Integrating MPI-Skeletons with Web Services for Grid Programming

Jan Dünnweber[a], Anne Benoit[b], Murray Cole[b], Sergei Gorlatch[a]

[a]University of Münster, Münster, Germany

[b]School of Informatics, The University of Edinburgh, Scotland, UK

Interoperating components, implemented in multiple programming languages, are one of the key requirements of grid computing that operates over the borders of individual hardware and software platforms. Modern grid middleware like WSRF facilitates interoperabilty through service-orientation but it also increases software complexity. We show that Higher-Order Components (HOCs) provide a service-oriented programming abstraction over middleware technology. By offering the pipeline skeleton from the MPI-based *eSkel* library as a HOC, we show how machine-oriented technologies can be made available via Web Services on grids. We bind a Java-based Web application to the HOC to demonstrate its connectivity: user defined input can be transformed in a highly performant manner by running wavelet computations remotely on parallel machines.

1. Introduction

In a grid infrastructure computers of varying architectures are connected, so that any task in an application can be delegated to the most appropriate processing platform. Programmers targeting a grid currently face a tradeoff when choosing the implementation technology for their applications. Machine-oriented parallel technologies like C and MPI [6] provide good performance, but they narrow the range of the possible execution platforms. This is due to the fact that C is compiled into native machine code, which cannot be interchanged among different machines offering unequal instruction sets. Moreover, the use of function pointers, as it is required, e. g. , for parameterising MPI collective operations implies a tight coupling between different software components: code of library functions implementing a generic functionality must be present in the same address space as application-specific parameter code.

In contrast, a service-oriented architecture (SOA [5]) based on grid middleware such as WSRF [10] loosely interconnects clients and compute nodes. The communication is handled via Web Service requests and the required APIs for issuing and processing such requests are available for interpreted languages and also for C. Despite of the gained connectivity advantages, the use of Web Services for handling the entire communication in an application usually imposes a loss of performance. The messaging protocol employed by Web Services is SOAP, which requires the time-consuming composition, transmission and parsing of an XML-tree structure, even for elementary data exchange.

Our goal is to provide the performance of a light-weight messaging system within a heterogeneous, distributed environment. Therefore, we combine the more traditional performance approach to parallel programming using C and MPI with the recent SOA efforts in grid computing. Our work extends Higher-Order Components (HOCs), which were shown useful for programming grids using Java in [7]. In this paper, a HOC is composed of a Web Service and an MPI-program. We developed a Web Serivce that we call gateway, which bridges between MPI and SOAP and allows to provide a skeleton from a C-library (*eSkel* [3]) as a HOC abstracting over all MPI-communication.

The next section shows how HOCs can be integrated with MPI. Then, in Section 3 we introduce the case study of the discrete wavelet transform (*dwt*). Section 4 presents an imaging application using our MPI-based HOC. We explain our gateway service in Section 5 and conclude in Section 6.

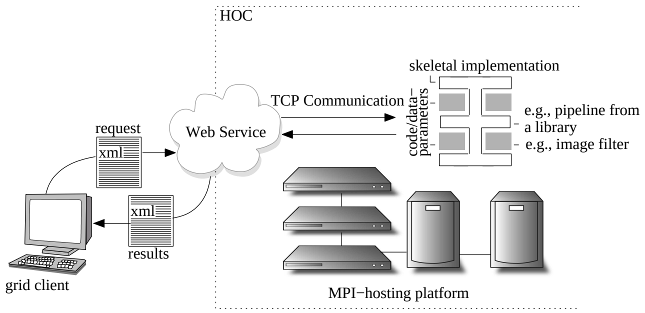


Figure 1. Abstracting over the runtime platform using a Higher-Order Component (HOC)

2. Integrating HOCs with C and MPI

In [7] we presented HOCs that abstract over the grid middleware and make the required middleware setup transparent for the user. HOCs offer a skeleton-like programming interface and include a grid-aware mechanism for shipping units of executable code across the network. HOCs and their parameters correlate to the skeleton model [2], but the implementation of a HOC takes into account the distinctive features of a SOA, e. g. , there is no standard format defined to exchange executable code between Web Services. The HOC corresponding to the map skeleton, e. g. , provides a service that applies functions in parallel to independent input. Its code parameter is the mapped function, which is portably represented by a string in the HOC implementation.

The most notable difference between a HOC and Web Service based job submission system, such as the Globus resource allocation manager (WS GRAM) or Unicore/WS [16], is that, in case of using a HOC, a skeletal implementation of a parallel algorithm is deployed upon the runtime platform before the HOC is used in an application. Figure 1 schematically depicts this scenario: Instead of the complete application code, the XML-data representing a request, which is uploaded by the client, only contains the code and data parameters that are specific for the given application. The parallel implementation for processing the request remotely in the grid can make use of MPI as suggested in the figure, or it can comprise multiple interconnected Web Services providing an alternative parallel processing platform, as described in [7].

Contrary to a typical MPI-application, where the client is running on top of the MPI platform itself as, e. g. process 0, the HOC client connects to a Web Service which maintains a TCP-connection to one process dedicated for handling the external communication (Section 5 explains this gateway in more detail). Thus, a HOC not only abstracts over the skeleton implementation, but it also decouples the client from the skeleton allowing both implementations to be exchanged without affecting each other's code, which promotes code reusability.

The interfaces of HOCs are designed such that application programmers can access them thinking in terms of skeletons, disregarding details about the runtime environment. Our experiments have shown that the abstraction offered by HOCs does not have a critical impact on the performance of this class of applications.

This paper deals with an application which has a pipeline structure: the discrete wavelet transform, which can be decomposed into multiple successive steps. Implementing this transform by mapping stages of the underlying pipeline model to different processors leads to frequent inter-processor communications, because of the fine grain of the single-stage operations. Therefore, a parallelisation should employ a light-weight message passing mechanism. MPI, as used, e. g. , by the pipeline skeleton in the C-library *eSkel*, is therefore a suitable candidate.

By embedding *eSkel*'s MPI-based pipeline into a Web Service that offers it as a HOC, we provide an interface for remote access via SOAP. The new HOC accepts parameter functions, which are shipped over the network and may be sent from a service consumer; the latter may be implemented in a programming language other than C. As an experiment, we connected the pipeline service to a Java-based Web interface which allows the user to upload and transform data via a Web browser.

3. Case Study: The Discrete Wavelet Transform

Wavelet transform is used in applications such as equalising measurements, denoising graphics and data compression. Such procedures are often applied iteratively to large amounts of data, which is time-consuming. Therefore a grid-enabled implementation which allows for the outsourcing of computations to high-performance multiprocessor servers is desirable. In an application, the transform is customised for a particular objective so that the transformed data exhibits properties which cannot be detected so easily in the source data. As an example, the contours in an image can be accentuated. Another popular application of wavelets is data compression via a customisation of the transform where the resultant data can be represented using less memory. Customising the wavelet transform is done by parameterising a general schema with application specific functions.

3.1. The Wavelet Lifting Scheme

Wavelet transforms are integral transforms, closely related to the (windowed) fast Fourier transform (*fft*), which decomposes a function into sines and cosines. The continuous wavelet transform (*cwt*) defined below projects function $f(t)$ onto a family of zero-mean functions (*wavelets* ψ):

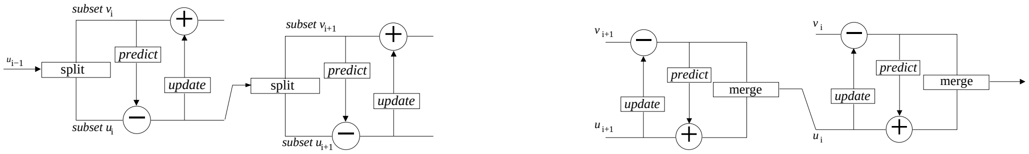
$$cwt(f;a,b) := \int_{-\infty}^{\infty} f(t)a^{-\frac{1}{2}}\overline{\psi}(a^{-1}(t-b))\mathrm{d}t \tag{1}$$

Instead of a continuous function, the discrete wavelet transform (*dwt*) processes a set x of samples (such as a list or a matrix) which are subdivided into two equally sized, discrete subsets u and v. The "lifting technique", proposed by W. Sweldens in 1994 [13], allows us to compute *dwt* iteratively as follows. Initially, $u_{0,j} = x_j$ for $j = 0..m_0$. The first index of $u_{i,j}$ (index i) represents the *lifting step*, and m_0 is the initial number of elements. $dwt(x)$ is computed by applying two functions called *predict* and *update* repeatedly, according to the following *lifting scheme*:

$$\begin{array}{ll} (u_i, v_i) := split(u_{i-1}) & \\ u_{i+1,j} := u_{i,j} - predict(v_{i,j}) & \text{for j} < \text{m}_\text{i} \\ v_{i+1,j-m_i} := v_{i,j-m_i} + update(u_{i+1,j-m_i}) & \text{for j} \geq \text{m}_\text{i} \end{array} \tag{2}$$

At each increment of i, index j iterates from 0 to $2.m_i$ to complete one *lifting step* (first step with $i = 1$). First, the set u_{i-1} (of size m_{i-1}) is split into subsets u_i and v_i, which are thus of size m_i ($m_i = m_{i-1}/2$). The *predict* function is then applied to the values in subset v_i ("predicting" the values in subset u_i). The samples u_i are then replaced by the differences between their predicted values and their actual values. These differences are processed by the *update* function and added to the samples in subset v_i ("correcting" it). Note that the workspace, i. e. , the data that is affected by subsequent steps is reduced in each lifting step: once computed, all $v_{i,j}$ values remain unchanged. The wiring diagram of two lifting steps in Figure 2(a) illustrates the structure of the lifting algorithm. The minus indicates that the input from the top is subtracted from the input from the left.

While the computation schema (2) is fixed, the functions *split*, *predict* and *update* can be customised. This customisation is done by the user, depending on the characteristics of the application.



(a) wavelet analysis (b) wavelet synthesis

Figure 2. The lifting scheme

If plain number series are processed, the *split* function can simply be defined to separate entries with odd and even indices. The choice of suitable *update* and *predict* functions requires making an appropriate assumption about the correlation of the single elements within the processed data.

3.2. Parallelising the Lifting Scheme

When the lifting scheme is applied to multiple independent data sets in parallel, the pipeline skeleton [2] can be used to parallelise the computation. The number of lifting steps that we apply to an input set (called the *scale* of the transform in classical wavelet theory [8]) depends on the size of the set (m_0): the number of steps is $log_2(m_0)$, since the input is bisected at each step. For a straightforward parallelisation, we use a pipeline wherein each stage corresponds to one lifting step and the number of stages is determined by the largest input set.

Wavelet transformation is reversible: the original input can be reconstructed from the transformed data using an inverse process, called the wavelet synthesis. In this context, the forward transform is usually called wavelet analysis. Our pipeline-based implementation of the wavelet transform allows us to run both a wavelet analysis and a wavelet synthesis. In the reversed schema, update and predict functions are swapped; updated values are subtracted and predicted ones are added as shown in Figure 2(b). A reverse pipeline with the same number of stages as the wavelet analysis pipeline can be used to reconstruct the source data. We use this output data for a comparison with the original input to verify the correctness of our implementation. The reverse process has another notable property: the workspace increases, since it re-introduces the stored v_{i+1} values to compute u_i (see Section 3.1).

3.3. An Application to Image Data

Figure 3 shows the effect of an example application of *dwt* on images. The input image is transformed up to the maximum scale and then reconstructed via the inverse transform as introduced in Section 3.2. The fractal image in Figure 3(a) is a Julia Set for $c = -0.16 - 0.65i$ (to construct

(a) an example input image (b) reconstruction (threshold 2.5)

Figure 3. Application of the transform on a grayscale fractal image

such diagrams, see [12]). It features very fine contours that become bolder in the reconstruction (Figure 3(b)), since all the pixel values below a given threshold are set to zero in the transform.

Contrary to the elementary splitting of number series, explained in Section 3.1, images require the splitting to be customised via a parameter function specifying a 2-dimensional partitioning. If we simply concatenate all rows or all columns of the image matrix into a 1-dimensional array, the image structure would be lost during the transform, as most neighbouring entries in the matrix are disjoint in such an array. Instead, we overlay the image with a lattice that classifies the pixels into two complementary partitions, preserving the data correlations.

4. Running the Pipeline HOC, customised for the Lifting Scheme

The middleware setup for running Web Service-based applications can be intricate. This is especially the case in the WSRF-context, where Web Service operations can affect component states, represented via resource properties. Moreover WS-N [10] is used for asynchronous messaging via service notifications. Thus, resources and notifications must be configured additionally to Web Services. Each setup step is error-prone and configuration files cannot be debugged by a stepwise execution in the manner of traditional executable code.

To provide a higher-level programming interface, we have developed a HOC which allows the wavelet lifting scheme to run in parallel. In Section 3.2, we have shown how to parallelise the lifting scheme using a pipeline skeleton. We explain here how this pipeline can be offered as a HOC and how it can be customised to run the image transformation presented in Section 3.3.

4.1. The Parameter Functions

In the wavelet lifting algorithm, the stages of the pipeline are defined through the parameter functions *split*, *predict* and *update*. For our imaging application, we define a *split* function which computes the so-called *quincunx* lattice. All the pixels of the processed image are alternately assigned to a subgroup of black pixels or white pixels. This quincunx pattern is just one possible partitioning among others. We refer to [9] for details on the implementation of such partitions.

The *predict* function rates the grayscale value of a pixel by computing the average of its nearest neighbours:

$$predict(x_{i,j}) = \frac{1}{4}(x_{i-1,j} + x_{i,j-1} + x_{i+1,j} + x_{i,j+1})$$

The corresponding *update* function returns half of the average computed by the *predict* function. The factor one half in *update* reflects the bisection performed by the *split* function in each lifting step. In this way, we preserve the average of the input during lifting, i. e. the grayscale value average over all pixels in both partitions equals half the average over all initial values. Some more sophisticated methods also bind neighbouring values, but with a different calculation rule. In any case, both *predict* and *update* can be represented via arithmetic expressions. These expressions can be encoded in an XML-compliant manner using, e. g. , *XPath*-expressions.

XPath is a language designed to select nodes, specify conditions and generate outputs in XML documents [18]. XML processing APIs, such as *Apache Xalan* [11], evaluate *XPath*-expressions and are available for multiple programming languages. We thus consider *XPath* as a suitable format for encoding functions and for exchanging them via the network.

The fractal image computation discussed in [7] uses a loop inside the skeleton parameter function. Loop statements are not supported within *XPath*-expressions and therefore this application required from the HOC a mechanism to transfer executable code without such limitations.

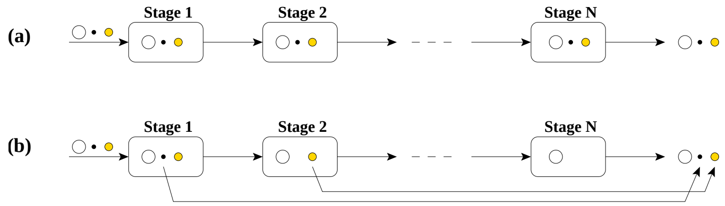


Figure 4. (a) Standard eSkel pipeline skeleton – (b) Stage skipping optimisation

As traditional HOCs support the transmission of parameters describing complex control flows, such as loops and nesting, in the format of Java or a scripting language, there is still a tradeoff between purely Java-based HOCs and potentially more efficient native skeletons, even if the latter can also be offered as HOCs.

4.2. The Stage-Skipping Optimisation

The HOC can be optimised to shortcut the lifting scheme in order to reduce the number of lifting steps in the pipeline. Indeed, the several inputs of the pipeline may be of varying sizes, and the number of lifting steps is directly related to their size. Hence, an input of short size does not need to go through all the stages of the pipeline. The adaptable design of skeletal components [4] helps to address this problem in our implementation.

The parallelisation described in Section 3.2 is not optimal because small-sized sets are to be passed through numerous pipeline stages, although no further processing is necessary. We could therefore envision an optimised pipeline skeleton which skips stages when needed. We perform the computation using the skeleton library *eSkel*, namely its pipeline skeleton which allows us to define so-called *explicit interactions*. In most of the skeletons libraries, the interactions between activities (i.e. the stages of a pipeline) are *implicit*: a pipeline stage is a function which takes input data as a parameter and returns one output for each input, thus being under constraint both in time and space. In *eSkel*, it is possible to define *explicit* interactions [1] between activities and release temporal constraints.

Figure 4(a) displays the standard pipeline behaviour with implicit interactions. The circles represent three examples of input items and the size of the circles is proportional to the size of the input. In our application the inputs of small size do not need to be processed by all the stages. Figure 4(b) presents the stage skipping optimisation, which allows small inputs to skip the unnecessary stages and to be directly sent as an output. As can be seen in Figure 4(b), the smallest input is finished in stage 1 and no more present in stage 2, while it is towed through all stages in Figure 4(a).

When the explicit interaction mode is used in *eSkel*, the user can control the timing of communications within the spatial constraints imposed by the skeleton. This means, a stage function does not need to receive all its input through its parameters and it does not necessarily pass output as a return value. Through direct calls to the *eSkel* functions `Give` and `Take`, new input can be retrieved or new output can be sent at any time within a stage function. For the skipping optimisation, we need to break the spatial constraints of the pipeline as well, i. e. any stage should be enabled to send an output not only to the next stage but also to the last.

5. The Gateway for Bridging between Web Services and MPI

In telecommunication terminology, a gateway is some hardware or software that addresses interconnection issues between systems using different protocols. For connecting our wavelet computation to a Web client, we developed a specially configured Web Service bridging between SOAP and an MPI-based pipeline. We call this service the gateway service and in the following, we explain its setup, which may be reused in other grid applications requiring an efficient pipeline implementation.

A particular feature of the gateway service is that it establishes a connection to an MPI-environment, which is external to the Web Service container and exhibits properties of its own. To maintain this connection the gateway service must store some *state data*, i. e., data persisting the execution of single operations such as `init`, `execute` and `getResults`. The minimum state data required for the gateway service consist in the output variables for holding results and the number of a TCP-port used for transferring application data between the MPI-environment and the Web Service.

While plain Web Services do not support state data at all, WSRF, as defined by OASIS [10], allows to bind a Web Service to state data. In order to use this feature, we deployed our HOC in the *globus-wsc-container* [15], which allows to run WSRF-compliant services written in C. The Globus middleware supports Web Service interfaces including a *resource property document*. This document defines, in XML-Schema [17] format, the structure of the state data persisting during the execution of single service operations. For our Pipeline HOC, we specified the resource properties of the gateway service corresponding to the parameters of the *eSkel* pipeline, i. e., except the mandatory data described above, we declared one string property per stage function, one array type property holding the time needs per process plus one integer property giving the number of processes to be executed within the external MPI-environment.

The separation of the MPI-environment from the Web Service container and the use of an extra communication channel (TCP in our implementation) inside the gateway is a necessity. Web Service containers like Globus are parallel applications which can process multiple requests simultaneously via multithreading. Therefore, they must not be run within a multiprocess environment like MPI themselves, which would lead to running one extra container per MPI-process, making resource sharing unfeasible and furthermore resulting in a process management overhead.

Our gateway service assembles a `command`-string wherein a platform dependent prefix holds the path to the MPI-installation directory, followed by `mpirun -np # pipeline` with the #-parameter reflecting the `numProcessors` resource property. Upon request of the `init`-operation, the service launches the MPI-program by calling `system(command)`. The MPI-program `pipeline` starts by opening a TCP server socket which accepts input in the form of number series or images. This connection is established only by process 0, i. e., processes with a higher rank wait until all input has arrived and is scattered amongst them.

6. Conclusion and Future work

This paper describes a high-level abstraction over native technologies on the grid using a Higher-Order Component (HOC). We implemented the lifting algorithm via sequential pipeline stages and applied it to multiple independent tasks in parallel. The choice of MPI was motivated by the fine grain of the computations in this application, which are not eligible to be dispersed across the grid. We implemented a Pipeline HOC allowing for local parallelism and for remote access. This HOC was customised for an application of the discrete wavelet transform. We also proposed a solution to avoid portability problems in the grid environments, where parameters must be exchangeable between different software components. When a well-defined format is used for representing parameters, even the presence of multiple protocols and programming languages within a single system does not put insuperable barriers in the way of communication between services and clients. We identified *XPath* as a suitable format for customising HOCs rather than using executable code, which would restrict our HOC implementation to the use of a single programming language.

Customising components by transferring user-defined functions across the network is a new usage for the *XPath* language, applicable in multiple domains. In our future work, we plan to integrate *Apache Xalan* for this purpose with our HOC, which currently still requires to hard-wire the definitions of Section 4.1 to run a particular transform.

We also proposed an adaptation of the pipeline skeleton, which optimises its behaviour when it is used in a lifting scheme application. To simplify such adaptations in *eSkel*, we plan to define a new function `GiveToLastStage`, which will allow any stage to communicate with the last one.

Finally, we plan to extend our HOC-based SOA approach to other skeleton implementations that use native machine code.

Acknowledgements: This work has been performed under the Project HPC-EUROPA (RII3-CT-2003-506079), with the support of the European Community - Research Infrastructure Action under the FP6 “Structuring the European Research Area” Programme and the EPSRC project Enhance (under grant number GR/S21717/01).

References

[1] Anne Benoit and Murray Cole. Two fundamental concepts in skeletal parallel programming. In P. Sloot V. Sunderam, D. van Albada and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 2005) , Part II*, LNCS 3515, pages 764–771. Springer Verlag, 2005.

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

[3] Murray I. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. In *Parallel Computing 30*, pages 389–406, 2002.

[4] Jan Dünnweber, Sergei Gorlatch, Sonia Campa, Marco Danelutto, and Marco Aldinucci. Adaptable components for grid programming. In *IEEE International Grid Computing Conference*, 2005. Submitted.

[5] Thomas Erl. *Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR, 2004.

[6] Ian Foster. *Designing and Building Parallel Programs. Concepts and Tools for Parallel Software Engineering.* Addison Wesley, 1995.

[7] Sergei Gorlatch and Jan Dünnweber. From grid middleware to grid applications: Bridging the gap with HOCs. In *Future Generation Grids*. Springer Verlag, 2005. to appear.

[8] Barbara Burke Hubbard. *The world according to wavelets*. A K Peters Ltd., Wellesley, MA, 1998. second ed.

[9] Arne Jensen and Anders la Cour-Harbo. *Ripples in mathematics: the discrete wavelet transform*. Springer Berlin, 2001.

[10] OASIS Technical Committee. WSRF: The Web Service Resource Framework, http://www.oasis-open.org/committees/wsrf.

[11] Apache Organization. Apache xalan. http://xml.apache.org/xalan-c.

[12] Heinz-Otto Peitgen and Peter H. Richter. *The Beauty of Fractals, Images of Complex Dynamical Systems*. Springer-Verlag New York Inc, June 1996.

[13] Wim Sweldens. The lifting scheme: A custom-design construction of biorthogonal wavelets. *Appl. Comput. Harmon. Anal.*, 3(2):186–200, 1996.

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

[15] The Globus Alliance. GT 4.0: C WS Core. http://www.globus.org/toolkit/docs/4.0/common/cwscore.

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

[17] World Wide Web Consortium, W3C. The XML Schema definition language recommendation. http://www.w3.org/XML/Schema.

[18] World Wide Web Consortium, W3C. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20.