Using Code Parameters for Component Adaptations

Jan Dünnweber, Sergei Gorlatch[1], Sonia Campa, Marco Danelutto[2], and Marco Aldinucci[3]

[1] Dept. of Computer Science – University of Münster – Germany
[2] Dept. of Computer Science – University of Pisa – Italy
[3] Inst. of Information Science and Technologies – CNR, Pisa, Italy

Abstract. Adaptation means that the behavior of a software component is adjusted to application- or platform-specific requirements: new components required in a particular application do not need to be developed from scratch when available components can be adapted accordingly. Instead of introducing a new adaptation syntax (as it is done, e. g. , in AOP), we describe adaptations in the context of Java-based Higher-Order Components (HOCs).
HOCs incorporate a code parameter plugin mechanism enabling adaptations on the grid. Our approach is illustrated using a case study of sequence alignment. We show how a HOC with the required provisions for data dependencies in this application can be generated by adapting a farm component, which is ”embarrassingly parallel”, i. e. , free of data dependencies. This way, we could reuse the efficient farm implementation from the Lithium library, although our case study exhibits the wavefront pattern of parallelism which is different from the farm.

1 Introduction

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 repository, which may contain several ready-made implementations of commonly used parallel computing schemata, e. g. , algorithmic skeletons (farm, divide-and-conquer, etc.) [4]. 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 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 more flexible, adaptable components is motivated by the fact that a fixed component framework is hardly able to cover all possible processing schemata. The sequential and parallel 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 using code parameters that can be shipped over the network of a grid.

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 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 [7]. 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, and present our grid-like testbed, highlighting the settings relevant for the system's adaptivity. Section 5 shows our first experimental results for the alignment problem in different, grid-like infrastructures. Section 6 summarizes the contributions of this paper in the context of related work.

2 Higher-Order Components (HOCs) and the Farm pattern

Higher-Order Components (HOCs) [6] are called so because they can be parameterized not only with data but also with code. We illustrate the HOC concept using a particular component, the Farm-HOC, which will be our example throughout the paper.

The farm pattern is a popular pattern of "embarrassing parallelism", without dependencies between tasks. 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`. The component expressing the farm schema, the 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.

These two parameters specify how the general farm schema should be applied in a particular situation. The `Master` parameter must contain a `split`-method for partitioning the input data and a corresponding `join` method for recombining it, while the `Worker` parameter must contain a `compute`-method for task processing. To use the Farm-HOC in our Java-based, grid-aware component framework, the programmer must provide implementations of 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, a specific implementation of the `Master` only has to be provided, if the input of a particular application should not be subdivided regularly, but it requires a special decomposition algorithm, e. g. , for preserving certain data correlations. Thus, in most applications, the user will only specify the `Worker` and pick a default `Master` implementation from our framework.

3 Case Study: Sequence Alignment

We illustrate the motivation for adaptation and its use by the following application case study.

One of the fundamental algorithms in bioinformatics is the computation of *distances* 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 [3].

The distance, which is the total number of the required transformations, quantifies the similarity of sequences [8] and is often called *global alignment* [12]. Mathematically, global alignment can be expressed using a so-called *similarity matrix S*, whose elements $s_{i,j}$ are defined as follows:

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

where

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

In Definition2 $\varepsilon_k(b)$ 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 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, such as a HOC, would capture the wavefront pattern.

Our approach is to take the Farm-HOC, as introduced in Section 2, adapt it to the required wavefront structure of parallelism and then customize it to the sequence alignment application.

Fig. 1 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, determining how to process single input data elements. Finally, this adapted component can be used for processing the sequence alignment application.

4 Adaptation with Globus & WSRF

Let us take a closer look at the currently most modern version of the Globus middleware and the enclosed implementation of the *Web Services Resource Framework* (WSRF) [9], before we present our extensions of this middleware for simplifying application development and for enabling component adaptations. 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

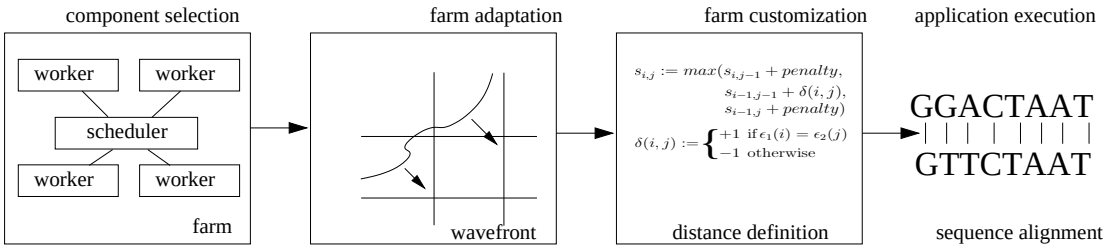

Fig. 1. Two-step process: adaptation and customization

offer only self-contained operations, which are decoupled from each other and from the caller, Web services hosted with Globus include the notion of a context; i. e. , multiple operations can affect the same data and changes within this data can trigger callbacks to the service consumer avoiding blocking invocations.

While making Web services more eligible for performance-critical applications, Globus is still too low-level to be used directly by application programmers: it requires the programmer to manually write multiple XML-configuration files and to place them properly within the grid servers' installation directories. These files must explicitly declare all resources, the services used to connect to them, their interfaces and the corresponding 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

Programming with adaptable and customizable components requires, besides the exchange of data, the exchange of *mobile code* across network boundaries. Therefore, we provide a special class-loading mechanism allowing class definitions to be exchanged among distributed servers. Interconnections between servers, which execute HOCs, and clients are established according to the WSRF standard.

Users of the HOC-framework are completely 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 the available HOCs in advance.

In the following, we illustrate the two-step process of adaptation and customization shown in Fig. 1. 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

The farm pattern is only one of many possible patterns of parallelism, arguably one of the simplest, which is available in many parallel component frameworks. When an application requires another 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 mechanism for enabling such adaptations.

Our framework includes a straightforward `Master` implementation for matrices, which partitions matrices into equally sized submatrices and recombines the submatrices after they have been processed. So, in the case of a matrix application, we do not need to write our own `Master` code parameter for partitioning the input data, but we can fetch the framework procedure from the code service by passing its ID (`matrixSplit`) to the Farm-HOC. 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, the alternative `Binder`-interface, which describes, specifically for matrix applications, how an element is computed depending on its indices:

```
1: public interface Binder<E>  {
2:  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 BinderParameter<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 for representing the distributed matrix data being split up among the workers by the HOC is 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 neighboring submatrices using overhang-indices (including negatives), which further simplifies the programming of distributed matrix algorithms. Obviously, these framework-specific utilities are quite helpful in the presented case study. Anyway, they are not necessary neither for customizing nor for adapting software components on the grid. Therefore, the implementation of these auxiliary classes is beyond the scope of this paper.

Farming the tasks described by the above `BinderParameter`, i. e., the matrix element computations, does not allow data dependencies between the elements. Therefore any farm implementation, including the one available 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.

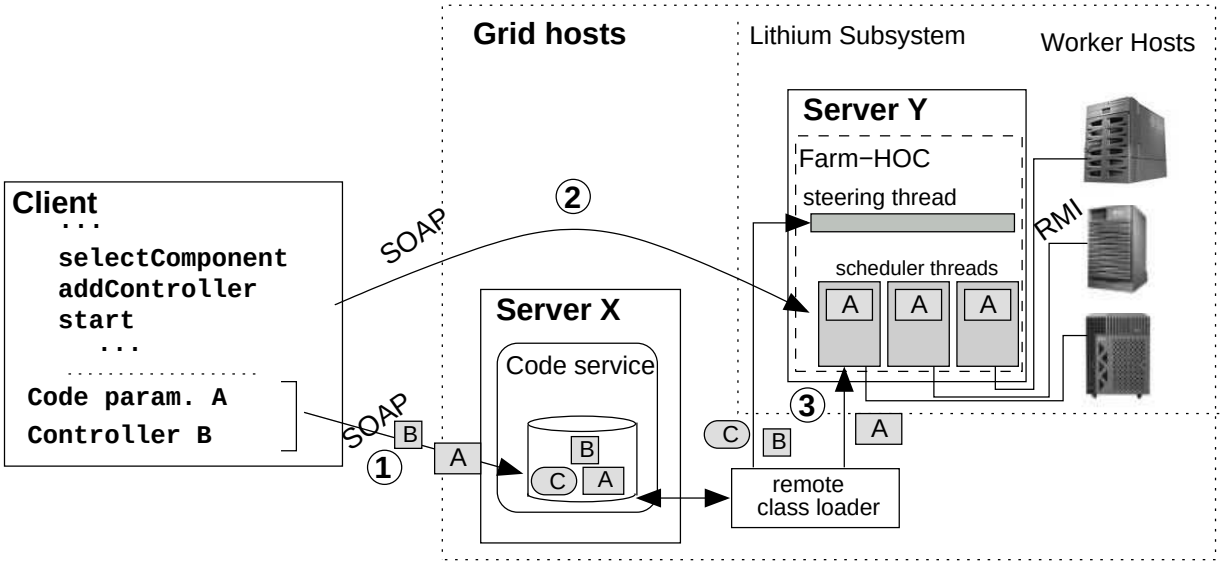

Fig. 2. Running the adapted component on the grid

4.3 Adapting the Farm-HOC to the Wavefront Pattern

In this section, we adapt the Farm-HOC to the wavefront pattern, so that it can be used for our example application. Like the farm customization described in the preceding section, the adaptation of the farm's parallel behavior is handled by means of code parameters, which are handled using our remote class loader and the code service providing a grid-aware code transfer mechanism as introduced above.

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 `matrixSplit-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 [11].

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. In this method, we also initialize the border row and column of the similarity matrix S, in our implementation.

4.4 Configuring the Runtime Environment

The client starts the configuring the runtime environment by uploading two code parameters, A and B, to the code service (step ① in Fig. 2). Parameter A is the farm worker parameter applying the `bind`-method from section 4.2; parameter B is the steering thread; i. e. , it defines the adaptation of the farm for wavefront processing. Parameter C , which represents the `Master`, is the only parameter not uploaded

by the client, but readily provided by the `matrixSplit`-implementation in our framework. In the final configuration step ③, Server Y retrieves the code parameters [A], [B] and (C) from the code service and installs them using the remote class loader.

Our Farm-HOC, which is now adapted to wavefront computations and customized for sequence alignment, then handles the whole distributed computation process on behalf of the client, which receives a notification once the process is finished.

5 Experimental Results

To investigate the scalability of our implementation over several servers, we ran it using two Pentium III servers under Linux at 800MHz. In the left plots in Fig. 3, we investigated the scalability using two multiprocessor servers: 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, for dispatching them to multiple processors. 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.

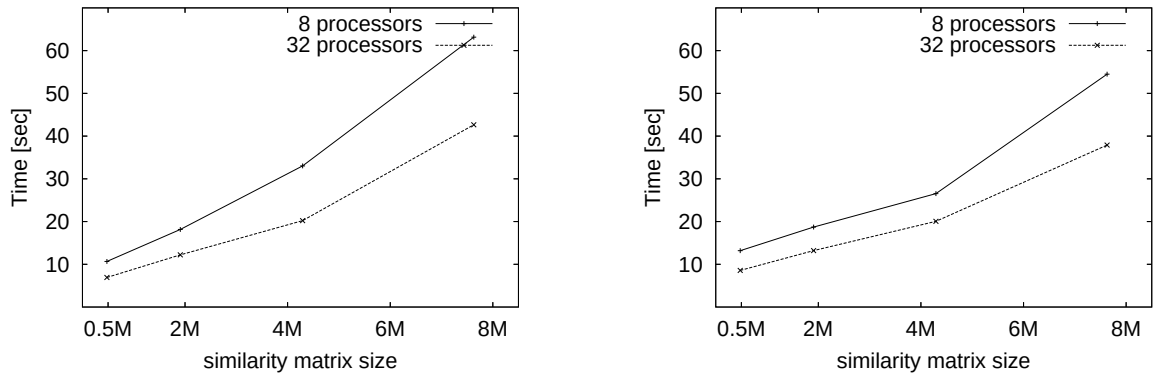

Fig. 3. Experimental results on grid-like testbeds. left: multiple multiprocessor servers; right: same input, zipped transmission

The right plots in Fig. 3 show 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, since the compression procedure slows down the process, for small-sized input, but the absolute times for larger matrices are improved.

6 Conclusion and Related Work

As its main contribution, this paper introduced, implemented and experimentally investigated a novel method for the adaptation of parallel programming components, in order

to optimize their behavior for grid applications. We have shown that the code parameter mechanism provided by our Higher-Order Components (HOCs) allows for building grid-aware applications via adaptation, which can free the programmer from developing and deploying new components in many use cases. To the best of our knowledge, adaptations of components have so far not been considered, neither in the general component model [10] nor in the skeleton approach to parallel programming [4]. Adaptation extends the previous notion of component customization, which was restricted to only specifying the computation part of a component.

Our farm implementation was taken from the Java-based system Lithium [5]. In [6], we described an alternative Farm-HOC implementation, in which not only a Web service was used to connect to the HOC, but also the communication within the farm itself was realized using Web services deployed into a Globus container. Generally any middleware, e. g. , CORBA or MPICH [2], can be used for providing HOC implementations, as long as the format used for representing mobile code is supported by the chosen technology. We use the Java-based Lithium system in this paper, because our alignment application is also written in Java, and because it requires frequent communication between the scheduler and the workers, which can be handled more efficiently via RMI as done in Lithium than via SOAP. Framework implementations, like Lithium, should be distinguished from abstract component models, like CCA or Fractal where adaptations of components are principally possible, but it is not specified how to apply them. The messaging model we used for stopping the farm activity, whenever data dependencies prevented continuation, does not require anything more then a possibility for broadcasting messages among processes. It can therefore be realised in the same way, in a CCA implementation like CCaffeine or in Julia (The reference implementation of Fractal).

Possible alternatives to the described adaptation of the Farm-HOC for wavefront algorithms include replacing the Lithium farm scheduler by another one operating in a wavefront manner or adding a completely new wavefront HOC to our component framework. The first alternative would be valid only for the Lithium system and, moreover, we would hard-wire the wavefront behavior into the farm. The second alternative involves more overhead for the programmer than the adaptation of an existing component.

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. Since our wavefront steering thread can also be plugged into the scheduler of the Lithium library without uploading it remotely, our solution can be viewed as a novel way of introducing a new skeleton to a skeleton-library, without changing its implementation.

Acknowledgment

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 and Supportive Environments*. IEEE Computer Society Press, 2002.
2. Argonne National Laboratory. The Message Passing Interface (MPI). http://www-unix.mcs.anl.gov/mpi.
3. C.-I. Branden, J. Tooze, and C. Branden. *Introduction to Protein Structure*. Garland Science, 1991.
4. M. I. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. Pitman, 1989.
5. 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.
6. S. Gorlatch and J. Dünnweber. From grid middleware to grid applications: Bridging the gap with HOCs. In *Future Generation Grids*. Springer Verlag, 2005.
7. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1999.
8. V. I. Levenshtein. Binary codes capable of correcting insertions and reversals. In *Soviet Physics Dokl. Volume 10*, pages 707–710, 1966.
9. OASIS Technical Committee. WSRF: The Web Service Resource Framework, http://www.oasis-open.org/committees/wsrf.
10. C. Szyperski. *Component software: Beyond object-oriented programming*. Addison Wesley, 1998.
11. M. Wolfe. Loop skewing: the wavefront method revisited. In *Journal of Parallel Programming, Volume 15*, pages 279–293, 1986.
12. X.Huang, R. 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.