ADAPTABLE PARALLEL COMPONENTS FOR GRID PROGRAMMING

Jan Dünnweber and Sergei Gorlatch
University of Münster, Department of Mathematics and Computer Science
Einsteinstrasse 62, 48149 Münster, Germany
duennweb@uni-muenster.de
gorlatch@uni-muenster.de

Marco Aldinucci, Sonia Campa and Marco Danelutto
Università di Pisa, Department of Computer Science
Largo B. Pontecorvo 3, 56127 Pisa, Italy
aldinuc@di.unipi.it
campa@di.unipi.it
marcod@di.unipi.it

Abstract We suggest that parallel software components used for grid computing should be adaptable to application-specific requirements, instead of developing new components from scratch for each particular application. As an example, we take a parallel farm component which is "embarrassingly parallel", i. e. , free of dependencies, and adapt it to the wavefront processing pattern with dependencies that impact its behavior. We describe our approach in the context of Higher-Order Components (HOCs), with the Java-based system Lithium as our implementation framework. The adaptation process relies on HOCs' mobile code parameters that are shipped over the network of the grid. We describe our implementation of the proposed component adaptation method and report first experimental results for a particular grid application – the alignment of DNA sequence pairs, a popular, time-critical problem in computational molecular biology.

Keywords: Grid Components, Adaptable Code, Wavefront Parallelism, Java, Web Services

1. Introduction

Grids are a promising platform for distributed computing with high demand on data throughput and computing power, but they are still difficult to program due to their highly heterogeneous and dynamic nature. Popular technologies for programming grids are Java, since it enables portabilty for executable code, and Web services, which facilitate the exchange of application data in a portable a format. Thus, multiple Java-based components, distributed across the Internet, can work together using Web services.

Besides interoperability, grid applications require from their runtime environments support for the sharing of data among multiple services and a possibility for issuing non-blocking service requests. The contemporary grid middleware systems, e. g. , the Globus Toolkit [6] and Unicore [15] address such recurring issues, thus freeing users from dealing with the same problems again and again. Middleware abstracts over the complex infrastructure of a grid: application code developed by middleware users (which still consists in Java-based Web services in most cases) is not so heavily concerned with the low-level details of network communication and the maintenance of distributed data.

While providing an infrastructure-level abstraction, middleware introduces numerous non-trivial configuration requirements on the system-level, which complicates the development of applications. Therefore, recent approaches to simplifying the programming of grid applications often introduce an additional layer of software components abstracting over the middleware used in the grid.

Software components for the grid aim to be easier to handle than raw middleware. In [14], components are defined as software building-blocks with no implicit dependencies regarding the runtime environment; i. e. , components for grid programming are readily integrated with the underlying middleware, hiding it from the grid users. An example for grid programming components is given by the CoreGRID *Grid Component Model* (GCM), a specification which emerged from the component models Fractal [2], HOCs [7], ASSIST [12] and other experimental studies, conducted within the CoreGRID community. While the GCM predecessors are accompanied by framework implementations, providing the users with an API, there is yet no GCM framework. Anyway, there are multiple implementations of Fractal, the HOC-SA [5] for programming with HOCs, the ASSIST framework for data-flow programming and its Java-based variant Lithium [4]. These frameworks allow to experiment with many GCM features and to preliminarily analyse limitations of the model.

This paper addresses grid application programming using a component framework, where applications are built by *selecting*, *customizing* and *combining* components. Selecting means choosing appropriate components from the framework, which may contain several ready-made implementations of commonly used parallel computing schemata (farm, divide-and-conquer, etc. [3]).

By customization, we mean specifying application-specific operations to be executed within the processing schema of a component, e. g. , parallel farming of application-specific tasks. Combining various parallel components together for accomplishing one task, can be done, e. g. , via Web services.

As our main contribution, we introduce *adaptations* of software components, which extends the traditional notion of *customization*: while customization applies a component's computing schema in a particular context, adaptation modifies the very schema of a component, with the purpose of incorporating new capabilities. Our thrust to use adaptable components is motivated by the fact that a fixed framework is hardly able to cover every potentially useful type of component. The behavior of adaptable components can be altered, thus allowing to apply them in use cases for which they have not been originally designed. We demonstrate that both, traditional customization and adaptation of components can be realized in a grid-aware manner (i. e. , also in the context of an upcoming GCM-framework). We use two kinds of components' parameters that are shipped over the network with the purpose of adaptation: these parameters may be either data or executable codes.

As a case study, we take a component that was originally designed for dependency-free *task farming*. By means of an additional code parameter, we adapt this component for the parallel processing of tasks exhibiting data dependencies with a *wavefront* structure.

In Section 2, we explain our *Higher-Order Components* (HOCs) and how they can be made adaptable. Section 3 describes our application case study used throughout the paper: the alignment of sequence pairs, which is a wavefront-type, time-critical problem in computational molecular biology. In Section 4, we show how the HOC-framework enables the use of mobile code, as it is required to apply a component adaptation in the grid context. Section 5 shows our first experimental results for applying the adapted farm component to the alignment problem in different, grid-like infrastructures. Section 6 summarizes the contributions of this paper in the context of related work.

2. Components and Adaptation

When an application requires a component, which is not provided by the employed framework, there are two possibilities: either to code the required component anew or to try and derive it from another available component. The former possibility is more direct, but it has to be done repeatedly for each new application. The latter possibility, which we call adaptation, provides more flexibility and potential for reuse of components. However, it requires from the employed framework to have a special adaptation mechanism.

2.1 Higher-Order Components (HOCs)

Higher-Order Components (HOCs) [7] are called so because they can be parameterized not only with data but also with code, in analogy to higher-order functions that may use other functions as arguments. We illustrate the HOC concept using a particular component, the Farm-HOC, which will be our example throughout the paper. We first present how the Farm-HOC is used in the context of Java and then explain the particular features of HOCs which make them well-suited for adaptation. While many different options (e. g. , C + MPI or Pthreads) are available for implementing HOCs, in this paper, our focus is on Java, where multithreading and the concurrency API are standardized parts of the language.

2.2 Example: The Farm-HOC

The farm pattern is only one of many possible patterns of parallelism, arguably one of the simplest, as all its parallel tasks are supposed to be independent from each other. There may be different implementations of the farm, depending on the target computer platform; all these implementations have, however, in common that the input data are partitioned using a code unit called the `Master` and the tasks on the data parts are processed in parallel using a code unit called the `Worker`. Our Farm-HOC, has therefore two so-called *customization code parameters*, the `Master`-parameter and the `Worker`-parameter, defining the corresponding code units in the farm implementation.

The code parameters specify how the Farm-HOC should be applied in a particular situation. The `Master` parameter must contain a `split` method for partitioning data and a corresponding `join` method for recombining it, while the `Worker` parameter must contain a `compute` method for task processing. Farm-HOC users declare these parameters by implementing the following two interfaces:

```
1: public interface Master<E>  {
2:  public E[][] split(E[] input, int grain);
3:  public E[] join(E[][] results);   }
4: public interface Worker<E>  {
5:  public E[] compute(E[] input);  }
```

The `Master` (line 1–3) determines how an input array of some type `E` is split into independent subsets, and the `Worker` (line 4–5) describes how a single subset is processed as a task in the farm. While the `Worker`-parameter differs in most applications, programmers typically pick the default implementation of the `Master` from our framework. This `Master` splits the input regularly, i. e. , into equally sized partations. A specific `Master`-implementation must only be provided, if a regular splitting is undesireable, e. g. , for preserving certain data correlations.

Unless an adaptation is applied to it, the processing schema of the Farm-HOC is very general, which is a common property of all HOCs. In the case of the Farm-HOC, after the splitting phase, the schema consists in the parallel execution of the tasks described by the implementation of the above `Worker`-interface. To allow the execution on multiple servers, the internal implementation of the Farm-HOC adheres to the widely used scheduler/worker-pattern of distributed computing: A single scheduler machine runs the `Master`-code (the first server given in the call to the `configureGrid` method, shown below) and the other servers each run a pool of threads, wherein each thread waits for tasks from the scheduler and then processes them using the `Worker` code parameter, passed during the farm initialization.

The following code shows how the Farm-HOC is invoked on the grid as a Web service via its remote interface `farmHOC`:

```
1: farmHOC.configureGrid( "masterHost",
2:                        "workerHost1",... ,
3:                        "workerHostN" );
4: farmHOC.process(input, LITHIUM, JAVA5);
```

The programmer can pick the servers to be employed for running the `Worker`-code via the `configureGrid`-method (line 1–3), which accepts either host names or IP addresses as parameters. Moreover, the programmer can select, among various implementations, the most adequate version for a particular network topology and for particular server architectures (in the above code, the version based on the grid programming library Lithium [4] is chosen). The `JAVA5`-constant, passed in the invocation (line 4), specifies that the format of the code parameters to be employed in the execution is Java bytecode compliant to Java virtual machine versions 1.5 or higher.

2.3 The Implementation of Adaptable HOCs

The need for adaptation arises if an application requires a processing schema which is not provided by the available components. Adaptation is used to derive a new component with a different behavior from the original HOC. Our approach is that a particular adaptation is also specified via a code parameter, similar to the customization shown in the preceding section. In contrast to a customizing code parameter, which is applied within the execution of the HOC's schema, a code parameter specifying an adaptation runs in parallel to the execution of the HOC. There is no fixed position for the adaptation code in the HOC implementation; rather the HOC exchanges messages with it in a publish/subscribe-manner. This way, a code parameter can, e. g. , block the execution of the HOC's standard processing schema at any time, until some condition is fulfilled.

Our implementation design can be viewed as a general method for making components adaptable. The two most notable, advantageous properties of our implementation are as follows: 1) Using HOCs, adaptation code is placed within one or multiple threads of its own, while the original framework code remains unchanged, and 2) An adaptation code parameter is connected to the HOC using only message exchange, leading to high flexibilty.
This design has the following advantageous properties:

- we clearly separate the adaptation code not only from the component implementation code, but also from the obligatory, customizing code parameters. When a new algorithm with new dependencies is implemented, the customization parameters can still be written as if this algorithm introduced no new data dependencies. This feature is especially obvious in case of the Farm-HOC, as there are no dependencies at all in a farm. Accordingly, the `Master` and `Worker` parameters of a component derived from the Farm-HOC are written dependency-free.

- we decouple the adaptation thread from the remaining component structure. There can be an arbitrary number of adaptations. Due to our messaging model, adaptation parameters can easily be changed. Our model promotes better code reusability as compared to passing information between the component implementations and the adaptation code directly via the parameters and return values of the adaptation codes' methods. Any thread can publish messages for delivery to other that provides the publisher with an appropriate interface for receiving messages. Thus, adaptations can also adapt other adaptations and so on.

- Our implementation offers a high degree of location independence: In the Farm-HOC, the data to be processed can be placed locally on the machine running the scheduler or they can be distributed among several remote servers. In contrast to coupling the adaptation code to the `Worker` code, which would be a consequence of placing it inside the same class, our adaptations are not restricted to affecting only the remote hosts, but can also have an impact on the scheduler host. In our case study, we use this feature to efficiently optimize the scheduling behavior with respect to exploiting data locality: processing a certain amount of data locally in the scheduler significantly increases the efficiency of the computations.

3. Case Study: Sequence Alignment

Our case study in this paper is one of the fundamental algorithms in bioinformatics – the computation of *distances* between DNA sequences, i. e. , finding the minimum number of operations needed to transform one sequence into another. Sequences are encoded using the nucleotide alphabet $\{A, C, G, T\}$.

The distance, which is the total number of the required transformations, quantifies the similarity of sequences [11] and is often called *global alignment*. Mathematically, global alignment can be expressed using a so-called *similarity matrix S*, whose elements $s_{i,j}$ are defined as follows:

$$s_{i,j} := max\big(\ s_{i,j-1}+plt, s_{i-1,j-1}+\delta(i,j), s_{i-1,j}+plt\ \big) \tag{1}$$

wherein

$$\delta(i,j) := \begin{cases} +1 & \text{, if } \varepsilon_1(i) = \varepsilon_2(j) \\ -1 & \text{, otherwise} \end{cases} \tag{2}$$

Here, $\varepsilon_k(b)$ denotes the b-th element of sequence k, and *plt* is a constant that weighs the costs for inserting a space into one of the sequences (typically, $plt = -2$, the “double price” of a mismatch).

The data dependencies imposed by definition (1) imply a particular order of computation of the matrix: elements which can be computed independently of each other, i. e., in parallel, are located on a so-called *wavefront* which “moves” across the matrix as computations proceed. The wavefront is degenerated into a straight line when it is drawn along the single independent elements, but its ”wavy” structure becomes apparent when it spans multi-element blocks. In higher-dimensional cases (3 or more input sequences), the wavefront becomes a hyperplane [9].

The wavefront pattern of parallel computation is not specific only to the sequence alignment problem, but is used also in other popular applications: searching in graphs represented via their adjacency matrices, system solvers, character stream conversion problems, motion planning algorithms in robotics etc. Therefore, programmers would benefit if a standard component would capture the wavefront pattern. Our approach is to take the Farm-HOC, as introduced in Section 2, adapt it to the wavefront structure of parallelism and then customize it to the sequence alignment application. Fig. 2 schematically shows this two-step procedure. First, the workspace, holding the partitioned tasks for farming, is sorted according to the wavefront pattern, whereby a new processing order is fixed, which is optimal with respect to the degree of parallelism. Then, the alignment definitions (1) and (2) are employed for processing the sequence alignment application.

4. Adaptations with Globus & WSRF

The Globus middleware and the enclosed implementation of the *Web Services Resource Framework* (WSRF) form the middleware platform used for running HOCs (`http://www.oasis-open.org/committees/wsrf`).

The WSRF allows to set up stateful resources and connect them to Web services. Such resources can represent application state data and thereby make Web services and their XML-based communication protocol (SOAP) more suitable for grid computing: while usual Web services offer only self-contained operations, which are decoupled from each other and from the caller, Web services hosted with Globus include the notion of context: multiple operations can affect the same data, and changes within this data can trigger callbacks to the service consumer, thus avoiding blocking invocations.

Globus requires from the programmer to manually write a configuration consisting in multiple XML files which must be placed properly within the grid servers' installation directories. These files must explicitly declare all resources, the services used to connect to them, their interfaces and bindings to the employed protocol, in order to make Globus applications accessible in a platform- and programming language-independent manner.

4.1 Enabling Mobile Code

Users of the HOC-framework are freed from the complicated WSRF-setup described above, as all the required files, which are specific for each HOC but independent from applications, are provided for all HOCs in advance.

We provide a special class-loading mechanism allowing class definitions to be exchanged among distributed servers. The code pieces being exchanged among the grid nodes hosting our HOCs are stored as properties of resources that have been configured according to the HOC-requirements; e. g. , the Farm-HOC is connected with a resource for holding an implementation of one `Master` and one `Worker` code parameter.

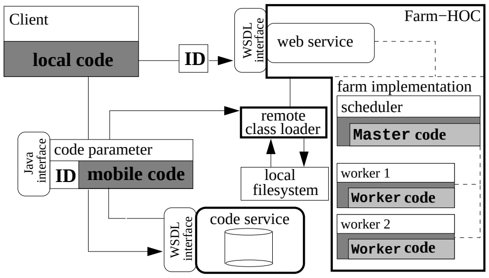

Figure 1. Transfer of code parameters

Fig. 1 illustrates the transfer of mobile code in the HOC-framework. The bold lines around the Farm-HOC, the *remote class loader* and the *code-service* indicate that these entities are parts of our framework implementation. The Farm-HOC, shown in the right part of the figure, contains an implementation of the farm schema with a scheduler that dispatches tasks to workers (two in the figure). The HOC implementation includes one Web service providing the publicly available interface to this HOC. Application programmers only

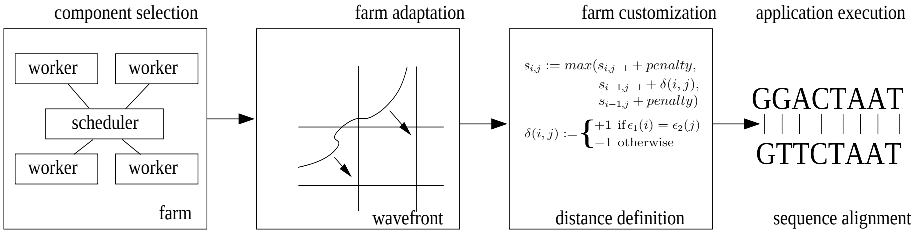
component selection
farm adaptation
farm customization
application execution
worker
worker
scheduler
worker
worker
farm
wavefront
$s_{i,j} := max(s_{i,j-1} + penalty, s_{i-1,j-1} + \delta(i,j), s_{i-1,j} + penalty)$
$\delta(i,j) := \begin{cases} +1 & \text{if} \, \epsilon_1(i) = \epsilon_2(j) \\ -1 & \text{otherwise} \end{cases}$
distance definition
GGACTAAT
| | | | | | | |
GTTCTAAT
sequence alignment

Figure 2. Two-step process: adaptation and customization

provide the code parameters. System programmers, who build HOCs, must assure that these parameters can be interpreted on the target nodes, which may be particularly difficult for heterogeneous grid nodes.

HOCs transfer each code unit as a record holding an identifier (`ID`) plus the a combination of the code itself and declaration of requirements for running the code. A requirement may, e. g. , be the availability of a certain Java virtual machine version. As the format for declaring such requirements, we use string literals, which must coincide with those used in the invocation of the HOC (e. g. , `JAVA5`, as shown in Section 2.2). This requirement-matching mechanism is necessary to bypass the problem that executable code is usually platform-specific, and therefore not mobile: not any code can be executed by an arbitrary host. Before we ship a code parameter, we guide it through the code-service – a Web service connected to a database, where the code parameters are filed as Java bytecode or in a scripting-language format. This design facilitates the reuse of code parameters and their mobility, at least across all nodes that run a compatible Java virtual machine or a portable scripting-language interpreter (e. g. , Apache BSF: `http://jakarta.apache.org/bsf`). The remote class loader in Fig. 1 loads class definitions from the code-service, if they are not available on the local filesystem.

In the following, we illustrate the two-step process of adaptation and customization shown in Fig. 2. For the sake of explanation, we start with the second step (HOC customization), and then consider the farm adaptation.

4.2 Customizing the Farm-HOC for Sequence Alignment

Our HOC framework includes several helper classes that simplify the processing of matrices. It is therefore, e. g. , not necessary to write any `Master` code, which splits matrices into equally sized submatrices, but we can fetch a standard framework procedure from the code service. The only code parameter we must write anew for computing the similarity matrix in our sequence alignment application is the `Worker` code. In our case study this parameter implements, instead of the general `Worker`-interface shown in Section 2.2, the alternative `Binder`-interface, which describes, specifically for matrix applications, how an element is computed depending on its indices:

```
public interface Binder<E>  {
 public E bind(int i, int j);  }
```

Before the HOC computes the matrix elements, it assigns an empty workspace matrix to the code parameter; i. e., a `matrix` reference is passed to the parameter object and, thus, made available to the customizing parameter code for accessing the matrix elements.

Our code parameter implementation for calculating matrix elements, accordingly to definition (1) from section 3, reads as follows:

```
1: new Binder<Integer>( ) {
2:  public Integer bind(int i, int j)  {
3:  return max( matrix.get(i, j - 1) + penalty,
4:   matrix.get(i - 1, j - 1) + delta(i, j),
5:   matrix.get(i - 1, j) + penalty );  }  }
```

The helper method `delta`, used in line 4 of the above code, implements definition (2).

The special `Matrix`-type used by the above code for representing the distributed matrix is also provided by our framework and it facilitates full location transparency, i. e., it allows to use the same interface for accessing remote elements and local elements. Actually, `Matrix` is an abstract class, and our framework includes two concrete implementations: `LocalMatrix` and `RemoteMatrix`. These classes allow to access elements in adjacent submatrices (using negative indices), which further simplifies the programming of distributed matrix algorithms. Obviously, these framework-specific utilities are quite helpful in the presented case study, but they are not necessary for adaptable components and therefore beyond the scope of this paper.

Farming the tasks described by the above `Binder`, i. e., the matrix element computations, does not allow data dependencies between the elements. Therefore any farm implementation, including the one in the Lithium library used in our case, would compute the alignment result as a single task, without parallelization, which is unsatisfactory and will be addressed by means of adaptation.

4.3 Adapting the Farm-HOC to the Wavefront Pattern

For the parallel processing of submatrices, the adapted component must, initially, fix the “wavefront order” for processing individual tasks, which is done by sorting the partitions of the workspace matrix arranged by the `Master` from the HOC-framework, such that independent submatrices are grouped in one wavefront. We compute this sorted partitioning, while iterating over the matrix-antidiagonals as a preliminary step of the adapted farm, similar to the loop-skewing algorithm described in [16]. The central role in our adaptation approach is played by the special *steering thread* that is installed by the user and runs the wavefront-sorting procedure in its initialization method.

After the initialization is finished, the steering thread keeps running concurrently to the original farm scheduler and periodically creates new tasks by executing the following loop:

```
 1: for (List<Task> waveFront : data)  {
 2:  if (waveFront.size( ) < localLimit)
 3:   scheduler.dispatch(wave, true);
 4:  else  {
 5:   remoteTasks = waveFront.size( ) / 2;
 6:   if ((surplus = remoteTasks % machines) != 0)
 7:    remoteTasks -= surplus;
 8:   localTasks = waveFront.size( ) - remoteTasks;
 9:   scheduler.dispatch(
10:    waveFront.subList(0, remoteTasks), false);
11:   scheduler.dispatch(
12:    waveFront.subList(remoteTasks,
13:    remoteTasks + localTasks), true);  }
14:  scheduler.assignAll( );  }
```

Here, the steering thread iterates over all wavefronts, i. e., the submatrices positioned along the anti-diagonals of the similarity matrix being computed.

The `assignAll` and the `dispatch` are not part of the standard Java API, but we implemented them ourselves to improve the efficiency of the scheduling as follows: The `assignAll`-method waits until the tasks to be processed have been assigned to workers. Method `dispatch`, in its first parameter, expects a list of new tasks to be processed. Via the second boolean parameter, the method allows the caller to decide whether these tasks should be processed locally by the scheduler (see lines 2–3 of the code above): the steering thread checks if the number of tasks is less than a limit set by the client. If so, then all tasks of such a “small” wavefront are marked for local processing, thus avoiding that communication costs exceed the time savings gained by employing remote servers. For wavefront sizes above the given limit, the balance of tasks for local and remote processing is computed in lines 5–8: half of the submatrices are

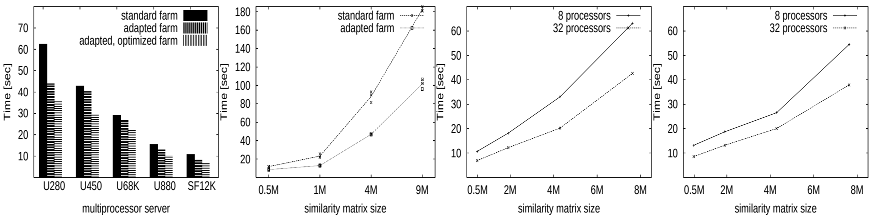

Figure 3. Experiments, from left to right: single multiprocessor servers; employing two servers; multiple multiprocessor servers; same input, zipped transmission

processed locally and the remaining submatrices are evenly distributed among the remote servers. If there is no even distribution, the surplus matrices are assigned for local processing. Then, all submatrices are dispatched, either for local or remote processing (lines 9—13) and the `assignAll`-method is called (line 14). The submatrices are processed asynchronously, as `assignAll` only waits until all tasks have been *assigned*, not until they are finished.

Without the `assignAll` and `dispatch`-method, the adaptation parameter can implement the same behavior using a `Condition` from the standard concurrency API for thread coordination, which is a more low-level solution.

5. Experimental Results

We investigated the run time of the application for processing the genome data of various fungi, as archived at `http://www.ncbi.nlm.nih.gov`. The scalability was measured in two dimensions: (1) with increasing number of processors in a single server, and (2) with increasing number of servers.

Server	*Architecture*	*Processors*	*Clock Speed*
SMP U280	Sparc II	2	750 Mhz
SMP U450	Sparc II	4	900 Mhz
SMP U880	Sparc II	8	900 Mhz
SMP U68K	UltraSparc III+	2	900 Mhz
SMP SF12K	UltraSparc III+	8	1200 Mhz

Table 1. The servers in our grid testbed

The first plot in Fig. 3 shows the results for computing a similarity matrix of 1 MB size using the SunFire machines listed above. We have deliberately chosen heterogeneous multiprocessor servers, in order to study a realistic, grid-like scenario.

A standard, non-adapted farm can carry out computations on a single pair of DNA sequences only sequentially, due to the wavefront-structured data dependencies. Using our Farm-HOC, we imitated this behavior by omitting the adaptation parameter and by specifying a partitioning grain equal to the size of an overall similarity matrix. This version was the slowest in our tests. Runtime measurements with the `localLimit` in the `steeringThread` set to a value $>= 0$ are labeled as *adapted, optimized farm*. The locality optimization, explained in Section 4.3, has an extra impact on the first plot in Fig. 3, since it avoids the use of sockets for local communication. To make the comparison with the standard farm version fairer, the `localLimit` was set to zero in a second series of measurements, which are labeled as *adapted farm* in Fig. 3. Both plots in Fig. 3 show the average results of three measurements. To obtain a measure for the spread, we always computed the variation coefficient; this turned to be less than 5% for all test series.

To investigate the scalability we ran the same application using two Pentium III servers under Linux. While the standard farm can only use one of the servers at a time, the adapted farm sends a part of the load to the second server, which improves the overall performance when the input sequence length increases (see the second plot). For more than two servers the performance was leveled off. We assume that this is due to the increase of communication, for distributing the `Binder`-tasks (shown in Section 4.2) over the network. The right plots in Fig. 3 support this assumption. We investigated the scalability using the U880 plus a second SunFire 6800 with 24 1350 MHz UltraSPARC-IV processors. As can be seen, the performance of our applications is significantly increased for the 32 processor configuration, since the SMP-machine-interconnection does not require the transmission of all tasks over the network. Curves for the standard farm are not shown in these diagrams, since they lie far above the shown curves and coincide for 8 and 32 processors, which only proves again that this version does not allow for parallelism within the processing of a single sequence pair. The outer right plot shows the effect of another interesting modification: When we compress the submatrices using the Java `util.zip` `Deflater`-class before we transmit them over the network, the curves do not grow so fast for small-sized input, but the absolute times for larger matrices are improved.

To estimate the overhead introduced by the adaptation and remote communication in our system, we compared our implementation to the *JAligner*-system, available from the *sourceforge.net* Web-site. Locally *JAligner* was about twice as fast as our system. On the distributed multiprocessor servers, the time for processing 9 MB using *JAligner* was about 1 min., while we measured execution times below 40 seconds for processing the same input using our system. This time advantage is explained by the fact that JAligner only benefits from the big caches of the grid servers, but it cannot make use of more than a single processor at a time. Thus, our adapted farm component outperforms the hand-tuned JAligner implementation, once the size of the processed genome data exceeds 10 MB.

6. Conclusion and Related Work

We adapted a farm component to wavefront computations. Although wavefront exhibits a different parallel behavior than farm, the remote interface, the resource configuration and most parts of a farm component's implementation could be reused due to the presented adaptation technique. Adaptations require that scheduling actions, crucial to the application progress, such as the loading of task data, can be extended by parameter code, which is provided to the component at runtime, as it is possible, e. g. , in the upcoming GCM, which includes the HOC code mobility mechanisms. A helpful analytical basis, which allows to derive new component adaptations from any application dependency graph, is given by the *Polytope*-model [10]. Polytope is also a possible starting point for future work on adaptable components, as it allows to automate the creation of adaptation code parameters.

Farm is a popular higher-order construct (i. e. , a component parameterized with code) that is available in several parallel programming systems. However, there is typically no wavefront component available. One of the reasons is that there are simply too many different parallel structures encountered in applications, so that it is practically impossible to every particular structure in a single, general component framework like, e. g. , CCA (`http://www.cca-forum.org`).

Of course, component adaptation is not restricted neither to farm components nor to wavefront algorithms. In an adaptation of other HOCs like the Divide-and-Conquer-HOC, our technique can take effect analogously: If, e. g. , an application of a divide-and-conquer algorithm allowed to conduct the join-phase in advance to the final data partitioning under certain circumstances, we could apply this optimization using an adaptation without any impact on the standard division-predicate of the algorithm.

Our case study shows that adaptable components allow for run-time rearrangements of software running on a distributed computer infrastructure, in the same flexible way as aspect-oriented programming simplifies code modifications at compile-time.

The use of the wavefront schema for parallel sequence alignment has been analyzed before in [1], where it is classified as a design pattern. While in the CO_2P_3S system the wavefront behavior is a fixed part of the pattern implementation, in our approach, it is only one of many possible adaptations that can be applied to a HOC. We used our adapted Farm-HOC for solving the DNA sequence pair alignment problem. In comparison with the extensive previous work on this challenging application [8, 13], we developed a high-level solution with competitive performance.

Acknowledgments

This research was conducted within the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

References

[1] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, and K. Tan. Generating parallel programs from the wavefront design pattern. In *7th Workshop on High-Level Parallel Programming Models*. IEEE Computer Society Press, 2002.

[2] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA)*. Springer LNCS, Catania, Sicily, 2003.

[3] M. I. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. Pitman, 1989.

[4] M. Danelutto and P. Teti. Lithium: A structured parallel programming enviroment in Java. In *Proceedings of Computational Science - ICCS*, number 2330 in Lecture Notes in Computer Science, pages 844–853. Springer-Verlag, Apr. 2002.

[5] J. Dünnweber and S. Gorlatch. HOC-SA: A grid service architecture for higher-order components. In *IEEE International Conference on Services Computing, Shanghai, China*, pages 288–294. IEEE Computer Society Press, Sept. 2004.

[6] Globus Alliance. http://www.globus.org, 1996.

[7] S. Gorlatch and J. Dünnweber. From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In *Future Generation Grids*. Springer Verlag, 2005.

[8] J. Kleinjung, N. Douglas, and J. Heringa. Parallelized multiple alignment. In *Bioinformatics 18*. Oxford University Press, 2002.

[9] L. Lamport. The parallel execution of do loops. In *Commun. ACM*, volume 17, 2, pages 83–93. ACM Press, 1974.

[10] C. Lengauer. Loop parallelization in the polytope model. In *International Conference on Concurrency Theory*, pages 398–416, 1993.

[11] V. I. Levenshtein. Binary codes capable of correcting insertions and reversals. In *Soviet Physics Dokl. Volume 10*, pages 707–710, 1966.

[12] M. Aldinucci, S. Campa et al. The implementation of ASSIST, an environment for parallel and distributed programming. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Proc. of the Euro-Par 2003*, number 2790 in lncs, pages 712–721. Springer, Aug. 2003.

[13] M. Schmollinger, K. Nieselt, M. Kaufmann, and B. Morgenstern. Dialign p: Fast pairwise and multiple sequence alignment using parallel processors. In *BMC Bioinformatics 5*. BioMed Central, 2004.

[14] C. Szyperski. *Component software: Beyond object-oriented programming*. Addison Wesley, 1998.

[15] Unicore Forum e.V. UNICORE-Grid, http://www.unicore.org, 1997.

[16] M. Wolfe. Loop skewing: the wavefront method revisited. In *Journal of Parallel Programming, Volume 15*, pages 279–293, 1986.