Optimization Techniques for Implementing Parallel Skeletons in Grid Environments

M. Aldinucci[1], M. Danelutto[2], and J. Dünnweber[3]

[1] Inst. of Information Science and Technologies – CNR, Via Moruzzi 1, Pisa, Italy
[2] Dept. of Computer Science – University of Pisa – Viale Buonarroti 2, Pisa, Italy
[3] Dept. of Computer Science – University of Münster – Einsteinstr. 62, Münster, Germany

Abstract. Skeletons are common patterns of parallelism like, e.g., farm, 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) implementation technology of Lithium. We report initial experimental results that demonstrate the achieved improvements through the proposed optimizations on a simple grid testbed.

1 Introduction

Since Cole's work [6], the term algorithmic skeleton has been used to denote commonly used patterns of parallel computation and communication. The idea of parallel programming with skeletons is to use skeletons as ready-to-use components that are customized to a particular application by supplying suitable parameters (data or code) [14, 12, 7].

This paper deals with the use of parallel skeletons in the emerging grid environments [10]. An inherent property of grids is a varying latency of communication between involved machines, i.e. clients and high-performance servers. Also, it is usual in grid environments not to be able to make any definite assumptions about the load and the availability of the processing elements involved. This brings new, challenging problems in the task of implementing skeletons efficiently on such systems, 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 features of grid architectures. In particular, we developed the optimizations in the context of Lithium, a full Java skeleton programming library [4]. As Lithium exploits plain Java-RMI [13] to coordinate and distribute parallel activities, we are interested in merging new optimizations of RMI [5] into Lithium. Although the techniques discussed here are described for RMI, they can also be applied to other structured parallel programming systems. As an example, we are considering the adoption of these techniques in ASSIST [17], a system that exploits in part the experiences gained from Lithium and that runs upon different middleware (plain TCP/IP and POSIX processes/threads [1], CORBA [11] and the Globus Toolkit [9]).

In this work we focus on performance issues. Apart from embarrassingly parallel programs (i.e., those exploiting task farm parallelism only), distributed applications may involve certain dependencies that need to be taken into account by an efficient evaluation mechanism. Lithium provides, among the others, task farm, divide-and-conquer, and pipeline skeletal constructs. The latter two actually involve such dependencies. In a pipeline, e.g., each stage of a single task evaluation depends on the previous ones, whereas multiple tasks can be processed independently.

We introduce an asynchronous interaction mechanism that improves the performance of grid applications, compared to using plain 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. We discuss how Lithium's load balancing features can be exploited in a Grid context, also taking advantage of the RMI optimizations introduced. We also report a case study, where we compare the performance of an image processing application based on a pipeline, implemented using both standard Lithium and the new optimized version.

The experimental results discussed in Sec. 5 will compare the effects of the proposed optimizations with respect to the original Lithium. We do not use any kind of classical grid middleware, such as Globus[9] or other kind of grid RPC systems [16]. Rather, we fully exploit Java RMI as middleware, building Lithium implementation and optimizations on top the of regular Sun JDK 1.4.2 environment [15].

2 The Lithium Skeleton Library

Lithium is a Java skeleton library that provides the programmer with a set of fully nestable skeletons, modeling both data and task/control parallelism [4]. Lithium implements the skeletons by fully exploiting a *macro data flow* (MDF) execution model [8]. According to this model, the user's skeleton programs are first compiled into a data flow graph. Each instruction (i.e. each node) in the resulting 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 computation of the output tokens uses a possibly large portion of code, rather than simple operators or functions (therefore the term *macro* data flow) and starts when all the input tokens are available.

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

Lithium users can encapsulate sequential portions of code in a sequential skeleton by creating a `JSkeleton` subclass[1]. Objects of the subclass can be used as parameters of other, different skeletons. All the Lithium skeletons implement parallel computation patterns that process a stream of input tasks to 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 item x_i producing the output stream with the corresponding data item equal to $f(x_i)$, whereas a pipeline with two stages computing function f and g, respectively, processes stream of x_i computing $g(f(x_i))$.

In order to write a parallel application, the Lithium programmer must first define 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();
```

[1] a `JSkeleton` object is an object having a `Object run(Object)` method that represents the sequential skeleton body

```
main.addStage(s1);
main.addWorker(s2);
main.addWorker(s3);
```

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

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

he 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:

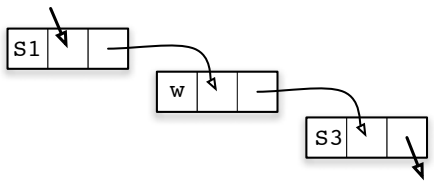
```
eval.pardo();
```

After the completion of the call, the program is 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();
```

As stated before, Lithium exploits the MDF schema for skeletons. Therefore, the `eval Ske` object transforms the skeleton program into an MDF graph as a consequence of the `eval.setProgram` call. Actually, Lithium implements so-called normal form optimizations. The *normal form* of a Lithium program is a semantically equivalent program obtained from the original program by means of source-to-source transformations. As a rule, it shows better performance and speedup with respect to the original program (under mild additional requirements) [2]. The normal form of a Lithium program is proved to exploit the same functional semantics of the original program (and a different parallel behavior) [3]. Basically, a Lithium program is in the normal form if 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 [4].

Also the normal form production is performed in the `eval.setProgram` code. Therefore, in our sample case, the graph obtained is a simple chain of three *macro data flow instructions* (MDFi):



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 setting up a task pool manager on the local machine and a remote server process on each of the available remote hosts. The

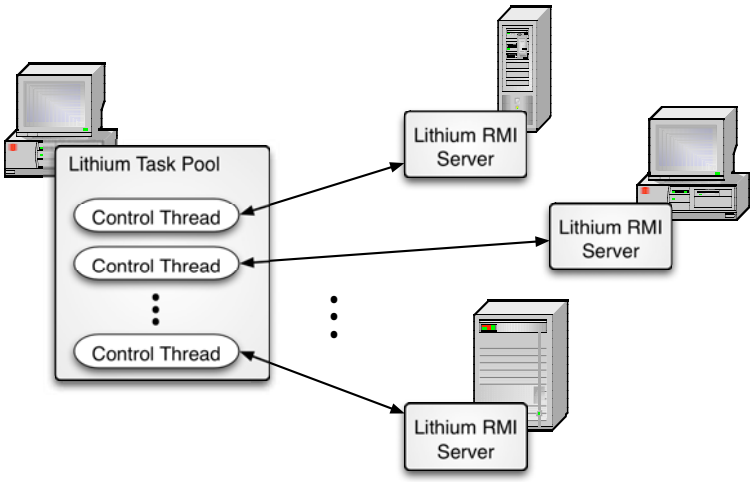


Fig. 1. Lithium implementation outline

task pool manager takes care of creating 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. In turn, the task pool manager stores the results in the proper place: if these are intermediate results, they are delivered to other MDFi (that, as a consequence, possibly become fireable); if these are final results, then they are stored in such a way that subsequent `eval.getResult` calls can retrieve them.

Remote servers are implemented as Java RMI servers[13]. The Lithium scheduler forks a control thread for each remote server. Such a control thread obtains a reference to one server, first; 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 1).

Lithium is particularly suitable for experimenting with RMI optimizations. All distribution mechanisms are implemented in the task pool manager, which is centralized in a single scheduler[2], which simplifies the task of adding new features to the system, e.g., concerning its availability. When network connections are highly transient, the scheduler is the single starting point where failover procedures need to intervene. The same holds for security, logging and other crosscutting concerns of the system that can not be covered so easily in a peer-to-peer setting, for example.

3 RMI Optimizations

Using the RMI (*Remote Method Invocation*, i.e. the object-oriented transposition of remote procedure call) 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 exactly the same way as local calls.

[2] A distributed version of the scheduler has also been investigated. For the sake of brevity, we consider the centralized scheduler.

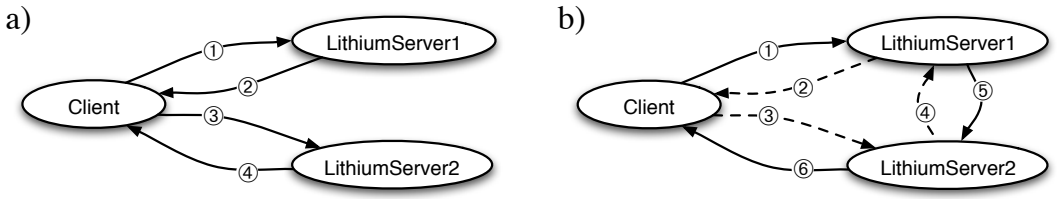


Fig. 2. Method composition. a) Using plain RMI. b) using future-based RMI.

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 exemplary 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 in the run-time of Lithium to dispatch data computed by stage i (`partialResult`) to stage $i + 1$.

When executing this composition of methods using standard RMI, the result of the remote method invocations will be sent back to the client. This is shown in the above example in Fig. 2 a). When `evalStage1` is invoked (①), the result is sent back to the client (②), 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 way of processing results in a very high time overhead.

To eliminate this overhead, we have developed so-called *future-based RMI*. As sketched in Fig. 2 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 future reference to (future) execution result (②). The 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. 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; the result itself is communicated directly between the servers. Moreover, communications and computations overlap, effectively hiding latencies of remote calls.

Server-to-server communication in RMI programs can also be found in [18], 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. Instead, they are delayed to find as many optimization possibilities as possible.

3.2 Implementation of Future-Based RMI

Using 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 result of the execution of a remote method. The `fut.getValue()` is the complementary method call. It can be issued to retrieve the value that has been binded to `fut` (`o` in this case). A call to `getValue()` blocks until a matching `setValue()` has been issued assigning 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. If matching methods `setValue()` and `getValue()` are called on different hosts, the binded 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.

If, otherwise, 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. The `getValue()` implementation achieves this behaviour by checking if the requested object is locally available. If that is the case, `getValue()` returns a local reference. Therefore, `RemoteReference` instances contain 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 where `getValue()` is invoked. If so, it uses the hashvalue as a key for a local hashtable (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 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 them.

4.1 Task Lookahead on RMI servers

We call our first optimization technique "task lookahead", which means that 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 parallel machines for remote computation,

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 version, each Lithium control thread performs the following loop [4]:

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

that is, 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. 3. Therefore, each Lithium server has an idle time between the execution of two consecutive tasks.

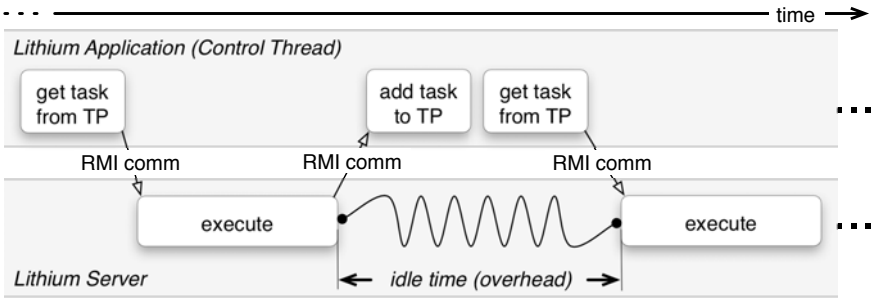


Fig. 3. Server's idle time in original Lithium implementation.

The lookahead-optimization aims at avoiding idle times at the Lithium servers. Servers are made multithreaded by equipping them with a thread pool. As soon as a server receives a task execution request, the server 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 some previous tasks.

As a result, we can have many threads running at the same time in a single server, thus exploiting the parallelism of the server. In any case, we eliminate control thread idle time by overlapping useful work in each Lithium 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.

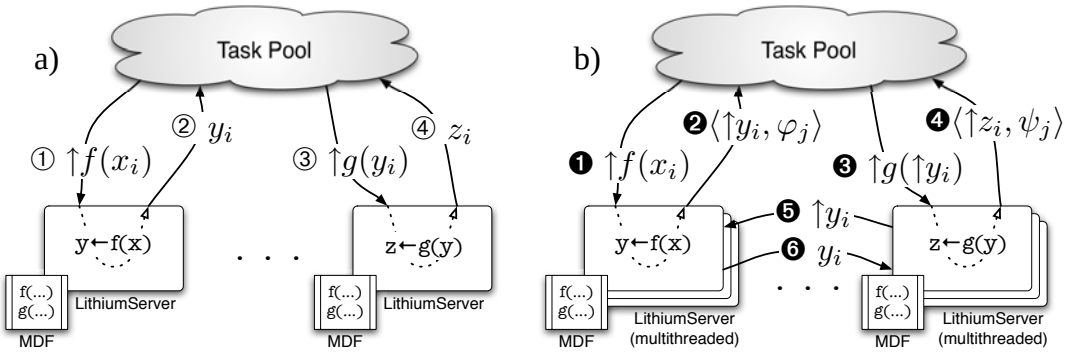


Fig. 4. Communications among Lithium Task Pool and Lithium Servers. a) Original Lithium implementation. b) Optimized Lithium implementation.

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 in order to avoid unnecessary communications between task pool 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.

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 implementation on this program is shown in Fig. 4 a):

1. 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.
2. The Lithium server computes the instruction and sends the resulting data y_i back to the control thread (②).
3. 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).
4. After the evaluation, the whole execution $z_i = g(f(x_i))$ is completed by (④).

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 by throwing away the task pool from the path. The optimized behavior is sketched in Fig. 4 b):

1. A control thread fetches a fireable MDF-instruction and sends it to its associated server (❶). The MDF-instruction includes a reference to the function $\uparrow f$ and an input data x_i.
2. 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 for a previously generated handle in order to make the control thread aware of the number of ongoing tasks.
3. 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.
4. Similarly to 2, the server immediately returns the handle to the control thread (❹).
5. To evaluate $\uparrow g(\uparrow y_i)$, the server invokes a `getValue()` method on $\uparrow y_i$ (❺).
6. The value y_i arrives at the server as the result of `getValue()` RMI invocation (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, single MDFi. With the proposed optimization, we recognize such sequences at the remote server and perform the computations locally.

Data-Parallel Computation Lazy binding is an optimization helping to reduce network traffic, which affects multistage task parallel programs, such as pipelines, and also simple data parallel computations.
Data-parallel computations are carried out by 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.

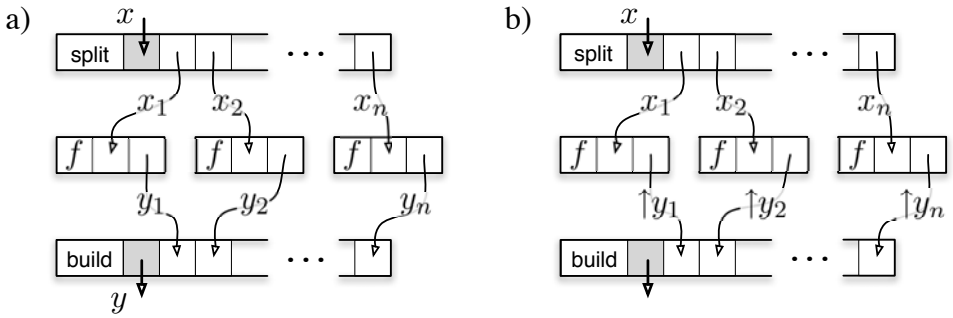


Fig. 5. Execution of a Data Parallel skeleton. a) Original Lithium implementation. b) Optimized Lithium implementation.

This implies the following communication overhead in the Lithium implementation (see Fig. 5 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. 5 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 to the remote servers. The remote servers immediately return handles (`RemoteReference`s) representing the value (in the subset) that will eventually be computed $\uparrow y_1 \cdots \uparrow y_n$;
– as soon as all the handles have been received, the task pool control thread dispatches the "gather" MDFi (i.e., the instruction packing all the sub-results into the final result data structure) to the remote server hosting the major amount of references to sub-results. When this instruction is eventually computed, the final result will be 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.

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 communication strategy leads 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 we also need to take into account that machines in a Grid are usually heterogeneous. So, the tasks on the distributed servers differ in time costs which will lead to an awkward partitioning of work without a suitable load-balancing strategy.

To balance the load in the system, we need a method to 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 holding a reference that is used to retrieve a future value. For our new load-balancing strategy, we extend each data record by a number that reports the actual work-load on the server back to the scheduler. So, every time the scheduler sends a task to a server, it gets the number of threads currently running on that server. Before sending a new task, 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. 6: Here, we exemplarily assume a maximum number of 6 active threads per server. As can be seen, dispatching tasks to server 1 and server n also yields the actual work-load (5 active

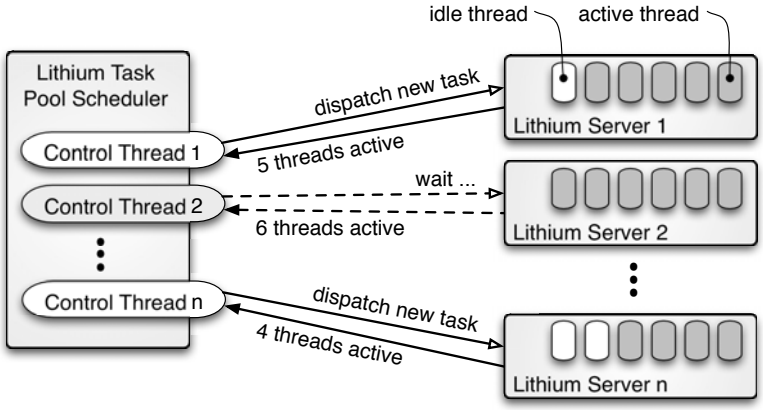


Fig. 6. Communication schema for the load balancing mechanism.

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 a lower limit again.

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 did some performance measurements using a dedicated Linux cluster at the University of Pisa, and a set of widely distributed heterogeneous servers (spread across Pisa, Münster and Berlin). The dedicated cluster in Pisa hosts 24 nodes: one node devoted to cluster administration, and 23 nodes (800Mhz Pentium III) exclusively devoted to parallel program execution. The distributed execution environment includes Linux and Sun SMP boxes. All the experiments have been performed using the J2SE Client VM SDK version 1.4.1.

The image processing application we used for the tests documented here uses the Lithium Pipeline skeleton, which applies two filters in sequence to 30 input images. All input images are true-color images (24 bit color depth) of 640x480 pixels size. We used filters that add a blur effect and an oil effect to the images, based on the MeanFilter and the OilFilter tools from the Java Imaging Utilities available at http://jiu.sourceforge.net. The filters were configured to involve 5 neighboring pixels in each calculation. The load-balancing setting for the future-based version was chosen at the maximum of 6 concurrent threads per node. The lower limit was set to 2 threads. So, the number of concurrent threads running on a server had to fall below 2, before the server was supplied with new tasks. Accordingly to these settings, the thread-pools were configured to hold 6 threads initially.

Fig. 7 (left) shows the measured time in milliseconds, for both the original Lithium and the optimized version running in the dedicated cluster in Pisa. The speedup, shown in the right part of the figure, is figured out with respect to the sequential version of the application running in the same cluster. Since both input and output data are stored in permanent files, execution times include data I/O time for both sequential and parallel version of the application. The plots show that the future-based version performed approximately twice as

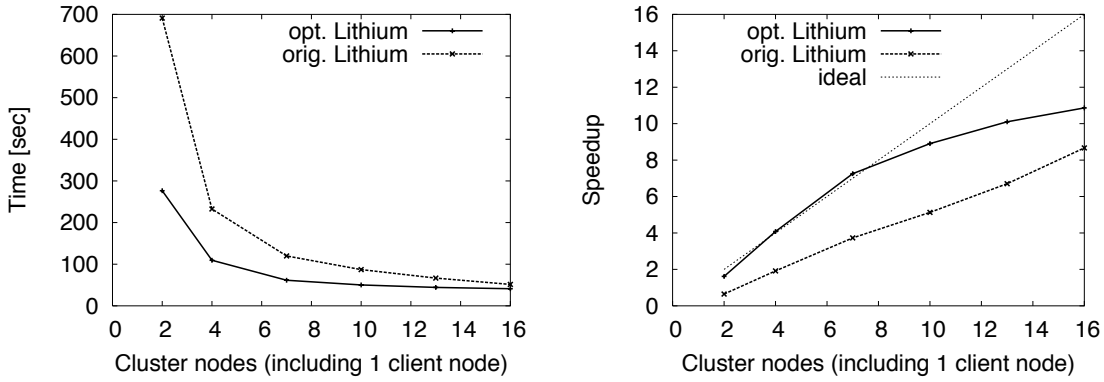


Fig. 7. Measured execution times and speedup on the Pisa cluster.

fast as standard Lithium. Observe the large gain enabled by proposed optimizations for 2 cluster nodes (1 client + 1 server). Despite the whole number crunching work is performed by the server, the application reaches 2 as overall speedup: task lookahead enables the client and the server to overlap communication, computation, and data I/O time, while lazy binding ensures that all data transfer between the stages of the Pipeline take place on the server side without client interaction. The optimized version maintains a clear advantage over the standard version along all tested configurations.

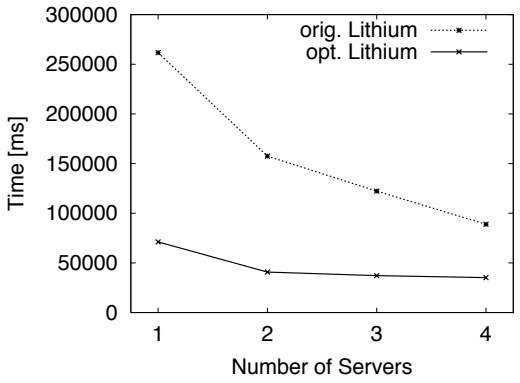


Fig. 8. Times measured using heterogeneous high-performance grid servers.

Some preliminary results for 4 different multiprocessor servers processing the same tasks are shown in Fig. 8. Also these tests documented an increase in performance due to proposed optimizations. However, for different server types the lookahead degree and the load-balancing needs to be adapted for each server separately and dynamically. As an example, an excessive lookahead degree may turn in the overload of a slow/heavy-loaded server which may become a performance bottleneck. We are planning to focus more on these adaptations and publish detailed results for a broad variety of servers in our future work.

6 Conclusions

We have described 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 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.

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

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

Acknowledgments. This work has been supported by a travel grant from the German-Italian exchange programme VIGONI.

References

[1] The ACE team. *The Adaptive Communication Environment home page.* (`http://www.cs.wustl.edu/~schmidt/ACE.html`).

[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 and M. Danelutto. An operational semantics for skeletons. In *Proc. of the International Conference ParCo 2003: Parallel Computing*, Dresden, Germany, September 2003.

[4] 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.

[5] 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 LNCS, pages 427–430. Springer, August 2003.

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

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

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

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

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

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

[12] 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.

[13] 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.

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

[15] Sun Microsystems. *The Java home page.* (`http://java.sun.com`).

[16] S. Vadhiyar, J. Dongarra, and A. YarKhan. GrADSolve - RPC for high performance computing on the Grid. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Proc. of the Euro-Par 2003*, number 2790 in LNCS, pages 394–403. Springer, August 2003.

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

[18] 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.