A Low Level Component Model enabling Resource Specialization of HPC Applications

Julien Bigot, Zhengxiong Hou, Christian Pérez, Vincent Pichon

▶ **To cite this version:**

Julien Bigot, Zhengxiong Hou, Christian Pérez, Vincent Pichon. A Low Level Component Model enabling Resource Specialization of HPC Applications. [Research Report] RR-7966, INRIA. 2012, pp.20. hal-00698573

HAL Id: hal-00698573

https://hal.inria.fr/hal-00698573

Submitted on 16 May 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Low Level Component Model enabling Resource Specialization of HPC Applications

Julien BIGOT, Zhengxiong HOU, Christian PÉREZ, and Vincent PICHON

RESEARCH REPORT
N° 7966
16 Mai 2012
Project-Team Avalon

ISSN 0249-6399 ISRN INRIA/RR--7966--FR+ENG

A Low Level Component Model enabling Resource Specialization of HPC Applications

Julien BIGOT, Zhengxiong HOU, Christian PÉREZ, and Vincent PICHON

Project-Team Avalon

Research Report n° 7966 — 16 Mai 2012 — 17 pages

Abstract: Scientific applications are still getting more complex, e.g. to improve their accuracy by taking into account more phenomena. Moreover, computing infrastructures are continuing their fast evolution. Therefore, software engineering is becoming a major issue to achieve easiness of development, portability, simple maintenance, while achieving high performance. Software component model is a promising approach, which enables to manipulate the software architecture of an application. However, existing models do not capture enough resource specificities.
This paper proposes a low level component model (L^2C) that supports directly native connectors such as MPI, shared memory and method invocation. L^2C is intended to be used as a back end by a "compiler" (such as HLCM) to generate an application assembly specific to a given machine. This paper shows on a typical domain decomposition use case that L^2C can achieve the same performance as native implementations, while gaining benefits such as enabling resource specialization capabilities.

Key-words: HPC, Component, Software Architecture, L^2C, Grid'5000, Domain decomposition

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Un modèle de composant logiciel de bas niveau permettant la spécialisation d'applications haute performance en fonction des ressources

Résumé : Les applications scientifiques continuent de devenir de plus en plus complexes, par exemple pour améliorer leur précision en intégrant davantage de phénomènes à simuler. Par ailleurs, les infrastructures de calcul continuent leur rapide évolution. Ainsi, l'ingénierie logicielle devient un défi très important afin de permettre une facilité de développement, la portabilité des codes, et une maintenance acceptable tout en permettant de hautes performances. Les modèles de composants logiciels offrent une approche prometteuse en permettant de manipuler l'architecture logicielle d'une application. Cependant, les modèles existant ne permettent pas de capturer suffisamment les spécificités des ressources de calcul.

Cet article propose un modèle de composant logiciel "bas niveau" (L^2C) qui permet l'intègration native de connecteurs tels que MPI, la mémoire partagée ou l'invocation de méthode. L^2C est destiné à être utilisé en tant que langage de sortie d'un "compilateur" (tel que HLCM) générant un assemblage d'une application spécifique à une machine et à une exécution. Cet article montre sur un cas d'étude typique de décomposition de domaines que L^2C permet d'atteindre les même performances que les applications natives, tout en offrant des possibilités d'optimisation par rapport aux capacités des ressources.

Mots-clés : HPC, Composant, Architecture logicielle, L^2C, Grid'5000, Décomposition de domaines

1 Introduction

Scientific applications require more and more computing power to simulate an increasing number of phenomena with increased accuracy. Computing power is provided by parallel hardware resources such as super-computers, clusters of nodes with multi-core CPUs, GPUs, etc. On one hand, these various parallel hardware architectures do offer very distinct features when it comes to memory models or support for communications, for example. This means that in order to obtain high performance, codes have to be specifically optimized for a targeted hardware. On the other hand, the implementation and validation of codes by specialists of the simulated domain is an expensive process. This means that once written, codes tend to be adapted and reused on different hardware and in different applications. Without careful attention, these variations of the codes tend to be maintained independently with very limited code sharing between them which leads to duplication of efforts. This could be avoided by using a suitable programming model to enable adaptation to various hardware while allowing reuse of the non hardware specific codes.

A common approach is to use a wrapping language around computation intensive kernel. Examples of wrapping languages are `C++` or `Python`. However, these imperative languages have been shown to have drawbacks when dealing with large codes, in particular with respect to code re-use, maintenance, and code evolution [1]. This has led the conception of a new approach for software development: component-based software engineering [1] that aims at avoiding these drawbacks. It focuses on the decoupling of the various aspects of an application into independent components with clearly identified points of interaction. This could be used to implement the separation of concerns between domain specific codes and hardware specific codes.

While software components offer many advantages, their use in High Performance Computing (HPC) is not as widespread as one could expect. One reason seems to be that some models introduce overheads at runtime that are not acceptable in HPC while other models do not offer the level of abstraction that would allow to adapt applications to distinct hardware resources.

This paper proposes the Low Level Components (L^2C), a component model close to hardware abstractions. It aims to be easily extensible to integrate distinct HPC hardware resources. The current version supports without overhead interactions such as native method call (C++ and soon FORTRAN), MPI, and remote method invocation (CORBA).

As L^2C aims to be very close to machine abstractions, it does not provide an adequate programming model for developing applications: it should not be directly used by a programmer. L^2C is intended to be used as a target model by a compiler. For example, we use L^2C in conjunction of High Level Component Model (HLCM) [2]. HLCM is an abstract component model that supports hierarchy, genericity, connectors, and component and connector implementation choice. Being abstract, HLCM needs a transformation process to generate a concrete application. During this phase, resource specificities can be taken into account.

However, before developing optimization algorithms for such a compiler, the question this paper aims to answer is whether L^2C can provide high enough performances and to studies which kinds of L^2C assemblies are well suited for various kinds of resource infrastructures. This evaluation is performed on a well known discrete differential equation approach, the heat equation using the Jacobi method.

The remaining of the paper is organized as follow: Section 2 presents the background and the related work while Section 3 describes L^2C. Section 4 analyzes how L^2C can be used to efficiently implement multiple variations of a benchmark Jacobi application on various hardware resources while Section 5 experimentally evaluates some metrics such as code reuse, performance, overhead and complexity. The applicability of the approach to a real world application is discussed in Section 6. Section 7 concludes the paper.

2 Background

This section introduces some challenges for the programming models of high performance computing applications and analyzes related work.

2.1 Programming Model Challenges

Scientific applications offer a complex challenge from a programming model point of view. Firstly, they require high computing power, which means that the programming model should introduce as little overhead as possible and that fine tuning is needed to take advantage of the hardware used to execute them. Secondly, their long life cycle means that they will very likely run on a wide range of parallel architectures ranging from supercomputers to clusters of NUMA nodes but also to future generation hardware not yet designed. Moreover, specialists of the simulated domain are usually not parallel computer experts.

For example, many real world phenomena can be simulated by the resolution of partial differential equations. A common approach is to use the finite difference method that offers a good spatial and temporal locality. It is usually parallelized using a decomposition of the spatial domain into subdomains, where each subdomain is assigned to a computing core. The computation iterates over time and the value of each subdomain is computed based on the value of this subdomain and its neighbors at the previous iteration.

However, many variations of the hardware can impact the performance of the application such as the number of cores, their computing power and their amount of memory. Another variation concerns the interconnection between these cores, with shared memory or via a network, the affinity between memory banks and cores [3] and the network topology.

These hardware variations require adaptations of the application to provide high performance. For example, for an application based on domain decomposition as previously introduces, the number of threads, the subdomains they handle and their placement on computing cores have to be chosen. Communications methods must also be chosen. They can include implicit memory sharing with synchronization, in-memory copy or message passing over a high speed network or a wide area network. The most efficient approach for a particular configuration may require a combination of several methods.

In order to support these variations, a programming model should ease the adaptation of applications to various hardware resources with the highest possible degree of reuse between versions, while not impacting performance or ease of development. This means that the model should support the separation of concern between the development of simulated domain specific code and optimizations for the various hardware architectures. The next subsection presents how existing programming models handle the separation of concern, performance and ease of development.

2.2 Existing programming models

Infrastructure specialized models A very common programming model for parallel applications is the use of a message passing library such as MPI [4]. Such a library offers a quite low level of abstraction to the developers but for collective operations. However, MPI assumed a flat and homogeneous network model between processes. Thus, complex strategies have to be designed to adapt an application to hardware [5] with few possibilities to let an application self-adapt to resources [6].

A second common programming model is multithreading which enables to efficiently make use of shared memory multi-core machines. Synchronization primitives such as POSIX Threads [7] also offer a level of abstraction very close to the hardware. Although there is many research work, there is not yet a standard for managing memory placement. As multithreading programming is difficult and error prone, many developers adopt a higher level language such as OpenMP [8]. From the point of view of the domain code developer, an OpenMP parallel program may look like similar to its sequential counterpart. In the case of the domain decomposition code, it consists in adding a few annotations to the code to let the loops that iterate over the domain be parallel and to control the degree of parallelism.

As one of the main computing infrastructures is interconnected multicore machines, applications based on MPI and OpenMP is an active field of research. MPI and OpenMP support two complementary models of parallelism. It must however be noted that both models require modifications waved with the domain code.

Thus, depending on the targeted machine, applications need quite a large effort to be adapted to MPI, Pthreads, OpenMP, or a mixed of these. This has even become worst with the advent of

GPGPUs that add another programming model, usually CUDA or OPENCL or now OPENACC. However, some work aims to transparently handle GPGPU through OPENMP [9].

Infrastructure agnostic models Some parallel programming models such as CHARM++ [10] or more generally Partitioned Global Address Space (PGAS) languages such as UPC or Co-Array FORTRAN [11] aim at offering a model that is able to compile and execute on various infrastructures. While this goal seems to be reached, it lacks a simple model for code reuse and maintenance [1]: they do not offer any mechanism to deal with the architecture of an application.

Software Component Models Software component models have been seen as an improvement over object oriented programming as they enable to manage the architecture of an application [1]. Many large sequential codes such as Eclipse (OSGi [12]), Firefox (XPCOM [13]) or OpenOffice (UNO [14]) are based on component models. Similarly, there are specialized models for distributed computing such as the CORBA Component Model (CCM) [15] or the Grid Component Model (GCM) [16]. These models support decomposition of applications as independent components that interact through a set of well defined interfaces.

However, these models only enable to describe a concrete component assembly, i.e. an assembly where all components are primitive and so also connections between them. GCM provides some supports for adaptability by enabling to control the mapping between abstract component containers and runtime containers [16]. Therefore, it enables the optimization of the placement of components on available resources. Another issue is that to be portable most of these models usually fix the kinds of supported interactions: usually they only support remote method invocation (RMI) with some support for heterogeneity, such as OMG IDL for CCM. However, it adds unnecessary overhead for components written with the same programming language and collocated within the same process, for example.

To satisfy the constrains of HPC applications, dedicated models have been proposed. A well-known model is the Common Component Architecture (CCA) [17]. Some implementations such as CCAFEINE have been conceived as process-local (*i.e.* without network transparency) in order to minimize runtime overhead. Support for inter-language calls is provided by the BABEL library that is used for inter-components interactions. When support for remote method invocations in addition to inter-language calls has been added to BABEL it has made its way into CCAFEINE. Though quite small, BABEL still introduces overheads for calls between components located within the same process that limit the granularity of CCAFEINE components. For parallelism oriented interactions, CCA components are expected to rely on external models such as MPI, but it does not appear in the interface of components. Thus, it does not help a lot to adapt an application to various machines as parallelism support is basically the same as with a pure MPI approach.

Analysis Models or API such as OPENMP, MPI and their combinations can be well suited for a specific kind of hardware respectively. Component models offer an interesting approach to let applications be adaptable. However, existing component models either imply too much overhead at execution for HPC applications or do not offer enough abstraction to let application be efficiently adapted to different hardware resources.

3 Low Level Component Model (L^2C)

An application can make use of various kinds of interactions that range from local method calls to interactions with language and network transparency such as MPI or CORBA and intermediate choices in term of overhead and functionality such as BABEL. The choice of a given type of interaction for inter-component communications impose for the component model designers to make a choice between efficiency and portability. There is not any good trade-off as these two properties are usually not compatible.

We advocate for another approach that consists in relying on a compilation process that enables to define a component model for program-

mers and another component model for execution. A transformation process is responsible for generating the "executable" assembly from a "source" assembly. The definition of a component model for the "source" assembly is out of scope of this paper. An example of such model is High Level Component Model (HLCM) [2].

A consequence of the proposed approach is that a model for the "executable" is also needed. As this model is not intended to be used by a programmer, its main property shall be to support native inter-component communications. Therefore it shall be able to support different forms of interactions. This is the goal of the Low Level Components (L^2C) model.

3.1 Overview of the L^2C model

The Low Level Components (L^2C) is a model that we designed to support multiple kinds of interactions between components with runtime overhead reduced to their minimum. This is achieved by having a very simple runtime with only three operations: component creation and destruction, component configuration (include connection establishment), and the transfer of the initial control to at most one component instance in each process. Once the control has been passed to a component, no more L^2C code is encountered in the execution path, thus ensuring the absence of L^2C related overhead after deployment.

A L^2C component is described by some very lightweight metadata in the code. It enables it to be instantiated, destroyed and gives read or write access to configurable elements. The read accesses enable to retrieve information from the components, usually in order to configure another instance. The write accesses enable to configure the component with user provided data or to configure it with elements coming from another component for connection purpose.

For example, the current L^2C implementation supports three kinds of interactions: C++, Corba, and MPI. C++ local method calls are supported by copying a pointer from the component providing the service to the component using the service. Ongoing work is applying the same approach to fortran interactions. Corba inter-process method calls are supported by similarly copying Corba references. MPI communications are supported by providing a MPI communicator though a point of interaction representing a communication group. Adding support for new kinds of interactions does not require any complex change to the implementation. It may just need some code to simplify the handling of the new type of element that should be configured.

An L^2C application can be build *by programming*, using an API to create and connect components. A more suitable alternative for our approach is to *describe* an L^2C assembly through a dedicated (XML) file. Such a file contains a complete description of component instances and connections.

Details on the implementation of L^2C can be found at the website `http://hlcm.gforge.inria.fr/l2c:start`.

3.2 Discussion

L^2C is a very basic component model, that only differs from other component models by its abilities to define setter and getter operations associated to interactions points. Hence, it does not limit the kinds of interactions it supports. This property was looked after to be able to add other kinds of primitive interactions without adding runtime overhead.

As L^2C aims to describe a version of application specific to a given machine, L^2C is not intended to be directly used by a programmer. It is intended to be used as the output of a compilation/transformation process. In particular, L^2C was designed to be the target model of HLCM [2, 18], a high level component model, though it is not specific to HLCM.

4 Usability of L^2C on a Jacobi application

In order to evaluate the advantages of L^2C for HPC, we study a classic implementation of a finite difference based computation of partial differential equations. Its basic sequential algorithm is represented in Algorithm 1. As a matter of fact, the function f used in this algorithm usually displays a high degree of locality and the value can be computed by only accessing the spatial neighborhood of the cell. Additionally, since only the values from the previous iteration are used, one can greatly reduce

Algorithm 1 Sequential finite difference computation. The computation iterates over time (loop Line 1) and then over space (loop Line 2). The value of each cell is computed as a function of the domain at the previous iteration.

```
1: for i ← 1 to N do
2:    forall pos ∈ DOMAIN do
3:       domain_i[pos] ← f(domain_{i−1}, pos)
4:    end for
5: end for
```

memory usage by allocating only two matrices and reusing them.

In order to take advantage of parallel resources, this algorithm can be parallelized by adopting a domain decomposition approach. This is possible because there are no data dependencies between the various operations executed in a single iteration of the external loop (Line 1). In this specific case, the domain specific code is the function f, so it is rather easy.

For a shared memory target machine, this can be done using OpenMP by adding a *parallel for* annotation to the loop of Line 2. A finer grain of control can be attained to better fit to hardware by explicitly iterating over subdomains and letting this loop be parallel as illustrated by Algorithm 2.

Algorithm 2 Parallel finite difference computation. The parallel loop over subdomains (Line 2) let the workload be distributed over the computing cores.

```
1: for i ← 1 to N do
2:    forall subdomain ∈ sub(DOMAIN) do
3:       forall pos ∈ subdomain do
4:          domain_i[pos] ← f(domain_{i−1}, pos)
5:       end for
6:    end for
7: end for
```

Memory can be handled in two different ways. A first approach is to allocate the whole *domain* matrix in a single block. In this case, some synchronizations should be used to ensure that all data have been computed before they are accessed (this can be done with a barrier at each iteration of the external loop). A second approach consists in allocating the domain matrix in multiple blocks, one for each thread. In this case, ghost/overlap zones are welcome: data must be exchanged at the end of each external iteration to ensure consistency. These exchanges also enable to synchronize the various control flows.

The same approach can be used for a distributed memory machine using MPI but it will require to explicitly manage communications.

However, in practice, there is a problem to make scientific applications parallelized. Usually the sequential code is implemented by a specialist of the studied domain. Then, various parallel versions (OpenMP, MPI, ...) are derived from the sequential version by various parallel experts depending of the targeted hardware. However, the sequential code continues to evolve, either to fix bug or to add support for additional aspects of the simulation. Porting such modifications to the parallel versions is usually cumbersome because of the divergence of the codes.

4.1 A first L^2C version of the domain decomposition code

Our goal is to study how to define component assemblies of this application for various kinds of machines – from sequential to various form of parallel machines – while trying to maximize code reuse and separation of concerns. All assemblies and component presented in this section have been implemented and are evaluated in Section 5.

4.1.1 Base (sequential) component version

Six elements of the application architecture can be identified:

1. the main application that at some point uses the domain decomposition algorithm,
2. memory allocation for the matrices,
3. iterations over the time dimension,
4. possibly parallel iterations over the subdomains,
5. iterations over the space dimension,
6. computation of the value at a given position.

Among those, the second and fourth depend on the parallelization choice. A possible L^2C decomposition of such an application that isolates code linked to parallelism is presented in Figure 1. All connections are C++ interfaces.

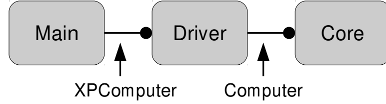

Figure 1: Application architecture based on three components. Only the `Driver` component is specific to a given parallelization strategy.

Algorithm 3 Algorithm of the (sequential) `Driver` component. *core* is the interaction point using a C++ `Computer` interface.

1: $mem \leftarrow allocate(DOMAIN)$
2: **for** $i \leftarrow 1$ **to** N **do**
3: $\quad core.compute(mem, DOMAIN)$
4: **end for**
5: $release(mem)$

The `Core` component represents the domain specific computing kernel, it comprises elements 5 and 6. The `Driver` component encapsulates the hardware specific part of the application, it comprises elements 2, 3 and 4. Finally, the `Main` component represents the rest of the application, in a real case it would very likely be made of a set of interconnected component instances.

The interface between the `Main` and the `Driver` component is called `XPComputer`. It lets the `Main` component specify the domain and the number of iterations required and receive in exchange the result of the computation. The interface between the `Driver` and the `Core` component is called `Computer`. Through it, the `Driver` component provides parts of already allocated memory area that correspond to the (sub-)domain being computed and the previous one as shown in Algorithm 3. The `core` component computes the values of the current (sub-)domain in function of the previous one.

4.1.2 Shared memory parallelization

The shared memory parallelization of this code has been implemented by a version of the `Driver` component called `ThreadDriver` that relies on the POSIX thread library. It is based on a parallel loop over the subdomains handled by a distinct thread each. Each thread of computation iterates over the time dimension and at each iteration it uses a specific instance of the `Core` component to compute the next iteration. At the end of each iteration, a barrier ensures that the whole domain has been computed. Algorithm 4 describes the pseudocode of the `ThreadDriver` component and the complete architecture of the application for four subdomains/threads is presented in Figure 2.

Algorithm 4 Algorithm of the shared memory parallel `ThreadDriver` component. The parallel loop on Line 2 makes use of a thread for each subdomain. The synchronization between threads is handled by the barrier on Line 6

1: $mem \leftarrow allocate(DOMAIN)$
2: **for** $thread \leftarrow 1$ **to** $Nthreads$ **do**
3: $\quad subdomain \in sub(DOMAIN, t)$
4: $\quad$ **for** $i \leftarrow 1$ **to** N **do**
5: $\quad\quad core_t.compute(mem, subdomain)$
6: $\quad\quad barrier()$
7: $\quad$ **end for**
8: **end for**
9: $release(mem)$

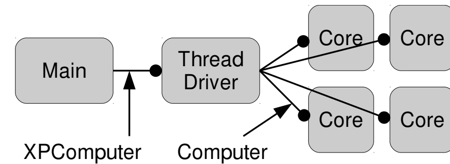

Figure 2: Application architecture with four threads of computation running in parallel for a shared memory machine. Each instance of the `Core` component runs in a distinct thread created by the `ThreadDriver` component.

4.1.3 Distributed memory parallelization

The distributed memory parallelization of this code is implemented by a version of the `Driver` component called `MpiDriver` that uses MPI for inter-process communication. An instance of this `MpiDriver` component run in each process together with a `Core` instance.

Algorithm 5 describes `MpiDriver` component in pseudo-code. The master `MpiDriver` component that gets called by the `Main` component broadcasts information it has received and lets each instance computes the subdomain it handles. Each

`MpiDriver` instance then iterates over the time dimension and at each iteration it make use of its bound `Core` component. At the end of each iteration, overlapping data are exchanged between the `MpiDriver` instances responsible of neighboring subdomains. The architecture of an application for four subdomains/processes is presented in Figure 3.

Algorithm 5 Algorithm of the distributed memory `MpiDriver` component. The memory allocation is done locally by each process and it includes the frontiers as shown on Line 3. The data synchronizations between processes are handled by the MPI exchange on Line 6 (typically, this would consists in $2D$ `isend/ireceive` groups where D is the number of dimension of the simulation.

1: $mpi.broadcast(DOMAIN)$
2: $subdomain \leftarrow sub(DOMAIN, MPI_RANK)$
3: $mem \leftarrow allocate(subdomain + frontiers)$
4: **for** $i \leftarrow 1$ **to** N **do**
5: $\quad core.compute(mem, subdomain)$
6: $\quad mpi.exchange(mem, frontiers)$
7: **end for**
8: $release(mem)$

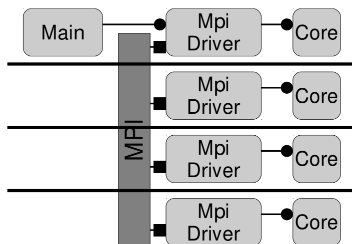

Figure 3: Application architecture with four domains/processes running in parallel in distinct memory space.

4.1.4 Discussion

This section has studied how components can be used to wrap the typical codes one would find for parallel domain decomposition of a finite difference code. By identifying the code that depends of the parallelization in a `Driver` component, this approach makes it possible to reuse the domain specific code of the `Main` and `Core` components in several parallel versions. However, while the two approaches (shared memory and distributed memory) have been implemented, the question arises in situations in which one would like to combine both approaches. In order to use shared memory inside a node and distributed memory between nodes in a multicore cluster, one would have to implement yet another variation of the driver.

As this driver would combine the two approaches used in `ThreadDriver` and `MpiDriver`, one would like to reuse these codes. However, since these components offer an interface distinct from the one they use, they cannot be combined. The next section analyses and proposes a more modular architecture that makes the combination of multiple parallelization approaches possible.

4.2 A modular L^{2}C version of the domain decomposition code

To increase code reuse among the various parallelization approaches, one has to decompose the aspects handled by the various `Driver` components at a finer grain. From what has been presented in the previous section, three aspects can be identified:

1. memory allocation;
2. iteration over the time dimension;
3. management of the decomposition, *i.e.* neighbor interactions.

Separating these three concerns in three distinct components enables the creation of assemblies combining them in various ways thus increasing code reuse compared to a more monolithic approach where each combination requires the development of a new `Driver` component.

In the previous approach, the (`Thread`)`Driver` component (or the set of fully interconnected `MpiDriver` instances) handled the interactions between all subdomains. In order to enable the choice of interaction implementations at a finer grain, including multiple distinct implementations in a single application, a new approach is proposed. It relies on components supporting interactions between a pair of domains each. These components offer an implementation agnostic C++ interface `Exchange` that comprises methods to:

1. notify the availability of the data of a given iteration;

2. specify where the data from the neighbor is expected;
3. wait for the data from the neighbor;
4. request authorization to reuse a memory space exposed via the first method.

With this approach, the iteration over the time dimension can be implemented independently of the parallelization. This is the role of the `JacobiCoreNiter` component. It exposes a service through a C++ interface `ComputerNiter` very similar to the `Computer` interface described in Section 4.1.1. Through this interface, a memory space on which to operate is specified but also a number of iterations to execute. At each iteration, `ComputerNiter` makes a call to the `Computer` interface to compute the data for the next iteration and makes exchanges with its neighbors via four `exchange` interfaces.

Finally, the memory allocation can be done after receiving a description of the domain via the `XPComputer` interface previously identified, and before calling the `ComputerNiter` to compute over this domain.

4.2.1 Shared memory parallelization

In order to support shared memory parallelism, the `ThreadXp` component is responsible for memory allocation. It also creates multiple threads that operate on the memory space it has allocated.

The interactions between each pair of neighboring components requires no memory copy, only synchronization. This is what a `ThreadConnector` component implements. It exposes four services `Exchange`, once for each of the four neighbor components in the 2D case. The "receive" operation shall just wait for the data to be available. It is implemented by waiting for the other neighbors to call the "send" operation that specifies that the data is available.

In order to implement the whole application based on these components, one instance of the `ThreadXp` component is created. It executes in parallel a set of instance of `JacobiCoreNiter` component organized in a grid together with the `Cores` on which they depend. Then neighboring `JacobiCoreNiter` components are connected through a `ThreadConnector`. This complete architecture for four subdomains/threads is presented in Figure 4.

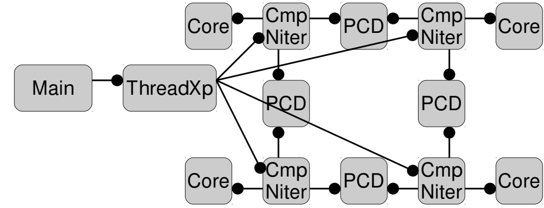

Figure 4: Application architecture with four threads running in parallel in a shared memory space. Only neighboring computational components are connected. PCD stands for Posix-`ThreadConnector`.

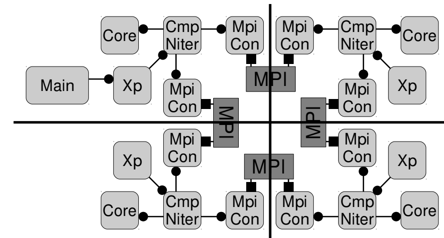

Figure 5: Application architecture with four processes running in parallel in distinct memory spaces. Only neighboring computational components are connected.

4.2.2 Distributed memory parallelization

In the distributed memory case, there are multiple components responsible for memory allocation that run in a distinct process respectively. The `ThreadXp` component with a number of threads to create set to 1 could be used. However, this would imply a dependency on the `pthread` library that might not be available. A version that does not support threads at all, simply called `Xp` has thus been created.

In this case however, interactions require memory copy across the process frontiers. This can not be achieved with a single component instance but requires two component instances, one in each process. The component `MpiConnector` relies on MPI to implement this behavior. It exposes a single instance of the `Exchange` service and relies on a MPI communicator to interact with the `MpiConnector` of the neighboring component. It maps the various operations of the `Exchange` interface on the corresponding asynchronous exchange methods of MPI.

The architecture of the whole application

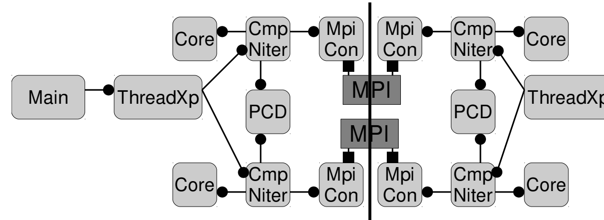

Figure 6: Application architecture with four processes running in parallel containing four threads each.

based on these component consists in a grid of processes containing one instance of the `Xp`, `JacobiCoreNiter` and `Core` components each. Neighboring `JacobiCoreNiter` components are connected by a pair of `MpiConnector` component instances. The architecture for four domains/processes is displayed in Figure 5.

4.2.3 Hierarchic parallelization

No additional component is needed to support a two level hierarchy infrastructure with MPI used between nodes of a cluster and shared memory used within nodes. One `ThreadXp` instance is used inside each process to allocate memory and create the threads.

Interactions between components depend on whether two neighboring component share the same memory space or not. If they do, a `ThreadConnector` component is used while if they do not, a pair of `MpiConnector` components is used. It results in the architecture described in Figure 6.

4.3 Single process multiple memory allocation

Another variation that can be easily implemented is a version with multiple threads in a single process but that does not share a single memory allocation. In this case, the interactions between neighbors require a data copy but this copy does not require the use of MPI. A `CopyConnector` component has been implemented that exposes the same `Exchange` interface and it is used similarly as `ThreadConnector` except that it makes a call to `memcopy` to copy the data. The addition of this simple new component makes it possible to implement many new variations and combinations, such as a three level hierarchy: multiple processes (interconnected with MPI) containing several distinct memory allocations but with several threads attached to each memory allocation. It can be relevant for clusters with large NUMA nodes.

4.4 Discussion

This section has shown that introducing components in an application can increase code reuse between various variations of hardware resources. A first step is to simply wrap code inside components keeping the legacy architecture. It enables the identification of the hardware induced variability in a component and its replacement independently of the domain specific code. A finer analysis and adaptation of the application architecture enables the identification of various aspects of variability at a finer grain, regarding memory allocation and interactions between threads of computation. This further increases reuse and adaptability to specific hardware resources by enabling multiple combinations of these choices at various levels.

These variations can be described by L^{2}C assemblies. However, they become more and more complex when the amount of points of variation increases as can be seen in the various figures. Each variation is described by a distinct assembly which means that a new point of variability may be introduced by a hardware specificity. That is why L^{2}C assemblies are not intended to be written by a programmer but should be generated by a compiler.

We can observe that there are a lot of similarity in these assemblies, both inside each assembly (e.g. the code running on each process) and between assemblies targeted at distinct hardware. Thus, with a careful design, code reuse and maintenance can be greatly improve. Let next section quantitatively evaluates this.

Table 1: Hardware Resource Architectures

Hardware Architecture	Application version
Single node, single core	Sequential version
Single node, multi-core	Multithreaded version
Multi-node, single core	MPI version
Multi-node, multi-core	hierarchical version (MPI + multithread)

Table 2: Number of lines for the various versions.

Jacobi Version	Number of lines (C++ code)		
	Non component	Driver	Connector
Sequential	161	239	388
Multithreaded	338	386	643
MPI	261	285	446

5 Experimental Evaluation

This section evaluates the approach proposed by this paper. It is based on the implementations of the Jacobi domain decomposition application mentioned in the previous section. Table 1 sums up the resource architectures used for the experiments: sequential, shared memory, distributed memory and hierarchical architecture. Experiments have been done on a multi-core based cluster (Griffon) of Grid'5000 made of 92 nodes, with 4 cores per CPU, 2 CPUs, 16 GB RAM. Nodes are interconnected with an Infiniband-20G network. MPI enabled components are used for the uni-core based cluster; multi thread based components are used for multi-core node. For the multi-core based cluster, a two level hierarchical model is provided with MPI used between nodes and shared memory used inside the multi-core nodes.

To evaluate the proposed approach, five criteria are studied: code reuse, speedup, performance, performance overhead, cyclomatic complexity.

The main advantage of our approach is the level of code reuse made possible: this is our first criteria of evaluation. Another important aspect in HPC is the efficiency of the code. This is evaluated by three criteria: the raw performances of the application, the speedup when parallelizing and the overhead due to the component approach in comparison to a legacy version of the code. Finally, the last criterion is the cyclomatic complexity of the codes.

5.1 Code Reuse

As shown in Section 4, designing application using component model increases code reuse. To quantify code reuse, we firstly count the number of lines of code in the non-component version of the Jacobi application. Table 2 presents these results.

Table 3 presents the number of lines for different component versions of Jacobi described in Section 4. “Driver” stands for the version based on the wrapping of legacy code in components. “Connector” stands for the advanced version. From the table, we can see the number of lines for each basic component. The table also mentions which components were reused from the sequential component assemblies. Let analyze it by assembly kind.

Driver versions As described in Section 4, the sequential version of the component based application is made of three components: `Main`, `SeqDriver` and `JacobiCore`. The total number of code lines is 239. The multithreaded version of the component based application is also made of three components: `Main`, `ThreadDriver` and `JacobiCore` for a total of 405 lines of code. The MPI version is still made of three components: `Main`, `MpiDriver` and `JacobiCore` which accounts for 285 lines of codes. The total numbers of lines are of the same order of magnitude than the non component version but components enable reuse: The `Main` and `JacobiCore` components are shared between the three versions. This results in a rate of reuse of ***26%*** between the sequential and multi-threaded versions and ***32%*** between the sequential and MPI

Table 3: Detailed SLOC for all components.

Assembly Version	Component Name	SLOC	Reused from seq. version
Driver & Connector	JacobiCore	25	yes
Driver & Connector	DataInitializer	68	yes
Driver & Connector	Main	105	yes
Driver	SeqDriver	109	
Driver	MpiDriver	155	
Driver	ThreadDriver	256	
Connector	XP	71	yes
Connector	JacobiCoreNiter	187	yes
Connector	ThreadXP	186	
Connector	ThreadConnector	140	
Connector	MpiConnector	58	

versions.

Connector versions For the second approach, the sequential version requires the `XP` and `JacobiCoreNiter` components in addition to `Main` and `JacobiCore`. The multithreaded version is based on the `ThreadXP` and `ThreadConnector` components, while the MPI version is based on the `XP` and `MpiConnector` components. The hierarchical version uses the `XP`, `MpiConnector` and `ThreadConnector` components. Code reuse between the sequential and multithreaded versions is ***31%*** and it is ***87%*** between the sequential and MPI versions. The hierarchical version does not require any new code, just a new component composition. The rate of reuse is thus ***100%*** with respect to the multithread and MPI versions.

Components increases reuse of code. Indeed, the basic component (Jacobi algorithm in `JacobiCore` component) is used in all of the component based applications. If changes are needed to the Jacobi algorithm, only this component has to be modified. Separation of concerns into components can help to quickly and simply create applications.

5.2 Speedup

An important aspect of HPC is the ability to efficiently use hardware resources. This means that the application performances should scale gracefully with the number of computing core available. Figure 7 displays the speedup and the efficiency obtained for various version of the Jacobi application on various deployment scenarios. The horizontal axis represents different parallelisms with several size of the array of Jacobi application. The threads and processes scenarios are obtained using one node. For multi-node scenarios, a core per node is used unless specified: 8x2 means using 8 nodes and 2 cores per node. The left vertical axis is the speedup while the right vertical axis is the parallel efficiency.

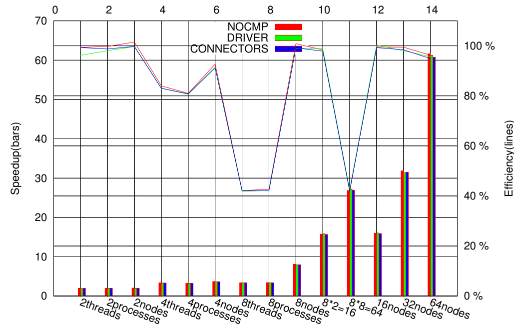

Figure 7: Speedup of the Jacobi HPC application with a feasible scalability.

The main objective of this experiment is to show that similar performance than native applications can be obtain with the L^2C component based versions and that all versions can achieve very good efficiency but for some situations: when using 8 cores per node — either 8 threads or 8 processes — the efficiency drops because of a problem of memory bandwidth. Too many cores access the main memory. The next section goes further in the de-

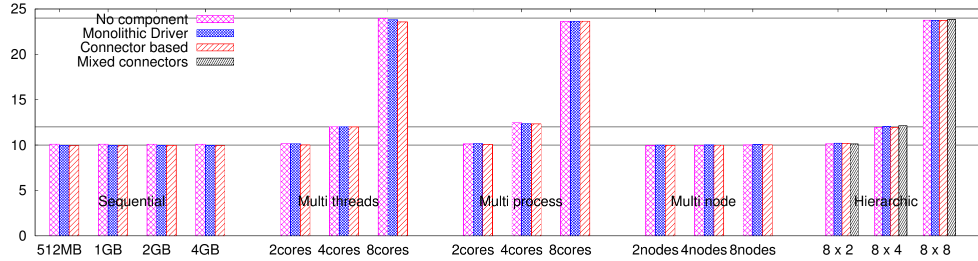

Figure 8: Duration in nanoseconds for the computation of one cell normalized with respect to the number of core used.

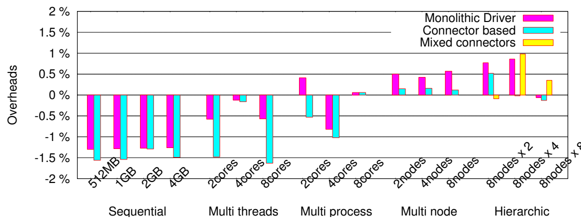

Figure 9: Overhead by percentage of the component model based application against native application

tails of these experiments.

5.3 Performance overhead

Figure 8 reports results obtained without components, with the monolithic driver and with the connector based versions on the configuration than the previous experiments. However, it shows the duration (in nano seconds) for the computation of one cell of the array per core. The results are in accordance with the previous speedup figure. One can see that memory contention appears when more than two threads access the same memory. It is acceptable for 4 cores but not for 8 cores.

Figure 9 display the same experiment results but normalized with respect to the non component version. It shows that the overhead of the component version is always below 1%. In some cases, the component based application is even better than the native application. However, as it is in the range of 1.5 %, it does some relevant.

From these experiments, we can conclude that the Jacobi application can be turns into a component one without impacting performance. If choosing carefully the number of active core per node, the time to compute a matrix cell can be the same as the sequential version (considering a large enough data size).

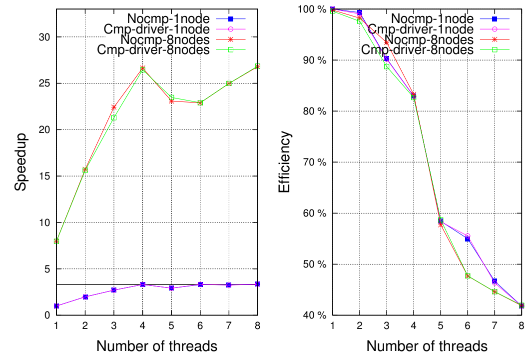

Figure 10: Jacobi speedup and efficiency for 1 to 8 threads per node.

5.4 Multi-core Performance

To better understand how to tune the application to maximize performance, some experiments were conducted with different threads with a one node and a multi-node scenarios: the expected increase of the duration to compute a matrix cell when increasing the number of cores inside one node may be compensated by the increase of parallelism.

Figure 10 presents the result for a cluster with 8 cores per node. 8 threads usually get the best speedup, while 4 threads is a better choice considering it is about the speedup and thus a much higher efficiency.

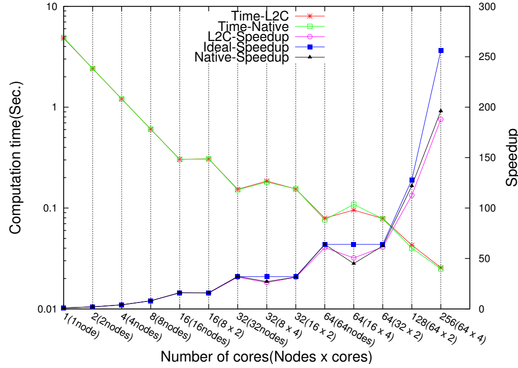

Figure 11: Performance of the benchmark Jacobi HPC application for selected non component and component applications (strong scalability).

5.5 Strong Scalability

Figure 11 presents the results obtained for native implementations (threads or MPI) and for some selected component assemblies (Connector version). The array size is fixed to 22016 × 22016. Under this strong scaling, the best configuration is usually achieved with four threads per node. Component versions performs similarly than native versions. For the 256 core experiments, each core handle less than 16 MB of data.

5.6 Cyclomatic Complexity

The cyclomatic complexity is a measurement of the complexity of a code. It measures the number of linearly independent paths through the source code. We computed the cyclomatic complexity using `pmccabe`, a standard package to calculate cyclomatic complexity.

Table 4 shows the total cyclomatic complexity of non component and component codes. Components reduces the cyclomatic complexity because of their promotion of separation of concerns. Only the driver version of component model increases the cyclomatic complexity compared with native code. It is due to the integration of many functions into one big component.

5.7 Discussion

The proposed L^{2}C model enable to increase code reuse, to reduce code complexity and to achieve the same performance than non component versions. However, the component model introduces a new task to be done: to turn an existing code into a component based code, in addition of creating the components, ones needs to describe an assembly for each specific hardware architecture. As it is very fastidious and error prone, such assembly description should be automatically generated. This is one of the purposes of HLCM [2].

Table 4: Cyclomatic complexity of the component model based codes and native codes

Version	No Compo.	Driver	Connector
Sequential	28	32	8
Threaded	76	41	26
MPI	55	22	13

6 Application to A Domain Specific Application

NEMO [19] is an ocean modeling framework which is composed of "engines" nested in an "environment". The "engines" provide numerical solutions of ocean, sea-ice, tracers and biochemistry equations and their related physics. The "environment" consists of the pre- and post-processing tools, the interface to the other components of the Earth System, the user interface, the computer dependent functions and the documentation of the system. It is written in FORTRAN 90 and parallelized using MPI with a regular domain decomposition in latitude/longitude. The governing equations are solved in finite-difference form upon a tri-polar 'ORCA' grid to get rid of the north pole singularity.

With respect to our goal, this application is very similar to the considered Jacobi application. We are currently applying the experience gained with it to modify NEMO so as to study auto-tuning on specific hardware platforms, especially on petascale super-computers.

7 Conclusion

For the development and adaptability of high performance computing applications on various hardware resources, we proposed and evaluated a low level component model (L^{2}C) enabling resource

specialization. L^2C supports multiple kinds of interactions between components with runtime overhead reduced to their minimum. Its evaluations have been conducted on some typical hardware architectures with respect to a well-known benchmark Jacobi application. The experimental results demonstrate that L^2C succeeds in implementing the separation of concern between domain specific codes and hardware specific codes. It is adaptable to different specific hardware resources. Moreover it brings the benefit of code reuse, without degrading performances.

While L^2C makes it possible to describe the architecture of a parallel application based on local method calls, MPI and CORBA, it does not support adaptation to hardware by itself. It has to be used in conjunction of a more abstract model, such as HLCM whose role is to generate these hardware specific concrete assemblies. Future work include developing auto-tuning algorithms for HLCM to be able to generate well suited L^2C assemblies for Jacobi but also for NEMO for different kinds of hardware platforms.

Acknowledgment

The authors would like to thank PRACE association (see http://www.prace-ri.eu) for the support to the work. And experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] Szyperski, C. *Component Software: Beyond Object-Oriented Programming.* Boston, MA, USA:Addison-Wesley Longman Publishing Co., 2002.

[2] J. Bigot, C.Pérez. *On High Performance Composition Operators in Component Models.* High Performance Scientific Computing with special emphasis on Current Capabilities and Future Perspectives,2011.

[3] P. Ribeiro, Christiane, et al. *Improving Memory Affinity of Geophysics Applications on NUMA Platforms Using Minas.* High Performance Computing for Computational Science – VECPAR 2010, Springer Berlin / Heidelberg, Germany:279-292, 2011.

[4] W. Gropp, E. Lusk,A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press,Cambridge, MA, USA, 1999.

[5] E. Jeannot and G. Mercier. *Near-optimal placement of MPI processes on hierarchical NUMA architectures.* Euro-Par'10, Ischia, Italy:199-210,2010.

[6] N. T. Karonis, B. Toonen and I. Foster. *MPICH-G2: A Grid-enabled implementation of the Message Passing Interface.* Journal of Parallel and Distributed Computing:551-563, 2003.

[7] Butenhof, R. David. *Programming with POSIX threads.* Addison-Wesley Longman Publishing Co.,Boston, MA, USA, 1997.

[8] C. Barbara, J. Gabriele, P. Ruud van der. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation).* MIT Press,Cambridge, MA, USA, 2007.

[9] L. Seyong, M. Seung-Jai, E. Rudolf. *OpenMP to GPGPU: a compiler framework for automatic translation and optimization.* Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming(PPoPP '09), Raleigh, NC, USA:101-110, 2009.

[10] L. V. Kale, E. Bohm, et al. *Programming Petascale Applications with Charm++ and AMPI.* Petascale Computing: Algorithms and Applications, Chapman & Hall / CRC Press:421-441, 2008.

[11] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. *An evaluation of global address space languages: co-array fortran and unified parallel C.* In Proceedings of the tenth ACM SIGPLAN symposium on

Principles and practice of parallel programming (PPoPP '05). ACM, New York, NY, USA, 36-47, 2005. DOI=10.1145/1065944.1065950 http://doi.acm.org/10.1145/1065944.1065950

[12] The OSGI Alliance. *OSGi Service Platform Specifications*. Available at `http://www.osgi.or`

[13] Mozilla. *The XPCom projet*. Available at `http://www.mozilla.org/projets/xpcom`

[14] Apache OpenOffice. *UNO Component Model*. Available at `http://www.openoffice.org/udk/common/man/componentmodel.html`

[15] Object Management Group,*Common Object Request Broker Architecture Specification, Version 3.1, Part 3: CORBA Component Model*. 2008.

[16] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. *GCM: A Grid Extension to F RACTAL for Autonomous Distributed Components*. Special Issue of Annals of Telecommunications: Software Components – The Fractal Initiative, 64(1):5, 2009.

[17] B. A. Allan, R. Armstrong, et al. *A Component Architecture for High-Performance Scientific Computing*. International. Journal of High Performance Computing Applications,Thousand Oaks, CA, USA:163-202, 2006.

[18] J. Bigot and C. Pérez, *Enabling Connectors in Hierarchical Component Models*. INRIA, RR-7204, 2010.

[19] G. Madec.*NEMO ocean engine, Note du Pole de modélisation*. Institut Pierre-Simon Laplace (IPSL), France, 2008.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l’Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399