# Optimization Techniques for Skeletons on Grids

Marco Aldinucci[a], Marco Danelutto[b], Jan Dünnweber[c] and Sergei Gorlatch[c]

[a]Italian National Research Council (ISTI–CNR),
Via Moruzzi 1, Pisa, Italy

[b]Dept. of Computer Science – University of Pisa
Largo B. Pontecorvo 3, Pisa, Italy

[c]Dept. of Computer Science – University of Münster
Einsteinstr. 62, Münster, Germany

Skeletons are common patterns of parallelism, such as farm and pipeline, that can be abstracted and offered to the application programmer as programming primitives. We describe the use and implementation of skeletons on emerging computational grids, with the skeleton system Lithium, based on Java and RMI, as our reference programming system. Our main contribution is the exploration of optimization techniques for implementing skeletons on grids based on an optimized, *future-based RMI* mechanism, which we integrate into the macro-dataflow evaluation mechanism of Lithium. We discuss three optimizations: 1) a lookahead mechanism that allows to process multiple tasks concurrently at each grid server and thereby increases the overall degree of parallelism, 2) a lazy task-binding technique that reduces interactions between grid servers and the task dispatcher, and 3) dynamic improvements that optimize the collecting of results and the work-load balancing. We report experimental results that demonstrate the improvements due to our optimizations on various testbeds, including a heterogeneous grid-like environment.

## 1. Introduction

Parallel and distributed computer systems have recently become less expensive and easier to build from the engineering point of view. However, the development of correct and efficient software for such systems remains a complicated and error-prone task. Most application programs are still being written at a low level of the C or Java programming language, combined with a communication library or mechanism like MPI or RMI. For performance reasons, programs are often tuned towards one specific machine configuration.

In sequential programming, low-level coding for a specific machine also prevailed three decades ago. The software engineering solution to overcome this problem was to introduce levels of abstraction, effectively yielding a tree of refinements, from the problem specification to alternative target programs [1]. In the parallel setting, high-level programming constructs and a refinement framework for them would be very desirable, because of the inherent difficulties in maintaining the portability of low-level parallelism [2].

Since the 1990s, the "skeleton approach" [3] deals with high-level languages and methods for parallel programming; see [4] for the current state of the art. In this approach, parallel algorithms are characterised and classified by their adherence to generic patterns of computation and interaction. For example, many applications from different fields share the well-known pipeline structure of control and/or data flow, probably in somewhat different flavours. Skeletons express and implement such common patterns and can be used as program building blocks, which can be customized to a particular application to express its particular flavour of the generic pattern.

Traditionally, skeletons are used to extend existing sequential programming languages or parallel programming frameworks (e.g., C+MPI or Java+RMI). Examples of available parallel libraries that support skeletal programming include, among others: *SKElib* [5], SKiPPER [6], *Lithium* (see Section 3), and *eSkel* [7]. Several real world applications have been programmed and experimented with to validate the effectiveness of the approach. Application fields where skeletons have proven to be useful include computational chemistry [8], massive data-mining [9], remote sensing and image analysis [10,11], numerical computing [12], and web search engines, e.g., Google [13].

In this work, we focus on the performance issues arising when skeletons are put into modern heterogeneous and highly dynamic distributed environments, so-called *grids*. The particular contribution of this paper is a set of novel optimization techniques that aim at solving some of the performance problems of skeleton-based programming that originate from the specific characteristics of grids. In particular, we develop optimizations in the context of Lithium, a Java-based skeleton programming library [14]. Lithium uses Java-RMI [15] to coordinate and distribute parallel activities, such that we were able to integrate our recently developed optimizations of RMI presented in [16] into Lithium.

In Section 2 and 3 we motivate our work and describe the Lithium system. Section 4 explains an optimized RMI mechanism, *future-based RMI*, which is used in Section 5 to implement the three proposed optimizations in Lithium. We describe our experiments in Section 6, where we study the performance improvement of an image processing application due to the proposed optimizations. We compare our results to related work and discuss the general applicability of our approach in Section 7.

## 2. Motivation for Optimizing Skeletons on Grids

*Skeletons* [3] are commonly used patterns of parallel or distributed 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) [17,18]. Correspondingly to the frequently used control structures in parallel programming, the most popular skeletons are: pipeline, farm, reduce, scan and divide&conquer.

Our motivation in this paper is the performance of parallel skeletons in emerging distributed environments like grids [19]. A specific property of grids is the varying latency and bandwidth 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 individual computers. This raises new, challenging problems in using and implementing skeletons efficiently, as compared

to traditional multiprocessors.

Our goal is to develop specific optimization techniques that aim at improving the coordination and communication between the servers of the grid during the execution of a skeletal program. There are two extreme levels of abstraction, at which such optimizations are in principle possible:

- at the high level of algorithmic design: such semantics-preserving transformations were studied by ourselves, e.g., in [20];
- at the low level of the particular communication mechanism.

Our approach is to take the middle path between these two alternatives, which has not been covered by previous research. This corresponds also nicely to the general philosophy of skeletons: patterns are abstracted and provided as a programmer's toolkit, with specifications which transcend architectural variations but implementations which recognize these to enhance performance. Our optimizations demonstrate that skeletons

- enhance portability and re-use by absolving the programmer of responsibility for detailed realization of the underlying patterns;
- offer scope for static and dynamic optimization, by explicitly documenting information on algorithmic structure (e.g., data dependencies) which would often be impossible to extract from equivalent unstructured programs.

Apart from "embarrassingly parallel" programs, distributed applications often involve data and control dependencies that need to be taken into account by the skeletons' evaluation mechanism. Our further objective is to present optimizations that are applicable for programs both with and without dependencies between tasks and data.

## 3. Skeleton-based Programming in Lithium

Our reference skeleton programming system is Lithium [14]. Lithium is a full Java library that provides the programmer with a set of parallel nestable skeletons. The skeletons implemented in Lithium include the `Farm` skeleton, modeling task farm computations, the `Pipeline` skeleton, modeling computations structured in independent stages, the `Map` skeleton, modeling data parallel computations with independent subtasks and the `DivideConquer` skeleton, modeling divide&conquer computations. All these skeletons process *streams* of *input tasks* to produce streams of *results*.

Figure 1 shows the structure of a typical Lithium application: the programmer first declares the program to be executed, then instantiates a controller/scheduler for the application; he requires the parallel execution of the program after providing the input data, and eventually processes the results computed.

Lithium implements the skeletons according to the *Macro Data Flow* (MDF) execution model [21]. Skeleton programs are first compiled into a data flow graph: each *Macro Data Flow instruction* (MDFi, i. e. ,each node of the MDF 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.

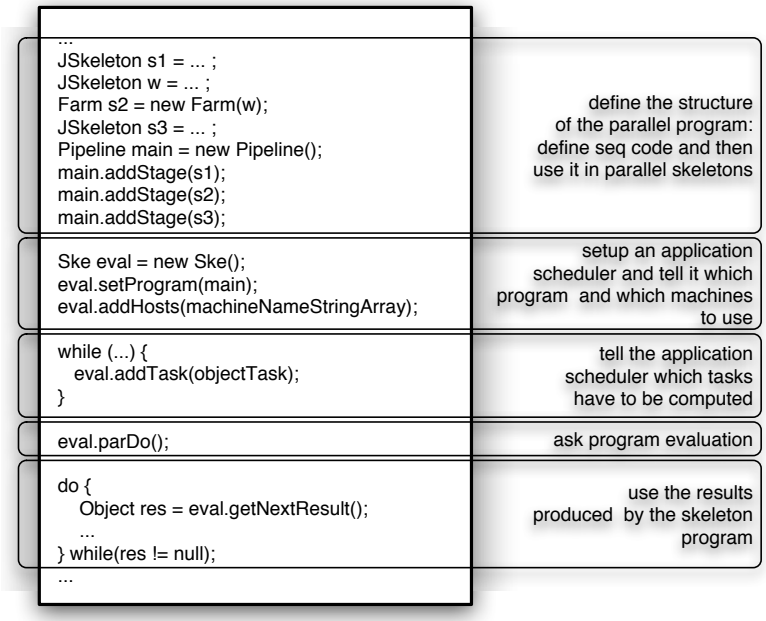


Figure 1. Sample Lithium code

An MDFi may represent large portions of code, rather than only simple operators or functions; therefore the term *Macro* Data Flow [22–25] is used.

The skeleton program is transformed into an MDF graph transparently to the programmer as a consequence of the `eval.setProgram` call. Lithium also allows to statically optimize the MDF graph associated with the skeleton program, such that a more compact and efficient MDF graph (the MDF "normal form") is actually executed. These optimizations are not taken into account in this work; we refer to [14,26] for further details on this subject.

The skeleton program is then executed by running the scheduler (`eval`) 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 execute 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, see Figure 2. The scheduler forks a control thread for each remote server. Such a control thread looks up a reference to one

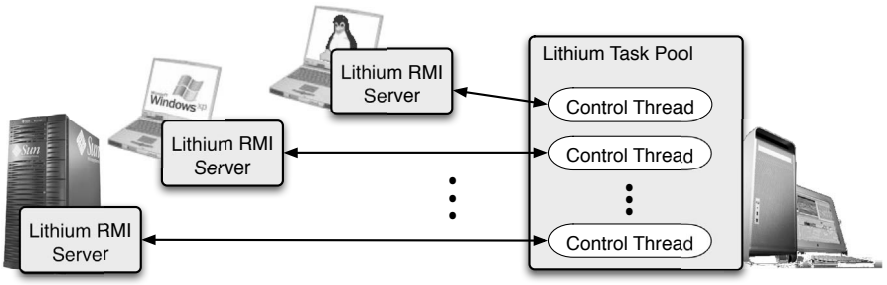


Figure 2. Lithium implementation outline.

server in the RMI-registry, by issuing a call to the static `lookup`-method of the standard `java.rmi.Naming`-class. 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 task pool that is joined to the scheduler on the local machine, asks the remote server to execute the MDFi and deposits the result in the task pool.

## 4. Future-Based RMI

Using the RMI (*Remote Method Invocation*) mechanism in distributed programming 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.

### 4.1. The Idea of Future-Based RMI

Since the RMI mechanism was developed for traditional client-server systems, it is not optimal for systems with several servers where server/server interaction is required. We illustrate this with an example of a 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, such 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, as shown in Figure 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.

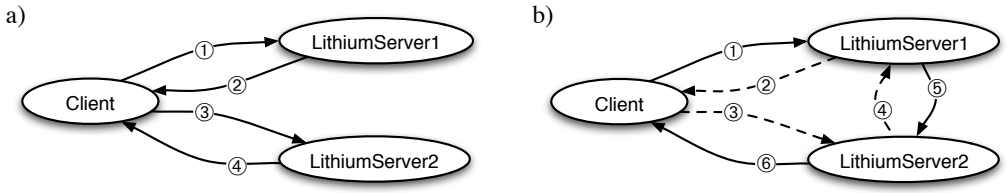


Figure 3. Method composition: a) using plain RMI, and b) using future-based RMI.

To eliminate this overhead, we have implemented a *future-based RMI*, based on the concepts that have been proven useful for other programming languages and communication technologies [27–29]. As shown in Figure 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 the (future) execution result (②). This future reference is then 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 the completion of the second method, the result is sent to the client (⑥).

Compared with traditional 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, whereas remote data itself is communicated directly between the servers. Moreover, communications and computations overlap, thus hiding latencies of remote calls.

### 4.2. Implementation of Future-Based RMI

In future-based RMI, a remote method invocation does not directly return the result of the computations. Remote hosts immediately start the execution of each call, but rather than waiting for its completion, an opaque object representing a (remote, future) reference to the result is returned. The opaque object has type `RemoteReference`, and provides two methods:

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

Let us suppose that `fut` is a `RemoteReference` object. The `fut.setValue(o)` method call triggers the availability of the result 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 call 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:

1. 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.

2. 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. 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. A remote call for retrieving a value from a `RemoteReference`, is only executed, if the object holding the value is actually located at a remote server.

Internally, remote references handle the availability of results using a simple boolean flag. Once a `RemoteReference`'s `setValue` method was called for the first time, this object's `getValue`-method will not wait anymore. It is in general not a good practice to issue subsequent calls to `setValue` for resetting `RemoteReference`s, because `RemoteReference`s can not be cached properly using a simple hash table when they are assigned more than once, nor does the class provide any kind of multistage notification mechanism. As `RemoteReference`s are cached by the servers, they are not garbage collected automatically. The scheduler explicitly causes a permanent deletion of a `RemoteReference`, once the corresponding MDFi has no more successors in the MDF graph, i. e. there is no remote post-processing necessary

## 5. Optimization Techniques Applied to Skeletons

In this section, we describe three optimization techniques for the Lithium skeleton system, which are based on the future-based RMI mechanism 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 benefit from them.

### 5.1. 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 scheduler 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 reference to the result. This is particularly important if we use multi-processor servers, because it allows the multithreaded implementation to exploit all available processors. 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 its execution by initializing the available servers and binding their names to the local `rmiregistry`. Then the servers wait for RMI calls. There are two kinds of calls that can be issued to a server:

- `setMDFGraph`, used within the `setProgram`-method of the scheduler to send a macro data flow graph to a server. This remote call happens transparently to the programmer, who uses `setProgram` as shown in Section 3. The information in the transferred 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. Also this remote method is never called by the programmer directly, but it is called by the control threads during the evaluation of the `parDo`-call shown in Section 3.

In the original Lithium, each control thread performs the following loop [14]:

```
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 Figure 4. Therefore, each Lithium server has an idle time between the execution of two consecutive tasks:

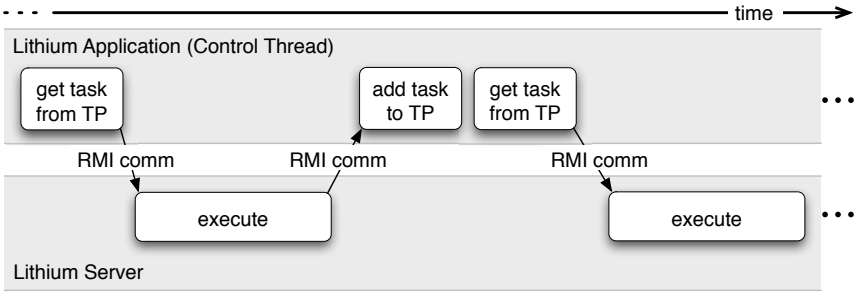


Figure 4. Server's idle time in original Lithium implementation.

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, extracting as many new fireable tasks from the task pool as currently available and assigning them to the same server. Simultaneously, some of the server's threads may be still running on previous tasks. As we shall see in Section 5.3, the amount of tasks looked ahead can be bounded within a heuristically chosen range in order to prevent load unbalance among servers.

As a result, we can have many threads running on 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 server with running its control thread. Since having multiple threads per server also increases the control thread pressure, the MDFi computational grain should be not too small w.r.t. communication grain. MDFi computational grain directly depends on code enclosed within sequential skeletons (run method of `JSkeleton` coded by the programmer), while communication grain depends on their input and output data size. In general, finding a communication/computation balance is the programmer's responsibility. However, even if this has not been done, Lithium can coarsen the computational grain in the server by collapsing many MDFi(s) into a single one. This transformation converts tokens passing thru collapsed MFDi (i.e. communications) in local memory operations thus coarsening computational grain. The full description of this technique (a.k.a task normalization) is beyond the scope of this paper [14,26].

### 5.2. Server-to-Server Lazy Binding

Our second optimization technique is called "lazy binding": a remote server will only execute a new MDFi from the graph if necessary; analogously, the scheduler will not wait for a remote reference to produce the future result.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 will identify chains in which each task depends on previous ones and make sure that such sequences will be dispatched to a single remote machine. Thus, a sequence of dependent tasks will be processed locally on one server, which leads to a further reduction of communication.

If no post-processing of an MDFi is specified in the MDF graph, then it does not need to be cached anymore. Once such an MDFi has been processed, the associated `RemoteReference` is removed from the cache by the server, which makes it eligible for garbage collection.

Let us consider again the evaluation of the sequence of two functions, $f$ and $g$, on a stream of data as in our first example in Section 3. The program can be expressed using the `Pipeline` skeleton, whose two stages evaluate $f$ and $g$, respectively. The behavior of the original Lithium system on this program is shown in Figure 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 executes 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).

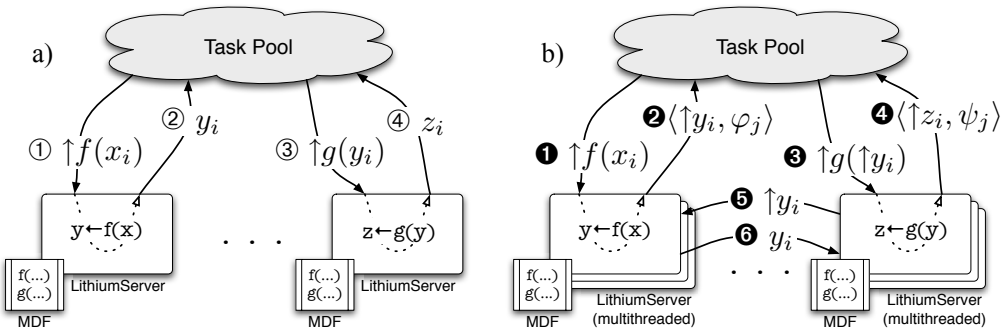


Figure 5. Communications between the Task Pool and Servers. a) Original Lithium. b) Optimized implementation using lazy binding.

The goal of the lazy binding optimization is reducing the size of communications ② and ③. Without lazy binding both, the reference to the function to be executed and its input data are transmitted in these communications, 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, then we move the values directly between servers, thus halving the number of large-size communications.

The optimized behavior is shown in Figure 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 $\varphi_j$ 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; it 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 `getValue()` on $\uparrow y_i$ (❺).

(vi) 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.

**Lazy Binding for Data-Parallel Computations.**

The execution of skeletons like `Map` or `Farm` with no dependencies between data elements are performed as data-parallel computations, i.e. these skeletons are parallelized by partitioning the input data and processing all partitions in parallel, which is carried out in Lithium as follows:

- a task $x$ 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 Figure 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 lazy-binding optimization implies the following behavior (Figure 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. Thereby, the dispatcher assigns the gathering task to the server that already holds the largest number of references. When this instruction is computed, the result is sent back to the task pool.

Thus, we avoid moving the intermediate results back and forth between the task pool threads and the servers in both the execution and gathering phases.

### 5.3. Load-Balancing

In this section, we describe how the load-balancing mechanism of Lithium is adapted to the optimized evaluation mechanisms presented above, 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, so that the 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

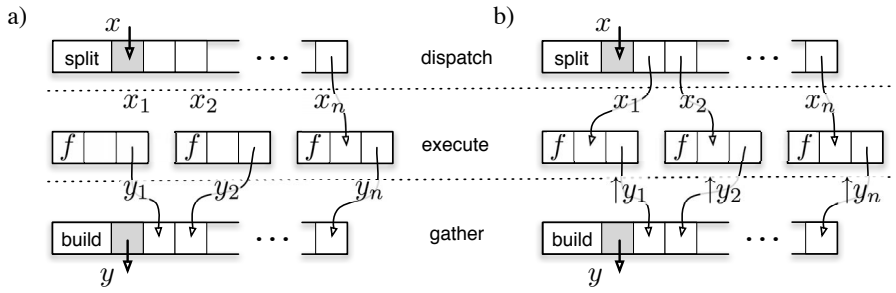


Figure 6. Executing a data-parallel skeleton: a) original Lithium; b) optimized.

machines in a Grid are usually heterogeneous. Without a suitable load-balancing strategy, this may lead to an inefficient partitioning of work.

To balance the load in the system, we measure the current load of each grid server. One possibility would be to use a new remote method, which, however, implies additional remote communication. Instead, we exploit the fact that the scheduler already communicates frequently with the remote servers when it dispatches tasks. We extend communicated data records by a value 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 as shown in Figure 7: Here, we have a maximum number of six active threads per server. Dispatching tasks to server 1 and server n yields the actual work-load (five active threads at server 1, six 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 the next task is sent. 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.

## 6. 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 Section 6.1.

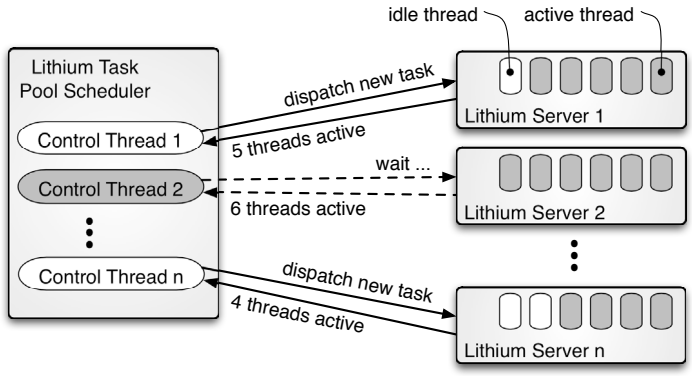


Figure 7. Communication schema for the load balancing mechanism.

(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 of Sun SMP machines in Berlin. Described in Section 6.2.

(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 several different Intel Pentium and Apple PowerPC computers, running Linux and Mac OS X respectively (The detailed configuration is shown in Figure 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 Section 6.3.

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 heterogeneous distribution of computing power and interconnection speed, typical for grids.

The image processing application we used for our tests employs 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 (available at http://jiu.sourceforge.net) that add a blur effect and an oil effect. Note that these are *area filter operations*, i.e. the computation of each pixel's color does not only impact its direct neighbors, but also an adjustable area of neighboring pixels. By choosing five neighboring pixels in each direction as filter workspaces, we made the application more complicated and enforced several iterations over the input data within each pipeline stage, which makes our filtering example a good representative for a compute intensive application.

Despite its simplicity, the application presented here, which is based on a single pipeline, represents a significant and complete test for our optimizations. Indeed, the Pipeline is

the simplest skeleton among all the skeletons offered by Lithium that exhibits control dependencies and a compositional behavior, like described in Figure 3 and Figure 5. This kind of interaction is typical for the MDF evaluation mechanism of Lithium and occurs in more sophisticated application repeatedly, but with no technical differences concerning the communication schema (see [14]). Actually, the only programs that do not interact like shown in Figure 3 and Figure 5 are applications composed of the plain Farm skeleton only, which can not benefit from Lazy Binding since it does not involve any Server-to-Server interaction.

Load-balancing for the future-based version was adjusted to the maximum of six concurrent threads per node. The lower limit was set to two threads. These values enable the exploitation of both task-lookahead optimization and thread parallelism on standard SMP systems, while not introducing excessive overhead due to multithreading management. All experiments were performed using the Sun J2SE Client VM SDK version 1.4.1.

As shown in the following sections, the optimized Lithium demonstrates a clear time advantage over the standard version along all tested configurations.

### 6.1. Dedicated Cluster (environment i)

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 plain Java sequential version of the application running on one node of the same cluster (execution time 446 Sec). The plots show that the future-based version performs approximately twice as fast as standard Lithium.

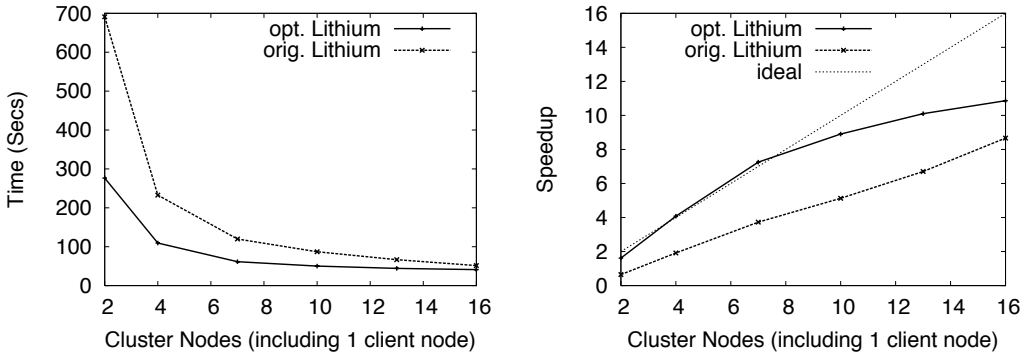


Figure 8. Measured execution times and speedup on the cluster (i).

In the example application, three kinds of activities take place: communication, computation, and data I/O. Task lookahead allows to overlap these activities, while lazy binding ensures that the whole data transfer between the stages of the pipeline takes place on the server side without client interaction.

### 6.2. Distributed Environment (ii)

Figure 9 shows the execution time for both the original Lithium and the optimized version running in the environment (ii). The tests demonstrate a clear increase in performance due to our optimizations, in particular to the lazy-binding mechanism.

By introducing server-to-server communication, the transmission of data between client and servers is reduced considerably. The fastest connections of the network (the interconnection between the servers in Berlin) are used for handling the biggest part of data exchange instead of tracing back each communication to the scheduler host with a much slower connection.

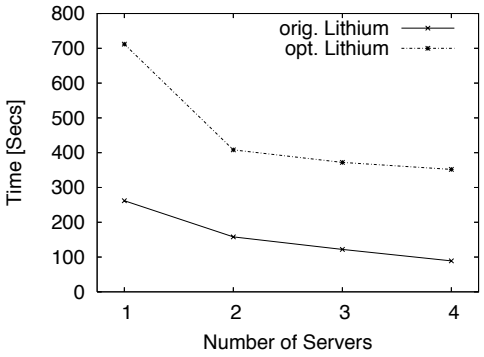


Figure 9: Run times in environment (ii)

### 6.3. Heterogeneous Environment (iii)

Figure 10 compares the optimized version against the standard one in the heterogeneous environment (iii), whose composition is listed in Figure 10 left. To take the varying computing power of different machines into account, the performance increase is documented by means of the *BogoPower* measure, which enables the comparison between application's actual parallel performance and the application ideal performance (Figure 10 right).

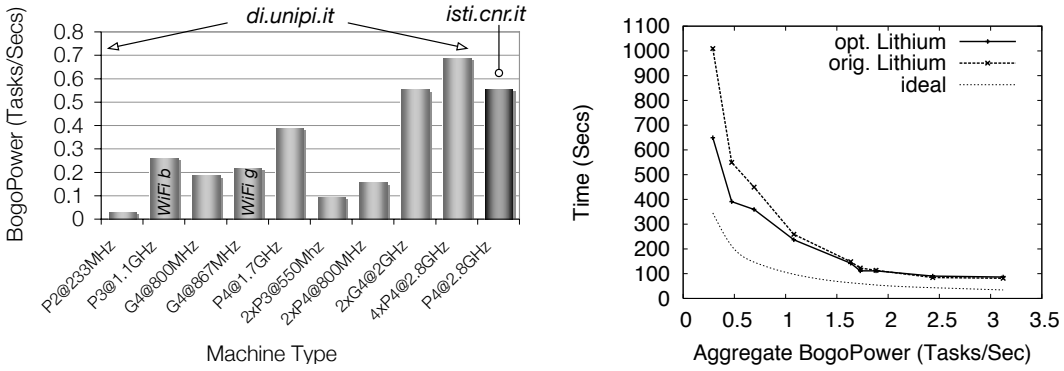


Figure 10. Left: Test environment (iii). Right: Execution times vs. increasing (Bogo-)powerful ordered server set.

Parallel speedup usually assumes that all machines in the running environment have the same computational power, which is often not true in grid-like environments. The BogoPower measure describes the aggregate BogoPower of a heterogeneous distributed system as the sum of individual BogoPower contributions (Figure 10 left). Application

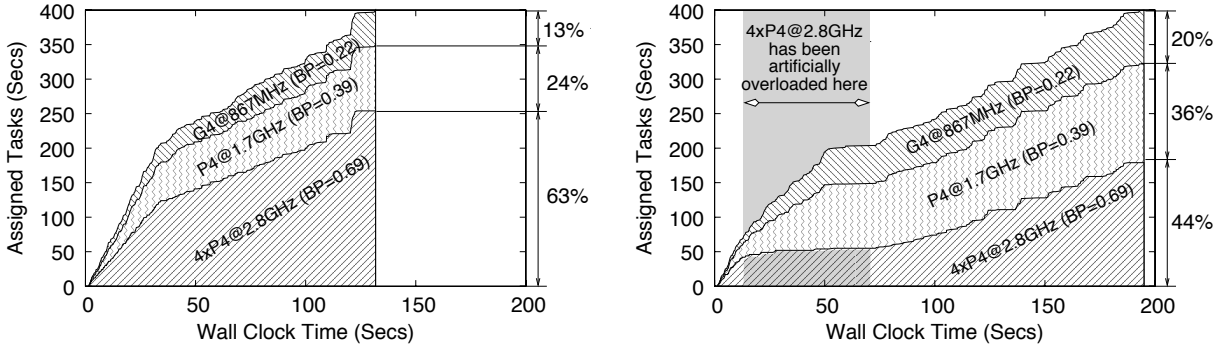


Figure 11. Detailed history of tasks scheduling. Left: All machines are dedicated to the experiment. Right: One of the machines is externally overloaded from time 10 to 70.

ideal performance curve is evaluated w.r.t. a system exploiting a given BogoPower rank, assuming both an optimal scheduling of tasks and zero communication costs.

We use environment (iii) for studying to what extent the heterogeneity challenge of grids is met by our improved load-balancing strategy. The Lithium runtime system continuously monitors the servers' states and raises or drops the number of threads correspondingly. To demonstrate the dynamic load-balancing behavior of the scheduler, we performed two identical experiments with differing load on one of the involved machines. First (Figure 11 left) all machines were dedicated to the experiment. Then, in the second experiment (Figure 11 right) fewer tasks are dispatched to the most powerful machine, because this machine was heavily loaded by another application.

Figure 12 shows the history of threads issued to the overloaded machine: When the machine load grows too much, the scheduler drops the number of active threads (which range from 2 to 12 in the experiment). This decision is supported by the system-wide historical statistics, maintained by the load-balancing module of the client.

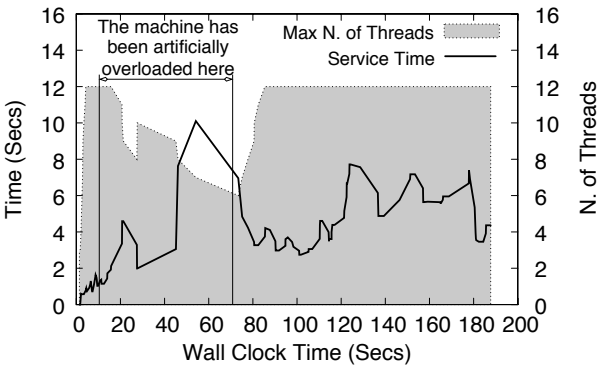


Figure 12: Throttling in task scheduling.

## 7. Related Work and Conclusions

In this paper, we have described three optimization techniques aimed at an efficient implementation of parallel skeletons in distributed grid-like environments with high communication latencies. As a reference implementation, we took the Lithium skeleton library.

We studied the effects of three 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 strategy to the multithreaded evaluation mechanism initiated by the task lookahead and implementing it without a large increase in remote communication: the messages carry one additional value, but the number of messages remains unchanged.

Several research projects investigated asynchronous RMI in Java [16,30,31]. In almost all of them, the programmer is responsible for writing the asynchronous invocations. In contrast, our future-based implementation encapsulates the modified communication schema into the skeleton evaluation mechanisms of the Lithium system, without demanding from the programmer to take care of the internal changes. Internally, our implementation uses the future-mechanism introduced in [16], but these core methods could also be replaced by e.g. ProActive that implements concepts described in [30], without affecting the high-level interface provided by Lithium. In [32], RMI calls are optimized using call-aggregation, where a server can directly invoke methods on another server. While this approach optimizes RMI calls by reducing the amount of communicated data, the method invocations are not asynchronous as in our implementation: they are delayed to find as many optimization possibilities as possible.

All three optimization 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 optimizations can easily be applied to other skeleton programming environments than Lithium. Furthermore, they are not restricted to RMI as a communication mechanism.

The broad applicability of our optimizations is confirmed by the fact that currently we are considering the adoption of these techniques in ASSIST [33], a system that exploits in part the experiences gained from Lithium and that runs upon different kinds of modern grid middleware (plain TCP/IP and POSIX processes/threads, CORBA and the Globus Toolkit [34]).

**Acknowledgments**

We are grateful to the anonymous referee of the draft version of this paper for very helpful remarks, and to Julia Kaiser-Mariani for her help in improving the presentation. This research was partially supported by the FP6 Network of Excellence *CoreGRID* funded by the European Commission (Contract IST-2002-004265) and by a travel grant from the German-Italian exchange programme *Vigoni*.

## REFERENCES

1. D. L. Parnas, On the design and development of program families, IEEE Trans. on Software Engineering SE-2 (1) (1976) 1–9.
2. M. Cole, S. Gorlatch, J. Prins, D. Skillicorn (Eds.), High Level Parallel Programming: Applicability, Analysis and Performance, Dagstuhl-Seminar Report 238, Schloß Dagstuhl, 1999.
3. M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computations, Research Monographs in Parallel and Distributed Computing, Pitman, 1989.

4. F. A. Rabhi, S. Gorlatch (Eds.), Patterns and Skeletons for Parallel and Distributed Computing, Springer-Verlag, 2002.
5. M. Danelutto, M. Stigliani, SKElib: parallel programming with skeletons in C, in: A. Bode, T. Ludwing, W. Karl, R. Wismüller (Eds.), Proc. of Euro-Par 2000, no. 1900 in LNCS, Springer-Verlag, 2000, pp. 1175–1184.
6. J. Sérot, D. Ginhac, Skeletons for parallel image processing: an overview of the SKiPPER project, Parallel Computing 28 (12) (2002) 1685–1708.
7. M. Cole, Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming, Parallel Computing 30 (3) (2004) 389–406.
8. S. Crocchianti, A. Laganà, L. Pacifici, V. Piermarini, Parallel skeletons and computational grain in quantum reactive scattering calculations, in: G. R. Joubert, A. Murli, F. J. Peters, M. Vanneschi (Eds.), Parallel Computing: Advances and Current Issues. Proceedings of the International Conference ParCo2001, Imperial College Press, 2002, pp. 91–100.
9. M. Coppola, M. Vanneschi, High performance data mining with skeleton-based structured parallel programming, Parallel Computing 28 (5) (2002) 793–813.
10. G. Sardisco, A. Machì, Development of parallel paradigms templates for semi-automatic digital film restoration algorithms, in: G. R. Joubert, A. Murli, F. J. Peters, M. Vanneschi (Eds.), Parallel Computing: Advances and Current Issues. Proceedings of the International Conference ParCo2001, Imperial College Press, 2002, pp. 498–509.
11. A. Giancaspro, L. Candela, E. Lopinto, V. A. Lorè, G. Milillo, SAR images co-registration parallel implementation, in: Proc. of the International Geoscience and Remote Sensing Symposium and the 24th Canadian Symposium on Remote Sensing (Igarss 2002), IEEE, 2002.
12. P. D'Ambra, M. Danelutto, D. di Serafino, M. Lapegna, Integrating MPI-based numerical software into an advanced parallel computing environment, in: Proc. of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing, IEEE, 2003, pp. 283–291.
13. J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters., in: 6th Symp. on Operating System Design and Implementation (OSDI 2004), 2004, pp. 137–150.
14. M. Aldinucci, M. Danelutto, P. Teti, An advanced environment supporting structured parallel programming in Java, Future Generation Computer Systems 19 (5) (2003) 611–626.
15. C. Nester, R. Philippsen, B. Haumacher, A more efficient RMI for Java, in: Proc. of the Java Grande Conference, ACM, 1999, pp. 152–157.
16. M. Alt, S. Gorlatch, Future-based RMI: Optimizing compositions of remote method calls on the grid, in: H. Kosch, L. Böszörményi, H. Hellwagner (Eds.), Proc. of the Euro-Par 2003, no. 2790 in LNCS, Springer, 2003, pp. 427–430.
17. H. Kuchen, A skeleton library, in: B. Monien, R. Feldmann (Eds.), Proc. of Euro-Par 2002, no. 2400 in LNCS, Springer-Verlag, 2002, pp. 620–629.
18. S. Pelagatti, Structured Development of Parallel Programs, Taylor&Francis, 1998.
19. I. Foster, C. Kesselmann (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 1998.

20. S. Gorlatch, Optimizing compositions of components in parallel and distributed programming, in: Christian Lengauer et al. (Ed.), Domain-Specific Program Generation, Vol. 3016 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 274–290.
21. M. Danelutto, Efficient support for skeletons on workstation clusters, Parallel Processing Letters 11 (1) (2001) 41–56.
22. Ö. Babaoglu, L. Alvisi, A. Amoreso, R. Davoli, L. A. Giachini, Paralex: An environment for parallel programming in distributed systems, in: 6th ACM International Conference on Supercomputing, 1992.
23. R. F. Babb, Parallel processing with large-grain data flow techniques, in: IEEE Computer, 1984, pp. pp. 55–61.
24. A. S. Grimshaw, Object-oriented parallel processing with Mentat, Information Sciences 93 (1) (1996) 9–34.
25. V. Sarkar, J. Hennessy, Compile-time partitioning and scheduling of parallel programs, in: Proceedings of the 1986 SIGPLAN symposium on Compiler contruction, ACM Press, 1986, pp. 17–26.
26. M. Aldinucci, M. Danelutto, Stream parallel skeleton optimization, in: Proc. of the 11th IASTED Intl. Conference on Parallel and Distributed Computing and Systems (PDCS'99), IASTED/ACTA press, Cambridge, MA, USA, 1999.
27. A. L. Ananda, B. H. Tay, E. K. Koh, A survey of asynchronous remote procedure calls, ACM SIGOPS Operating Systems Review 26 (2) (1992) 92–109.
28. K. A. Hawick, H. A. James, A. J. Silis, D. A. Grove, K. E. Kerry, J. A. Mathew, P. D. Coddington, C. J. Patten, J. F. Hercus, F. A. Vaughan, DISCWorld: An Environment for Service-Based Metacomputing, Future Generation Computer Systems 15 (5) (1999) 623–635.
29. B. Liskov, L. Shrira, Promises: linguistic support for efficient asynchronous procedure calls in distributed systems, in: Proc. of the ACM SIGPLAN conference on Programming Language design and Implementation, ACM Press, 1988.
30. D. Caromel, A general model for concurrent and distributed object-oriented programming., SIGPLAN Notices 24 (4) (1989) 102–104.
31. D. Caromel, W. Klauser, J. Vayssiere, Towards seamless computing and metacomputing in Java, Concurrency Practice and Experience (1998) 10 (11–13).
32. K. C. Yeung, P. H. J. Kelly, Optimising Java RMI programs by communication restructuring, in: D. Schmidt, M. Endler (Eds.), Middleware 2003: ACM/IFIP/USENIX Intl. Middleware Conference, Springer-Verlag, 2003.
33. M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, C. Zoccolo, ASSIST as a research framework for high-performance Grid programming environments, in: J. C. Cunha, O. F. Rana (Eds.), Grid Computing: Software environments and Tools, Springer, 2005, (to appear, draft available as University of Pisa Tech. Rep. TR-04-09).
34. I. Foster, C. Kesselman, J. M. Nick, S. Tuecke, Grid services for distributed system integration, Computer 35 (6) (2002) 37–46.
URL `http://www.computer.org/computer/co2002/r6037abs.htm;`