Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 1 (2012) 2095–2103

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

Map, Reduce and MapReduce, the skeleton way[☆]

D. Buono[a], M. Danelutto[a], S. Lametti[a]

[a]*Dept. of Computer Science, Univ. of Pisa, Italy*

Abstract

Composition of Map and Reduce algorithmic skeletons have been widely studied at the end of the last century and it has demonstrated effective on a wide class of problems. We recall the theoretical results motivating the introduction of these skeletons, then we discuss an experiment implementing three algorithmic skeletons, a *map*, a *reduce* and an optimized composition of a *map* followed by a *reduce* skeleton (map+reduce). The map+reduce skeleton implemented computes the same kind of problems computed by Google MapReduce, but the data flow through the skeleton is streamed rather than relying on already distributed (and possibly quite large) data items. We discuss the implementation of the three skeletons on top of ProActive/GCM in the MareMare prototype and we present some experimental obtained on a COTS cluster.

Keywords:
Algorithmic skeletons, map, reduce, list theory, homomorphisms

1. Introduction

Algorithmic skeletons have been around since Murray Cole's PhD thesis [1]. An algorithmic skeleton is a parametric, reusable, efficient and composable programming construct that exploits a well know, efficient parallelism exploitation pattern. A skeleton programmer may build parallel applications simply picking up a (composition of) skeleton from the skeleton library available, providing the required parameters and running some kind of compiler+run time library tool specific of the target architecture at hand [2].

Several skeleton frameworks have been developed in the past 20 years. Among the others, we mention the skeleton libraries eSkel [2], Muesli [3] and SkeTo [4], targeting C/C++ MPI architectures, Muskel [5], Calcium [6] and Skandium [7], targeting Java (RMI POSIX/TCP or ProActive/GCM) architectures, P3L [8], SkIE [9] and ASSIST [10] that instead provide skeletons as constructs of new programming/coordination languages.

Most of the skeleton frameworks developed up to now, including those not cited above, provide the programmer with two kinds of skeletons: *stream parallel* skeletons, exploiting parallelism in the execution of parts of the application on different items belonging to the input task stream and *data parallel* skeletons, exploiting parallelism in the execution of sub-computations relative to (possibly overlapping) partitions of the input task. Stream parallel skeletons include task farm, modelling embarrassingly parallel computations, and pipeline, modelling computations in stages.

[☆]This work has been partially supported by Italian FIRB Project InSyEME and by the ERCIM CoreGRID WG, *Advanced programming model* topic

Email addresses: `d.buono@di.unipi.it` (D. Buono), `marcod@di.unipi.it` (M. Danelutto), `lametti@di.unipi.it` (S. Lametti)

1877-0509 © 2012 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.
doi:10.1016/j.procs.2010.04.234

Data parallel skeletons typically include map, modelling embarrassingly parallel computations, reduce, modelling tree structured "sums" of structured data computed with an associative and commutative operator, prefix, modelling "all value sums" plus a variety of stencil skeletons, modelling maps with data dependencies.

In this paper, we discuss an experiment aimed at implementing a full set of data parallel skeletons on top of ProActive/GCM [11]. The set hosts three skeletons: a map, a reduce and an optimized version of their composition, that is a mapreduce. It also implements advanced features such as fault tolerance and the possibility to dynamically adapt the parallelism degree of an ongoing computation. Sec. 2 introduces the formal background needed to appreciate these algorithmic skeletons. Sec. 3 concisely introduces ProActive/GCM, our target (sw) architecture). Then Sec. 4 and Sec. 5 introduce the MareMare skeleton prototype and its advanced features. Eventually Sec. 6 discusses the experimental results we observed running the prototype on a COTS Linux cluster.

2. Skeleton rewriting and formal properties

A quite consistent research activity took place starting in the '90 to understand whether a skeleton algebra exists that can be used to support program rewriting and optimization. Starting from an approach quite similar to the one followed in Backus's masterpiece [12], skeletons have been modelled as higher order functions. Formal tools have been developed to support the reasoning about equivalent programs and about transformation techniques that can be used to increase performance or some other non functional[1] feature of our skeleton programs. As an example, the following rewriting rule has been demonstrated correct

$$(\alpha\ f) \circ (\alpha\ g) \equiv \alpha\ (f\ \circ\ g) \tag{1}$$

were map (α) is defined as

$$\alpha\ f\ \langle x_1, \ldots, x_n \rangle = \langle f(x_1), \ldots, f(x_n) \rangle$$

and reduce (/) as

$$/ \oplus\ \langle x_1, \ldots, x_n \rangle = x_1 \oplus x_2 \oplus \ldots \oplus x_n$$

The meaning of rule (1), in the skeleton perspective, can be stated as follows:

> if you have to compute a composition of maps, you can compute instead the map of the compositions of the two map "function" parameters.

This seems a naive result, but in many skeleton frameworks the implementation of a map ends with a barrier to await all the sub-results needed to build the final results. In case of load unbalancing, the barrier may impair the composition of map performance, while the map of composition (with its single barrier at the end) may spot an almost ideal efficiency. In a sense, a rule such as rule (1) may be used either to drive efficient implementation of skeleton compositions in the skeleton framework or to optimize *on-the-fly* the user defined skeleton program by rewriting the user supplied skeleton composition to an equivalent, more efficient one, right before actually compiling the skeleton program.

In the '80, some more theoretical results have been developed, supporting data parallel skeleton design and implementation. In particular, it has been shown that functions operating on data structures can be conveniently expressed as homomorphisms after the data structure *constructive* definition. Bird introduced the concept using lists [13]. A list is constructively defined as

$$\alpha\ \mathsf{list} = \mathsf{emptyList} \mid \alpha :: \alpha\ \mathsf{list}$$

Then list homomorphisms are those functions inductively defined as

$$\mathsf{h(emptyList) = k}$$

$$\mathsf{h(x::l) = f(x)} \oplus \mathsf{h(l)}$$

[1]that is not related to *what* the program computes but rather to *how* the program computes the final result.

where k is a constant and $\oplus$ and f are proper binary and unary functions. List homomorphism have nice compositional properties, can be efficiently implemented in parallel and it can be shown that the useful functions on lists can be expressed as homomorphism [13]. These properties also hold in case of homomorphisms defined on different data structures, such as trees or arrays [14]. Map and reduce can be defined themselves as homomorphisms. $\alpha\ g$ is the homomorphism with where $f = g$ and $\oplus =::$. $/\ominus$ is the homomorphism where $f = identity$ and $\oplus = \ominus$. The nice result here is that a list homomorphism can itself be defined using map and reduce:

$$hom(f, \oplus) = (/\oplus) \circ (\alpha\ f)$$

In turn, this result can be interpreted as "*useful functions on data structures may be defined as mapreduce*". These results clearly indicate that homomorphisms may conveniently be used to define data parallel skeletons and that map+reduce is a first order citizen in the algorithmic skeletons scenario. Google mapreduce[2] has not been proposed as an algorithmic skeleton, although it is *de-facto* built on these results from the algorithmic skeleton research. Not being defined in a skeleton framework, Google-like mapreduce operate on data which is partially already distributed on the collection of processing elements used to compute the mapreduce itself. Also, Google-like map reduce usually operates on data stored in files, while algorithmic skeletons usually operate on data items passed through data flow streams and stored in main memory.

3. ProActive

ProActive (`http://proactive.inria.fr/`) is a Java based, open source framework supporting parallel and distributed computing on a variety of target architectures, including multicores, cluster/networks of workstations and grids. ProActive provides a reference implementation of the Grid Component Model (GCM) [17, 18] as an extension of the Fractal component model [19]. In particular, GCM adds to Fractal the concept of *collective ports* to support data parallel computations. GCM includes two kind of collective ports, namely the *multicast* and the *gathercast* ports [20]. Multicast ports support customizable one-to-many multicast and broadcast communication patterns, while gathercast ports support many-to-one collection communications[3].

The support to GCM components, along with their collective ports, and the fact ProActive/GCM runs on a variety of distributed system including POSIX/TCP workstation networks and distributed platforms running different middleware such as Globus, gLite, made it an ideal candidate to experiment our prototype skeleton framework.

4. MareMare: A component-based Map-Reduce framework

MareMare is a skeleton framework written in Java that allows programmers to write and develop data parallel applications using Map, Reduce and optimized compositions of the two skeletons. MareMare is provided as a library and its parallel and distributed features are implemented exploiting ProActive/GCM. The idea is to provide single Map and Reduce skeletons to be used alone or in composition with the other skeletons that will possibly be available in the future, and to provide the optimized implementation of a Map followed by a Reduce which turns out to be a very useful skeleton composition pattern as outlined in Sec. 2.

MareMare provides a couple of advanced features, not usually included in other data parallel skeleton frameworks. MareMare is able to efficiently handle faults at the worker component nodes and may be asked to reconfigure the ongoing computation in such a way more (less) computing resources are used depending on the dynamic performance requirements of the application. Both featured are discussed in Sec. 5.

4.1. Map-Reduce applications in ProActive

MareMare provides a factory that is responsible for creation, activation and deployment of the various components needed to run the application. The users only have to implement the function parameters of the *Map* and *Reduce* skeletons as well as the code used to deliver the results. In particular, the user have i) to extend the Drain, Function

[2]as well as the follow-up Hadoop[15] framework and the other mapreduce implemented after Google first paper [16]

[3]`http://docs.huihoo.com/proactive/3.9.0/ComponentsTutorial.html`

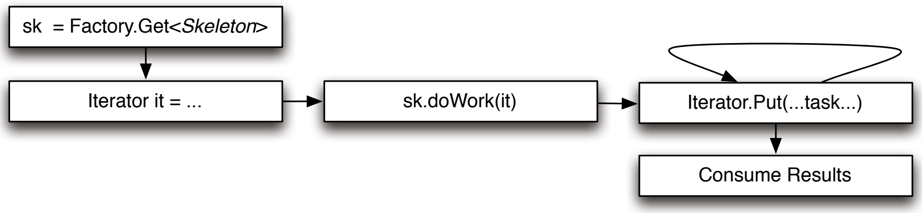


Figure 1: Typical structure of a MareMare application

and Reduce interfaces to express the sequential code of the program and ii) to specify the parallelism degree and the ProActive file descriptor describing all the available nodes. Giving those parameters to our factory the entire parallel application is deployed. Stream behaviour is emulated by providing data using a particular iterator. Furthermore, the end of the stream is implemented by closing the iterator. A specific *hasTerminated* method allows programmer to wait for the end of the entire computation. The completed tasks are sent to the Drain object, that is in charge of consuming the results. Those basic steps are depicted in figure 1. Moreover stream elements must be encapsulated into a container, that is an object of type *Bag*, in order to efficiently scatter data to the workers.

MareMare provides a simple way to describe the application deployment by means of a class *MareMareDeploymentDescriptor* which automatically generates a valid ProActive deployment file from a simple XML file. The user only needs to list in the file all available nodes and provide the username and IP address of remote hosts to allow a `ssh` connection. Furthermore, the classes provided by the user can be used to perform a sequential simulation to verify the correctness of the results obtained from parallel and distributed version.

4.2. MareMare *architectural design*

A generic MareMare program consists of three main components: i) the *input component*, which receives input data and distributes data to executors, ii) the *worker component*, which actually performs the computations, iii) and the *output component*, which is in charge of the final part of the computation and of the delivering of the results to the user. In addition, MareMare implements a dispatcher object that is used to connect user code to the relevant MareMare components. Multiple instances of the worker component are used to obtain the required parallelism degree and each one of these instances is allocated on a different physical node. The rest of the MareMare components are allocated on a single node–usually the user machine–that represents the application front-end.

The Input component is responsible for the reception of data from the input stream using the class *"MareMareIterator"*. For each data item it distributes tasks to worker components using a *multicast* interface (GCM) [17, 18] to scatter the input data. This interface divides each *"Bag"*, that is a single task, into a number of "sub-Bag" equal to the number of available workers and distribute the subtasks to them. It should be noted that GCM offers an automated way to distribute a set of tasks to a certain number of workers; however we implemented an *ad hoc* mechanism that achieves a better performance and provides fault-tolerance and reconfiguration mechanisms as well.

After dispatching tasks to the Worker components, the Input component starts waiting the results from Worker components. Results come back as *future* objects and the Input component delivers them to the Output component.

In order to implement the various types of skeletons provided, a worker component is composed of several subcomponents according to the form of parallelism that is going to be implemented (Map, Reduce and Map+Reduce). For each task received from the Input component, a Worker component uses the Function and Reduce subcomponents in order to apply the Map function and the first step of the Reduce function to all the items of the (sub-)Bag received.

Output component receives the results from workers expressed as futures. In particular, the component waits the end of worker execution and, having received all results of the same task, applies the second step of the Reduce function, that is the same applied by the worker to local results relative to the computation of a single partition. At this point, the Output component delivers the actual MapReduce results using the Drain function provided by the user.

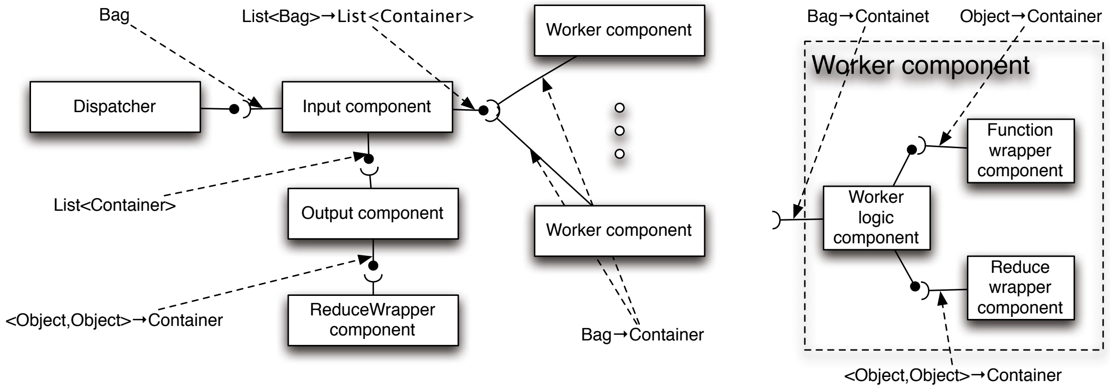


Figure 2: MareMare architecture

5. Implementing advanced features with ProActive

Capitalizing on our previous experience with skeleton programming environments, we extended the base Mare-Mare library with a set of advanced features, aimed at better supporting execution of programs on highly distributed and dynamic environments such as clusters and grids. In particular we allow the programmer to dynamically change the number of processing nodes used during the computation and we transparently support the failure of one or more nodes, exploiting, in both case, the knowledge deriving from the skeletons used. The results discussed here relative to Map and Reduce skeletons have been already proved to be generalizable to a wider set of skeletons and even to skeleton compositions [21, 22, 23].

Run-time reconfigurations are generally useful with problems with unpredictable performance (as an example in case of irregular data parallel algorithms or in case highly dynamic and possibly heterogeneous target architectures are used) or with varying user provided performance requirements (as an example, in case the user requires/accepts faster/slower execution times). In both cases a monitoring infrastructure is needed that exposes the current performance of the entire application and, more important, of its single parts. We therefore implemented a monitoring layer inside the MareMare library, that can be used by the application programmer in its code to continuously check the performance of the entire applications or its single constituent parts, in such a way it can require a run-time reconfiguration of the application, if needed.

5.1. Supporting run-time reconfigurations

One of the most common reconfiguration considered in parallel programming frameworks is the variation of the parallelism degree, i.e. the variation of the number of processing elements cooperating to calculate the results. At the moment being, this is the only the run-time reconfiguration supported in MareMare.

While adding (or removing) a process seems an easy task, a simple implementation could have unexpected behaviour in a stream-based library. In fact the biggest problem is the handling of previously scheduled but not yet executed tasks. For performance reasons, usually tasks are sent to the processing nodes (Worker components) as soon as possible. This has an important effect on both the increase and the decrease of Worker components. When *removing a node*, if the Worker component we are going to remove has a queue of task to be executed, those tasks should be rescheduled to other Worker components. On the other hand, when *adding a node* the tasks previously sent to the map, which have already been scheduled for a lower number of nodes, cannot use the added nodes. This means that the new nodes can be used much later than when they are actually added to the execution framework and, in some cases (i.e. in case the tasks are completely scheduled to the Worker components when the application is started) they eventually will be not used at all. In both cases the behaviour of the reconfiguration is somehow "delayed". Therefore, we decided to minimize this effect by re-scheduling all those tasks that are in the workers queues when a new Worker

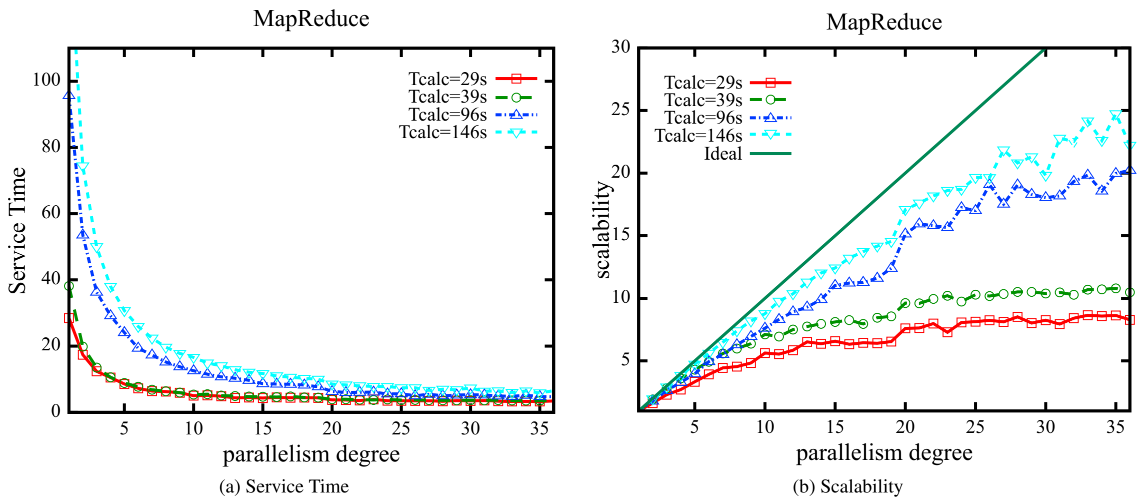


Figure 3: MapReduce

component is added to the computation. Following this approach, the new parallelism degree is immediately and fully exploited. This more reactive behaviour obviously causes a slightly higher network traffic.

Run-time reconfigurations are not a new in parallel programming. However, with more traditional parallel programming frameworks (e.g. OpenMP, MPI and even with ProActive) those reconfigurations must be entirely handled (and implemented) by the application programmer.The MareMare reconfiguration support, instead, provides a simple and user friendly API to issue requests of addition or removal of worker nodes. Despite the fact several works demonstrated it is possible to create autonomic components that will automatically add or remove nodes in order to respect a given QoS contract [22, 23], MareMare programmers must explicitly invoke these APIs to implement parallelism degree adaptation.

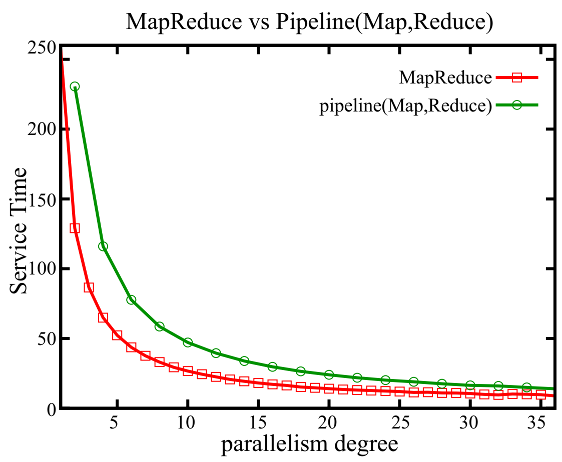
5.2. *Fault Tolerance*

Another important problem that must be taken into account in large distributed environments is the possibility of node failures. Current version of MareMare transparently handles failures of the Worker component nodes, and we are working to support failures also on Input and Output nodes.

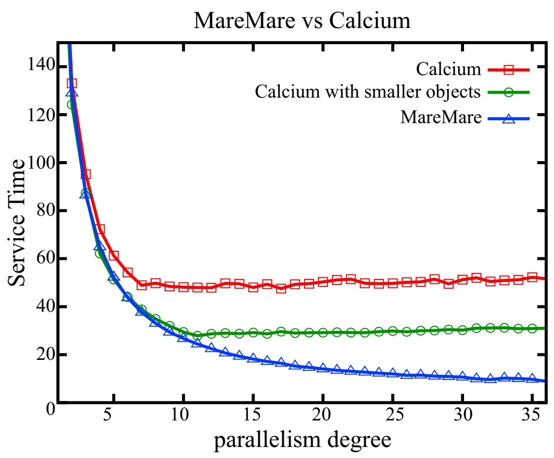
Fault management requires both failure detection and failure handling. MareMare handles failure detection using a proper heartbeat mechanism, implemented in a specific thread of the Input component that receives heartbeat signals from the workers and constantly looks for non responsive nodes. Failure handling is much more complex. When a fault occurs, we need to change the application structure, removing the components deployed on faulty nodes. After this, the application can continue processing tasks, but we must reschedule the tasks that were sent to the faulty node.

In the data-parallel computations we are considering, each task is divided into subtasks that are then scattered to the Worker components. When a fault occurs, we allow non faulty Worker components to deliver their sub-task results to the Output component. The Output component accumulates these partial results and waits for the missing items. The Input component, in the meanwhile, manages to re-schedule the tasks already scheduled to the faulty Worker components to other Worker components, recruited “on-the-fly” to substitute the faulty ones. When eventually the missing result items are delivered to the Output component, it will merge them to the other partial results previously cached and then it will send the complete task result to the Drain object.

This fault management algorithm strictly relies on the fact is is applied to data-parallel skeletons, of course. It gives better (and predictable) performance with respect to standard checkpointing techniques such as communication-induced checkpointing. The approach is also completely different from standard checkpoint based approach. In that case, when a fault is detected, the whole computation are restarted from the last “safe” checkpoint. In our case, we try



(a) pipeline



(b) Calcium

Figure 4: Synthetic application benchmark comparisons

to preserve as much as possible the results already computed by the non faulty nodes and we manage to restart only the "computations" (i.e. the subtasks) allocated onto the faulty nodes.

6. Experimental results

In order to assess MareMare design and implementation, we run some experiments on a 20 nodes Fast Ethernet Core 2 duo E7400 cluster, operated with Debian Linux, kernel 2.6.26. We used synthetic application benchmarks, in such a way we have been able to analyze both service time and scalability with different parallelism degrees and computational grains. These tests clearly exposed the impact of communication in a Map-Reduce parallelization: if the computation grain is too fine, when a large amount of workers is used the service time is dominated by the communication latencies, and adding more nodes does not improve the efficiency any more. When coarser and coarser grain tasks are computed, the efficiency gets closer and closer to the ideal one, as expected.

Fig. 3 shows typical service time and scalability measured. In these cases, an synthetic application using a single map+reduce skeletons was used (T_{calc} represents the number of seconds spent computing a single task).

We compared the service time achieved by map+reduce skeleton with respect to the one achieved using a composition of a map and a reduce skeleton: map+reduce obviously performed better, due to the lack of synchronization in between the two stages and on the fact that part of the reduction in map+reduce is actually executed on the nodes that computed the partition of the map result (Fig. 4a). The differences become less evident when adding more nodes, however, due to the impact of the network latencies.

We compared our library with Calcium [6, 24], the skeleton framework embedded in ProActive. This is the only skeleton framework running on top of ProActive, as far as we know, and supports both map and (a limited form of) reduce skeletons. We implemented the same map+reduce application using the map skeleton of Calcium. The service times measured using Calcium and MareMare are very close when a few processing elements are used. When parallelism is increased, MareMare starts performing much better than Calcium (Fig. 4b). This is mainly due to two distinct factors: i) the map skeleton in Calcium does not offer an efficient way to express the reduce phase, that is therefore executed sequentially; ii) it seems that the network latency and bandwith have a greater impact in Calcium. In particular, to assess point ii) above, we modified the task size while mantaining the same computation time, and we got results which are actually close to those obtained with MareMare. The small differences still present are due to the sequential reduce step.

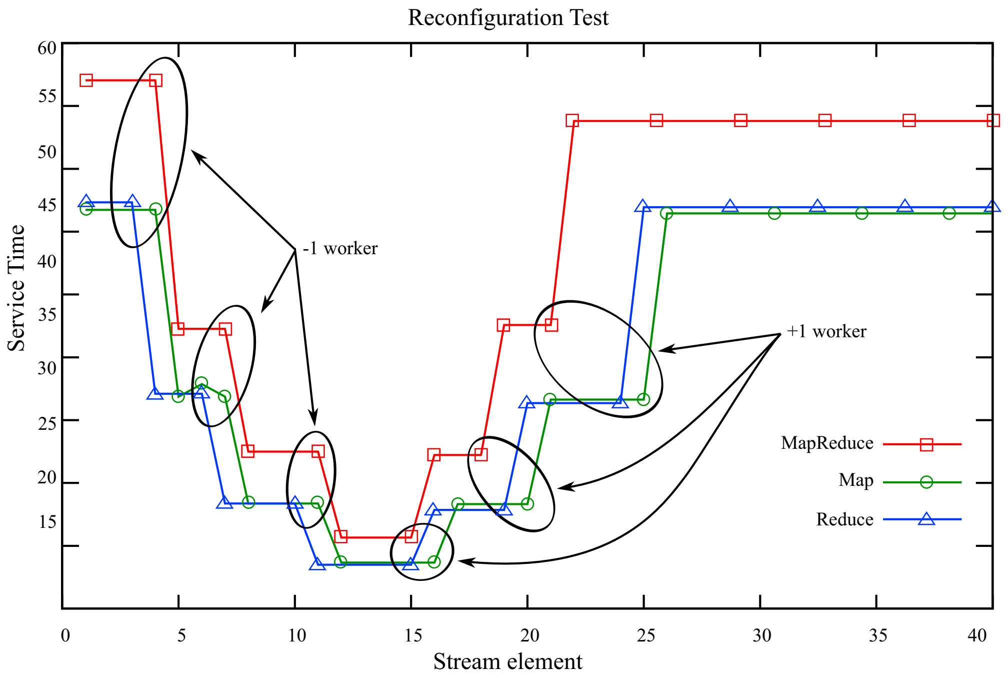


Figure 5: Reconfiguration behaviour of MareMare

Last but not least, we evaluated the behaviour of MareMare applications in case of reconfigurations. We modified our synthetic applications in such a way several reconfigurations where required at arbitrary points of the computation (an approach similar to the one adopted in GCM behavioural skeletons could be adopted to automatically trigger adaptations [23], but we used "manual" triggers to better evaluated effects on MareMare overall performance). Fig. 5 shows the typical behaviour of service time when a Map, a Reduce and a MapReduceare considered. By measuring the completion time of the applications with and without reconfigurations, we were able to conclude that allocation of new nodes is (almost completely) performed in parallel with the computation, and therefore the reconfiguration itself does not affect too much the overall execution time. This is a very nice results, also taking into account the limited amount of effort required to application programmers to trigger the reconfigurations.

7. Conclusions

MareMare is a skeleton library built on top of ProActive/GCM. It provides programmers with Map and Reduce algorithmic skeletons, as well as with an optimized composition of the two, i.e. a map+reduce skeleton. MareMare supports full and transparent handling of faults at worker nodes and it supports monitoring and reconfiguration of skeleton applications as well. The effort required to the application programmer to trigger reconfiguration once the monitoring values demonstrated reconfiguration necessity is modest. In particular, the *implementation* of the reconfiguration step (adding or removing resources) is completely in charge of MareMare. Experimental results validate the MareMare design and implementation.

These features, along with the possibility to run MareMare on a variety of target architectures due to ProActive/GCM portability, make MareMare an interesting instance of "skeleton framework". We are currently working to include autonomic adaptation mechanisms in MareMare to relieve application programmers also of the small duty of triggering adaptations, following the approach adopted in GCM Beheavioural skeletons.

References

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

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

[3] H. Kuchen, A skeleton library, in: B. Monien, R. Feldman (Eds.), Proc. of 8th Intl. Euro-Par 2002 Parallel Processing, Vol. 2400 of Lecture Notes in Computer Science, Springer-Verlag, Paderborn, Germany, 2002, pp. 620–629.

[4] SkeTo Home Page, `http://www.ipl.t.u-tokyo.ac.jp/sketo/` (2009).

[5] M. Aldinucci, M. Danelutto, P. Dazzi, Muskel: an expandable skeleton environment, Scalable Computing: Practice and Experience 8 (4) (2007) 325–341.
URL `http://www.di.unipi.it/~aldinuc/papers/2007_SCPE_muskel.pdf`

[6] D. Caromel, M. Leyton, Fine tuning algorithmic skeletons, in: 13th International Euro-par Conference: Parallel Processing, Vol. 4641 of Lecture Notes in Computer Science, Springer-Verlag, 2007, pp. 72–81.

[7] M. Leyton, Skandium Home Page, `http://skandium.niclabs.cl/` (2009).

[8] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, M. Vanneschi, P3L: A Structured High level programming language and its structured support, Concurrency Practice and Experience 7 (3) (1995) 225–255.

[9] B. Bacci, M. Danelutto, S. Pelagatti, M. Vanneschi, SkIE: a heterogeneous environment for HPC applications, Parallel Computing 25 (1999) 1827–1852.

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

[11] ProActive home page, `http://www-sop.inria.fr/oasis/proactive/` (2010).

[12] J. Backus, Can Programming Be Liberated from the von Neumann Style? A Functional Programming Style and its Algebra of Programs, Commun. ACM 21 (8).

[13] R. S. Bird, An introduction to the Theory of Lists, in: M. Broy (Ed.), Logic of programming and calculi of discrete design, NATO ASI Series, 1987.

[14] K. Emoto, K. Matsuzaki, Z. Hu, M. Takeichi, Surrounding theorem: Developing parallel programs for matrix-convolutions, in: W. E. Nagel, W. V. Walter, W. Lehner (Eds.), Euro-Par, Vol. 4128 of Lecture Notes in Computer Science, Springer, 2006, pp. 605–614.

[15] HADOOP Home Page, `http://hadoop.apache.org/` (2010).

[16] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, in: Usenix OSDI ’04, 2004, pp. 137–150.

[17] CoreGRID NoE deliverable series, Institute on Programming Model, Deliverable D.PM.02 – Proposals for a Grid Component Model, `http://www.coregrid.net`. (Nov. 2005).
URL `http://backus.di.unipi.it/~marcod/wiki/doku.php?id=wp3deliverables`

[18] CoreGRID NoE deliverable series, Institute on Programming Model, Deliverable D.PM.04 – Basic Features of the Grid Component Model (assessed), `http://www.coregrid.net`. (Feb. 2007).
URL `http://backus.di.unipi.it/~marcod/wiki/doku.php?id=wp3deliverables`

[19] Fractal Home Page, `http://fractal.ow2.org/` (2010).

[20] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, C. Peréz, GCM: a grid extension to Fractal for autonomous distributed components, Annals of Telecommunications 64 (1–2).

[21] C. Bertolli, Fault tolerance for high-performance applications using structured parallelism models, Ph.D. thesis, Università degli Studi di Pisa, Dipartimento di Informatica (2008).

[22] M. Vanneschi, L. Veraldi, Dynamicity in distributed applications: issues, problems and the ASSIST approach, Parallel Computing 33 (12) (2007) 822–845. doi:10.1016/j.parco.2007.08.001.
URL `http://dx.doi.org/10.1016/j.parco.2007.08.001`

[23] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Dazzi, D. Laforenza, N. Tonellotto, P. Kilpatrick, Behavioural skeletons in GCM: autonomic management of grid components, in: D. E. Baz, J. Bourgeois, F. Spies (Eds.), Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing, IEEE, Toulouse, France, 2008, pp. 54–63. doi:http://dx.doi.org/10.1109/PDP.2008.46.

[24] M. Leyton, ProActive/Calcium manual, `http://docs.huihoo.com/proactive/3.2.1/Calcium.html` (2008).