Prototyping and reasoning about distributed systems: an Orc based framework

M. Aldinucci
Dept. Computer Science
Univ. Pisa – Italy

M. Danelutto
Dept. Computer Science
Univ. Pisa – Italy

P. Kilpatrick
Dept. Computer Science
Queen's University Belfast – UK

CoreGRID Technical Report
Number TR-0102
August 1st, 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

Prototyping and reasoning about distributed systems: an Orc based framework

M. Aldinucci
Dept. Computer Science
Univ. Pisa – Italy

M. Danelutto
Dept. Computer Science
Univ. Pisa – Italy

P. Kilpatrick
Dept. Computer Science
Queen's University Belfast – UK

CoreGRID TR-0102
August 1st, 2007

Abstract

We discuss a framework supporting fast prototyping as well as tuning of distributed applications. The approach is based on the adoption of a formal model that is used to describe the orchestration of distributed applications. The formal model adopted (Orc by Misra and Cook) can be used to support semi-formal reasoning about the applications at hand. We build on results achieved earlier and show here how the framework can be used to derive and evaluate alternative orchestrations of a well know parallel/distributed computation pattern; and describe how the same formal model can also be used to support generation of prototypes of distributed applications skeletons directly from the application description.

1 Introduction

The programming of large distributed systems, including grids, presents significant new challenges. For example, the "invisible grid" and "service and knowledge utility" (SOKU) concepts advocated in the EU NGG expert group documents [1, 2] both identify new challenges to be addressed. In particular, an increasingly substantial programming effort is required to set up appropriate "computing structure" for a distributed application: significant efforts have to be invested in the development of a suitable, manageable and maintainable distributed application skeleton, and, ideally, this skeleton should be validated, at least informally, *before* starting actual coding, to avoid spending large amounts of time coding only to discover when running application tuning experiments that one or more of the original skeleton features is wrong.

What is needed is a framework that allows the application programmer to develop a specification of a distributed system using a user-friendly formal notation. Existing formal tools such as the π-calculus[3] are usually perceived to be distant from the "reasoning schemas" which are typical of programmers, and, moreover, their usage requires significant formal calculus capability and experience combined with substantial effort. Therefore, we need something that can replace full formal reasoning about a system's properties with "lightweight" reasoning combining properties of the notation with the developer's domain expertise and experience. Typically such reasoning will allow focus on the particular case, rather than the general and, in this way, significantly reduce the overhead of formal development. In addition, the availability of such a specification will afford the possibility of generating automatically a skeletal implementation that may be used for experimentation prior to full implementation.

We earlier identified Orc by Misra and Cook [4, 5] as a suitable notation to act as a basis for the framework proposed above. We showed how Orc is particularly suitable for addressing *dynamic* properties of distributed systems, although in the current work, for the sake of simplicity, we will consider only static examples/properties. We also showed how an Orc-based framework can be developed to support large distributed application development and tuning[6, 7]. Figure 1 outlines the main features of Orc, in particular those related to the Orc specifications used here. In this work we build on these previous results and the add two main contributions.

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

A *site*, 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. $>$ (sequential composition):$E_1 > x > E_2(x)$ evaluates E_1, receives a result x, calls E_2 with parameter x. 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. $|$(parallel composition): $(E_1 \mid E_2)$ evaluates E_1 and E_2 in parallel. 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, including ***RTimer(t)***, that always responds after t time units (can be used for time-outs). The notation $(|i : 1 \leq i \leq 3 : w_i)$ is used as an abbreviation for $(w_1|w_2|w_3)$.
In Orc processes may be represented as expressions which, typically, name channels which are shared with other expressions. In Orc a channel is represented by a site [4]. *c.put(m)* adds m to the end of the (FIFO) channel 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.

Figure 1: Minimal Orc *compendium*

On the one hand, we show how a cost analysis technique can be described and used to evaluate properties of alternative implementations of the same distributed application. We present Orc models of two alternative and equivalent implementations of a typical distributed use-case and we determine which should perform better by measuring the number and kind of communications performed using a measure of communication cost. To evaluate the communication patterns of the two implementations, we consider metadata denoting *locations* where parallel activities have to be performed. The metadata is set up according to the principles stated in our work at the CoreGRID Symposium[6]. While metadata can be used to describe several distinct aspects of a system, here we consider only metadata items of the form $\langle site/epression, location\rangle$ representing the fact that $site/expression$ is run on the resource $location$. A methodology is introduced to deal with partial user-supplied metadata that allows labelling of all the sites and processes appearing in the Orc specification with appropriate locations. The location of parallel activities, as inferred from the metadata, is then used to cost communications happening during the application execution.

On the other hand, we describe a compiler tool that allows generation of the skeleton of a distributed application directly from the corresponding Orc specification. The user can then complete the skeleton code by providing the functional (sequential) code implementing site and process internal logic, while the general orchestration logic, mechanisms, schedule, etc. are completely dealt with by the compiler generated code. Finally, we show how the code generated from the Orc specifications shown here perform as expected when executed on a distributed target architecture.

The two steps allows us to conclude that we have a framework supporting design, development and tuning of distributed grid applications.

2 Use Case: Parallel Computation of Global State Updates

We introduce a use case to discuss how a costing mechanism can be associated with Orc, via metadata, in such a way that alternative implementations/orchestrations of the same application can be compared. We consider a common parallel application pattern: data items in an input stream are processed independently, with the results being used to update a shared state (see Fig. 2). We consider the easy case, where state updates are performed through an associative and commutative function $\sigma : State \times Result \rightarrow State$. This kind of computation is quite common. For example, consider a parallel ray tracer: contributions of the single rays on the scene can be computed independently and their effect is "summed up" onto the final scene. Adding effects of a ray on the scene can be performed in any order, without affecting the final result. We provide two alternative implementations of this particular parallelism exploitation pattern, parametric in the type of actual computation performed, and we model them in Orc.

2.1 Design 1: no proxy

The first design is based on the classical master/worker implementation where centralized entities provide the items of the input stream and collate the resulting computations in a single state. The *system* comprises a taskpool (entclassnicseriesmodeling the input stream), ***TM***, a state manager, ***SM*** and a set of workers, W_i. The workers repeatedly take

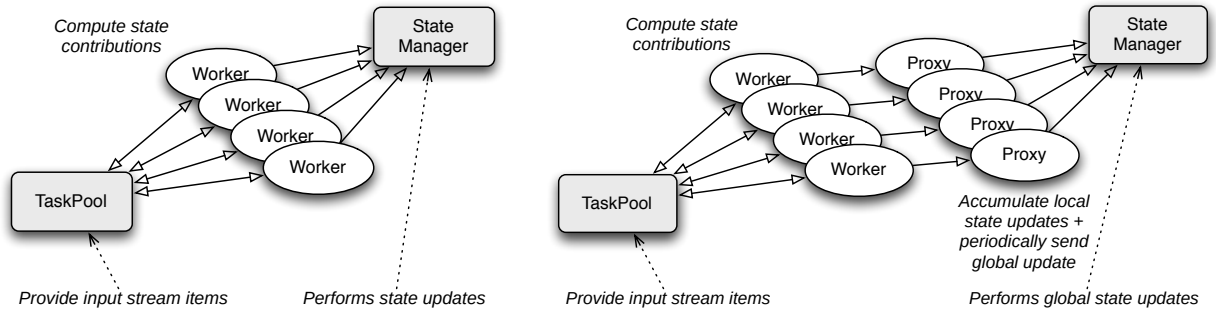

Figure 2: Logical schema corresponding to the two Orc specifications (grey boxes represent sites, white ovals represent processes)

tasks from the taskpool, process them and send the results to the state manager. The taskpool and state manager are represented by Orc sites; the workers are represented by processes (expressions). This specification corresponds to the logical schema in Fig. 2 left and can be formulated as follows:

$$system(TP, SM) \triangleq\ workers(TP, SM)$$
$$workers(TP, SM) \triangleq |\ i : 1 \leq i \leq N : W_i(TP, SM)$$
$$W_i(TP, SM) \triangleq TP.get >tk> compute(tk) >r> SM.update(r) \gg W_i(TP, SM)$$

2.2 Design 2: with proxy

In this design, for each worker, W_i, a proxy, *proxy*$_i$, is interposed between it and the state manager to allow accumulation of partial results before forwarding to the state manager. A proxy executes a *ctrlprocess* in parallel with a *commit* process. The control process receives from its worker, via a channel wc_i, a result and stores it in a local state manager, *LSM*$_i$. Periodically, the *commit* process stores the contents of the *LSM*$_i$ in the global state manager, *SM*. A control thread (and control process) is represented by a process; a local state manager is a site. This corresponds to the schema in Fig. 2 right.

$$system(TP, SM, LSM) \triangleq\ proxies(SM, LSM)\ \mid\ workers(TP)$$
$$proxies(SM, LSM) \triangleq |\ i : 1 \leq i \leq N : proxy_i(SM, LSM_i)$$
$$proxy_i(SM, LSM_i) \triangleq\ ctrlprocess_i(LSM_i)\ \mid\ commit(LSM_i)$$
$$ctrlprocess_i(LSM_i) \triangleq wc_i.get >r> LSM_i.update(r) \gg ctrlprocess_i(LSM_i)$$
$$commit_i(LSM_i) \triangleq\ Rtimer(t) \gg SM.update(LSM_i) \gg commit_i(LSM_i)$$
$$workers(TP) \triangleq |\ i : 1 \leq i \leq N : W_i(TP)$$
$$W_i(TP) \triangleq\ TP.get >tk> compute(tk) >r> wc_i.put(r) \gg W_i(TP)$$

3 Communication Cost Analysis

We introduce now a procedure to determine the cost of an Orc expression evaluation in terms of the communications performed to complete the computation modelled by the expression.

First we make some basic assumptions concerning communications. We assume that a site call constitutes 2 communications (one for the call, one for getting the ACK/result): we do not distinguish between transfer of "real" data and the ACK. We also assume an interprocess communication constitutes 2 communications. In Orc this is denoted by a *put* and a *get* on a channel. (Note: although, in Orc, this communication is represented by two complementary site (channel) calls, we do not consider this exchange to constitute 4 communications.) We identify two cases for sequential composition with passing of a value:

- *site* $>x>$ *site* represents a local transfer of x (cf. a local variable) and so is assumed not to represent a communication, while
- *site* $>x>$ *process*, *process* $>x>$ *site* and *process* $>x>$ *process* all constitute 2 communications.

Finally, we consider how communications may overlap in a parallel computation. We assume a simple and effective model: for Orc parallel commands the communication cost mechanism should take into account that communications happening "internally" to the parallel activities overlap while those involving *shared* external sites (or processes) do not. Therefore, when counting the communications happening within an Orc expression such as:

$$W_i(TP) \triangleq TP.get >tk> compute(tk) >r> wc_i.put(r) \gg W_i(TP)$$

we observe the calls to site *TP* are calls to external, shared sites, whereas the calls to wc_i are related to internal sites. When considering the calls related to the execution of N processes W_i computing M tasks, we count all the calls related to TP but we assume that all the communications related to different wc_i are actually overlapped. Therefore, assuming perfect load balancing, we will count for M calls to TP and $\frac{M}{N}$ calls to wc_i.

To evaluate the number and nature of communications involved in the computation of an Orc expression, we perform two steps.

First we determine the metadata associated with the Orc specification. We assume the user has provided metadata stating placement of relevant sites/processes as well as strategies to derive placement metadata for the sites/processes not explicitly targeted in the supplied metadata (as discussed in previous work[6]). Thus, for the first specification given above, assuming we initially have the metadata

$$\mathcal{M} = \{\langle site, TP\rangle, \langle site, SM\rangle, \langle system, strategy(FullyDistributed)\rangle\}$$

we can derive the metadata

$$\begin{aligned}\mathcal{M} = \ & \{\langle site, TP\rangle, \langle site, SM\rangle, \langle TP, freshLoc(\mathcal{M})\rangle, \langle SM, freshLoc(\mathcal{M})\rangle, \\ & \langle workers, freshLoc(\mathcal{M})\rangle\langle worker_i, freshLoc(\mathcal{M})\rangle\}\end{aligned}$$

where $freshLoc$ returns a resource name not previously used (thus implementing the $strategy(FullyDistributed)$ policy). The strategy *FullyDistributed* indicates that all processes/sites should be placed on unique locations, unless otherwise indicated (by explicit placement). This metadata is *ground* as all sites and all non-terminals have associated locations. For details of metadata derivation see Figures 3 and 4.

The second step counts the communications involved in evaluation of the Orc specification, following the assumptions made at the beginning of this Section.

Consider the first design above. In this case, we assume N worker processes, computing M tasks, with two sites providing the task pool and the state manager (*TP* and *SM*). The communications in each of the workers involve shared, non-local sites, and therefore the total number of communications involved is $2M + 2M$, the former term representing the communications with the task pool and the latter to those with the state manager. All these communications are remote (as the strategy is *FullyDistributed*), involving sending and receiving sites/processes located on different resources, and therefore we cost them r time units. (Had they involved sites/processes on the same processing resources we would have costed them l time units.) Therefore, the overall communication cost of the computation described by the first Orc specification with M input tasks and N workers is $4Mr$. This is independent of N, as expected, as there are only communications related to calls to global, shared sites.

For the version with proxy, we have N W_i, N $ctrlprocess_i$ processes and N LSM_i sites, plus the globally shared TP and SM sites. We assume, from the user supplied metadata, that each $\langle ctrlprocess_i, LSM_i\rangle$ pair is allocated on the same processing element, distinct from the processing element used for $worker_i$, and that all these processing elements are in turn distinct from the two used to host TP and SM. (Again, see Figures 3and 4 for full derivation details.) The communications involved in the computation of M tasks are $2M$ non-local communications (those taking tasks out of the *TP*), $2\frac{M}{k}$ non-local communications (those sending partial contributions to *SM*, assuming k state updates are accumulated locally before invoking a global update on *SM*) and $2 \times 2 \times \frac{M}{N}$ local communications (those performed by the $ctrlProcess$es to get the result of W computations and perform local LSM updates, assuming an even distribution of tasks to workers). Therefore the cost of the computation described by the second Orc specification with M input tasks and N workers is

$$2Mr + \frac{2Mr}{k} + \frac{4Ml}{N}$$

Design 1: no proxy

The (semi-)automatic generation of a communication profile from metadata has the following stages:

1. Add initial metadata.
2. Generate additional metadata for all sites and non-terminals within the specification.
3. Deduce the cost of communications from metadata.

Assume that *TP* and *SM* are sites and that all sites and workers are to be located on distinct locations: a fully distributed strategy.

1. Initially $\mathcal{M} = \{\langle site, TP\rangle, \langle site, SM\rangle\}$
2. Add $\langle system, strategy(FullyDistributed)\rangle \in \mathcal{M}$.
3. Thus $\mathcal{M} = \{\langle site, TP\rangle, \langle site, SM\rangle, \langle system, strategy(FullyDistributed)\rangle\}$.
4. Process the item $\langle system, strategy(FullyDistributed)\rangle$. This replaces this item with $\langle system, freshLoc(\mathcal{M})\rangle$ and adds $\langle workers, strategy(FullyDistributed)\rangle$.
5. Now process the metadata item $\langle workers, strategy(FullyDistributed)\rangle$ This replaces this metadata item with $\langle workers, freshLoc(\mathcal{M})\rangle$ and $\langle worker_i, strategy(FullyDistributed)\rangle$.
   Now $\mathcal{M} = \{\langle site, TP\rangle, \langle site, SM\rangle, \langle workers, freshLoc(\mathcal{M})\rangle, \langle worker_i, strategy(FullyDistributed)\rangle\}$.
6. Process the item $\langle worker_i, strategy(FullyDistributed)\rangle$. This replaces this item by $\langle worker_i, freshLoc(\mathcal{M})\rangle$, $\langle SM, freshLoc(\mathcal{M})\rangle$ and $\langle TP, freshLoc(\mathcal{M})\rangle$ giving:
   $\mathcal{M} = \{\langle site, TP\rangle, \langle site, SM\rangle \langle SM, freshLoc(\mathcal{M})\rangle \langle TP, freshLoc(\mathcal{M})\rangle, \langle SM, freshLoc(\mathcal{M})\rangle,$
   $\langle workers, freshLoc(\mathcal{M})\rangle \langle worker_i, freshLoc(\mathcal{M})\rangle\}$
7. $\mathcal{M}$ is now *ground* because all sites and all non-terminals have associated locations.
8. All the required metadata is now available to allow analysis of the specification with respect to communications. The specification is inspected for occurrences of:
   (a) site calls: there are two, both originating within the process W_i: *TP.get* and *SM.update*. From the metadata, it can be seen that, in each case, the site called and process W_i are in different locations and so each constitutes 2 *remote* communications.
   (b) explicit process communication via a channel: there is none in the specification.
   (c) communication as part of an instance of the sequential operator: there is none.
   (d) communication as part of an instance of the asymmetric parallel operator: there is none.

   Thus, for each task processed, there are four remote communications, so to process M tasks requires $4M$ remote communications.

Figure 3: Full derivation of metadata and communication count relative to site/process locations for design 1

Design 2: with proxy

Assume that *SM*, *TP* and $proxy_i$ are all in separate locations. Assume that W_i and LSM_i are in the same location as $proxy_i$. That is, the essential difference from Specification 1 is the local accumulation of results in LSM_i before forwarding to *SM*.
Assume that the average number of items stored locally before a commit to the SM is k.

1. Initially $\mathcal{M} = \{\langle site, TP\rangle, \langle site, SM\rangle, \langle site, LSM_i\rangle\}$.
2. Add $\langle system, strategy(FullyDistributed)\rangle$ and $\langle proxy_i, strategy(Conservative)\rangle$. The intention is that the (more local) *proxy*$_i$ directive will override the (more global) *system* directive.
3. $\mathcal{M} = \{\langle site, TP\rangle, \langle site, SM\rangle, \langle site, LSM_i\rangle, \langle system, strategy(FullyDistributed)\rangle, \langle proxy_i, strategy(Conservative)\rangle\}$.
4. Add $\langle worker_i, sameLoc(proxy_i)\rangle$ indicating that *Worker*$_i$ should be placed at the same location as *proxy*$_i$
5. Process the "FullyDistributed" directive and remove the corresponding metadatum.
6. $\mathcal{M} = \{\langle site, TP\rangle, \langle site, SM\rangle, \langle site, LSM_i\rangle, \langle system, freshLoc(\mathcal{M})\rangle, \langle proxies, strategy(FullyDistributed)\rangle, \langle workers, strategy(FullyDistributed)\rangle, \langle proxy_i, strategy(Conservative)\rangle, \langle worker_i, sameLoc(proxy_i)\rangle\}$.
7. Process $\langle proxies, strategy(FullyDistributed)\rangle$. Note, at this point $\langle workers, strategy(FullyDistributed)\rangle$ cannot be processed because of the $\langle worker_i, sameLoc(proxy_i)\rangle$ directive.
8. $\mathcal{M} = \{\langle site, TP\rangle, \langle site, SM\rangle, \langle site, LSM_i\rangle, \langle system, freshLoc(\mathcal{M})\rangle, \langle proxies, freshLoc(\mathcal{M})\rangle, \langle proxy_i, loc(PE_i)\rangle, \langle proxy_i, strategy(Conservative)\rangle, \langle workers, strategy(FullyDistributed)\rangle, \langle worker_i, sameLoc(proxy_i)\rangle\}$.
   The location of *proxy*$_i$ is named with an eye to the *sameLoc(proxy*$_i$*)* directive.
9. Process $\langle workers, strategy(FullyDistributed)\rangle$ together with $\langle worker_i, sameLoc(proxy_i)\rangle$.
10. $\mathcal{M} = \{\langle site, TP\rangle, \langle site, SM\rangle, \langle site, LSM_i\rangle, \langle system, freshLoc(\mathcal{M})\rangle, \langle proxies, freshLoc(\mathcal{M})\rangle, \langle proxy_i, loc(PE_i)\rangle, \langle proxy_i, strategy(Conservative)\rangle, \langle workers, freshLoc(\mathcal{M})\rangle, \langle worker_i, strategy(FullyDistributed)\rangle, \langle worker_i, loc(PE_i)\rangle\ \}$
11. Process $\langle proxy_i, strategy(Conservative)\rangle$ and $\langle worker_i, strategy(FullyDistributed)\rangle$ (the latter results in no new metadata).
12. $\mathcal{M} = \{\langle site, TP\rangle, \langle site, SM\rangle, \langle site, LSM_i\rangle, \langle system, freshLoc(\mathcal{M})\rangle, \langle proxies, freshLoc(\mathcal{M})\rangle, \langle proxy_i, loc(PE_i)\rangle, \langle ctrlprocess_i, loc(PE_i)\rangle, \langle commit_i, loc(PE_i)\rangle, \langle LSM_i, loc(PE_i)\rangle \langle workers, freshLoc(\mathcal{M})\rangle, \langle worker_i, loc(PE_i)\rangle\ \}$
13. The specification is now inspected as above to determine the numbers of the various communication types:
    (a) Site calls: *TP.get* is made from W_i to a *freshLoc*$(\mathcal{M})$ and so constitutes two remote communications. $LSM_i.update$ is made from PE_i to LSM_i and so constitutes 2 local comms.; note, however, that for different i these calls are overlapped in parallel: thus we count $\frac{2}{N}$ calls per task processed for N worker processes/proxies.
    $SM.update$ is made from PE_i to a *freshLoc*$(\mathcal{M})$ and so constitutes 2 remote comms.
    (b) Inter process comms.: $wc_i.get$ is between $proxy_i$ and W_i both placed on PE_i and so constitutes 2 local comms. Again, here we count $\frac{2}{N}$, as above.
    (c) Communication as part of an instance of the sequential operator: $wc_i.get >r> LSM_i.update(r)$. Here both $wc_i.get$ and $LSM_i.update(r)$ are executed within $ctrlprocess_i$ and so no comm. takes place.

    For each task processed there are 2 remote comunications + $\frac{4}{N}$ local communications. Moreover, for each k tasks processed there are 2 remote communications. Thus, to process M tasks requires $2M$ remote communications + $\frac{4M}{N}$ local communications + $2M/k$ remote communications.

Figure 4: Full derivation of metadata and communication count relative to site/process locations for design 2

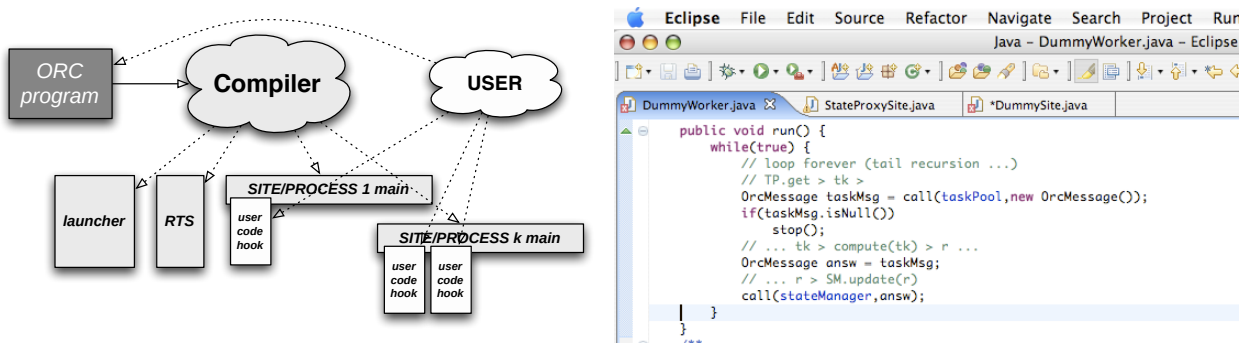

Figure 5: Orc2Java compiler tool (left) and user code implementing a dummy worker process (right)

Now we can compare the two designs with respect to communications. Simple algebraic manipulation allows us to conclude that the second will "perform better", that is will lead to a communication profile costing fewer time units, if and only if

$$k > \frac{rN}{rN - 2l}$$

That is, if and only if the number of local state updates accumulated at the LSM_is before sending update messages to SM is larger than one (typically $l \ll r$). In section 4 we show experimental results that validate this statement.

4 Automatic Distributed Code Framework Generation

To support experimentation with alternative orchestrations derived using the methodology discussed above, we developed a tool for the generation of distributed Java code from an Orc specification (Fig. 5 (left). Being a compiler producing actual Java distributed code, the tool is fundamentally different from the Orc simulator provided by the Orc designers on the Orc web page[8].

In particular, our fully automatic tool takes as input an Orc program (that is a set of Orc expression definitions plus a target expression to be evaluated) and produces a set of Java code files that completely implement 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 produced to implement the interactions among Orc parallel/distributed entities, and so on. Orc sites and processes are implemented by distinct JVMs running on either the same or on different machines and communications are implemented using plain TCP/IP sockets. A "script" program is also produced that takes as input a description of the target architecture (names of the nodes available, features of each node, interconnection framework information, and so on) and deploys the appropriate Java code to the target nodes and finally starts the execution of the resulting distributed application.

The Java code produced provides hooks to programmers to insert the needed "functional code" (i.e. the code actually implementing the sites for the computation of the application results). The system can be used to perform fast prototyping of grid applications, and the parallel application skeleton is automatically generated without the need to spend time programming all of the cumbersome details related to distributed application development (process decomposition, mapping and scheduling, communication and synchronization implementation, etc.) that usually takes such a long time. Also, the tool can be used to run and compare several different skeletons, such that the users may evaluate "in the field" which is the "best" implementation.

Using the preliminary version of our Java code generator[1], we were able to evaluate the performances of the application described in Section 2. Figure 5 (right) shows the code supplied by the user to implement a dummy worker process. The method `run` of the class is called by the framework when the corresponding process is started. The user has an `OrcMessage call(String sitename, OrcMessage message)` call available to interact with other sites, as well as one-way call mechanisms such as `void send(String sitename, OrcMessage msg)` and `OrcMessage receive()`. If a site is to be implemented, rather than a process, the user must only subclass the `Site` framework class providing an appropriate `OrcMessage react(OrcMessage callMsg)` method.

[1] We are indebted to Antonio Palladino who has been in charge of developing the prototype compiler

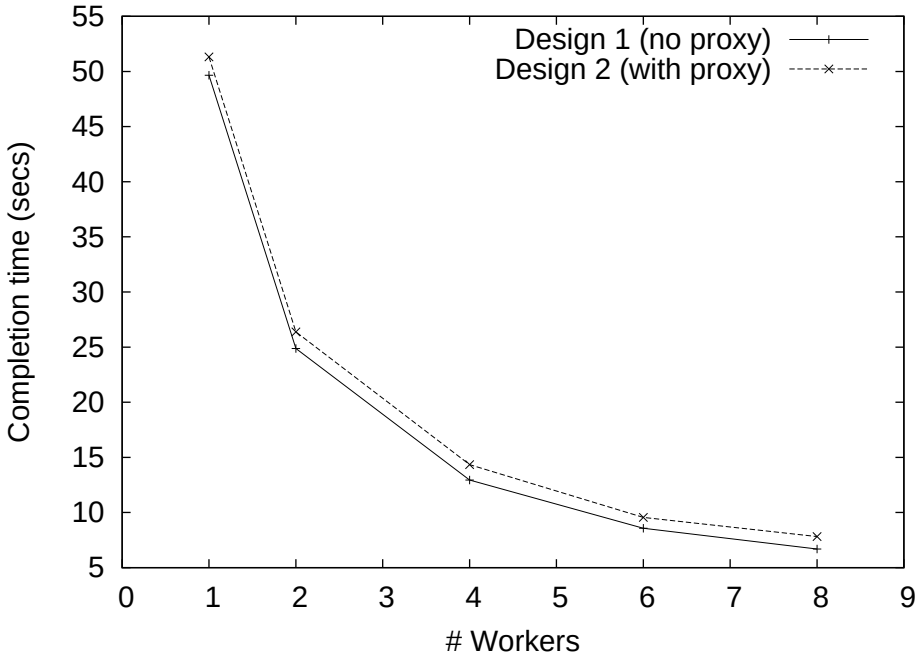

Figure 6: Comparison among alternative versions of the grid application described in Section 3

Fig. 6 shows the results we obtained when running the two versions of the application, automatically derived from the alternative Orc specifications previously discussed. The version labeled "no proxy" corresponds to the version with a single, centralized state manager site which is called by all the workers, while the version labelled "with proxy" corresponds to the version where workers pass results to proxies for temporary storage, and they, in turn, periodically call the global state manager to commit the local updates. The relative placement of processes and sites is determined by appropriate metadata as explained in previous sections. The code has been run on a network of Pentium based Linux workstations interconnected by a Fast Ethernet network and running Java 1.5.0_06-b05.

As expected, the version with proxy performs better than the one without, as part of the overhead deriving from state updates is distributed (and therefore parallelized) among the proxies, while the amount of task computation is unchanged and the state update times are negligible both w.r.t. task computation time and w.r.t. communication times.

This is consistent with what we have concluded in Sec. 2. On the target architecture considered, we measured

$$r = 653\mu secs$$

and

$$l = 35\mu secs$$

and therefore the proxy implementation is convenient when k is larger than 1.12 to 1.01 depending on the number of workers considered (1 to 8, in this case). In this experiment, LSM_i performed an average of 1.9 local updates before calling the SM site to perform a global state update and therefore the constraint above was satisfied.

5 Conclusions

Taking our earlier work as the starting point, we have shown here how alternative designs of a distributed application (schema) can be assessed by describing them in a formal notation and associating with this specification appropriate metadata to characterise the non-functional aspect of interest, in this case, communication cost. We derived cost estimations that show that the second implementation, the one including the proxy design pattern, should perform

better than the first one, in most cases. Then, using our prototype Orc compiler we "fast prototyped" two versions of the distributed application and ran these versions on a distributed set of Linux workstations. The times measured confirmed the predictions. Overall the whole procedure demonstrates that

- under the assumptions made here, we can, to a certain degree, evaluate alternative implementations of the same application using metadata-augmented specifications only, and in particular, without writing a single line of code;
- and that the Orc to Java compiler can be used to generate rapid prototypes that can be used to evaluate applications directly on the target architecture.

Future work will involve (semi-)automating the derivation of metadata from user-specified input (currently a manual process) and investigating the use of the framework with a wider range of skeletons[9]: experience suggests that skeletons potentially provide a restriction from the general that may prove to be fertile ground for our approach.

References

[1] Next Generation GRIDs Expert Group. *NGG2, Requirements and Options for European Grids Research 2005-2010 and Beyond*, July 2004.

[2] Next Generation GRIDs Expert Group. *NGG3, Future for European Grids: GRIDs and Service Oriented Knowledge Utilities. Vision and Research Directions 2010 and Beyond*, January 2006.

[3] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, Cambridge, 1999.

[4] 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.

[5] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. *CONCUR*, LNCS No. 4137, pages 477–491. Springer, 2006.

[6] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Adding metadata to Orc to support reasoning about grid programs. *Proceedings of the CoreGRID Symposium 2007*, pp. 205–214, Rennes (F), August 2007. Springer. ISBN: 978-0-387-72497-3

[7] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Management in distributed systems: a semi-formal approach. *Proc. of Euro-Par 2007 - Parallel Processing 13th Intl. Euro-Par Conference*, LNCS No. 4641, Rennes (F), August 2007. Springer.

[8] William R. Cook and Jayadev Misra. Orc web page, 2007. `http://www.cs.utexas.edu/users/wcook/projects/orc/`.

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