A Transparent non-Invasive File Data Model for Algorithmic Skeletons

Denis Caromel, and Mario Leyton
INRIA Sophia-Antipolis, Université de Nice Sophia-Antipolis, CNRS - I3S
2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France
First.Last@sophia.inria.fr

Abstract

A file data model for algorithmic skeletons is proposed, focusing on transparency and efficiency. Algorithmic skeletons correspond to a high-level programming model that takes advantage of nestable programming patterns to hide the complexity of parallel/distributed applications.

Transparency is achieved using a workspace factory abstraction and the proxy pattern to intercept calls on `File` type objects. Thus allowing programmers to continue using their accustomed programming libraries, without having the burden of explicitly introducing non-functional code to deal with the distribution aspects of their data.

A hybrid file fetching strategy is proposed (instead of lazy or eager), that takes advantage of annotated functions and pipelined multithreaded interpreters to transfer files in-advance or on-demand. Experimentally, using a BLAST skeleton application, it is shown that the hybrid strategy provides a good tradeoff between bandwidth usage and CPU idle time.

Keywords: Algorithmic skeletons, file transfer, transparency, aspect-oriented programming, aspects

1 Introduction

Scientific and engineering applications that require, handle, and generate large amounts of data represent an important part of distributed applications. For example, some of the areas that require handling large amounts of data are: bioinformatics, high-energy physics, astronomy, etc.

In this paper we focus on the integration of data abstractions with a high level programing model: *algorithmic skeletons*. We address the data problem by considering usability and performance from the programming model perspective.

Algorithmic skeletons (*skeletons* for short) are a high level programming model for parallel and distributed computing, introduced by Cole in [18]. Skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by nesting the basic ones.

To write an application programmers must compose the skeleton pattern of their program, and fill the skeleton with the muscle functions specific to their application. The skeleton pattern implicitly defines the parallelization and distribution aspects, while the muscle functions provide the application's specific functional aspects (i.e. business logic). As a result, skeletons achieve a natural separation of parallelization and functional aspects.

The support of file data access has been overlooked in many skeleton frameworks such as Eden [34], eSkel [9], JaSkel [25], Lithium [6, 21], Muesli [28, 29], Muskel [20], Skil [11]. Most of them could be enhanced with file data support by addressing file distribution aspects from inside muscles, as is the case with ASSIST [2]. Nevertheless, this strategy leads to the tangling of non-functional code (data distribution) with the functional code (business logic).

Therefore, the integration of data files with algorithmic skeletons must be achieved in a transparent non-invasive manner, as not to tangle data distribution with functional concerns, while also taking efficiency into consideration.

Non-invasive transparency Programmers should not have to worry about data location, movement, or storage. Furthermore, programmers should not have to change their standard way of working with data. This means that transparency should be non-invasive, i.e. without imposing an ad hoc language nor library.

Efficiency is a double edged problem: computation and bandwidth. A suitable approach must balance the tradeoff between idle computation time, and bandwidth usage.

This paper is organized as follows. Section 3 describes an algorithmic skeleton's programming model. Section 4 presents the file data model for algorithmic skeletons. Section 5 studies efficiency concerns using BLAST as a case study. Section 6 relates our approach with aspect-oriented

programming. Finally, section 7 provides the conclusions and future work.

2 Related Work

Since most skeleton frameworks do not provide file data transfer support, in the first part of this section we review how file transfer has been addressed in workflows. In the second part of this section we review the approach introduced by ASSIST to support file transfer in algorithmic skeletons.

Workflows are another high-level programming model for parallel and distributed computation. Workflow programing models usually provide abstractions to access data inside workflow units, and to transfer data between workflow units. For example, Java CoG Kit's [39] data transfer operations are explicitly defined like any other task, in the sense that a data transfer operation must be submitted for execution as a data-transfer-task [38, 40]. Another example is Unicore [24, 37], which uses a workflow programing model to order dependencies between tasks. All tasks belonging to the same job share a jobspace file system abstraction. The job description also specifies which files must be imported into the jobspace before the execution of the job, and which files must be exported after the job is finished. Files that must be imported and exported to the jobspace are staged before and after the job begins. Additionally, it is also possible to interact with sub-jobs (which have their own jobspace) by explicitly adding file transfer modules in the workflow. The file transfer modules handle the input and output of files between the jobspace and the sub-jobspaces.

Thus workflows require programmers to explicitly add data management units to their applications. Therefore, unlike worklflows, we would like to support data abstractions in skeletons transparently.

To our knowledge, the only other skeleton framework providing file data transfer support is ASSIST [2]. ASSIST provides programmers with a structured coordination language, which can express arbitrary graphs of software modules written in C++, interconnected by streams of data. AdHoc, a hierarchical and fault-tolerant DSM system is used to interconnect streams of data between processing elements by providing a repository with: get/put/remove/execute operations [1, 5, 30]. Research around AdHoc has focused on transparency, scalability, and fault-tolerance of the data repository.

Thus, throughout this paper we assume the existence of a *good* data repository providing basic operations and properties such as the ones described in AdHoc. On the other hand, while the view of ASSIST/AdHoc is to "*provide an abstract view of data, and a high-level API to access it*" [1], we feel programmers should not have to migrate to new abstractions with new APIs to manipulate data. Instead, we believe programmers should continue using, as much as possible, their accustomed data abstractions and APIs.

3 Algorithmic Skeletons in a Nutshell

As a skeleton framework we use Calcium [15], which is greatly inspired on Lithium [3, 4, 6, 21] and its successor Muskel [20]. Calcium is written in Java [32] and is provided as a library. To achieve distributed computation Calcium uses ProActive. ProActive is a Grid middleware [13] providing, among others, a deployment framework [8], a programming model based on active objects with transparent first class futures [12], and a data transfer model [7].

Basic task and data parallel skeletons supported in Calcium can be combined and nested in a type safe way [14], to solve more complex applications:

$$\begin{aligned}\triangle ::= & \; seq(f_e) \mid farm(\triangle) \mid pipe(\triangle_1, \triangle_2) \mid while(f_b, \triangle) \mid \\ & if(f_b, \triangle_{true}, \triangle_{false}) \mid for(i, \triangle) \mid map(f_d, \triangle, f_c) \mid \\ & fork(f_d, \{\triangle_i\}, f_c) \mid d\&c(f_d, f_b, \triangle, f_c)\end{aligned}$$

Where the task parallel skeletons are: seq for wrapping execution functions; $farm$ for task replication; $pipe$ for staged computation; $while/for$ for iteration; and if for conditional branching. The data parallel skeletons are: map for single instruction multiple data; $fork$ which is like map but applies multiple instructions to multiple data; and $d\&c$ for divide and conquer.

To program with algorithmic skeletons users have to define a nested skeleton pattern ($\triangle$), and provide the functional muscle codes specific to their problem:

$$\begin{aligned}f_b &: P \rightarrow boolean \\ f_c &: \vec{P} \rightarrow R \\ f_d &: P \rightarrow \vec{R} \\ f_e &: P \rightarrow R\end{aligned}$$

Where P is the parameter type, R the result type, and $\vec{X}$ a list of parameters or results.

Muscle functions (muscles for short) are black boxes to the skeleton language which will be invoked during the computation of the skeleton program. Muscles will be invoked, either sequentially or in parallel, in accordance with the defined skeleton pattern ($\triangle$). The result of a muscle is passed as a parameter to another muscle, until no further muscles have to be computed. When no further muscles have to be executed, the final result is returned to the user.

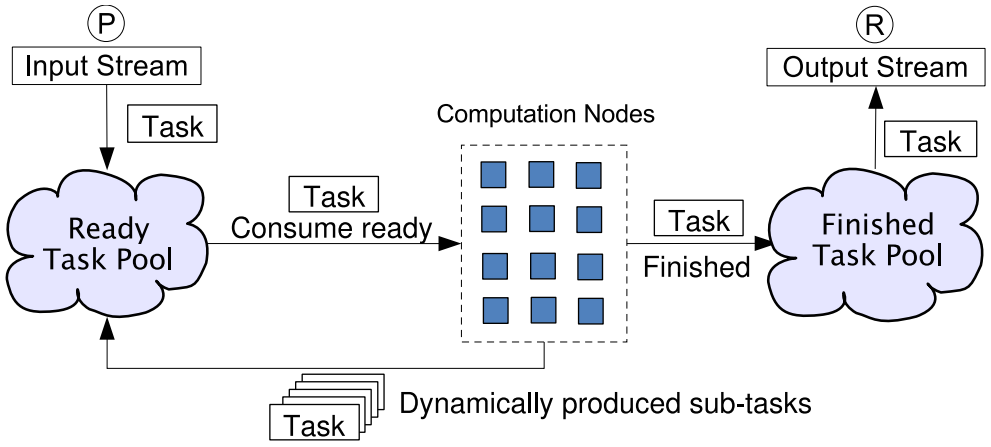


Figure 1. Calcium Framework Task Flow

3.1 Calcium Implementation

Internally in Calcium, a task abstraction is used to distribute and keep track of a program's execution. A task is mainly composed of a skeleton **instruction stack**, and the **state memory**.

The instruction stack is generated from the skeleton pattern ($\triangle$), and is capable of tracking the current execution of the program. Each skeleton instruction in the stack represents the weaving between the parallelism and the functional aspects of the program. When an instruction is popped from the stack its invocation can results in: the execution of a muscle and/or the addition of new instructions to the stack.

The state memory is the glue between the execution of muscles. The state memory is passed as parameter when a muscle is invoked and updated with the muscle's result.

The execution and distribution of the program is done in the following way. A task pool stores and keeps track of tasks. As shown in Figure 1, root-tasks are entered into the task pool by users who provide the initial state memory as a parameter. Interpreters consume tasks from the task pool, compute the tasks according to the skeleton instructions, and return the computed tasks to the task pool. Additionally, new tasks can be dynamically produced by the interpreters when data parallelism is encountered (map, $fork$, $d\&c$), in a similar fashion as in [34]. Dynamically produced tasks are referred to as sub-tasks, while the task that spawned them is referred to as the parent-task.

A task is finished when all of its sub-tasks are finished, and no further skeleton instructions need to be executed. When all sub-tasks are finished, they are returned to their parent-task for reduction. The parent-task may then continue with the execution of its own skeleton, and perhaps generate new sub-tasks. When a root-task reaches the finished state, its state memory can be delivered to the user as the final result.

3.2 Limitations on Data Size

The model presented in this section supposes that the data passed between muscles is small enough to be encapsulated inside tasks' state memory. This is suitable for transferring small amounts of data between muscles, as would be done in regular non-parallel programming. Nevertheless, when the size of the data is too big to hold in runtime memory, non-parallel programming uses secondary memory storage abstraction: files.

Therefore, skeleton programming requires a mechanism that allows programmers to use their standard non-parallel way of reading/writing files inside muscles (non-invasive). Which, at the same time, does not force programmers to specify code for transferring files; i.e. enables transparent and efficient support for transferring files between the execution of muscles. As we shall discuss in the following section, the complexity arises because, from the skeleton language perspective, muscles are black boxes.

4 File Transfer Model for Skeletons

4.1 Transparency with FileProxy

The Proxy Pattern [26] is used to achieve transparent access to files. Files are rendered accessible using the `FileProxy` object as shown in Figure 2. By intercepting calls at the proxy level, the framework is able to determine when a muscle is accessing a file. In a way, the `FileProxy` illuminates a specific aspect inside black box muscles.

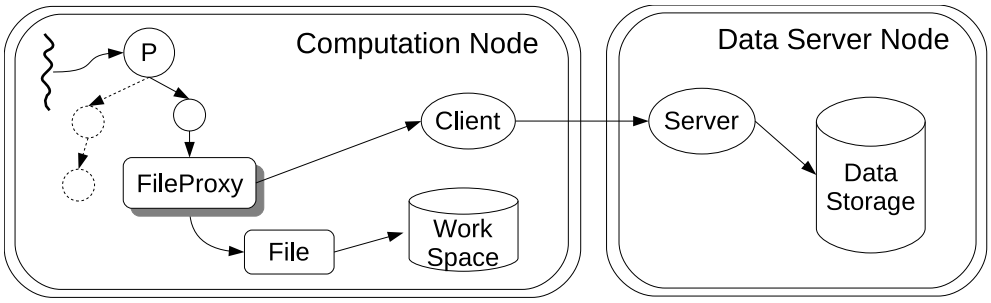


Figure 2. Proxy Pattern for Files

Figure 3 provides an example. When an interpreter thread invokes a muscle, all `File` type references inside P are indeed `FileProxy` instances. A `FileProxy` can transparently intercept a thread's access to the actual `File` object. A `FileProxy` can add new non-functional behavior such as caching of metadata (file names, size, etc...), transparent file fetching on demand, and blocking on a wait-by-necessity style [12]. Afterwards, the `FileProxy` can resume the thread's execution by delegating method calls to the *real* `File` object.

4.2 Stage-in and Stage-out

Listing 1 provides an example on the usage of Calcium. Line 1 defines the skeleton pattern, and is omitted here but detailed in Figure 5(a). Lines 3-4 instantiates an execution environment, which in this case corresponds to a `ProActiveEnvironment`, and creates the `Calcium` instance. The boot and shutdown of the framework are done in lines 6 and 23 respectively. Then in lines 8-9, a new input `Stream` is created with the blast skeleton pattern. Lines 12-15 illustrate the creation of a new `BlastQuery` paremeter, which receives three `File` type arguments: blast binary, query, and database files on the client machine.

The interesting part takes place in line 17. The `BlastQuery` is entered into the framework, and a `Future<File>` is created to hold the result once it is available. During the `input` process each file's data is remotely stored; and all `File` type objects are replaced by `FileProxy` instances, capable of fetching the data when required by remote nodes during the computation. Once the result is available, and before unblocking threads waiting on line 21, all remotely stored data referenced by `FileProxy` instances are copied to the client machine, and all `FileProxy` instances are replaced with regular `File` type instances. Hence, the result in line 21 is a regular `File` with its data stored on the client machine.

```
1 Skeleton skel = ...;

3 Environment env = new ProActiveEnvironment(...);
  Calcium calcium = new Calcium(env);
5
  calcium.boot();
7
    Stream<BlastQuery,File> stream =
9                     calcium.newStream(skel);

11  //Initial stage-in
    BlastQuery blast = new BlastQuery(
13                     new File("/home/user/blast.bin"),
                       new File("/home/user/query.dat"),
15                     new File("/home/user/db.dat"));

17  Future<File> future = stream.input(blast);

19  ...
    //Final stage-out, the file is locally available
21  File result = future.get();

23 calcium.shutdown();
```

Listing 1. Calcium Input and Output Example

4.2.1 Initial and Final Staging

In general, when a parameter P is submitted into the skeleton framework, as shown in Listing 1 (line 17), a `File` stage-in takes place. First, all references of type `File` in P's object graph are replaced with `FileProxy` references. Then, the files' data are stored in the data server. If a name clash occurs or a data transfer error takes place, an exception is immediately raised to the user, before the parameter is actually entered into the skeleton framework.

When the final result R has been computed, but before it is returned, a stage-out process takes place. Every reference of type `FileProxy` in R's object graph is replaced by a regular `File` type pointing to a local file, and the remote data is stored in the local file, before returning R to the user.

4.2.2 Intermediate Staging

Before an interpreter invokes a muscle, a staging process takes place on the interpreter nodes. If not already present,

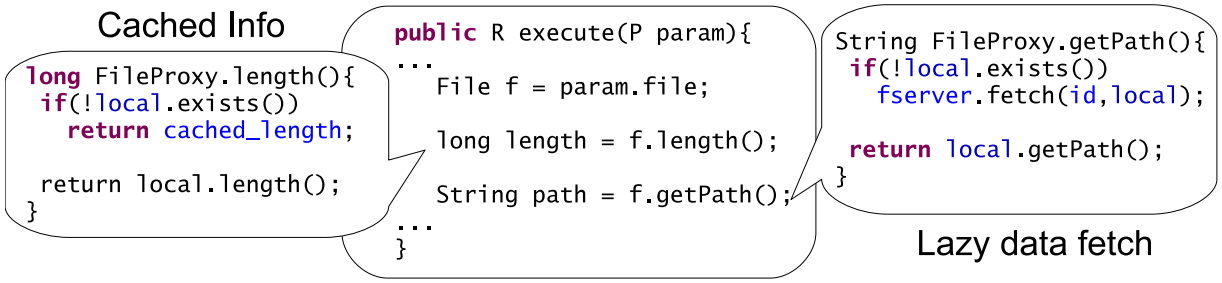


Figure 3. FileProxy Behavior Example

a unique and independent workspace is created. Then, depending on the desired behavior (see section 5) all, some, or none of the `FileProxy` type objects in P's object graph are downloaded into the workspace, and the `FileProxy` references are updated with the new location of the file.

After the invocation of a muscle, new files referenced in R's object graph, and present in the workspace, are stored on the data server. Actually, files are only stored on the data server if the file reference is passed on to other tasks, i.e. returned to the task pool. Further details of how the references are updated are discussed in Section 4.4.

4.3 The Workspace Abstraction

The workspace abstraction provides muscles with a local disk space on the computation nodes. If several muscles are executed simultaneously on the same interpreter node, each muscle is guaranteed to have its own independent workspace.

The workspace abstraction provides the following methods:

```
interface WSpace{
  public File newFile(String name);
  public void exec(File bin, String args);
}
```

Where the `WSpace.newFile()` factory can be used to create a file reference on the workspace, and `WSpace.exec(...)` can be used to execute a native command with a properly configured execution environment (e.g. current working directory).

Listing 2 provides an example. A muscle of type f_e : `BlastQuery` $\rightarrow$ `File` is shown. Lines 4-5 get a reference on the native `command` and its `arguments`. For the programmer, `command` is of type `File`, but is indeed a `FileProxy` instance. The `command`'s data was stored somewhere else during the computation (Listing 1 line 17), and is transparently made available on the interpreter node. Line 8 invokes the native blast command which outputs its results to a file named `result.blast`, located in some

```
1  public File execute(WSpace wspace, BlastQuery blast){
2
3    //Get parameters
4    File command = blast.blastProg;
5    String arguments = blast.getArguments();
6
7    //Execute the native blast in the wspace
8    wspace.exec(command, arguments);
9
10   //Create a reference to a file in the wspace
11   File result = wspace.newFile("result.blast");
12
13   return result;
14 }
```

Listing 2. Muscle Function Example

directory, specified by the workspace, on the interpreter node. Then line 11 uses the workspace factory to get a reference on the `result.blast` file. The workspace factory returns a reference object of type `File` which is indeed an instance of type `FileProxy`. Finally, line 13 returns the `File` object as a result. If the result is passed to another computation node, or delivered as final result to the user, then the file will be transparently transferred.

An alternative approach to providing a workspace factory method would have been to use other aspect-oriented programming [27] methodologies that, for example, manipulate Java bytecode to intercept calls on the `File` class constructor. Nevertheless, as noted by Cohen et al. [16], factories provide several benefits over traditional constructor anomalies.

After a `File` reference is created through the workspace abstraction, the framework transparently handles reference passing; creation, modification and deletion of file's data; and remote data storage/fetching.

4.3.1 Data Division

When data parallelism is encountered, such as in $\{d\&c, map, fork\}$ skeletons, new sub-tasks are spawned and assigned with a new workspace.

Instead of copying all of the original workspace's files

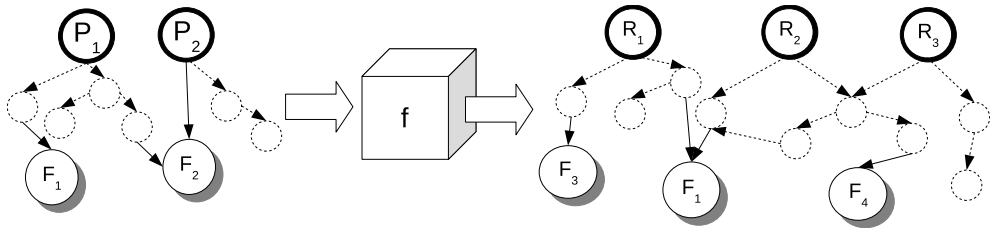


Figure 4. File Reference Passing Example

into each sub-task's workspace, only referenced files are copied. For example, if the muscle $f_d : P \rightarrow \vec{R}$ assigns at some point

$$R_i.file \leftarrow P.file_1$$
$$R_j.file \leftarrow P.file_2$$

then only $file_1$ will be copied into R_i's workspace, while $file_2$ will be copied into R_j's workspace.

The advantage of this approach is that the mapping of files with workspaces is transparent for the programmer. Contrary to what happens on workflow environments (see section 2), there is no need for the programmer to explicitly map which files are copied into which workspace. This is automatically inferred from the `FileProxy` references.

4.3.2 Data Reduction

The symmetrical case is the reduction (conquer) case, where several sub-tasks are reduced into a single one. This is done with a muscle of type $f_c : \vec{P} \rightarrow R$, which takes n object elements and returns a single one.

Before invoking the conquer muscle, a new workspace is created, and all the files referenced in $\vec{P}$ are copied into the new workspace. Nevertheless, a name space clash is likely to happen when two files originating from different workspaces have the same name.

A simple solution is to have a renaming function which provides a unique name when a name clash is detected. The clashing file is then renamed, and the `FileProxy` reference is transparently updated with the new name. While this solution can yield unexpected file renaming behavior for the programmer, no problems will be encountered as long as the programmer consistently uses the `File` references.

4.4 File References and Data

4.4.1 Storage Server

We assume the existence of a data storage server[1], capable of storing data, retrieving data, and keeping track on the reference count of each data. The storage server provides the following operations:

- $store(F_x, k) \rightarrow i$, stores the data represented in file F_x, with an initial reference count $k > 0$. The function returns a unique identifier for this data on the storage server.
- $retrieve(i) \rightarrow F_x$, retrieves the data located on the server and identified by i.
- $count(i, \delta) \rightarrow boolean$, updates de reference count by δ, and returns $true$ if the reference count is equal to or smaller than zero, and $false$ otherwise.

Once the reference count reaches zero for a file's data, no further operations are permitted on the data, and the server may delete the data at its own discretion.

4.4.2 Reference Counting

During the execution of a skeleton program, data can be created, modified, and deleted. Also, `File` references pointing to data can be created, deleted, and passed (copied). Therefore, it is up to the skeleton framework to provide support for these behaviors, by storing new/modified data; and keeping track of `File` references, to delete data when it is no longer accessible.

Consider the example shown in Figure 4, where $\{P_1, P_2\}$ are input parameters of a muscle $f : \vec{P} \rightarrow \vec{R}$, $\{R_1, R_2, R_3\}$ are the output results, and F_i are `FileProxy` references. We are interested on knowing, for a given F_i, how many P_j/R_k have a directed path from P_j/R_k to F_i, before/after the execution. We call this the reference count, and we write it as [before,after].

In the example, the reference counts are:

$$F_1 \rightarrow [1, 3] \qquad F_2 \rightarrow [2, 0]$$
$$F_3 \rightarrow [0, 1] \qquad F_4 \rightarrow [0, 2]$$

Thus we know that F_1 has incremented its reference count by 2; F_2 is no longer referenced and has decreased its reference count by 2; and F_3, F_4 are new files created inside f.

[1] For an example of a scalable data storage system refer to [1].

4.4.3 Update Cases

In general, after invoking a muscle f, a file F_x can be in one of the cases shown in Table 1.

Case	[Before,After]	Action
New	$[b = 0, a > 0]$	store(F_x, a) $\rightarrow i$
Normal	$[b > 0, a > 0]$	count(i, $a - b$)
Modified		count(i, $-b$) store(F_x, a) $\rightarrow j$
Dereferenced	$[b > 0, a = 0]$	count(i,$-b$)
Unreferenced	$[b = 0, a = 0]$	———

Table 1. File Scenarios after muscle invocation

Where the cases are described as follows:

- **New** files are created during the execution of f. A new file's data is uploaded to the storage server with its after reference count by invoking $store(F_x, k) \rightarrow i$, with $k = a$.
- **Normal** files only require an update on their reference count, since data has not been modified. This is done by invoking $count(i, \delta)$ with $\delta = b - a$.
- **Modified** files have been modified during the execution of f. Conceptually, modified files are treated as new files. Therefore if i is the identifier of the original file on the storage server, then $count(i, \delta)$ with $\delta = -b$ is invoked to discount the before references on the original file. Then, the modified file is treated as a new one, by uploading its data to the storage server and obtaining a new file identifier: $store(F_x, k) \rightarrow j$ with $k = a$.
- **Dereferenced** files have no references after the execution of f, and therefore it is irrelevant if the file was modified during the execution. Thus they only require a $count(\delta)$ on the server, with $\delta = -b$.
- **Unreferenced** files are temporary files used inside f, and can be locally deleted from the workspace after the execution of f.

5 Efficiency

An efficient approach minimizes both bandwidth usage and CPU idle time (blocked waiting for data). To minimize the CPU idle time, a file's data should already be locally available when a muscle wants to access it. On the other hand, to minimize the bandwidth usage, a file's data should only be transferred if it is going to be used by the muscle. This presents a problem since muscles are black boxes.

Given a three staged pipeline on each interpreters where: the first stage is the *prefetch*, which downloads candidate files in advance; the *compute* stage invokes the muscles; and the *store* stage uploads files' data to the storage server. Thus, in any given moment, three tasks can be present on an interpreter pipeline performing different aspects: download, computation, and upload.

Thus we can identify two strategies, a **lazy** strategy which transfers a file's data on demand using the `FileProxy` (bandwidth friendly), and an **eager** strategy which transfers all the files' data in advance (CPU friendly) using the interpreter pipeline. Additionally, we propose a third **hybrid** strategy which uses annotated muscles to decide which file's data to transfer in advance.

For example, a muscle can be annotated to prefetch files matching a regular expression pattern or files bigger/smaller than a specified size:

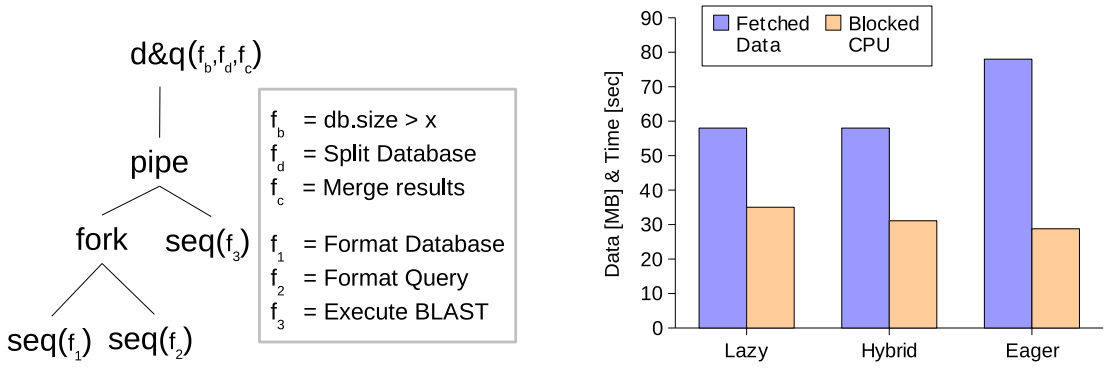
```
@PrefetchFilesMatching(name="db.*|query.*",
                       sizeMin=10000,
                       sizeMax=20000)
public File execute(WSpace wspace, BlastQuery param){
  ...
}
```

While the separation of concerns is kept using the proposed annotation, one may argue that the transparency of the approach is hindered. Nevertheless, it is important to emphasize that the annotation is not a file transfer definition (source and destination are not specified), and as such does not fall back into the non-transparent case. Furthermore, the presence of the annotation is not mandatory, being its only objective the improvement of performance.

5.1 BLAST Case Study

BLAST [10] corresponds to Basic Local Alignment Search Tool. It is a popular tool used in bioinformatics to perform sequence alignment of DNA and proteins. In short, BLAST reads a query file and performs an alignment of this query against a database file. The results of the alignment are then stored in an output file. BLAST is a good case study because it performs intensive data access, computation, and requires the execution of native code.

A BLAST parallelization using skeleton programming is shown in Figure 5(a). The strategy is to divide the database until a suitable size is reached and then merge the results of the BLAST alignment. The result of applying lazy, hybrid and eager strategies are shown in Figure 5(b). The figure shows that a lazy strategy performs the least amount of data transfers, but blocks the application for the longest time waiting for the data. On the other hand, an eager strategy performs the most data transfer, blocking the application for the least time.



(a) BLAST Skeleton Program

(b) Bandwidth / CPU Tradeoff

Figure 5. BLAST Case Study

For BLAST, a good tradeoff can be reached using the proposed hybrid strategy, which can transfer as few data bytes as the lazy strategy, and block the application at least as the eager strategy. In general, the performance of the hybrid strategy may vary, depending on the application, but the hybrid strategy's performance is bounded by the lazy and eager strategies.

6 Discussion on Skeletons and AOP

Readers familiar with Aspect-Oriented Programming (AOP) [27] will have noticed that many of the techniques used in this paper resemble those of AOP.

Indeed, the idea of weaving non-functional aspects using inheritance [19], in the same way that the `FileProxy` abstraction has been used to intercept calls on `File` type objects is not new. The dilemma of instantiating aspect augmented objects has been addressed in AOP using factories [16] in similar fashion as the workspace abstraction factory introduced in section 4.3. And the transformation of `File` → `FileProxy` → `File`, can be framed in the domain of dynamic aspects [35] and object reclassification [22, 23].

From the AOP perspective, this paper has provided a specific methodology for weaving file transfer aspects with algorithmic skeletons, and as such has shown that AOP like methodologies can be applied to algorithmic skeletons. More generally the integration of AOP with distributed programming has already been proposed for other middlewares such as JAC[33], J2EE [17], ReflexD [36], etc.

Therefore, as [31], we believe that the integration of AOP with algorithmic skeleton is a promissing mechanism to support other non-functional aspects in skeleton programming.

7 Conclusions and Future Work

This paper has proposed a file data access model for algorithmic skeletons by focusing on transparency and efficiency.

Transparency is achieved using a workspace abstraction and the Proxy pattern. A `FileProxy` type intercepts calls on the real `File` type objects, providing transparent access to the workspace. Thus allowing programmers to continue using their accustomed programming libraries, without having the burden of explicitly introducing non-functional code to deal with the distribution aspects of their data.

From the efficiency perspective we have proposed a hybrid approach that takes advantage of annotated muscle functions and pipelined interpreters to transfer files in advance, but can also transfer the file's data on-demand using the `FileProxy`. We have experimentally shown with a BLAST skeleton, that a hybrid approach provides a good tradeoff between bandwidth usage and CPU idle time.

As current and future work we are working on a generalization of the methodologies presented in this paper to support other non-functional aspects in algorithmic skeletons. Indeed, our goal is to provide an AOP model for algorithmic skeleton, which will allow a tailored integration of other non-functional aspects into skeletons, such as stateful muscles, statistics gathering, event dispatching, etc.

Acknowledgments: This work was partially funded by CONICYT Chile, CoreGrid EU Project (FP6-004265), and GridComp EU Project (FP6-034442)

References

[1] Marco Aldinucci, Gabriel Antoniu, Marco Danelutto, and Mathieu Jan. Fault-tolerant data sharing for high-level grid programming: A hierarchical storage architecture. In Marian Bubak, Sergei Golatch, and Thierry Priol, editors, *Achievements in European Research on Grid Systems*, CoreGRID Series, pages 67–82. Springer-Verlag, 2007.

[2] Marco Aldinucci, Massimo Coppola, Marco Danelutto, Nicola Tonellotto, Marco Vanneschi, and Corrado Zoccolo. High level grid programming with ASSIST. *Computational Methods in Science and Technology*, 12(1):21–32, 2006.

[3] Marco Aldinucci and Marco Danelutto. Stream parallel skeleton optimization. In *Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems*, pages 955–962, Cambridge, Massachusetts, USA, November 1999. IASTED, ACTA press.

[4] Marco Aldinucci, Marco Danelutto, and Jan Dünnweber. Optimization techniques for implementing parallel skeletons in grid environments. In S. Gorlatch, editor, *Proc. of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, pages 35–47, Stirling, Scotland, UK, July 2004. Universität Münster, Germany.

[5] Marco Aldinucci, Marco Danelutto, Gianni Giaccherini, Massimo Torquati, and Marco Vanneschi. Towards a distributed scalable data service for the grid. In G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, and E. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing (Proc. of PARCO 2005, Malaga, Spain)*, volume 33 of *NIC*, pages 73–80, Germany, December 2005. John von Neumann Institute for Computing.

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

[7] Françoise Baude, Denis Caromel, Mario Leyton, and Romain Quilici. Grid file transfer during deployment, execution, and retrieval. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4276 of *Lecture Notes in Computer Science*, pages 1191–1202. Springer-Verlag, 2006.

[8] Françoise Baude, Denis Caromel, Lionel Mestre, Fabrice Huet, and Julien Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[9] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible skeletal programming with eskel. In *11th International Euro-Par Conference: Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 761–770. Springer-Verlag, 2005.

[10] BLAST. Basic local alignment search tool. http://www.ncbi.nlm.nih.gov/blast/.

[11] G. H. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing*, pages 718–731, London, UK, 1996. Springer-Verlag.

[12] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[13] Denis Caromel, Christian Delbé, Alexandre di Costanzo, and Mario Leyton. Proactive: an integrated platform for programming and running applications on grids and p2p systems. *Computational Methods in Science and Technology*, 12, 2006.

[14] Denis Caromel, Ludovic Henrio, and Mario Leyton. Type safe algorithmic skeletons. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE, 2008.

[15] Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In *13th International Euro-Par Conference: Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 2007.

[16] Tal Cohen and Joseph Gil. Better construction with factories. *Journal of Object Technology*, 6(6):109–129, 2007.

[17] Tal Cohen and Joseph (Yossi) Gil. Aspectj2ee = aop + j2ee: Towards an aspect based, programmable and extensible middleware framework. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 219–243. Springer-Verlag, 2004.

[18] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[19] Constantinos A. Constantinides, Tzilla Elrad, and Mohamed E. Fayad. Extending the object model to provide explicit support for crosscutting concerns. *Softw. Pract. Exper.*, 32(7):703–734, 2002.

[20] Marco Danelutto and Patrizio Dazzi. Joint structured/unstructured parallelism exploitation in Muskel. In *Proc. of ICCS 2006 / PAPP 2006*, LNCS. Springer Verlag, May 2006. to appear.

[21] Marco Danelutto and Paolo Teti. Lithium: A structured parallel programming enviroment in Java. In *Proc. of ICCS: International Conference on Computational Science*, volume 2330 of *LNCS*, pages 844–853. Springer Verlag, April 2002.

[22] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object re-classification. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 130–149, London, UK, 2001. Springer-Verlag.

[23] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. More dynamic object reclassification: Fickle∥. *ACM Trans. Program. Lang. Syst.*, 24(2):153–191, 2002.

[24] Dietmar W. Erwin and David F. Snelling. Unicore: A grid computing environment. In *Lecture Notes in Computer Science*, volume 2150, pages 825–834. Springer, 2001.

[25] J. F. Ferreira, J. L. Sobral, and A. J. Proenca. Jaskel: A java skeleton-based framework for structured cluster and grid computing*. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 301–304, Washington, DC, USA, 2006. IEEE Computer Society.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[27] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[28] Herbert Kuchen and Jörg Striegnitz. Higher-order functions and partial applications for a c++ skeleton library. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 122–130, New York, NY, USA, 2002. ACM Press.

[29] Herbert Kuchen and Jörg Striegnitz. Features from functional programming for a c++ skeleton library: Research articles. *Concurrency and Computation: Practice and Experience*, 17(7-8):739–756, 2005.

[30] Aldinucci Marco and Massimo Torquati. Accelerating apache farms through ad-hoc distributed scalable object repository. In M. Danelutto, M. Vanneschi, and Domenico Laforenza, editors, *Proc. of 10th Intl. Euro-Par 2004: Parallel and Distributed Computing*, volume 3149 of *LNCS*, pages 596–605. Springer Verlag, August 2004.

[31] Marco Danelutto Marco Aldinucci and Patrizio Dazzi. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4), December 2007. To appear.

[32] Sun Microsystems. Java. http://java.sun.com.

[33] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, and Laurent Martelli. Jac: an aspect-based distributed dynamic framework. *Softw. Pract. Exper.*, 34(12):1119–1148, 2004.

[34] Steffen Priebe. Dynamic task generation and transformation within a nestable workpool skeleton. In *Proceedings of the 12th International Euro-Par Conference: Parallel Processing*, volume 4128 of *LNCS*, pages 615–624, Dresden, Germany, August 2006. Springer.

[35] Éric Tanter. On dynamically-scoped crosscutting mechanisms. *ACM SIGPLAN Notices*, 42(2):27–33, February 2007.

[36] Éric Tanter and Rodolfo Toledo. A versatile kernel for distributed AOP. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, volume 4025 of *Lecture Notes in Computer Science*, pages 316–331. Springer-Verlag, June 2006.

[37] Unicore. http://www.unicore.org.

[38] Gregor von Laszewski, Beulah Alunkal, Jarek Gawor, Ravi Madhuri, Pawel Plaszczak, and Xian-He Sun. A File Transfer Component for Grids. In H.R. Arabnia and Youngson Mun, editors, *Proceedings of the International Conferenece on Parallel and Distributed Processing Techniques and Applications*, volume 1, pages 24–30. CSREA Press, 2003.

[39] Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A Java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):645–662, /2001.

[40] Gregor von Laszewski, Jarek Gawor, Pawel Plaszczak, Mike Hategan, Kaizar Amin, Ravi Madduri, and Scott Gose. An overview of grid file transfer patterns and their implementation in the java cog kit. *Neural, Parallel Sci. Comput.*, 12(3):329–352, 2004.