

# Behavior Customization of Parallel Components for Grid Application Programming

*J. Dünnweber, S. Gorlatch*

*{duennweb,gorlatch}@math.uni-muenster.de*

*WWU Muenster*

*Dept. of Computer Science – University of Münster*

*Einsteinstr. 62, Münster, Germany*

*M. Aldinucci, S. Campa, M. Danelutto*

*{aldinuc,campa,marcod}@di.unipi.it*

*UNIFI*

*Dept. of Computer Science – University of Pisa*

*Largo B. Pontecorvo 3, Pisa, Italy*



CoreGRID Technical Report

Number TR-0002

April 8, 2005

Institute on Programming Model

CoreGRID - Network of Excellence

URL: <http://www.coregrid.net>

# Behavior Customization of Parallel Components for Grid Application Programming

J. Dünneweber, S. Gorlatch  
{duennweb,gorlatch}@math.uni-muenster.de  
WWU Muenster  
Dept. of Computer Science – University of Münster  
Einsteinstr. 62, Münster, Germany

M. Aldinucci, S. Campa, M. Danelutto  
{aldinuc,campa,marcod}@di.unipi.it  
UNIPI  
Dept. of Computer Science – University of Pisa  
Largo B. Pontecorvo 3, Pisa, Italy

*CoreGRID TR-0002*

April 8, 2005

## Abstract

Components are usually provided as program building blocks with a fixed parallel structure that can be customized for a particular grid application by providing data and code parameters. We propose *behavior customization* of components, which changes components' parallel structure and behavior. As an example, we show how an “embarrassingly parallel” (dependency-free) *farm* component can be customized towards a *wavefront* component with dependencies between elements. In the grid context, our behavior customization makes a grid-aware component from a parallel component using suitable implementation mechanisms, e.g., web services. We illustrate our approach using an example application: the alignment of DNA sequence pairs, which is a popular, time-critical problem in computational molecular biology. We develop a prototypical implementation of the behavior customization using Java, RMI and SOAP, and report experimental results for the sequence alignment problem on a grid-like testbed.

## 1 Introduction

Component-based systems have become increasingly popular in parallel and distributed computing. They provide different kinds of reusable program building blocks, for which various names are used: skeletons, archetypes, paradigms, etc. . Components capture program structures that are often exploited in applications (e.g., pipeline, farm, divide-and-conquer) and are offered to the programmer together with an implementation on typical computer architectures. While the algorithmic structure of a component is usually fixed, the application-specific computations performed inside the structure can be supplied as component parameters – this is called *customization*. The programming process proceeds as follows: the programmer expresses his/her application by choosing suitable components, customizes them by providing application-specific parameters and then composes the customized components into a target program.

Our work is motivated by questions which are critical for every component-based programming system: (1) does a “right” set of components exist? (2) in which cases should a new component be added explicitly to the repository? (3) should such extension be the task of the system expert or rather of the application programmer? Obviously, it is hardly possible to create a fixed “universal” component repository that would cover all parallel and distributed control structures relevant in practice – the set of components would quickly become too complex and unmanageable. Another possibility is to allow the users to define and introduce new components from scratch, which requires the

application programmers to have too much system knowledge on a low level of abstraction and may lead to problems with correctness and performance.

We propose a novel approach to extending the set of components, which we call *behavior customization*. The idea is that not only the computations within a component are customizable (as in the traditional approach, for which we use the name *application customization*), but also the very structure of the component, and thereby its parallel behavior can be customized. In the grid context, our behavior customization accomplishes an additional task: it makes a grid-aware component from a parallel component using suitable implementation mechanisms, e.g., web services.

As a practical example, we show how an “embarrassingly parallel” (dependency-free) *farm* component can be customized towards a *wavefront* component with dependencies between elements, which can run efficiently in a grid environment. We illustrate our approach using a particular application example: the alignment of sequence pairs, which is a popular, time-critical problem in computational molecular biology. We develop a prototypical implementation of behavior customizations using Java, RMI and SOAP, and report experimental results for the sequence alignment problem on a grid-like testbed.

## 2 Customizing Components using Code Parameters

The new behavior customization is an optional step which precedes the traditional application customization, as shown in Fig. 1 for one of the components from a repository. In the figure, there are two possible behavior customizations, after which two application customizations are possible.

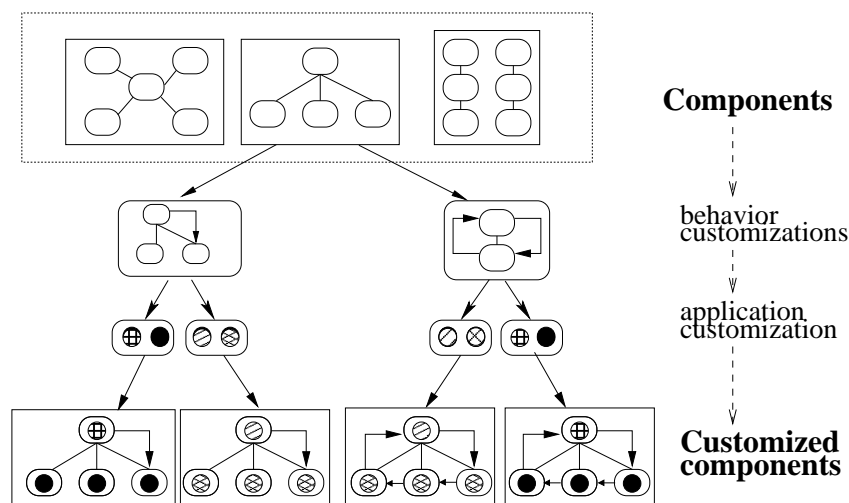


Figure 1: The two-step component customization process

Both customization steps are realized using code parameters, which in our system [6] are stored by the so-called *code service* and can be shipped over the network to a remote machine on demand. After selecting suitable components, the programmer can customize them by passing references to particular code parameters that interact with the component implementations. Once the behavior and/or application customization is over, the application is constructed by composing customized components and then executed on a parallel or distributed platform.

## 3 Application Example: DNA Sequence Alignment

A fundamental algorithm in bioinformatics is the *distance*-computation between DNA sequences, i.e., finding the minimum number of insertion, deletion or substitution operations needed to transform one sequence into another. Sequences are encoded using the alphabet  $\{A, C, G, T\}$ , where each letter stands for one of the nucleotide types [5]. The total distance, which is the sum of the required transformations, quantifies the similarity of sequences [11] and is often called *global alignment* [17].

Mathematically, global alignment can be expressed using a so-called *similarity matrix*  $S$ , whose elements  $s_{i,j}$  are defined as follows:

$$s_{i,j} := \max(s_{i,j-1} + \text{penalty}, s_{i-1,j-1} + \delta(i,j), s_{i-1,j} + \text{penalty}) \quad (1)$$

where

$$\delta(i,j) := \begin{cases} +1 & , \text{ if } \epsilon_1(i) = \epsilon_2(j) \\ -1 & , \text{ otherwise} \end{cases} \quad (2)$$

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

Definition (1) imposes data dependencies between the matrix elements, shown by arrows in Fig. 2 left. These dependencies 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* shown in Fig. 2 right, which “moves” across the matrix as computations proceed. In multidimensional cases, wavefront is often called hyperplane [10].

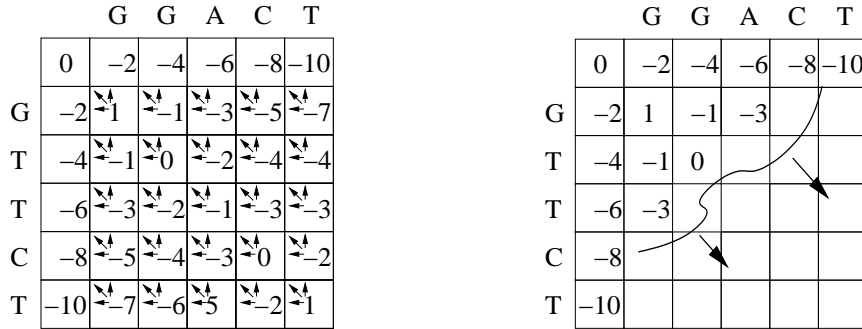


Figure 2: The wavefront processing schema imposed by the element dependencies

The parallelization structure based on a wavefront is typical for many different applications, so it would be desirable to provide a generic “wavefront” component which could be adapted as needed to a particular application. In practice-relevant cases (e.g., huge DNA sequences), it would be also useful to process in parallel blocks of the similarity matrix rather than single elements, in order to improve the processor utilization.

In the next section, we will study how this can be accomplished using a particular component-based programming system.

## 4 Component Customization: From Farm to Wavefront

Since our sequence alignment application follows the wavefront pattern of parallel computation, the application programmer would benefit from a component that captures this pattern. Unfortunately, there is no such component in the currently available component-based programming systems. One of the reasons is that there are simply too many different parallel structures encountered in applications, so that it is practically impossible to include all of them in the component repository.

Our approach is to take the farm component that is available in many component-based systems and customize its behavior to the required wavefront structure of parallelism. Fig. 3 shows how we apply the two-step customization in the sequence alignment example. We start from a compute farm. First, the behavior customization is applied to achieve a partitioning and to specify the wavefront processing order. Then the application customization determines how to process single input data elements. Finally, the adapted component can be used for parallel sequence alignment.

### 4.1 Application Customization

For the sake of explanation, we start with the second step of the customization process, the application customization, and then consider the behavior customization. As an example implementation, we take the farm component from the

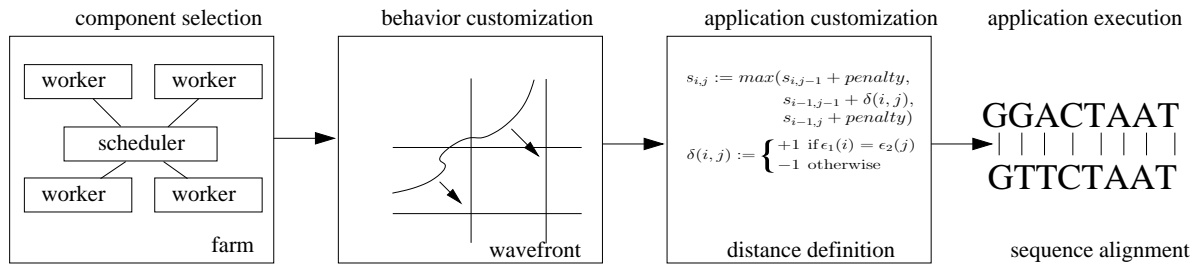


Figure 3: Customization steps from a farm to the alignment application

Java/RMI-based skeleton system Lithium [2] and use our SOAP-based HOC-SA environment [6] for, firstly, customizing the farm to the waverfront component and, secondly, making it suitable for grids. Neither Lithium nor HOC-SA is a requirement for the presented customization techniques; they are rather typical representatives of current systems in the area.

Lithium is a Java-based system that contains a repository of *skeletons*, parallel components which can be customized by supplying them with code parameters. The farm skeleton in Lithium is implemented according to the *scheduler/worker*-pattern: A single *scheduler* continuously dispatches independent *tasks* to several remote servers called *workers*. Each worker may run multiple threads waiting for tasks from the scheduler and processing the tasks by applying the code parameter passed during the farm initialization. The use of the code parameter is a typical example of application customization.

If the farm skeleton computes matrices, then its code parameter should implement the following interface describing how an element is computed depending on its indices:

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

For our sequence alignment application, the code parameter should compute all matrix elements according to definition (1), i.e.:

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

Unfortunately, the farm in Lithium does not allow data dependencies between matrix elements, so our sequence alignment would be computed as a single task, without parallelization, which is unsatisfactory.

## 4.2 Behavior Customization

In this section, we customize the behavior of the farm in Lithium, so that our example application can be parallelized. We exploit the following feature of the farm implementation: Once a worker has finished a task, the scheduler publishes this state change according to the *observer pattern* [7]. Thus, any class implementing the *Observer*-interface from the Java standard API can trigger a modified treatment of tasks.

First of all, a new “wavefront” order for processing individual tasks must be defined, which is done by partitioning the similarity matrix, so that independent submatrices are grouped in one wavefront. Such a partitioning can be computed using an algorithm like the one described in [16], as a preliminary step of the new component.

The central role in our behavior customization is played by the *steering thread* that runs concurrently to the original farm scheduler. This thread periodically creates new tasks by performing the following loop:

```
for (List<Task> waveFront : data) {
    if (waveFront.size( ) < localLimit)
        scheduler.dispatch(wave, true);
    else {
        remoteTasks = waveFront.size( ) / 2;
        if ((surplus = remoteTasks % machines) != 0)
```

```

        remoteTasks += surplus;
    localTasks = waveFront.size( ) - remoteTasks;
    scheduler.dispatch(
        waveFront.subList(0, remoteTasks), false );
    scheduler.dispatch(
        waveFront.subList(remoteTasks,
            remoteTasks + localTasks, true );
    }
    scheduler.assignAll( );
}

```

The steering thread iterates over all wavefronts, i. e. the lists of submatrices positioned along the anti-diagonals of the similarity matrix. This thread exploits only two methods that are not standard Java, but specific to the scheduler of the Lithium farm component. Method `assignAll` waits until the pool of tasks to be processed has been emptied. Method `dispatch` puts a list of tasks into the pool and allows the caller to decide whether the tasks should be sent to a remote server or 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, all tasks of this “small” wavefront are marked for local processing, thus avoiding sending submatrices across network boundaries if the communication costs exceed the time savings gained by employing remote servers. For wavefronts consisting of more submatrices than the given limit, the balance of tasks for local and remote processing is computed in lines 5–8. Finally, for any distribution of tasks, the steering thread calls the `assignAll` method before continuing with the next wavefront. This assures that the scheduling adheres to the wavefront processing order and that the steering thread blocks as long as not all tasks of the currently processed diagonal have been assigned to a remote server or to the scheduler for local processing. Note that `assignAll` only waits until all tasks have been *assigned*, not until they are finished. Therefore, our behavior customization has no impact on the asynchronous processing schema of the Lithium farm, with all its optimizations discussed in [1].

Possible alternatives to the described behavior customization include replacing the original Lithium scheduler by another one operating in a wavefront manner or adding a completely new component type for wavefront algorithms. The first alternative would be valid exclusively for the current Lithium version and moreover, we would hard-wire the wavefront behavior into the farm. The second alternative involves much more overhead than the described customization of an existing component.

## 5 Customizing Components for Grids

In this section, we show how our farm-wavefront component can be customized further, making it suitable for working in a grid environment. One possible problem could be a grid host running a non-Java application that requires our Java-based sequence alignment as a subtask within a larger application. Moreover, the Java RMI mechanism would constrain the use of our component: as soon as a firewall blocks the used port, the component cannot be accessed anymore.

### 5.1 Bringing Grid-Awareness with Web Services

For the purpose of “gridifying customization” of parallel components, we use our Higher-Order Component Service Architecture (HOC-SA) [6]. It provides a runtime environment based on partially implemented web services (these correspond to not-yet-customized components), with a mechanism for shipping code parameters (these are used for component customization). Fig. 4 shows both the customization and execution process; it will be explained step-by-step in this section.

We have developed the following setup to combine the behavior customization of components with the technique of making them grid-aware. With Lithium as our example, to plug-in a steering thread, the `Scheduler` is wrapped into a web service that is deployed to the same host that runs the scheduler. This service offers public methods to select code parameters and to start a distributed computation. Here is a part of the service interface written for this purpose, showing the required WSDL definitions:

```

<wsdl:portType name="HOCPortType">...
    <operation name="selectComponent">...</operation>

```

```

<operation name="addController">...</operation>
<operation name="start">...</operation>
</wsdl:portType>

```

Component implementations and sequential code parameters are set by the client using the `selectComponent` method of this service interface. To modify the behavior of a component, its `addController`-method is called with the code parameter identifier. The identifier refers to the actual code parameter expressed in a portable format (Java bytecode or script file); it is transferred by the *code service* (Fig. 4) using a special class loading mechanism [6], because standard web services cannot handle mobile code.

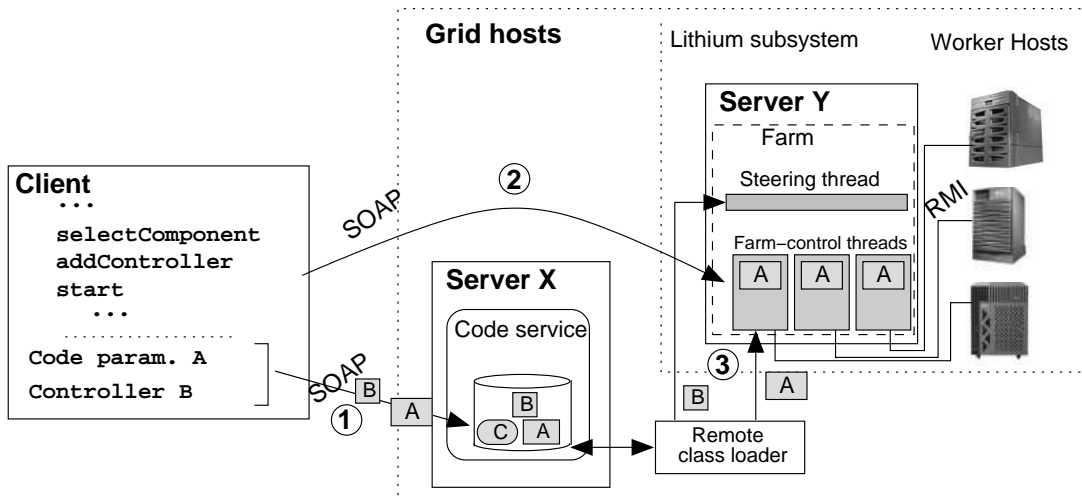


Figure 4: Customizing the Lithium farm using the HOC-SA code service

For the sequence alignment application, we set up a Lithium farm that iterates over the input matrix and applies the `bind`-operation defined in Section 4.1 to each element. The `addController`-method, shown in Figure 4, allows to upload an arbitrary number of steering threads that affect the behavior of the Lithium scheduler. The uploaded code must implement the `Controller` interface, by means of which it is accessed by the scheduler. The `Controller` interface is derived from `Runnable` and `Observer` as defined in the Java standard API, inheriting the `run` and `update` method declarations and declaring one additional `init` method.

Fig. 4 extends the customization procedure illustrated in Fig. 3 by the code transfer mechanisms used for customizations in our grid testbed. It shows one arbitrary machine, called `Server X`, hosting the code service and another server, called `Scheduler Server`, hosting the Lithium farm. The client starts by uploading two code parameters, `A` and `B`, to the code service (step ①) that are then selected in the calls to `selectComponent` and `addController`. Parameter `A` is the farm parameter applying the `bind`-operation from section 4.1 (so it defines the application customization) and parameter `B` is the steering thread, i.e. it defines the behavior customization of the Lithium farm.

## 5.2 Execution on the Grid

To initiate the execution of our customized, grid-aware component, the programmer calls the `start`-method of Lithium (step ② in the figure), which in turn calls the steering threads' `init` methods (one thread in our sequence alignment example), passing them a reference to the wrapped scheduler. This allows the steering threads to interact with the scheduler and to reprocess tasks. Once the `start` method is called by the user, the component contacts the code service and installs the code parameters (step ③ in the figure). The component, which is now customized, activates the distributed wavefront processing within Lithium, involving RMI communication only. The encapsulation of all RMI communications ensures that we do not limit the application to this particular communication technology. By providing an own implementation of the `init`-method in the `Controller`-interface defined above, the programmer is free to arrange any presetting needed for an application. Then the component starts the steering threads that are waiting for tasks to reprocess and the distributed process is initiated by delegating the call to the farm scheduler.

In our case study, we have a single steering thread whose `init`-method is overridden to perform an initialization of the border row and column of our similarity matrix  $S$  and to compute the required partitioning coordinates. The `run` method of our steering thread performs the loop shown in section 4.2 and its `update` method reawakens the Scheduler by calling `notify`, if it is waiting in the `assignAll`-method for tasks remaining to be assigned. Upon notification, the Scheduler re-checks the task pool fill level and eventually terminates the `assignAll`-method. If a wavefront is passed completely, the program flow returns to the steering thread's `run`-method that can thereupon trigger the dispatch of the next wavefront.

## 6 Experimental Results

We did experiments with the customized grid-aware farm-wavefront component which we applied to the sequence alignment problem. We investigate the run time of the application, as well as the scalability in two dimensions: (1) when the number of processors in a single server increases, and (2) when more servers are employed on the grid.

The left plot in Fig. 5 shows the results for using several multiprocessor servers: a SunFire 280R equipped with two UltraSPARC-III 750 Mhz processors (U280), a SunEnterprise 450 with four UltraSPARC-II processors (U450), a Sun 6800 Server with 2 UltraSPARC-III+ 900 Mhz processors (U68K), a SunFire 880 with 8 900 Mhz processors (U880) and another SunFire Server with 8 1200 Mhz processors (SF12K). The standard Lithium version was the slowest in all tests, although it still benefits from more powerful servers. Delegating computations to a faster server is the main motivation for deploying the application on a grid. The version with the optimized scheduler processes small wavefronts without involving any RMI communication.

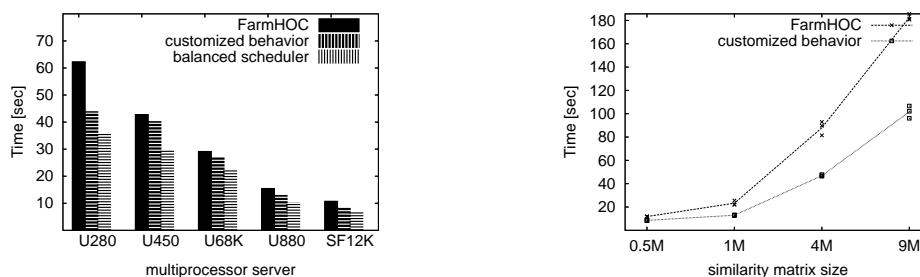


Figure 5: Experimental results for the sequence alignment. Left: single multiprocessor servers; Right: employing two servers

To investigate the scalability over several servers, we ran the same application using two Intel Pentium III PCs running Linux at 800MHz. While standard Lithium can only use one of the servers at a time, the customized version sends a part of the load to the second server, which improves the overall performance when the input sequence length increases (see the right plot). Both diagrams show the average results of three measurements. To obtain a measure for the spread, we always computed the variation coefficient which was less than 5% for all test series.

## 7 Conclusion and Related Work

As its main contribution, the paper introduced, implemented and experimentally investigated a novel method of extending the repository of components. Rather than defining a new component for each new application, standard components can be adjusted to the new requirements by means of *behavior customization*. Our approach extends the previous notion of component customization, which was restricted to only specifying the computation part of a component; we call it application customization. The two kinds of customization complement each other and provide greater flexibility in changing either the component's behavior or the component's computation part, or both, depending on particular requirements of the system and the application.

As an example, we demonstrated how the standard farm component of the Lithium skeleton system can be customized into a wavefront component and made grid-aware. The use of the wavefront schema for parallel sequence alignment has been analyzed before in [3], 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 one of many possible behavior customizations. Since the steering thread for wavefront can also be plugged into the Lithium scheduler without uploading it remotely, our solution can be viewed as a novel way of introducing a new skeleton to a skeleton-based system, without changing its standard scheduler.

Our work is one of the current efforts on hiding the complexity of Grid-aware application development (a good overview can be found in [9]). Related work includes the Ibis programming system [15] and the ProActive [4] system, both based on Java/RMI, and the PaCO++ library based on the CORBA communication mechanism [13].

We used our customized farm-wavefront component for implementing the DNA sequence alignment problem. In comparison with the extensive previous work on this application [8, 12, 14], we developed a generic, flexible solution, which at the same time is competitive w.r.t. performance. Our future work will investigate conditions under which the application should be executed remotely on the grid or locally on a multiprocessor.

## Acknowledgments

This research was conducted within the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265) and was supported by a travel grant from the German-Italian exchange programme VIGONI.

## References

- [1] M. Aldinucci, M. Danelutto, J. D'unnweber, and S. Gorlatch. Optimization techniques for implementing parallel skeletons in distributed environments. Technical Report TR-0001, Institute on Programming Model, CoreGRID - Network of Excellence, January 2005. Submitted to *Parallel Processing Letters*.
- [2] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.
- [3] John Anvik, Steve MacDonald, Duane Szafron, Jonathan Schaeffer, Steve Bromling, and Kai Tan. Generating parallel programs from the wavefront design pattern. In *7th International Workshop on High-Level Parallel Programming Models and Supportive Environments*. IEEE Computer Society Press, 2002.
- [4] Laurent Baduel, Fran coise Baude, and Denis Caromel. Efficient, flexible, and typed group communications in Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande (JGI-02)*, pages 28–36, Seattle, November 2002. ACM Press.
- [5] Carl-Ivar Branden, John Tooze, and Carl Branden. *Introduction to Protein Structure*. Garland Science, 1991.
- [6] Jan D'unnweber and Sergei 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, September 2004. ISBN 0-7695-2225-4.
- [7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison Wesley, 1995.
- [8] Jens Kleinjung, Nigel Douglas, and Jaap Heringa. Parallelized multiple alignment. In *Bioinformatics 18*. Oxford University Press, 2002.
- [9] D. Laforenza. Grid programming: some indications where we are headed. *Parallel Computing*, 28(12):1733–1752, December 2002.
- [10] Leslie Lamport. The parallel execution of do loops. In *Commun. ACM*, volume 17, 2, pages 83–93. ACM Press, 1974.
- [11] Vladimir I. Levenshtein. Binary codes capable of correcting insertions and reversals. In *Soviet Physics Dokl. Volume 10*, pages 707–710, 1966.
- [12] Kuo-Bin Li. Clustalw-mpi: Clustalw analysis using distributed and parallel computing. In *Bioinformatics 19*. Oxford University Press, 2003.

- [13] Christian Pérez, Thierry Priol, and André Ribes. PaCO++: A parallel object model for high performance distributed systems. In *Proceedings of the Conference on System Sciences (HICSS-37)*. IEEE Computer Society Press, January 2004.
- [14] Martin Schmollinger, Kay Nieselt, Michael Kaufmann, and Burkhard Morgenstern. Dialign p: Fast pair-wise and multiple sequence alignment using parallel processors. In *BMC Bioinformatics 5*. BioMed Central, 2004.
- [15] Robi V. van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, and Henri E. Bal. Ibis: an efficient Java-based Grid programming environment. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 18–27. ACM Press, November 2002.
- [16] Michael Wolfe. Loop skewing: the wavefront method revisited. In *International Journal of Parallel Programming, Volume 15*, pages 279–293, 1986.
- [17] X.Huang, R.C. Hardison, and W.Miller. A space-efficient algorithm for local similarities. In *Computer Applications in the Biosciences*, volume 6(4), pages 373–381. Oxford University Press, 1990.