Skeletons for multi/many-core systems

Marco ALDINUCCI [a] Marco DANELUTTO [b] Peter KILPATRICK [c]
[a] *Dept. Computer Science – Univ. of Torino – Italy*
[b] *Dept. Computer Science – Univ. of Pisa – Italy*
[c] *Dept. Computer Science – Queen's Univ. Belfast – UK*

Abstract. We discuss how algorithmic skeletons (and structured parallel programming models in general) can be used to efficiently and seamlessly program multi-core as well as many-core systems. We introduce a new version of the `muskel` skeleton library that can be used to target multi/many-core systems and we present experimental results that demonstrate the feasibility of the approach. The experimental results presented also give an idea of the computational grains that can be exploited on current, state-of-the-art multi-core systems.

Keywords. algorithmic skeletons, multi-core, many-core, multi-threading

1. Introduction

It is evident that multi/many-core systems (M^2C, from now on) are going to replace single core systems in the immediate future. At the moment, dual/quad core chips already provide the same aggregated performance as single core chips at a fraction of the frequency (and therefore of power) [10]. At the same time, many-core chips have been demonstrated that reach performances in the Teraflop range with power consumption which is orders of magnitude smaller than that of clusters with comparable processing power [11]. Unfortunately, M^2C systems do not implement the same architectural model as single core ones: single core systems basically present the user/compiler with a Von Neumann architecture. M^2C systems present instead a (possibly, but not necessarily, shared memory) multiprocessor architecture. Therefore *parallel programs* are needed to exploit their power. It is not possible to take a current, sequential program and run it twice as fast on a dual core without modifying it, unless it is already a multi-threaded. Even if the code has already been implemented as multi-threaded, it is not clear whether it could be run on high end multi-core chips, those sporting tens to hundreds of cores, with decent performance and scalability.

The main issue related to M^2C system exploitation consists in being able to feed these processing elements with a large number of independent threads [9]. A fundamental principle underpinning pursuit of high performance from parallel systems is to always have a thread ready to be scheduled any time one of the independent cores is available to execute a new task. However, this is not the only issue. In the case of the M^2C systems

[0]This work has been partially supported by EU FP6 NoE CoreGRID, EU FP6 STREP GridCOMP and Italian FIRB Insyeme projects.

with a traditional, hierarchical memory subsystem–e.g. those systems where each core accesses memory through one or more levels of local caches–the threads should be such that the memory hierarchy works effectively. The number and the kind of accesses of the different threads should not impair locality in the accesses to the memory subsystem. When the number of threads is large, this might not be so easy. If the M^2C cores do not have access to a traditional cache/memory system–for example, as in the Cell synergic processing elements–or the interconnection network used on the chip is not uniform–e.g. it is a mesh, as in the 80 cores Intel experimental chip–more problems arise, leading either to the necessity of being able to identify the memory block transfers needed to execute a thread or to the necessity to move from threads to processes and, moreover, to map the resulting (possibly internally multi-threaded) processes in such a way that locality at the process level is preserved at the interconnection level.

Algorithmic skeletons have been presented for many years as an effective means of supporting parallel computing, in contrast with the traditional one where programmers intervene at the source code level to handle all the details related to parallelism exploitation. Many skeleton programming environments have been developed, either as libraries with bindings in well-known sequential languages ([12,5,13] in C/C++, [7,1,8,4] in Java, to name but a few) or as new languages, perhaps providing the possibility to reuse existing sequential portions of code [3,16].

Recently, skeleton system designers have developed versions of their systems effectively targeting multi-threaded systems. Calcium [4] allows the programmer/user to choose one of three different "execution environments" via a very simple library call. These execution environments basically represent optimized run times relative to different types of target architectures. One of them targets exactly those architectures capable of running a number of independent threads, such as symmetric multi-processors (SMPs) or M^2C. However, there is as yet no possibility of having different run time systems (execution environments) coexisting and cooperating during the same program execution to target, for example, a network of SMP/M^2C processing elements. Muesli [15] exploits both MPI and openMP to achieve efficient implementation of skeletons on clusters of multi-core processing elements. However, the openMP pragmas only affect data parallel skeletons (distributed arrays and map/reduce/gather operations), as evinced from the source code (see code at [15]). Stream parallel skeleton implementation–e.g. the implementation of pipelines and farms–appears not to take any advantage of the multi-core potentialities.

Here we propose an evolution of the `muskel` skeleton programming environment that seamlessly supports networked M^2C and single core processing elements (Sec. 2). The new `muskel` version ($\texttt{nmc}_{\texttt{muskel}}$, **n**etworked **m**ulti-**c**ore `muskel`) will be used to assess the feasibility of using skeletons as a programming model for M^2C and networked M^2C systems (Sec. 3). It will also be used to assess (once again) the principle that structuring parallel activities at a high level of abstraction removes much of the burden from the programmer by allowing the compiler and run time system of the skeleton framework to implement very effective policies and strategies.

2. Skeletons going multi/many-core

`muskel` is a full Java skeleton library implementing skeletons according to the macro data flow model [6]. It currently provides a subset of stream parallel skeletons (pipeline

```
public static void main(String [] args) {
  // declare the skeleton program
  Compute stage1 = new F(...);
  Compute worker = new W(...);
  Compute stage2 = new Farm(W);
  Compute stage3 = new G(...);
  Compute main = new Pipe(stage1,
       new Pipe(stage2,stage3));

  // declare the manager, with input and output stream
  // performance contract and program to compute
  Manager mgr = new Manager(
   new InputStreamManager("input.dat"),
   new ConsoleOutputStreamManager(),
   new ParDegreeContract(Integer.parseInt(args[0]))
   main);

  // start computation of program, in parallel
  // according to the contract
  mgr.eval();
}
```

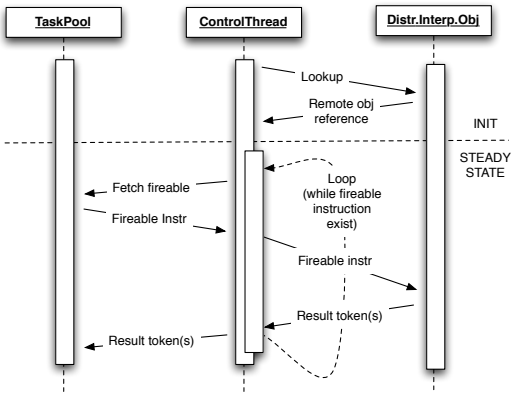

Figure 1. Sample `muskel` code (*left*). Structure of a `muskel` `ControlThread` (*right*).

and task farm) and a simple data parallel skeleton (map). Furthermore, it allows users to extend the skeleton set by defining new entries in the skeleton library and providing a macro data flow implementation of these new skeleton entries. Originally, `muskel` targeted clusters and networks of workstations, provided the processing elements support Java and RMI accesses.

A `muskel` program has the form depicted in Fig. 1, left (this is a three stage pipeline, with parallel second stage, computing $G(W(F(x_i)))$ for each input task x_i). First, the code to be executed is defined as a composition of available skeletons and sequential portions of code subclassing the `Compute` class. This class assumes only the existence of a method sequentially computing the result relative to the input data passed as parameter(s), without any side effect. All the details relative to the parallel execution of the program defined by the user are hidden in the `muskel` application `Manager`. The user instantiates a `Manager` and then uses the manager's `eval()` method to execute the skeleton program. When the `eval()` method terminates, the parallel execution of the skeleton program is terminated and its results can be processed. The parallelism degree for the execution of the *whole* program is passed to the `Manager` (in this case taking it from the command line parameters). An input and an output stream manager should be given to the `Manager` to handle skeleton program I/O. The `InputStreamManager` is basically a Java `Iterator` providing input tasks, while the `OutputStreamManager` provides a `deliver(Object res)` method handling (post-processing, storing, etc.) a single item of the result stream. Programmers should instantiate a manager and provide a program, an input and an output stream and then request computation of the program. When the `eval()` method of the `Manager` is called

- The program is compiled into a macro data flow graph (a data flow graph where instructions compute sequential, side effect free portions of code modelled by `Compute` subclasses).
- Then each item on the input stream is used to instantiate an input token of a new graph instance in the `muskel` `TaskPool`. The `TaskPool` is the repository of macro data flow instructions processed by the `muskel` distributed interpreter. It is a logically centralized data structure. Its implementation may obviously be

implemented in a distributed way to avoid bottlenecks, but in the current version of `muskel` it happens to be centralized.

- A set of `ControlThread`s (set up by the `Manager`, according to the user supplied `PerformanceContract`[1], each managing a different remote processing element hosting a distributed macro data flow interpreter instance) fetches fireable macro data flow instructions from the instruction `TaskPool` and dispatches them for execution on remote interpreters, ensuring load balancing and fairness in remote interpreter usage. The results of the remote executions are dispatched as new tokens to the instructions in the `TaskPool` (if intermediate) or to the output stream (in the case of final results). The operation of a `ControlThread` is shown in Fig. 1 right.

The computation of a `muskel` program therefore consists in the execution of the `Manager` (with the associated `ControlThread`s) on the local user machine and on the execution of distributed macro data flow interpreter instances (`RemoteInterpreter` objects) on remote resources. The exact number of remote interpreters required by the `Manager` according to the user supplied `PerformanceContract` are dynamically recruited through a multicast based discovery protocol run by the `Manager` and by the `RemoteInterpreter`s. Interactions between the `Manager` and the remote interpreter instances use plain Java RMI. In the initial phase of a `muskel` program execution the `Compute` subclasses used to execute the different macro data flow instructions are staged to the remote computing elements before program execution actually starts. Then, each time an instruction has to be executed, only the input tokens are serialized to the remote node for execution of the macro data flow instruction. Fig. 2 shows the Orc [14] model of the `muskel` skeleton interpreter, as introduced in [2][2].

Moving to $\mathsf{M}^2\mathsf{C}$ we decided to perform two different sets of experiments with `muskel`: i) modify the `muskel Manager` in such a way that a single $\mathsf{M}^2\mathsf{C}$ resource is efficiently targeted, and ii) modify the `muskel Manager` and the `RemoteInterpreter` in such a way that a cluster of $\mathsf{M}^2\mathsf{C}$ resources can be efficiently targeted. The resulting prototype will eventually turn out to be $\mathtt{nmc}_{\mathtt{muskel}}$. Both the modifications required a minimal set of changes in the current version of the `muskel` interpreter:

i) in order to have a `Manager` targeting a single $\mathsf{M}^2\mathsf{C}$ resource, we modified the way `ControlThread`s are paired with the remote resources. Instead of passing the `ControlThread` a resource name obtained from the discovery service, we simply passed the `ControlThread` a new instance of the RMI `Remote` object run by the `RemoteInterpreter` in the original version of `muskel`. In terms of the Orc model of Fig. 2, this means the $discovery(G, pgm, t)$ definition is substituted by

$$discovery(G, pgm, t) \triangleq (|_{i=1,n_{core}} let(worker.new())) > w > rmworkerpool.add(w)$$

where $worker.new()$ is the instantiation of a new local object of the same class as the one instantiated by the remote interpreter on remote resources in the original `muskel`. The rest of the interpreter is left unmodified.

[1] `ParDegree` is a subclass of `PerformanceContract`

[2] The reader not familiar with Orc may refer to the Orc web site `http://orc.csres.utexas.edu/index.shtml` for tutorial and documentation

$$
\begin{array}{l}
system(pgm, tasks, contract, G, t) \triangleq \\
\quad taskpool.add(tasks) \mid discovery(G, pgm, t) \mid manager(pgm, contract, t) \\
\\
discovery(G, pgm, t) \triangleq \\
\quad (|_{g \in G} \ (\ \text{if } remw \neq \text{false} \gg rworkerpool.add(remw) \\
\qquad \text{where } remw \in (\ g.can\_execute(pgm) \mid Rtimer(t) \gg let(false)\ )\ )\ ) \\
\qquad\qquad \gg discovery(G, pgm, t) \\
\\
manager(pgm, contract, t) \triangleq \\
\quad |_{i\ 1 \leq i \leq contract} (rworkerpool.get > remw > ctrlthread_i(pgm, remw, t)) \\
\quad \mid monitor \\
\\
ctrlthread_i(pgm, remw, t) \triangleq taskpool.get > tk > \\
\quad (\ \ \text{if } valid \gg resultpool.add(r) \gg ctrlthread_i(pgm, remw, t) \\
\quad\ \mid \text{if } \neg valid \gg (\ taskpool.add(tk) \\
\qquad\qquad\qquad \mid alarm.put(i) \gg c_i.get > w > ctrlthread_i(pgm, w, t)\ )\ ) \\
\qquad \text{where } (valid, r) \in \\
\qquad\quad (\ \ remw(pgm, tk) > r > let(true, r) \ \mid\ Rtimer(t) \gg let(false, 0)\ \ ) \\
\\
monitor \triangleq alarm.get > i > rworkerpool.get > remw > c_i.put(remw) \\
\qquad\qquad \gg monitor
\end{array}
$$

Figure 2. `muskel` interpreter modelled: Orc modelling/specification

ii) in order to target M^{2}C remote resources, instead, we changed the discovery service in such a way that remote resources publish their number of cores in response to the discovery protocol messages. Then, the `Manager` is free to recruit multiple instances of `RemoteInterpreter`s on the remote resources sporting multiple cores. The original multithreaded implementation of RMI servers in Java guarantees that multiple execution requests of macro data flow instructions directed to the same remote object are executed concurrently. Referring to the Orc model of Fig. 2, this means the portion

$$
\begin{array}{l}
|_{g \in G} \ (\ \text{if } remw \neq \text{false} \gg rworkerpool.add(remw) \\
\qquad \text{where } remw \in (g.can\_execute(pgm) | Rtimer(t) \gg let(false)\ )
\end{array}
$$

is changed to

$$
\begin{array}{l}
(|_{g \in G} \ (\ \text{if } remw \neq \text{false} \gg (|_{i \in 1, \#ncore} \ rworkerpool.add(remw)\ ) \\
\quad \text{where } (remw, \#ncore) \in \\
\qquad (g.can\_execute(pgm) | Rtimer(t) \gg let(false)\ )\ )\ )
\end{array}
$$

The result is that for each remote resource declaring `#ncores` available, up to *#ncore* `ControlThread`s will eventually be forked, if needed according to the `PerfConctract`. The rest of the interpreter is left unmodified.

We modified the `muskel` prototype in accordance with i) and ii) above to get the first version of the nmc$_{\texttt{muskel}}$ interpreter. Using nmc$_{\texttt{muskel}}$, we are able to demonstrate that heterogeneous workstation networks, hosting different CPUs, single or multi-core, can be efficiently exploited using skeletons. In the general case, and provided the average computational grain of the macro data flow instructions executed after compiling the skeleton application is not too fine, a network with k processing elements and a total number of M cores can achieve a speedup proportional to M ($speedup = \alpha \times M$) rather than k (values of α depend on the processing power of the single cores involved).

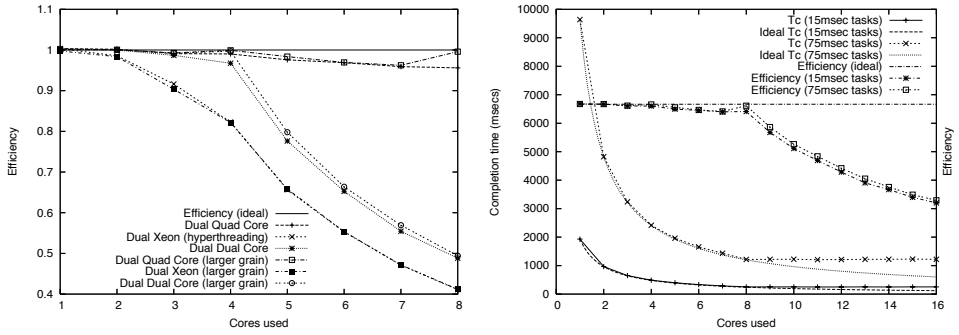

Figure 3. `nmc`$_{\texttt{muskel}}$: synthetic (floating point) benchmark on different SMP configurations (left) and effect of computational grain on completion time and efficiency of a synthetic (floating point) benchmark on a dual quad core (length of the tasks varied through command line parameters) (right)

3. Experimental results

With the `nmc`$_{\texttt{muskel}}$ prototype we performed a set of experiments aimed at validating our $\mathsf{M}^2\mathsf{C}$ targeting strategy. The experiments have been run on a network of multi-core systems in Pisa, including dual quad core, dual dual core and dual core Linux systems and dual core Mac OS X systems. The interconnection network used is a (shared) 100Mbit Fast Ethernet.

Fig. 3 shows results achieved running `nmc`$_{\texttt{muskel}}$ on a single $\mathsf{M}^2\mathsf{C}$ system. In this case, the user program, the `Manager` and the distributed macro data flow interpreter instances all run on a single SMP target machine. The left part of this Figure shows the results achieved when executing a synthetic floating point benchmark on different kinds of single $\mathsf{M}^2\mathsf{C}$ configurations: a dual Xeon machine, a dual dual core and a dual quad core machine, all running Linux (kernel 2.6). As expected, the dual (single core) Xeon stops scaling at 2, while keeping efficiency higher than 80% up to 4 threads, the dual dual core scales up to 4 and the dual quad core scales up to 8. The right part of the same Figure shows completion times and efficiency in relation to average computational grain in the macro data flow instructions derived from the skeleton code according to the `muskel` semantics. Larger grain values present better efficiency than computations whose grain is smaller. These results have been accomplished on a single dual quad core Linux 2.6 system.

Fig. 4 shows the results achieved using `nmc`$_{\texttt{muskel}}$ targeting a cluster of multicore machines interconnected by a Fast Ethernet and running either Linux (kernel 2.6, different distributions) or Mac OS X (10.5.7). All the machines used were running Java 1.6. The left part of the Figure plots completion time and efficiency relative to synthetic applications with quite coarse grain macro data flow instructions. Each macro data flow instruction executed to compute the program took around 0.8 secs to execute on the different machines, after receiving a small number of bytes representing the input data (less

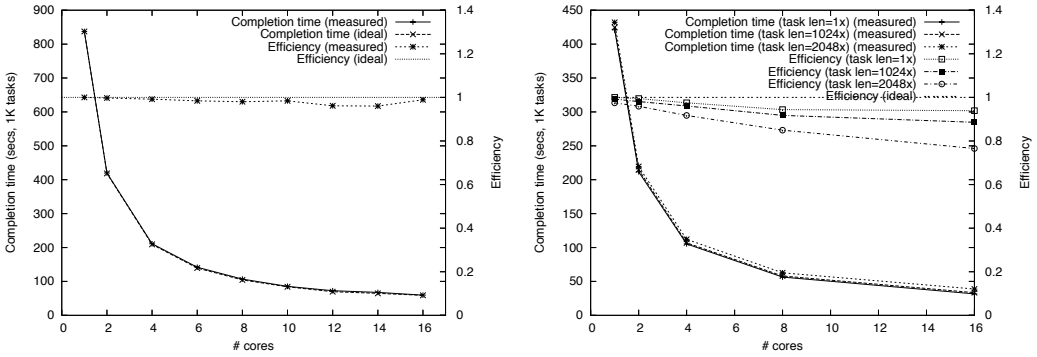

Figure 4. `nmc_muskel`: efficiency (left) and effect of computational grain (right) relative to the execution of skeleton programs on a cluster of multi core machines (1 dual quad core (Xeon E5420, 2.5GHz, Linux RedHat, kernel 2.6, $n_p = 1$), 1 dual dual core (Xeon E5150, 2.66GHz, Linux Debian, kernel 2.6, $n_p = 1.1$), 1 dual core hyperthreading (Xeon, 2.80GHz, Linux RedHat, kernel 2.6, $n_p = 0.48$), 1 dual core (Core 2 duo 2.0GHz, Mac OS 10.5.7, $n_p = 0.85$))

that the Fast Ethernet MTU) and delivering a comparable sized result. Both ideal curve and efficiency consider normalized weight for the different cores in the systems used. The numbers shown as n_p in the Figure caption represent the relative speed of the processors, taking into account both the processor speed *and* the time spend to "ping" the machine, as a raw measure of its "network speed". The right part of the Figure shows effects of computational grain on efficiency. Here we used a synthetic program generating smaller grain macro data flow instructions and we varied the amount of data consumed (input token(s)) and produced (output token(s)) by the instructions. It is clear that as soon as the computational grain decreases, the efficiency also decreases consistently. We should point out that the decrease in efficiency is partially due to the high serialization penalty incurred when sending input tokens (retrieving output tokens) to (from) the remote interpreters with standard Java RMI.

Last but not least, we compared the execution times achieved using `nmc_muskel` with the times achieved using a dedicated, hand coded implementation programmed using plain TCP/IP sockets. For the synthetic benchmarks used in the experiments relative to Fig. 3 and 4 we measured a constant initialization overhead of about 2 seconds, but then the time spent executing the application with `nmc_muskel` does not exceed the time of the hand coded implementation by more than 8%.

4. Conclusions

We discussed how a macro data flow based skeleton framework conceived to target workstation clusters can be transformed in such a way that multi/many core clusters may be

efficiently targeted. The proposed implementation uses only standard Java mechanisms and, this notwithstanding, achieves respectable performance in medium to coarse grain skeleton programs. This is due to the structure of the distributed macro data flow used to implement skeleton programs in `muskel`.

We did not consider targeting heterogeneous multi/many-core systems at the moment, such as those including GPUs or FPGAs. GPUs, in particular, are efficient in executing data parallel (structured) code and we are investigating how to accelerate data parallel skeleton execution when GPUs are available. This research area is being investigated using `muskel`-like C-based prototypes and we will soon have results to present in this area.

References

[1] Marco Aldinucci, M. Danelutto, and Patrizio Dazzi. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4):325–341, December 2007.

[2] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Management in distributed systems: a semi-formal approach. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Proc. of 13th Intl. Euro-Par 2007 Parallel Processing*, volume 4641 of *LNCS*, pages 651–661, Rennes, France, August 2007. Springer.

[3] Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. P^3L: a structured high level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[4] Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In *13th International Euro-par Conference: Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81, Rennes, France, 2007. Springer-Verlag.

[5] Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[6] Marco Danelutto. Efficient support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(1):41–56, 2001.

[7] Marco Danelutto. QoS in parallel programming through application managers. In *Proc. of Intl. Euromicro PDP: Parallel Distributed and network-based Processing*, pages 282–289, Lugano, Switzerland, February 2005. IEEE.

[8] J. Dünnweber and S. Gorlatch. HOC-SA: A grid service architecture for higher-order components. In *IEEE International Conference on Services Computing, Shanghai, China*, pages 288–294. IEEE, September 2004.

[9] Antonio Gonzales. Elastic Parallel Architectures, 2008. Invited talk at Euro-Par 2008, Las Palmas, Gran Canaria (E), available at `http://europar2008.caos.uab.es/documentos/presentation_gonzalez.pdf`.

[10] Empowering Advancement with revolutionary energy-efficient performance, 2007. `http://www.intel.com/technology/computing/dual-core/index.htm`.

[11] TeraFLOPS Research Chip, 2008. `http://techresearch.intel.com/articles/Tera-Scale/1449.htm`.

[12] Herbert Kuchen. A skeleton library. In B. Monien and R. Feldman, editors, *Proc. of 8th Intl. Euro-Par 2002 Parallel Processing*, volume 2400 of *LNCS*, pages 620–629, Paderborn, Germany, August 2002. Springer.

[13] Kiminori Matsuzaki. Sketo home page, 2009. `http://www.ipl.t.u-tokyo.ac.jp/sketo/`.

[14] Jayadev Misra and William R. Cook. Computation orchestration: A basis for a wide-area computing. *Software and Systems Modeling*, 2006. DOI 10.1007/s10270-006-0012-1.

[15] The muesli home page, 2006. `http://www-wi.uni-muenster.de/pi/personal/kuchen.php`.

[16] Marco Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.