Parallel Processing Letters

eskimo: EXPERIMENTING WITH SKELETONS IN THE SHARED ADDRESS MODEL

MARCO ALDINUCCI
Inst. of Information Science and Technologies (ISTI) – National Research Council (CNR)
Via Moruzzi 1, I–56124 PISA, Italy
(aldinuc@di.unipi.it)

Received April 2003
Revised July 2003
Communicated by Gaétan Hains and Frédéric Loulergue

ABSTRACT

We discuss the lack of expressivity in some skeleton-based parallel programming frameworks. The problem is further exacerbated when approaching irregular problems and dealing with dynamic data structures. Shared memory programming has been argued to have substantial ease of programming advantages for this class of problems. We present the eskimo library which represents an attempt to merge the two programming models by introducing skeletons in a shared memory framework.

Keywords: Skeletons, dynamic data structures, software DSM, cluster of workstations.

1. Introduction

The development of efficient parallel programs is a quite hard task. Besides coding the algorithm, the programmer must also take care of the details involved in parallelism exploitation, i.e. concurrent activity set up, mapping and scheduling, synchronization handling and data allocation. In unstructured, low-level parallel programming approaches these activities are usually the full responsibility of the programmer and constitute a difficult and error prone programming effort. From Cole's seminal work [1] the skeleton research community has been active in experimenting with new technologies in order to simplify parallel programming by raising the level of abstraction. In the past decade we have designed and developed several skeleton-based parallel programming environments and have tested their effectiveness on a number of real world applications. While the skeletal approach has 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 this paper we present eskimo [Easy SKeleton Interface (Memory Oriented)], a new skeletal programming environment which represents a preliminary attempt to defeat the constraints of expressivity apparent in skeletal languages, especially when dealing with irregular problems and dynamic data structures. eskimo is based on the shared address programming model; its run-time is built upon a software DSM. In

the next section we present a brief (self-critical) history of the evolution of parallel programming frameworks. In Section *3* we present the eskimo design principles. In Section *4* we discuss the pay-back we expect from the skeletal approach. The paper is completed by some experimental results (Sec. *5*) and the related work (Sec. *6*).

2. Motivation and Historical Perspective

Historically a couple of works are due particular attention: the P^{3}L project [2] and the SCL co-ordination language [3]. They sought to integrate imperative code chunks within a structured parallel framework. As an example, the P^{3}L 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*.

Later on, all our experiences with P^{3}L have fed into the SkIE language and its compiler [4]. Existing sequential codes can be used in SkIE to instance skeletons with little or no amendment at all to the sources; it supports several guest sequential and parallel languages (C, C++, Fortran, Java, HPF) within the same application. A SkIE program is basically a composition of skeletons which are equipped with a compositional functional semantics. They behave like higher-order functions which can be evaluated efficiently in parallel. Furthermore, the skeletons' functional and parallel semantics enable the optimization of programs by means of performance-driven source-to-source code transformations [5,6,7].

During the SkIE project, several real world applications* have been used as a test-bed to validate the effectiveness of the programming environment [8]. A lack of expressivity emerged for some of them. 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, while others need a greater design effort. The boundary between the two classes depends on many factors, among the others the particular programming environment and the skeleton set chosen for applications development. Some common flaws emerge in both SkIE and other research group works (see also [9]):

(i) The selection of skeletons available in the language skeleton set is a quite critical design issue. Despite several endeavors to classify and close the parallel programming skeleton set [10], in many cases during application development we experienced a need for the "missing skeleton", or at least missing functionality of 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 or ad-hoc interaction primitives.

(iii) Although all kind of languages may be equipped with a skeletal super-structure, skeletal languages have been historically designed in a functional programming style fashion [4,11]. In this setting the non-functional code is embodied into the skeletal framework by providing the language with wrappers acting as pure functions. Actually, the fully functional view (by its very nature) does not enhance programmer control over data storage. This feature may happen to

*In the area of massive data mining, computational chemistry, remote sensing, image analysis, etc.

be useful in the design of applications managing large, distributed, randomly accessed data sets.

Indeed, the role of skeletons in the programming language has evolved and matured over the past decade. Such evolution has been designed to overcome (among other things) skeletal languages lack of expressiveness while preserving their ease of use. In particular skeletons lose 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 with skeletons. Rather, it can be viewed as "recipe" that can be used to achieve different solutions to common programming problems. Parallel skeleton support may be implemented using a layered, OO design [12]. Parallel skeletons can be declared as members of proper skeletons/patterns. Exploiting standard OO visibility mechanisms, part of the framework may be made visible to the programmer to undertake different tasks: fine performance tuning, introduction of new, more efficient implementation schemes, etc. [8,13].

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 in this category, such as: `SKElib` that extends C language with SkIE-like skeletons and enables the programmer to use standard Unix communication mechanisms [14]. `Skil` extends C++, providing the programmer with an SPMD environment with task (pipeline and farm) and data parallel (map, fold, ...) skeletons, that are seen as collective operations [15]. `Lithium` is the first *pure Java* structured parallel programming environment based on skeletons and exploiting macro data flow implementation techniques [16]. `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 extends the standard C binding to MPI with skeletal operations [9].

eskimo has been influenced by both previous approaches. eskimo extends C with "proto-skeletons" or constructs, which represent skeletons' building blocks. Skeletons do not really exist in eskimo programs as language elements, they are instead particular programming idioms.

3. eskimo: A New Skeletal Language

eskimo is a parallel extension of C based on the shared address programming model. The target architectures for the language are Beowulf class machines, i.e. POSIX boxes equipped with TCP/IP networks. In this setting eskimo is conceived to be a framework in which to experiment with the feasibility of the skeletal approach with dynamic data structures in parallel programming.

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

software distributed shared memory. Notably it is not just another DSM, instead it relies on existing DSM technologies to experiment with the co-design of dynamic data structures and parallel programming patterns, enforcing locality in the access to distributed memory. We outline the main features of eskimo as follows:

Abstraction. eskimo is a skeleton based programming language. It aims to simplify programming by raising the level of abstraction, providing the programmer with performance and portability for applications. In order to convey this simplicity to programmers we must be careful not to bundle it with excessive conceptual baggage. To this end we enriched C in such a way that the language extension raises the level of abstraction. The main sources of abstraction concern *data structures, the flow of control, and the interaction between them.* All abstractions rely on solid concepts like concurrency and abstract data types.

Expressiveness. We propose a structured programming environment that allows the programmer to deal with (dynamic) spread shared data structures. In particular, the programmer deals with an abstraction of data structures represented as a single entity (as in [17]). These parts are kept consistent by the run-time support following a (very) lazy memory consistency model (see Sec. *3.3*). The chosen consistency model enforces the high-level approach of the language since it allows reading/writing of data objects without the need for 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. a high-order function), it is rather a code pattern built directly by the programmer using language primitives. Ad-hoc parallel pattern may be coded using both eskimo and other libraries primitives.

Framework and design principles. eskimo's main target architecture class are Beowulf 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 the target architecture class, eskimo exposes to the programmer a (virtual) shared NUMA address space. The programmer is required to make decisions about the relationship among data structures (e.g. locality) but not to deal with all the cumbersome facets of data mapping. The underlying design principle favors a programming environment in which performance improves gradually with increased programming effort (taking advantage of a deep application knowledge) with respect to one capable of ultimately delivering better performances but that requires an inordinate programming effort. This can consist of either programming each detail of the application (as in low-level approaches) or expressing the application by using a fixed set of ready-made parallel paradigms.

In summary eskimo extends the C language with three classes of primitives: flows of control management (Sec. *3.1/3.2*); *Shared Data Types* declaration, allocation and management (Sec. *3.3*).

3.1. Exploiting Parallelism in eskimo

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. In any part of the program, the programmer may split the flow of control through the asynchronous call of a number of functions;

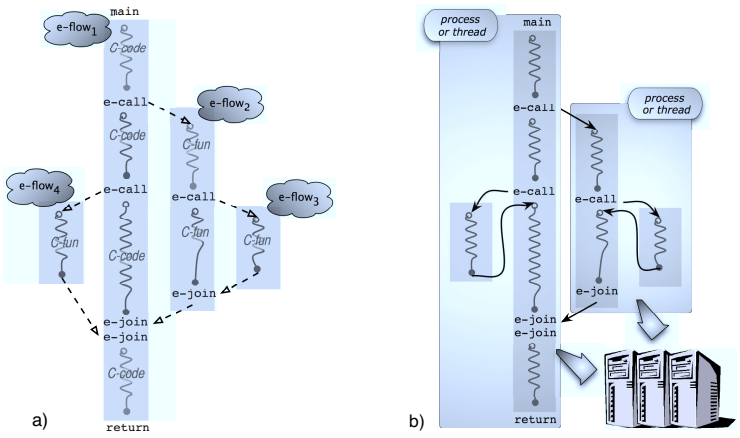


Fig. 1. An eskimo program execution intuitive view. a) Relationship among *e-calls, e-joins* and *e-flows* (grey boxes). b) A possible execution of the program.

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; we call them *e-call/e-join*. Also, we call *e-flows* eskimo flows of control. *e-flows* share the virtual memory address space. The relationship between *e-calls, e-joins* and *e-flows* is discussed in Section *3.2* and intuitively sketched in Fig. 1 a).

e-call/e-join primitives enable the programmer to set up a dynamic and variable number of *e-flows*, an important feature when dealing with dynamic data structures (in particular linked data structures as lists and trees). Almost all interesting algorithms on these data structures explore them in a recursive fashion, following the Divide&Conquer ($\mathcal{D}$&$\mathcal{C}$) paradigm and exploiting a variable concurrency availability during the program run.

Actually $\mathcal{D}$&$\mathcal{C}$ has been already present in classical skeleton sets. This skeleton, as others (e.g. scan and map), has often been interpreted as a collective operation on a given data structure by the skeleton community [6,19]. In the best case this inspired a family of variants for each skeleton, each of them optimizing a particular behavior of the algorithm on the data structure [20]. In other cases, application programmers have been compelled to match a given shape of the data structure with a given behavior of the skeleton (which 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 classical sequential programming. There is no ready-made $\mathcal{D}$&$\mathcal{C}$ skeleton in eskimo, there are rather all the ingredients to build it in such a way that 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. Even if the hints are wrong or missing the program does not lose its correctness (even if we cannot expect an optimal performance from it).

3.2. Concurrency and Flows of Control

Actually *e-flows* do not necessarily match any concrete entity at the eskimo run-time support level or its underlying run-time layers (for example threads or processes). In particular an *e-call* should be considered as the declaration of a "concurrency capability" with respect to a given function instance: an *e-called* function instance might be both concurrently executed or sequentialized with respect to the caller function. In the former case, the matching *e-join* represents the 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 algorithm points where it is possible to proceed in more than one way, even for the same input data set, they are *non-deterministic* primitives. As an example, the eskimo program sketched in Fig. 1 a) exploits four *e-flows* that represent its top concurrency degree. However, as shown in Fig. 1 b) the language run-time might choose to halve the concurrency degree and project the four *e-flows* into two real concurrent entities. Serialized *e-flows* do not lead to any overhead related to parallelism exploitation (thread creation, activation, synchronization).

As might be expected, *e-flows* have to be mapped to 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 the eskimo run-time; corresponding to 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* really be concurrently executed (either in parallel or interleaved) or must it be serialized with respect to the calling one?
2 In the 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 contrasting needs. In the former case, it tries to keep 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. In the latter case, the run-time tries to balance workload across PEs while maintaining data locality as much as possible.

We 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, but in some cases are not. Consider some *e-flows* called in a sequence that establish a (direct or indirect) precedence relation of one over another; or some *e-flows* sequentialized within the same concurrent entity. In both cases *e-flows* do not correspond to any actual concurrent code.

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 Fig. 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-*

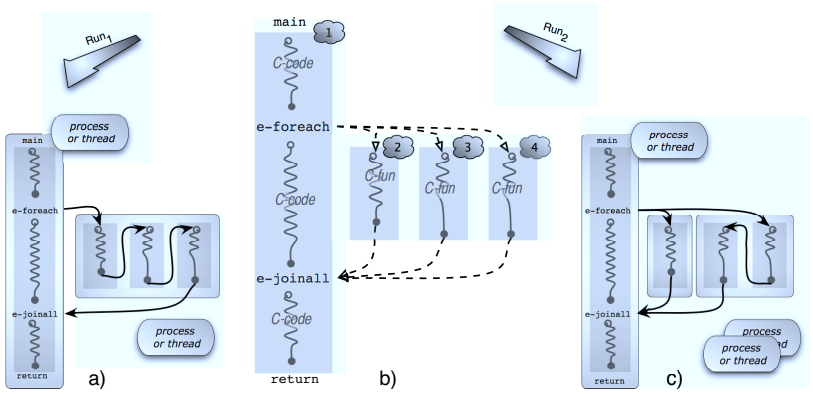


Fig. 2. An eskimo program resulting in different mapping and scheduling in different runs.

joinall have an additional degree of freedom with respect to a sequence of *e-call/e-join*: the order in which *e-flows* are mapped/scheduled and joined. This produces some additional advantages with respect to the simple 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. The eskimo run-time uses this 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, Fig. 2 a) and c) sketch two runs of the same program that have been subjected to different mapping and scheduling decisions.

3.3. Sharing Memory among Flows of Control

The eskimo language is specifically designed for distributed memory architectures. These architectures naturally supply a very efficient memory access to local memory and a more expensive access to remote memory. The language abstracts these memory categories, but it does not obscure the differences between them. eskimo language exposes to the programmer two memory spaces: *private* and *shared*.

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 arrays k-trees and regions (containers for any C type). 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 exceptions) forbidden in eskimo programs. These must be replaced 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. Therefore shared variables might be subjected to concurrent/parallel accesses. These accesses are regulated by the eskimo run-time.

Overall, SADTs and their constructors provide a device to make any C data type sharable. These are data structure containers allowing access to contained data (i.e. private or shared variables) by several *e-flows*. In case of trees and arrays the container is structured and holds many data items; such data are spread across the virtual architecture and their total size may exceed the memory size 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 provide globally addressable data structures matching the same relation between C pointers and variables.

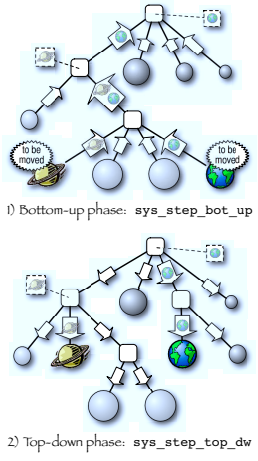
All SDTs are designed to be concurrently accessed by *e-flows*. SDTs may be statically or dynamically (and incrementally) allocated, in particular $k-$tree nodes must be dynamically allocated by means of a language primitive (`e_node_add`). Shared data items are transparently blocked into segments at allocation time; a segment's size may be configured to match a suitable working size for the PEs/network power tradeoff. Beyond trees, the programmer may build any shared linked data structure by using references. Shared variables obey DAG consistency [18] and can be accessed through eskimo primitives. This basically means that 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 that different *e-flows* must write different shared address locations.

Most software DSMs rely on MMU and operating system traps in order to make the access of remote data transparent. However the interrupt cost, associated with receiving a message, has been proved to be the largest component of remote latency, not the actual wire delay in the network or the software implementing the protocol [21]. We followed a different approach: shared variables, in contrast to private ones, cannot be read and written directly. In order to access a shared variable the programmer must explicitly bind it to a private pointer (see Fig. 3 line 4). This technique avoids the use of operating system traps. In addition it has a pragmatic importance: since accessing a shared variable may be expensive the programmer is required to evaluate really needed accesses[†] eskimo provides a pair of language primitives enabling access to a shared variable: `r` (for read-only) and `rw` (for read/write). Both `r` and `rw` get a reference and return a void pointer that can be used (after a suitable type cast) to access the shared variable value.

4. 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:

[†] Anyway, transparency may be achieved by preprocessing/instrumenting the user code.




```
1  eref_t sys_step_bot_up(eref_t anode) {
2    eref_t ret_array[4]; e_handler_t e_handler;
3    eref_t float_list, sink_list; node_t *np;
4    np = (node_t *) rw(anode); /* bind np to anode */
5    if (np->leaf) {     /* recursion base case */
6      <Figure out acceleration (visit the tree from root)>
7      <Update bodies position (np->x = ...; np->y = ...;)>
8      if (!within_borders(elem)) push(float_list,anode);}
9    else { /* Divide */
10     e_foreach_child(e_handler,sys_step_bot_up,np);
11     e_joinall(e_handler,ret_array);
12     /* Conquer */
13     for (i=0;i<4;i++)
14       while (elem=pop(ret_array[i]))
15         if (within_borders(elem))  /* read *np */
16           push(sink_list,elem);
17         else
18           push(float_list,elem);
19     <handle chain elimination and other particular cases>}
20   return (float_list); }
```

Fig. 3. An n-body system step in two phases and the eskimo pseudocode of the first phase.

`e_foreach_child` runs through non-null children of a spread tree node;
`e_foreach_cell` runs through cells of a spread array;
`e_foreach_refset` runs 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 the 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, the $\mathcal{D}\&\mathcal{C}$ paradigm may be interpreted as a special (dynamic, recursive) case of data parallelism. As an example consider the Barnes-Hut application code fragment shown in Fig. 3.

eskimo allows the programmer 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 $\mathcal{D}\&\mathcal{C}$. The two paradigms may be freely interleaved and are both introduced at the language level by just one primitive (family), namely the *e-foreach*. If the application is not obviously expressible as instances of the 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 that they are *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 a task, the run-time takes account of 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 a Beowulf class cluster exploits an unbalanced communication/computation power tradeoff, the run-time tries to schedule an *e-flow* on the same PE as the

data it needs and takes in account load-balancing only as secondary constraint. Clearly, the key issue here is to know in advance which data a function will access. The run-time uses two kind of information: *(i) The data blocking.* The run-time makes scheduling decisions only when a function parameter crosses the current data segment boundary. As a result *e-flows* accessing the data in segment are sequentialized, enhancing locality and coarsening computation grain. *(ii) 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 to be allocated "close" to data referenced by it. A wrong/missing hint has no impact on program correctness but a great impact on performance.

5. Implementation and Experiments

A prototype implementation of eskimo already runs on Linux clusters. It has a layered implementation design: each layer provides mechanisms and policies to solve some of parallel programming support issues. In particular mapping and scheduling of *e-flows*, as well as shared data structure mapping are implemented within the top tier. At this level a lot of information about the system status may be accessed (e.g. PEs load, memory load). These data are maintained by a lower level tier that basically wraps the communication stack (currently the TCP stack). System status information is exchanged among PEs with no additional communications with respect to those required by the algorithm synchronizations. The run-time exploits both multiprocessing (inter-PEs) and multithreading (intra-PEs).

We tested eskimo on a Barnes-Hut application. It presents non-trivial performance challenges due to the irregular and dynamically changing nature of data structures. We developed three different versions of the Barnes-Hut algorithm: 1) The sequential version (seq), implemented in C. It is a reduction to the bare bones of the original Barnes-Hut code in the bidimensional space. 2) The eskimo version. It is obtained from the sequential version by substituting recursive calls with recursive *e-foreach*es. 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, then they are pulled down to target quadrant (see Fig. 3 left). The main data structure is a spread quad-tree. 3) 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 in order to speed up the accesses to frequently accessed items (i.e. an ad-hoc cache). Each PE keeps the replicated part of the forest coherent by exchanging messages with other PEs.

We tested the three application versions on two different datasets: *plummer* and *uniform*. The two distributions may be hierarchically represented by strongly-unbalanced and fairly-balanced trees respectively. Table 1 shows the performance and speedup figures of the three versions of the algorithm on the SMP cluster for four different datasets. The eskimo version of the code was as fast as the MPI version for the uniform dataset, and slightly but significantly better for the plummer dataset (highlighted in the table). The point here is that the performance of the MPI version does not scale up with the number of PEs for the plummer dataset. The unbalanced tree leads to a heavy load imbalance in the MPI version. Our

dataset·#bodies	Pl·10k	Pl·20k	Un·10k	Un·20k	Pl·10k	Pl·20k	Un·10k	Un·20k	Optimal
	time				speedup				
seq	6.47	14.53	4.80	2.34	–	–	–	–	–
MPI 1x2-way SMP	6.60	14.36	2.54	1.27	0.9	1.0	1.9	1.8	2
MPI 2x2-way SMP	6.64	14.58	1.50	0.75	0.9	1.0	3.2	3.1	4
eskimo 1x2-way SMP	5.30	13.20	2.45	1.33	1.2	1.1	1.9	1.8	2
eskimo 2x2-way SMP	4.10	8.10	1.55	0.77	1.6	1.8	3.1	3.0	4

Table 1. Barnes-Hut performances (secs) and speedups on several Plummer (Pl) and uniform (Un) datasets on a SMP cluster (2-way 550MHz Pentium III).

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 to 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. In contrast, eskimo code can be written without any concern for load balancing and data mapping. As a matter of fact the sequential version is just 300 lines of code, the eskimo version 500 and the MPI version 850.

6. Conclusions, Related Work and Acknowledgments

The eskimo programming framework was 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* [22] and *Athapascan* [23]. In particular, eskimo inherits its memory consistency model from *Cilk*. The layered implementation design and the idea to make customizable the scheduling are quite similar to *Athapascan*. Nevertheless, eskimo differs from them in a number of design issues. It does not use work stealing (which has load-balancing as main target); it tries to exploit the combined mobility of data and computations with the aim of reducing network traffic by using programmer insight on frequent data access patterns. It implements dynamically allocatable dynamic data structures (trees). These are designed to be blocked in order to reach an acceptable working grain for the target architecture class in a transparent way. It is conceived for loosely coupled architectures. It does not rely on any shared stack for function calls: the stack never moves across PEs and may be optimized by the standard C compiler (e.g. in-lining). Last but not least, it abstracts flow of control and data management in a skeletal perspective.

eskimo exhibits good performance on a non-trivial problem using dynamic data structures. However, this is preliminary work and a number of features remains to be fixed. We believe that the adoption of an OO (as in [17,16]) host language should greatly simplify the SADT definition and implementation, the language syntax, and eventually enforce program correctness by suitable type checking. Also, we ought rethink eskimo in terms of mainstream memory models. The adoption of an object-based memory is likely to preserve the eskimo "memory centric" viewpoint, thus preserving the ability to enable an easy design of recursive algorithms without bundling in many cumbersome synchronization details.

Acknowledgements. *I wish to thank M. Danelutto and M. Vanneschi for many fruitful discussions. This research has been partially supported by the Italian Space Agency under ASI-PQE2000 Project, by CNR under Agenzia2000 Project, by Italian MIUR under Strategic Project*

"legge 449/97" *year 2000 No. 02.00640.ST97 and FIRB Project* "GRID.it" *No. RBNE01KNFP.*

References

[1] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[2] 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), May 1995.

[3] J. Darlington, Y. Guo, Y. Jing, and H. W. To. Skeletons for structured parallel composition. In *Proc. of 15th PPoPP*, 1995.

[4] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25(13–14), Dec. 1999.

[5] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimization. In *Proc. of 11th PDCS*. IASTED/ACTA press, Nov. 1999.

[6] 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), 2001.

[7] M. Aldinucci. Automatic program transformation: The Meta tool for skeleton-based languages. In *Constructive Methods for Parallel Programming*, Advances in Computation: Theory and Practice, chapter 5. Nova Science Publishers, 2002.

[8] G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, ed. *Parallel Computing: Advances and Current Issues. Proc. of ParCo2001*. Imperial College Press, 2002.

[9] M. Cole. Bringing skeletons out of the closet. Technical report, Uni. Edinburgh, 2002.

[10] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor&Francis, 1998.

[11] J. Sérot and D. Ginhac. Skeletons for parallel image processing: an overview of the SKiPPER project. *Parallel Computing*, 28(12), Dec. 2002.

[12] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12), Dec. 2002.

[13] F. A. Rabhi and S. Gorlatch, ed. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, 2002.

[14] M. Danelutto and M. Stigliani. SKElib: parallel programming with skeletons in C. In *Proc. of Euro-Par 2000*, LNCS n. 1900, Sep. 2000.

[15] G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proc. of 5th HPDC*. IEEE, 1996.

[16] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Fut. Generation Computer Systems*, 19(5), 2003.

[17] H. Kuchen. A skeleton library. In *Proc. of Euro-Par 2002*, LNCS n. 2400, 2002.

[18] 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 8th SPAA*, 1996.

[19] S. Gorlatch. Send-Recv considered harmful? Myths and truths about parallel programming. In *Proc. of PACT 2001*, LNCS n. 2127, 2001.

[20] C. A. Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursion*. PhD thesis, FMI Uni. Passau, Germany, 2001.

[21] Z. Radović and E. Hagersten. Removing the overhead from software-based shared memory. In *Proc. of Supercomputing 2001*. ACM, Nov. 2001.

[22] 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 *Proc. of 5th PPoPP*, 1995.

[23] 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 *Proc. of PACT '98*. IEEE, 1998.