John von Neumann Institute for Computing

A Framework for Prototyping and Reasoning about Distributed Systems

Marco Aldinucci, Marco Danelutto, Peter Kilpatrick

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 235-242, 2007.

© 2007 by John von Neumann Institute for Computing
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

http://www.fz-juelich.de/nic-series/volume38

A Framework for Prototyping and Reasoning about Distributed Systems

Marco Aldinucci[1], **Marco Danelutto**[1], **and Peter Kilpatrick**[2]

[1] Dept. Computer Science – University of Pisa – Italy
E-mail: {aldinuc, marcod}@di.unipi.it

[2] Dept. Computer Science – Queen's University Belfast – United Kingdom
E-mail: p.kilpatrick@qub.ac.uk

A framework supporting fast prototyping as well as tuning of distributed applications is presented. The approach is based on the adoption of a formal model that is used to describe the orchestration of distributed applications. The formal model (Orc by Misra and Cook) can be used to support semi-formal reasoning about the applications at hand. The paper describes how the framework can be used to derive and evaluate alternative orchestrations of a well know parallel/distributed computation pattern; and shows how the same formal model can 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 inappropriate.

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 as being 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, an approach is required 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.

This research 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*

Earlier work by the authors identified Orc by Misra and Cook[4,5] as a suitable notation to act as a basis for the framework proposed above. There it was shown how Orc is particularly suitable for addressing *dynamic* properties of distributed systems, although in the current work, for the sake of simplicity, only static examples/properties are considered. Subsequent work demonstrated how an Orc-based framework can be developed to support large distributed application development and tuning[6,8]. Figure 1 outlines the main features of Orc, in particular those related to the Orc specifications used here. In the current work these previous results are extended by two further contributions.

First it is shown that a cost analysis technique can be described and used to evaluate properties of alternative implementations of the same distributed application. Orc models of two (functionally equivalent) implementations of a typical distributed use-case are presented. For each, the number and kind of communications performed is determined and the overall communication cost is estimated using a measure of communication cost. To evaluate the communication patterns of the two implementations metadata denoting *locations* where parallel activities have to be performed is considered. The metadata is formulated in accordance with the principles stated in earlier work presented at the CoreGRID Symposium[6]. While metadata can be used to describe several distinct aspects of a system, here only metadata items of the form $\langle site/expression, location\rangle$ representing the fact that $site/expression$ is run on the resource $location$ are considered. 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 occurring during the application execution.

Second a compiler tool that allows generation of the skeleton of a distributed application directly from the corresponding Orc specification is described. 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, sched-

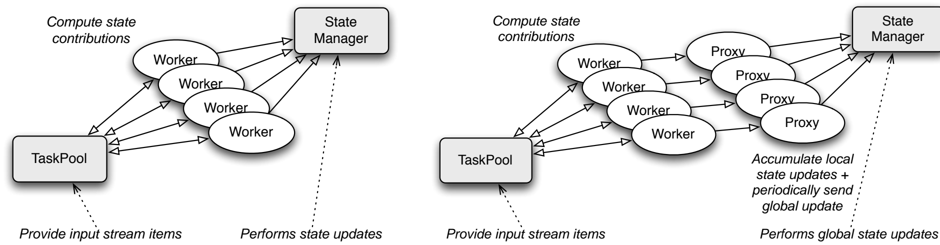

Figure 2. Logical schema corresponding to the two Orc specifications (grey boxes represent sites, white ovals represent processes)

ule, etc. are completely dealt with by the compiler generated code. Finally, results are presented which demonstrate that the code generated from the Orc specifications given here perform as expected when executed on a distributed target architecture.

These two further developments complete a framework supporting design, development and tuning of distributed grid applications.

2 Use Case: Parallel Computation of Global State Updates

A use case is now introduced as a vehicle to demonstrate 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. A common parallel application pattern is used for illustration: data items in an input stream are processed independently, with the results being used to update a shared state (see Fig. 2). For simplicity, only the easy case where state updates are performed through an associative and commutative function $\sigma : State \times Result \rightarrow State$ is considered. 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 "summed" onto the final scene. Adding effects of a ray on the scene can be performed in any order without affecting the final result. Two alternative designs of this particular parallelism exploitation pattern, parametric in the type of actual computation performed, are modelled 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 (modelling the input stream), *TM*, a state manager, *SM* and a set of workers, W_i. The workers repeatedly take 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

A procedure is now introduced to determine the cost of an Orc expression evaluation in terms of the communications performed to complete the computation modelled by the expression.

First some basic assumptions concerning communications are made. It is assumed that a site call constitutes 2 communications (one for the call, one for getting the ACK/result): no distinction is made between transfer of "real" data and the ACK. It is also assumed that 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, this exchange is not considered to constitute 4 communications.) Two cases for sequential composition with passing of a value may be identified: *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, care must be taken in treating communications that may overlap in a parallel computation. A simple and effective model is assumed: for Orc parallel commands the communication cost mechanism should take into account the fact that communications happening "internally" to the parallel activities overlap while those involving *shared* external sites (or processes) do not. Thus, when counting the communications occurring within an Orc expression such as:

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

a distinction is drawn between calls to site *TP* which are calls to external, shared sites, and calls to wc_i which are related to internal sites. When assessing the calls related to the execution of N processes W_i computing M tasks, all the calls to TP are counted while it is assumed that the communications related to different wc_i are overlapped. Therefore, assuming perfect load balancing, the total will include 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 two steps are performed.

First the metadata associated with the Orc specification is determined. It is assumed that 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 design given above, with initial metadata $\mathcal{M} = \{\langle site,\ TP\rangle,\ \langle site,\ SM\rangle,\ \langle system,\ strategy(FullyDistributed)\rangle\}$, the metadata

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

can be derived, 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 stated (by explicit placement). This metadata is *ground* as all sites and all non-terminals have associated locations. For details of metadata derivation see the CoreGRID technical report[7].

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, assume N worker processes, computing M tasks, with two sites providing the taskpool 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 taskpool 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 are costed at r time units. (Had they involved sites/processes on the same processing resources the cost would have been 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, there are N W_i, N $ctrlprocess_i$ processes and N LSM_i sites, plus the globally shared TP and SM sites. 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 the CoreGRID technical report[7] for details of the metadata determination.) 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 $2Mr + \frac{2Mr}{k} + \frac{4Ml}{N}$.

Now the two designs can be compared with respect to communications. Simple algebraic manipulation shows 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

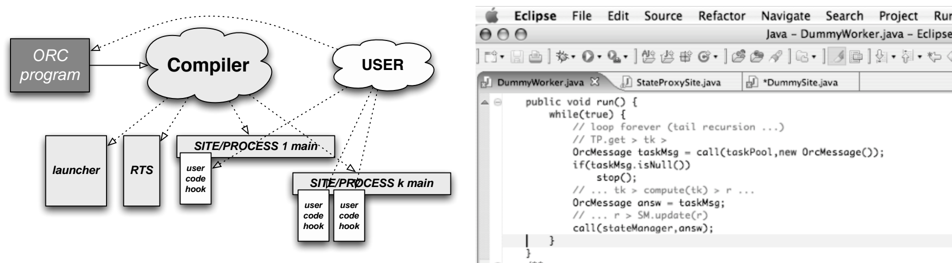

Figure 3. Orc2Java compiler tool (left) and user code implementing a dummy worker process (right)

messages to SM is larger than one (typically $l \ll r$). In Section 4 experimental results that validate this statement are presented.

4 Automatic Distributed Code Framework Generation

To support experimentation with alternative orchestrations derived using the methodology discussed above a tool for the generation of distributed Java code from an Orc specification was developed (Fig. 3 (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[9].

In particular, the 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 users may evaluate "in the field" which is the "best" implementation.

Using the preliminary version of the Java code generator[a], the performances of

[a]The authors are indebted to Antonio Palladino who has been in charge of developing the prototype compiler

the alternative designs of the application described in Section 2 were evaluated. Figure 3 (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.

Fig. 4 shows the results obtained when running several versions of the application, automatically derived from the alternative Orc specifications previously discussed. The version labelled "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 versions with proxy perform better than the one without, as part of the overhead deriving from state updates is distributed (and therefore parallelized) among the proxies.

This is consistent with what was predicted in Section 3. On the target architecture considered, values of $r = 653\mu secs$ and $l = 35\mu secs$ were obtained and so the proxy implementation is to be preferred when k is larger than 1.12 to 1.02 depending on the number of workers considered (1 to 4, in this case). In this experiment, LSM_i performed an average of 2.5 local updates before calling the SM site to perform a global state update and therefore the constraint above was satisfied.

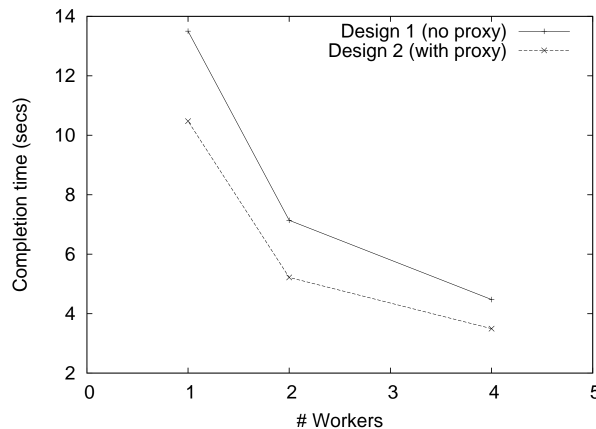

Figure 4. Comparison among alternative versions of the grid application described in Section 3

5 Conclusions

The work described here shows 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. Cost estimations were derived 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 the prototype Orc compiler two versions of the distributed application were "fast prototyped" and run on a distributed set of Linux workstations. The times measured confirmed the predictions. Overall the whole procedure demonstrates that 1) under the assumptions made here, one 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 2) 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 1) (semi-)automating the derivation of metadata from user-specified input (currently a manual process) and 2) investigating the use of the framework with a wider range of skeletons[10]: experience suggests that skeletons potentially provide a restriction from the general that may prove to be fertile ground for the approach.

References

1. Next Generation GRIDs Expert Group, *NGG2, Requirements and options for European grids research 2005–2010 and beyond*, (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*, (2006).
3. R. Milner, *Communicating and Mobile Systems: the Pi-Calculus*, (Cambridge University Press, 1999).
4. J. Misra and W. R. Cook, *Computation orchestration: a basis for wide-area computing*, Software and Systems Modeling, DOI 10.1007/s10270-006-0012-1, (2006).
5. D. Kitchin, W. R. Cook and J. Misra, *A language for task orchestration and its semantic properties*, CONCUR, in LNCS **4137**, pp. 477–491. (Springer, 2006).
6. M. Aldinucci, M. Danelutto and P. Kilpatrick, *Adding metadata to Orc to support reasoning about grid programs*, in: *Proc. CoreGRID Symposium 2007*, pp. 205–214, Rennes (F), Springer, 2007. ISBN: 978-0-387-72497-3
7. M. Aldinucci, M. Danelutto and P. Kilpatrick, *Prototyping and reasoning about distributed systems: an Orc based framework*, CoreGRID TR-0102, available at `www.coregrid.net`, (2007).
8. M. Aldinucci, M. Danelutto and P. Kilpatrick, *Management in distributed systems: a semi-formal approach*, in: Proc. Euro-Par 2007 – Parallel Processing 13th Intl. Euro-Par Conference, LNCS **4641**, Rennes (F), Springer, 2007.
9. W. R. Cook and J. Misra, *Orc web page*, (2007). `http://www.cs.utexas.edu/users/wcook/projects/orc/`.
10. M. Cole, *Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming*, Parallel Computing, **30**, 389–406, (2004).