Parallel Processing Letters,

OPTIMIZATION TECHNIQUES FOR IMPLEMENTING PARALLEL SKELETONS IN GRID ENVIRONMENTS

MARCO ALDINUCCI
Italian National Research Council (ISTI–CNR) – Via Moruzzi 1, I–56124 Pisa, Italy,

MARCO DANELUTTO
Dept. of Computer Science – University of Pisa – Viale Buonarroti 2, Pisa, Italy,

JAN DÜNNWEBER
Dept. of Computer Science – University of Münster – Einsteinstr. 62, Münster, Germany

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

Skeletons are common patterns of parallelism like, e.g., farm and pipeline that can be abstracted and offered to the application programmer as programming primitives. We describe the use and implementation of skeletons in a distributed grid environment, with the Java-based system Lithium as our reference implementation. Our main contribution are optimization techniques based on an asynchronous, optimized RMI interaction mechanism, which we integrated into the *macro data flow* (MDF) evaluation technology of Lithium. We report experimental results that demonstrate the achieved improvements through the proposed optimizations on various testbeds.

1. Introduction

The term *algorithmic skeleton* has been used since Cole's work [5] to denote commonly used patterns of parallel computation and communication. The idea is to employ skeletons as pre-implemented, ready-to-use components that are customized to a particular application by supplying suitable parameters (data or code) [10,12].

This paper deals with the use of parallel skeletons in the emerging grid environments [8]. An inherent property of computational grids is a varying latency of communication between involved machines, i. e. clients and high-performance servers. Moreover in grid environments it is difficult to make definite assumptions about the load and the availability of the computers involved. This brings new, challenging problems in using and implementing skeletons efficiently on grids, as compared to traditional multiprocessors.

Our contribution is a set of new optimization techniques that aim at solving some of the performance problems originating from the latency characteristics of grid architectures. In particular, we developed the optimizations in the context of Lithium, a Java-based skeleton programming library [3]. Apart from "embarrass-

ingly parallel" programs, distributed applications often involve data and control dependencies that need to be taken into account by the skeletons' evaluation mechanism. As Lithium exploits Java-RMI [11] to coordinate and distribute parallel activities, we are interested in integrating new optimizations of RMI [4] into Lithium. The techniques discussed here for RMI can also be applied to other structured parallel programming systems, e.g. ASSIST [1]. As an example, we are considering the adoption of these techniques in ASSIST [1], a system that exploits in part the experiences gained from Lithium and that runs upon different middleware (plain TCP/IP and POSIX processes/threads, CORBA [9] and the Globus Toolkit [7]).

Section 2 of this paper introduces the Lithium programming library, and in Section 3 we introduce an asynchronous interaction mechanism in RMI that improves the performance of grid applications as compared to using standard RMI communication. We show how a mechanism of this kind can be adapted to Lithium and we explain how it helps reducing idle times during program execution in Section 4. Section 4.3 discusses how Lithium's load balancing features can be exploited in the grid context, also taking advantage of the introduced RMI optimizations. We describe our experiments in Section 5, where we study the performance of an image processing application based on a pipeline. Our experimental results compare the effects of the proposed optimizations. We conclude and compare our results to related work in Section 6.

2. Skeleton-based Programming in Lithium

Lithium is a Java-based skeleton library that provides the programmer with a set of nestable skeletons, modeling both data and task/control parallelism [3]. Lithium implements the skeletons according to the *macro data flow* (MDF) execution model [6]. Skeleton programs are first compiled into a data flow graph: Each instruction (i. e. each node) in the graph is a plain data flow instruction. It processes a set of input tokens (Java `Object` items in our case) and produces a set of output tokens (again Java `Object` items) that are either directed to other data flow instructions in the graph or directly presented to the user as the computation results. The output tokens may represent large portions of code, rather than only simple operators or functions (therefore the term *macro* data flow).

The set of Lithium skeletons includes the `Farm` skeleton, modeling task farm computations, the `Pipeline` skeleton, modeling computations structured in independent stages, the `Loop` and the `While` skeleton, modeling determinate and indeterminate iterative computations, the `If` skeleton, modeling conditional computations, the `Map` skeleton, modeling data parallel computations with independent subtasks, and the `DivideConquer` skeleton, modeling divide and conquer computations. All these skeletons are provided as subclasses of the `JSkeleton` abstract class.

Lithium users can encapsulate sequential portions of code in a sequential skeleton by creating a `JSkeleton` subclass.* Objects of the subclass can be used as parameters of the other skeletons. All the Lithium skeletons implement parallel computation

*A `JSkeleton` object has a `run` method that represents the skeleton body

patterns that process a stream of input tasks and compute a stream of results. As an example, a farm with a worker that computes the function f, processes an input task stream with data items x_i, producing the output stream with the corresponding data items equal to $f(x_i)$, whereas a pipeline with two stages computing function f and g, respectively, processes stream of x_i and computes $g(f(x_i))$.

In order to write a parallel application, the Lithium programmer first defines the skeleton structure. As an example, a three-stage pipeline with a task farm as second stage requires the following code:

```
JSkeleton s1 = new s1(...);
JSkeleton w = new w(..);
Farm s2 = new Farm(w);
JSkeleton s3 = new s3(...);
Pipeline main = new Pipeline();
main.addStage(s1);
main.addWorker(s2);
main.addWorker(s3);
```

Then, the programmer declares an application controller, possibly specifying the machines that have to be used:

```
Ske eval = new eval();
eval.setProgram(main);
eval.addHosts(machineNameStringArray);
```

the user can specify the tasks to be computed issuing a number of calls such as:

```
eval.addTask(objectTask);
```

and request the parallel evaluation by issuing the following call:

```
eval.parDo();
```

The program is then executed using the machines whose names were specified in the `machineNameStringArray`, and the programmer can retrieve the results by issuing a number of calls such as:

```
Object res = eval.getResult();
```

The skeleton program is transferred into an MDF graph as a consequence of the `eval.setProgram` call. Lithium implements the so-called *normal form* optimization: The *normal form* of a Lithium program is a semantically equivalent program obtained by means of source-to-source transformations [2]: it consists of a `Farm` evaluated on a sequential program. Any Lithium program can be reduced to the normal form by transforming it to a sequential program composed of the juxtaposition of parallel parts (in the correct order) and farming out the result. The normal form can be computed both statically and just-in-time [3]. The normal form is produced in the `eval.setProgram` code. In our sample case, the data-flow graph is a simple chain of three *macro data flow instructions* (MDFi) is shown in Fig. 1 a), while its normalized version is shown in Fig. 1 b).

For each of the input tasks x_i computed by the program (i. e. for each one of the `Objects` used as argument of a `eval.addTask` call), an MDF graph such as the one presented before is instantiated. The skeleton program is then executed by setting

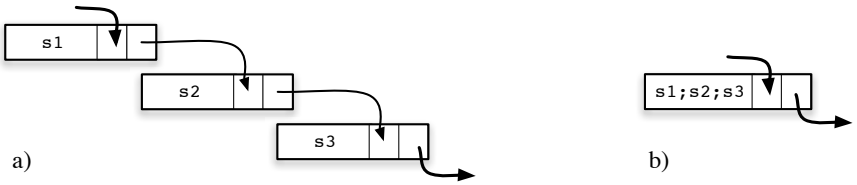

Figure 1: A simple data flow graph (a) and its normal form (b).

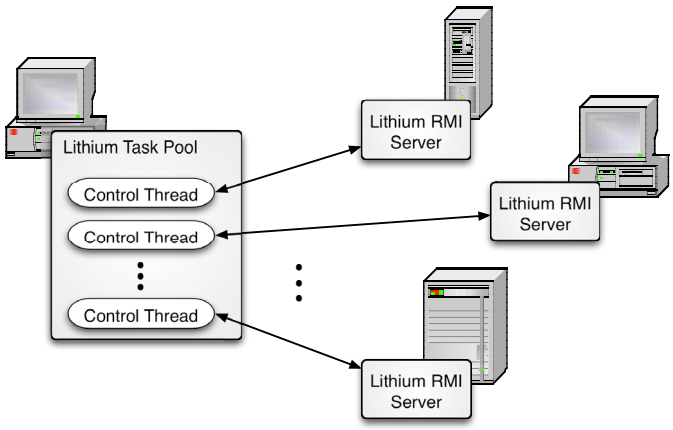

Fig. 2. Lithium implementation outline.

up a task pool manager on the local machine and a remote server process on each of the available remote hosts. The task pool manager creates a new MDF graph for each new input task added via the `eval.addTask` call and dispatches fireable MDFi (that is, MDFi with all the input tokens available) to the remote servers. The remote servers compute the fireable MDFi in the graph(s) and dispatch the results back to the task pool manager. The task pool manager stores the results in the proper place: intermediate results are delivered to other MDFi (that, as a consequence, may become fireable); final results are stored, such that subsequent `eval.getResult` calls can retrieve them.

Remote servers are implemented as Java RMI servers [11]. The Lithium scheduler forks a control thread for each remote server. Such a control thread obtains a reference to one server, then it sends the MDF graph to be executed and eventually enters a loop. In the loop body, the thread fetches a fireable instruction from the taskpool, asks the remote server to compute the MDFi and deposits the result in the task pool (see Figure 2).

3. RMI Optimizations

Using the RMI (*Remote Method Invocation*) mechanism in distributed programming in general and on grids in particular, has the important advantage that the network communication involved in calling methods on remote servers is transparent for the programmer: remote calls are coded in the same way as local calls.

3.1. The Idea of Optimizations

Since the RMI mechanism was developed for traditional client-server systems, it is not optimal for systems with several servers where also server/server interaction is required. We illustrate this with an example of a Lithium Pipeline application: here, the result of a first call evaluating one stage is the argument of a second call (`lithiumServer1` and `lithiumServer2` are remote references):

```
partialResult = lithiumServer1.evalStage1(input);
overallResult = lithiumServer2.evalStage2(partialResult);
```

Such a code is not directly produced by the programmer, but rather by the run-time support of Lithium. In particular, any time a `Pipeline` skeleton is used, this code will be executed by the run-time system of Lithium to dispatch data computed by stage i (`partialResult`) to stage $i + 1$.

When executing this example composition of methods using standard RMI, the result of the remote method invocations will be sent back to the client. This is shown in Fig. 3 a). When `evalStage1` is invoked (arrow labeled by ①), the result is sent back to the client (②), and then to LithiumServer2 (③). Finally, the result is sent back to the client (④). For applications consisting of many composed methods like multistage pipelines, this schema results in a quite high time overhead.

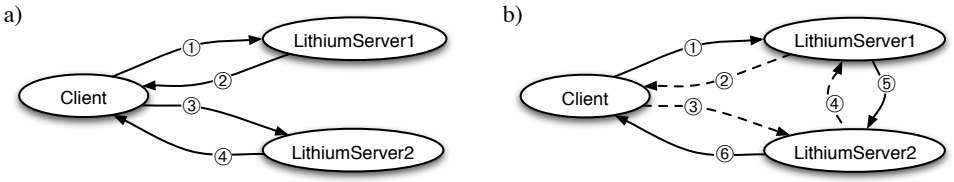

Fig. 3. Method composition: a) using plain RMI, and b) using future-based RMI.

To eliminate this overhead, we have developed so-called *future-based RMI*. As shown in Fig. 3 b), an invocation of the first method on a server initiates the method's execution. The method call returns immediately (without waiting for the method's completion) carrying a reference to (future) execution result (②). This future reference can be used as a parameter for invoking the second method (③). When the future reference is dereferenced (④), the dereferencing thread on the server is blocked until the result is available, i. e. until the first method actually completes. The result is then sent directly to the server dereferencing the future reference

(⑤). After completion of the second method, the result is sent to the client (⑥). Compared with plain RMI, the future-based mechanism can substantially reduce the amount of data sent over the network, because only a reference to the data is sent to the client, while the result itself is communicated directly between the servers. Moreover, communications and computations overlap, thus hiding latencies of remote calls.

3.2. Implementation of Future-Based RMI

In future-based RMI, a remote method invocation does not directly return the result of the computations. It rather returns an opaque object representing a (remote, future) reference to the result. The opaque object has type `RemoteReference`, and provides two methods:

```
public void setValue(Object o) ...;
public Object getValue() ...;
```

Let us suppose `fut` is a `RemoteReference` object. The `fut.setValue(o)` method call triggers the availability and binds `Object o` to `fut`, which has been previously returned to the client as the result of the execution of a remote method. The `fut.getValue()` is the complementary method call. It can be issued to retrieve the value bound to `fut` (`o` in this case). A call to `getValue()` blocks until a matching `setValue(o)` has been issued that assigns a value to the future reference.

The `getValue()` method can be issued either by the same host that executed `setValue(...)` or by a different host, therefore `RemoteReference` cannot be implemented as remote (RMI) class. It is rather implemented as a standard class acting as a proxy. There are two possible situations. First, if matching methods `setValue(...)` and `getValue()` are called on different hosts, the bound value is remotely requested and then sent over the network. In order to remotely retrieve the value, we introduce the class `RemoteValue` (having the same methods as `RemoteReference`), accessible remotely. Each instance of `RemoteReference` has a reference to a `RemoteValue` instance, which is used to retrieve an object from a remote host if it is not available locally. The translation of remote to local references is handled automatically by the `RemoteReference` implementation. Second, if matching methods `setValue(...)` and `getValue()` are called on the same host, no data is sent over the network to prevent unnecessary transmissions of data over local sockets. A `RemoteReference` contains the IP address of the object's host and the (standard Java) hashvalue of the object, thus uniquely identifying it. When `getValue()` is invoked, it first checks if the IP address is the address of the local host.If so, it uses the hashvalue as a key for a table (which is static for class `RemoteReference`) to obtain a local reference to the object. This reference is then returned to the calling method.

4. Optimization Techniques Applied to Lithium

In this section, we describe three optimization techniques for Lithium which are

based on the RMI-optimizations presented in the previous section. All three enhancements are transparent to the application programmer, i. e. an existing Lithium application does not require any changes to use it.

4.1. Task Lookahead on RMI servers

We call our first optimization technique "task lookahead": a server will not have to get back to the task pool manager every time it is ready to process a new task. The immediate return of a remote reference enables the task manager to dispatch multiple tasks instead of single tasks. When a server is presented with a new set of tasks, it starts a thread for every single task that will process this task asynchronously, producing a future result. This is particularly important if we use multi-processor servers, because the multithreaded implementation will exploit all available processors to compute the future results. However, even a single-processor server benefits from look-ahead, because transferring multiple tasks right at the beginning avoids idle times between consecutive tasks.

A Lithium program starts execution by initializing the available servers and binding their names to the local `rmiregistry`. Then the servers wait for RMI calls. In particular, two kinds of calls can be issued to a server:

- A `setRemoteWorker` call is used to send a macro data flow graph to a server. The information in the graph is used to properly execute the MDFi that will be assigned later to the server for execution.
- An `execute` call is used to force the execution of MDFi on a remote node.

In the original Lithium, each control thread performs the following loop [3]:

```
while (!taskPool.isEmpty() && !end) {
   tmpVal = (TaskItem[])taskPool.getTask();
   taskPool.addTask(Ske.slave[im].execute(tmpVal));
}
```

i. e. it looks for a fireable instruction (a *task* according to Lithium terminology), invokes the `execute` method on the remote server and puts the resulting task back to the task pool for further processing. Actually, each control thread and its associated server work in sequence; the behavior is sketched in Fig. 4. Therefore, each Lithium server has an idle time between the execution of two consecutive tasks.

The lookahead-optimization aims at avoiding idle times at the servers. Servers are made multithreaded by equipping them with a thread pool. As soon as a server receives a task execution request, it selects a thread from its pool and starts it on the task. After this invocation (and before the thread completes the task), the server returns a handle to its control thread, thus completing the RMI call. In this way, the control thread may continue to run, possibly extracting another task from the task pool and delivering it to the same server. During this time, some of the server's threads may be still running on previous tasks.

As a result, we can have many threads running at the same time on a single server, thus exploiting the parallelism of the server. In any case, we eliminate

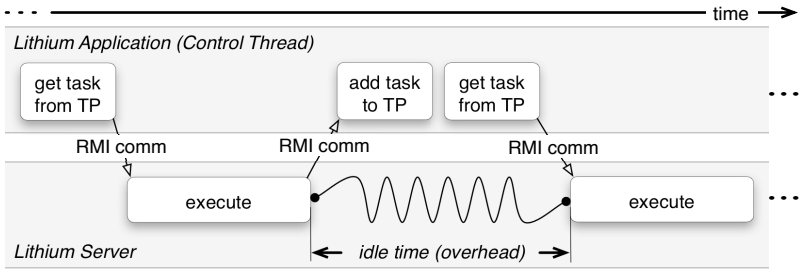

Fig. 4. Server's idle time in original Lithium implementation.

control thread idle time by overlapping useful work in each server and its control thread. Task lookahead is an optimization that improves both normal form and non-normal form program execution times, provided that the machine hosting the task pool (and therefore the control threads) does not become the bottleneck.

4.2. Server-to-Server Lazy Binding

Our second optimization technique is called "lazy binding": a remote server will only bind a new MDFi from the graph if necessary, and analogously the task pool manager will not wait for a remote reference to produce the future result unless it is needed. Here, we use remote references to avoid unnecessary communications between control threads and remote servers. Our implementation of remote references uses hash-tables as local caches, which leads to the caching of intermediate results of the MDF evaluation. The system may identify sequences of tasks that depend on previous ones and make sure that such sequences will be dispatched to a single remote machine. Thus, a sequence of dependent tasks can be processed locally on one server which leads to a further reduction of communication. We will show that the lazy binding technique can be compared to the normal form mechanism.

4.2.1. Normal Form Computation

Let us consider the evaluation of the sequence of two functions, f and g, on a stream of data. In Lithium, the program can be expressed by a two-stage `Pipeline`, whose stages evaluate f and g, respectively. The behavior of the original Lithium system on this program is shown in Fig. 5 a):

(i) The control thread fetches a fireable MDF-instruction and sends it to the associated server (①). The MDF-instruction includes a reference to the function $\uparrow f$ and the input data x_i.
(ii) The Lithium server computes the instruction and sends the resulting data y_i back to the control thread (②).
(iii) The control thread deposits the result in the task pool that makes another

MDF-instruction $\uparrow g(y_i)$ fireable. It will be then fetched by either the same or another control thread and sent to the server (③).

(iv) After the evaluation, the whole execution $z_i = g(f(x_i))$ is completed (④).

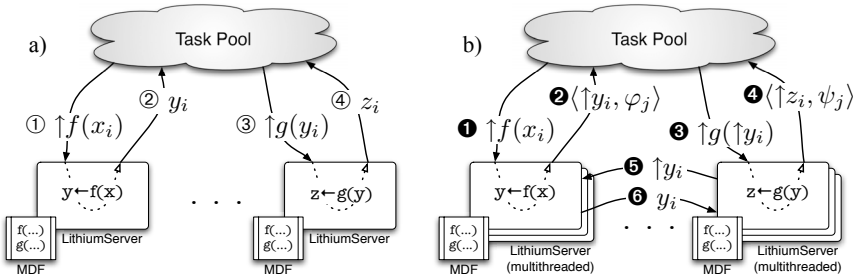

Fig. 5. Communications among Task Pool and Servers. a) Original Lithium. b) Optimized implementation.

The goal of the optimization is reducing the size of communications ② and ③. These communications carry both the reference to the function to be executed and its input data, the latter being the large part. Since the input data might be computed in a previous step by the same server, we can communicate a handle (the `RemoteReference`) for the input/output data instead of their actual values. In this way, each server retains computed values in its cache until these values are used. If they are used by the same server, we greatly reduce the size of round trip communication with the control thread. If they are used by another thread, we move the values directly between servers, thus halving the number of large-size communication. The optimized behavior is shown in Fig. 5 b):

(i) The control thread fetches an MDF-instruction and sends it to a server (❶).
(ii) The Lithium server assigns the work to a thread in the pool and, immediately, sends back the result handle $\uparrow y_i$ (❷). The message may be extended with the completing token φ_j for a previously generated handle j $(i > j)$ in order to make the control thread aware of the number of ongoing tasks.
(iii) The control thread deposits the result in the task pool that makes another MDF-instruction $\uparrow g(\uparrow y_i)$ fireable. This will be fetched by either the same or another control thread and sent to its associated server (❸). Let us suppose the instruction is fetched by another control thread.
(iv) The server immediately returns the handle to the control thread (❹).
(v) To evaluate $\uparrow g(\uparrow y_i)$, the server invokes a `getValue()` method on $\uparrow y_i$ (❺).
(vi) The value y_i arrives at the server (❻), thus enabling the evaluation of $g(y_i)$.

Note that if f and g are evaluated on the same server, then the communications ❺ and ❻ do not take place at all, since references are resolved locally.

The described process can be viewed as a dynamic, runtime version of the normal form optimization. Normal form transforms sequences of calls into an equivalent,

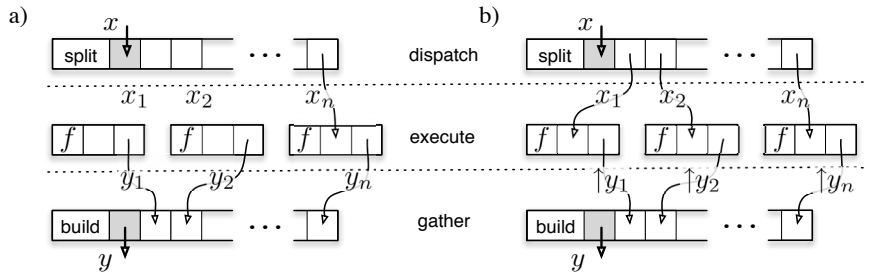

Fig. 6. Execution of a Data Parallel skeleton: a) original Lithium; b) optimized version.

single MDFi. With the proposed optimization, we recognize such sequences at the remote server and perform the computations locally.

4.2.2. Data-Parallel Computation

Lazy binding helps to reduce network traffic, which affects also simple data-parallel computations, which are carried out in Lithium as follows:

- a task (data item) x is divided into a set of (possibly overlapping) n subsets $x_1 \cdots x_n$;
- each subset is assigned to a remote server;
- the results of the computation of all the subsets are used to build the overall result of the data-parallel computation.

This implies the following communication overhead (see Fig. 6 a):

- n communications from the task pool control thread to the remote servers are needed to dispatch subsets;
- n communications from the remote servers to the task pool control threads are needed to collect the subsets of the result;
- one communication from the control thread to a remote server is needed to send the subsets in order to compute the final result;
- one communication from the remote server to the task pool control thread is needed to gather the final result of the data-parallel computation.

The suggested optimization is as follows (see Fig. 6 b):

- each time a data-parallel computation is performed, the task pool control thread generates and dispatches all "body" instructions, i. e. instructions that compute a subset of the final result. The remote servers immediately return handles $\uparrow y_1 \cdots \uparrow y_n$ (`RemoteReference`s) representing the values still being computed;

- after receiving all handles, the control thread dispatches the "gather" MDFi (i. e. the instruction packing all the sub-results into the result data structure) to the remote server hosting the major amount of references to sub-results. When this instruction is computed, the result is sent back to the task pool.

In this way, we avoid moving the intermediate results back and forth between the task pool threads and the remote servers during execute and gather phases (see 6).

4.3. Load Balancing

In this section, we describe how we adapted the load-balancing mechanism of Lithium to the optimized evaluation mechanisms, in order to achieve a stable level of parallelism on all servers. This is accomplished by measuring the number of active threads on the servers.

Our asynchronous communications lead to a multithreaded task evaluation on the servers. The scheduler can dispatch a task by sending it to a server, which is already evaluating other tasks. This server will start evaluating the new task in parallel. We implemented this server-side multithreading using a thread pool, which is more efficient than spawning a new thread for each task. However, tasks may differ in size, and machines in a Grid are usually heterogeneous. Without a suitable load-balancing strategy, this may lead to an awkward partitioning of work.

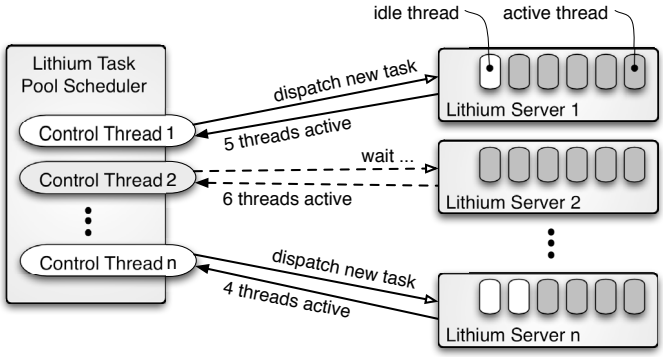

Fig. 7. Communication schema for the load balancing mechanism.

To balance the load in the system, we must measure the current load of each server. One possibility would be to use a new remote method, which, however, is inefficient since it implies more remote communication. Instead, we exploit the fact that the scheduler already communicates frequently with the remote servers by sending tasks, i. e. data records with a reference to a future value. We extend each data record by a number that reports to the scheduler the actual work-load on the server. So, every time the scheduler sends a task to a server, it gets the number of

threads currently running on that server. The scheduler can re-check this number and, if there is already much load on this server, it can decide to release the task again and wait instead. Accordingly, another scheduler thread will process the task by sending it to another server. So, dispatching tasks and measuring work-load can be done in one remote communication like shown in Fig. 7: Here, we have a maximum number of 6 active threads per server. Dispatching tasks to server 1 and server n yields the actual work-load (5 active threads at server 1, 6 active threads at server n), which means that the scheduler can continue to dispatch tasks to these servers. But for a server that has already reached the maximum number of active threads (server 2 in the figure), the scheduler waits until the number of active threads has fallen below the limit.

With many remote servers and, correspondingly, control threads running in the scheduler, the measured value may already be obsolete when we send the next task. However, since asynchronous communication causes tasks to be dispatched with a high frequency, the suggested technique is precise enough for an efficient load balancing. This has also been proved by our experiments that included checkpointing.

5. Experiments

For the evaluation of our optimizations, we conducted performance measurements on three different distributed platforms.

(i) A dedicated Linux cluster at the University of Pisa. The cluster hosts 24 nodes: one node devoted to cluster administration and 23 nodes (P3@800Mhz) exclusively devoted to parallel program execution. Described in Sec. *5.1*.

(ii) A distributed execution environment including Linux and Sun SMP machines. The client runs on a Linux machine in Münster and the servers run on a set Sun SMP machines in Berlin. Described in Sec. *5.2*.

(iii) A Grid-like environment, including two organizations: the University of Pisa *(di.unipi.it)* and an institute of the Italian National Research Council in Pisa *(isti.cnr.it)*. The server set is composed of various different Intel Pentium and Apple PPC computers, running Linux and Mac OS X respectively (The detailed configuration is shown in Fig. 10 left). The comparison of computing power of machines is performed in terms of BogoPower, i. e. the number of tasks per second which a given machine can compute running the sequential version of the application (described in Sec. *5.3*).

The three testing environments represent significantly different scenarios:
(i) is characterized by uniform computing power and high-bandwidth communications across the whole system (client and servers);
(ii) has low latency and high bandwidth for server-to-server communication, while the client is connected to the servers with a fairly slow connection.
(iii) shows a heterogenous distribution of computing power and connection speed.

The image processing application we used for our tests uses the Pipeline skeleton, which applies two filters in sequence to 30 input images. All input images are true-

color (24 bit color depth) of 640x480 pixels size. We used filters from the Java Imaging Utilities that add a blur effect and an oil effect. The filters were configured to involve 5 neighboring pixels in each calculation. load-balancing for the future-based version was adjusted to the maximum of 6 concurrent threads per node. The lower limit was set to 2 threads. All the experiments have been performed using the J2SE Client VM SDK version 1.4.1.

As described below, the optimized Lithium version shows a clear time advantage over the standard version along all tested configurations.

5.1. Dedicated Cluster

Figure 8 (left) shows the measured time in seconds, for both the original Lithium and the optimized version running on the dedicated cluster in Pisa. The speedup in the right part of the figure is calculated with respect to the sequential version of the application running on the same cluster. The plots show that the future-based version performed approximately twice as fast as standard Lithium.

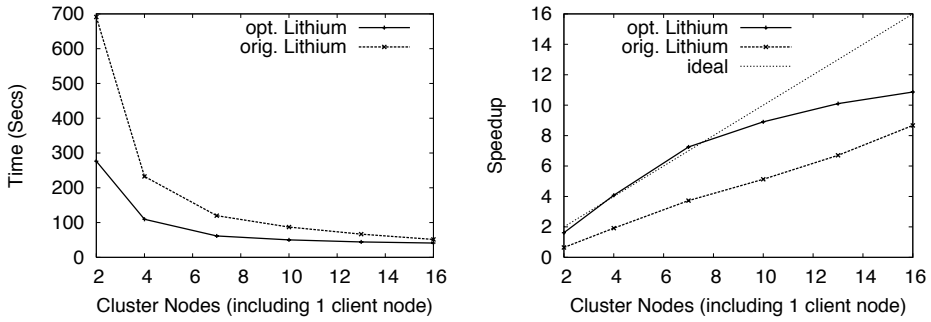

Fig. 8. Measured execution times and speedup on the Pisa cluster (i).

Task lookahead allows to overlap communication, computation, and data I/O time, while lazy binding ensures that all data transfer between the stages of the pipeline takes place on the server side without client interaction.

5.2. Distributed Environment

Figure 9 shows the measured time in seconds, for both the original Lithium and the optimized version running in the environment (ii). These tests demonstrate a clear increase in performance due to the proposed optimizations. By introducing a server-to-server communication, data exchange between client and servers is reduced considerably. This feature exactly matches the strength of environment (ii) in communicating using the fastest paths.

5.3. Grid-like Environment

As shown in Fig. 10, the optimized version performs better than the standard one, also in a very heterogenous environment.

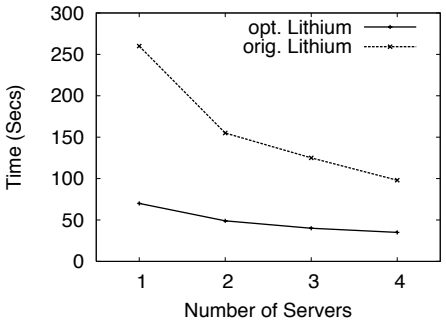

Fig. 9. Execution times for environment (ii).

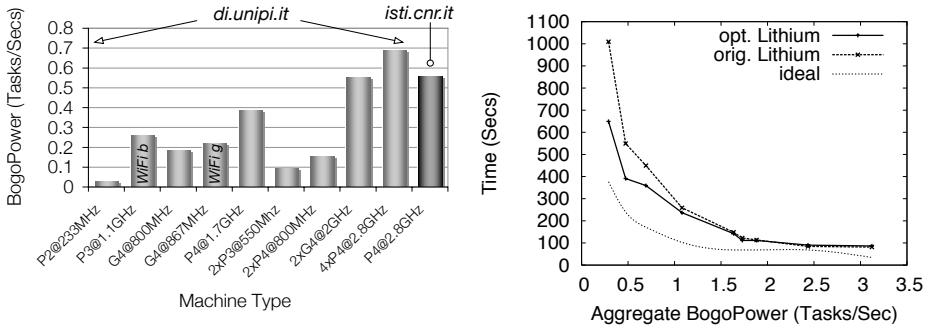

Fig. 10. Left) Testing environment (iii). Right) Execution times vs. increasing (Bogo-)powerful ordered server set.

To take the varying computing power of the different resources into account, the performance increase is documented by mean of the *BogoPower* measure, which enables the comparison between application's actual parallel performance and application ideal performance (see Fig. 10 right). A speedup graph usually measures the aggregate power of a system by the bare number servers, which does not satisfy the conditions given in Grid-like environments. The BogoPower-measure enables the description of the aggregate BogoPower of a heterogeneous distributed system as the sum of each box individual BogoPower contribution (see. Fig. 10 left). Application ideal performance curve is evaluated w.r.t. a system exploiting a given BogoPower rank and assuming both an optimal scheduling of tasks and zero communication costs.

We use environment (iii) for demonstrating to what extent the requirements of a Grid system are met by our improved load balancing strategy.
The Lithium runtime system continuously monitors the servers' states and raises or drops the number of threads running on the servers in respect of the current status. To demonstrate the dynamic load balancing behaviour of the scheduler, we performed two identical experiments in environment (iii) with differing load on one of the involved machines. Fig. 11 shows the results of these experiments. In Fig. 11 right) less tasks are dispatched to the most powerful machine because in

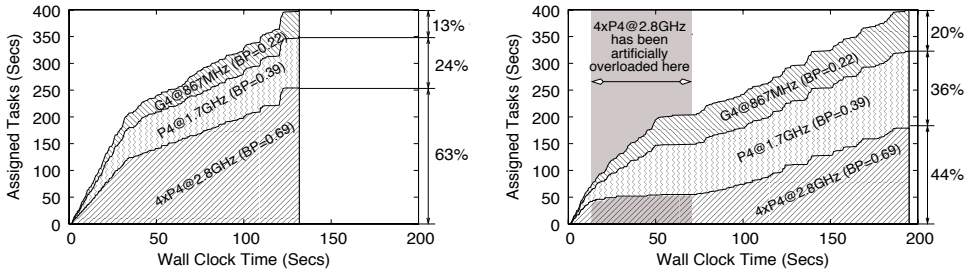

Fig. 11. Detailed history of tasks scheduling Left) All machines are dedicated to the experiment. Right) One of the machine is externally overloaded from time 10 to 70.

this experiments, this machine was charged heavily by another application. Fig. 12 shows the history of threads issued to the overloaded machine. As evident from the figure, when the machine load grows too much, the scheduler drops the number of active threads. This decision is supported by system-wide historical statistics maintained by the load balancing module of the client.

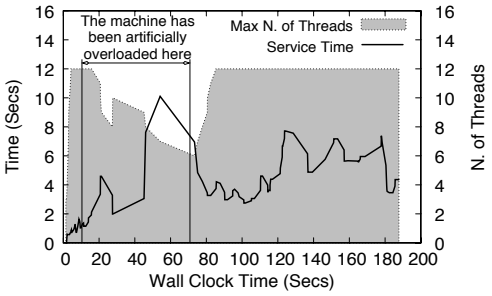

Fig. 12. Throttling in task scheduling.

6. Conclusions

We have described several novel optimization techniques aimed at an efficient implementation of parallel skeletons in distributed grid environments with high communication latencies. As a reference implementation, we took the Lithium programming system and studied the effects of three different optimizations based on the asynchronous, future-based RMI mechanism: (1) dispatching batches of tasks, rather than single tasks, to remote servers ("task lookahead"); (2) caching intermediate results on the remote servers, thus allowing to reduce the communication overhead ("lazy binding"); (3) adapting the load-balancing strategies to the multithreaded evaluation mechanism initiated by the "task lookahead" and implementing it without an increase in remote communication.

Server-to-server communication in RMI programs can also be found in [13], where RMI calls are optimized using call-aggregation and where a server can directly invoke methods on another server. While this approach optimizes RMI calls by reducing the amount of data, the method invocations are not asynchronous as in our implementation: They are delayed to find as many optimization possibilities as possible.

Note that all three techniques have been integrated into Lithium transparently to the user, i. e. applications developed on top of the original framework can directly use the optimized version without any changes in the code. The presented opti-

mization techniques can easily be applied to grid environments other than Lithium. Furthermore, they are not restricted to RMI as a communication mechanism.

Acknowledgements

Our acknowledgements go to Julia Kaiser-Mariani and the anonymous referees of the draft version of this paper for many helpful remarks.

References

[1] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance grid programming environments. In José C. Cunha and Omer F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer, 2004. (to appear, draft available as TR-04-09, Dept. of Computer Science, University of Pisa, Italy, Feb. 2004).

[2] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimization. In *Proc. of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'99)*, pages 955–962, Cambridge, Massachusetts, USA, November 1999. IASTED/ACTA press.

[3] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.

[4] M. Alt and S. Gorlatch. Future-based RMI: Optimizing compositions of remote method calls on the grid. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Proc. of the Euro-Par 2003*, number 2790 in Lecture Notes in Computer Science, pages 427–430. Springer, August 2003.

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

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

[7] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6):37–46, June 2002.

[8] Ian Foster and Carl Kesselmann, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[9] The CORBA & CCM home page. `http://ditec.um.es/~dsevilla/ccm/` .

[10] H. Kuchen. A skeleton library. In B. Monien and R. Feldmann, editors, *Proc. of Euro-Par 2002*, number 2400 in Lecture Notes in Computer Science, pages 620–629. Springer-Verlag, 2002.

[11] C. Nester, R. Philippsen, and B. Haumacher. A more efficient RMI for Java. In *Proc. of the Java Grande Conference*, pages 152–157. ACM, June 1999.

[12] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor&Francis, 1998.

[13] K. C. Yeung and P. H. J. Kelly. Optimising Java RMI programs by communication restructuring. In D. Schmidt and M. Endler, editors, *Middleware 2003: ACM/IFIP/USENIX International Middleware Conference*, pages 324–343. Springer-Verlag, 2003.