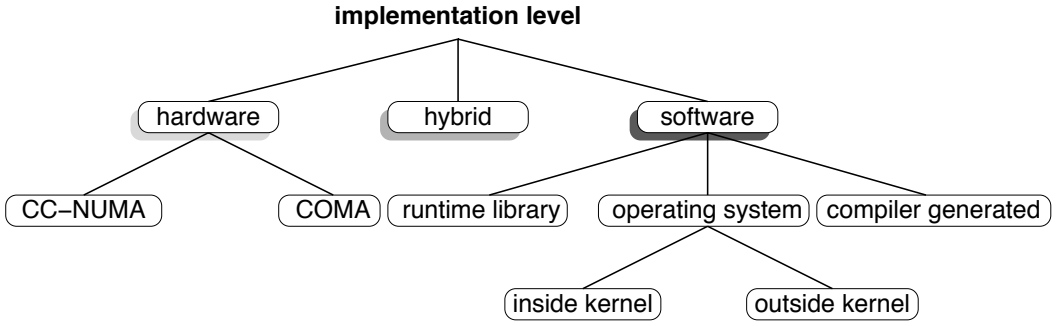


Figure 3.2: DSM implementation level taxonomy.

In the following we discuss several key functional aspects in DSM design: the implementation level, the memory consistency model, the algorithms used to provide memory coherence and eventually the programming model. Each functional aspect is introduced by means of a taxonomy and the description of the mainstream technical solutions proposed by the research community. Observe that presented functional aspect are rather orthogonal one each other. However, through the presentation, we underline the significant correlations among different functional aspects and the implication of each design choice.

3.2 Implementation level

The DSM shared address space is distributed (in some way) across different memories. Realizing a shared address space raises two basic design issues with respect to the implementation level:

1. *At which level data lookups are executed?*

2. *At which level consistency actions related to write access are executed?*

Both lookups and consistency actions can execute in software, hardware or a combination of both. Following a classification similar to that of Protić, Tomašević and Milutinović [163, 143], we refer to these class in Figure 3.2 as hardware, software and hybrid implementations.

The choice of the implementation level essentially depends on price versus performance trade-offs. Hardware implementations are in general superior in performance while require a greater design complexity respect to software solutions. High performance (and cost) machine resolve both previous questions equipping machine with hardware co-processors. Low-end systems, including Beowulf class clusters, entrust completely to software solutions to keep low the machine cost. Hybrid, mid-range systems employ low-cost additional hardware to commodity component such as “smart network interfaces”

It is worth pointing out in parallel systems a simple consideration trammel the price/performance trade-off: A parallel system is expected to be fast. In some cases, the implementation in software of a hardware device yields either unacceptable overheads in performance or too strong constrains on programming model.

3.2.1 Hardware

Hardware implemented mechanisms for shared memory guarantees automatic management of shared data in local memories, transparently for software layers. Hardware implementations efficiently supports fine grain sharing. The address space is non-structured and the unit of coherence is small (typically the cache line). The small unit of sharing and coherence characterizes hardware solutions for look-up and coherence/consistency actions. The granularity of data sharing is (in general) related with the address translation mechanism granularity, since an all software address translation may introduce unacceptable overheads. Therefore, the minimum unit of sharing often correspond to the minimum unit the hardware can discriminates memory lookups, even if some exceptions exists.

CC-NUMA *Cache Coherent Non Uniform Memory Access.* This kind of systems statically distributes the shared virtual address space across local memories of nodes. The static organization facilitates direct access to a unique memory location in one of the nodes. A processor node can access both local memory and remote memory, although with pretty different access latencies. Cache coherence frequently relies on directories. Both sequential and relaxed memory consistency models may be adopted as memory consistency model. An application running on this architecture exhibits the best performance when its working set is contained within the memory hierarchy of the node. Static data partitioning should be done carefully to maximize the frequency of local access.

COMA *Cache Only Memory Access.* This kind of systems uses the memory associated with each node as a higher level cache (attraction memory). There is no memory home location predetermined for a particular data item, and it can be replicated and migrated. They are less sensitive to static distribution of data than NUMA systems, since data distribution quickly adapt to the dynamic memory reference behavior of executing applications. Nevertheless, there are many unresolved issues associated with COMA architectures, including hardware and system support for locating and replacing data blocks.

3.2.2 Software

The last decade has exhibited the proliferation of low cost distributed systems, especially in the form of Beowulf class systems. This kind of systems are often assembled

using complete computers equipped with a standard or proprietary network interface, and they do not provide any specialized hardware for address sharing among different nodes, while they provide a native message passing programming model. Software DSMs supplies the shared memory programming model on top of message passing native mechanism. Generally, this can be achieved via operating system primitives, via user level run-time libraries, via compiler generated instrumented code, or a combination of the previous approaches (see Figure 3.2).

Run-time library The DSM mechanism is implemented via run-time library routines linked to with the application program using the (virtual) shared address space. The approach is flexible and particularly indicated for experimentation since the library code may be easily augmented and corrected.

Operating system The DSM mechanism is incorporated in the operating system on specialized software controller, that can live either inside or outside the kernel. The inside kernel solution may profit from the fine control over the scheduling and interrupt processing. The outside kernel solution sacrifices part of these advantages in the name of portability over different kernels or different evolutions of the same kernel.

Compiler generated The DSM is implemented at the level of parallel programming language via data types and language primitives, and then compiled/ translated into the appropriate message passing code. The programming language have to support specific shared data objects and the primitives to manage them. Portability across different systems is realized recompiling the code (thus the compiler is required to exists).

3.2.3 Hybrid

Hybrid implementations represent a variety of trade-offs between hardware and software solutions. We classify as hybrid the approaches that make use of hardware co-processors that are programmable or anyway visible at the programming level. In general a hybrid implementation reduces the cost and complexity of hardware design while allows the compiler/programmer to suggest particular behaviors to the hardware [52]. Let us see a couple of examples.

MIT Alewife [5] implements a hardware-software, space efficient directory coherence protocol. The hardware implemented part of directory mechanism holds a limited part of directory entries in order to manage common cases. In exceptional circumstances, when more entries are needed, an interrupt is generated (fast-trap) and the directory protocol is software managed. A multiple context capability efficiently support fast-trap mechanism.

Name	Level	Year	Notes	Ref.
DDM	hw	1992	COMA	[97]
Stanford Dash	hw	1992	CC-NUMA	[124]
KSR1	hw	1993	COMA	[84]
SGI Origin 2000	hw	1996	CC-NUMA	[123]
IVY	sw	1986	runtime library + OS modifications	[128]
Munin	sw	1990		[49]
Threadmarks	sw	1992	runtime library	[118]
Midway	sw	1993	runtime library + compiler	[37]
CLR	sw	1995		
Shasta	sw	1996		
DSM-PM2	sw	1999	run-time library	[19]
MIT Alewife	hybrid	1990	hw-based coherence supported by a sw mechanism	[5]
Stanford FLASH	hybrid	1994	programmable protocol engine into mem. controller	[121]

The Stanford FLASH [121] multiprocessor uses the specialized programmable protocol processor *MAGIC* (Memory and General Interconnection Controller) to efficiently execute coherence actions (in a pipelined fashion[2]). The coherence protocol is implemented in software but it is executed in cooperation with the MAGIC coprocessor. The approach give a considerable flexibility in experimenting and testing coherence and consistency protocols.

3.3 Memory consistency models

The challenge of a consistency model is to enable the developing of a programming model that is simple and yet allows a high performance implementation. Such implementation may be built totally in hardware or both in hardware and software (the operating system, the compiler or a library may be involved in assuring a given consistency model). In our survey, we will reason about two observable features, irrespectively of which is the layer that supply the features:

- what program orders among memory operations are guaranteed to be preserved by system as whole;
- what mechanism the system provides to enforce order explicitly when desired (such mechanisms are usually called *fences*).

It is worth pointing out that fences are mechanisms to enforce order among reads, writes and read-writes, both among operations in a single thread and between those

[2]See also pipelined write notices, page 98.

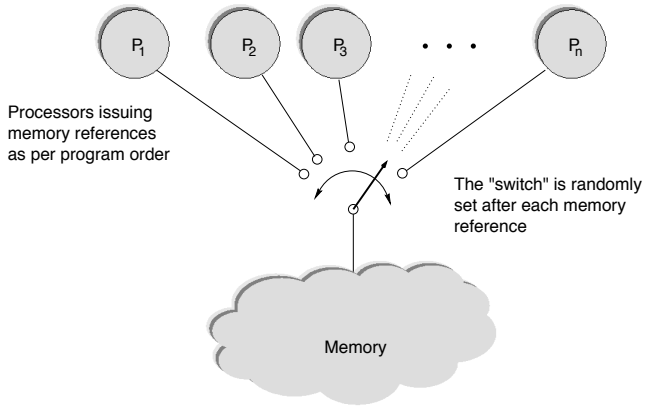


Figure 3.3: Abstraction of the memory subsystem under the sequential consistency model

of different threads, at the hardware/firmware level. Pretty often programmer level synchronizations (locks and barriers) are built using atomic read-writes operations (exchange, fetch&add ...) that act as fences. However, the compiler (or the assembler programmer) may also decide to use a fence to enforce the order between two given regular memory operations (read, write).

In the following we consider sequential consistency, that is for historical reasons the seminal memory consistency model.

3.3.1 Sequential consistency

The intuitive concept of the extension of the uniprocessor behavior to shared memory (multiprocessor) model was first formalized by Lamport as follow [122]:

> *A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program.*

There are two relevant aspects in sequential consistency:

1. Maintaining program order among memory operations from individual processors.

2. Maintaining a single sequential order among operations from all processors, i.e. memory operations must execute atomically or instantaneously with respect to other memory operations.

Sequential consistency provides a simple abstraction of the system as illustrated in Figure 3.3. Conceptually, there is a single global memory and a switch that connects an arbitrary processor to memory at any time step. Each processor issues memory operations in program order and the switch provides the global serialization among all memory operations [82, 3].

Consider the following example. Let us assume that `A` and `B` are shared between processors Pi and Pj and initial values of `A` and `B` are `0`.

	Pi	Pj
`op1(i)`	`write(A,1);`	`vt1=read(B);`
`op2(i)`	`write(B,2);`	`vt2=read(A);`

As far as sequential consistency is concerned, it is not important in what order memory operation between Pi and Pj are interleaved. What matters for sequential consistency is that they appear to complete in a way that satisfies the constraints just described. Let `op1(i)` be the first operation issued by processor `i`, in the example just shown the memory operation may execute and complete in the following orders:

`op1(i);op2(i);op1(j);op2(j);`	`(vt1=2,vt1=1)`
`op1(i);op1(j);op2(i);op2(j);`	`(vt1=0,vt2=1)`
`op1(i);op1(j);op2(j);op2(i);`	`(vt1=0,vt2=1)`
`op1(j);op2(j);op1(i);op2(i);`	`(vt1=0,vt2=0)`

These are all the orders in which, for all processors `k` the memory operation `op1(k)` is issued and completed before `op2(k)`. Under sequential consistency the result `(vt1=2,vt1=0)` would not be allowed, because:

- `op1(j)` must be executed before `op2(j)`, thus `vt1` will be set before `vt2`.
- if `vt1==2` then both `op1(i)` and `op2(i)` have already been executed and sooner or later `op2(j)` will be executed, thus `vt1==1`.

Let us consider a more significant example:

Pi	Pj
`write(A,0);`	`write(B,0);`
`...`	`...`
`write(A,1);`	`write(B,1);`
`if (B==0) ...`	`if (A==0) ...`

Sequential consistency assure that both `if` statement are not evaluate to true, maintaining the programmer intuition.

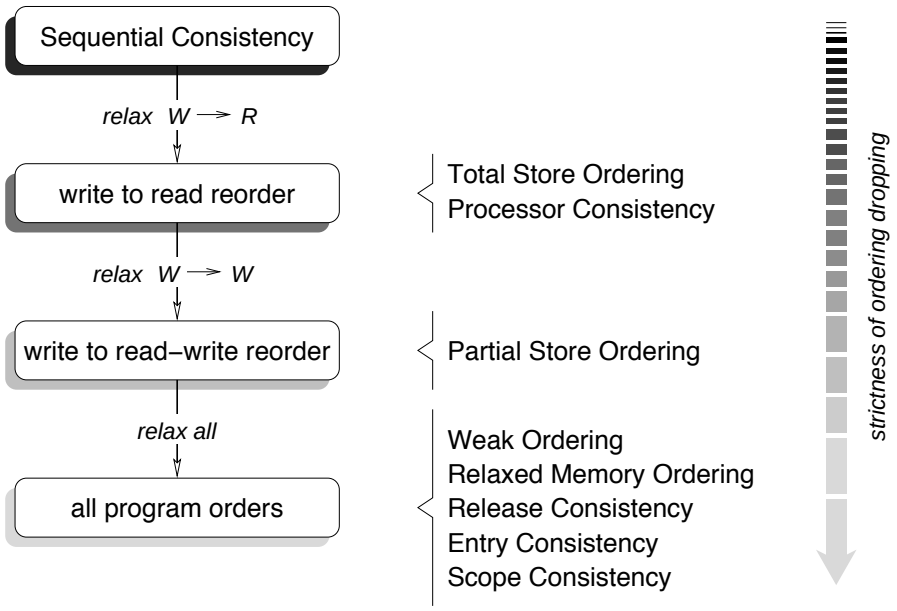


Figure 3.4: Relaxation relations among various system specification.

3.3.2 Relaxed consistency models

As an alternative to sequential consistency, several relaxed memory consistency models have been proposed in both academic and commercial settings to enable latency tolerant memory optimization in programs. Many currently shipped (multi)processor support one of more of them, each with its own mechanisms for enforcing orders. We try here to summarize the seminal concepts over these models rely on, for a detailed description of all models we refer back to the suggested literature in the field.

Relaxed models can be characterized and grouped using as key feature the relaxations in program order they allow, i.e. the ordering among reads (R) and writes (W) performed by a single processor to different address [3, 90, 100]. For instance we will sketch W→R if we don't allow a write to be bypassed (completed after) by a read. An atomic read-write operation (e.g. fetch&add, compare&swap, load-linked/store-conditional) is treated as being both a read and a write, so it can be reordered with respect to another operation only if both a read and a write can be reordered with respect to that operation. All models in the class all program orders in Figure 3.4 allow also the reordering of two reads to the same location. Figure 3.5 shows orders imposed in a program by several consistency models.

As discussed in Section 3.1.2 relaxed memory consistency models can be used to optimize the memory sub-system performance in two different ways (at least). In the next section we briefly digress on the first method, i.e. the *hiding of remote memory access latency*. Then, we focus on the *reduction of coherence oriented*

communication, that is even more important for our work. In the following we will take into account, from time to time, the most important (for our work) among all consistency models presented in Figure 3.4. Clearly, the presentation does not pretend to conclude the analysis of the incredibly large set of existing consistency models, but tries only to characterize them and present some significant examples.

Hiding long latency memory accesses

We discuss relaxed memory consistency models in their capacity of hiding of remote memory access latency. The idea is hiding memory latency by finding something else useful for the processor to do, while remote memory accesses are ongoing. The memory operation may be generated where it naturally occurs in the program, but the processor is allowed to proceed past it and find other independent computation or memory access that would come later in the same thread of execution. The technique has been mainly used in hardware implementations of shared address space for many factors. Since the goal is to keep the processor busy in useful work, all actions undertaken to make the memory operation non-blocking have to be either faster than the latency itself or run by an independent processing element. In both cases, a co-processor is the natural candidate. Moreover, if processors issue memory operations in program order, to make memory operations non-blocking we have to operate at memory transactions protocol level, that is commonly hardwired.

The main motivation of the R→W relaxation is to allow the hardware to hide the latency of write operations. While the write miss is still outstanding and not visible to other processors, the processor can issue and complete reads that hits in its cache or even a single read that miss in its cache. This class of models includes Total Store Ordering (TSO) and Processor Consistency (PC), that mainly differ in when they allow a read to return the value of a write. The TSO model allows a read to return the value of its own processor's write even before the write is serialized with respect to other writes to the same location. The PC model is even less strict with respect to TSO allowing a read to return the value of any write before the write is serialized or made visible to other processors. Relaxing the program order from a write followed by a read can improve performance substantially at the hardware level by effectively hiding the latency of a write operations. For compiler optimization, however this relaxation alone in not effective in practice. The reason is that reads and writes are usually finely interleaved in a program; therefore, most reordering optimizations effectively results in reordering with respect to both reads and writes. Thus, most compiler optimizations require full flexibility of reordering any two operations in program order. The ability to only reorder a write with respect to a following read is not sufficiently flexible [91, 3].

The program order requirements can be further relaxed by eliminating constraints between writes to different locations (R→W, W→W relaxations). The Sun SPARC V8 Partial Store Ordering (PSO) model is the only example of such a model. As the previous set of model, the optimizations allowed by PSO are not sufficiently

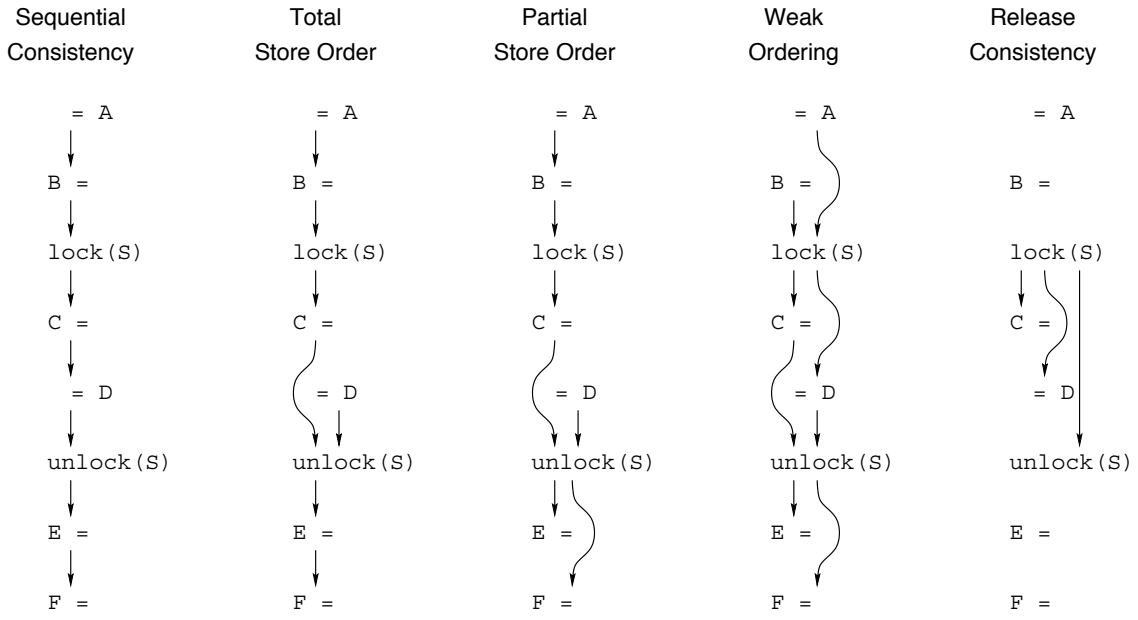


Figure 3.5: Orders imposed in a program by various consistency models. An arrow show a mandatory order, arrows' transitive closure show the partial order among the instructions.

flexible to be useful in practice to a compiler [91].

Next step in relaxing all program order requirements between all operations to different locations. Relatively to previous consistency models the benefit is that multiple read requests can be outstanding at the same time, can be bypassed by later writes in program order, and can themselves complete out of order, thus allowing the hiding of read latency. These models are particularly well matched to dynamically scheduled processors whose implementation indeed allows them to proceed past read misses to other memory references. They are also the only models that allow many of the key reordering and elimination of accesses as done by compiler optimizations. Weak Ordering (WO) is the seminal model, Release Consistency (RC) is an extension of WO supported by the Stanford DASH prototype.

The motivation behind WO model is that most parallel programs use synchronization operation to coordinate accesses to data when necessary. Between synchronization operations, they do not rely on the order of accesses to be preserved. Before a synchronization operation is issued, the processor waits for all previous operation in program order to have completed. Similarly, memory accesses that follow the synchronization are not issued until the synchronization operation completes. Read, write and read-write operations that are not labeled as a synchronization can be arbitrarily reordered between synchronization operations.

The RC model further relax WO model distinguishing among types of synchronization operations and exploiting their semantics. In particular, it splits synchro-

Model	Used in	Op.Ordering	Fences
SC	most machines as optional mode	R→R, R→W, W→R, W→W	All memory operations
TSO	IBM S/370, DEC VAX, SGI challenge	R→R, R→W, W→W	memory barrier, read-write
PC	Intel Pentium	R→R, R→W, W→W	memory barrier, read-write
PSO	Sun SPARC (V8)	R→R, R→W	memory barrier, read-write
WO	IBM PowerPC		explicit synchronization
RC	DEC Alpha, Stanford DASH		release, acquire, read-write
RMO	Sun SPARC (V9)		four flavors memory barriers

Table 3.1: Memory models supported by various processors and systems.

nization operations into acquires and releases. An *acquire* is a read operation that is performed to gain access to a set of operations or variables. A *release* is a write operation that grant permission to another processor to gain access to some operations or variables. The splitting of synchronization operations in classes cuts down the number synchronization required.

It is clear that this class of models provides considerable reordering freedom to the hardware and the compiler. The prominent specifications or relaxed models just described are summarized in Table 3.1 [3, 65]. Notice, all modern processors support at least one of the relaxed memory consistency model.

Performance impact

A processor can proceed past a memory operation to other instructions if the memory operation is made non-blocking. The room for latency tolerant overlapping of memory operations is greatly restricted by sequential consistency requirements. Relaxed models allow the compiler (or the programmer) to deal more effectively with optimization towards latency tolerance.

Actually latency tolerance methods by overlapping of memory operations, both one each other and with processor busy time, represent the instantiation of memory consistency model we have seen in the previous section. Starting from the more conservative (Sequential Consistency) to the more aggressive (Release Consistency), our ability to introduce memory tolerant optimization grows at the same pace of complexity of hardware/software design.

To proceed past write misses (W→R, W→W relaxations), the only support we need in the processor is a *write buffer*. Most uniprocessors already include an aggressive write buffer placed before the first level cache. In a uniprocessor, this approach

Dynamic scheduling or **out-of-order** scheduling means that instructions are fetched and decoded in program order as presented by the compiler, but they are executed by the functional units in the order in which the operands become available at run time. In other words, a processor with out-of-order scheduling simultaneously examines several consecutive instructions within an instruction window, using one of the well known methods like scoreboarding or reservation stations (Tomasulo's algorithm) [100].

Speculative execution allows the processor to look at and schedule for execution instructions that are not necessarily going to be useful to the program's execution. Instruction after the speculation point (e.g. branch) continue to be decoded, issued and executed, but these are not allowed to commit their values into the architectural state of the processor until all prior speculation have been resolved [100, 99].

Observe that these methods are independent from memory consistency models, since out-of-order execution does not means that the results of instructions is made visible out-of-order at memory system level. It is quite possible that operations completed (in any order) keep their results in reorder buffer, without temporarily committing the register file. Thus, even with aggressive out-of-order execution, memory operation can complete in program order.

Table 3.2: Concepts recap: Dynamic scheduling and speculative execution.

is very effective as long as reads check the write buffer to satisfy dependences, i.e. the read may be allowed to bypass the write buffer as long as a write to the same location is not pending in the write buffer. In multiprocessors, the write buffer is responsible for controlling the visibility of writes to the rest of the extended memory hierarchy, and hence to other processors. This is enough to implement models like PC and TSO. If in addition, multiple writes in the write buffers may merged (coalesced) or retired, so they may complete out of order, we obtain much like PSO model. Moreover, write buffer enables the writes to be pipelined through the memory hierarchy. Of course having multiple writes outstanding in the memory system requires that the caches allows multiple outstanding misses.

Relaxing W→R constraint usually is enough to hide most of the write latency from the processor, while relaxing W→W does not help masking read latency since reads are blocking. As matter of fact, Sun SPARC V8 PSO model has not got a great success in commercial processors design and has been replaced by Relaxed Memory Order (RMO) in later Sun SPARC V9 design.

The last step is to make also reads non-blocking (all relaxations) and introduce a mechanisms to look ahead beyond dependent instructions. These instructions might be finely interleaved with branches. Thus, going beyond reads requires effective branch prediction as well as speculative execution[3] past predicted branches. Nowadays, the trend is towards increasingly sophisticated processors that provide all these features in hardware (for instance they are included by the Intel PentiumPro

[3]See Table 3.2.

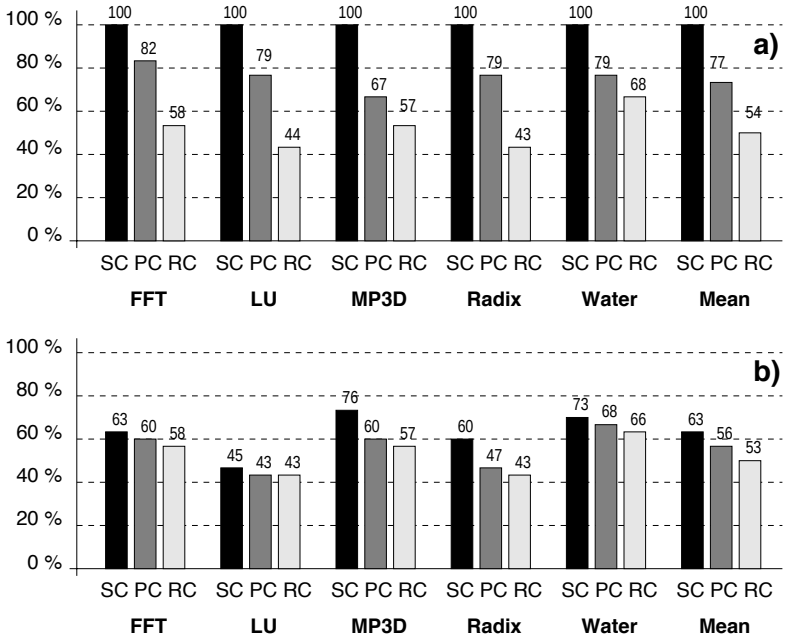


Figure 3.6: Performance of straightforward implementation of memory consistency models versus speculative out-of-order implementation. (Performance figures taken from Adve et al. [4])

and the Sun UltraSparc). Theoretically, the compiler has a great opportunity to make latency hiding optimization using non-blocking reads. The most interesting question is whether, with aggressive, dynamically scheduled processor, relaxed methods that allow non-blocking reads still buys substantial performance gains over Sequential Consistency.

A recent study (Adve, Pai and Ranganathan [4]) seems to indicate that, in absence of sophisticated hardware mechanisms like out-of-order and speculative execution, Release Consistency is still beneficial with respect to both Sequential Consistency (SC) and Processor Consistency (PC). The result is shown is Figure 3.6. The top series of histograms (Fig. 3.6a) show the performance of PC and SC normalized with respect to SC for six applications of the Stanford SPLASH/SPLASH-2 suites [156] on a simulated hardware cache coherent shared-memory multiprocessor system (8-16 PEs). In the bottom series of histograms (Fig. 3.6b) is shown the execution time of the same tests in case of speculative out-of-order processor implementation normalized with respect to straightforward SC implementation. We see that nowadays hardware optimizations at single processor level results in a significant narrowing of the performance gap between consistency models.

Reduction of coherence oriented communications

Let us now see relaxed memory consistency models from another viewpoint. As previously mentioned, relaxed memory consistency models might enable the *reduction of coherence oriented communications* in DSM systems. The primarily source of overhead in a DSM system is the large amount of communication that is required to maintain consistency, in other words to maintain the shared memory abstraction. The issue become particularly critical in software DSM where communications are expected to be expensive, and totally in charge of the processor. In fact, from early software DSM systems such as IVY [128] relying on sequential consistency, software DSM designer has rapidly converged towards less strict memory consistency models.

Ideally, the amount of communication for an application executing on a software DSM system should be comparable to the amount of communication for the same application executing directly on the underlying message passing system. In reality, the message passing programming model forces the programmer to distribute by hand (and carefully) data structures in application threads. This actually represent both the principal drawback of the programming model and the trump card of its run-time support. Conversely, DSM support may suffer from data *false sharing* and other phenomena that raise the traffic of the underlying communication layer. False sharing occurs when two threads on different machines concurrently update different shared data items, ideally independent one each other, but lying in the same coherence unit (i.e. the same virtual memory page). False sharing may cause a coherence unit to be "ping-ponged" back and forth between different machines. This kind of communications are extraneous to the logic of the parallel program, and they are a serious caveat for the application performance. Another issue is the size of coherence oriented messages, that has strong relation with both false sharing and the size of coherence unit.

The mechanism to reduce the coherence oriented traffic basically consist in delaying coherence oriented actions (namely write notices) until certain synchronization points. The choice of synchronization points have important implication both on theoretical performance and programming model.

As discussed, the problem is particularly critical in software DSM implementations. In fact the trend in software DSM systems is to move towards increasingly relaxed memory consistency models. For this reason we start here from the point we stopped in the previous section, the Release Consistency and moving towards even more relaxed models (see Figure 3.4).

Release Consistency

Release Consistency (RC) originate from the observation that programmers use synchronization to separate accesses to shared variables by different threads; typically a pair of synchronizations trammel a critical section. As previously discussed, RC splits synchronization operations into *acquires* and *releases*. Lock acquires and lock

releases map in the natural way onto *acquires* and *releases*, barriers are treated as a release-acquire pair. The idea is that memory consistency is guaranteed only at synchronization points, while temporarily inconsistencies are allowed between synchronization points.

Formally to ensure RC the memory subsystem have to respect the following constraints:

- Before a read or a write is allowed to perform with respect to any other processor, all previous acquire accesses must be performed.
- Before a release access is allowed to perform with respect to any other processor, all previous read and write accesses must be performed.
- Synchronization accesses must be sequentially consistent one each other.

Notice the model establishes that *before* the release all written values after the acquire have to be committed: Any moment between the write and the release is legal moment to issue and complete the write notice. For example, if the thread T_1 issues a write on variable V, another thread T_2 may have access to the value of V, written by T_1, only after T_1 has issued a release.

RC relaxes the constraints of sequential consistency in the following ways:

1. Reads and writes can be *buffered* or *pipelined* between synchronization points.
2. Reads and writes following a release do not need to be delayed for the release to complete. Release only signals the state of past accesses to shared data.
3. An acquire access does not need to delay for previous reads and writes to complete. An acquire only controls the state of future accesses to shared data.

The first point actually represent the main source of performance gain of RC with respect to sequential consistency [50]. Moreover, it describes the dual use of RC: *pipelining* is mainly used to mask communications latency, *buffering* to reduce frequency of communications.

Hardware solutions, in general used to avoid the stalling of the processor, adopt pipelining. They do not guarantee that the effect of the write will be see until the release, but in fact they usually will be. Since the goal is to make the memory operation non-blocking, there are no reason to delay the write notice (e.g. invalidations) at the release point: a specialized hardware device does all the work without any processor intervention, and it does as soon as possible. Since multiple write notices may be issued in a acquire-release region, they are pipelined reducing the total latency of communications. However, pipelining does not help too much neither in reducing the frequency of communication nor the false sharing phenomena.

The significant difference from hardware and software solutions lie in the fact that software solutions adopt the buffering behavior in order to reduce coherence related

communications. In this behavior, write notices are really not propagated until a synchronization point, avoiding the proliferation of write notices. When a processor writes to several different replicated data items within a critical section, the buffering implementation buffers writes to shared data until a *subsequent synchronization point*, at which point it transmits the buffered writes.

The choice of the *subsequent synchronization point* represent another key issue for RC models:

Eager Release Consistency (ERC), introduced in *Munin* [49, 50], send write notices at release points. Exiting from a critical section, a thread invalidate all copies of dirty coherence units (memory pages), i.e. memory pages that have been modified in the critical section. Only the first access to an invalidated page after a synchronization point generates a request for that page.

Lazy Release Consistency (LRC) [117] delays write notices from a release to the next acquire in temporal order. On an acquire, the thread obtains the write notices corresponding to all previous release operations that occurred between its previous acquire and its current acquire and applies them to the relevant coherence units. By further postponing coherence actions to acquires, LRC alleviate several performance problems of ERC. First, false sharing on a coherence unit occurring before the acquire, but not after it, have no effects. Second, the invalidation message (due to release) may be coalesced with acquire messages. Third, in ERC the system have to send invalidations to all thread that have used a certain coherence unit, even if it will no longer use it. In LRC we exactly known what threads will use a coherence unit (since the thread is acquiring a lock), thus only interested threads will receive invalidation messages. In contrast, LRC is significantly more complex to implement than ERC.

All relaxed consistency models have deep implications on how a program is written and executed. Let us consider some example on how a couple of simple programs are executed under ERC and LRC.

In Figure 3.7 is sketched executions of the same program under ERC and LRC. Because of effect just described LRC uses considerably less messages with respect to ERC. Notice the program have different semantics in the two models. Supposing variable `A` initially zero, a read out from a critical section issued by P2 return 0 in ERC and 1 in LRC. In the latter model the read is locally resolved, while in the former the earlier invalidation cause a local copy to be refresh.

Let us underline the difference in the programming model supported by the two consistency model. Suppose the variable `A` in the Figure 3.8 code is initialized to 0. Sooner or later Pi will grab the lock (acquire), execute the `write`, and finally unlock (release) leaving the critical section. Pj is cycling at the `while` loop.

Under ERC, when Pi issues the release, thus notify to Pj the new value of `A` that is 1. At this moment Pj jumps out of the loop and eventually execute its

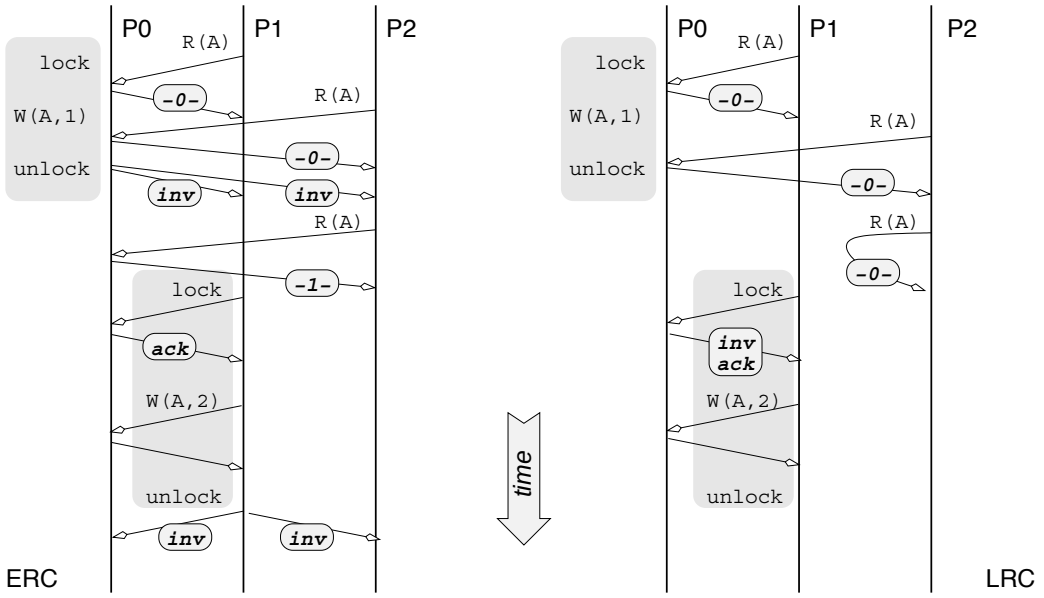


Figure 3.7: Eager Release Consistency (ERC) and Lazy Release Consistency (LRC). ERC propagate invalidations at release point, while LRC coalesces invalidation with lock grant at acquire point.

Pi	Pj
`lock(L);`	`...`
`write(A,1);`	`...`
`unlock(L)`	`while (A==0) nop;`
`...`	`lock(L);`
`...`	`vt1=read(A);`
`...`	`unlock(L);`

Figure 3.8: An example highlighting differences between ERC and LRC programming models.

critical section. on the contrary, in LRC write notices are propagated only at the moment of acquire operation (lock), thus Pj will known the new value of `A` at the moment of the `lock`, but since Pj will never spontaneously goes out from the loop, Pj is stuck in the loop.

Entry Consistency

Entry Consistency (EC) has been introduced in the Midway DSM system [37]. Like both variants of release consistency, it requires the programmer to use lock (acquire) and unlock (release) at the start and end of each critical section, respectively. However, unlike release consistency, entry consistency requires each ordinary shared variable to be associated with some synchronization variable such as a lock or barrier. If it is desired that elements of an array be accessed independently in parallel, then different array elements must be associated with different locks. When an acquire is done on a synchronization variable, only those ordinary shared variables guarded by that synchronization variable are made consistent. Entry consistency (EC) differs from lazy release consistency in that the latter does not associate shared variables with locks or barriers and at acquire time has to determine empirically which variables it needs.

Formally, a memory exhibits entry consistency if it meets all the following conditions :

1. An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.

2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in non-exclusive mode.

3. After an exclusive mode access to a synchronization variable has been performed, any other process next non-exclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

Scope Consistency

Scope Consistency (ScC) tries to bridge the potential performance advantages of ease of programming offered by Release Consistency. The basic idea under Scope Consistency is to use a concept of consistency scope to implicitly establish the relationship between data and synchronization events, thus further relaxing Lazy Release Consistency.

Data modifications performed within a scope are guaranteed to be visible within that scope only. A scope is the union of critical sections protected by a given lock,

with an additional global consistency scope for barriers which include the whole program. Data modifications performed within a scope are guaranteed to be visible within that scope only. Updates made outside a scope session (critical section) may not be visible to other threads. For a formal definition of rules defining Scope consistency we refer back to Iftode, Singh and Li paper [109]. *Jiajia* DSM system [107] implements in software Scope Consistency.

The potentially innovative contribute of ScC with respect to EC is the implicit binding of data to synchronization. Such binding is dynamical and transparent, and in most cases program written for LRC can run with ScC without modifications. However, not all LRC programs are correct under ScC. Authors report that LRC programs can be easily modified to run under ScC expanding critical sections by moving synchronization operations or adding more scopes (more locks). However, the former solution have a non trivial performance penalty due to the increased contentions on locks. The latter solution require partial rewriting of the program.

The relation of ScC with EC is more intriguing. In ScC scopes are related to tasks or section of codes: the control flow of thread enter into a (memory) scope and exit from it, within such session the thread can see the variables of that scope. In EC critical sections are related to data structures. This difference maps in a presumed ease of programming of ScC programs with respect to LRC.

Indeed, have anyone tried to write a parallel version of the quicksort with scope consistency? Quicksort, and in general divide&conquer algorithms seem to us very hard to be written with ScC.

DAG consistency

All consistency models we have seen up to now have had one thing in common: they are "processor centric" in the sense that they define consistency in terms of actions by physical processors. Leiserson et al. [42, 112] proposed a pretty different consistency model: DAG consistency. DAG consistency is defined on the DAG of threads that makeup a parallel computation. Intuitively, a read can "see" a write in the DAG consistency model only if there is some serial execution order consistent with the DAG in which the read sees the write. Unlike sequential consistency, but similar to certain processor-centric models, DAG consistency allows different reads to return values that are based on different serial orders, but the values returned must respect the dependencies in the DAG.

DAG consistency is a relaxed consistency model for distributed shared memory. It has been used in the *Cilk* language developed at MIT by the Leiserson's group [40, 42, 41, 85, 86, 146]. In *Cilk* DAG consistency is maintained by means of the BACKER [42] algorithm.

As far DAG consistency concern, shared memory consists of a set of objects that instructions can read and write. When an instruction performs a read of an object, it receives some value, but the particular value it receives depends upon the consistency model. DAG consistency is defined separately for each object in shared

memory.

In order to give the definition, we first define some terminology. Let $G = (V; E)$ be the DAG of a multithreaded computation. For $i, j \in V$, if a path of nonzero length from instruction i to j exists in G, we say that i (strictly) *precedes* j, which we write $i \prec j$. To track which instruction is responsible for an object's value, we imagine that each shared-memory object has a tag which the write operation sets to the name of the instruction performing the write. We make the technical assumption that an initial sequence of instructions writes a value to every object. We can now define DAG consistency.

Definition 1 *The shared memory M of a multithreaded computation is DAG consistent if for every location x in the global address space, there exists a function $f_x : V \to V$ such that the following condition hold.*

1. *For all instructions $i \in V$, the instruction $f_x(i)$ writes on x.*
2. *If an instruction i writes on x, then we have $f_x(i) = i$.*
3. *If an instruction i reads, it receives a value tagged with $f_x(i)$.*
4. *For all instruction $i \in V$, we have $i \not\prec f_x(i)$.*
5. *For each triple of instruction i, j and k, such that $i \prec j \prec k$, if $f_x(j) \neq i$ holds, then we have $f_x(k) \neq i$.*

Informally, the function $f_x(i)$ represents the viewpoint of instruction i on the contents of location x, that is, the tag of x from i's perspective. Therefore, if an instruction i writes, the tag of x becomes i (part 2 of the definition), and when it reads, it reads something tagged with $f_x(i)$ (part 3). Moreover, part 4 requires that future execution does not have any influence on the current value of the memory. The rationale behind part 5 is the following: when there is a path from i to k through j, then j "masks" i, in the sense that if i's view of x is no longer current when j executes, then it cannot be current when k executes. Instruction k can still have a different viewpoint on the memory than j, for instance, it can see writes performed by other instructions (such as l in the figure) incomparable with j.

A comparison among DAG consistency and others classical consistency models can be found in [86].

The BACKER algorithm. Let us suppose that all of the threads of a multithreaded algorithm have access to a single, shared virtual address space, and in order to support such a shared memory abstraction. The BACKER coherence algorithm assumes that each processor's memory is divided into two regions, each containing pages of shared memory objects. One region is a page cache of C pages of objects that have been recently accessed by that processor. The rest of each processors'

memory is maintained as a backing store of pages that have been allocated in the virtual address space. Each allocated page is assigned to the backing store of a processor chosen by hashing the page's virtual address. In order for a processor to operate on an object, the object must be resident in the processor's page cache; otherwise, a page fault occurs, and BACKER must "fetch" the object's page from backing store into the page cache. When a page fault occurs, no progress can be made on the computation during the time it takes to service the fault. In addition to servicing page faults, BACKER must "reconcile" pages between the processor page caches and the backing store so that the semantics of the execution obey the assumptions of DAG consistency.

In the BACKER coherence algorithm, versions of shared memory objects can reside simultaneously in any of the processor caches and the backing store. Each processor's cache contains objects recently used by the threads that have executed on that processor, and the backing store provides default global storage for each object. In order for a thread executing on the processor to read or write an object, the object must be in the processor's cache. Each object in the cache has a dirty bit to record whether the object has been modified since it was brought into the cache. BACKER uses three basic operations to manipulate shared-memory objects: fetch, reconcile, and flush.

A fetch copies an object from the backing store to a processor cache and marks the cached object as clean. A reconcile copies a dirty object from a processor cache to the backing store and marks the cached object as clean. Finally, a flush removes a clean object from a processor cache. The BACKER coherence algorithm operates as follows. When the application code performs a read or write operation on an object, the operation is performed directly on a cached copy of the object. If the object is not in the cache, it is fetched from the backing store before the operation is performed. If the operation is a write, the dirty bit of the object is set. To make space in the cache for a new object, a clean object can be removed by flushing it from the cache. To remove a dirty object, it is reconciled and then flushed.

Besides performing these basic operations in response to user reads and writes, BACKER performs additional reconciles and flushes to enforce DAG consistency. For each edge $i \rightarrow j$ in the computation DAG, if instructions i and j are executed on different processors, say P and Q, then BACKER causes P to reconcile all its cached objects after executing i but before enabling j, and it causes Q to reconcile and flush its entire cache before executing j. Note that if Q's cache is flushed for some other reason after P has reconciled its cache but before Q executes j (perhaps because of another interprocessor DAG edge), it need not be flushed again before executing j.

3.3.3 Multi-protocol consistency

It has been noticed that different applications exploits different performances with different consistency model. Moreover, as seen comparing ERC with respect to

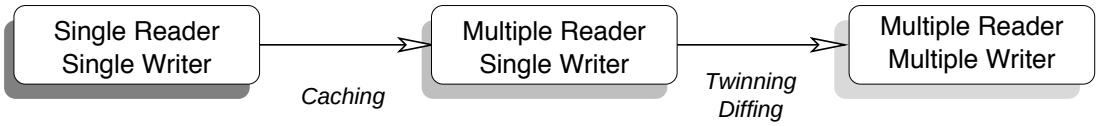


Figure 3.9: DSM algorithms taxonomy.

LRC (see Figure 3.8), the consistency models have profound implications on how a program is written and executed. It would clearly useful provide the programmer with many consistency protocols. It would be very useful is them can be exploited in the same program.

DSM-PM2 platform supports this features [19]. It provides the programmer with several ready-made consistency protocols: sequential (in two flavors), eager release, lazy release, Java (in two flavors). Also, DSM-PM2 provide the programmer with the possibility to define their own consistency protocol: a feature that makes it a valuable platform to experiment consistency protocols.

DSM-PM2 has noticed that all DSM systems share a number of common features. As an example, a DSM core should provide routines to detect page faults, to extract information related to each fault (address, fault type, etc.) and to associate protocol-specific consistency actions to a page-fault event. Overall, DSM-PM2 triggers 8 kinds of events (page faults, receipt of a page request, receipt of the requested page, receipt of an invalidation request, lock acquire, lock release and barrier calls). The programmer can build his new protocol simply by fulfilling handlers of these events.

3.4 Data replication

DSM systems can be categorized by whether they allow data copies. As depicted in Figure 3.9, our classification starts from the simpler model in which no data copy is allowed and proceed towards more complex models by successive relaxations. Relaxations are actually made allowing read-only data copies and read-write data copies.

SRSW. *Single Reader Single Writer* class systems permit just one copy of data items in the distributed memory. Memory addresses may be centrally managed by a server or statically distributed. The centralized server solution suffers serious scalability problem since the manager rapidly becomes a bottleneck. The static distribution of memory addresses allows the static distribution of the functionalities of the server, i.e. it permits the distribution of the server onto the nodes. Simple mapping functions can serve to locate the appropriate server for the correspondent memory block. This class of methods may be implemented using migration (even if is not required). Anyway, the SRSW class solutions are rarely used in DSM systems

Manager. The node managing the write access to a data block. **Owner.** The node owning the only writable copy of data block. **Copy set.** The nodes holding copies of the data block.
Migration. The data is shipped to the location of the data access request allowing subsequent accesses to the data to be performed locally. Typically, the whole page or block containing the data item migrates instead of an individual item requested. This algorithm takes advantage of the locality of reference exhibited by programs by amortizing the cost of migration over multiple accesses to the migrated data. However, this approach is susceptible to thrashing, where pages frequently migrate between nodes while servicing only a few requests. To reduce thrashing, we could use a tunable parameter that determines the duration for which a node can possess a shared data item. This allows a node to make a number of accesses to the page before it is migrated to another node.

Table 3.3: Concepts recap: Manager, owner, copy set, and migration mechanism

since they do not exploit important performance improvements such as replication of read-only data items.

MRSW. *Multiple Reader Single Writer* (or read-replication) class systems exploits the possible parallelism in read sharing application patterns. The writable data copy have to be unique in the system. Since the read sharing tend to be the prevalent pattern in parallel applications, the room for performance improvements with respect to SRSW is remarkable. A MRSW systems can be implemented in several ways.

- The centralized manager gathers and process all read and write requests. On a memory operation, the manager mediates between the requesting node and the owner. The manager keeps trace and authorize data coping/migration for the owner to the requester. It knows in each moment the copy set and the status of all memory blocks.

- The previous organization may be improved distributing a predetermined set of data blocks for each node and distributing functionalities accordingly. The distributions proceeds according to a static mapping function. The organization may tolerate a pretty high memory operation pressure provided a "good" mapping function. The quality of mapping function depends on the application, and may be instantiated depending on the application or the data structures.

- Li and Hudak [129] have proposed another distributed organization based on the concept of "probable owner". For each data block the probable owner (not necessarily the real owner) is saved in the support data structures. All requests

are sent to the probable owner, who is ready to forward the message in the case it no longer is the real owner. All memory operations and invalidations hold additional information to update owner field during the application run in such a way to limited message bouncing. The algorithm allows a dynamic distribution of memory block in memory. Authors report a logarithmic degradation of support performance with respect to numbers of nodes accessing the same data block.

MRMW. *Multiple Reader Multiple Writer* (or full-replication) class systems allow replication of data blocks with both read and write permissions. It is an extension of the read replication algorithm. Because many nodes can write shared data concurrently, the access to shared data must be controlled to maintain its consistency. Observe that a multiple writes protocol, anyway, takes account of concurrent writes to the same memory block provided data structures within the block are not subject to a mutual exclusion discipline. The most frequent case is the one in which two different data structures are mapped in the same block (coherence unit), i.e. the false sharing case. It is pretty clear that the problem gains a particular importance in the case coherence unit is large, as for example in software implemented DSMs.

The first implementation of a MRMW protocol was in TreadMarks [118]. TreadMarks adopt as coherence unit the memory page. A shared page is initially protected, the first write after a synchronization point, a protection fault is raised. The fault manager makes a copy of the page (namely the *twin*) and then get rid of the write protection from the page. At next synchronization point[4] the twin and the (dirty) memory page are compared to create a *diff*, i.e. a compact representation of the difference between the two pages. Diff are then propagated to make all copies coherent. Of course, "when" multi-written pages are made coherent depends on the consistency discipline.

Anyway, a multiple writer protocol does not make sense if not completed by an "enough lazy" memory consistency model. Clearly, performance benefits of multiple writer protocol can be exploited only if write operation can be issued and completed in parallel, i.e. if we don't require each write operation to ordered each other (but only at certain synchronization points).

Moreover, the creation of twin, together with diff processing and application may produce a significant processing overhead due also to the node cache hierarchy pollution. In addition, twin and diff have to be stored, thus they require additional memory room. The actual memory overhead due to twin pages depends on many factors, among the other the memory consistency model, the particular implementation of MRMW protocol and how the application has been written. In TreadMarks the shared space is limited to the physical memory of one node (because the memory room allocated for twins is not dynamically managed). Jiajia does not have such limitation but require a fixed additional space for twin pages, that are managed with

[4]Release in ERC, acquire in LRC, see also Release Consistency at page 97.

a replacing mechanism.

3.5 Software implementation issues

Software DSM implementations provide a shared address space with no specific additional hardware support. Typically, software DSM are implemented on top of a Beowulf class cluster, i.e. a collection of complete processing node interconnected by means of a (standard or proprietary) network. The processing node may be in turn a parallel machine, as for example a bus-based *Symmetric Multi-Processor* (SMP). This case is quite common and it has important effects on DSM design; nevertheless, at the coherence and consistency end we consider them as a single node since they resolve these issues in hardware.

In general, DSM are implemented at the software level by leveraging to virtual memory support provided both by processor MMU and operating system[5]. In software DSM systems the typical DSM functions are performed in software.

Denning [78] summarize the three major concerns for the virtual memory designers as follows: *"(1) Address mapping, the process of translating virtual addresses to memory addresses, should easily accommodate the kinds of objects that programmers are working with. (2) Address translation should be efficient, costing no more than 3% of hardware execution speed. (3) Overall system performance, measured by throughput and response time, should be within 10% of the best possible performance attainable for a given workload."*

In message-passing systems, local references incur no more overhead than on a uniprocessor. In the shared address space the simple address translation for shared data access in software as opposed to hardware may heavily penalize application performance. As example Scales and Lam [151] experiment (in Burnes-Hut application) a reduced performance by about 20% due only to software address translation for local memory references. This performance slowdown is due to a number of factors, among the others: determining if the referred address is local or remote, determining the address of a possible local copy and the cost of cache management. In addition, the presence of additional code related to local memory reference might prevent possible optimization of code during compilation.

It is clear that the address translation overhead can be fully minimized only adopting an all hardware solution. Target architecture for DSM, i.e. Beowulf class architectures, have no such support by definition. In order to mitigate the address translation overhead many DSM are built using as much as possible the hardware available on off-the-self processors.

[5]Indeed, software implemented DSM systems are sometime called *Shared Virtual Memories* (SVM).

Page based

One popular way do to it is the following: *Try to emulate the cache of a multiprocessor using the MMU and operating system software.* In a DSM system, the address space is divided up into chunks, with the chunks being spread over (in some way) all the processors in the system. When a processor references an address that is not local, a trap occurs, and the DSM software fetches the chunk containing the address and restarts the faulting instruction, which now completes successfully. Clearly, the trap is triggered by a page fault event, under the control of processor MMU. The method makes the local access for non faulting instructions as cheap as in uniprocessor systems. Faulting references incur the overhead of fetching the page from a remote node and the overhead of managing internal tables, that may be pretty large with respect to uniprocessor virtual memory page replacement. Several key issues emerge:

- Standard MMU can be programmed via operating system to discriminate memory accesses at the granularity of memory pages. Using off-the-shelf processors there are no ways to discriminate at hardware level two accesses in the same memory page. Due to the distributed nature of the underlying memory system any parallel application will require the replication or the migration of data structures (shared memory chunks), that can be done at the minimum granularity of underlying memory page.

  In this setting, a careless allocation of data structure raises the likelihood of false sharing. *False sharing* of a page occurs when two different data items, not shared but accessed by two different processing elements, are allocated to a single page. False sharing yields unnecessary contention on memory pages resulting in some cases to a significant performance slowdown of the DSM.

- As previously mentioned, one possible improvement to the basic system that can improve performance considerably is to replicate chunks that are read only. Another possibility is to replicate not only read-only chunks, but all chunks. As long as reads are being done, there is effectively no difference between replicating a read-only chunk and replicating a read-write chunk. However, if a replicated chunk is suddenly modified, inconsistent copies are in existence. The inconsistency have to be prevented by using some *consistency protocols.*

- Since page faults involving remote pages are particularly expensive, the full exploitation of all sources of locality (spatial, temporal, group spatial, group temporal) in applications is a key issue in DSM based systems. Data locality is affected by several factors: data structures allocation and mapping, threads mapping with respect to data mapping, data replication management (caches management). We shall examine in great detail all of them.

3.6 Athapascan

Athapascan is parallel programming library extending C++. It is built on top of a two-tier run-time: Athapascan-0 and Athapascan-1.

Athapascan allows built the macro data flow graph of the application starting from explicitly parallel program based on the shared address model. It strongly characterizes shared memory accesses. These are distinguished as: read ("r"), read-write ("r_w"), write ("w"), and accumulation ("cw"). Athapascan is mostly inspired from Jade [148] concerning characterization of memory accesses and Cilk [43] concerning parallelism expression. It deals with data and control flow at a grain defined by the user. Parallelism is expressed through asynchronous remote procedure calls, denoted as tasks, that communicate and are synchronized only via access to a shared memory. A task definition is similar to a C procedure definition (the void returned type replaced by the task keyword). A task implements a sequential computation; it is created in program statements by prefixing a standard C++ procedure call by the "fork" keyword.

During the execution of an Athapascan program, the interpretation of the "fork" and "shared" directives are performed and the data flow graph is built. The semantics of Athapascan assumes that a task does not make side effects, otherwise the semantics of the program may be not guaranteed by the data flow graph (as in Lithium, see Section 2.3). Afterwards ready tasks can be executed. In case of recursive parallel programs forks must be re-interpreted (leading to an on-line dynamic building of the data flow graph [87]). The scheduling algorithm sort out the mapping of tasks and data using this graph. The scheduler uses the work-stealing principle [51]. However, the scheduler is configurable. Athapascan provides the possibility to design new scheduling algorithms. These algorithms may use scheduling informations expressed as attributes of the fork. Some attributes are predefined, one of them enables to apply the owner computes rule for the newly created task. In this case the task will rather be mapped onto the processor holding a chosen shared object.

The Athapascan semantics relies on shared data access and ensure that the value returned by the read statements is the last written value according to the lexicographic order defined by the program. This order defines a total ordering on all tasks during the execution.

The control of the accesses semantic during execution is entirely data driven: the precedences between the tasks, the needed communications or the data copies are ensured automatically by the runtime system. It is based on an entry release consistency scheme (see also page 101); the objects entries are always done at beginning of tasks and the corresponding release at the end of tasks.

Chapter 4

eskimo: design principles

Readers' road-map. In this chapter we introduce "eskimo" language, i.e. a *skeletal* extension of C language for parallel programming based on the shared address model. We introduce the topic by briefly analyzing the lacks of previous skeletal programming framework (in particular our group's ones), thus motivating (yet) another evolution of skeletal frameworks. In Section 4.1 we present eskimo basic design principles. These are developed along parallelism exploitation (Sections 4.1.1 and 4.1.2), and memory sharing (Sections 4.1.3 and 4.1.4). The expected pay-back of the skeletal approach is discussed in Section 4.2. Eventually, we conclude sketching the differences between eskimo and some related works (*Cilk* and *Athapascan*). eskimo C language extension has been fully designed by the author himself. Part of this chapter will appear in [10].

The development of efficient parallel programs is a quite hard task. As discussed in Chapter 2, the unstructured/low-level approach exposes to the programmer many cumbersome aspects of parallelism exploitation such as: concurrent activity set up, communication/synchronization handling and data allocation. From more than a decade our research group has been active in experimenting new technologies in order to simplify parallel programming by raising the programming model level of abstraction. Skeletons have been present all along in such programming environments.

Several real world applications have been used as test-bed to validate the effectiveness of our programming environments; these includes applications in the following research areas: computational chemistry, massive data-mining, remote sensing and image analysis, visual and numerical computing [64, 150, 92, 67, 24, 63, 35, 168]. Even if the skeletal approach has been proved to be effective for some of them, the overall feedback we received cannot be considered fully satisfactory. Actually a lack of expressivity emerged, at least for some complex applications. In principle, the skeletal approach is not particularly targeted towards a class of applications. However, we experienced that some applications can be straightforwardly formulated in terms of skeleton composition, others need a greater design effort. The boundary between the two classes depends on many factors, such as the particular programming

environment and the skeleton set chosen for applications development. Anyway, some common flaws may be recognized in both the environments we designed and other research group works (see also [60]):

i) The selection of skeletons to make available in the language skeleton set is a quite critical design issue. Despite several endeavors to classify and close the parallel programming skeleton set [47, 140], in many cases during application development we experienced the need of the "missing skeleton", or at least the missing functionality for an existing skeleton.

ii) Many parallel applications are not obviously expressible as instances of (nested) skeletons, whether existing or imagined. Some have phases which require the use of less structured interaction primitives. Others have conceptually layered parallelism, in which skeletal behavior at one layer controls the invocation of operations involving ad-hoc parallelism within [67].

iii) Although all kind of languages may be equipped with a skeletal super-structure, skeletal languages has been historically designed in a functional programming style fashion [27, 76, 29, 153]. In this setting the non functional code[1] is embodied into the skeletal framework by providing the language with wrappers acting as pure functions (i.e. the seq skeleton, see also Chapter 2). Actually, the fully functional view (by its very nature) does not enhance programmer control over data storage. This feature may happen to be useful in the design of applications managing large, distributed, randomly accessed data sets.

As discussed in Section 2.1, the role of skeletons in the programming language has evolved and matured along the past decade. Such evolution has been designed to overcome (among other things) skeletal languages lack of expressiveness while preserving their ease of use.

The former two issues have already been faced in a variety of ways. As an example, SKElib [75] allows the programmer to mix skeletons and ad-hoc parallelism exploited via message passing primitives. Lithium [70], allows the programmer to modify/extend basic skeletons behavior via the standard Java OO class extension mechanism.

The third issue is a bit more subtle. A skeleton designed as a pure high-order function simplifies a number of points in the design of both the language and its run-time support; in addition it enables the smart compilation and the optimization of the parallel code [140, 18, 9, 17]. From the language viewpoint, it eases the nesting of skeletons via their functional interfaces (represented by arguments and results of functions). From the run-time viewpoint, it enables the structural assembling of implementation templates: a skeleton can be implemented by composing (in some way) implementations of called skeletons. Since user code is embodied into a

[1] As for example C/C++/Java code chunks.

wrapper skeleton and skeletons have no side-effects, in principle the run-time does not need a shared storage among skeletons. All programming environments we designed (except ASSIST) rely on these principles. For such kind of languages we eventually proved that any skeletal program admit a *normal form*, i.e. a semantically equivalent program obtained from the original program by means of source-to-source transformations and exposing a no worse (in many cases better) performance and speedup with respect to the original program (under mild additional requirements) [15]. In other words, reducing a skeletal parallel program to the normal form consists in to squeeze the skeleton nesting in a flat program in such a way it can be easily farmed out (while preserving the functional semantics).

Normal form reduction technique has been successfully used in the optimization engine of the Lithium programming framework (see also Section 2.3). On the other hand, the bare existence of the normal form led us to reflect upon the expressive power of such kind of languages. As a matter of fact, the normal form (whether applicable) suggests that *a program exploiting any skeletons nesting can be rewritten in flat,* farm*-based program with no performance slowdown.* In turn, this clearly suggests that skeleton languages admitting normal form reduction lack in expressive power. This ultimately springs from the limited number of interconnection patterns admitted among constructs (that makes normal form reduction applicable).

In particular, our skeletal programming frameworks have exhibited the strongest limitations when dealing with applications exploiting highly irregular access patterns to large data sets. A couple of cases are noteworthy:

i) The application data-set can be naturally described in a hierarchical fashion by a recursive or irregular data structure (as in the *C4.5* data-mining and clustering applications [35, 24]). In addition, the relationship among data items dynamically change along the application run (as in tree-based simulation methods [170]). These applications often visit data-structures in a Divide&Conquer fashion.

ii) Even if the application data-set may be statically bounded (e.g. stored in an array), the application access pattern to data items is highly irregular and cannot be statically predetermined (as in some computational chemical applications [64]).

Due to irregular and dynamically changing nature of such applications, a coherent shared address space programming model has been argued to have substantial ease of programming advantages for them, and also to deliver very good performances when cache coherence is efficiently supported in hardware. Our previous programming frameworks (P^3L, SkIE) do not natively support neither distributed dynamic data structures nor the Divide&Conquer skeleton. Nevertheless, in these languages the Divide&Conquer behavior may be mimed by means of (quite tricky) combination of the others skeletons. Some preliminary experiments using SkIE equipped with a

shared memory abstraction[2] have shown a expressivity boost in the deployment of application classes mentioned before [35, 48].

In summary, our skeletal programming environments have exhibited several shortcomings during the design of real world applications. These shortcoming mainly regard language expressive power, especially in the deployment irregular applications. From this point we moved over to design eskimo, a new skeletal programming framework exploiting a novel skeleton role in the language.

4.1 eskimo: A new skeletal language

eskimo [Easy SKeleton Interface (Memory Oriented)] is a parallel extension of a "host" language (the C language) based on shared address programming model. The target architectures for the language are parallel machines without native support for sharing memory among processing elements. In this setting, eskimo is conceived to be a framework to experiment the feasibility of the skeletal approach with dynamic data structures in parallel programming. eskimo run-time support is based on a software distributed shared memory, and allows the programmer to freely access data items in the shared memory. Notably it is not yet another DSM, it rather relies on DSM already known technologies to experiments the co-design of dynamic data structures and parallel programming patterns enforcing locality in the distributed memory access. We outline the main features of eskimo as follows.

Abstraction. eskimo is a skeleton based programming language. The core principle of the skeletal programming is neither particularly deep nor particularly complex. Skeletal programming would simplify programming by raising the level of abstraction, providing the programmer with performance and portability for its applications. In order to convey this simplicity to practitioners we must be careful not to bundle it with other conceptual baggage, no matter how natural this may seem from the perspective of the researcher [60].

At this end we enriched C language in such a way the language extension fairly raises the level of abstraction. The main sources of abstraction regard *data structures, the flow of control, and the interaction between them.* All abstractions rely on solid concepts like concurrency and abstract data types. A rather big effort has been spent to keep the gap between level of abstraction of C native programming model and extended environment as low as possible in order to *preserve the C programming pragmatics.* The basic idea is the programmer is not compelled to learn a new programming language but only the new primitives effect. Moreover, eskimo languages support does *not interfere* with C functions not using eskimo primitives. Such functions may use all C features (e.g. pointers) and standard libraries (e.g. pthread).

[2] Actually a simple non-coherent software DSM [31].

Expressiveness. As previously argued, the main reason motivating eskimo is the lack of expressiveness of almost all previous skeletal parallel programming frameworks [74, 27, 29, 17]. We propose a structured programming environment that allows the programmer to deal directly with (dynamic) shared data structures by means of language primitives. In particular, the programmer deals with an abstraction of data structures represented as single entities, even in case it is constituted by a collection of parts spread across the system. These parts are kept consistent by the run-time support following a (very) lazy memory consistency model, namely DAG consistency (see Section 3.3.2). The particular consistency model adopted enforces the high-level approach of the language since it enables to read/write data objects avoiding the need of explicit low-level synchronization primitives (like locks and barriers). In this setting the skeleton is no longer a ready-made object of the language (e.g. an high-order function), it is rather a code pattern build directly by the programmer using language primitives. Since the language does not force the programmer to use ready-made patterns but slightly lower level primitives integrated into the host language, ad-hoc parallel patterns may be coded using both eskimo and other libraries primitives. It greatly improves language interoperability with standard developing tools with respect to "classical" skeleton frameworks.

Framework and design principles. eskimo is specifically designed for loosely coupled parallel architectures, in particular Beowulf class clusters. Such architectures, that are becoming pretty popular due to their limited cost, present several difficulties in drawing good steady performance from applications (particularly dynamic ones). Following the nature of target architecture class, eskimo exposes to the programmer a (virtual) shared NUMA address space. However, the language does not expose to the programmer all parallel exploitation details. The main idea consists in supplying the programmer with a framework enabling him to make some decisions about the relationship among data structures or their sub-parts (as for example spatial locality) without forcing him to deal with low level orchestration of parallel activities (e.g. data and processes mapping, load balancing, etc.). The underlying design principle consists in considering preferable a programming environment on which performance improves gradually with increased programming effort with respect to one that is capable of ultimately delivering better performances but that requires an inordinate programming effort. This can consist in either programming each detail of the application (as in low-level approaches) or expressing the application attempting to use a fixed set of ready-made parallel paradigms. eskimo provides the programmer with language hooks to take advantage from a possible deep knowledge of application memory access patterns. In other words, eskimo programmers may applications as they like, but they know that some combinations of primitives achieve a better performance than others on the target architecture class.

Layered implementation design. eskimo is built on top of a hierarchy of run-time layers, each of them providing mechanisms and policies to solve some of parallel programming run-time support issues (i.e. data mapping, processes scheduling, shared memory abstraction, caching, etc). In particular flows of control mapping and scheduling as well as shared data structure mapping are implemented in the top tier, i.e. *etier-1*. At this level many information about the system status may be accessed (e.g. processing elements load, memory load, etc.). These information are maintained by a lower level tier (i.e. *etier-0*) that basically wraps the communication stack (i.e. the TCP stack in the current implementation) and provides a framework to manage pools of threads.

In summary eskimo extends the C language with three classes of primitives abstracting data structures (and their management) and parallelism exploitation:

1. flows of control management (Sections 4.1.1 and 4.1.2);
2. *Shared Data Types* declaration, allocation and management (Section 4.1.3);
3. shared variables read/write primitives (Section 4.1.4).

In the rest of the chapter we shall briefly describe these classes in order to provide an intuitive insight of the issues. Later on (Chapter 5) we shall formally return back on the same issues by describing the syntax of language primitives and exemplifying their use.

4.1.1 Exploiting parallelism in eskimo

The basic idea behind eskimo is that a programmer should concentrate on co-designing his data structures and his algorithms. Moreover, in order to obtain a high-performance application, the programmer ought structure its application properly, and eventually suggest to run-time important information about algorithm data access patterns. eskimo run-time takes care of all other details like process scheduling and load balancing.

The parallelism is exploited through concurrency. The minimal unit for concurrency exploitation is the C function. Just as in a serial program, an eskimo program starts as a single control flow, i.e. the *main* control flow. In any part of the program, the programmer may split the flow of control through the asynchronous call of a number of functions; such flows must, sooner or later, converge to a single flow of control. The basic primitives managing program flow of control behave like Dennis' fork/join [79], we call them *e-call/e-join*[3]. eskimo flows of control are called *e-flows*. These flows of control share a virtual memory address space. The relationship among *e-calls, e-joins* and *e-flows* is intuitively sketched in Figure 4.1 a).

[3]From now on we use the *e-* prefix to distinguish abstractions provided by eskimo.

e-call/e-join primitives enable the programmer to set up a dynamic and variable number of *e-flows*. This features is especially important in eskimo design. eskimo is basically a language to experiment the feasibility of the skeletal approach with dynamic data structures in parallel programming, in particular linked data structures as lists and trees. Almost all interesting algorithms using these data structures explore them in a recursive fashion, following the Divide&Conquer paradigm thus exploiting a variable concurrency availability along the program run lifespan.

Actually Divide&Conquer has been already present in classical skeleton sets. This skeleton, as others (e.g. scan and map), has been often interpreted as a collective operation on a given data structure by the skeleton community [18, 93, 95, 30, 38]. In the best case this originated a family of variants for each skeleton, each of them optimizing a particular behavior of the algorithm on the data structure [101]. In other cases, application programmers have been compelled to match a given shape of the data structure with a given behavior of the skeleton (that is likely to be a frustrating task). eskimo approaches the problem from a lower-level viewpoint: it enables the co-design of algorithms and data structures as in the very classical sequential programming [171]. There is no ready-made Divide&Conquer skeleton in eskimo, there are rather all ingredients to build it in such a way the programmer may express the suitable variant of the skeleton for its data structure. Then, the language run-time tries to understand from the ingredients and from the programmer hints what is the expected parallel behavior for the particular variant, and even in case hints are wrong or missing the program does not lose its correctness (even if we cannot expect from it an optimal performance).

4.1.2 Concurrency and flows of control

Actually *e-flows* do not necessarily match any concrete entity at eskimo run-time support level or its underneath run-time layers (as for example threads or processes). In particular an *e-call* has to be considered as the declaration of a "concurrency capability" with respect to a given function instance: an *e-called* function instance might be either concurrently executed or sequentialized with respect to the caller function. In the former case, the matching *e-join* represents the (last) point along the execution unfolding where a called *e-flow* must converge into the caller *e-flow*.

The two *e-flows* coming out from an *e-call* may be executed in parallel, interleaved or serialized in any order depending on the algorithm, the input data and the system status. Since *e-call/e-join* primitives denote in the algorithms points where it is possible to proceed in more than one way, even for the same input data set, they are *non-deterministic* primitives [83, 57]. At such points, the language compiler or run-time may chose to either parallelize or serialize (choosing their order) the resulting *e-flows*. As an example, eskimo program sketched in Figure 4.1 a) exploits four *e-flows* that represent its top concurrency degree. However, as shown in Figure 4.1 b) the language run-time might choose to halve the concurrency degree and project the four *e-flows* in two actual concurrent entities. Serialized *e-flows* do not

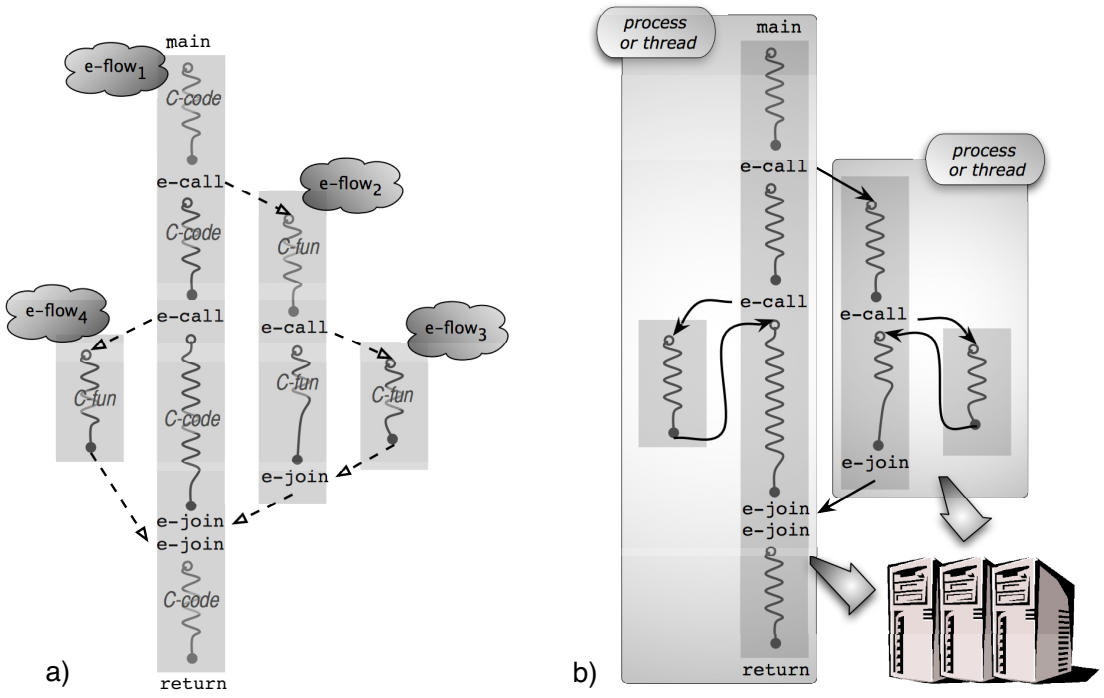


Figure 4.1: An eskimo program execution intuitive view. a) Relationship among *e-calls, e-joins* and *e-flows* (grey boxes). b) A possible execution of the program.

lead to any overhead related to parallelism exploitation (thread creation, activation, synchronization).

As it might be expected, *e-flows* have to be mapped on concrete concurrent entities (i.e. processes or threads) at the language run-time level. This mapping is sketched on-the-fly, step by step (in a distributed fashion) by eskimo run-time; in correspondence of an *e-call* the calling *e-flow* maps the new *e-flow*. Actually, the mapping process must answer two key questions:

1. Must the new *e-flow* be really concurrently executed (either in parallel or interleaved) or must it be serialized with respect to the calling one?

2. In case of concurrent execution, must the fresh *e-flow* be locally or remotely spawned?

In both cases the language run-time looks for a trade-off between opposite needs. In the former case, it tries to maintain the amount of concurrent flows in the system within acceptable bounds: enough to exploit the potential speedup of the system, but not too much in order to avoid unnecessary overheads (in time and memory space) due to parallelism management: this may be considered as a mapping problem. In the latter case, the run-time tries to balance workload on PEs while keeping data locality as much as possible: this is a scheduling problem.

We shall deeply discuss both mapping and scheduling of *e-flows* in Chapter 6; here we just would like to highlight that the relationship among *e-flows* is slightly more abstract than concurrency. Different *e-flows* encompass parts of an application which might be concurrent each other, but in some cases they are not. As an example consider the following two cases:

- the *e-flows* are called in a sequence that establish a (direct or indirect) precedence relation of one over another;
- the *e-flows* are mapped by the language run-time on the same concurrent entity of the operating system (process or thread).

In both cases *e-flows* do not correspond to actual concurrent code. We shall talk about the former case in Chapter 5 where we introduce the formal machinery to describe *e-flows* and their relationship. The latter case regards *e-flows* implementation policy we shall discuss in Chapter 6.

eskimo *e-call/e-join* are the basic primitives to create and destroy an *e-flow*. In addition to them, eskimo provides the programmer with their generalization, i.e. *e-foreach/e-joinall*. They work basically in the same way but, as shown in Figure 4.2 b), they can create and destroy an arbitrary number of *e-flows*. Since *e-flows* created by means of *e-foreach* have no data dependencies one each other, they can be non-deterministically executed. Complementary *e-joinall* non-deterministically waits for the completion of all *e-flows* created by the matching *e-foreach*. *e-foreach/e-joinall* have an additional freedom degree with respect to a sequence of *e-call/e-join*: the order in which *e-flows* are mapped/scheduled and joined. This turns in some additional advantages with respect to the basic case from the run-time viewpoint. In particular, at *e-foreach* time the language run-time knows how many and which *e-flows* have to be executed, and may choose their execution order depending on data availability. eskimo run-time uses the possibility to enhance locality of data accesses by running first tasks that are likely to have needed data (or a part of them) already present (or cached) in the PE. In the same way, the run-time non-deterministically joins *e-flows* in the order they complete. Therefore, the same program may be subjected to different mapping and scheduling decisions on different runs (even on the same input data). As an example, Figure 4.2 a) and c) sketch two runs of the same program that have been subjected to different mapping and scheduling decisions.

4.1.3 Sharing memory among flows of control

eskimo language is specifically thought for NUMA parallel computing frameworks, in particular for distributed memory architectures. These architectures naturally supply a very efficient memory access to local memory and a more expensive access to remote memory (through the network). The language abstracts these memory categories, but it does not obscure the differences between them. eskimo language

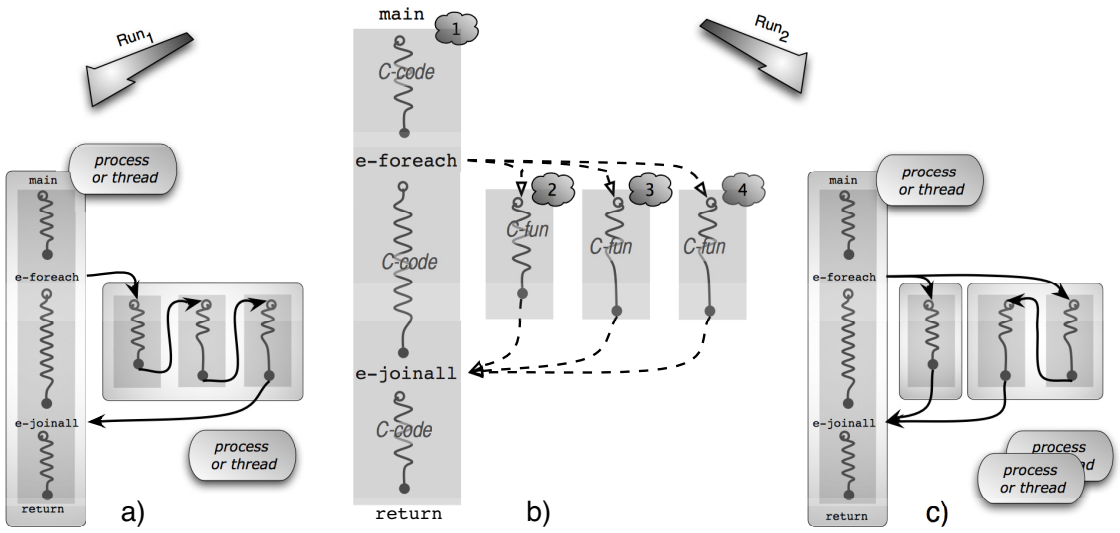


Figure 4.2: An eskimo program resulting in different mapping and scheduling in different runs.

exposes to the programmer two class of memory spaces: *private* and *shared*, which hold the respective classes of program variables.

Private and shared variables are distinguished by their types. Shared variables must have a *Shared Data Types* (SDT). SDTs are obtained by instancing a fixed set of *Shared Abstract Data Types* (SADT), i.e. parametric types, including spread arrays, spread k-trees and shared regions[4]. All variables with a different type are private. However, not all C variables are allowed as private variables, in particular C global variables are (with some exception) forbidden in eskimo programs. These must be substituted with shared variables. The relationship between *e-flows* and memory classes is the following:

- Each *e-flow* has its private address space; thus private variables are always bound to the *e-flow* where they have been declared. Private variables can be accessed without any eskimo (neither static nor run-time) mediation.

- All *e-flows* can access the shared address space, that is spread across the system. Shared variables may be accessed by several *e-flows*, thus might be subjected to concurrent accesses. These accesses are regulated by the eskimo run-time.

Overall, SADTs and their constructors provide a device to make sharable any C data type by building SDTs. These are data structure containers allowing the access to contained data (i.e. private or shared variables) by several *e-flows*. In case

[4]A contiguous region of memory that is shared but not spread.

of trees and arrays the container is structured and holds many contained data in a lattice; such data is distributed across the virtual architecture thus it may have a greater total size than memories of the single PE. Shared variables (or part of them) may be referenced across *e-flows* by using *references*, i.e. void pointers into shared address space. References are the basic mechanism to pass shared variables across *e-calls*. Pragmatically, eskimo references and shared variables would provide globally addressable data structures matching the same relation between C pointers and variables.

All SDTs are designed to be concurrently accessed by *e-flows*. SDTs may be statically or dynamically (and incrementally) allocated, in particular $k-$tree nodes must be dynamically allocated by means of a language primitive (see Chapter 5). Nodes of each tree are blocked in segments by the language run-time following programmer hints (as an example as an heap or in first-fit order), but not forcing him to directly deal with data blocking. Data structure allocation is always performed in the current processing element (even if all processing elements may access it). The programmer may require locality for two data structures (e.g. a tree node and an array) simply allocating them along the same *e-flow*.

Beyond trees, the programmer may build any shared linked data structure by using references. Shared variables obey to DAG consistency (see Section 3.3.2) and can be accessed through eskimo primitives (see next section). That basically means two different *e-flows* may have a non coherent view of a given address in the shared memory space; the coherence is then reconciled at *e-join* time. Pragmatically it means different *e-flows* must write different shared address locations.

4.1.4 Reading and Writing Shared Variables

Most software DSMs rely on MMU and operating system traps in order to make the access of remote data transparent. When a processor references an address that is not local, a trap occurs, and the DSM software fetches the chunk containing the address and restarts the faulting instruction, which now completes successfully (see Section 3.5). However, the interrupt cost, associated with receiving a message has been proved to be the largest component of the slow remote latency, not the actual wire delay in the network or the software implementing the protocol [145].

We followed a different approach: shared variables, differently from private ones, can not be read and written directly. In order to access to a shared variable the programmer must explicitly bind it to a private pointer. The technique avoids the use of signal handlers and operating system traps. In addition it has a pragmatic importance: since binding operations may be expensive the programmer is required to reduce their number by exploiting temporal/spatial locality in shared memory accesses[5].

[5]Anyway, transparency may be achieved by preprocessing/instrumenting the user code. The topic is further developed in Section 6.1.2.

eskimo provides a pair of language primitives enabling the access to a shared variable: `r` (for read-only) and `rw` (for read/write). Both `r` and `rw` take as argument a reference and return a void (private) pointer that can be used to access the (private) variable value. Such value may be the node of a tree, the cell of an array or the data value contained into a region. In all cases an explicit cast of the void (private) pointer to the correct type is required. The prototypes of the two primitives, their exact semantics and examples of use are shown in Section 5.2.2.

4.2 Skeletons and their expected pay-back

eskimo provides the programmer with a family of constructs that specialize *e-foreach*. Each of them run through elements of a given type of shared variables. Currently there are three types of *e-foreach* constructs:

e-foreach-child run through non-null children of a spread tree node;

e-foreach-cell run through cells of a spread array;

e-foreach-refset run through a given set of references to shared variables or elements of them (as for example the set of tree leafs, or array cells).

The *e-foreach* constructs introduce a form of data parallelism, which is based on domain decomposition principle. eskimo allows the programmer to dynamically define and decompose the domain. In fact, the parallel application of a function to (sub)set of children in a tree may be interpreted as a form of data parallelism (where the dynamic domain of children is decomposed). As a result, also Divide&Conquer paradigm may be interpreted as a special (dynamic, recursive) case of data parallelism.

eskimo offers to the programmer the possibility to store data using a flat (arrays) or hierarchical (trees) structure, then offers two standard parallelization paradigms for the two cases: forall (or map) and Divide&Conquer. The two paradigms may be freely interleaved and are both introduced at the language level by just one primitive (family), namely the *e-foreach*. In case the application is not obviously expressible as instances of proposed paradigms or their interleaving, the programmer may ad-hoc parallelize it by using the standard C language enriched with *e-calls*/*e-joins*.

The basic idea under eskimo is to abstract both (dynamic) data structures and application flow of control in such a way they result *orthogonalized*. Creating an *e-flow*, either an *e-called* function may be migrated to the PE holding the data it needs or vice-versa. Clearly, the principal decision is to evaluate which is the better choice in each case. In such task the run-time takes in account a number of elements: the system status (load balancing), the shared memory space status and the programmer hints. Several policies may be implemented by using such information. Since Beowulf class cluster exploits an unbalanced communication/computation power tradeoff, the run-time tries to schedule an *e-flow* on the same PE of the data it

needed and takes in account load-balancing only as secondary constraint. Clearly, the key issue here is to know in advance what data a function will access. The run-time uses two kind of information:

1. *The data blocking.* The run-time makes scheduling decisions only when a function parameters cross the current data segment boundary. As result *e-flows* accessing to data in segment are sequentialized enhancing locality and coarsening computation grain.

2. *The programmer hints.* We assumed that the first parameter of each *e-called* function (that must be a reference) is considered as a dominant factor, the run-time expects the majority of data allocated "close" to data referenced by it. A wrong/missing information has no impact on program correctness but a great impact on its performance.

eskimo pragmatics is more similar to sequential programming than shared memory parallel programming. This is a precise design issue that found its main motivation in the wish to experiment shared dynamic data structures in loosely coupled target architectures as Beowulf class clusters. As shown is Chapter 3, the literature is quite rich of techniques to turn a Beowulf cluster in a shared memory parallel machine. Actually, software distributed shared memory systems can create the illusion of a single shared memory across the whole cluster by means of a software run-time layer. Some DSMs relies on weak memory consistency models in order to hide some of the false sharing created by their coherence strategies. Some of these consistency models (LRC as an example) require an even stricter programming discipline in the use of synchronization primitives in respect of sequential consistency. Others are more friendly with the programmer but seriously lack in expressivity. As an example a quite simple program like quicksort cannot be easily written by using scope consistency memory model.

Let us take spatial locality as an example. Normally DSMs cannot control the shared memory at the granularity of the machine word. All of them group data in blocks (typically in pages) in order to reach an acceptable working grain for the architecture. Designing a parallel algorithm we should be sure that such process does not destroy spatial-locality, on the contrary we should turn data blocking into space-locality exploitation. At this end we should consider the peculiarities of dynamic data structures. Let us take trees as an example. Trees are rarely accessed in random way (as we can expect for arrays). Trees are often used to describe a hierarchically organized data set. In many cases the programmer will follow the tree structure, from the root down to leaves and vice-versa. We can spread tree nodes among processing elements, let us say, using a hash function, but is this a good organization for a tree? We believe we should respect the nature of tree. More ambitiously, we would like the programmer would express his insight of the algorithm by co-allocating data that he known expose a good temporal/spatial locality.

In this task the *e-flow* concept has a foremost significance. Currently eskimo run-time always assumes data structures (or their parts) allocated along the same *e-flow* as "spatially related" items, thus it tries to keep them close one another (within some other constraints like keeping a fair memory distribution).

In addition, programmers may rely on data-driven mapping/scheduling policies embedded in the *e-foreach* construct. Accessing a data structure they can use the *e-foreach* construct to describe the minimal synchronization requirements of the algorithm (i.e. the maximum concurrency) and rely on the language support to reduce the concurrency availability to the correct grain for the architecture as well as access to task mapping/scheduling that enhances locality in data accesses. This may be achieved moving data toward computation or vice-versa depending on the overall system status.

Overall, eskimo provides the programmer with some ready-made shared data types (trees, arrays, regions) and a method to build shared data structures that respect their nature and their typical access patterns. Also, the programmer may force some relationship (locality actually) among data structures (or their parts) using his insight of the algorithm by means of language hooks. As an example the same tree might be build in different ways depending of the typical access patterns the programmer expect to exploit on it. Program performance is then enforced by the eskimo run-time data-driven scheduling. As briefly described before, the language run-time tries to schedule *e-flows* on such a way an *e-flow* happens to be executed on a processing element that already hold data item (or its cached copy) that are likely to be accessed either because spatially related with some previous accessed data or due to a specific hint given by the programmer by means of language hooks.

Eventually, it is worth noticing that layered implementation of eskimo run-time support enables the skilled programmer to experiment several SDTs mapping and *e-flow* mapping and scheduling policies by operating at *etier-1* (the top tier) run-time level, that actually holds all mapping and scheduling policies.

4.3 Related work and discussion

eskimo programming framework is designed and developed from scratch but it has several analogies with a number of well-known research works. Among the others it is worth mentioning *Cilk* (see also Section 3.3.2) and *Athapascan* (see also Section 3.6). In particular, eskimo inherits memory consistency model from *Cilk*. The layered implementation design and the idea to make customizable the scheduling are quite similar to *Athapascan*. Moreover the *Athapascan* "task" concept is similar to *e-flow* one at the first glance. *e-flows* differently *Athapascan* may produce side effects according to DAG consistency. *Athapascan* "computes owner rule" scheduling policy should not be confused with eskimo behaviour: an *e-flow* running on a processing element may write other processor element data.

Neither *Cilk* nor *Athapascan* implements spread dynamic data structures as single entities (and their dynamic allocation). These are designed to be grouped in segments in order to reach an acceptable working grain for the target architecture class in a transparent way (see Chapter 6). As we shall see in Chapter 6, it is designed for loosely coupled architectures. It does not rely on any shared stack (e.g. cactus task) for function calls: the stack never moves across PEs and may be optimized by the standard C compiler (e.g. inlining). It does not use work stealing, that has load-balancing as main target. It tries to exploit a mix of data locality and load-balancing. In addition eskimo exploits skeletal primitives (the *e-foreach* primitives family, at least) enabling a *e-flows* scheduling that enanches locality of data accesses.

Chapter 5

eskimo: language usage

Readers' road-map. In this chapter we take again eskimo concepts intuitively presented in Chapter 4. In particular we describe and exemplify eskimo primitives syntax and pragmatics. In Section 5.1 we present the eskimo computational model. Then in Section 5.2 we present eskimo details: How to write an eskimo program; eskimo primitives dealing with data type abstraction; eskimo primitives dealing with parallelism exploitation. eskimo pragmatics is developed in Section 5.3 by means of a running example. eskimo is a pretty young programming platform and it is explicitly targeted to dynamic data structure experimentation in a skeletal framework. Therefore it is subjected to continuous modifications and improvements. We present here the current assessed status only, avoiding to mention possible improvements (that are already underway). eskimo languages have been designed and developed by the author himself. Part of this chapter will appear in [10].

eskimo [Easy SKeleton Interface (Memory Oriented)] is a *skeletal* extension of the C language for parallel programming. The primary aim of eskimo is to experiment the feasibility of the skeletal approach in parallel programming with dynamic data structures. Shared memory programming has been argued to have substantial ease of programming advantages for this class of problems. We present eskimo library which constitutes an attempt[1] to merge the two programming models by introducing skeletons in a shared memory framework. At this end we take a step back with respect to the classical interpretation of skeletal programming (in our opinion moving back to their original meaning [59, 60]). *Skeletal programming would simplify programming by raising the level of abstraction, providing the programmer with performance and portability for its applications.* Indeed, eskimo abstracts the shared address programming model. It provides the programmer on one hand with data structure abstraction (SADT, SDT) and constructs to deal with their management in a distributed shared address space; on the other hand with flow of control abstraction (*e-flows*) and constructs to deal with their management (*e-call*/*e-join*, *e-foreach*/*e-joinall*). These constructs abstract both data mapping and process

[1] The first attempt, at the best of our knowledge.

mapping/scheduling. For this reason they can be considered "skeletons" (even if we call them construct to distinguish them from classical skeletons). Constructs may be easily composed to implement paradigms in the classical skeleton set such as map, reduce, Divide&Conquer.

It is worth remarking that abstraction is not an exclusive prerogative of the skeleton research community. eskimo is pretty similar to other languages exploiting multithreading in a shared address space such as *Cilk* and *Athapascan* [43, 87], besides eskimo has been designed with previous research experiences in mind. Actually, eskimo inherits memory consistency model from *Cilk* (*Athapascan* relies on entry release consistency, see also Chapter 3). eskimo like *Athapascan* (unlike *Cilk*) is thought for loosely coupled NUMA distributed memory systems and exploits a layered implementation design. Neither *Cilk* nor *Athapascan* can cope with dynamic data structures presented as single entities at the language level. eskimo, unlike its predecessors, abstracts both (dynamic) data structures and application flow of control in such a way that they result *orthogonalized*. Either function execution and shared data may migrate one towards the other with the aim of improving memory accesses locality. This is actually the ultimate eskimo aim.

In the next sections we first describe the computation model of eskimo language, then eskimo in details. Afterwards a running example is presented in order to explain eskimo features.

5.1 eskimo computation model

We briefly recap main eskimo concepts introduced in Chapter 4. eskimo C language extension provides a high-level parallel programming environment. Parallelism is exploited through concurrency. The minimal unit for concurrency exploitation is the C function. Just as in a serial program, an eskimo program starts as a single flow of control, i.e. the *main* control flow. In any part of the program, the programmer may split the flow of control through the asynchronous call of a number of functions; such flows must, sooner or later, converge to a single flow of control. eskimo flows of control are called *e-flows*. The basic primitives managing program flow of control behave like Dennis' fork/join, and we call them *e-call/e-join*. eskimo flows of control are called *e-flows*. Their n-way extensions are named *e-foreach/e-joinall*; each of them representing a family of primitives that distinguish one another for the type (and the concrete behavior). As discussed in Section 4.1.2, our primitives do not manage directly actual concurrency (threads or processes), they only describe the (partial) order among instances of program functions.

We call *e-functions* the C functions that can be *e-called*. These are standard C function with two additional requirements (extensively described in Section 5.2.3):

1. The function prototype is fixed, with regards to return and formal parameters type.

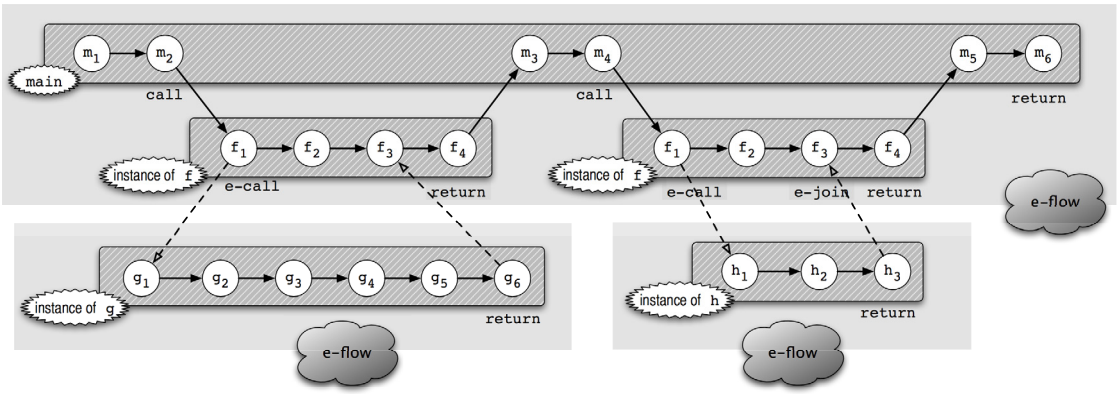


Figure 5.1: An eskimo computation described by the *e-flow* graph. Dashed arrows highlight *e-calls*/*e-joins* and solid arrows highlight standard C function calls.

2. The first statement of the function is a language primitive (that basically initializes the function run-time support)

An *e-flow* is basically an *e-called* (i.e. asynchronously spawned) *e-function* instance. An eskimo computation can be viewed as graph that unfolds dynamically. The graph representation models very intuitively the dynamic evolution of flow of control in a program.

Definition 2 (*e-flows* graph) *The flow of control flow $G = (V, E)$ of an* eskimo *program is a directed graph where executed instructions are vertices of the graph. Vertices are connected each other by two kinds of oriented edges:*

1. Seq-edges *representing the execution order of program instructions. Calls and returns to and from functions are also represented by seq-edges.*

2. e-edges *representing* e-calls *and returns from* e-calls*.*

In this graph each path from two connected e-edges *is an* e-flow*.*

The *e-flow* graph represents program instructions and the order they are executed at the grain of the source program unit-size instructions. Since C functions are the basic objects for concurrency exploitation, vertices belonging to the same function instances can be grouped together in macro-vertices labeled with the function name. It is worth noticing that, since the graph sketches the flow of control among *e-functions* instances, the graph is acyclic, thus it is a DAG. The *e-flows* DAG describes the *e-flows* dependencies only; shared data synchronizations do not appear on it. However, since DAG consistency defines shared data dependencies on the ground of dependencies among flow of control, *e-flows* DAG implicitly gives

information about data synchronization also. Intuitively a read can "see" a write in the DAG consistency model only if there is some serial execution order consistent with the *e-flows* DAG in which the read "sees" the write. DAG consistency allows different reads to return values that are based on different serial orders, but the values returned must respect the dependencies in the *e-flows* DAG (see Section 3.3.2). Therefore we can claim what follows:

Lemma 1 (Independent *e-flows*) *Let e_1, e_2 two vertices of the* e-flows *DAG (i.e. two C instructions). If it does not exist any direct path along the* e-flows *DAG including both vertices, then e_1 and e_2 never lead to shared data dependencies. Therefore,* e-flows *including e_1 and e_2 have the same propriety. This basically means the two* e-flows *can independently complete their execution.*

Figure 5.1 shows an example of *e-flow* graph. The modeled program, starting from the `main` function, calls the function `f` twice (in sequence); the first function `f` instance *e-calls* the function `g`, while the second function `f` instance *e-calls* the function `h`.

As described in Section 4.1.3, eskimo provides the programmer with two classes of memory spaces: *private* and *shared* (where private is referred to the *e-flow*). An *e-flow* is always executed in a sequential fashion and bound to a given processing element. Therefore, we can safely assume that *private* address space is always implemented using the most efficient memory the hardware supplies to the operating system. Accesses to private memory do not need any run-time mediation, thus they are both very efficient and fully compliant to any C language memory access implementation.

As far as shared data concern, eskimo level does not provide the programmer with direct view of processes and processors, therefore in this setting it makes no sense arguing about where exactly shared data are placed; the programmer only knows that they are (possibly spread) shared data and therefore accessing to them may be expensive. Also, *e-flow* concept has important properties with respect to shared address space. As an example two shared data items that have been allocated along the same *e-flow* are allocated "close" one each other (on the memory of the same processing element). A function accessing to one of them may probably access to the other cheaply due to the scheduling policy, that tries to execute an *e-flow* on the processing element holding the data.

5.2 eskimo language

The eskimo language extension is a parallel skeleton language. The basic idea behind eskimo is that a programmer should concentrate on structuring his data structures and his (possibly non-deterministic) algorithms. Moreover, in order to obtain a high-performance application, the programmer ought to structure its application properly, and eventually enrich the program with important information about algorithm data

access patterns (via language primitives). eskimo run-time takes care of all other details like process scheduling and load balancing.

An eskimo program is not parallel ab initio. The programmer may split the flow of control by means of language primitives.

We envision a C program as a sequence of logical phases. Many applications might be thought as sequences of phases. As an example consider an application that in sequence 1) takes an input from a file and stores data into a data structure 2) makes an elaboration of data 3) sends results to an output device. Parallelism would be not necessarily useful in all phases of an application, or rather it is a source of either useless complexity or interference with other libraries. Consider as example the case where input data come from a single entry point (file, channel, memory, etc). It is pretty clear that parallelism may be useless in the first phase.

In the next section we will go into language details and we describe: how to set up language run-time support, how language extends C types and variables, how global variable can be read and written, and how language deals with parallelism. We will not describe any implementation detail, that is discussed in Chapter 6.

5.2.1 Writing and running eskimo programs

The eskimo language is implemented as a library that extends C language. At first glance an eskimo program is simply a C program that calls eskimo primitives. The programmer must include the `eskimo.h` header file, compile the program using a C compiler and link the object code[2] to the library archive file. Actually, an eskimo parallel program is designed for multicomputer architecture (e.g. a cluster of workstations). Compiling an eskimo program a single executable file is produced; in order to run it on a multicomputer we rely on a specialized launcher program called `erun`. `erun` is the eskimo analogous to `mpirun` command of MPI programs. `erun` requires a (ASCII) machine file to specify target processing elements and some command line parameters like the number of processing elements and the program name.

The whole process is shown in Figure 5.2 by means of a terminal working session snippet. First the "hello.c" program is shown. It simply echoes command line arguments onto the terminal. Notice the program has no *e-calls*, thus it is purely sequential. Nevertheless, since it includes `eskimo.h`, it is an eskimo program and must be linked to the appropriate libraries. eskimo library is implemented using POSIX threads and some floating pointer functions, therefore any eskimo program needs to be linked with `libeskimo`, `libm` and `libpthread` libraries. Once compiled, a program can be launched using `erun`, provided an appropriate machine file exists. In Figure 5.2 four processing elements are indicated in the machine file; for each processing element the number of processors[3] and the amount of memory may be

[2]Currently eskimo is implemented as static library. However, it may be converted into a dynamic shared library.

[3]In the case the processing element is actually an SMP box.

```
montecristo:~/Test> cat hello.c
#include "eskimo.h"
/* includes, defines, typedefs */
/* Global Data Types declarations */

int main(int argc, char **argv) {
  int i;

  for(i=1;i<argc;i++) {
    printf("%s ",argv[i]);
  }
  printf("\n");

  return(0);
}

montecristo:~/Test> make hello
gcc -D_REENTRANT hello.c -o hello -lm -lpthread -leskimo

montecristo:~/Test> cat machines
# machine_name          n_cpu  memory_MB
  cassiopea.di.unipi.it   2      512
  andromeda.di.unipi.it   2      256
  capraia.di.unipi.it     2      512
  icaro.di.unipi.it       1      256

montecristo:~/Test> erun -v -m machines -n 2 hello Verbose Hello Word
STARTING: montecristo -> cassiopea [1] hints: [n_cpu 2] [mem 512 MB]
STARTING: montecristo -> andromeda [0] hints: [n_cpu 2] [mem 256 MB]
------------------------------------------------------------
Verbose Hello Word
------------------------------------------------------------
SHUTTING DOWN: Process terminated:[0] on andromeda (exit: 0)
SHUTTING DOWN: Process terminated:[1] on cassiopea (exit: 0)

montecristo:~/Test> erun -m machines -n 4 test Hello Word
Hello Word
```

Figure 5.2: Writing, configuring, compiling and running an eskimo program.

specified as optional hint to the run-time support. A number of these may be selected using the `-n num` option of `erun`; in case the file does not contain sufficient processing element names, they will be cyclically replicated. Selected processing elements constitute the (virtual) distributed architecture to run eskimo programs that have the sum of parallelism degree of each processing element as total parallelism degree (see also Section 6.1). As we shall see, `erun` and the library run-time support will set up the (virtual) distributed architecture by means of a completely connected network among processing elements. In this network, the first processing element acts as master element, while the others act as slave elements. The master processing element – running on the first machine of the machine file – is delegated to execute at least the main *e-flow*. At the language level the master element is a distinguished element, it is the only processing element we can associate with a given machine name. All resources (non-NFS files, devices, GUI, etc.) accessed along the main *e-flow* are referred to the master element. It is worth noticing that the total parallelism degree of the virtual distributed architecture represents an upper bound for the exploited parallelism in the program. In Figure 5.2 a couple of runs are shown: the former uses two out of four processing elements (`-n 2`), the latter uses all processing elements (`-n 4`). Observe the execution is independent from parallelism degree in both cases since the program exploits a single *e-flow*, therefore it exploits no parallelism.

Observe that since "hello.c" is a fully sequential program (see Figure 5.2) there is no need to activate eskimo support. In the general case eskimo support can be activated and terminated by using the following primitives: `e_initialize()` and `e_terminate()`. The eskimo support can be activated and terminated just once in a program. All language primitives except declarations can be called only if the support is active.

5.2.2 Types and variables

eskimo extends C types introducing *Shared Abstract Data Types.* These types constitute the building blocks of eskimo distributed data structures. In the following we will discuss shared abstract data types and their instances, i.e. *Shared Data Types.* Afterwards we introduce *shared variables*, i.e. variables having a shared data type. Eventually, we discuss *references* to shared variables and we discuss how programmer can deal with all the presented objects.

Shared Abstract Data Types

A *Shared Abstract Data Type* (SADT) is a parameterized type, i.e. a (simple) type template. In order to be used, a SADT must be instanced through the proper constructor into a *Shared Data Type* (SDT). The first parameter of the constructor is the name of the SDT. Instancing a SADT constitutes a *declaration* of a SDT. There can be many SDT declarations, provided they have different names.

Type constructors

Abstract syntax	Concrete syntax
`tree_t` $\equiv \mathcal{T}\langle$`node_t, int k_sons`$\rangle$	`e_declare_tree(tree_t, node_t, k_sons)`
`array_t` $\equiv \mathcal{A}\langle$`elem_t, int n_memb`$\rangle$	`e_declare_array(array_t, elem_t, n_memb)`
`region_t` $\equiv \mathcal{R}\langle$`elem_t`$\rangle$	`e_declare_region(region_t, elem_t)`

Table 5.1: Type constructors for spread trees, spread arrays and shared regions.

Since the C language framework does not natively support parametric types,we introduce two syntaxes for SADT constructors in order to nominate them: the concrete and the abstract syntax. Concrete syntax is the real language syntax for constructors. The application of a concrete constructor in a program constitutes a type declaration. The motivation under abstract syntax copes with the need of a handy and brief notation to describe language type system and type of parameters in function prototypes.

We define three SADT: *spread trees, spread arrays* and *shared regions*. Each SADT has its constructor. We denote SADT constructors in abstract syntax with $\mathcal{T}(trees), \mathcal{A}$ (arrays) and $\mathcal{R}$ (regions). Overall, a eskimo legal types τ may be summarized as follows:

$$\tau ::= \langle any\ C\ type \rangle \mid \mathcal{T}\langle \tau, \texttt{int} \rangle \mid \mathcal{A}\langle \tau, \texttt{int} \rangle \mid \mathcal{R}\langle \tau \rangle$$

Abstract and concrete syntax of SADT constructors are described in Table 5.1. Spread trees abstract the distributed tree concept, while spread arrays the distributed array one. Shared regions abstract the opposite concept: a contiguous region of memory that cannot be spread on memory. A shared region acts as data container for any C variable, making it accessible out from the single *e-flow*.

Let us take trees as example. The spread tree SADT must be instanced using `e_declare_tree` constructor. `e_declare_tree(tree_t,node_t,k_sons)` adds a SDT named `tree_t` to program types. It represents a spread k-tree instance where `node_t` is the type of the tree node and `k_sons` is the number of children of each node. `node_t` may be any C type, including references and already declared SDT.[4] `e_declare_tree` must be used by the programmer as a type declaration; from the `e_declare_tree` on and accordingly with C scope rules, `tree_t` may be used as legal type. A binary spread tree type holding `int` values may be declared as follows:

```
e_declare_tree(binary_tree_t,int,2);
```

Instances of spread arrays and shared regions are introduced in the same way using `e_declare_array` and `e_declare_region` respectively.

[4]SDT nesting is legal, even if pragmatically not advised for performance reasons.

Static initialization	Dynamic initialization
`TREE_INITIALIZER`	`e_tree_init(`$\mathcal{T}\langle\tau,$ `k`$\rangle$ `*the_tree)`
`ARRAY_INITIALIZER`	`e_array_init(`$\mathcal{A}\langle\tau,$ `k`$\rangle$ `*the_array)`
`REGION_INITIALIZER`	`e_region_init(`$\mathcal{R}\langle\tau\rangle$ `*the_region)`

Table 5.2: Static and dynamic initializer for spread trees, spread arrays and shared regions. $\mathcal{T}\langle\tau, k\rangle$, $\mathcal{A}\langle\tau, k\rangle$, $\mathcal{R}\langle\tau\rangle$ are type variables in abstract syntax.

Private and shared variables

Program variables are distinguished in two classes: *private* and *shared*. They are name-value bindings into respective memory address spaces:

- Each *e-flow* has its private address space; thus private variables are always bound to the *e-flow* where they have been declared. Since they are not subjected to concurrent accesses they can be accessed without any eskimo (neither static nor run-time) mediation. They are fully managed by the underlying C compiler.

- All *e-flows* share a unique shared address space. Shared variables may be accessed by several *e-flows*, thus they might be subjected to concurrent accesses. These accesses are regulated by the eskimo run-time (according to DAG consistency). The shared address space is spread across the distributed memory of the system.

From the language syntax viewpoint, all variables declared to have a Shared Data Type are shared variables. All other variables are private variables. However, eskimo imposes some constraints on the use of standard C variables. These are due to the fact each *e-flow* has its own private address space, thus a private variable cannot be safely accessed in different *e-flows*. In particular:

- C (private) global variables cannot be referred within any *e-flow* different from the main *e-flow*. A conservative rule consists in completely avoiding the use of private global variables. These can be substituted with shared global variables.

- C (private) automatic and heap (malloc'd) variables must be referred within the same *e-flow* where they are declared. Pointers cannot be used to pass the address of a private variable to a function living in a different *e-flow*. As we shall see, this programming paradigm must be implemented using shared variables.

- C (private) automatic static variables are completely forbidden.

```
eref_t  e_add_node(T<τ, k> tree, eref_t father, int n)
void    e_del_node(T<τ, k> tree, eref_t father, int n)
void    e_setchild(T<τ, k> tree, eref_t father, int pos, eref_t child)
eref_t  e_getchild(T<τ, k> tree, eref_t father, int pos)
```

Table 5.3: Primitives for spread trees SDTs.

Currently eskimo weakly enforces the respect of constraints on private variables. As discussed in Section 8.3, this is mainly due to current implementation design choices. In particular, the incorrect use of a (private) global variable does not trigger a compile time error. As private automatic variables concern, the language forbids their sharing by means of the *e-call* primitive formal parameter type. The only way to create an *e-flow* is by means of an *e-call* primitive family. These primitives may accept references as formal parameters.

Shared variables must be declared as standard C variables. Both static and dynamic allocation are allowed. Unlike C variables, these variables must be initialized before being used; the initialization step actually turns a C standard variable into a shared variable. Statically allocated shared variables must be initialized using a proper constant, while dynamic initialization is enabled via initialization functions. Constants and functions for shared variables initialization are shown in Table 5.2.

Since shared variables may be accessed within different functions, and some of corresponding function instances may belong to different *e-flows*, shared variables can be concurrently accessed. In this case a consistency issue arises. Shared variables are computation consistent: values stored in variables depend on the logical dependencies among instructions, not on the processor that happens to execute them. In particular they are DAG consistent [41], and can be accessed through eskimo primitives. That essentially means two different *e-flows* may have a non-coherent view of a given address in the spread memory space; the coherence is then reconciled at *e-join* time. From the programming pragmatics viewpoint it means that different *e-flows* must write different shared address locations (see also Section 3.3.2, page 102).

Let us describe how to declare a shared variable by means of an example. According to the previous SDT declaration (see page 134), a couple of shared variables may be declared as follows:

```
/* declare the SDT named ``binary_tree_t'' */
/* i.e. a 2-tree holding int values        */
e_declare_tree(binary_tree_t, int, 2);

/* statically declare&init a shared var */
binary_tree_t t1 = TREE_INITIALIZER;
/* statically declare a private pointer to binary_tree_t vars */
```

```
binary_tree_t *t2;
 ...
/* dynamically alloc a candidate shared var */
/* it is private up to init time            */
t2 = (binary_tree_t *) malloc(sizeof(binary_tree_t));
/* turn t2 pointed var into a shared var    */
e_tree_init(t2);
```

where `t1` is statically declared, allocated and initialized, while `t2` pointer enables the dynamic allocation and initialization by means of the referred shared variable `*t2`. The `t2` pointer is a private variable instead. Both `t1` and `*t2` represent a binary spread tree holding a `int` value in each node. After the initialization, the trees are empty and they would be dynamically populated with the root and the rest of nodes. The complete list of primitives working on spread trees is shown in Table 5.3.

References

Besides shared data types, eskimo extends C types by providing `eref_t` type, i.e. pointer to shared variable type. References are declared as standard C variables. References are used to refer to an arbitrary shared variable (they are "pointer to shared variables"). References are "void" pointers: each of them can refer any shared variable irrespectively of the type. `eref_t` is not a shared data type, but a standard C type, therefore references are natively private variables. As other private variables, they can be contained into a SDT becoming parts of a shared variable.

As an example, we can define a (spread, shared) binary tree holding a (shared) linked list in each node as follows:

```
typedef struct {
  int foo;
  eref_t next;        /* reference */
} list_cell_t;

e_declare_tree(binary_tree_ll_t,list_cell_t,2);

binary_tree_ll_t t1 = TREE_INITIALIZER;
```

References can be initialized, copied, compared one another and converted into standard C pointers via language primitives. A reference can be initialized and copied as standard C variables (through the "=" assignment). They can be compared one another by means of the function

```
int e_cmp(eref_t ref1, eref_t ref2)
```

that returns an integer equal zero if the two references are equal. They can be converted into standard C pointers by means of `r/rw` language primitives (presented later on in the section). A reference is an opaque object, it contains a shared memory address but it cannot be directly managed by the programmer, in particular references do not admit any arithmetic operation. `E_NULL` is a distinguished value for references; pragmatically its role is identical to C `NULL` constant. As usual in C frameworks, the programmer must directly guarantee the validity of a reference.

Reading and writing shared variables

Differently from private ones shared variables cannot be read and written directly. eskimo provides a pair of language primitives enabling the access to a private variable: `r` (for read) and `rw` (for read/write). Both `r` and `rw` take as argument a reference and return a void private pointer that can be used to access the shared variable value. Such value may be the node of a tree, the cell of an array or the data value contained into a region. In all cases an explicit cast of the void private pointer to the correct type is required (see also Section 8.3). The prototypes of the primitives are the following:

```
void *r(eref_t the_reference)
void *rw(eref_t the_reference)
```

Figure 5.3 a) explains the use of the two primitives. In the program we can recognize the following key steps:

1. A binary tree SDT holding `int` values is declared (line 3);
2. a `binary_tree_t` shared variable is declared and initialized (line 4);
3. the `t1` tree is populated with the root node (line 11);
4. the `node_body` private variable is linked (through an appropriate type cast) in read/write mode to datum contained into the root node (line 12);
5. the root node is first written, then read (lines 13 and 14).

It is worth observing the `node_body` private pointer is bound once to the root node and used twice (to write and read the contained data). In general the following rule holds: a private pointer linked to a shared variable keeps the validity along its scope until:

- An eskimo function is either called or *e-called*;
- k `r/rw` primitives with other shared variables as parameter are called. In other words, the support can maintain alive k links between shared and private variables; links are removed in FIFO order. The constant k is a parameter of the runtime support.

Pragmatically, the most conservative way to deal with the rule is to pair up a `r/rw` primitive with each read/write access to the private pointer. A more relaxed (and effective) approach may be followed, especially in cases of complex expressions. As an example, line 15 can be legitimately substituted with `*node_body += (*node_body % 3)+1`. The concept is exemplified in Figure 5.3 b), in particular at line 14 two active private pointers are deferenced.

5.2.3 Exploiting parallelism

Just as in a serial program, an eskimo program starts as a single flow, i.e. the main *e-flow*. In any part of the program, the programmer may create other *e-flows* by *e-calling* a number of *e-functions*. As discussed, not necessarily the whole possible parallelism is exploited, the language run-time retains the faculty to exploit only the parallelism it feels useful (according to criteria exposed in Section 4.1.2). Clearly, the run-time can drain away the parallelism degree, not increase it beyond the number of *e-flows* available. Pragmatically, this means that programmers are required to exploit all the sources of "concurrency" in the application they feel useful. As discussed in Section 4.1.2 eskimo does not enable the programmer to deal with actual concurrency, but it rather enables them to denote activities that have a "concurrency capability". These are denoted by *e-calling* eskimo functions (see next section). At the run time these operations may be seen as the call of a number of asynchronous functions that split the flow of control in a number of *e-flows*. Eventually some *e-flows* may match actual concurrent/parallel activities.

eskimo functions

An eskimo function (*e-function*) is a C function that might be concurrently executed on the eskimo virtual (parallel) architecture. An eskimo function must be defined according to a fixed schema. It has a reference as return value and three arguments; the first one is a reference, the second and the third are strictly linked each other and represent the pointer to a value and the size of the referred value respectively. The prototype is the following:

```
eref_t fun_name(eref_t the_reference, const void *arg, int arg_size)
```

the type `efun_t` is an alias for eskimo functions type. The second and third parameters provide a mechanism to pass a generic C value to the function; it is actually the only way to directly pass a private value to an eskimo function. The value may have any C type, but it is required to be contiguous in the private address space. The passed value must be used as read-only value in the function. Currently eskimo does not perform any compile time check on that. Pragmatically a type cast of the `arg` void pointer in to a `const` pointer may help in enforcing the correct read-only use of referenced values.

a) *read and write*

```
#include "eskimo.h"

e_declare_tree(binary_tree_t,int,2);  /* SDT declaration */
binary_tree_t t1 = TREE_INITIALIZER;  /* shared variable decl. */

int main(int argc, char **argv) {
  int *node_body;
  eref_t root;                        /* a reference declaration */
  e_initialize();                     /* start&init the run-time */
  /* E_NULL point out that we are creating root of the tree */
  root = e_add_node(t1,E_NULL,0);     /* populate t1 with the root */
  node_body = (int *) rw(root);       /* link node_body to root */
  *node_body = 7;                     /* put 7 in the node body */
  printf("Root node of t1 tree contain %d\n",*node_body);
  e_terminate();                      /* finalize&stop run-time  */
  return(0);
}
```

b) *read and write II*

```
#include "eskimo.h"
e_declare_tree(binary_tree_t,int,2);
binary_tree_t t1 = TREE_INITIALIZER(2), t2 = TREE_INITIALIZER(2),
  t3 = TREE_INITIALIZER(2);

int main(int argc, char **argv) {
  int *node_body1,*node_body2;
  eref_t root1,root2,root3;
  e_initialize();                    /* start&init the run-time */
  root1 = e_add_node(t1,E_NULL,0);   /* return a t1 node  */
  root2 = e_add_node(t2,E_NULL,0);
  root3 = e_add_node(t3,E_NULL,0);
  node_body1 = (int *) rw(root1); *node_body1 = 1;
  node_body2 = (int *) rw(root2); *node_body2 = 2;
  *((int *) rw(root3)) = *node_body1 + *node_body2;
  printf("The root node of t3 contain %d\n",*((int *) r(root3)));
  e_terminate();
  return(0);
}
```

Figure 5.3: Reading and writing shared variables.

```
e_call(ehandler_t *h, efun_t f, eref_t first, void *pp, int pp_size)
e_join(ehandler_t *h, eref_t ret)
```

```
ehandler_init(ehandler_t *h, int n)
```

Table 5.4: *e-call* and *e-join* primitives. In addition the primitive to initialize eskimo handlers.

In addition, any eskimo function is required to call the `efun_init()` function as the first instruction. A simple example of eskimo function (`print`) is shown in Table 5.4; observe that `print` function is called as a standard C function. Despite being an *e-function*, it will be executed as a standard C function. In this case the `efun_init()` primitive has no effect.

Exploiting concurrency

eskimo programmer deals with *e-flows* by means of two classes of primitives: *e-call*/*e-foreach* and *e-join*/*e-joinall*. Primitives belonging to the former family split a flow of control, ones belonging to the latter class join many flows of control (see Section 4.1.2). Each primitive from the former class must be followed by the suitable primitive of the latter class within the same *e-function*. An eskimo function cannot safely use the return value of an *e-called* function until it executes an *e-join* statement. *e-join* statements passively wait for all related *e-flows* to complete. The *e-join* statement acts as a local "barrier", not as a global one (as sometimes used in message-passing programming). In eskimo, an *e-join* waits only for the *e-called* children of the function to complete, not for the whole world. When all of its related functions return, the execution of the *e-join*'s *e-flow* resumes at the point immediately following the join statement.

Table 5.4 shows the syntax of *e-call* ad *e-join* primitives. `e_call` may be used anywhere within an *e-function*. The primitives need the following parameters:

`ehandler_t *h` It is (the pointer to) an handler enabling the matching between an *e-call* and *e-join*. The `ehandler_t` is an eskimo defined type. Handlers are not shared variables and must be defined as automatic variables. They must be initialized by using `ehandler_init` primitives that need an array of handlers and its length as parameters.

`efun_t f` It is the pointer to the *e-function* to be *e-called*. `efun_t` is the *e-function* type defined in the previous section.

`eref_t first` It is a reference to be used as the first actual parameter of `f`.

`void *pp` and `int pp_size` They are the second and the third actual parameters

```
montecristo:~/Test> cat print.c

#include "eskimo.h"

typedef struct{
  int n;
  int data[10];
} vector_t;

eref_t print(eref_t dummy, void *arg, int argsize) {
  int i;
  /* *arg is a read-only value. Declaring v as const helps */
  /* in preventing silly errors.                           */
  const vector_t *v = (vector_t *) arg;
  efun_init(); /* This makes print an eskimo function */
  for (i=0;i<v->n;i++)
    printf("%d ",v->data[i]);
  printf("\n");
  return(E_NULL);
}

int main(int argc, char **argv) {
  int i;
  vector_t v;
  e_initialize();  /* Activate and initialize the run-time support */
  for(i=0;i<5;i++)
    v.data[i]=-i;
  v.n=5;
  print(E_NULL,&v,sizeof(v));
  e_terminate();   /* Finalize and switch-off the run-time support */
  return(0);
}

montecristo:~/Test> make print
gcc -D_REENTRANT print.c -o print -lm -lpthread -leskimo

montecristo:~/Test> erun -m machines -n 1 print
0 -1 -2 -3 -4
```

Figure 5.4: A simple eskimo program.

for `f`. They represent the private pointer to a private variable[5] and its size respectively.

The *e-foreach*/*e-joinall* primitives introduced in Section 4.2 are actually implemented in the language by means of a sequence of primitives. These are shown in Table 5.5. In order to use an *e-foreach*/*e-joinall* pair the programmer has to code the following paradigm:

```
/* declare an iterator */
eiterator_t it;
/* initialize it - this is an iterator over a k-tree node children */
eiterator_init_child(it,< a k-tree type here >);
...
e_foreach(it, the_ref) { /* the_ref must be a node of a k-tree */
  < possibly set pp or change *pp >
  e_callit(f , it, pp, sizeof(pp));
}
...
e_joinall(it,ret_array);
```

The only noteworthy difference with respect to the *e-call* is that eskimo handler has been substituted for an *iterator*. It embeds both loop index and the handler. The kind of foreach (see also Section 4.2) is chosen by selecting a suitable initialization for the iterator. Variants of the `e_foreach` primitives are: `e_foreach_child`, `e_foreach_cell`, `e_foreach_refset`. These behave like `e_foreach` but statically enforces the use of the correct iterator.

`foreach_child` runs through non `E_NULL` children of a spread tree node;

`foreach_cell` runs through cells of a spread array;

`foreach_refset` runs through a given set of shared variables or elements of them (as for examples the set of tree leafs, or array cells). Items of the set are specified by means of a reference to it. Sets of references are currently arrays of references.

The *e-foreach* concept might also be introduced in the language by using just one primitive instead of a sequence of primitives. The sequence of primitives solution provides the programmer with a greater flexibility since it enables the change of the second and third parameter across the unfolding of the *e-foreach*. Anyway, the one primitive solution may be achieved by collecting needed code into a parameterized macro.

[5]It must be contiguous in memory.

```
e_foreach(eiterator_t it, eref_t the_ref)
e_joinall(eiterator_t it, eref_t ret)
e_callit(efun_t f, eiterator_t it, void *pp, int pp_size)
```

```
eiterator_init_child(eiterator_t it,T⟨τ, k⟩ the_tree)
eiterator_init_cell(eiterator_t it,A⟨τ, k⟩ the_array))
eiterator_init_refset(eiterator_t it,eref_t the_set[],int len)
```

Table 5.5: *e-foreach*, *e-joinall*, *e-callit* primitives. In addition the primitive to initialize eskimo iterators.

As a matter of fact, the *e-foreach* turns the fork/join style paradigm into a cobegin/coend one (Dijkstra, [80]) in order to apply newly created *e-flows* to subparts of the domain. As discussed in Chapter 4 this can be considered as a form of data parallelism. Due to the chosen programming model the *e-foreach* created *e-flows* are independent, thus they can be turned in loosely coupled parallel activities.

Programming and design remarks

The DAG consistency model has a profound impact on how programs are written. First of all the language does not need locks or barriers. From the language viewpoint consistency actions (acquire, release) are taken at *e-call*/*e-join* time. From the runtime support viewpoint they are really taken only if a given flow of control moves from a processing element to another or the cache is flushed. This is a quite relaxed memory model [86].

The main idea under eskimo (as in other environments like *Cilk* [42, 112]) is that writing a parallel program without locks is easier. This simplicity is not for free. The foremost impact of DAG consistency on programs is that a shared variable cannot be used to trigger actions in independent *e-flows* (see also Lemma 1) since their view of the memory is matched only when they join. A similar phenomenon happens in Lazy Release Consistency. Since the release does not invalidate other processing element memory, as shown in Figures 3.8 and 3.7, even a simple producer-consumer behavior cannot be established without an explicit synchronization event on both communication sides (a release-acquire pair). The problem is not present in eager consistency models. DAG consistency, that is even lazier than LRC[6], further exacerbates the problem since DAG consistency does not keep the coherence (location consistency) among independent *e-flows*. When two independent *e-flows* write the same memory word, it is undefined at the moment of the join. It is up to the programmer to avoid that. Observe this is not an eskimo specific problem, but it is rather a DAG consistency one (*Cilk* has the same behavior).

Unfortunately, there are many cases in which the programmer needs to "accumu-

[6]The problem is extensively discussed in [86].

late" a result in a given variable or memory region. A possible extension of eskimo language that copes with this situation is the following one. We can extend join functions parameters with a pointer to a *user defined* function. This function must implement an associative and commutative binary operation on a given type. A so typed variable has been enclosed in a shared region. At the moment of the join (at the moment memory consistency is reconciled) the function is applied against the two (inconsistent) versions of the shared region and the result is stored into the shared region. As an example if two independent threads increment a shared variable representing a counter, at the join time we can set up the value of the shared variable as the sum of values held in the two versions of the shared variable (since they represent a partial sum). This mechanism basically implements the classical "reduce" skeleton (commutative fold) within eskimo language.

However, even extended as mentioned, the programming model is not still completely satisfactory. A future extension of run-time support may include the support for many (possibly coexisting at the same time) consistency models in order to associate different behaviors to different memory classes. A possible way towards the result may consist in bringing into eskimo a flexible software DSM that may support multiple-protocols (as for example [19]).

5.3 A running example

In this section we develop a simple application in order to exploit main eskimo features. The application builds a spread tree, then visits it. For the sake of simplicity we present the application using a pseudo-language, neglecting trivial or already presented concepts.

In the application (see Figure 5.5) a spread tree SDT (`k_tree_t`), then a shared variable (`a_tree`) are declared. The main procedure first initializes the run-time support, then calls the build function and the visit function, and eventually it terminates the run-time support.

The `seq_build_tree` function (see Figure 5.6) recursively builds a complete k-tree with a given depth; each node of the tree holds an integer value. Observe that there are not *e-calls* within `seq_build_tree` function, thus the tree will be built along a single *e-flow*. Since shared data items allocated within an *e-flow* have the same home node, the tree exploits a very strong spatial locality. Nevertheless, the tree (and its nodes) are accessible from any *e-flow* provided they have a reference to the tree (or to its nodes).

After the tree has been built, we can visit it; the `tree_visit` function is shown in Figure 5.7. Starting from the root the function accesses to node body, then recursively applies itself to all (not null) children of the node by means of an *e-foreach*. The function waits on the *e-joinall* for all recursively *e-called* functions before returning. *e-called* functions return values are discarded because they are useless here (by using NULL as return vector address in the *e-joinall* primitive).

The function implements a top-down (concurrent, nondeterministic) visit of the tree. We expect it will accomplish the task exploiting a certain degree of parallelism. The run-time support turns created *e-flows* into parallelism by evaluating the parallelism-overhead trade-off as discussed is Section 4.1.2. In the particular case, we know that the tree is allocated in a single processing element. Therefore, the support tries to exploit first multithreading parallelism on the tree home node in order to exploit access locality. The solution has a low overhead since memory accesses are local to the node; this kind of parallelism is very effective in presence of SMP nodes and fits very well the parallelization of small tasks (fine-grain). If the task to be computed is bigger, keeping a strong locality may lead to a heavy computation imbalance, thus to a resource waste. When the run-time detects a significant computation imbalance it tries to restore a good resource usage by spawning future *e-flows* across the system. Since our visit function does just few basic operations, we expect just a multithreading parallelism, at least in the case of small trees.

With regard to exploited parallelism, a key role is played by the first parameter *e-functions*; it is really a hint to the run-time support. eskimo scheduling is data-driven, the language run-time tries to schedule *e-called e-functions* in such a way that the first parameter can be accessed as fast as possible. Notice in the case the first parameter of an *e-functions* is `E_NULL` the run-time schedules the resulting *e-flow* taking into account a mix of workload and memory-load balance. The current policy consists in considering *e-calls* with an `E_NULL` parameter as functions having as main target the allocation shared data items (as in the case of `tree_par_build` function). Thus the run-time privileges the memory-load balance. Alternatively, the programmer may use *e-call* variants exploiting a major concern for workload balance. Since first eskimo aim is experimentation, the programmer may also configure their own policy by leveraging on workload and memory-load information. At this end (as we shall see in Chapter 6) eskimo run-time maintains a lazy table of each processing element status (including load and memory status). The lazy table is updated together with each message exchange among processing elements (see also Chapter 6).

Let us now parallelize also the function that builds the tree. The `tree_par_build` function is shown in Figure 5.8. The main structure of the function is pretty similar to its sequential counterpart, the main difference regards recursive function calls that have been substituted for *e-calls*. Together with *e-calls*, a join primitive has been introduced. Since the function no longer exploits a single *e-flow*, the tree may be built in parallel and nodes of the tree may result spread across the system. The latter fact has nontrivial effects also on visit function performance. The spread tree is now really spread across distributed memory.

The *read and write* program writes or reads a tree in distinct, successive phases. However, creating a function that modifies a tree (i.e. reads and writes) is straightforward. It is enough to interleave the order in which `tree_visit` and `tree_par_build` are called, or at least join them in a single recursive function.

main *(build and visit)*

```
1  e_declare_tree(k_tree_t,int,K);
2  k_tree_t a_tree = TREE_INITIALIZER(K);   /* SDT declaration */
3
4  main {
5    < declarations >
6    e_initialize();               /* start and init the run-time */
7    root = tree_seq_build(E_NULL,&arg);
8    tree_visit(root,&arg);
9    e_terminate();
10 }
```

Figure 5.5: The `main` of *build and visit* eskimo program.

tree_seq_build *(build and visit)*

```
1  eref_t tree_seq_build(eref_t father,void *argsv) {
2    eref_t node,a_child;
3    < other declarations and initializations >
4    efun_init();
5    if (< requested depth not reached >) {
6      node = e_add_node(a_tree,father,args.child_n);
7      body = ((int *) rw(node));  /* bind body to node in rw mode */
8      *body= ...                  /* write node body              */
9      for (i=0;i<K;i++) {
10       < Change args >
11       a_child = tree_seq_build(node,&args);
12       e_setchild(k_tree_t,node,i,a_child); /* link a_child to node */
13     }
14   }
15   return(node);
16 }
```

Figure 5.6: The `tree_seq_build` *e-function* (part of *build and visit* eskimo program).

tree_visit *(build and visit)*

```
1  eref_t tree_visit(eref_t node, void *foo) {
2    eref_t body; int *body;
3    eiterator_t it;                /* An eskimo iterator */
4    efun_init();                   /* this is an eskimo function */
5    eiterator_init_child(it,a_tree);
6    body = r(node);                /* bind body to node in r mode */
7    /* foreach child of body apply tree_visit(child,foo,sizeof(foo))  */
8    e_foreach_child(it,body) {
9      e_callit(tree_visit,it,foo,sizeof(foo));
10   }
11   /* wait the completion of all instances, discard return values  */
12   e_joinall(it ,NULL);
13   return(E_NULL);
14 }
```

Figure 5.7: The `tree_visit` *e-function* (part of *build and visit* eskimo program).

tree_par_build *(build and visit)*

```
1  eref_t tree_par_build(eref_t father,void *argsv) {
2    < declarations and initializations >
3    efun_init();            /* this is an eskimo function */
4    if (< requested depth not reached >) {
5      e_handler_t h[K];
6      e_handler_init(h, K);
7      node = e_add_node(a_tree,father,args.child_n);
8      body = ((int *) rw(node));
9      *body= ...
10     for (i=0;i<K;i++) {
11       < Change args >
12       e_call(&h[i],tree_par_build,node,&args,sizeof(args));
13     }
14     e_joinall(a_child,tid,K);
15     for (i=0;i<K;i++)
16       e_setchild(k_tree_t,node,i,a_child[i]);
17   }
18   return(node);
19 }
```

Figure 5.8: The `tree_par_build` *e-function* (part of *build and visit* eskimo program).

Chapter 6

eskimo: implementation

Readers' road-map. In this chapter we present eskimo run-time support design principles. Section 6.1 describes how eskimo abstracts the parallel architecture (consistency model, cache). Section 6.2 briefly introduce the implementation of Shared Data Types. Some experimental results are presented in order to support design choices.

A new technology can only gain acceptance if it can be demonstrated that adoption offers some improvement to the status quo. Skeletal approach first concern are programming simplicity and performance portability. We should be able to show that structured programs can be ported to new architectures, with little or no amendment to the source, and with sustained performance. This can be contrasted with the performance pitfalls inherent in transferring semantically portable but performance vulnerable ad-hoc programs. We should be able to show that skeletal programs may outperform the conventional implementation constructed with an "equivalent" programming effort. This basically means that problems that make difficult the implementation of a low-level parallel program must be (partially) resolved at skeletal language support level.

eskimo is built on top of a hierarchy of two run-time layers: *etier-0* and *etier-1* (the former being the lower-level tier). The former one wraps the communication stack, implements a pool of thread, keeps updated system statistic information, and provides the basic mechanisms for caching and scheduling and memory consistency. The latter one implements data structure mapping and provides scheduling policies. These policies may be supported by statistic information about the system status (e.g. processing elements load, memory load, etc.).

In next section we sketch basic features of the language run-time. Some experiments that have been supported the run-time design are also presented. Experiments regarding eskimo applications are instead presented in Chapter 7.

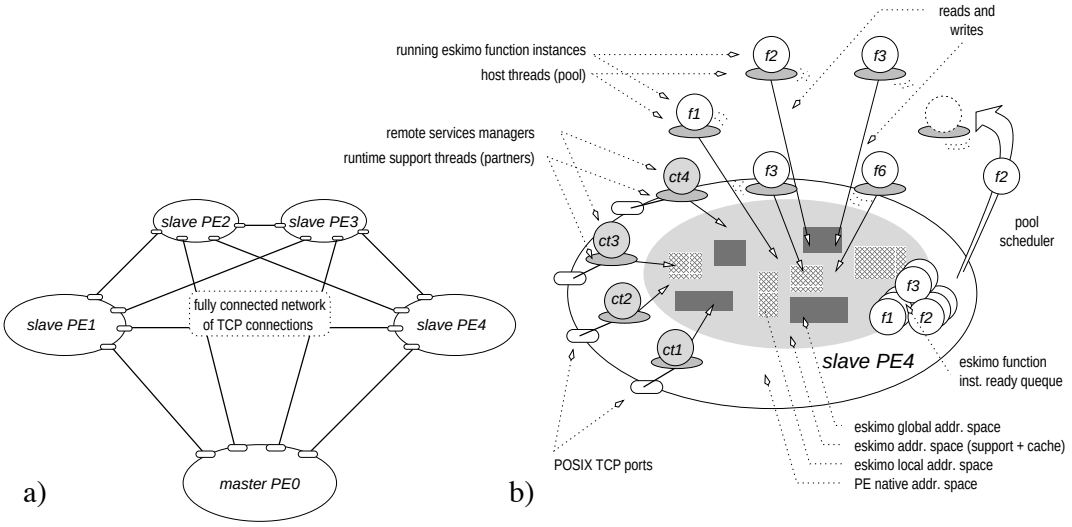


Figure 6.1: a) Virtual architecture in the case of 5 PEs. b) PE internal organization.

6.1 Abstracting the architecture

eskimo languages support is specifically designed for multicomputers, in particular for clusters of SMP. Multicomputers consists of multiple independent processing elements (PEs) with local memory modules, connected by a general interconnection network (see also Section 1.2.1). Multicomputers do not provide a shared address space at the hardware level. However, SMP nodes of a multicomputer provide shared address space (in general implemented in hardware) among processors.

eskimo languages provide the programmer with *SDT*s and *references*, i.e. variables and addresses in the shared address space. The shared address space is a "virtual creature" of the run-time support. The run-time support bridges the gap between the hardware and the programming abstraction by means of a software DSM (see Section 6.1.2). Both eskimo DSM and synchronization primitives are built on top of a connection-based message passing transport layer, namely POSIX TCP sockets. These are included within a simple wrappers at *etier-0* level. Currently eskimo run-time does not support other communication layers (see also Section 8.3).

At the bare bones, an eskimo application is a "particular" SPMD C program. Once compiled it can be executed on a *virtual architecture* by means of the specialized launcher `erun`. `erun` launches a copy of the application in each PE of the virtual architecture via a secure remote shell connection. Each copy of the application contacts the other copies in order to establish a fully connected network across the virtual architecture. A eskimo program (at *etier-1* level) sets up the network as first operation (before execution enters in main), while at *etier-0* it is possible to set up the network in any part of the application by calling an explicit primitive. Anyway, before any other eskimo primitive could be called, the architecture must be

fully set up and initialized. After the initialization the virtual architecture appears as sketched in figure 6.1 a). At the operating system level, an eskimo application consists of a set of identical processes. Many processes may be run on the same PE, even if this does not lead to any effective speedup because the internal PE parallelism is already exploited by the multithreaded nature of each process. The basic structure of each one of such processes is sketched in figure 6.1 b); we will first take a look at the general organization, then we will discuss each point in detail.

At the initialization time eskimo support creates two sets of threads: the first set (*partners*) includes a fixed number of threads acting as communication/protocol co-processors. Each process relies on the protocol in figure 3.1 (see Section 3.1) in order to interact with each other processes. Threads in the set act as partners of the protocol. They wait for a request, do something (typically they read/write the memory), then reply the transaction.[1] The other part of the protocol is included/distributed into language primitives and is run by threads in the second set.

Threads in the second set (*pool*) do the rest of the work. Actually they are arranged in a pool. The number of threads in the pool may change during the execution, but it has a run-time support constant as a bound. Each thread runs a simple distributed scheduler (*pool scheduler*) that takes a ready *e-function* instance from a ready queue, then executes it and restarts its cycle.

Notably, running function instances can access the private address space without any run-time mediation and the shared address space with eskimo support mediation. In case a function reads or writes a remote shared data item (as an example a node of a tree) a local cached copy of the segment holding the data item is created and used instead of the remote one. From the moment the cached copy is created until it is reconciled no run-time support actions are needed and the cached copy is accessed without any mediation.

Threads synchronization – both among threads in partners set and pool set – relies on POSIX thread synchronization primitives (namely mutex_lock/unlock and cond_wait/broadcast). eskimo programmers do not see any of them for they are enclosed into eskimo primitives.

The performance of the virtual architecture for producer-consumer pattern (miming the not cached write operation) is shown in figure 6.2. eskimo performance includes thread synchronizations overhead (very few are needed in the described case) with respect to the same communication pattern implemented in pure C/MPI (MPICH v1.2.5).

In the Figure 6.3 the eskimo virtual architecture communication performance are compared for the same producer-consumer pattern in the case each request is served by a different thread (thread per request model) with respect to the described architecture with a single thread acting as communication co-processor. The latter solution has exploited a better performance with respect to the former, thus it has been adopted.

[1] Referring to figure 3.1, they implement actions of the protocol under the "Destination" column.

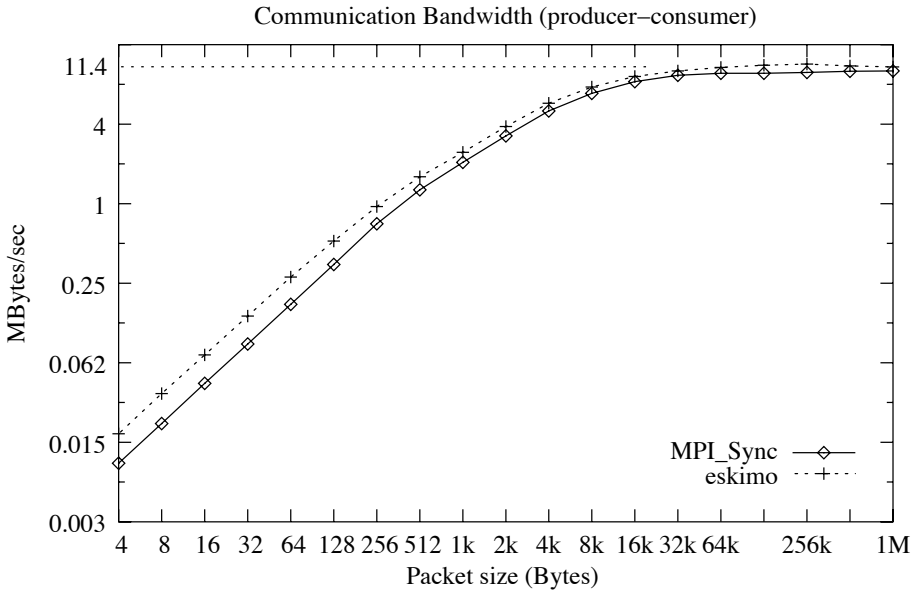


Figure 6.2: *etier-0* communications performance for a producer-consumer pattern with respect to MPI communications on the `backus` cluster (2 PentiumII@266MHz, switched Ethernet 100MBit/sec). eskimo mimes the protocol shown in figure 3.1 for the write operation. MPI version uses `MPI_Sync/MPI_Recv` primitives.

In the remain of the section we shall examine important features of eskimo support.

6.1.1 A multithreaded support

Multithreading is becoming a pretty popular programming approach primarily because threads provide a clean and simple manner in which programmers may express logical parallelism in applications. For this reason, threads libraries designers do not often care too much about efficiency of context switching. Our preliminary experiments using Solaris threads suggest a gloomy 10% time gap among thread and process switch.[2]

Nevertheless, threads exploit parallelism within a single PE (at least in SMP processing elements) and may access to the (native) shared memory much more efficiently than processes. Furthermore, threads may be used also to tolerate (part of) communication latency with other PEs.

[2]Experiments on a Sun UltraSPARC running Solaris O.S. and Solaris user threads.

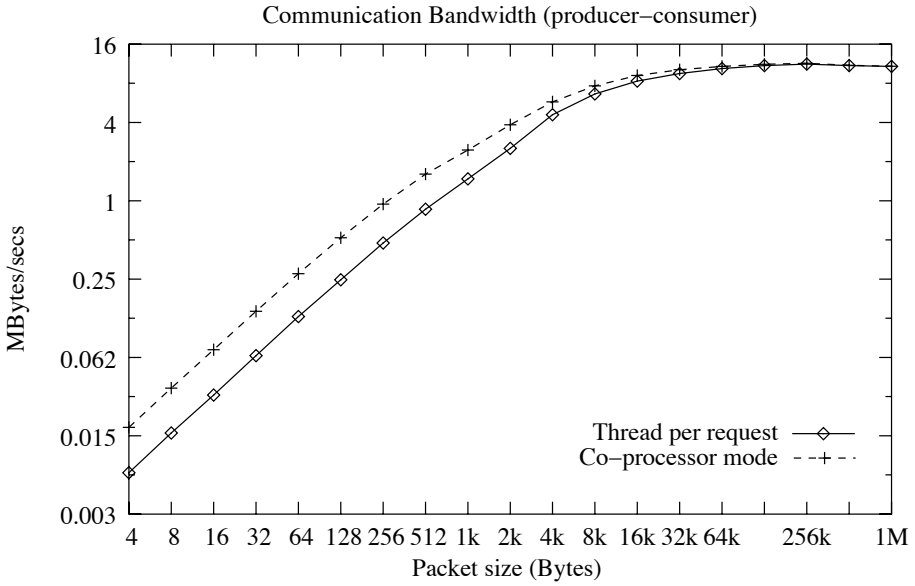


Figure 6.3: *etier-0* communications performance for two different multithreaded organization schemes on the `backus` cluster (2 PentiumII@266MHz, switched Ethernet 100MBit/sec).

Preliminary experiments

Our experiments show that software multithreading may be used to hide remote memory access, and that in many cases the technique work better than prefetching. In our experiments we take in account the *Quadrics CS2* and *QM1* multiprocessors, which are basically scalable clusters of nodes connected by means of a fast fat-tree network (more than 40MByte/sec full-duplex). In turn, each node is a Sun Enterprise, a cache-coherent bus-based symmetric multiprocessor running an adapted version of Solaris UNIX[3]. Each node is tightly connected to the network through a communication co-processor *(Elan III)* that enables nodes to communicate in a message passing way via DMA transfers; processors into a single node communicate by sharing memory in a native way.

We consider the `farm` skeleton (see Chapter 2), an embarrassingly parallel paradigm in which a set of independent *workers* compute a function on a stream of tasks. The `farm` belongs to the skeleton set of SkIE programming language [167, 29]. It has traditionally been implemented using a software prefetching technique and the message passing library MPI (see [7] for any further detail). We developed a new multithreaded implementation of the `farm` skeleton in order to understand the performance ratio between the support based on multithreading and the one based on prefetching/precommunication.

[3]The nodes of Quadrics CS2 are Sun SparcStation 10 with 2 PEs.

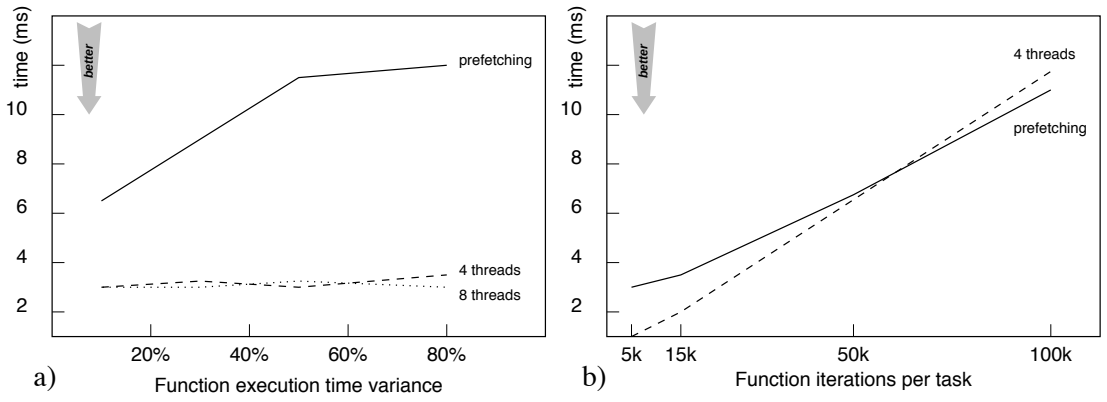


Figure 6.4: SkIE farm: multithreading and prefetching implementation. a) Variance of tasks load versus service time. b) Tasks load versus service time [36].

Figure 6.4 shows a comparison among the traditional implementation and a new multithreaded implementation, first (a) with respect to the load variance with fixed mean, then (b) with respect to the absolute load with fixed 30% variance. The load is simulated iterating on a dummy function (1000 iterations = 130 μsec), the number of iteration follows a normal distribution with configurable mean and variance, the threads load is linear in the number of iterations.

Let us briefly comment on these results: Figure 6.4 a) shows that multithreaded version is far more stable in execution time with respect to the variance of load, thus on bursty or highly data dependent loads. Figure 6.4 b) shows how multithreaded implementation outperforms prefetching implementation on fine grain computations, while the context switch overhead is a limiting factor for coarse grain computations where prefetching can completely overlap computations with communications [36].

Clearly the switch cost affects the types of latency that we can hide; in our case, the context switch time is about 10–20 μsecs and a transfer of 4 KBytes remote page costs about 150 μsecs. Our experiments show that we can tolerate pretty well a load of about 600 μsecs with a variance up to 80% using only 4 threads per processor. Smaller run lengths are also tolerable with a greater number of threads (not with the same efficiency), but it is pretty clear that we can not tolerate transfers less than 4 KBytes (a page) due to the high context switch cost. The argument is really straightforward: we can not tolerate a latency by switching to another thread if the switch costs about the same time.

A similar set of experiments has been performed in a Beowulf with similar results (266MHz PentiumII workstations running Linux RedHat and connected by means of a switched fast Ethernet). The only significant difference consists in the fact that MPI on this platform does not exploit any valuable parallelism between processor work and data precommunication/prefetching by means of its asynchronous primitives.

Starting from these results we decided to adopt the multithreading solution for the eskimo support.

How many threads?

eskimo language enables the programmer to create a number of *e-flows* through *e-calls*. *e-flow* concept may be thought as the direct abstraction of a thread. Along this path a straightforward implementation strategy might be followed: we can implement an *e-call* using a thread_create primitive, provided we have an opportune software DSM and a mechanism to spawn threads among different PEs. As discussed in Chapter 3 tens of DSM exist. Also, a few of them are able to exploit the power of modern thread libraries to provide multithreaded protocols, or at least to provide thread-safe versions of the consistency protocols. At the best of our knowledge very few of them exploit the thread migration feature (the DSM-PM2 platform is one of them [22, 19]).

However, eskimo language is targeted to experimentation of dynamic data structure (trees in particular), designing its support we should take in account its peculiarities. The language is designed to adhere to the C programming style. We imagined trees as the natural lattice to store hierarchical information, and Divide&Conquer as the natural way to build, to read and to write trees in parallel. We expect most of eskimo applications to be recursive, at least in parts of them dealing with trees. Let $P \equiv$ `if` B `then` $\mathcal{C}[S, P]$ the general schema of a recursive algorithm, where $\mathcal{C}$ is a composition of S (that does not contain P) and P itself. As clear, a recursive function solving a problem wait the solution of sub-problems before conquering the results and return. As seen in Chapter 5, eskimo enables the programmer to substitute calls with *e-calls*. Executing a recursive *e-call* we expect that each function instance will be executed in a different flow of control. The majority of such flows does nothing, they are stuck waiting in a join statement in order to conquer sub-problems. We should expect a huge number of them, since only function instances in the fringe of the activation tree are really able to run.

Let us return back to threads. Current Linux-threads library may tolerate hundreds of threads and no more [127] (even if the Linux-threads implementation is improving over time [81]). In addition DAG consistency has not explicit mechanisms matching directly pthread synchronization queues (e.g. mutex, condition variable) and the number of synchronization events (related to *e-flows* number) is not bound. On the one hand, we need a different event/synchronization queue for each *e-flow*. This may lead to an excessive use of system resources (memory for synchronization queues and time for their management) dedicated to synchronization only. On the other hand, due to pthread synchronization mechanisms (pthread_cond_broadcast/wait), the cost of synchronizations grows with the number of threads waiting of a given event. As result we choose to associate many different synchronization events (related to *e-joins*) to a limited number of condition variables (one condition variable per thread), and at the same time, to bound the number of

threads per process. Ready *e-flow* are then scheduled onto available threads.

The pool of threads

We implement a pool of threads. Each thread in the pool permanently runs a simple distributed scheduler (*pool scheduler*) that gets a job in LIFO order (i.e. an eskimo function instance) from a ready queue, executes it, then restarts the cycle. We can add a job in the local PE ready queue or in a remote PE one by means of an *eskimo_1* primitive. The run-time may dynamically change the number of threads in the pool according to the run-time status[4]. The scheduler does not implement preemption for two main reasons:

1. We would completely rely on standard programming and operating system tools. Even if we bind an *e-flow* to a given PE, in order to implement preemption we need at least a mechanism to checkpoint and resume a running function environment. If in addition we would like to migrate a suspended *e-flow* to another PE we must decouple the run-time from standard allocation mechanism (as an example DSM-PM2 implementing threads migration relies on a "iso-allocation" strategy among PEs, see [19, 21] for any further detail). In some sense we feel that it is difficult to design such mechanisms better than existing.

2. POSIX threads are anyway subjected to operating system preemptive scheduling. Introducing another level of preemption on top of the operating system one might create interferences between the policies used in the two levels.

Since we have no preemption, an *e-flow* once extracted from the ready queue is completely executed up to the return statement, as far as eskimo support concerned[5]. It follows that:

- An *e-flow* (see Section 5.1) is always linked to a given PE (and to a given thread within a PE). That enables us to implement primitive variables with standard C variables. Moreover, *e-functions* may migrate only at *e-call* time (remotely spawned actually), thus they do not carry private data except parameters. This reduces communication volume and enhances locality of accessed data.

- A further scheduling mechanism is needed. Since an *e-flow* do not leave its hosting thread before return, if we saturate all threads in the pool with functions waiting for other functions in the ready queue we enter in a deadlock state. Such further scheduling is enclosed into *e-calls*. This implements a simple deadlock avoidance strategy: in the case the pool is close to be empty no

[4]Such are infrequent events. The run-time does not create (destroy) a thread for any job, but only when it "feels" more (less) threads are needed.

[5]Actually it is subjected to the operating system POSIX thread scheduler.

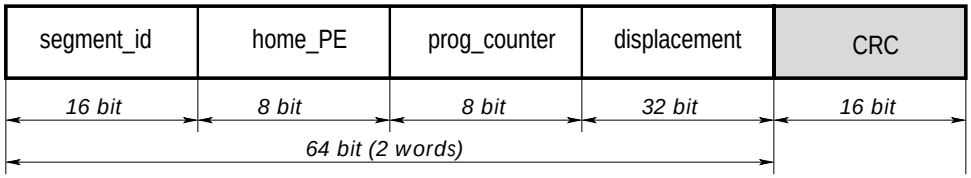


Figure 6.5: Shared address implementation (`eref_t`): CRC part (in gray) is optional and normally used only during debugging.

further *e-flows* are mapped into threads, these are sequentialized. In the case processing element usage drop below a given limit more resources (threads) are added to the pool (then removed when they are no longer needed).

6.1.2 eskimo Shared Virtual Memory

eskimo software DSM has been specifically designed for the eskimo run-time support. According to Chapter 3 taxonomies it is a software implemented DSM, and supports DAG consistency [42] by means of a Multiple Read Multiple Writer protocol. eskimo shared virtual memory is engineered starting from already known technologies. We will not go into all details of it, but we will only describe its original aspects referring to the abundant literature in the field for already known techniques.

Shared address space. eskimo shared virtual is a segment-based DSM. All shared variables are allocated in segments, each of them holding a part of the variable. In principle segment size have no relationship with page size since segments are allocated via standard malloc (however, they may be set to a multiple of page size for other reasons). A segment has a home node, and should be considered as an atomic chunk of memory: a PE may have or have not a (copy of the) segment, but it cannot have a part of it. An address in the shared address space (i.e. a reference) is implemented as shown in Figure 6.5. It includes: segment_id that represents the segment identifier. It is the key that distinguish segments with the same home_PE value. home_PE that is the home node of the segment. prog_counter that is an ever increasing counter (modulo its maximum). When a shared variable (or a part of it) is freed, the hosting segment is also deallocated and its segment_id is reused. It may happen (because an error in the program) that an *e-flow* uses a reference (i.e. a shared address) having as key a segment_id that has been first freed then reallocated to another purpose. prog_counter is used to distinguish the two addresses and to raise a run-time error. displacement represents the address offset in respect of the base of the segment. CRC that is a Cyclic Redundancy Check of the other address field used for debugging purposes. Notice that widths of fields of the address may be changed (widen for example), even if it would be desirable for performance reasons that the whole structure match typical architectural parameters.

Shared addresses translation is embedded into r/rw eskimo primitives. As example we sketch how r works. For the sake of simplicity we neglect consistency related actions and flags setting (e.g. dirty bit):

```
void * r(eref_t addr) {
  if (< segment not cached and remote >) {
    < request to >  home_PE < segment > segment_id ;
    < wait the answer >;
  }
return(segment_table[segment_id].base + displacement);
}
```

r will find a communication partner in the home_PE node, in particular among threads in the *partners* set[6]. A shared address identifies a system-wide unique data object. A shared address is translated into a PE logical address by using the segment_table. The segment_table is (partially) replicated in each copy of eskimo processes, and it is shared among all threads of the process. The table enables the match of a shared address with either a segment of the memory local to PE or a cached copy of it. segment_table copies across the system are keep coherent accordingly to the (relaxed) memory consistency chosen (see also Section 6.1.2). After a shared address has been translated into a local logical address, then we can use the latter to access to the data object. eskimo support does not make any physical copy of the data object, we can consider it a pseudo zero-copy protocol[7].

Many address translation mechanisms has been proposed in literature. Some of them rely on specialized hardware devices [78], some others emulate the cache of a multiprocessor using the MMU and operating system software [129]. In the latter approach, the address space is divided up into chunks, with the chunks being spread over (in some way) all the processors in the system. When a processor references an address that is not local, a trap occurs, and the DSM software fetches the chunk containing the address and restarts the faulting instruction, which now completes successfully. Clearly, the trap is triggered by a page fault event, under the control of processor MMU. The method makes the local access for non faulting instructions as cheap as in uniprocessor systems. However, the interrupt cost, associated with receiving a message has been proved to be the largest component of the slow remote latency, not the actual wire delay in the network or the software implementing the protocol [145].

We choose to a bit more abstract approach, we do not rely on the iso-allocation of logical addresses. We exploit the shared address space obtained as the sum of the logical address space of all PEs in the virtual architecture. At this end, we defined shared addresses as opaque objects not limited by machine word length. Dynamic

[6]Referring to Figure 3.1, r implements actions of the protocol under the "Source" column.

[7]A copy really happens within the TCP stack, moreover consistency protocol may need to perform a data copy.

test_A

```
1 < force in cache all vars >
2 < start the counter >)
3 for(i=0;i<K;i++) {
4   value = (int *) r(root);
5   *value += i;
6 }
7 < stop the counter >
```

test_B

```
1 < force in cache all vars>
2 value = (int *) r(root);
3 < start the counter >
4 for(i=0;i<K;i++) {
5   *value += i;
6 }
7 < stop the counter >
```

Figure 6.6: Experimenting address translation overhead.

data structures mainly motivates the choice. Let us take trees as example, during vertex allocation we expect to have a really lazy knowledge of tree vertex distribution among the system, especially because we would not like to synchronize PE to allocate vertices. In this setting, a 32 bit logical space divided up the processing elements becomes a little logical space.

The price to pay such flexibility is pretty high. We cannot rely on any hardware support performing shared address translation. Our shared addresses translation is completely software implemented. Anyway, we implemented a quite fast translation code, in case of cache hit it consist in one conditional branch and 5 elementary statements, no one of them is a memory locking statement. We measured a 31 clock cycles overhead for `r` translation on a 450 MHz Pentium III workstation.[8] On the same architecture a sum between (cached) integer costs about 6 clock cycles, and a (processor) cache miss costs thousand of clock cycles. In both cases the evaluation does not takes in account the performance gain due to pipelining of many instructions. Our address translation code is really macro-expanded into source code and benefits from C compiler classical optimization and processor instruction pipelining. A more significant experiment is shown in Figure 6.6. `test_A` executes a cycle including an address translation (`r`) and an integer sum, while `test_B` cycle does not include address translation. Both chunks of code read and write in the fastest processor cache. Depending on the architecture we experienced a 5–20 slowdown factor of `test_A` with respect to `test_B`, suggesting a non-negligible address translation overhead. This overhead may unacceptably hight in the case large fraction of executed code consists in `r`/`rw` operations (see also experiments in Section 7.1). However, consider that `r`/`rw` are not reads and writes. These are operations that link a shared variable address to a private pointers. Actual reads and writes are then performed by using the private pointer with no further eskimo run-time mediation. The point here is that eskimo programs ought to be written as `test_B` even if they

[8]The measure is performed using the hardware "Time-stamp Counter" of PentiumPro class processors. The counter has resolution of the clock cycle and does not interfere with normal processor operations [110].

can be written also as `test_A`. The argument is quite similar to others in sequential programming. C programmers arrange programs in such a way that arrays are visited in row major order when possible; Fortran ones in column major order just because they know that is more efficient.

Consistency

eskimo run-time keeps the distributed memory consistent according with a DAG consistency, i.e. a relaxed consistency (see Section 3.3, page 102).

Currently eskimo run-time maintain DAG consistency into the shared memory by using a version of the BACKER algorithm [41], (see page 103) that has been originally conceived for the *Cilk* run-time system [42, 146]. Differences between the original and our implementation exists but are minimal. One of them regards the granularity at memory data objects are kept consistent that is the segment in eskimo. As in the original formulation we adopt a MRMW protocol using the classical twinning technique [116].

A version of eskimo run-time support exploit Pentium's streaming extension (MMX/SSE) to perform all tasks related with diffing operations. Since our current developing platform (gcc3.2 on Linux RedHat) does not support an easy access[9] to the MMX sub-system we rely on the Intel Performance libraries. These libraries are neither shareware nor open source, we take them as a black-box. Out of the block we registered an effective speed-up (up to 800%) with respect to the classic code for very long stream only. Unfortunately the function call latency (that probably involve a register re-arrangement) is too high to make the function effective for our aims. Recently we ported eskimo on MacOS X (Version 10.2.6, Mach 3.0 kernel, OS services based on 4.4BSD, Berkeley Software Distribution). PowerPC processor architecture and software development tool (gcc 3.1) seem enabling an easier and effective usage of the vector co-processor. In particular the processor includes a full-fledged set of dedicated vector registers (see Figure 6.7) and the gcc compiler is able to automatically divert some code loops toward the vector unit[10]. The "functional" porting of the eskimo library has been immediate. Yet we have no experimental evaluations of eskimo on this architecture (these are ongoing).

Cache

eskimo cache sub-system is specifically thought for a multithreaded framework; threads within a processing elements cooperatively manage the cache. Each cached segment in a processing element may be accessed by all threads (of the processing element). The consistency model together with scheduling policy ensures that all threads can access to a shared cached copy concurrently (without any lock in the

[9] Actually the MMX can be used by inlined asm or by compiler builtins.

[10] Probably the former issue enables the latter.

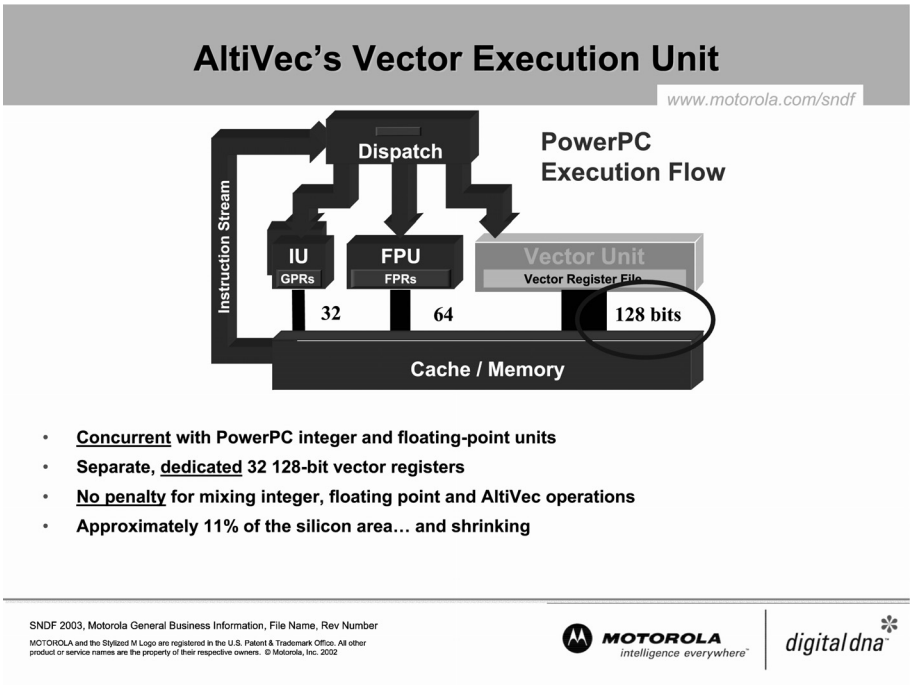


Figure 6.7: PowerPC Altivec's Vector Unit. Taken from [130].

run-time support). Moreover any thread may concurrently create/destroy a segment cached copy without synchronizing with others threads.

In particular each thread in the pool has a private stack of cache slots identifiers. At the time a faulting `r/rw` operation is issued the requesting thread gets an identifier from its stack from its private stack. In case of success, it mallocs the space needed to hold the whole segment (including the `r/rw` requested reference) and store the segment base address starting (machine) address into the `segment_table`. The process needs no lock at the support level. In case the private stack of identifiers is empty, the thread tries to get some identifiers from a global[11] stack. In case of success the previously described procedure is activated. In case, for any reason, a thread cannot find a empty cache slot identifier, a cache flush is performed (thus the segment is reconciled at the home-node, as described in Section 3.3.2).

The flushing operation is a bit more complex. A cached segment may be concurrently accessed by many threads. These threads do not grab any lock in order to access the memory. Therefore, before flushing a segment we must ensure that the segment is in use to the current thread only. In case the segment is in use to many threads, the current threads asks to the others to consider the segment as unavailable (for a while). Each thread "hears" other thread requests at each language primitive. Possible race conditions during the operations are managed by means of

[11] With respect to all threads in the processing element, and in mutual exclusion.

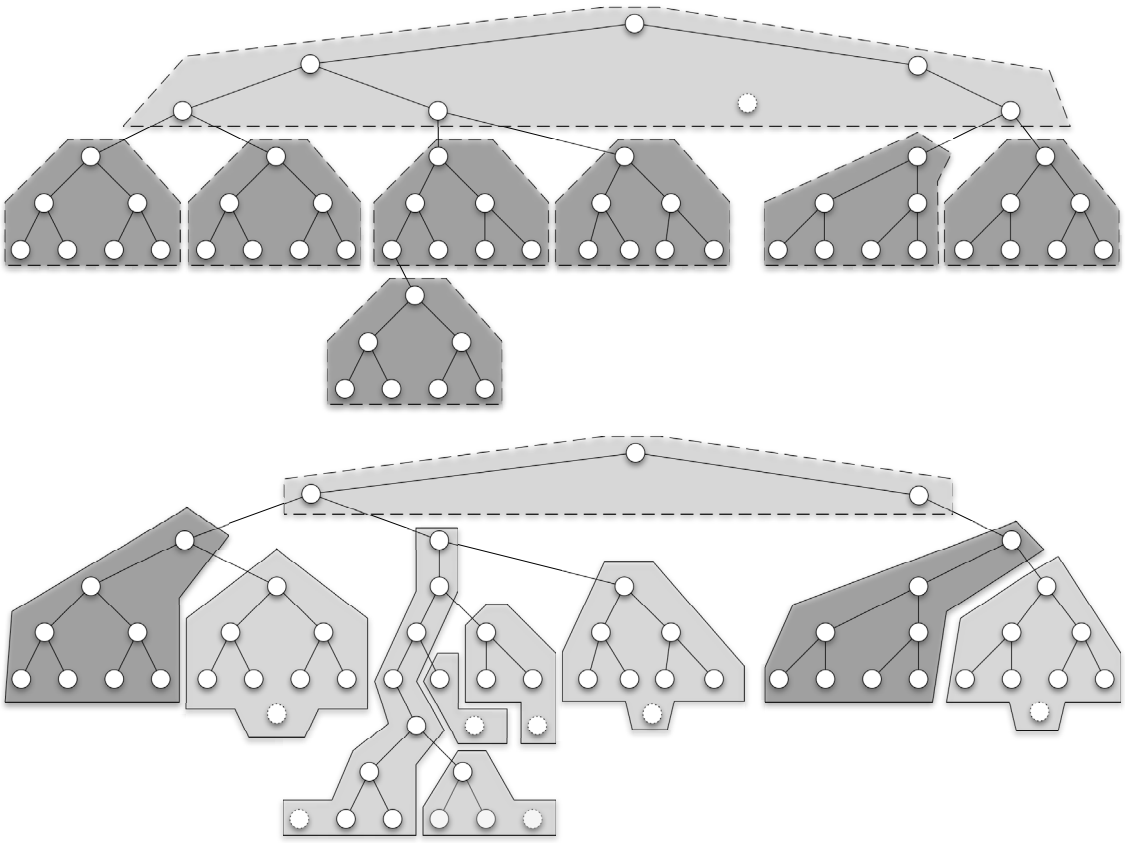


Figure 6.8: A spread tree stored in two different ways: heap (top) and heap+fist-fit (bottom). Dashed box are heap segments, solid box are first-fit segments. Dark grey boxes are completely fulfilled. Light grey boxes are incomplete.

atomic operations in memory (an exchange instruction basically). Since each thread may perform many access to the same cached segment without performing any language primitive (see also Figures 6.6 and 5.3), we must also ensure that dynamically evolving set of segments is not subject to the flush.

The (pretty complex) mechanism ensures that in the majority of cases cache is managed without any lock, thus in parallel within a SMP node.

6.2 Shared Data Types

As discussed the architectural framework must operate on pretty large grains in order to mitigate language primitive overheads and to exploit a significant efficiency in communications. Moreover, we must ensure that SDT typical access patterns exploit a good spatial locality.

As far as shared regions concern they are simply allocated in segments tailored

for their size. The implementation of spread arrays has been extensively studied and tested in literature [107]. We adopt the very simple strategy to divide them in regular segments. Such segments are spread across the architecture accordingly with a hash function[12].

Trees are more interesting from out viewpoint. We have already discussed how to declare and populate a spread tree in Chapter 5. In summary in order to use a spread tree the programmer should follow these steps:

1. Instance a spread tree SDT with a C type representing node held data. The operation is performed by means of the `e_declare_tree`.

2. Declare a spread tree shared variable using as type the name given to the SDT at the previous step (either statically or dynamically via standard malloc). This phase declares an empty tree.

3. Populate the tree starting from the root using the `e_add_node` primitive.

Let us describe now what really happens in the run-time support:

1. Two new types are created. One of them represent the tree, the other its generic node. Both of them are really are C `struct`. The former type is named according to the SDT requested name. It hold few information such as the number of children and a dummy variable holding a prototype of the generic node. The latter struct holds the C type used as parameter and some additional information such as an array of references to children.

2. Nothing happens apart for the allocation of the first struct mentioned above.

3. Starting from the reference to the father the run-time decides if the requested child must be placed in the same segment or in a new one. This decision is made according to the mapping policy. Clearly if the father is null a new segment is created (and the node is the root of the tree).

   - In the case the child is placed in the father's segment it inherits father segment_id. The run-time choose only node displacement within the segment according to the "internal" mapping policy (heap, first-fit). Some information are kept within the segment to trace segment current status.
   - In the case the child is placed in a new segment a segment_id is requested and a suitable memory room is allocated (via malloc). Then the previous step is performed to figure out the displacement.

   In both cases a reference is composed by using segment_id and displacement then returned.

[12] Array must be accessed through a language primitive

Segments may have different sizes, even in the same tree. Nodes in a segment are placed according with a customizable strategy. Currently eskimo run-time supports two strategies: heap and first-fit.

Figure 6.8 (top) sketches a spread tree allocated with heap policy. Starting from a node, heap strategy maximizes the likely of finding children nodes in the same segment while visiting the tree (in a random way). Moreover, a segment holding a heap allocated sub-tree exposes a very uniform fringe with respect to lower levels of the tree. Let us consider the *outer-tree*, i.e. the tree obtained by considering each segment (dashed boxes in the figure) as a node of *outer-tree* and all edges among different segments as edges of the *outer-tree*. The original tree and the *outer-tree* have the same shape, thus heap allocation keeps the shape of the tree. Unfortunately, heap policy suffers from internal fragmentation. As an example adding a level to the tree in Figure 6.8 lead to creation of 56 additional segments. These are mostly empty and filling percentage drops from 98% to 25%. Even a balanced tree cannot effectively stored blocked in this way. Fragmentation has a great impact both on memory load and performance. A spread tree composed by mostly empty segments destroys the spatial locality of accesses and pays additional overheads in communications among processing element.

Aiming of resolving the problem we implemented the first-fit policy. As can be imagined, the strategy is antithetical with respect to heap one. It offer a good "compression" of data but may destroy the shape of the tree. In Figure 6.8 (bottom) the same tree is blocked using first-fit strategy (except for the top part, as discussed in the following). In this case adding a level to the tree lead to creation of 8 additional segments and the filling percentage grows from 73% to 80%. In the general case filling percentage may grow or drop but in a limited way (even though some pathological cases exists). Moreover, during experiments with eskimo we observed that several application tend to allocate and visit nodes of a tree following similar paths. In such a case first-fit strategy works pretty well. The general effect is similar to tree-threading one, i.e. a technique that enriches the tree with additional edges forming a chain. The chain is then followed in order to speed up the tree visit. First-fit policy behaves similarly forming multiple chains during tree allocation, these links nodes stores in the same segment.

However, also first-fit strategy has its own drawbacks. Those come from the eskimo parallelism model. In order to limit the number of threads and scheduling actions, eskimo run-time takes scheduling actions during the segments border crossing only. Since an *e-function* references data (through `r/rw`) within the same segment no scheduling actions are taken, that means that possibly produced *e-flows* are sequentialized. This may lead to an unbalanced mapping of *e-flows* into threads. As an example consider *build and visit* program shown in Section 5.3. Let us suppose to use the first-fit strategy to store the tree starting from the root. The will be stored in a depth-first way. Since eskimo scheduling tries to execute first *e-flows* accessing to segments already present in the processing element memory (or cached), a pretty long path toward the leafs is explored before an *e-flow* is mapped in another

thread (either in the same processing element or another one).

In order to mitigate the problem eskimo uses a mixed strategy to allocate tree nodes. It alternates the two kinds of segments (heap and first-fit). In particular it use heap strategy in two cases:

- every time a node is allocated and the father of the node has another processing elements as home node;
- for the tree root and its direct children.

The strategy has been proven to be effective from fragmentation viewpoint and from performance viewpoint. Since all scheduling actions in eskimo are taken at the granularity of segments, the mixed strategy ensures both early scheduling decisions and a good locality in allocation.

Chapter 7

eskimo: experiments

Readers' road-map. This chapter reports experimental result obtained by using eskimo on a small test-suite. The test-suite includes some micro-benchmarks in order to test efficiency of some significant features of the run-time support, as for example address translation overhead. Test-suite also includes a significant application (a n-body simulation) in order to experiment both expressiveness and performance of the language on a dynamic application.

In order to assess eskimo performance, we performed a set of experiments on a Beowulf class Linux cluster operated at our department, as well as on a set of "production" workstations available at our department.

The cluster based experiments were aimed at demonstrating eskimo performance features, mainly. The cluster used for the experiments hosts 17 nodes: one node (`backus.cli.di.unipi.it`) is devoted to cluster management, code development and user interface. The other 16 nodes (ten 266Mhz Pentium II and six 400Mhz Celeron nodes) are exclusively devoted to parallel program execution. All the nodes are interconnected by a (private, dedicated) switched Fast Ethernet network. `backus` is a dual hosted node and provides access to the rest of the cluster node from Internet hosts. All nodes in the cluster run Linux Red-Hat 7.2 (kernel 2.4.7-10/gcc 2.96).

The production workstations used are a pair of dual 550MHz Pentium III Linux workstations running Linux Red-Hat 8.0 (kernel 2.4.18-14smp/gcc 3.2). The machines are interconnected by means of a plain 100Mbit Ethernet network that happens to be very busy all the time. All presented performance figures are sorted out by compiling applications with full optimization enabled (-O3 switch).

Moreover, speedup figures are figured out against "real sequential application", i.e. the pure C application implementing the same algorithm of the parallel version. Sequential version does not include eskimo nor any other DSMs or communication libraries.

7.1 Building and visiting a tree

In this section we present experimental figures of the application described in Section 5.3. The application builds a spread tree then visits it. In spite of its simplicity, it is a fairly important test since build and visit are the most common operations on trees.

Along this section we shall compare eskimo application against a sequential application written in pure C language. Both applications implement the same algorithm and store in the tree the same amount of data. However, it is worth noticing that the sequential application always writes and reads the *local* memory while eskimo application write and read *global* memory. Moreover, the application can not benefit in any way from temporal locality since read and write each node once. For the same reason, the application does not exploit spatial locality; however due to the tree node blocking operated by eskimo run-time support during node allocation the application can benefit from it.

We measured the overhead of reading/writing global memory with respect to local memory. Figure 7.1 shows the time needed to allocate a spread tree against several processing elements. The spread tree is a balanced binary tree holding 4M nodes, each node of the tree holds just three integers (it can be considered a pretty small node). A horizontal straight line (seq_build) in the figure denotes the time needed by the sequential version to allocate the same tree. The third line of the figure shows the ratio between eskimo and sequential applications allocation times. Since the application does nothing apart from reading and writing the memory, the test highlights library overheads due to global addresses translation and parallelism management (multithreading, communications, caching and consistency). As expected the sequential version is 3–4 times faster than the parallel version. However, since the ratio line decrease when the number of processing elements grows, we can conclude that the tree have been really distributed among nodes (the fact can be also verified consulting statistic information supplied by the library).

Tree visit (Figure 7.2) behave similarly to build. Observe that the complete tree have a size of 48 MBytes, even assuming the optimal network performance (12 MBytes/sec) and a perfect tree blocking, the time to move the tree among two nodes is about 4 secs. Since the visit completes in the same time on 12 processing elements, we can conclude that in the majority of the cases eskimo functions migrates toward data.

Figure 7.3 shows tree visit time for a 64k nodes balanced binary trees. The visit function run 37 μsecs (active) dummy load for each node of the tree in order to simulate a computation on the tree node data. As clear from the figure the application can profitably use up to 10 nodes of the cluster. Observe that the application runs on a single node is slightly slower than the pure sequential version. Figure 7.4 shows application speedup with respect to data plotted in Figure 7.3, the maximum speedup is reached on the ten processing elements configuration. Figure 7.3 and 7.6 reports application performance and speedup on the same load but on a bigger

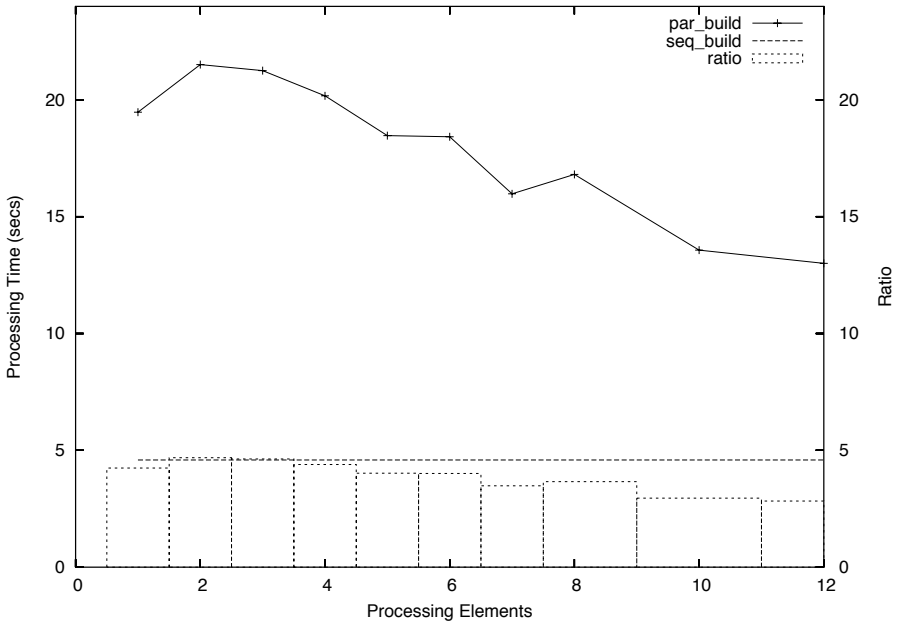


Figure 7.1: Overhead in tree building versus #PEs on `backus`. Balanced binary tree (depth 22, 4M nodes, 48MBytes).

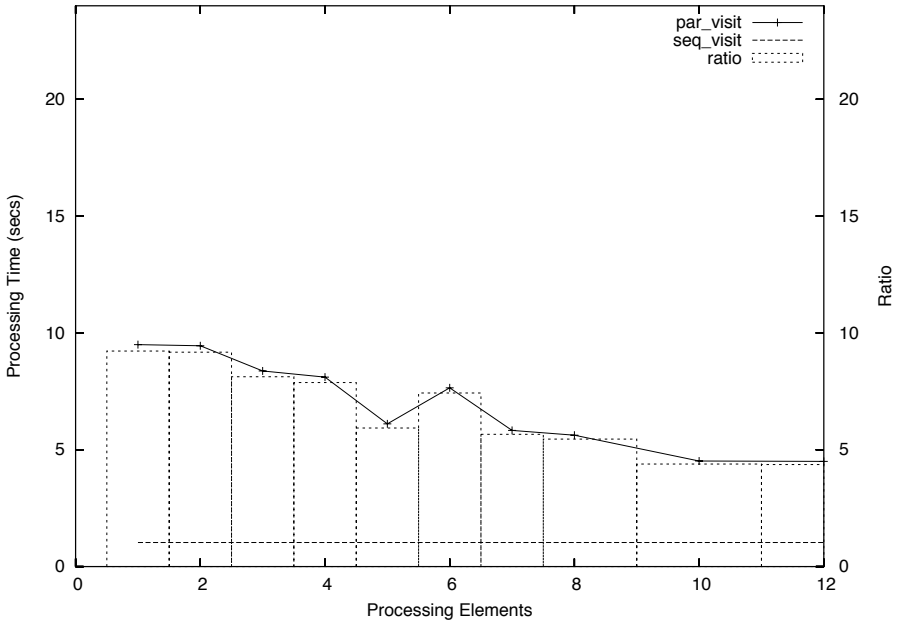


Figure 7.2: Overhead in tree visiting versus #PEs on `backus`. Balanced binary tree (depth 22, 4M nodes, 48MBytes).

tree (depth 20, 1M nodes). In this case the application speedup scales up the maximum number of nodes, proving that eskimo library respects iso-efficiency, i.e. bigger problems can be parallelized more than little ones.

Figure 7.7 shows the performance of runs on several cluster configuration against the dummy load associated to each node of the tree. eskimo parallel versions correctly linearly grows with respect to dummy load.

Exploiting multithreading

In clusters of single processor boxes (e.g `backus`) parts of eskimo capabilities get lost. eskimo is able to automatically turn independent *e-flows* both in parallel multiprocessing and multithreading. In the case the cluster includes SMP nodes eskimo support tries to use all processors in the SMP node and tries to keep eskimo functions working on the same data local to the SMP nodes as much as possible in order to exploit locality. Table 7.8 shows eskimo overhead on a pair of 2-way SMP nodes. Run-time overhead in this case is pretty high due to several reasons. In addition to eskimo overhead, the application pays O.S. inefficiencies in this case. As discussed, Linux pthread implementation is not particularly efficient, and most of Linux system tasks are really executed in mutual exclusion. Moreover, the memory bus of our (cheap) dual Pentium boxes are not always able to sustain the pressure of two processors continuously reading and writing the main memory. As shown in Table 7.9, performance figures becomes fairly good increasing the (active) load of the application.

7.2 N-body Barnes-Hut algorithm

As hierarchical techniques are applied to more and more problem domains, and applications in these domains model more complete and irregular phenomena, building irregular trees from leaf entries efficiently in parallel becomes more important. N-body problems are among the most important applications of tree-based simulation methods today. The performance of N-body applications has been well studied on two kinds of platforms: message passing machines [170, 149] and tightly coupled hardware cache-coherent multiprocessors [157]. Due to their irregular and dynamically changing nature, a coherent shared address space programming model has been argued to have substantial ease of programming advantages for them, and to also deliver very good performance when cache coherence is supported efficiently in hardware.

Although message passing may have ease of programming disadvantages, it ports quite well in performance across all these systems. This performance portability advantage may overcome the ease of programming advantages of the coherent shared address space model whether the latter cannot deliver good performance on commodity-based systems, so users in domains like tree-based N-body applications

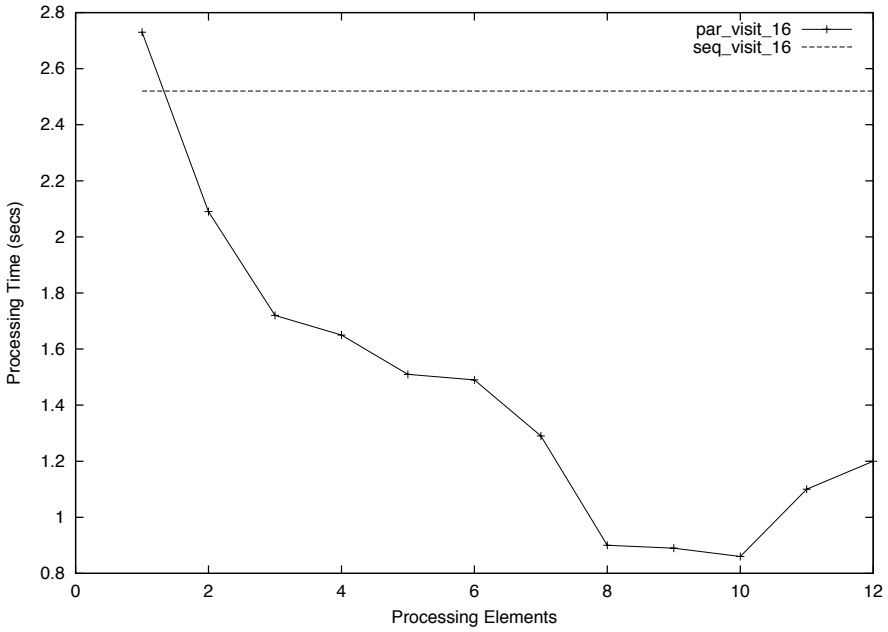


Figure 7.3: Tree visiting time versus #PEs on `backus`. Balanced binary tree (depth 16, 64k nodes, 768 KBytes, 37 μsecs of computational load per node).

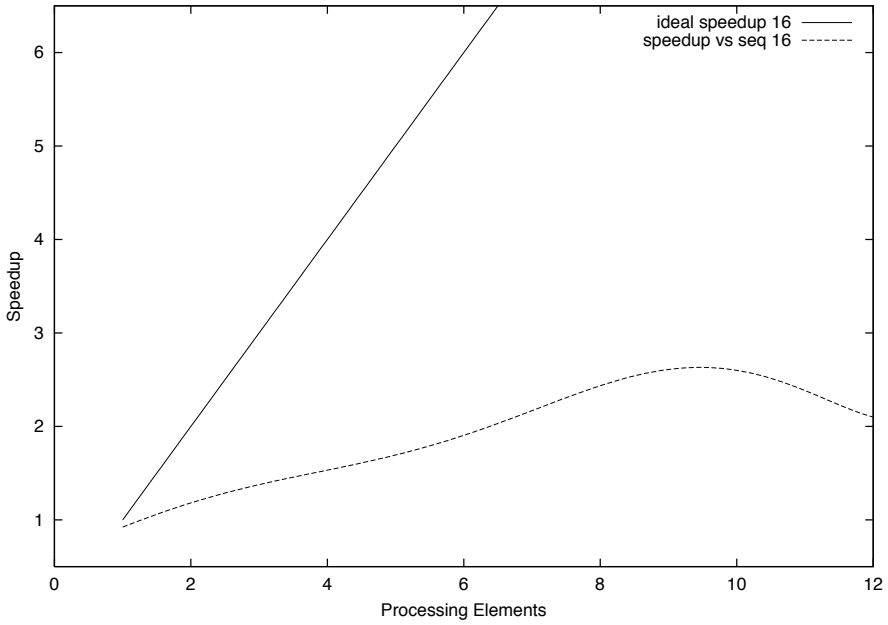


Figure 7.4: Tree visiting speedup on `backus`. Balanced binary tree (depth 16, 64k nodes, 768 KBytes, 37 μsecs of computational load per node).

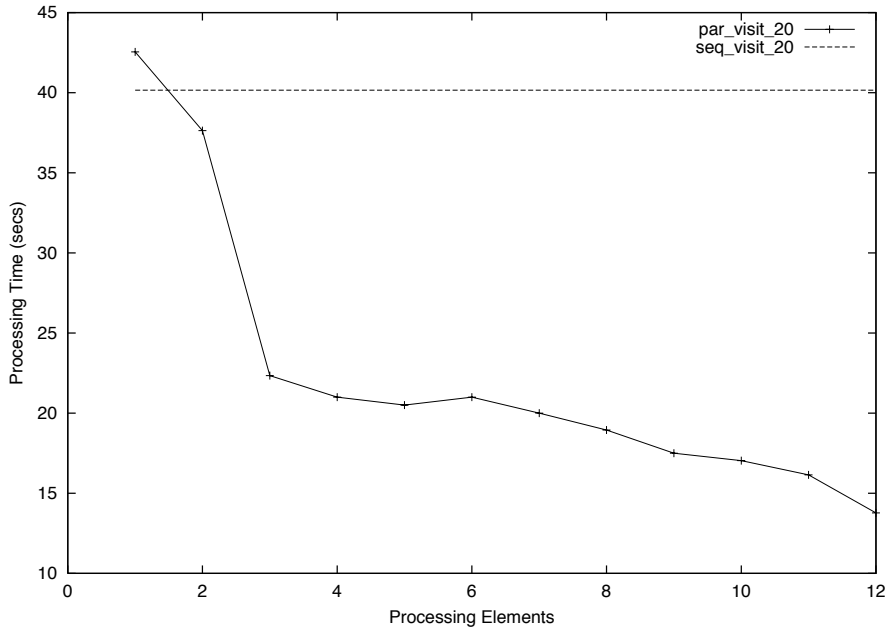


Figure 7.5: Tree visit time versus #PEs on `backus`. Balanced binary tree (depth 20, 1M nodes, 12MBytes, 37 μsecs of computational load per node).

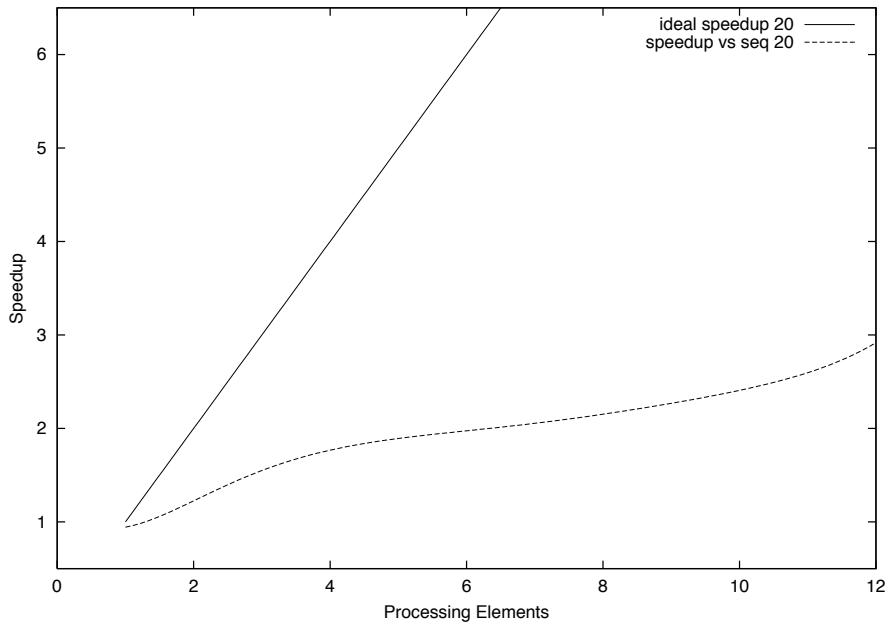


Figure 7.6: Tree visiting speedup on `backus`. Balanced binary tree (depth 20, 1M nodes, 12MBytes, 37 μsecs of computational load per node).

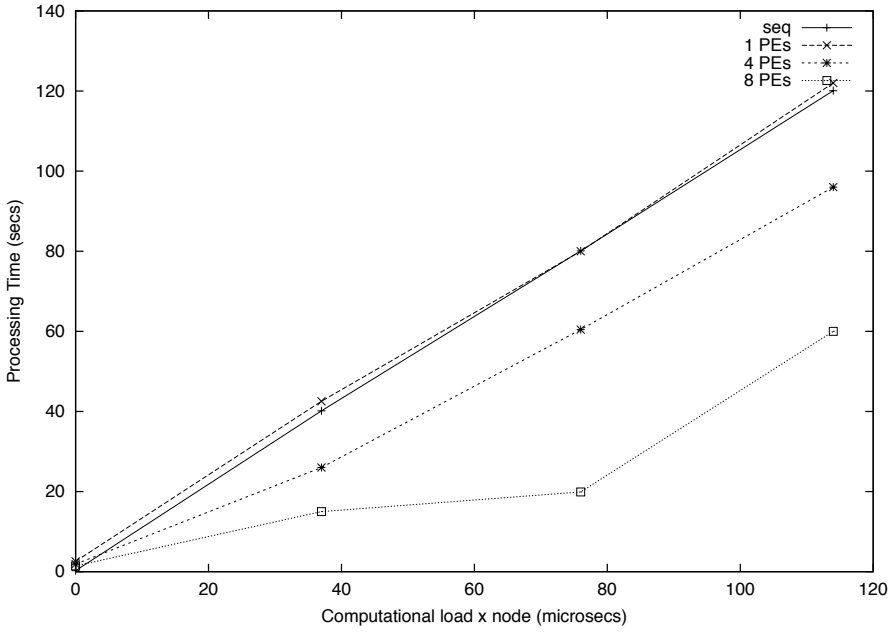


Figure 7.7: Tree visit time versus computational load on `backus`. Balanced binary tree (depth 20, 1M nodes, 12MBytes).

tree depth	16	18	20
# nodes	64k	256k	1M
size (MB)	768k	3M	12M
seq (sec)	0.01	0.03	0.15
1 x 2-way SMP (sec)	0.80	0.30	1.50
2 x 2-way SMP (sec)	0.40	0.15	0.70

Figure 7.8: Tree visiting overhead on a SMP cluster (2-way 550MHz PIII).

	comp. load	0 μsec	37 μsec	73 μsec	optimal
time	seq	0.03	9.55	19.01	–
(sec)	1 x 2-way SMP	0.30	7.03	12.07	–
	2 x 2-way SMP	0.15	4.80	8.51	–
speedup	1 x 2-way SMP	0.10	1.35	1.57	2
	2 x 2-way SMP	0.20	1.98	2.23	4
efficiency	1 x 2-way SMP	0.05	0.68	0.79	1
	2 x 2-way SMP	0.05	0.50	0.58	1

Figure 7.9: Tree visiting time, speedup and efficiency on a SMP cluster (2-way 550MHz PIII). Balanced binary tree (depth 18, 256k nodes, 3MBytes).

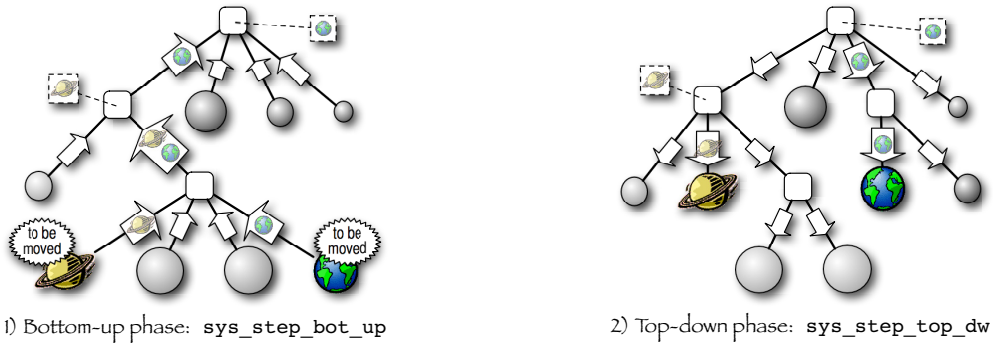


Figure 7.10: A n-body system step in two phases (force calculation phase, in two sub-phases: bottom-up and top-down).

sys_step_bot_up

```
1   eref_t sys_step_bot_up(eref_t anode) {
2   eref_t ret_array[4];
3   eiterator_t it;              /* An eskimo iterator */
4   eref_t float_list, sink_list; node_t *np;
5   efun_init();                 /* this is an eskimo function */
6   np = (node_t *) rw(anode); /* bind np to anode */
7   if (np->leaf) {    /* recursion base case */
8     < Figure out acceleration (visit the tree from root) >
9     < Update bodies position (np->x = ...; np->y = ...;) >
10    if (!within_borders(elem))
11      push(float_list,anode);
12  }
13  else {
14    /* Divide */
15    e_foreach_child(it,np) {
16      e_callit(sys_step_bot_up,it,foo,sizeof(foo));
17    }
18    e_joinall(it,ret_array);
19
20    /* Conquer */
21    for (i=0;i<4;i++)
22      while (elem=pop(ret_array[i]))
23        if (within_borders(elem))  /* read *np */
24          push(sink_list,elem);
25        else
26          push(float_list,elem);
27    < handle chain elimination and other particular cases >
28  }
29  return (float_list);
30 }
```

Figure 7.11: eskimo pesudo-code of the bottom-up phase, see also Figure 7.10

may prefer to use the more difficult model.

Barnes-Hut algorithm. Having specified the initial positions and velocities of the n bodies, the classical N-body problem is to find their positions after a number of time steps. In the last decade, several O(NlogN) algorithms have been proposed. The Barnes-Hut method [34] is the one widely used on sequential and parallel machines today; while the tree building issues and algorithms we discuss apply to all the methods, we use a 2-dimensional Barnes-Hut galaxy simulation as an example application. The sequential Barnes-Hut method has three phases in each time step of a simulation. In the first tree-building phase, a quad-tree is built to represent the distribution of all the bodies. This is implemented by recursively partitioning the space into eight sub-spaces until the number of particles in the subspace is below a threshold value. The lowest level cells contain bodies, and higher level cells contain the summary effect of the bodies in their rooted sub-trees. The root cell represents the whole computational space. The second phase computes the force interactions. In this phase, each body traverses the tree starting from the root. If the distance between the body and a visited cell is large enough, the whole sub-tree rooted at it will be approximated by that cell; otherwise, the body will visit all the children of the cell, computing their effect individually and recursively in this way. In the third phase each body updates its position and velocity according to the computed forces. The sequentially dominant phase is the force calculation phase (97%).

Barnes-Hut application is a good example to study because it presents non-trivial performance challenges, and it is relatively small and manageable.

7.2.1 Barnes-Hut experiments

We developed three different versions of the Barnes-Hut algorithm:

- The sequential version (seq), implemented in C language. It is a reduction to the bare bones of the original Barnes-Hut code in the bidimensional space.

- The eskimo version. It is obtained from the sequential version substituting recursive calls with recursive *e-foreach*es. Body information is stored directly in the tree instead of in an array. The n-body system evolves through two phases: (references to) bodies leaving their current quadrant are first lifted to the smallest quadrant including both source and target positions (bottom-up phase), then they are pulled down to target quadrant (top-down phase). Algorithm behavior is sketched in Figure 7.10. A snippet of the eskimo pseudo-code of the bottom-up phase is shown in Figure 7.11

- The C+MPI version. The body data is partitioned among nodes. The hierarchical relationship among bodies is maintained in a forest of trees. Each tree of the forest is linked to a "root" tree replicated in each PE. The structure

simulates a spread tree with the top part cached in each PE for a faster access. A processing element maintains the top part of the tree coherent and reads/writes other parts by exchanging messages with other processing elements.

We tested the three application versions on two different datasets: *cross* and *ellipse*. The two dataset have the same peculiarities of the classical *plummer* and *uniform distribution* models respectively. The two data distribution are represented in Figures 7.12 and 7.13. The two distribution may be hierarchically represented by a strongly-unbalanced and fairly-balanced trees respectively. These trees are depicted in the bottom parts of the two figures (in both cases the dataset includes 30 bodies, that are the leafs of the tree). In all cases, we use $\theta = 0.5$ as acceptance criterion[1]

Table 7.1 shows the performance figures of the three versions of the algorithm on the SMP cluster for four different datasets. Tables 7.2 and 7.3 reports the speedup ($\mathcal{S} = t_{seq}/t_{par}$) and efficiency ($\mathcal{E} = t_{seq}/(\#PE * t_{par})$) figures respectively relative to the same runs. The eskimo version of the code result as fast as the MPI version for the ellipse dataset and slightly but significantly better for the cross dataset (highlighted in the tables). The point here is that the performance MPI version does not scale up with the number of processing elements for the cross dataset. The tree unbalanced leads to a heavy load imbalance in the MPI version. Our version of the MPI code does not include a dynamic load balancing strategy but fixes the data distribution during the first iteration. It is certainly possible add a dynamic balancing strategy in the MPI code (even if not so easy), but it has to be explicitly programmed by the application programmer and specifically tailored for the problem. eskimo code instead can be written without any concern for load balancing and data mapping. As matter of facts the sequential version is just 300 lines of code, the eskimo version 500 and the MPI version 850.

[1]The value of θ has a strong impact on the performance of the algorithm, we refer back to the literature any other detail.

dataset x #bodies	cross x 10k	cross x 20k	ellipse x 10k	ellipse x 20k
seq	6.47	14.53	4.80	2.34
MPI 1 x 2-way SMP	6.60	14.36	2.54	1.27
MPI 2 x 2-way SMP	6.64	14.58	1.50	0.75
eskimo 1 x 2-way SMP	5.30	13.20	2.45	1.33
eskimo 2 x 2-way SMP	4.10	8.10	1.55	0.77

Table 7.1: Barnes-Hut performance (secs) on several ellipse and cross datasets for Barnes-Hut application (sequential, MPI and eskimo) on a SMP cluster (2-way 550MHz PIII).

dataset x #bodies	cross x 10k	cross x 20k	ellipse x 10k	ellipse x 20k	optimal
MPI 1 x 2-way SMP	0.9	1.0	1.9	1.8	2
MPI 2 x 2-way SMP	0.9	1.0	3.2	3.1	4
eskimo 1 x 2-way SMP	1.2	1.1	1.9	1.8	2
eskimo 2 x 2-way SMP	1.6	1.8	3.1	3.0	4

Table 7.2: Barnes-Hut speedup on several ellipse and cross datasets for Barnes-Hut application (sequential, MPI and eskimo) on a SMP cluster (2-way 550MHz PIII).

dataset x #bodies	cross x 10k	cross x 20k	ellipse x 10k	ellipse x 20k	optimal
MPI 1 x 2-way SMP	0.45	0.50	0.97	0.90	1
MPI 2 x 2-way SMP	0.22	0.25	0.80	0.77	1
eskimo 1 x 2-way SMP	0.60	0.55	0.95	0.90	1
eskimo 2 x 2-way SMP	0.40	0.45	0.77	0.75	1

Table 7.3: Barnes-Hut efficiency on several ellipse and cross datasets for Barnes-Hut application (sequential, MPI and eskimo) on a SMP cluster (2-way 550MHz PIII).

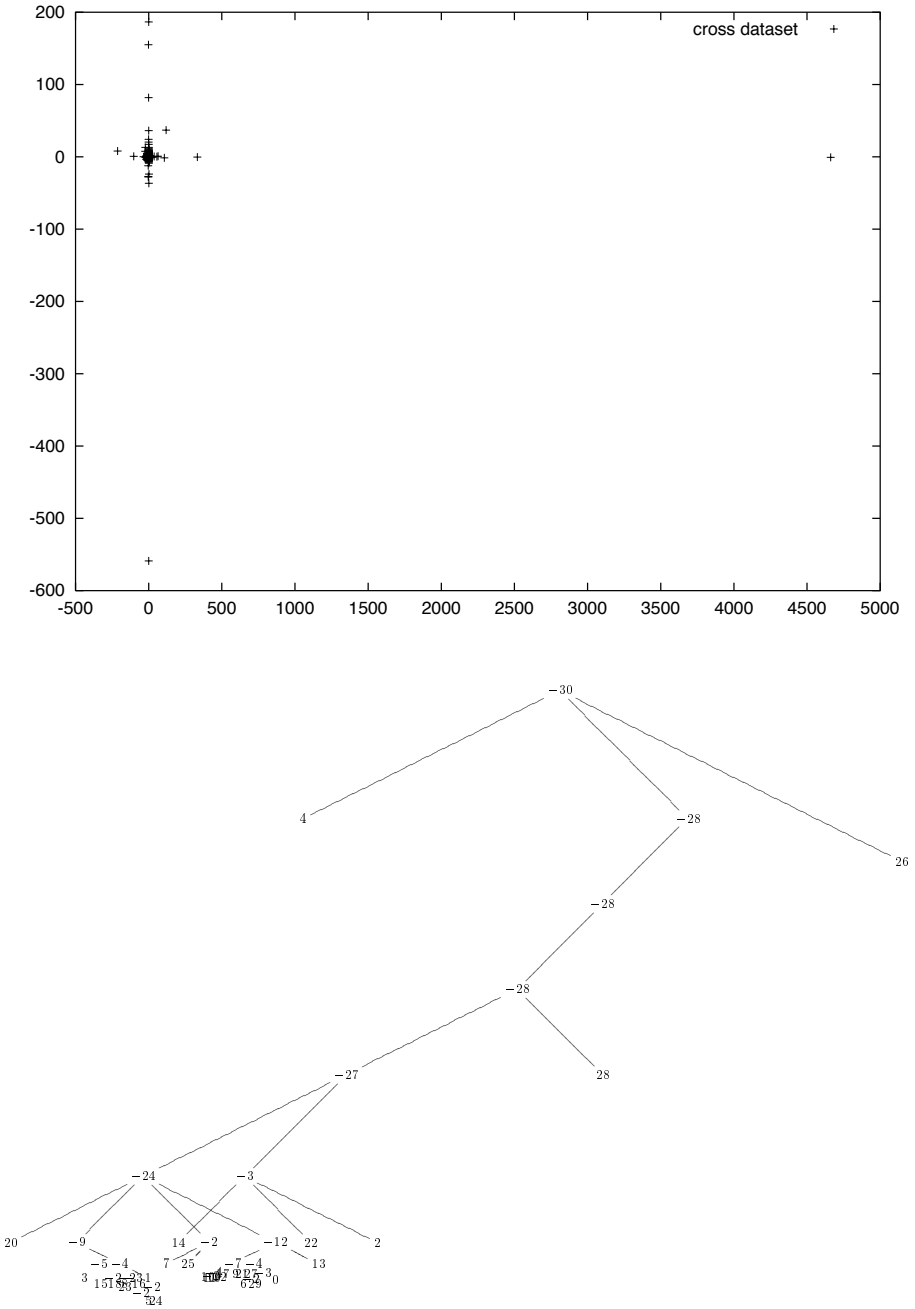


Figure 7.12: Cross dataset for the Barnes-Hut application and its hierarchical representation. Positive numbers represents leafs while negative numbers represents the number of leafs dominated by the node.

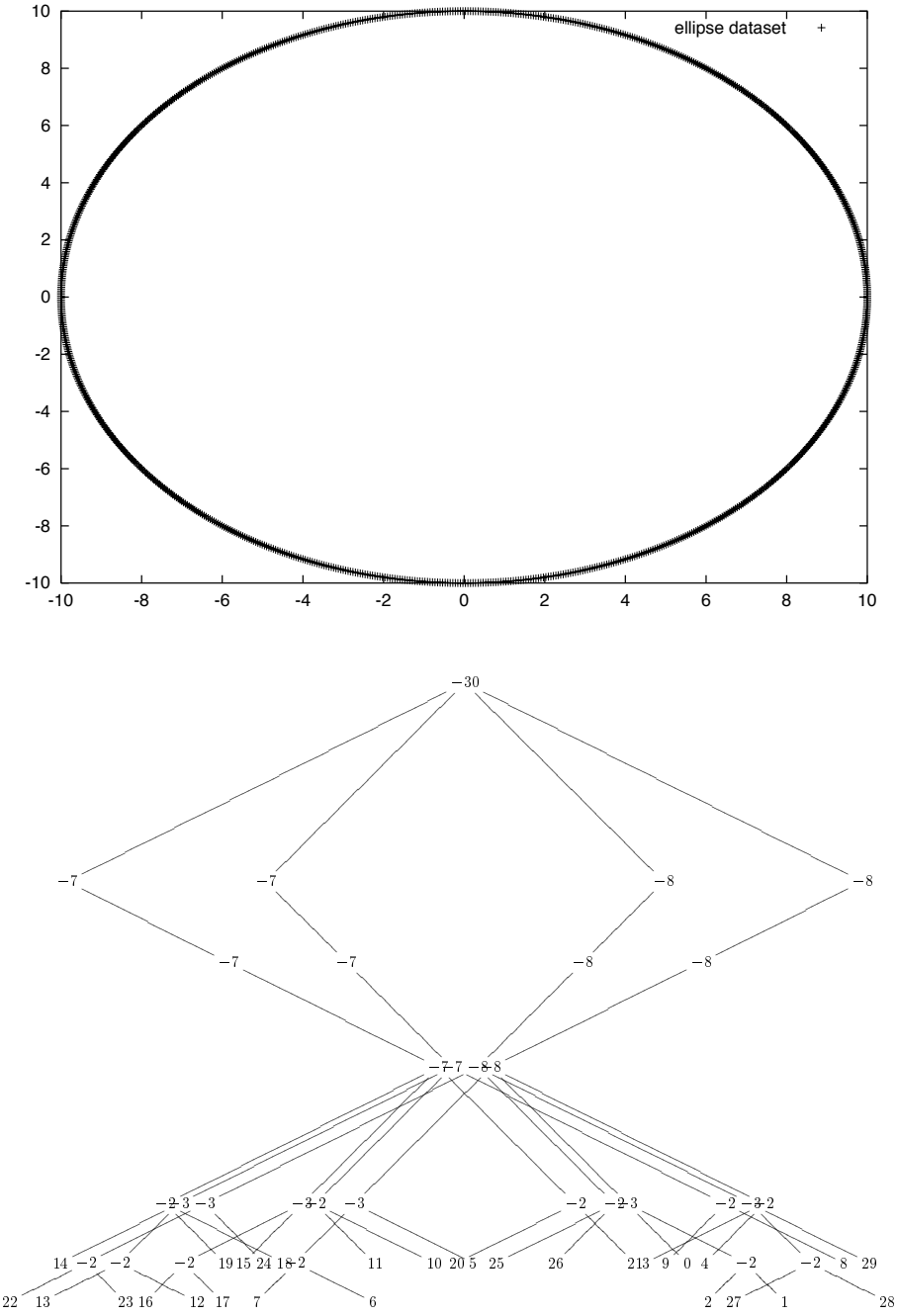


Figure 7.13: Ellipse dataset for the Barnes-Hut application and its hierarchical representation. Positive numbers represents leafs while negative numbers represents the number of leafs dominated by the node.

Chapter 8

Discussion and concluding remarks

Readers' road-map. The chapter summarizes the materials contained in the previous chapters and discusses the conclusions of the thesis. The extent to wish the goals of the thesis have been met is discussed. Finally the future work related to the thesis is introduced.

Our research experience at computer architecture and high-performance computing lab of Pisa Computer Science Dept. embraces several years of activity. Along these years, the parallel computing is rapidly evolved, and along with our research projects. We shortly presents main results achieved in these years by framing them in thesis chapters and published papers. These should be considered as snapshots taken from a continuously evolving process, currently focused at eskimo programming framework, but still underway. eskimo assessments are enumerated in Section 8.1 then discussed in Section 8.2. Eventually, we propose eskimo planned evolutions and future wishes.

Summary. I participated to the design or development of the following programming environments:

- the SkIE programming environment and its compiler [7, 14]. Section 2.1.
- FAN, a functional skeletal parallel programming framework [18].
- Lithium, a Java parallel programming environment. Section 2.3 [15, 16, 17].
- The ASSIST programming environment. Section 2.4 [11, 12, 13].

also, I designed and developed the following programming platforms:

- the Meta optimization tool for skeleton-based languages and the Skel-BSP skeletal language and its run-time. Section 2.2 [8, 9].
- The eskimo language and its run-time support. Chapters 4, 5 and 6 [10].

8.1 Assessments

The original contribution of the thesis can be summarized in the following elements:

- We designed and developed eskimo: a parallel extension of the C language based on the shared address model and running on cluster of workstations. We showed that irregular parallel applications using dynamic data structures can be handily coded in eskimo. These applications exploit a good performance even when compared with their hand-tuned MPI versions.

- eskimo supports dynamic shared data structures in a parallel framework. Those spring from the instantiation of simple parametric types including trees, arrays and regions. Shared variables can be dynamically and incrementally allocated. These are managed in segments in order to match target architecture working grain. The eskimo programming model enables different processing elements to allocate different parts of the same data structures without synchronizing one another.

- eskimo introduces data/task co-scheduling. Both shared variables and function calls may be moved one towards the processing element holding the other or vice-versa depending on system the status and according to a configurable scheduling policy.

- eskimo programming model frames skeletal programming in the shared address space. It abstracts the programming model with a specific concern for shared memory programming in a distributed framework by seeking efficiency through locality of memory accesses. It can spring from programmer hints, scheduling policy, data mapping, data caching, and eventually from the orchestration of above-mentioned issues. To the best of our knowledge this is an original interpretation of the skeletal approach in parallel programming.

These assessments are further discussed in the following section.

8.2 Discussion

The thesis deals with two facets of parallel programming that at first glance seem pretty distant one another: Structured programming (in particular in skeleton-based languages) and shared address programming (in particular in DSMs, that are surveyed and discussed in Chapter 3).

eskimo is a parallel extension of C language based on the shared address programming model. C programming model is extended with distributed data structures (i.e. *Shared Abstract Data Types*) and flows of control (i.e. *e-flows*) in a parallel/distributed framework. These are actually abstractions of the classic concepts of concurrent programming. In turn, these abstractions enable the programmer to

design programs for loosely coupled distributed platforms by relying on concurrent programming pragmatical experience, i.e. the experience in designing programs exploiting fork/join-like primitives and the shared memory concept.

We designed three kinds of *Shared Abstract Data Types*: spread k-trees, spread arrays, and shared regions. They are simple parametric types that can be instanced with a C type to obtain shared variables. These are spread across the eskimo virtual architecture and may grow beyond the limit of the single processing element (and beyond its logical address space). Nevertheless shared variables are conveniently presented to the programmer as single entities. Moreover, they can be statically or dynamically allocated. In particular trees can be dynamically allocated node-by-node (as usual in C programs). Shared variables may be reached through *references*, i.e. addresses in the shared space. Using references and k-trees a non-native form of graphs may be also represented in the shared memory (by enriching their spanning tree with additional edges implemented as references in the node body). Lists are actually 1-trees. The language run-time decouples the fine-grain programmer's view of shared variables from the target architecture representation. It dynamically groups data-items in order to reach a suitable (coarse) working grain for the target architecture. The grouping is performed (in segments) in a pragmatically-significant manner in order to enhance locality in typical access patterns of each data type.

eskimo provides the programmer with the *e-flow* concept and the primitives to split and join *e-flows*, i.e. the *e-call*/*e-join* constructs and their n-way extensions *e-foreach*/*e-joinall*. These primitives work on data collections exploiting data parallelism, thus they are skeleton themselves. In addition these can be composed in simple code patterns to build many variants of data-parallel and control-parallel skeletons (as sometimes Divide&Conquer is considered [120]), or either interleaved with external communication primitives (e.g. POSIX pipes, sockets, ...).

It is worth observing that eskimo is a skeletal language even without *e-foreach*/*e-joinall*. As discussed in Sections 1.2.2, 2.1 and 4.1, *skeletal programming would simplify programming by raising the level of abstraction, providing the programmer with performance and portability for their applications. But skeletal programming is not functional programming, even though it may be concisely explained and expressed as such. Skeletal programming is not object oriented programming, even though this may be a similarly attractive vehicle.* eskimo raises C programming model level of abstraction by introducing *e-flows* and *shared variables*. They do not directly match any entity at underlying implementation level. These are entities playing scheduling/mapping game: they can be moved one towards the other or vice-versa according to current system status with the aim of improving data accesses locality. Programmer insight on the algorithm play a major role in the game. In order to obtain an efficient code the programmer ought to design the algorithm along these guidelines:

- A considerable amount of independent *e-flows* should be exploited both during data allocation and data access. In the former case the best bet consist in using

e-foreach/*e-joinall* primitives.

- Data items exploiting a medium/strong temporal correlation in the access (read or write) should be allocated along the same *e-flow*. This boosts the likely that accesses following the first one will pay a short latency access time.

- The first parameter of *e-functions* should be used to refer the shared variable mostly accessed along the function. The eskimo run-time use this information to make *e-flow* scheduling decisions. In the case this parameter is `E_NULL` the run-time uses a heuristic scheduling policy based on system statistics (like workload and memory status on the processing element in the system).

In addition the skilled programmer may experiment his own scheduling policy by modifying the scheduling function (in the top tier of eskimo run-time).

eskimo tries to manage tasks scheduling and data/processeses/threads mapping in an efficient way. These aspirations are common to a number of models which have proved very successful within the wider world of software DSM, software engineering, object oriented programming and design patterns. Indeed eskimo is pretty similar to *Cilk* and *Athapascan* at the language level. Despite specific differences (discussed along Chapters 4, 5, 6) in memory consistency model, target architectures, implementation technique, the differences between eskimo and the others mentioned research works can be summarized as follows:

- It has dynamic data structures. These can be dynamically and incrementally allocated.

- It has a configurable scheduling policy. It does not rely on work stealing (as *Cilk* and *Athapascan*). Work stealing has load balancing as first target. eskimo tries to exploit a mapping/scheduling policy that takes in account also shared memory speed access (thus network latency and bandwidth) and memory usage distribution (in term of data item and their cached copies). These targets are pursued by trying to exploit data accesses locality by means of the co-scheduling of shared data and *e-flows*(relying on programmer hints).

- It offers a slightly higher-level constructs with respect to others environments (such as *e-foreach*/*e-joinall*). These constructs abstract the implicit nondeterminism in the execution order of independent *e-flows*.

- It is a skeletal language. It applies the skeletal abstraction to the shared memory model by proposing skeletons as medium to take care of data accesses efficiency, that ultimately springs from a suitable data allocation and tasks scheduling. To the best of our knowledge the first one relying on the shared address space.

Overall, programs exploiting recursive algorithms (e.g. Divide&Conquer) on dynamic data structures (e.g. k-tree) can be written in eskimo and run on a cluster of workstations. This class of application can hardly be written by using our previous programming frameworks (or not written at all). Yet eskimo has a fairly efficient run-time support: eskimo Barnes-Hut application (surely an application in the class) runs with an equal or better performance than the hand-tuned MPI version. The eskimo source code is more compact than MPI code and has no lines of code dedicated to load and memory space balancing.

8.3 Future works

As discussed eskimo is an experiment. There are a lot of points that can be improved:

The language engineering.

- The syntax. Almost the half of Barnes-Hut code are initializations (*e-functions*, *shared variables*, handlers, iterators).
- Type checking. The language heavily relies on void pointers. These are mainly used to overcome the C lack of polymorphism.
- The scheduling configuration. In order to change the scheduling policy the programmer must change and recompile the library.

These points may be addressed by moving to an OO framework. Initialization may be included in the class constructors and we can rely on the native ability of the language to cope with parametric types. Eventually, scheduling configuration hooks may be exposed to the programmer exploiting OO class visibility mechanisms.

The language support engineering. eskimo programs does not rely on any compiling phase. Almost all choices are taken at run-time. This choice comes from the wish to make eskimo a C library. However, as shown also by some experiments, this choice has a heavy impact on programs performance. The problem may be partially addressed by moving to an object oriented language as C++. C++ templates may enable the migration of some of run-time choices at compile time (as an example leveraging on ad-hoc polymorphism). Some function calls may be statically replaced with their "fast" versions, i.e. pure sequential versions with any concern for scheduling and mapping. Clearly, the adoption of an OO language does not resolve all the problems by itself. A clearer decoupling between static and dynamic aspects of the run-time must be performed.

The language run-time exploits two kinds of mechanisms for data exchange and synchronizations: POSIX TCP/IP stack for inter-node communications and POSIX threads for intra-node sharing. These mechanism must be generalized, especially

in the light of the ever increasing role of intelligent/communication boards. A core of functionalities must be selected and wrapped into a proper layer providing the needed decoupling from communication and synchronization mechanisms and the language run-time. Functionalities may include event handling and thread pool management also (as for example in the ACE library [1]).

The language programming model. eskimo is based on DAG memory consistency model. As discussed in Section 3.3.2 is a "memory centric" very lazy consistency model. As shown in Chapters 4 and 5 it enables the programmer to write a parallel program without any concern for processing elements but for the algorithm only. But it prevents the exploitation of some common behavior a programmer typically expects from a shared memory. Actually the DAG consistency semantics seems to give us a data model which is in one sense on the very fringe of what might be considered "shared memory". Other memory consistencies have similar problems (as for example entry release consistency adopted by *Athapascan*). It would interesting to see how the implied extra effort from the programmer trades off with performance achieved in comparison to one of the more conventional DSM models. It would interesting also if it is possible to change the memory consistency while maintaining the *e-flow* concept. From a preliminary evaluation an object-based memory consistency may be well suited for that (even better if this consistency may be finely tuned, or its behavior may be co-designed with abstract data types).

Currently, the *e-flows* scheduling is just intuitively described and implemented. The implementation relies on several run-time constants to manage fall-back heuristic scheduling. A more formal description of the programming model is needed. This should enable the formalization of the scheduling (and in turn of an execution cost model). Currently a cost model does not exist for eskimo. The *Cilk* programming model is close enough, but the scheduling completely differs (thus the cost model). Malleable Task model is also close enough to eskimo programming model. For the "Malleable Task" model a good scheduling formalizations exists [125, 126] that can be used as starting point (even if some assumptions often made on this model are hardly satisfied in eskimo, as an example the "monotonus penalty assumptions" [39]).

8.4 The ASSIST perspective

Currently, our research group efforts are mainly focused to refining and testing ASSIST design. Current ASSIST version [13] includes a software DSM not supporting dynamic data structures (DVSA [31], a non-coherent non-consistent DSM). A re-engineered version of eskimo should replace DVSA, thus introducing dynamic data structures as native objects into ASSIST. Currently the integration process is in progress. The re-engineering takes in account the ASSIST framework (that has its communication/synchronization layer) and takes advantages of ASSIST compiler in

order to transfer some run-time operation at compile time.

Moreover, a brand new version of ASSIST in under design within the just started, three years, FIRB project *Grid.it*. This new version has the ambitious aim to design a high-performance environment to develop programs running on the GRID.

In general, the applicability of DSM to large-scale grids is rather disputable (even if it has been argued that providing a shared memory abstraction can offer new services and capabilities to GRID programming environments. For instance fault tolerance (by means of data replication and/or checkpointing), and persistence could be handled transparently at the DSM level [141]). Anyway, as a limited extent, shared memory model can be thought as part of the bulk of technologies needed to extract high-performance (in the widest meaning of processors performance, memory room, network bandwidth smart usage, etc.) from the GRID. A first step toward this end may consist in:

- Enclosing shared data into objects (as an example CORBA objects as in [142]). This solution may also benefit from object DSM experience.
- Consider the shared data confined within a group of processing elements exploiting particular properties in the GRID world such as: clusters of trusted processing elements logically running a given component or service.

We believe that some of the features exploited by eskimo might have interesting developments in the GRID world. In particular, the programming model and the very lazy memory consistency force the programmer to think to the algorithm as a collection of almost independent tasks (we envision this programmer as the run-time support designer not as the application programmer). These tasks are split in a completely distributed fashion (typically in a DAG fashion), without any centralization point. Moreover, eskimo programs never assumes to deal with processing elements, but rather with the pretty abstract concept of the *e-flow*. *e-flows* are neither fixed in number nor a priori bound to any processing element. The program unfolding is step-by-step mapped on the concrete architecture. This mapping may easily support a dynamic, reconfigurable architecture platforms. Nowadays nobody probably known what the GRID will be exactly, but everybody agree that it will be a dynamic world.

Bibliography

[1] The ACE team. *The Adaptive Communication Environment home page*, 2003. (`http://www.cs.wustl.edu/~schmidt/ACE.html`).

[2] Advanced Micro Device Inc. *AMD Athlon^{TM} Processor Architecture*, white paper edition. (Available at `http://www.amd.com/`).

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[4] S. V. Adve, V. S. Pai, and P. Ranganathan. Recent advances in memory consistency models for hardware shared-memory systems. *Proc. of the IEEE, special issue on distributed shared-memory*, 1999.

[5] A. Agarwal, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT alewife machine: A large-scale distributed-memory multiprocessor. In *Proc. of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic, 1991.

[6] J. Ahrens, P. Beckman, and K. Keahey. Ligature: component architecture for high performance applications. *The International Journal of High Performance Computing Applications*, 14(4):347–356, 2000.

[7] M. Aldinucci. Design and validation of sequential, `pipe` & `farm` SkIE templates. Technical report, PQE2000 Project – Consorzio Pisa Ricerche, December 1997.

[8] M. Aldinucci. The Meta transformation tool for skeleton-based languages. In S. Gorlatch and C. Lengauer, editors, *Proc. of the 2nd International Workshop on Constructive Methods for Parallel Programming (CMPP2000)*, pages 53–68. Fakultät für mathematik und informatik, Uni. Passau, Germany, July 2000.

[9] M. Aldinucci. Automatic program transformation: The Meta tool for skeleton-based languages. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation: Theory and Practice, chapter 5, pages 59–78. Nova Science Publishers, NY, USA, 2002.

[10] M. Aldinucci. eskimo: experimenting with skeletons in the shared address model. *Parallel Processing Letters*, 13(3), 2003.

[11] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. A framework for experimenting with structured parallel programming environment design. Proc. of the International Conference ParCo2003 (to appear), 2003.

[12] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. ASSIST demo: a high level, high performance, portable, structured parallel programming environment at work. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Proc. of the Euro-Par 2003*, number 2790 in Lecture Notes in Computer Science, pages 1295–1300. Springer, August 2003.

[13] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. The implementation of ASSIST, an environment for parallel and distributed programming. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Proc. of the Euro-Par 2003*, number 2790 in Lecture Notes in Computer Science, pages 712–721. Springer, August 2003.

[14] M. Aldinucci, M. Coppola, and M. Danelutto. Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. In S. Gorlatch, editor, *Proc of the 1st International Workshop on Constructive Methods for Parallel Programming (CMPP'98)*, pages 44–58. Fakultät für mathematik und informatik, Uni. Passau, Germany, May 1998.

[15] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimization. In *Proc. of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'99)*, pages 955–962, Cambridge, Massachusetts, USA, November 1999. IASTED/ACTA press.

[16] M. Aldinucci and M. Danelutto. An operational semantics for skeletons. Proc. of the International Conference ParCo2003 (to appear). Draft available as University of Pisa Tech. Rep. TR-02-13.
(`ftp://ftp.di.unipi.it/pub/Papers/aldinuc/TR-02-13.ps.Z`), 2003.

[17] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003.

[18] M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Towards parallel programming by transformation: The FAN skeleton framework. *Parallel Algorithms and Applications*, 16(2–3):87–122, 2001.

[19] G. Antoniu and L. Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In *Proc. of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)*, number 2026 in Lecture Notes in Computer Science, pages 55–70. Springer-Verlag, April 2001.

[20] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Compiling multithreaded Java bytecode for distributed execution. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Proc. of Euro-Par 2000*, number 1900 in Lecture Notes in Computer Science, pages 1039–1052. Springer-Verlag, September 2000.

[21] G. Antoniu, L. Bougé, R. Namys, and C. Pérez. Compiling data-parallel programs to a distributed runtime environment with thread isomigration. *Parallel Processing Letters*, 10(2–3):201–214, June 2000.

[22] G. Antoniu, L. Bougé, and R. Namyst. An efficient and transparent thread migration scheme in the PM2 runtime system. In *Proc. of the 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, number 1586 in Lecture Notes in Computer Science, pages 496–510. Springer-Verlag, April 1999.

[23] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *Proc. of the International Conference on Parallel Processing*, September 1999. (see also `http://www.haifa.il.ibm.com/projects/systems/cjvm/index.html`).

[24] D. Arlia and M. Coppola. Experiments in parallel clustering with DBSCAN. In *Proc. of Euro-Par 2001*, number 2150 in Lecture Notes in Computer Science. Springer-Verlag, August 2001.

[25] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high performance scientific computing. In *Proc. of the 8th International Symposium on High Performance Distributed Computing (HPDC'99)*, 1999.

[26] P. Au, J. Darlington, M. Ghanem, Y. Guo, H.W. To, and J. Yang. Coordinating heterogeneous parallel computation. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proc. of Euro-Par 1996*, pages 601–614. Springer-Verlag, 1996.

[27] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[28] B. Bacci, M. Danelutto, S. Pelagatti, S. Orlando, and M. Vanneschi. Unbalanced computations onto a transputer grid. In *Proc. of the 1994 Transputer Research and Application Conference*, pages 268–282. IOS Press, October 1994.

[29] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25(13–14):1827–1852, December 1999.

[30] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8), August 1978.

[31] F. Baiardi, D. Guerri, P. Mori, L. Moroni, and L. Ricci. Two layers distributed shared memory. In *Proc. of High Performance Computing and Networking Europe (HPCN2001)*, number 2110 in Lecture Notes in Computer Science, 2001.

[32] F. Baiardi and M. Vanneschi. *Linguaggi per la programmazione concorrente*. Franco Angeli, 1992.

[33] H. E. Bal and M. Heines. Approaches for integrating task and data parallelism. *IEEE Concurrency*, 6(3):74–84, 1998.

[34] J. Barnes and P. Hut. A hierarchical o(nlogn) force calculation algorithm. *Nature*, 324, 1986.

[35] P. Becuzzi, M. Coppola, S. Ruggieri, and M. Vanneschi. Parallelisation of C4.5 as a particular divide & conquer computation. In *Proc. of the 3rd Workshop on High Performance Data Mining*, number 1800 in Lecture Notes in Computer Science. Springer-Verlag, May 2000.

[36] R. Belli and M. Cappagli. Il multithreading nei sistemi a parallelismo massiccio. Master's thesis, Computer Science Department, University of Pisa, Italy, Italy, October 1999. (In italian).

[37] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proc. of the 38th International Computer Conference (COMPCON'93)*, pages 528–537. IEEE, February 1993.

[38] R. S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computing Science*. NATO ASI Series, 1988.

[39] E. Blayo, L. Debreu, G. Mounié, and D. Trystram. Dynamic load balancing for ocean circulation with adaptive meshing. In *Proc. of Euro-Par'99*, number

1685 in Lecture Notes in Computer Science, pages 303–312. Springer-Verlag, 1997.

[40] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical and Computer Science, Massachusetts Institute of Technology, U.S.A., September 1995. MIT/LCS/TR-677.

[41] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. of the 8th Annual Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 297–308. SIGARCH, ACM, June 1996.

[42] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *Proc. of the 10th International Parallel Processing Symposium (IPPS'96)*. IEEE, April 1996.

[43] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, July 1995.

[44] O. Bonorden, N. Hüppelshäuser, B. Juurlink, and I. Rieping. *PUB Library. User Guide and Function Reference (release 7.0)*. University of Paderborn, Fürstenallee 11, 33102 Padeborn, Germany, December 1999. (`http://www.uni-paderborn.de/~pub`).

[45] G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proc. of the 5th International Symposium on High Performance Distributed Computing (HPDC'96)*, pages 243–252. IEEE Computer Society Press, 1996.

[46] D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *Proc. of the 23rd International Symposium on Computer Architecture (ISCA'96)*. SIGARCH, ACM, May 1996.

[47] H. Burkhart and S. Gutzwiller. Steps towards reusability and portability in parallel programming. In K. M. Decker and R. M. Rehmann, editors, *Programming Environment for Massively Parallel Distributed Systems*, pages 147–157. Birkhäuser, April 1994.

[48] G. Carletti and M. Coppola. Structured parallel programming and shared objects: experiences in data mining classifiers. In G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, editors, *Parallel Computing: Advances and Current Issues. Proceedings of the International Conference ParCo2001*, pages 409–416. Imperial College Press, 2002.

[49] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proc. of the 13th Symposium on Operating Systems Principles (SOSP'91)*, pages 152–164. ACM, October 1991.

[50] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Trans. on Computer Systems*, 13(3):205–243, August 1995.

[51] G. Cavalheiro, M. Doreille, F. Galilee, T. Gautier, and J.-L. Roch. Scheduling parallel programs on non-uniform memory architectures. In *Proc. of the Workshop on Parallel Computing for Irregular Applications (WPCIA1)*, Orlando, USA, January 1999.
(`http://www-id.imag.fr/Laboratoire/Membres/Roch_Jean-Louis/perso.html/publications.html`).

[52] R. Chandra, K. Gharachorloo, V. Soundararajan, and A. Gupta. Performance eval of hybrid hardware and software distributed shared memory protocols. In *Proc. of the 8th International Conference on Supercomputing*, pages 274–288. IEEE, July 1994.

[53] M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng. Integrated support for task and data parallelism. *The International Journal of Supercomputer Applications*, 8(2):80–98, Summer 1994.

[54] W. H. Chou, C. T. King, and L. M. Ni. Pipelined data-parallel algorithms: Part I – Concept and Modeling. *IEEE Transactions on Parallel and Distributed Systems*, 1(4), October 1990.

[55] W. H. Chou, C. T. King, and L. M. Ni. Pipelined data-parallel algorithms: Part II - Design. *IEEE Transactions on Parallel and Distributed Systems*, 1(4), October 1990.

[56] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. ANACLETO: a template-based P3L compiler. In *Proc. of the PCW'97*, 1997.

[57] J. Cohen. Non-deterministic algorithms. *ACM Computing Surveys*, 11(2):79–94, June 1979.

[58] M. Cole. `eSkel` *library home page*.
(`http://www.dcs.ed.ac.uk/home/mic/eSkel`).

[59] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[60] M. Cole. Bringing skeletons out of the closet. Technical report, Institute for Computing Systems Architecture, Division of Informatics, University of Edinburgh, 2002.
(`http://www.dcs.ed.ac.uk/home/mic/eSkel/eSkelmanifesto.ps`).

[61] M. Cole, S. Gorlatch, J. Prins, and D. Skillicorn, editors. *High Level Parallel Programming: Applicability, Analysis and Performance*. Dagstuhl-Seminar Report 238, Schloß Dagstuhl, 1999.

[62] Computer Science Dept., University of Pisa. Lithium *home page*, 2001.
(`http://www.di.unipi.it/~marcod/Lithium`).

[63] M. Coppola and M. Vanneschi. High performance data mining with skeleton-based structured parallel programming. *Parallel Computing*, 28(5):793–813, May 2002.

[64] S. Crocchianti, A. Laganà, L. Pacifici, and V. Piermarini. Parallel skeletons and computational grain in quantum reactive scattering calculations. In G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, editors, *Parallel Computing: Advances and Current Issues. Proceedings of the International Conference ParCo2001*, pages 91–100. Imperial College Press, 2002.

[65] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel computer architecture. A hardware/software approach.* Morgan Kaufmann, 1999.

[66] P. D'Ambra, M. Danelutto, D. di Serafino, and M. Lapegna. Advanced environments for parallel and distributed applications: a view of current status. *Parallel Computing*, 28(12):1637–1662, December 2002.

[67] P. D'Ambra, M. Danelutto, D. di Serafino, and M. Lapegna. Integrating MPI-based numerical software into an advanced parallel computing environment. In *Proc. of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 283–291. IEEE, 2003.

[68] M. Danelutto. Dynamic run time support for skeletons. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and H. J. Sips, editors, *Proc. of the International Conference ParCo99*, volume Parallel Computing Fundamentals & Applications, pages 460–467. Imperial College Press, 1999.

[69] M. Danelutto. Task farm computations in Java. In Buback, Afsarmanesh, Williams, and Hertzberger, editors, *High Performance Computing and Networking Europe (HPCN2000)*, number 1823 in Lecture Notes in Computer Science, pages 385–394. Springer-Verlag, May 2000.

[70] M. Danelutto. Efficient support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(1):41–56, March 2001.

[71] M. Danelutto. On skeletons and design patterns. In G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, editors, *Parallel Computing: Advances and Current Issues. Proceedings of the International Conference ParCo2001*, pages 425–432. Imperial College Press, 2002.

[72] M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. Parallel functional programming with skeletons: the OCAMLP3L experiment. In *ACM Sigplan Workshop on ML*, pages 31–39, 1998.

[73] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for data parallelism in P3L. In C. Lengauer, Griebl, and S. Gorlatch, editors, *Proc. of the Euro-Par 1997*, number 1300 in Lecture Notes in Computer Science, pages 619–628. Springer-Verlag, 1997.

[74] M. Danelutto, S. Pelagatti, and M. Vanneschi. High level languages for easy massively parallel programming. Technical Report HPL–PSC–91–16, Hewlett Packard Laboratories, Pisa Science Center (Italy), 1991.

[75] M. Danelutto and M. Stigliani. SKElib: parallel programming with skeletons in C. In A. Bode, T. Ludwing, W. Karl, and R. Wismüller, editors, *Proc. of Euro-Par 2000*, number 1900 in Lecture Notes in Computer Science, pages 1175–1184. Springer-Verlag, September 2000.

[76] J. Darlington, A. J. Field, P.G. Harrison, P. H. J. Kelly, D. W. N. Sharp, R. L. While, and Q. Wu. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *Proc. of the Parallel Architectures and Langauges Europe (PARLE'93)*, number 694 in Lecture Notes in Computer Science. Springer-Verlag, June 1993.

[77] J. Darlington, Y. Guo, Y. Jing, and H. W. To. Skeletons for structured parallel composition. In *Proc. of the 15th Symposium on Principles and Practice of Parallel Programming*, 1995.

[78] P. J. Denning. Virtual memory. *ACM Computing Surveys*, 1996.

[79] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3), March 1966.

[80] E. W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5), May 1968.

[81] U. Drepper and I. Molnar. *The Native POSIX Thread Library for Linux*. Red Hat, Inc, January 2003.
Available at `http://people.redhat.com/drepper/nptl-design.pdf`.

[82] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. of the 13th International Symposium on Computer Architecture (ISCA'86)*, pages 434–442. SIGARCH, ACM, June 1986. Published as Proc. 13th Annual International Symposium on Computer Architecture, Computer Architecture News, volume 14, number 2.

[83] R. W. Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14(4):636–644, October 1967.

[84] S. Frank, H. Burkhardt, III, and J. Rothnie. The KSR1: Bridging the gap between shared memeory and MPPs. In *Proc. of International Computer Conference (COMPCON'93)*, pages 285–294, February 1993.

[85] M. Frigo. The weakest reasonable memory. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, U.S.A., 1998.

[86] M. Frigo and V. Luchangco. Computation-centric memory models. In *Proc. of the 10th Annual Symposium on Parallel Algorithms and Architectures (SPAA'98)*, pages 240–249. SIGARCH, ACM, June 28–July 2, 1998.

[87] F. Gallilée, J.-L. Roch, G. G. H. Cavalheiro, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 88–95, Paris, October 1998. IEEE Computer Society Press.

[88] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[89] D. Gannon and K. Keahey. PARDIS: a parallel approach to CORBA. In *Proc. of the 6th International Symposium on High Performance Distributed Computing (HPDC'97)*, pages 31–39. IEEE, 1997.

[90] K. Gharachorloo. *Memory consistency models for shared-memory multiprocessor*. PhD thesis, Computer Science Laboratory, Stanford University, December 1995.

[91] K. Gharachorloo, A. Gupta, and J. L. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proc. of the 4th symposium on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[92] A. Giancaspro, L. Candela, E. Lopinto, V. A. Lorè, and G. Milillo. SAR images co-registration parallel implementation. In *Proc. of the International Geoscience and Remote Sensing Symposium and the 24th Canadian Symposium on Remote Sensing (Igarss 2002)*. IEEE, June 2002.

[93] S. Gorlatch. Send-Recv considered harmful? Myths and truths about parallel programming. In *Proc. of PaCT 2001*, number 2127 in Lecture Notes in Computer Science, pages 243–257. Springer-Verlag, 2001.

[94] S. Gorlatch, C. Lengauer, and C. Wedler. Optimization rules for programming with collective operations. In *Proc. of the 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP'99)*, IEEE Computer Society Press, pages 492–499, 1999.

[95] S. Gorlatch and S. Pelagatti. A transformational framework for skeletal programs: Overview and case study. In J. Rohlim, editor, *Proc. of Parallel and Distributed Processing*, number 1586 in Lecture Notes in Computer Science, pages 123–137. Springer-Verlag, 1999.

[96] M. W. Goudreau, J. M. D. Hill, K. Lang, B. McColl, S. B. Rao, D. C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for the BSP worldwide standard library. Technical report, Oxford University Computing Laboratory, April 1996.

[97] E. Hagersten, A. Landin, and S. Haridi. DDM — A cache-only memory architecture. *Computer*, 25(9):44–54, September 1992.

[98] M. Hamdan, P. King, and G. Michaelson. A scheme for nesting algorithmic skeletons. In K. Hammond, T. Davie, and C. Clack, editors, *Proc. of the 10th International Workshop on the Implementation of Functional Languages (IFL'98)*, pages 195–211. Department of Computer Science, University College London, 1998.

[99] J. L. Hennessy and D. A. Patterson. *Computer organization & design. The hardware/software interface*. Morgan Kaufmann, 1994.

[100] J. L. Hennessy and D. A. Patterson. *Computer architecture. A quantitative approach*. Morgan Kaufmann, 2nd edition, 1996.

[101] C. A. Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursion*. PhD thesis, Fakultät für Mathematik und Informatik, Uni. Passau, Germany, 2001.

[102] High Performance Fortran Forum. *High Performance Fortran Language Specification (Version 2.0)*, January 1997.

[103] H. Hillis and G. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[104] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[105] C. A. R. Hoare. Communicating Sequential Processes. *Communications of ACM*, 21(8):666–677, August 1978.

[106] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[107] W. Hu, W. Shi, and Z. Tang. JIAJIA: An SVM system based on a new cache coherence protocol. In *Proc. of the High Performance Computing and Networking Europe (HPCN'99)*, pages 463–472, April 1999.

[108] D. C. Hyde. Java and different flavors of parallel programming models. In R. Buyya, editor, *High Performance Cluster Computing*, pages 274–290. Prentice Hall, 1999.

[109] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th Annual Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287. SIGARCH, ACM, June 1996.

[110] Intel Corporation. *Intel Architecture Software Delevoper's Manual. Volume 2: Instruction Set Reference*, 1999.

[111] JavaGrande. *The Java Grande home page*, 2002. (`http://www.javagrande.org`).

[112] C. F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Department of Electrical and Computer Science, Massachusetts Institute of Technology, U.S.A., January 1996.

[113] jPVM. *The jPVM home page*, 2001. (`http://www.chmsr.gatech.edu/jPVM/`).

[114] H. Kasahara, H. Honda, M. Iwata, and M. Hirota. A compilation scheme for macro-dataflow computations in hierarchical multiprocessor systems. In *Proc. of the 1990 International Conference on Parallel Processing*, pages II–294 – II–295, 1990.

[115] S. R. Kasaraju. Efficient tree pattern matching. In *Proc. of the 30th Annual Symposium on Foundations of Computer Science*, pages 178–183. IEEE, 1989.

[116] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. An evaluation of software-based release consistent protocols. *Journal of Parallel and Distributed Computing*, 29(2):126–141, September 1995.

[117] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In D. Abramson and J.-L. Gaudiot, editors, *Proc. of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, pages 13–21. SIGARCH, ACM, May 1992.

[118] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[119] C. W. Keßler. Pattern-driven automatic program transformation and parallelization. In *Proc. of the 3rd EUROMICRO Workshop on Parallel and Distributed Processing*. IEEE, January 1995.

[120] H. Kuchen. A skeleton library. In B. Monien and R. Feldmann, editors, *Proc. of Euro-Par 2002*, number 2400 in Lecture Notes in Computer Science, pages 620–629. Springer-Verlag, 2002.

[121] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford FLASH multiprocessor. In *Proc. of the 21st International Symposium on Computer Architecture (ISCA'94)*, pages 302–313. SIGARCH, ACM, April 1994. Published as Proc. of the 21st Symp. on Computer Architecture (21st ISCA'94), Computer Architecture News, volume 22, number 2.

[122] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, January 1979.

[123] J. Laudon and D. Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proc. of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, pages 241–251. SIGARCH, ACM, June 1997. Published as Proc. of the 24th Symp. on Computer Architecture (24th ISCA'97), Computer Architecture News, volume 25, number 2.

[124] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, March 1992.

[125] R. Lepère, G. Mounié, and D. Trystram. An approximation algorithm for scheduling trees of malleable tasks. European Journal of Operational Research. (to appear, `http://www-id.imag.fr/Laboratoire/Membres/Trystram_Denis/publis_malleable/`).

[126] R. Lepère, D. Trystram, and G. J. Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. *Lecture Notes in Computer Science*, 2161, 2001.

[127] X. Leroy. *The LinuxThreads library*. INRIA, Rocquencourt, France. (`http://pauillac.inria.fr/~xleroy/linuxthreads/`).

[128] K. Li. IVY: A shared virtual memory system for parallel computing. In *Proc. of the International Conference on Parallel Processing*, volume II, Software, pages 94–101, August 1988.

[129] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

[130] R. Low and C. Corley. High performance host microprocessor family, today and tomorrow. Smart Networks Developer Forum 2003, Paris, France, June 2003.
(`http://e-www.motorola.com/collateral/SNDF2003_EUROPE_H1101.pdf`).

[131] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1683, December 2002.

[132] S. MacDonald, D. Szafron, J. Schaeffer, and S. Bromling. Generating parallel program frameworks from parallel design patterns. In A. Bode, T. Ludwing, W. Karl, and R. Wismüller, editors, *proc of Euro-Par 2000*, number 1900 in Lecture Notes in Computer Science, pages 95–105. Springer-Verlag, September 2000.

[133] E. Mäkinen. On the subtree isomorphism problem for ordered trees. *Information Processing Letters*, 32:271–273, September 1989.

[134] MpiJava. *The MpiJava home page*, 2001.
(`http://www.npac.syr.edu/projects/pcrc/mpiJava/`).

[135] W. A. Najjar, E. A. Lee, and G. R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25:1907–1929, 1999.

[136] C. Nester, R. Philippsen, and B. Haumacher. A more efficient RMI for Java. In *Proc. of the Java Grande Conference*, pages 152–157. ACM, June 1999.

[137] D. L. Parnas. On the design and development of program families. *IEEE Trans. on Software Engineering*, SE-2(1):1–9, March 1976.

[138] P. J. Parsons and F. A. Rabhi. Generating parallel programs from paradigm based specifications. *Journal of Systems Architecture*, 45(4):261–283, 1998.

[139] S. Pelagatti. *A methodology for the development and the support of massively parallel programs.* PhD thesis, Computer Science Department, University of Pisa, Italy, 1993.

[140] S. Pelagatti. *Structured Development of Parallel Programs.* Taylor&Francis, 1998.

[141] T. Priol. A Grid programming model based on shared memory abstraction. Document distributed during the GF5 meeting, Boston, USA, October 2000.

[142] T. Priol. Programming the Grid with distributed objects. In *Proc. of Workshop on Performance Analysis and Distributed Computing (PACD 2002)*. Schloss Dagstuhl, Germany, 2002.
(found at `http://www.irisa.fr/orap/Forums/Forum12/Priol.pdf`).

[143] J. Protić, M. Tomašević, and V. Milutinović. Distributed shared memory: Concepts and systems. *IEEE parallel and distributed technology: systems and applications*, 4(2):63–79, Summer 1996.

[144] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, 2002.

[145] Z. Radović and E. Hagersten. Removing the overhead from software-based shared memory. In *Proc. of Supercomputing 2001*. ACM, November 2001.

[146] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical and Computer Science, Massachusetts Institute of Technology, U.S.A., 1998.

[147] T. Rauber and G. Rünger. A coordination language for mixed task and data parallel programs. In *Proc. of the 3rd Annual Symposium on Applied Computing (SAC'99)*, pages 146–155. ACM Press, 1999.

[148] M. Rinard. *The design, implementation and evaluation of Jade: a portable, implicitly parallel programming language*. PhD thesis, Stanford University, USA, September 1994.

[149] J. K. Salmon. *Parallel Hierarchical N-body Methods*. PhD thesis, California Institute of Technology, U.S.A., 1990.

[150] G. Sardisco and A. Machì. Development of parallel paradigms templates for semi-automatic digital film restoration algorithms. In G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, editors, *Parallel Computing: Advances and Current Issues. Proceedings of the International Conference ParCo2001*, pages 498–509. Imperial College Press, 2002.

[151] D. J. Scales and M.S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proc. of the 1st Symposium on Operating System Design and Implementation*, pages 101–114, November 1994.

[152] Semiconductor Industry Association. *International Technology Roadmap for Semiconductors: Overall Roadmap Technology Characteristics & Glossary*, 1999 edition. (Available at `http://www.semichips.org/`).

[153] J. Sérot and D. Ginhac. Skeletons for parallel image processing: an overview of the SKiPPER project. *Parallel Computing*, 28(12):1685–1708, December 2002.

[154] J. Sérot, D. Ginhac, R. Chapuis, and J. Derutin. Fast prototyping of parallel-vision applications using functional skeletons. *Machine Vision and Applications*, 12:217–290, 2001.

[155] L. M. Silva. Web-based parallel computing with Java. In R. Buyya, editor, *High Performance Cluster Computing*, pages 310–326. Prentice Hall, 1999.

[156] J. P. Singh, A. Gupta, and W. D. Weber. SPLASH: the Stanford ParalleL Applications for SHared memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44, January 1992.

[157] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, June 1995.

[158] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.

[159] M. Südholt. Data distribution algebras — a formal basis for programming using skeletons. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET'94)*, pages 19–38. Elsevier, 1994.

[160] Sun Microsystems. *The Java home page*, 2002. (`http://java.sun.com`).

[161] P. Teti. Lithium: a Java skeleton environment. Master's thesis, Computer Science Department, University of Pisa, Italy, October 2001. *in italian.*

[162] Thinking machines. *Getting started in CM Fortran*, November 1991.

[163] M. Tomašević, J. Protić, and V. Milutinović. A survey of distributed shared memory systems. In T. N. Mudge and B. D. Shriver, editors, *Proc. of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture*, pages 74–84. IEEE Computer Society Press, January 1995.

[164] Top500.org. *Top500 supercomputers sites*, 2002. (`http://www.top500.org`).

[165] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–11, August 1990.

[166] M. Vanneschi. Heterogeneous HPC environments. In D. Pritchard and J. Reeve, editors, *Proc. of Euro-Par 1998 (invited paper)*, number 1470 in Lecture Notes in Computer Science, pages 21–34. Springer-Verlag, September 1998.

[167] M. Vanneschi. PQE2000: HPC tools for industrial applications. *IEEE Concurrency. Parallel, distributed & mobile computing*, 6(4):68–73, October 1998.

[168] M. Vanneschi. Parallel paradigms for scientific computing. In *Proc. of the European School on Computational Chemistry (1999, Perugia, Italy)*, number 75 in Lecture Notes in Chemistry, pages 170–183. Springer-Verlag, 2000.

[169] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.

[170] M. Warren and J. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proc. of Supercomputing'93*. IEEE, 1993.

[171] N. Wirth. *Algorithms + data structures = programs*. Prentice-Hall, 1976.

[172] A. Zavanella. *Skeletons and BSP: Performance portability for parallel programming*. PhD thesis, Computer Science Department, University of Pisa, Italy, March 2000.