Deriving Grid Applications from Abstract Models

Peter Kilpatrick
p.kilpatrick@qub.ac.uk
QUB
Dept. of Computer Science – Queen's University Belfast
University Road, Belfast BT7 1NN, UK.

Marco Danelutto, Marco Aldinucci
{marcod,aldinuc}@di.unipi.it
UNIPI
Dept. of Computer Science – University of Pisa
Largo B. Pontecorvo 3, Pisa, Italy

CoreGRID Technical Report
Number TR-0085
April 4, 2007

Institute on Programming Model

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

Deriving Grid Applications from Abstract Models

Peter Kilpatrick
p.kilpatrick@qub.ac.uk
QUB
Dept. of Computer Science – Queen's University Belfast
University Road, Belfast BT7 1NN, UK.

Marco Danelutto, Marco Aldinucci
{marcod,aldinuc}@di.unipi.it
UNIPI
Dept. of Computer Science – University of Pisa
Largo B. Pontecorvo 3, Pisa, Italy

CoreGRID TR-0085
April 4, 2007

Abstract

An Orc-based abstract model of grid computation is used as a basis for re-engineering a grid-oriented skeleton based programming environment to remove one of the skeleton system's recognized weak points. Lightweight reasoning about non-functional properties is used to guide the redesign of the system. It is argued that such a use of a formal model delivers significant return for small investment. The approach is then extended by enhancing Orc specifications with metadata to capture non-functional properties of systems being specified. It is shown that this approach allows reasoning about cost and security in a qualitive way that is consistent with actual experimental results.

1 Introduction

In [13] Stewart et al. presented an abstract component-based model of grid computation described using Misra and Cooke's Orc notation [11]. The focus of that work was on capturing dynamic aspects of grid programming by specifying a manager responsible for grid site selection, component placement, execution monitoring and component re-placement. Ideally, such a model acts not only as a means of describing systems but also provides a basis for reasoning about their properties. The work reported on here is aimed at assessing the suitability of the model of [13] for such reasoning by using it as the basis on which to re-engineer the grid-oriented `muskel` skeleton programming environment [2] to modify some of its non-functional properties.

The `muskel` system, introduced by Danelutto in [7] and further elaborated in [8, 2], reflects two modern trends in distributed system programming: the use of program skeletons and the provision of means for marshalling resources in the presence of the dynamicity that typifies many current distributed computing environments, e.g. grids. `muskel` allows the user to describe an application in terms of generic skeleton compositions. The description is then translated to a macro data flow graph [6] and the graph computed by a distributed data flow interpreter [8]. Central to the `muskel` system is the concept of a *manager* that is responsible for recruiting the computing resources used to implement the distributed data flow interpreter, distributing the fireable data flow instructions (tasks) and monitoring the activity of the computations.

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

While the performance results demonstrated the utility of muskel, it was noted in [8] that the centralized data flow instruction repository (taskpool) represented a bottleneck and the manager a potential single point of failure. The work reported on here addresses the latter of these issues.

The Orc-based model of [13] was viewed as being apt for two reasons. First, the job of the muskel manager is one of orchestrating computational resources and tasks and is thus closely related to the manager of [13]. And, second, while there are many process calculi that may be used to describe and reason about distributed systems, the syntax of Orc was felt to be more appealing to the distributed system developer whose primary interest lies not in describing and proving formal properties of systems. This was an important consideration in this study. For, the intent was not to embark upon a full-blown formal development of a modified muskel manager, with attendant formulation and proof of its properties, but rather to discover what return might be obtained by a system designer from the use of such a formal notation for *modest* effort. In this sense, the aim was in keeping with the lightweight approach to formal methods as advocated by, inter alia, Agerholm and Larsen [1].

The approach taken was to reverse engineer the original muskel manager implementation to obtain an Orc description; attempt to derive, in semi-formal fashion, a specification of a modified manager based on decentralized management; and, use this derived specification as a basis for modifying the original code to obtain the decentralized management version of muskel.

Following experience with the muskel specifications, a further step along the path of using Orc specifications together with semi-formal reasoning to aid grid system design was taken: Orc was extended with *metadata* to describe non-functional properties such as deployment information. This could be used, for example, to describe the mapping of application parts (e.g. components, modules) onto a grid platform. The approach is consistent with the current trend of keeping decoupled the functional and non-functional aspects of an application. It is proposed that the use of metadata in this way introduces a new dimension for reasoning about the orchestration of a distributed system by allowing a narrowing of the focus from the general case.

Section 2 introduces the muskel system and section 3 the Orc notation. Section 4 describes the re-engineering of muskel and presents experimental results comparing the two versions. In section 5 the use of metadata with Orc is introduced and in section 6 its use with the muskel specifications is described. Finally, in section 7 the work is summarised and pointers to future work are given.

2 muskel: an overview

muskel is a skeleton based parallel programming environment written in Java. The distinguishing feature of muskel with respect to other skeleton environments [5, 9] is the presence of an application manager. The muskel user instantiates a manager by providing the skeleton program to be executed, the input and the output streams containing the (independent) tasks to be computed and the results, respectively, and a performance contract modelling user performance expectations (currently, the only contract supported is the ParDegree one, requesting the manager to maintain a constant parallelism degree during application computation). The user then requests invocation of the eval() method of the manager and the application manager takes care of all the details relating to the parallel execution of the skeleton program.

When the user requires execution of a skeleton program, the muskel system behaves as follows. The skeleton program is compiled to a macro data flow graph, i.e. a data flow graph of instructions modelled by significant portions of Java code corresponding to user Sequential skeletons [6]. A number of processing resources (sufficient to ensure the user performance contract) running an instance of the muskel run time are recruited from the network. The muskel run time on these remote resources provides an RMI object that can be used to compute arbitrary macro data flow instructions, such as those derived from the skeleton program. For each task appearing on the input stream, a copy of the macro data flow graph is instantiated in a centralized TaskPool, with a fresh graph id [8]. A ControlThread is started for each of the muskel remote resources (RemoteWorkers) just discovered. The ControlThread repeatedly looks for a fireable instruction in the task pool (the data-flow implementation model ensures that all fireable instructions are independent and can be computed in parallel) and sends it to its associated RemoteWorker. That RemoteWorker computes the instruction and returns the results. The results are either stored in the appropriate data flow instruction(s) in the task pool or delivered to the output stream, depending on whether they are intermediate results or final ones. In the event of RemoteWorker failure, i.e. if either the remote node or the network connecting it to the local machine fails, the ControlThread informs the manager and it, in turn, requests the name of another machine running the muskel run time support from a centralized *discovery service*

and forks a new `ControlThread` to manage it, while the `ControlThread` managing the failed remote node terminates after reinserting in the `TaskPool` the macro data flow instruction whose computation failed [7]. Note that the failures handled by the `muskel` manager are *fail-stop* failures, i.e. it is assumed that an unreachable remote worker will not simply restart working again, or, if it restarts, it does so in its initial state. `muskel` has already been demonstrated to be effective on both clusters and more widely distributed workstation networks and grids (see also `www.di.unipi.it/~marcod/Muskel`).

3 The Orc notation

The orchestration language Orc has been introduced by Misra and Cook [11]. Orc is targeted at the description of systems where the challenge lies in organising a set of computations, rather than in the computations themselves. Orc has, as primitive, the notion of a site call, which is intended to represent basic computations. A site, which represents the simplest form of Orc expression, either returns a *single* value or remains silent. Three operators (plus recursion) are provided for the orchestration of site calls:

1. operator $>$ (sequential composition)
   $E_1 > x > E_2(x)$ evaluates E_1, receives a result x, calls E_2 with parameter x. If E_1 produces two results, say x and y, then E_2 is evaluated twice, once with argument x and once with argument y. The abbreviation $E_1 \gg E_2$ is used for $E_1 > x > E_2$ when evaluation of E_2 is independent of x.

2. operator $|$ (parallel composition)
   $(E_1 \mid E_2)$ evaluates E_1 and E_2 in parallel. Both evaluations may produce replies. Evaluation of the expression returns the merged output streams of E_1 and E_2.

3. where (asymmetric parallel composition)
   E_1 where $x :\in E_2$ begins evaluation of both E_1 and $x :\in E_2$ in parallel. Expression E_1 may name x in some of its site calls. Evaluation of E_1 may proceed until a dependency on x is encountered; evaluation is then delayed. The first value delivered by E_2 is returned in x; evaluation of E_1 can proceed and the thread E_2 is halted.

Orc has a number of special sites:

- 0 never responds (0 can be used to terminate execution of threads);
- if b returns a signal if b is true and remains silent otherwise;
- *RTimer(t)*, always responds after t time units (can be used for time-outs);
- let always returns (publishes) its argument.

Finally, the notation $(|i : 1 \leq i \leq 3 : worker_i)$ is used as an abbreviation for $(worker_1|worker_2|worker_3)$.

4 Re-engineering `muskel` using Orc

4.1 muskel manager: an Orc description

The Orc description presented focusses on the management component of `muskel`, and in particular on the discovery and recruitment of new remote workers in the event of remote worker failure. The compilation of the skeleton program to a data flow graph is not considered.

While Orc does not have an explicit concept of "process", processes may be represented as expressions which, typically, name channels that are shared with other expressions. In Orc a channel is represented by a site [11]. *c.put(m)* adds m to the end of the (FIFO) channel, *c*, and publishes a signal. If the channel is non-empty *c.get* publishes the value at the head and removes it; otherwise the caller of *c.get* suspends until a value is available.

The activities of the processes of the `muskel` system are now described, followed by the Orc specification.

System The *system* comprises a program, *pgm*, to be executed (for simplicity a single program is considered: in general, a set of programs may be provided here); a set of tasks which are initially placed in a *taskpool*; a *discovery* mechanism which makes available processing engines *(remoteworkers)*; and a *manager* which creates control threads

and supplies them with remote workers. t is the time interval at which potential remote worker sites are polled; and, for simplicity, also the time allowed for a remote worker to perform its calculation before presumption of failure.

Discovery It is assumed that the call *g.can_execute(pgm)* to a *remote worker* site returns its name, g, if it is capable of (in terms of resources) and willing to execute the program *pgm*, and remains silent otherwise. The call *rworkerpool.add(g)* adds the remote worker name g to the pool provided it is not already there. The *discovery* mechanism carries on indefinitely to cater for possible communication failure.

Manager The *manager* creates a number *(contract)* of control threads, supplies them with remote worker handles, monitors the control threads for failed remote workers and, where necessary, supplies a control thread with a new remote worker.

Control thread A control thread *(ctrlthread)* repeatedly takes a task from the *taskpool* and uses its remote worker to execute the program *pgm* on this task. A result is added to the *resultpool*. (Note: Here, for simplicity, it is assumed that all results are final.) A time-out indicates remote worker failure which causes the control thread to execute a call on an *alarm* channel while returning the unprocessed task to the *resultpool*. The replacement remote worker is delivered to the control thread via a channel, c_i.

Monitor The *monitor* awaits a call on the *alarm* channel and, when received, recruits and supplies the appropriate control thread, i, with a new remote worker via the channel, c_i.

$$
\begin{array}{l}
system(pgm, tasks, contract, G, t) \triangleq \\
\quad taskpool.add(tasks) \\
\quad \mid discovery(G, pgm, t) \\
\quad \mid manager(pgm, contract, t)
\end{array}
$$

$$
\begin{array}{l}
discovery(G, pgm, t) \triangleq (\mid_{g \in G} \ (\ \textsf{if}\ remw \neq false \ \gg rworkerpool.add(remw) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textsf{where}\ remw :\in \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\ \ g.can\_execute(pgm) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid Rtimer(t) \gg let(false)\ ) \\
\qquad\qquad\qquad\qquad\qquad)\\
\qquad\qquad\qquad\qquad) \gg discovery(G, pgm, t)
\end{array}
$$

$$
\begin{array}{l}
manager(pgm, contract, t) \triangleq \\
\quad \mid i : 1 \leq i \leq contract : (rworkerpool.get > remw > ctrlthread_i(pgm, remw, t)) \\
\quad \mid monitor
\end{array}
$$

$$
\begin{array}{l}
ctrlthread_i(pgm, remw, t) \triangleq taskpool.get > tk > \\
\quad (\ \ \textsf{if}\ valid \ \gg \ resultpool.add(r) \ \gg \ ctrlthread_i(pgm, remw, t) \\
\quad\ \mid \textsf{if}\ \neg valid \ \gg \ (\ \ taskpool.add(tk) \\
\qquad\qquad\qquad\qquad \mid alarm.put(i) \ \gg \ c_i.get > w > ctrlthread_i(pgm, w, t) \\
\qquad\qquad\qquad\quad) \\
\quad) \\
\qquad \textsf{where}\ (valid, r) :\in \\
\qquad\quad (\ \ remw(pgm, tk) > r > let(true, r) \\
\qquad\quad \mid Rtimer(t) \ \gg \ let(false, 0) \\
\qquad\quad)
\end{array}
$$

$$
\begin{array}{l}
monitor \triangleq alarm.get > i > rworkerpool.get(remw) > remw > c_i.put(remw) \\
\qquad\qquad \gg \ monitor
\end{array}
$$

4.2 Decentralized management: derivation

In the `muskel` system described thus far, the *manager* is responsible for the recruitment and supply of (remote) workers to control threads, both initially and in the event of worker failure. Clearly, if the manager fails, then, depending on the time of failure, the fault recovery mechanism will cease or, at worst, the entire system of control thread recruitment will fail to initiate properly. Thus, the aim is to devolve this management activity to the control threads themselves, making each responsible for its own worker recruitment.

The strategy adopted is to examine the execution of the system in terms of traces of the site calls made by the processes and highlight management related communications. The idea is to use these communications as a means of identifying where/how functionality may be dispersed. In detail, the strategy proceeds as follows:

1. Focus on communication actions concerned with management. Look for patterns based on the following observation. Typically communication occurs when a process, A, generates a value, x, and communicates it to B. Identify occurrences of this pattern and consider if generation of the item could be shifted to B and the communication removed, with the "receive" in B being replaced by the actions leading to x's generation. For example:
   $A:\quad \ldots a1, a2, a3, send(x), a4, a5, \ldots$
   $B:\quad \ldots b1, b2, b3, receive(i), b4, b5, \ldots$
   Assume that a2, a3 (which, in general, may not be contiguous) are responsible for generation of x, and it is reasonable to transfer this functonality to B. Then the above can be replaced by:
   $A:\quad \ldots a1, a4, a5, \ldots$
   $B:\quad \ldots b1, b2, b3, a2, a3, (b4, b5, \ldots)_{[i/x]}$

2. The following trace subsequences are identified:
   - In control thread: $\ldots$ *alarm.put(i)* $\gg c_i.get > e >$ *ctrlthread*$_i(pgm, e, t) \ldots$
   - In monitor: $\ldots$ *alarm.get* $> i >$ *rworkerpool.get* $> e > c_i$*.put(e)*$\gg \ldots$

3. The subsequence *rworkerpool.get* $> e >$ *ci.put(e)* of *monitor* actions is responsible for generation of a value (a remote worker) and its forwarding to a *ctrlthread* process. In the *ctrlthread* process the corresponding "receive" is $c_i.get$. So, the two trace subsequences are modified to:
   - In control thread: $\ldots$ *alarm.put(i)* $\gg$ *rworkerpool.get* $> e >$ *ctrlthread*$_i$*(pgm,e,t)* $\ldots$
   - In monitor: $\ldots$ *alarm.get* $> i > \ldots$

4. The derived trace subsequences now include the communication of the control thread number, i from *ctrlthread*$_i$ to the *monitor*, but this is no longer required by *monitor*; so, this communication can be removed.

5. Thus the two trace subsequences become:
   - In control thread: $\ldots \gg$ *rworkerpool.get* $> e >$ *ctrlthread*$_i(pgm, e, t) \ldots$
   - In monitor: $\ldots \gg \ldots$

6. Now the specifications of the processes *ctrlthread*$_i$ and *monitor* are examined to see how their definition can be changed to achieve the above trace modification, and consideration is given as to whether such modification makes sense and achieves the overall goal.

   (a) In monitor the entire body apart from the recursive call is eliminated thus prompting the removal of the *monitor* process entirely. This is as would be expected: if management is successfuly distributed then there is no need for centralized monitoring of control threads with respect to remote worker failure.

   (b) In control thread the clause:

   | *alarm.put(i)* $\gg$ $c_i.get > e >$ *ctrlthread*$_i$*(pgm, e, t)*

   becomes

   | *rworkerpool.get* $> w >$ *ctrlthread*$_i$*(pgm,w,t)*

   This now suggests that *ctrlthread*$_i$ requires access to the *rworkerpool*. But the *rworkerpool* is an artefact of the (centralized) manager and the overall intent is to eliminate this manager. Thus, the action *rworkerpool.get* must be replaced by some action(s), local to *ctrlthread*$_i$, which has the effect of supplying a new remote worker. Since there is no longer a remote worker pool, on-the-fly recruitment of a remote worker is required. This can be achieved by using a discovery mechanism similar to that of the centralized manager and replacing *rworkerpool.get* by *discover(G, pgm)*:

   discover(G, pgm) $\triangleq$ *let(rw)* **where** *rw* $:\in |_{g\in G}$ *g.can_execute(pgm)*

   (c) Finally, as there is no longer centralized recruitment of remote workers, the control thread processes are no longer instantiated with their initial remote worker but must recruit it themselves. This requires that

      i. the control thread process be further amended to allow initial recruitment of a remote worker, with the (formerly) recursive body of the process now defined within a subsidiary process, *ctrlprocess*, as shown below.

      ii. the parameter *remw* in *ctrlthread* be replaced by G as the control thread is no longer supplied with an (initial) remote worker, but must handle its own remote worker recruitment by reference to the grid, G.

The result of these modifications is shown in the decentralized manager specification below.

Decentralized Management Here each control thread is responsible for recruiting its own remote worker (using a discovery mechanism similar to that of the centralized manager specification) and replacing it in the event of failure.

$$
\begin{array}{l}
systemD(pgm, tasks, contract, G, t) \triangleq \\
\quad taskpool.add(tasks) \\
\quad |i : 1 \leq i \leq contract : ctrlthread_i(pgm, t, G) \\
\\
ctrlthread_i(pgm, t, G) \triangleq discover(G, pgm) > rw > ctrlprocess(pgm, rw, t, G) \\
\\
discover(G, pgm) \triangleq let(rw) \text{ where } rw :\in |_{g \in G}\ g.can\_execute(pgm) \\
\\
ctrlprocess(pgm, remw, t, G) \triangleq taskpool.get > tk > \\
\quad (\ \text{if } valid \gg resultpool.add(r) \gg ctrlprocess(pgm, rw, t, G) \\
\quad\ |\ \text{if } \neg valid \gg taskpool.add(tk) \\
\qquad\qquad\qquad |\ discover(G, pgm) > w > \\
\qquad\qquad\qquad\quad ctrlprocess(pgm, w, t, G) \\
\quad) \\
\qquad \text{where } (valid, r) :\in \\
\qquad\quad (\ remw(pgm, tk) > r > let(true, r) \\
\qquad\quad\ |\ Rtimer(t) \gg let(false, 0) \\
\qquad\quad)
\end{array}
$$

4.3 Analysis

Having derived a decentralized manager specification, the "equivalence" of the two versions must be established. In this context, equivalent means that the same input/output relationship holds, as clearly the two systems are designed to exhibit different non-functional behaviour.

The input/output relationship (i.e. functional semantics) is driven almost entirely by the *taskpool*, whose contents change dynamically to represent the data-flow execution. This execution primarily consists in establishing an on-line partial order among the execution of fireable tasks. All execution traces compliant to this partial order exhibit the same functional semantics by definition of the underlying data-flow execution model. This can be formally proved by showing that all possible execution traces respecting data-dependencies among tasks are functionally confluent (see [3] for the full proof), even if they do not exhibit the same performance.

Informally, one can observe that a global order among the execution of tasks can not be established ex ante, since it depends on the program and the execution environment (e.g. task duration, remote workers' availability and their relative speed, network connection speed, etc.). So, different runs of the centralized version will typically generate different orders of task execution. The separation of management issues from core functionality, which is a central plank of the `muskel` philosophy, allows the functional semantics of the centralized system to carry over intact to the decentralized version as this semantics is clearly independent of the means of recruiting remote workers.

One can also make an observation about how the overall performance of the system might be affected by these changes. In the centralized management system, the discovery activity is composed with the "real work" of the remote workers by the parallel composition operator: discovery can unfold in parallel with computation. In the revised system, the discovery process is composed with core computation using the sequence operator, $\gg$. This suggests a possible price to pay for fault recovery.

4.4 Decentralized management: implementation

Following the derivation of the decentralized manager version outlined above, the existing `muskel` prototype was modified to introduce distributed fault management and to evaluate the relative cost in terms of performance. As shown above, in the decentralized manager, the *discovery(G, pgm, t)* parallel component of the *system(. . .)* expression become part (the *discover(G, pgm)* expression) of the *ctrlprocess(. . .)* expression. The *discovery* and *discover* definitions are not exactly the same, but *discover* is easily derived from *discovery*. Thus, the code implementing *discovery(G, pgm, t)* was moved and transformed appropriately to give an implementation of *discover(G, pgm)*. This required the modification of just one of the files in the `muskel` package (194 lines of code out of a total of 2575, less than 8%), the one implementing the control thread.

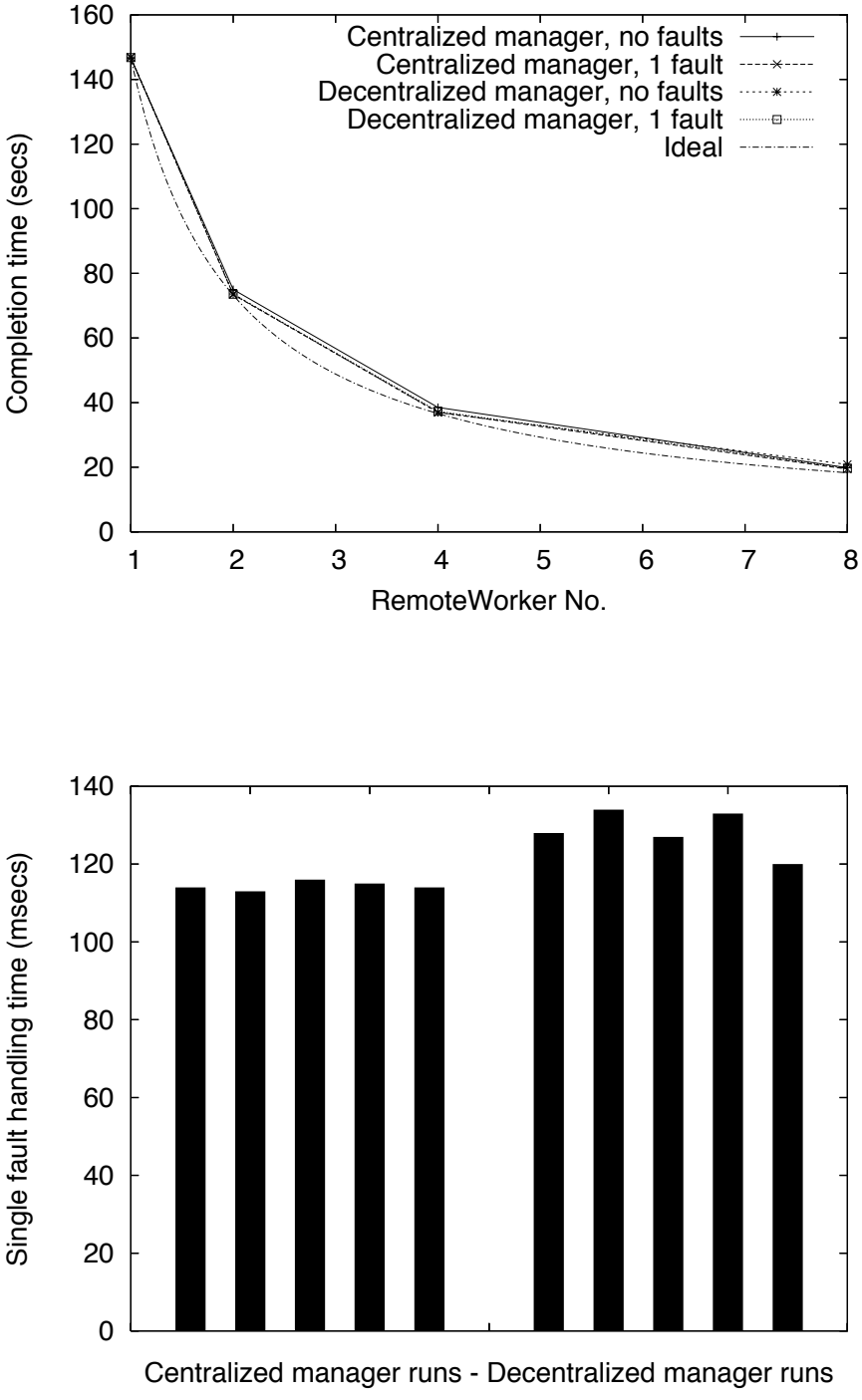

Figure 1: Scalability (upper) and fault handling cost (lower) of modified vs. original `muskel`

Experiments were run using the original and the modified versions to test the functionality and cost of the new implementation. The experiments were run on a Fast Ethernet network of Pentium III machines running Linux and Java 1.5. First the scalability of the decentralized manager version was verified. Figure 1 (upper plot) shows almost perfect scalability up to 8 nodes, comparable to that achieved when running the same program with the original `muskel`, both in the case of no faults and in the case of a single fault per computation. Then the times spent in managing a node fault in the centralized and decentralized versions were compared (Figure 1 lower part). The plot is relative to the time spent handling a single fault. The centralized version performs slightly better than the decentralized one, as anticipated. In the centralized version the discovery of the name of the remote machines hosting the `muskel` RTS is performed *concurrently* with the computation, whereas it is performed *serially* to the main computation in the decentralized version. The rest of the activities performed to handle the fault (lookup of the remote worker RMI object and delivery of the macro data flow) is the same in the two cases.

5 Orc metadata

Following experience of using Orc to specify and reason semi-formally about non-functional properties of the `muskel` system, the next logical step was to attempt somehow to capture such properties "in" the specification. To this end, the association of metadata with an Orc specification was considered. The aim was to devise a mechanism whereby metadata describing non-functional properties of a system could be associated with Orc constructs in such a way that some qualitative analysis could be undertaken, perhaps automatically. In this section some first steps toward this goal are presented.

A generic Orc program, as described in [11], is a set of Orc *definitions* followed by an Orc *goal expression*. The goal expression is the expression to be evaluated when executing the program. Assume $\mathcal{S} \equiv \{s_1, \ldots, s_s\}$ is the set of *sites* used in the program, i.e. the set of all the sites *called* during the evaluation of the top goal expression (the set does not include the pre-defined sites, such as *if* and *Rtimer*, as they are assumed to be available at any user defined site), and $\mathcal{E} \equiv \{e_0, \ldots, e_e\}$ is the set including the goal expression (e_0) and all the "head" expressions appearing in the left hand sides of Orc definitions.

The set of *metadata* associated with an Orc program may be defined as the set: $\mathcal{M} \equiv \{\mu_1, \ldots, \mu_n\}$ where $\mu_i \equiv \langle t_j, md_k \rangle$ with $t_j \in \mathcal{S} \cup \mathcal{E}$ and $md_k = f(p_1, \ldots, p_{n_k})$. f is a generic "functor" (represented by an identifier) and p_i are generic "parameters" (variables, ground values, etc.). The metadata md_k are not further defined as, in general, metadata structure depends on the kind of metadata to be represented. In the following, examples of such metadata are presented.

As is usual, the semantics of Orc is not affected when *meta*data is taken into account. Rather, the introduction of metadata provides a means to restrict the set of actual implementations which satisfy an Orc specification and thereby eases the burden of reasoning about properties of the specification. For example, restrictions can be placed on the relative physical placement of Orc sites in such a way that conclusions can be drawn about their interaction which would not be possible in the general case.

Suppose one wishes to reason about Orc program site "placement", i.e. about information concerning the relative positioning of Orc sites with respect to a given set of *physical resources* potentially able to host one or more Orc sites. Let $\mathcal{R} = \{r_1, \ldots, r_r\}$ be the set of available physical resources. Then, given a program with $\mathcal{S} = \{siteA, siteB\}$ we can consider adding to the program metadata such as $\mathcal{M} = \{\langle siteA, loc(r_1)\rangle, \langle siteB, loc(r_2)\rangle\}$ modelling the situation where $siteA$ and $siteB$ are placed on distinct processing resources. Define also the auxiliary function $location(x) : \mathcal{S} \times \mathcal{E} \rightarrow \mathcal{R}$ as the function returning the location of a site/expression and consider a metadata set *ground* if it contains location tuples relative to *all* the sites in the program.

loc metadata can be used to support reasoning about the "communication costs" of Orc programs. For example, the cost of a communication with respect to the placement of the sites involved can be characterized by distinguishing cases:

$$k_{Comm} = \begin{cases} k_{nonloc} & \textit{iff } location(s_1) \neq location(s_2) \\ k_{loc} & \textit{otherwise} \end{cases}$$

where s_1 and s_2 are the source and destination sites of the communication, respectively and, typically, $k_{nonloc} \gg k_{loc}$.

Consider now a second example of metadata. Suppose "secure" and "unsecure" site locations are to be represented. Secure locations can be reached through trusted network segments and can therefore be communicated with taking no particular care; insecure locations are not trusted, and can be reached only by passing through untrusted network segments, therefore requiring some kind of explicit data encryption to guarantee security. This representation can

be achieved by simply adding to the metadata tuples such as $\langle s_i, trusted() \rangle$ or $\langle s_i, untrusted() \rangle$. Then a costing model for communications that takes into account that transmission of encrypted data may cost significantly more than transmission of plain data can be devised.

$$k_{SecComm} = \begin{cases} k_{UnSecComm} & \textit{iff } \langle s_1, untrusted() \rangle \in \mathcal{M} \\ & \quad \vee \langle s_2, untrusted() \rangle \in \mathcal{M} \\ k_{Comm} & \textit{otherwise} \end{cases}$$

5.1 Generating metadata

So far the metadata considered has been identified explicitly by the user. In some cases he/she may not wish to, or indeed be able to, supply all of the metadata and so it may be appropriate to allow generation of metadata from partial metadata supplied by the user. For example, suppose the user provides only partial location metadata, i.e. metadata relative to the goal expression location and/or the metadata relative to the location of the components of the topmost parallel command found in the Orc program execution. Metadata information available can be used to infer ground location metadata (i.e. location metadata for all $s \in \mathcal{S}$) as follows. Consider two cases: in the first (completely distributed strategy) it is assumed that each time a new site in the Orc program is encountered, the site is "allocated" on a location that is distinct from the locations already used. In the second case (conservative strategy) new sites are allocated in the same location as their parent (w.r.t. the syntactic structure of the Orc program), unless the user/programmer specifies something different in the provided metadata.

More formally, in the first case:

$$\left.\begin{array}{l} E \triangleq f \mid g \\ E \triangleq f(x) \textit{ where } x :\in g \\ E \triangleq f \gg g \\ E \triangleq f > x > g \end{array}\right\} \left\{\begin{array}{l} \langle f, loc(freshLoc(\mathcal{M}) \rangle \\ \langle g, loc(freshLoc(\mathcal{M}) \rangle \\ \textit{ are both added to } \mathcal{M} \end{array}\right.$$

whereas in the second case:

$$\left.\begin{array}{l} E \triangleq f \mid g \\ E \triangleq f(x) \textit{ where } x :\in g \\ E \triangleq f \gg g \\ E \triangleq f > x > g \end{array}\right\} \left\{\begin{array}{ll} \textit{if } \langle f, loc(X) \rangle \in \mathcal{M} & \textit{add nothing} \\ \textit{if } \langle g, loc(X) \rangle \in \mathcal{M} & \textit{add nothing} \\ \textit{if } \langle f, \perp \rangle \in \mathcal{M} & \textit{add } \langle f, location(E) \rangle \\ \textit{if } \langle g, \perp \rangle \in \mathcal{M} & \textit{add } \langle g, location(E) \rangle \end{array}\right.$$

Example
To illustrate the use of metadata, consider the following description of a classical task farm (embarrassingly parallel computation):

$farm(pgm, nw) \triangleq tasksource \mid resultsink \mid workers(pgm, nw)$
$workers(pgm, nw) \triangleq \mid i : 1 \leq i \leq nw : worker_i(pgm)$
$worker(pgm) \triangleq tasksource > t > pgm > y > resultsink(y) \gg worker(pgm)$

The typical goal expression corresponding to this program will be something like $farm(myPgm, 10)$. Suppose the user provides the metadata:

$\forall i \in [1, nw] \langle worker_i, loc(PE_i) \rangle \in \mathcal{M}$
$\langle farm(myPgm, 10), strategy(fullyDistributed) \rangle \in \mathcal{M}$

where $strategy(fullyDistributed)$ means the user explicitly requires that a "completely distributed implementation" be used. An attempt to infer metadata about the goal expression identifies $location(farm(myPgm, 10)) = \perp$ but, as the strategy requested by the user is *fullyDistributed* and as $farm(pgm, nw)$ is defined as a parallel command, the following metadata is added to $\mathcal{M}$:

$\langle tasksource, loc(freshLoc(\mathcal{M})) \rangle$
$\langle resultsink, loc(freshLoc(\mathcal{M})) \rangle$
$\langle workers(pgm, nw), loc(freshLoc(\mathcal{M})) \rangle$.

Next, expanding the *workers* term, gives the term $\mid i : 1 \leq i \leq nw : worker_i(pgm)$, but in this case metadata relative to *worker*$_i$ has already been supplied by the user. At this point

$\mathcal{M} = \{\ \langle tasksource, loc(freshLoc(\mathcal{M}))\rangle, \langle resultsink, loc(freshLoc(\mathcal{M}))\rangle,$
$\langle workers(pgm, nw), loc(freshLoc(\mathcal{M}))\rangle, \langle worker_1, loc(PE_1\rangle, \ldots, \langle worker_{nw}, loc(PE_{nw})\rangle\}$

and therefore is *ground* w.r.t. the program.

Thus, in addition to the location metadata provided by the user it was possible to derive the fact that the locations of *tasksource* and *resultsink* are distinct and, in addition, are different from the locations related to $worker_i$.

Suppose now that the user has also inserted the metadata item $\langle PE_2, untrusted()\rangle$ in addition to those already mentioned. That is, one of the placement locations is untrusted. This raises the issue of how it can be determined whether or not a communication must be performed in a secure way. This information may be inferred from the available metadata as follows. Let functions $source(C)$ denote a site "sending" data and $sink(C)$ denote a site "receiving" data in communication C. Then C *must* be secured iff

$source(C) = X \wedge sink(C) = Y \wedge \langle X, loc(LX\rangle \in \mathcal{M} \wedge \langle Y, loc(LY\rangle \in \mathcal{M}$
$\wedge\ (\langle LX, untrusted()\rangle \in \mathcal{M} \vee \langle LY, untrusted()\rangle \in \mathcal{M})$.

Thus, for the farm example above, the metadata $\langle worker_2, PE_2\rangle$ and $\langle PE_2, untrusted()\rangle$ and the definition

$worker_2(pgm) \triangleq tasksource > t > pgm > y > resultsink \gg worker_2(pgm)$

together with the metadata $\langle tasksource, loc(TS)\rangle$, $\langle resultsink, loc(RS)\rangle$, $\langle TS, trusted()\rangle$, $\langle RS, trusted()\rangle$ lead to the conclusion that the communications represented in the Orc code by

$tasksource > t > pgm.compute(t)$

and by

$pgm.compute(t) > y > resultsink$

within *worker*$_2$ must be secured.

It is worth pointing out that the metadata considered here is typical of the information needed when running grid applications. For example, constraints such as the *loc* ones can be generated to force code (that is, sites) to be executed on processing elements having particular features and information such as that modelled by *untrusted* metadata can be used to denote those cluster nodes that happen to be outside a given network administrative domain and therefore may be more easily subject to "man in the middle" attacks or to some other kind of security related leaks.

6 Metadata exploitation: a case study

In this section the specifications of the two alternative versions of the `muskel` system are enhanced with metadata to facilitate analysis of their performance and security properties.

6.1 Comparison of communication costs

In comparing the two versions of `muskel`, as is typical in such studies, the focus will be on the "steady state" performance, that is, the typical activity of a control thread when it is processing tasks. There are two possibilities: the task is processed normally and the result placed in the *resultpool* or the remote worker fails and the control thread requires a new worker. In analysing the specifications a conservative placement strategy will be assumed; that is, the sub-parts of an entity are assumed to be co-located with their parent unless otherwise stated.

Given the following metadata supplied by the developer:

$\forall rw_i \in G.\langle rw_i, loc(PE_i)\rangle \in \mathcal{M}$
$\langle system, loc(C)\rangle \in \mathcal{M}$
$\langle system(myPgm, tasks, 10, G, 50), strategy(conservative)\rangle \in \mathcal{M}$

the rules for propagation and the strategy adopted ensure that the following metadata are present for both versions:

$\langle rw_i, loc(PE_i)\rangle$, $\langle ctrlthread_i, loc(C)\rangle$, $\langle taskpool, loc(C)\rangle$, $\langle resultpool, loc(C)\rangle$, $\langle rworkerpool, loc(C)\rangle$.

In addition, for the decentralized version, $\langle cntrlprocess, loc(C)\rangle$ is present.

Normal processing For the centralized version, examination of the definition of *cntrlthread* shows that in the case of a normal calculation the following sequence of actions will occur:

$taskpool.get > tk > remw(pgm, tk) > r > let(true, r) \gg resultpool.add(r)$.

Using the metadata, and reasoning in the same way as in the farm example, it can be seen that the communication

of the task *tk* to the remote worker and the subsequent return of the result *r* to the control thread represent non-local communications; all other communications in this sequence are local.

Similar analysis of the decentralized version reveals an identical series of actions for normal processing and an identical pattern of communications. Naturally then, similar results from the two versions for normal processing would be expected, and indeed this is borne out by experiment - see section 6.3.

Fault processing Now consider the situation where a remote worker fails during the processing of a task. In both versions the *Rtimer* timeout occurs, the task being processed is returned to the *taskpool* and a new worker is recruited.

In the centralized version the following sequence of events occurs:

$taskpool.get \gg Rtimer(t) \gg let(false, 0) \gg taskpool.add(tk) \gg rworkerpool.get(remw)$

while in the decentralized version the events are effectively:

$taskpool.get \gg Rtimer(t) \gg let(false, 0) \gg taskpool.add(tk) \gg rw.can\_execute(pgm) > rw > let(g)$

where *rw* is the first site in *G* to respond.

Analysis of these sequences together with the metadata reveals that the comparison reduces to the local communication to the *rworkerpool* in the centralized version versus the non-local call to the remote site *rw* in the decentralized version. This comparison would suggest that, in the case of fault handling, the centralized version would be faster than the decentralized version and, again, this is borne out by experiment.

6.2 Comparison of security costs

Consider now the issue of security. Suppose that one of the remote workers, say rw_2, is in a non-trusted location (that is $\langle PE_2, untrusted() \rangle \in \mathcal{M}$).The implications of this can be determined by analysing the specification together with the metadata. In this case, as $\langle rw_2, loc(PE_2) \rangle \in \mathcal{M}$ we can conclude that $cntrlthread_2$ will be affected (while it is operating with its initially allocated remote worker) to the extent that the communications to and from its remoteworker must be secured. This prompts reworking of the specification to split the control threads into two parallel sets: those requiring secure communications and those operating exclusively in trusted environments. In this way the effect, and hence cost, of securing communications can be minimised. Experimental results in section 6.3 illustrate the cost of securing the communications with differing numbers of control threads.

6.3 Experimental results

A number of experiments were run, on a distributed configuration of Linux machines, aimed at verifying that the kind of results derived by working on Orc specifications of `muskel` together with metadata can be considered realistic.

First it was confirmed that centralized and decentralized manager versions of `muskel` actually "behave almost the same" when no faults occur in the resources used for remote program execution. The following table presents completion times (in seconds) for runs of the same program with the two `muskel` versions, on a variable amount of remote resources. The average difference between the centralized and decentralized version was 1% ($\sigma = 1.2$), demonstrating the two versions are substantially equivalent in the case of no faults.

`muskel` *version*	*1 PE*	*2 PEs*	*3 PEs*	*4 PEs*
Centralized manager	146.46	74.02	38.53	19.86
Decentralized manager	146.93	73.64	36.93	20.87
Difference	0.4%	-1.2%	-1.6%	1.0%

Then the case where remote resources fail was considered. The table below shows the time spent in handling a single fault (in msecs) in 5 different runs. The distributed manager `muskel` version takes longer to handle a single fault, as expected. Moreover, the standard deviation of the time spent handling the single fault is $\sigma = 1.1$ in the centralized case but it is $\sigma = 5.6$ in the decentralized case, reflecting the fact that in this case remote resources must be contacted to recruit a new worker and the time spent depends on the location of the remote resource recruited.

`muskel` *version*	Run_1	Run_2	Run_3	Run_4	Run_5	*Average*
Centralized manager	114	113	116	115	114	114.4
Decentralized manager	128	134	127	133	129	128.4

Finally, the effect of selectively deciding (using user-defined metadata) which `muskel` remote workers have to be handled by means of secure (SSL based, in this case) communications was measured. The plot in Figure 2 shows the completion time of a `muskel` program whose remote worker sites are running on a variable mix of *trusted* and *untrusted* locations. The more *untrusted* locations are considered, the poorer the scalability that is achieved. This demonstrates that metadata exploitation to identify the minimal set of remote workers that actually *need* to be handled with secure communications can be very effective.

Some of the experimental results presented here were generated *prior* to the development of the Orc based techniques discussed in this work. They were aimed at verifying exactly the differences in the centralized and distributed manager versions of `muskel` and the impact of adopting secure communications with remote `muskel` workers. This required substantial programming effort, debugging and fine tuning of the new `muskel` versions and, last but not least, extensive grid experimental sessions. The approach discussed in this work allowed the same qualitative results to be obtained by just developing Orc specifications of the `muskel` prototypes, enhancing these specifications with metadata and then reasoning with these enhanced specifications, without actually writing a single line of code and without the need for running experiments on a real grid.

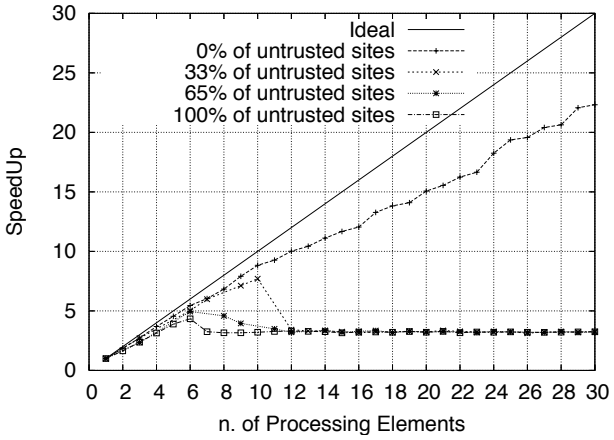

Figure 2: Comparison of runs involving different percentages of *untrusted* locations

7 Conclusions

The manager component of the `muskel` system has been re-engineered to provide distributed remote worker discovery and fault recovery. A formal specification of the component, described in Orc, was developed. The specification provided the developer with a representation of the manager that allowed exploration of its properties and the development of what-if scenarios while hiding the inessential detail. By studying the communication patterns present within the process traces, the developers were able to derive a system exhibiting equivalent core functionality, while having the desired decentralized management properties. The derivation proceeded in a series of semi-formally justified steps, with incorporation of insight and experience as exemplified by the inclusion of expressions such as "reasonable to transfer this functionality" and "such modification makes sense".

The claim is that the creation of such a specification facilitates exploration (and documentation) of ideas and delivers much return for small investment. Also, lightweight reasoning about the derived specification gave the developers some insight into the expected performance of the derived implementation relative to its parent.

The authors suggest that Orc is an appropriate vehicle for the description of management systems of the sort described here. Its syntax is small and readable; its constructs allow for easy description of the sorts of activities that typify these systems (in particular the asymmetric parallel composition operator facilitates easy expression of concepts such as time-out and parallel searching); and the $site$ abstraction allows clear separation of management activity from core functionality.

Finally, the use of Orc specifications to describe and reason about management systems has been taken a step further by the addition of metadata capturing aspects of the non-functional properties of systems. Experiments with the `muskel` system have indicated that this use of metadata enabled qualitative performance comparison between two different versions to be performed, as well as the determination of how the overhead introduced by security techniques can be minimized. These theoretical results were compared with actual experimental results and it was verified that they qualitatively match. Thus, the availability of an Orc model on which to "hang" the metadata allows metadata to be exploited *before* the actual implementation is available.

Current work is aimed at formalizing and automating the metadata based techniques discussed in this work. In particular, the intent is to implement tools to support the reasoning procedures adopted. The whole approach, based on Orc, encourages the usage of semi-formal reasoning to support program development (both program design and refinement) and has the potential to substantially reduce experimentation by allowing the exploration of alternatives prior to costly implementation.

In a separate but related thread, work is ongoing to develop an automatic tool that takes as input an Orc program and produces a set of Java code files that:

- completely implements the "parallel structure" of the application modelled by the Orc program. That is, a `main` Java code is produced for each of the parallel/distributed entities in the Orc code, suitable communication code is introduced to implement the interactions among Orc parallel/distributed entities, and so on. Moreover, a "script" is created that, taking as input an XML file with the description of the target architecture (names of the nodes available, features of each node, interconnection framework information, and so on), deploys the Java code to the target nodes and starts its distributed execution.
- provides hooks to programmers to insert the needed "functional code" (i.e. the code actually computing the application results) in the distributed application parallel framework.
- can be used to "prototype" a parallel application skeleton without the need to spend time to program all of the cumbersome details related to distributed application development (process decomposition, mapping and scheduling, communication and synchronization implementation, etc.) that usually take such a long time.
- can be used to run and compare several different skeletons, to evaluate which is the "best" implementation.

References

[1] S. Agerholm and P. G. Larsen. A lightweight approach to formal methods. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *FM-Trends*, volume 1641 of *LNCS*, pages 168–183. Springer, 1998.

[2] M. Aldinucci and M. Danelutto. Algorithmic skeletons meeting grids. *Parallel Computing*, 32(7):449–462, 2006. DOI:10.1016/j.parco.2006.04.001.

[3] M. Aldinucci and M. Danelutto. Skeleton based parallel programming: functional and parallel semantic in a single shot. *Computer Languages, Systems and Structures*, 2006. DOI 10.1016/j.cl.2006.07.004, in press.

[4] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Management in distributed systems: a semi-formal approach. Technical Report TR-07-05, Università di Pisa, Dipartimento di Informatica, Feb. 2007.

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

[6] 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 Intl. PARCO 99: Parallel Computing*, Parallel Computing Fundamentals & Applications, pages 460–467. Imperial College Press, 1999.

[7] M. 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, Feb. 2005. IEEE.

[8] M. Danelutto and P. Dazzi. Joint structured/non structured parallelism exploitation through data flow. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *Proc. of ICCS: Intl. Conference on Computational Science, Workshop on Practical Aspects of High-level Parallel Programming*, LNCS, Reading, UK, May 2006. Springer.

[9] H. Kuchen. The muesli home page, 2006. `http://www.wi.uni-muenster.de/PI/forschung/Skeletons/`.

[10] M. Aldinucci and M. Danelutto and P. Kilpatrick. A framework for prototyping and reasoning about grid systems, 2007. submitted to PARCO 2007.

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

[12] A. Stewart, J. Gabarró, M. Clint, T. J. Harmer, P. Kilpatrick, and R. Perrott. Estimating the reliability of web and grid orchestrations. In S. Gorlatch, M. Bubak, and T. Priol, editors, *Integrated Reserach in Grid Computing*, pages 141–152, Kraków, Poland, Oct. 2006. CoreGRID, Academic Computer Centre CYFRONET AGH.

[13] A. Stewart, J. Gabarró, M. Clint, T. J. Harmer, P. Kilpatrick, and R. Perrott. Managing grid computations: An orc-based approach. In M. Guo, L. T. Yang, B. D. Martino, H. P. Zima, J. Dongarra, and F. Tang, editors, *ISPA*, volume 4330 of *LNCS*, pages 278–291. Springer, 2006.