Native Services for Grid Applications

Native Services for Grid Applications

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 features of service-oriented architectures. Middleware, like the WSRF implementation in the Globus Toolkit 4, facilitates service-oriented grid applications and thereby widens the potentials of traditional approaches to component-based grid programming focusing primarily on code reuse and the separation of concerns. In a SOA, every concern can be addressed using the most appropriate implementation technology. Such integrated infrastructures possibly increase a system's efficiency, but, in the majority of cases, also its complexity. Using the discrete wavelet transform as an example application, we show that skeletal programming allows to handle systems comprising a multitude of different technologies in a structured manner. The computational core of our system is implemented using *eSkel*, a C/MPI-based library for parallel programming, which we embedded into a grid service and connected to a Java-based web application presenting the user with an abstract, convenient and high level interface.

1. Introduction

Writing large-scale programs using the low-level MPI functions `send` and `recv` is hardly reasonable for grid programmers, because by this means even simple applications quickly become unmanageable with an increasing number of processors as demonstrated in [10]. MPI collective operations and *algorithmic skeletons* [2] can abstract over process communication details, however even user-friendly skeleton libraries, e. g. , *eSkel* [3] still require from the programmer to arrange code accordingly to the MPI runtime environment, i. e. , an explicit distinction of processes and associated behaviors has to be specified, accordingly to a *rank* determined via an MPI primitive.

Another drawback of skeletons libraries is that the transfer of code parameters is typically handled using the low-level mechanism of passing function pointers. These pointers refer to code units present in the same address space as the skeleton, which tightly couples the application to the service provided by the skeleton.

Despite of the depicted difficulties, some time critical applications are still inconceivable without C and MPI, mostly because of performance reasons. However the programming paradigms for performance-critical applications have been shifted from machine-oriented technologies like C towards service-oriented ones like the WSRF [13] in the context of heterogeneous and widely dispersed platforms like computational grids. Instead of optimal processor utilization, service-orientation deals with communication issues, such as message exchange across LAN boundaries and interoperability between component-based [**?**] and legacy systems.

A drawback of SOA is that providing and consuming services via middleware like Globus [7] requires numerous tedious low-level configuration steps. Each of this steps is error-prone, since configuration files cannot be debugged by a stepwise execution like executable code. This elaborate setup is necessary, since the service ports must be declared in an implementation-independent manner. In [4], we presented Higher-Order Components (HOCs) that can abstract over grid middleware, offer a skeleton-like programming interface and include a grid-aware mechanism for shipping units

of executable code across network boundaries.

In basic HOC applications, such as the one computation discussed in [4], independent computational steps outlast vast quantities of iterations. An efficient parallel execution is therefore possible via task farming, where no communication among processes is required during the most compute intensive phase of the application. Our experiments have shown, that the abstraction offered by HOCs does not have a critical impact on the performance of farm applications.

In this paper, we...pipeline/wavelet/eSkel...

The central question we study thereby is: can we provide a service-level abstraction over native implementation technologies by creating a framework of parallel software components plus a set of possible customizations?

In the next section, we introduce the case study of the discrete wavelet transform (*dwt*) using the pipeline skeleton in eSkel. We separate the inherent parts of the algorithm that must be supported on the lowest level of the used system and the parts that are adapted in each application and can be expressed via customizations. Section 3 shows how the concrete customizations look like in our example. Experimental results are shown in Section 4. Section 5 summarizes our results and discusses options for portable representations of the presented customizations.

2. Case Study: The Discrete Wavelet Transform

Wavelet analysis includes applications such as equalising measurements, denoising graphics and data compression that must often be applied iteratively to large amounts of data. Therefore an efficient parallel implementation is desirable. The wavelet transform is reversible: the original data can be reconstructed from the transformed data using an inverse process, called the wavelet synthesis. In an application, the transform is customised 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, or the transformed data can be represented using less memory. This customisation is done by parameterising a general schema with application specific functions, making it an ideal candidate for experimenting with higher-order constructs such as skeletons, HOCs or customisable services in general. We introduce the wavelet lifting transform, then we show how to parallelise the schema, and finally illustrate the transform on images in Section 2.3.

2.1. The Wavelet Lifting Schema

Wavelet transforms are integral transforms, closely related to the (windowed) fast Fourier transform (*fft*). While *fft* decomposes a function into sines and cosines, the continuous wavelet transform is defined as

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

Here, the function $f(t)$ is projected onto a family of zero-mean functions (*wavelets*).

Instead of a continuous function, the discrete wavelet transform (*dwt*) processes a set x of samples (such as a list or a matrix) that can be split into two equally sized subsets u and v, each holding m elements. The "lifting technique" [14] was discovered by W. Sweldens in 1994.

Initially, $u_{0,j} = x_j$ for $j = 0..2m$. The first index of u (0) represents the *lifting step*. $dwt(x)$ is computed by applying two functions called *predict* and *update* repeatedly, according to the following

lifting scheme:

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

At each increment of i, index j cycles from 0 to $2m$ to complete one *lifting step* (first step with $i = 1$). Firstly, the set u_{i-1} is split into subsets u_i and v_i. The *predict* function is then applied to the values in subset v_i ("predicting" subset u). 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 ("correcting" it).

While the schema is fixed, the functions *split*, *predict* and *update* can be customised. This customisation is done by the user, who has to consider characteristics of the application to find suitable assignments for the three functions. 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 for an application requires making an appropriate assumption about the correlation of the single elements within the processed data, e. g. to take advantage of linear or polynomial dependencies.

2.2. Parallelising the Schema

The wiring-diagram of two lifting steps in Figure 1 graphically illustrates the structure of the lifting algorithm introduced formally in Section 2.1. The minus means that the input from the top is subtracted from the input from the left.

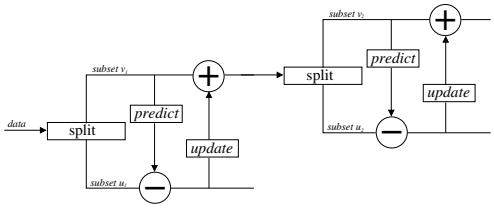


Figure 1. The lifting scheme, forward transform

When the algorithm is applied to multiple independent data sets in parallel, the pipeline skeleton [2] can be used to parallelise the algorithm. The number of lifting steps that we apply to an input set (called the *scale* of the transform in classical wavelet theory [11]) is limited. If the first splitting produces subsets of m elements, the maximum number of steps is $log_2(m) + 1$, as the input is bisected in each step. For a straightforward parallelization, we use a pipeline wherein each stage corresponds to one lifting step and the total number of stages is determined by the largest input set.

It can easily be verified that this basic setup works by reversing the schema: update and predict functions are swapped and updated values are subtracted and predicted ones are added as shown in Figure 2. A reverse pipeline with the same number of stages as the transform pipeline can be used for a synthesis to reconstruct the source data.

2.3. An Application to Image Data

Figure 3 shows the effect of a simple 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

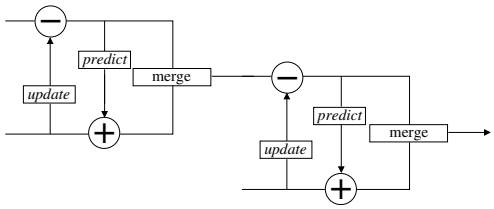
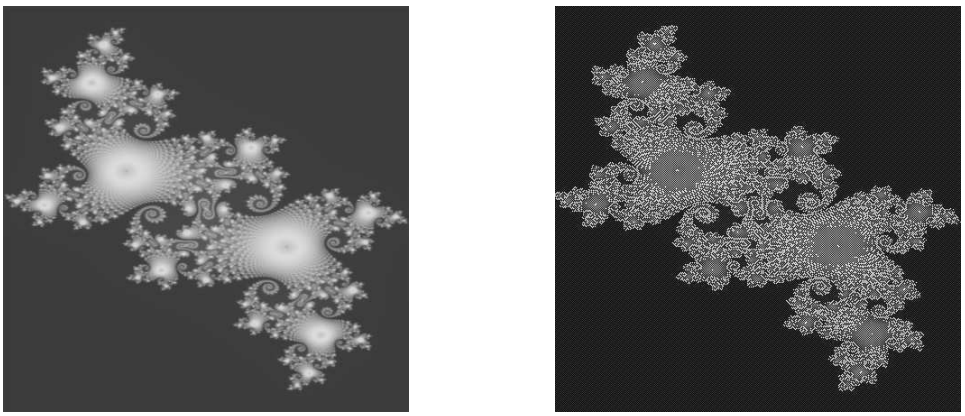


Figure 2. Data synthesis using the backward transform

Section 2.2. By setting all pixel values below a given threshold to zero in the transform, some detail information has been blanked in the reconstruction in Figure 3(b). The fractal image used in this example features very fine contours that become bolder in the replication. However the original structure can still be recognised.



(a) a Julia Set
for $c = -0.16 - 0.65i$

(b) reconstruction
with threshold 2.5

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

Contrary to number series images require a customisation of the split function that specifies an adapted 2-dimensional partitioning. If we simply concatenated all rows or all columns of the image matrix into an array, the image structure would get lost during the transform, as most neighbouring entries in the matrix are disjoined in such an array. Instead, we overlay the image with a lattice that classifies the pixels into two complementary partitions, preserving the data correlations. Some more details will be provided in the next section.

3. Customising the Image Transformation Service

This section focuses about the application and behaviour customisation of the wavelet lifting scheme. We explain the customisations for the image transformation example presented in Section 2.3.

We have explained in Section 2.2 how to parallelise the lifting scheme using a pipeline skeleton. This is done through a mapping of the lifting steps to the stages of the pipeline. The *application customisation* consists in defining the pipeline stages according to the application. For the wavelet transform application, we therefore need to define and code the functions *split*, *predict* and *update*.

We propose as well a *behaviour customisation*, which consists in shortcutting the lifting scheme in order to reduce the number of lifting steps. 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.

3.1. Application Customisation

For the image transformation, we define a *split* function which computes a so-called *quincunx* lattice. All the pixels of the processed image are alternatively assigned to one subgroup of black pixels or to a subgroup of white pixels arranged like a chessboard, i. e. , the color pattern is shifted by one pixel in each row. This quincunx pattern is just one possible partitioning among others that use e. g. , hexagonal or octagonal lattices. We refer to [12] for details on the implementation of such partitions.

In this image application, the *predict* function rates the grayscale value of a pixel by computing the average of its nearest neighbours. This computation is done with the following predict function.

```
double predict(int i, int j, matrix m) {
  int neighbourNum;  double neighbourSum;
  neighbourNum = numNeighbours(i, j);
  neighbourSum = sumNeighbours(i, j, m);
  return neighbourSum / neighbourNum;
}
```

Each pixel has between 2 and 4 nearest neighbours, depending on its position. The function `numNeighbours` returns this number, for example the result is 4 for a non-border pixel. The function `sumNeighbours` returns the sum of the grayscale values of the neighbouring pixels. For the pixel at position (x, y), the nearest neighbours are the pixels at positions $(x + 1, y)$, $(x, y + 1)$, $(x - 1, y)$, and $(x, y - 1)$.

The corresponding *update* function returns half of the average computed by of the *predict* function. The code is almost the same as above, except the return statement which is `return neighbourSum / 2 * neighbourNum` for this function. The factor one half 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.

3.2. Behaviour Customisation

The parallelisation setup described in Section 2.2 is non-optimal because small-sized sets are to be passed through numerous pipeline stages, although no further processing is necessary. To avoid such inefficiencies, we introduced the behaviour customisation concept in [5]. The idea is based on the classic *observer* pattern. According to the standard design pattern catalogue [6], this pattern can be implemented using an object-oriented programming language as follows. If the state of a processed data object, which is represented via its attribute values, changes during a computation, a user-defined *callback* function is invoked.

We could therefore envision a *lifting scheme* customised skeleton which skips stages when needed. We developed our application using the skeleton library *eSkel* and its pipeline. Even though this approach is not object-oriented as in [5], it is possible to customise *eSkel*'s pipeline in order to apply a stage-skipping optimisation.

In most of the skeletons environments, the interactions between activities (i.e. the stages of a pipeline) are *implicit*. In this case, a pipeline stage is a function which takes input data as a parameter and returns one output for each input. In *eSkel*, it is possible to define *explicit* interactions [1] between activities to express a more complex behaviour of the application. We can for example filter the data inside a pipeline stage or generate additional output.

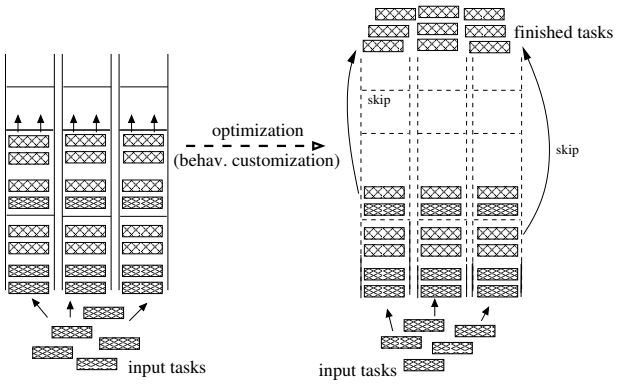


Figure 4. Behaviour customisation of eSkel; left: standard pipeline skeleton; right: lifting with stage skipping

Anne: we should still work a lot on the text below and the figure...

The parallel pipeline behaviour implied with *predict* and *update* functions controlled by the skeleton is depicted in the left part of Figure 4. Although half of the samples in each task becomes obsolete in each lifting step, they are towed through all remaining stages, thus congesting the pipeline more and more.

The more efficient alternative is shown in the right half of Figure 4. The *update* function is arranged for an explicit interaction by setting the according flag during the skeleton setup. This flag indicates that the propagation of input and output data is handled within the customising code using *eSkel*'s `Give` and `Take` functions. These functions block the execution of customising code and wait until their arguments have been processed by the skeleton or respectively until the skeleton provides new input. Whenever the *update* function has processed a single sample, the task it belongs to is immediately finished by a call to a function called `GiveToLastStage`. This function skips all the remaining pipeline stages. The according initialisation procedure copies all samples into task data records with a number of elements that is a power of two and pads unused elements with zeroes. This ensures that the successive bisection process in the pipeline always results in a single sample for each task.

4. Experiments

To evaluate the performance of our native dwt implementation and especially the effectiveness of the stage skipping optimization shown in Section 3.2, we deployed the described image processing

service described in Section 2.3 to the HPCx system [15] at the Edinburgh parallel computing center (*EPCC*), composed of 1600 1.7 GHz POWER4 processors with a throughput of at least 4.8 Tflops (4800 AU/hr). In our tests, we employed one processor per input data record.

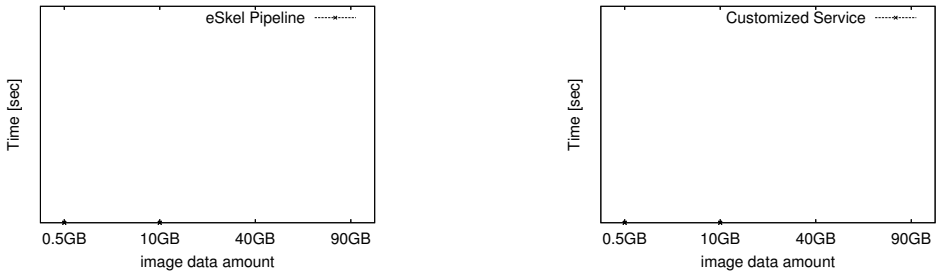


Figure 5. Experimental results for the image processing application.

5. Conclusion and Outlook

Parallel wavelet lifting using MPI was analyzed in [8]. Here, a parallel application of the lifting algorithm for single tasks is shown that works similar to the parallel implementation of *fft* presented in [9]. The implementation of *dwt* presented in this paper is simpler than the parallel *fft* implementation in [9] as, in the present work, there is no parallelism within a single transform. We implemented the lifting algorithm via sequential pipeline stages and applied it to multiple independent tasks in parallel. Contrary to the presentation in [8] that only covers the basic implementation of dwt in parallel, this paper also shows an application of *dwt* and the image processing experiments in Section 4 prove that large amounts of data can be processed very efficiently using the customized pipeline. Moreover the skeletal structure allows us to adapt the dwt procedure by customizing the *split*, *predict* and *update* functions, which cannot be done so easily in the approach presented in[8].

Let us come back to our motivating quest for an high-level abstraction over native technologies. The customizations presented in Section 3 can be completely specified without writing any MPI code. However all customizations have been represented via C-Code. So far, we also have not shown a better code transmission technique than the standard C-mechanism of passing function pointers, which requires the complete code of each application to be present on all machines involved in a distributed execution.

The technical barrier of crossing network boundaries, when the implementation is embedded into a distributed infrastructure using , e. g. , web services can be addressed using a little trick: we have written a *cut-out*-procedure that copies functions into arrays which can be posted across networks using the SOAP protocol. The conversion from an array back into a function is possible via a typecast. Unfortunately, we have not found any better solution to determine function boundaries than scanning the binary code of a function for the return statement in assembly language. Of course, a workaround like this is not portable.

By reviewing our customizations carefully, it can be seen that the *predict* and *update* functions as well as the skipping criterion in the behavior customization can be represented via a simple arithmetic expression, so we can write a simple parser for that purpose and pass the customizations

as character strings. A portable format for splitting lattices might be possible via e. g. an XML-representation of graphs.

Thus, we conclude that our two phases customization concept is a promising approach to abstract over MPI-like primitives as it is required for serious large-scale application programming.

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 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, September 2004.

[5] Jan Dünnweber, Sergei Gorlatch, Sonia Campa, Marco Danelutto, and Marco Aldinucci. Adaptable componets for grid ptrogramming. Submitted, 2005.

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

[7] Globus Alliance. http://www.globus.org.

[8] Patricia Gonzlez, Jos C. Cabaleiro, and Toms F. Pena. Parallel computation of wavelet transforms using the lifting scheme. *J. Supercomput.*, 18(2):141–152, 2001.

[9] Sergei Gorlatch. Programming with divide-and-conquer skeletons: an application to FFT. *J. Supercomputing*, 12(1-2):85–97, 1998.

[10] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM TOPLAS*, 26(1):47–56, 2004.

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

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

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

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

[15] University of Edinburgh, Computational Science and Engineering Department. http://www.hpcx.ac.uk.