Computing (2014) 96:1115–1130
DOI 10.1007/s00607-013-0368-3

A low level component model easing performance portability of HPC applications

Julien Bigot · Zhengxiong Hou · Christian Pérez · Vincent Pichon

Received: 28 March 2013 / Accepted: 31 October 2013 / Published online: 23 November 2013

Abstract Scientific applications are getting increasingly complex, e.g., to improve their accuracy by taking into account more phenomena. Meanwhile, computing infrastructures are continuing their fast evolution. Thus, software engineering is becoming a major issue to offer ease of development, portability and maintainability while achieving high performance. Component based software engineering offers a promising approach that enables the manipulation of the software architecture of applications. However, existing models do not provide an adequate support for performance portability of HPC applications. This paper proposes a low level component model (L^{2}C) that supports inter-component interactions for typical scenarios of high performance computing, such as process-local shared memory and function invocation (C++ and FORTRAN), MPI, and CORBA. To study the benefits of using L^{2}C, this paper walks through an example of stencil computation, i.e. a structured mesh Jacobi implementation of the 2D heat equation parallelized through domain decomposition. The experimental results obtained on the GRID'5000 testbed and on the Curie supercomputer show that L^{2}C can achieve performance similar to that of native implementations, while easing performance portability.

Keywords HPC · Component models · Performance portability · Curie supercomputer · Grid'5000

Mathematics Subject Classification 68N99 · 68W10 · 68M99

J. Bigot
CEA, Maison de la Simulation, USR 3441, 91191 Gif-sur-Yvette Cedex, France
e-mail: julien.bigot@cea.fr

Z. Hou · C. Pérez (✉) · V. Pichon
Inria, LIP, ENS Lyon, 46 allée d'Italie, 69364 Lyon Cedex, France
e-mail: christian.perez@inria.fr

1 Introduction

Scientific applications require more and more computing power to simulate an increasing number of phenomena with a better accuracy. Computing power is provided by parallel hardware resources such as super-computers, clusters of multi-CPU, multi-core nodes with GPUs, etc. On one hand, these various parallel hardware architectures offer very distinct features when it comes to computing unit types, memory models, communication support, etc. Codes have thus to be specifically optimized for a target hardware to obtain high performance. On the other hand, code implementation and validation by simulated domain specialists is an expensive process; 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. It leads to duplication of efforts and thus adaptation of a code to a new kind of hardware is very expensive. This could be avoided by using a suitable approach that supports adaptation to various hardware while allowing reuse of the parts of the code, in particular those that are not hardware specific. We refer to this ability to reuse code while remaining close to the theoretical peak performance of each computer as *performance portability*. Various approaches exist to support that, such as conditional compilation with pre-processor directives, architecture specific optimizations applied by compilers, static and dynamic polymorphism provided by some programming languages or compile-time, launch-time and run-time choice of library implementations.

The imperative languages on which these approaches are based have drawbacks when dealing with large codes, in particular with respect to code re-use, maintenance, and code evolution. This has led the conception of a new approach for software development: component-based software engineering [1] that aims at avoiding these drawbacks. However, their use in high performance computing (HPC) is not as widespread as one could expect. One reason is that some models introduce overheads at runtime that are not acceptable in HPC while other models do not offer an adequate level of abstraction that would allow to efficiently adapt applications to hardware resources.

This paper proposes low level component (L^2C), a model close to hardware abstractions that supports the manipulation of the whole application structure through its assembly. The current version supports interactions such as data sharing, function and procedure calls (C++ and FORTRAN), message passing (MPI) and remote method invocation (CORBA). L^2C aims at supporting assemblies for typical HPC scenarios, which are well suited for various kinds of resource infrastructures. This is studied through a well known case study that demonstrates how L^2C enables to easily adapt the structure of the applications to various situations.

The remaining of the paper is organized as follows: Sect. 2 details the problem and deals with related work. Sect. 3 describes the L^2C model. Section 4 walks through an example of stencil computation, and analyzes how L^2C can be used to efficiently implement its multiple variations targeted at various hardware resources. Section 5 experimentally evaluates several metrics such as code reuse, performance, overhead and software complexity on the Grid'5000 experimental testbed as well as on the Curie supercomputer. Section 6 concludes the paper.

2 Related work

To support hardware variations, a programming model should ease the application performance portability with a maximal reuse between different versions, while enabling performance and ease of development. Hence, the model should support separation of concerns between the development of simulated domain specific codes and optimizations for the various hardware architectures. Various approaches have been proposed such as specific compilation, multiple implementation support, and software components.

2.1 Target specific compilation

A first approach is to handle the required software variations through the compiler. It enables to efficiently choose the best assembly instructions for a given processor and can even support a certain amount of parallelism with vector instructions. However, typical imperative languages are inherently sequential. As automatic parallelization of these code is very difficult if not impossible, parallel language extensions are needed. For example, one can explicitly mark loops that have to be parallelized with annotations such as OpenMP [2] or OpenACC [3], or with an API such as OpenCL and Cuda. There are also approaches that rely on more fundamental language modifications such as Charm++ [4] or partitioned global address space (PGAS) languages such as UPC or co-array Fortran [5]. They embeds parallel constructs that aim to abstract actual machines.

All these approaches offer portability of code to multiple architectures. However the structure of the application has to remain the same and as such they can not offer complete performance portability. For example, they can not handle easily algorithmic variations in function of the target architecture.

2.2 Multiple implementation support

To efficiently run on distinct hardware, it is thus usually required to provide variations of the algorithms specifically targeting each architecture. The application developer can explicitly implement these choices in the code, choices handled either statically with conditional compilation or at execution with plain conditional constructs. By waving the aspect of implementation choice in the domain specific code, these solutions lead to very obfuscated code, difficult to maintain and to make evolve.

Thus, a solution is to rely on a compiler to handle these choices [6]. Recent advances relying on a run-time scheduler even make it possible to choose between execution on CPU, GPU or other accelerators for each invocation of a function [3,7]. Some works, such as Peppher [8], attach performance models to performance critical parts of an application by supporting multiple variants depending on core type, usage context and performance criteria. Variants can then be selected at runtime by a performance-aware scheduler.

All these approaches are however limited to a choice between multiple function implementations. They do not enable more invasive changes that would require change at the scale of the whole application structure.

2.3 Software component models

Component based software engineering (CBSE) [1] extends the concept of class by specifying in its interface not only the services it offers (public methods), but all its possible interactions with the outer world, including the services it requires. Components instances can thus be configured externally. This inversion of control [9] eases the configuration of the variations to use for every algorithm of the application.

Distributed component models such as the CORBA component model (CCM) [10], or the grid component model (GCM) [11] offer process and even network transparency, but at the price of some runtime overhead that is not acceptable for HPC. Common component architecture (CCA) [12] is a HPC dedicated component model. BABEL offers CCA inter-language support and also network transparency. As it does not use actual connections between components, it introduces a small overhead. For parallelism oriented interactions, CCA components are expected to rely on external models such as MPI: such interactions do not appear in the interface of components.

The architecture of a component-based application is usually fully described by its assembly, i.e., the set of component instances and their interactions, and possibly their placement on resources. Assemblies can be optimized for each and every specific hardware configuration, making it possible to isolate the hardware adaptation concern. However, rewriting a component assembly can be a tedious process to do manually for each new hardware variation. Therefore, a more abstract approach supported for example by the high level component model (HLCM) [13,14] is to rely on a compilation of a hardware agnostic abstract assembly into a hardware specific concrete assembly.

2.4 Analysis

Compiler-level optimizations offer an efficient solution for processor specific optimizations at the scale of a few instructions. Nevertheless, optimization of HPC applications do also require optimizations that work on the whole structure of the application. Many solutions handle selection between multiple function variations. However most of those lead to the interleaving of code handling the optimizations with domain specific code.

By combining the advantages of both the compilation and library approach, launch-time compilation of an application assembly offers an interesting compromise. Such an approach requires the existence of an execution model that supports (1) interactions between components with minimal overhead; and (2) inter-process and over the network interactions. The following section presents L^2C, a component model designed with such goals.

3 Low level component model

3.1 Overview

With modular compilation, applications are typically made of a set of object files, each containing some code and data as well as an external interface defined with exported and required symbols. The linking of these object files into a full application mostly relies on implicit rules. It is very effective to enable various kinds of optimization at link time. However it has two main limitations when it comes to parallel applications. First, it is local to each process. Second, it does not have an explicit concept of instance that would make it possible to use two distinct libraries for two uses of an API in a process.

L^2C aims at overcoming these limitations. It does not target to offer final users a simple model but rather to define an execution model for compilers: i.e., an assembly for parallel applications structure. L^2C is an execution model that can be seen either as an extension of modular compilation or as a low level component model that does not hide system issues. The core of L^2C is thus the support of the description of interactions between component instances inside each process but also between distinct processes. This is achieved by enabling L^2C components to directly make uses of "native" interactions such as memory sharing, C++/FORTRAN procedure invocations, message passing with MPI and remote procedure calls with CORBA.

3.2 L^2C components

At the lowest level, L^2C components build upon object files, with some required specific entry points (usually functions). Two entry points are used to create and destroy component instances. The other two entry points are used to get and set the values of component instance symbols, a.k.a. properties. When creating a component instance, a structure containing its metadata is created, all other operations receive it as an argument. The creation entry point has a specific name for each component type, the others are accessed through function pointers via the metadata structure. As it is quite low level, L^2C provides mapping for languages such as C++, FORTRAN, and CHARM++ [4]. These mappings are based on pre-processor directives and a very small runtime.

The C++ mapping defines L^2C components as plain C++ classes with a few additional annotations to generate the metadata. The annotations identify classes used as components and they associate a type and name to their properties. The FORTRAN mapping defines L^2C components as FORTRAN 2003 data types to store the state of a component.

3.3 L^2C assembly

A configuration file (a L^2C assembly descriptor or LAD) specifies how many instances of each component to create on each process and how to configure and to connect them. There are two kinds of assemblies: one for C++/FORTRAN assemblies, and one for CHARM++ assemblies. The C++/FORTRAN assembly represents sets of (MPI) processes containing component instances. Each process has a particular entry point, i.e., a function property of one component instance used to start the application within this process. It can be seen as a configurable `main` function. Each component instance has its properties configured either by directly specifying values in the assembly descriptor or by connecting it to a property exported by another component instance.

L^{2}C also provides a simple and straightforward API for instantiating, destroying, configuring, and connecting components.

4 Study of the usability of L^{2}C on a Jacobi application

To evaluate L^{2}C, a classic implementation of a finite difference based computation is studied. The implementation has been done in C++.

Memory can be handled in two different ways. A first approach is to allocate the whole *domain* matrix in a single block. Synchronizations primitives are required to ensure that all data has been computed before it is accessed (this can be done with a barrier at each iteration). A second approach consists in allocating the *domain* matrix in multiple blocks, one for each thread. Ghost zones are used so that each thread only work on its own block. These ghosts are then exchanged at each iteration.

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

4.1 Driver-based L^{2}C assemblies

The first approach adopts a modularization approach similar to what could be implemented with plain libraries, without taking advantage of the specific features of L^{2}C.

Base (sequential) component version Six elements of the application architecture can be identified as component: (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) parallel iterations over the subdomains, (5) iterations over the space dimension, and (6) computation of the value at a given position. Among those, the second and fourth depend on the parallelization choice.

A possible L^{2}C assembly of such an application that isolates code linked to parallelism is presented in Fig. 1. All connections are C++ interfaces. The `Core` component represents the domain specific computing kernel made of 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.

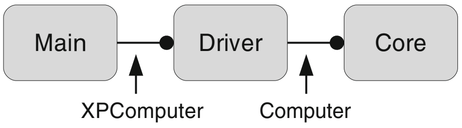

Fig. 1 Architecture of the application based on three components. Only the `Driver` component is specific to a given parallelization strategy

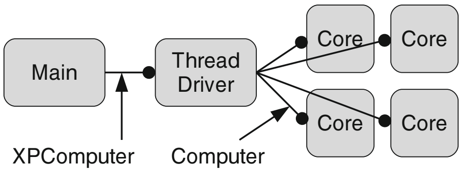

Fig. 2 Architecture of the application with four threads of computation running in parallel in a shared memory machine. Each instance of the `Core` component runs in a distinct thread created by the `ThreadDriver` component

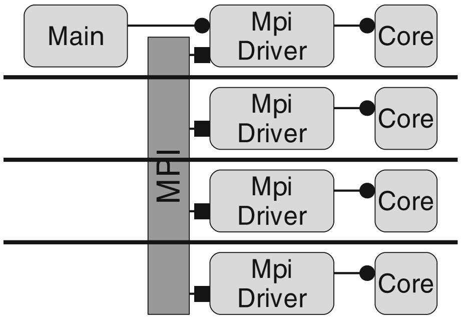

Fig. 3 Architecture of the application with four domains running in distinct processes

The interface between the `Main` and the `Driver` component lets the `Main` component specify the domain and the number of iterations required. It receives in return the result of the computation. The interface between the `Driver` and the `Core` component lets the `Driver` component specify the already allocated memory area on which to operate.

Shared memory parallelization The `ThreadDriver` component is a variant of the `Driver` component that relies on the POSIX thread library. It contains a parallel loop over the subdomains handled by a distinct thread each. Each thread uses a specific instance of the `Core` component to compute the next iteration. Figure 2 displays the application structure for four subdomains.

Distributed memory parallelization The `MpiDriver` component is a variant of the `Driver` component that uses MPI for inter-process communications. An instance of the `MpiDriver` component runs in each process together with a `Core` instance. The master `MpiDriver` component that gets called by the `Main` component broadcasts information it has received and lets each instance compute the subdomain it handles. Each `MpiDriver` instance makes use of its bound `Core` component. At the end of each iteration, data are exchanged between the `MpiDriver` instances responsible of neighboring subdomains. Figure 3 gives an example of the application architecture for four subdomains.

The reusability of the “driver” approach does not appear very good. For example, for a multi-core cluster, one would have to implement yet another variation of the driver. The next section proposes a more modular architecture that makes the combination of multiple parallelization approaches possible.

4.2 Connector-based L^2C assemblies

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. A solution is to separate the three following concerns into three components: (1) memory allocation, (2) iteration over the time dimension, and (3) management of the decomposition, i.e. neighbor interactions. The extraction of a component for neighbor interactions is possible thanks to the notion of instance in L^2C. As a matter of fact, there might be multiple neighbors in a single process, not all of those requiring the same kind of interaction.

The considered approach relies on components supporting asynchronous interactions between a pair of domains each, through the (implementation agnostic) interface `Exchange`. The iteration over the time dimension is implemented independently of the parallelization by a component of type `CmpNiter`. Such a component exposes a service that enables to specify the memory space on which to operate and the number of iterations to execute. At each iteration, it makes a call to a `Core` component and makes neighbor exchanges through four `Exchange` interfaces.

Shared memory parallelization 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 `PCD` component implements. It exposes two `Exchange` services, once for each of the two neighbor components.

A `ThreadXp` component instance executes in parallel a set of instances of `CmpNiter` component organized in a 2D grid together with the `Core` Components. Neighboring `CmpNiter` components are connected through a `PCD` component. Figure 4 illustrates this architecture for four subdomains.

Distributed memory parallelization Multiple components are responsible for memory allocation that run in a distinct process respectively. This could be achieved by the `ThreadXp` component, but in order not to depend on the thread library, the more simple component `Xp` is reused from the pure sequential version (not presented here).

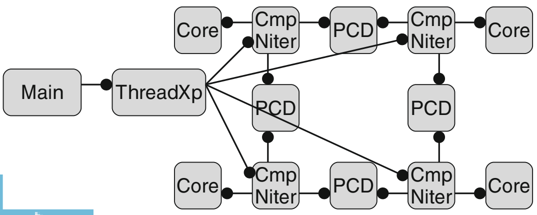

Fig. 4 Architecture of the application with four threads running in parallel in a shared memory space. PCD stands for PosixThread Connector Dual

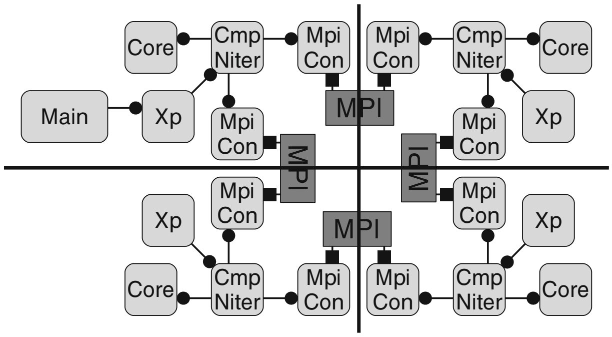

Fig. 5 Architecture of the application with four processes running in parallel in distinct memory spaces. Only neighboring computational components are connected

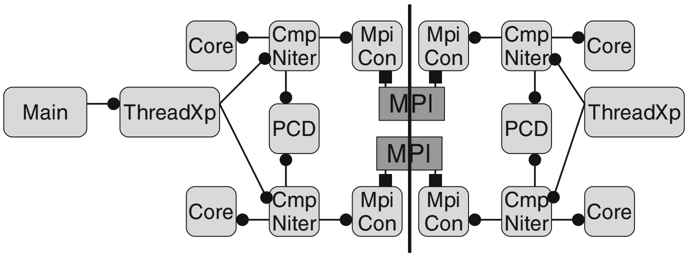

Fig. 6 Architecture of the application with hierarchical parallelization

Interactions between distinct processes involve memory copy for the ghost zones. It requires two component instances, one in each process. The component `MpiCon` 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 `MpiCon` of the neighboring component.

The application consists of a grid of processes containing one instance of `Xp`, of `CmpNiter`, and of `Core`. Neighboring `CmpNiter` components are connected by a pair of `MpiCon` component. Figure 5 displays the application architecture for four subdomains.

Hierarchical 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. Figure 6 displays the application architecture for four subdomains.

5 Experimental evaluation

Experiments have been done on two typical platforms, a cluster and a supercomputer. The cluster is the Griffon multi-core cluster of the GRID'5000 experimental platform. It is made of 92 nodes, 2 CPUs per node, 4 cores per CPU, and 16 GB RAM per node.

Table 1 Total number of lines for the various versions of the Jacobi application

	Number of lines (C++ code)		
	Non component	Driver version	Connector version
Sequential	161	239	388
Multithreaded	338	405	643
MPI	261	285	446

"Driver" is the version described in Sect. 4.1. "Connector" is in the version described in Sect. 4.2

Nodes are interconnected with an Infiniband-20 GB network. The supercomputer is the Curie supercomputer from PRACE, restricted to thin nodes. It contains 5,040 thin blades with 2 sockets per node, each socket has 8 cores that share a 20 MB L3 cache. Each node has 64 GB RAM. The interconnection network is an Infiniband QDR full fat tree.

MPI enabled components are used for the single-core multi-node experiments. Multithread based components are used for multi-core experiments. For the multi-core multi-node experiments, a two level hierarchical model is provided with MPI used between nodes and threads inside multi-core nodes.

Four criteria are studied: code reuse, speedup, performance overhead, and cyclomatic complexity [15].

5.1 Code reuse

Table 1 displays the number of lines of the non-component version and of the two component versions of the Jacobi application ("Driver" and "Connector" versions). The number of lines for the component versions are slightly larger, mainly because of the lines needed to declare the component type. It appears high because the total number of line is pretty low.

Table 2 breaks down the number of lines for all components, their usage in the "Driver" and "Connector" version, as well as whether they are reused from the sequential component assemblies.

For the "Driver" version, the total numbers of lines are of the same order of magnitude than the non component version but components increase reuse: the `Main` and `JacobiCore` components are shared between the three versions. This results in a reuse of 26 % of lines of code between the sequential and multi-threaded versions and of 32 % between the sequential and MPI versions.

For the "connector" version, code reuse is 31 % between the sequential and multi-threaded versions, and 87 % between the sequential and MPI versions. The hierarchical version does not require any new code, it just needs a new assembly, leading to 100 % code reuse with respect to the multithreaded and MPI versions.

5.2 Speedup

The Jacobi application is known to have an inherently good parallelism. Figure 7 displays the speedup and the efficiency obtained for various versions of the Jacobi appli-

Table 2 Detailed number of lines (SLOC) for all components and re-use

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	

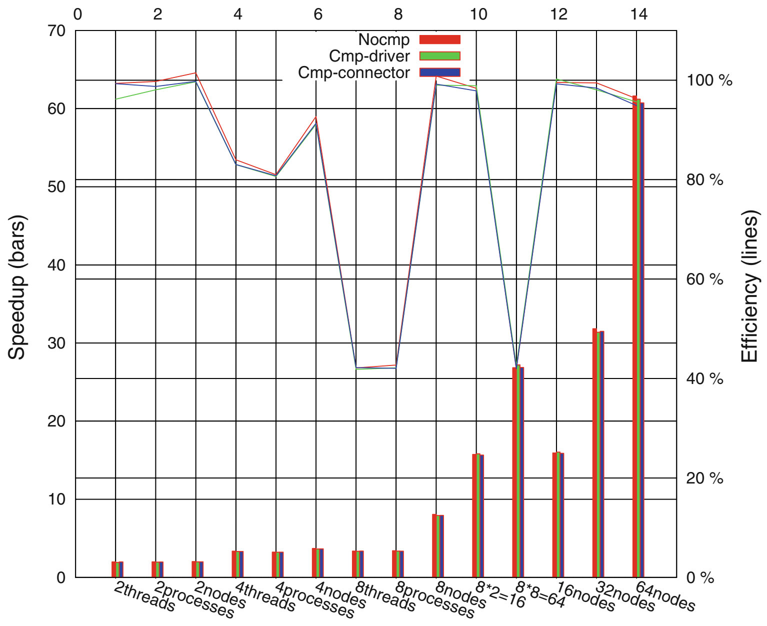

Fig. 7 Speedup of the Jacobi application. The *horizontal axis* represents distinct scenarios: “threads” and “processes” scenarios are obtained using one node. For multi-node scenarios (“nodes”), a core per node is used by default. 8 × 2 means 8 nodes and 2 cores per node

cation on various deployment scenarios. These experiments show that performance similar to the native applications (NOCMP) can be obtained with the L^{2}C component based versions and that all versions can achieve very good efficiency but for some situations: when using 8 cores per node (8 threads or 8 processes), the efficiency drops because of a problem of memory bandwidth as too many cores access the main memory.

5.3 Performance overhead

Figure 8 reports the duration of the computation for one cell of the domain per core in four settings: native (without components), driver based version, connector based version in two variations: one connector type (thread or MPI) and with two connector types (thread and MPI). First, the results are in accordance with the speedup figure. Memory contention appears when more than two threads are used within one node. Second, all settings seems to perform equally: the overhead of the component version is always below 1 %. In some cases, component versions are even slightly better than the native application. However, it is in the range of about 1.5 %.

Figure 9 presents the results for 1 to 8 threads per node, and for 1 node and 8 node configuration. 8 threads usually get the best speedup. 4 threads is a better choice considering a similar speedup but with a much higher efficiency.

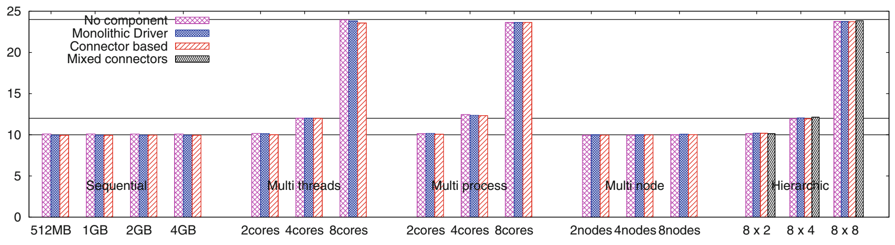

Fig. 8 Duration in nanoseconds for the computation of one cell normalized with respect to the number of cores used

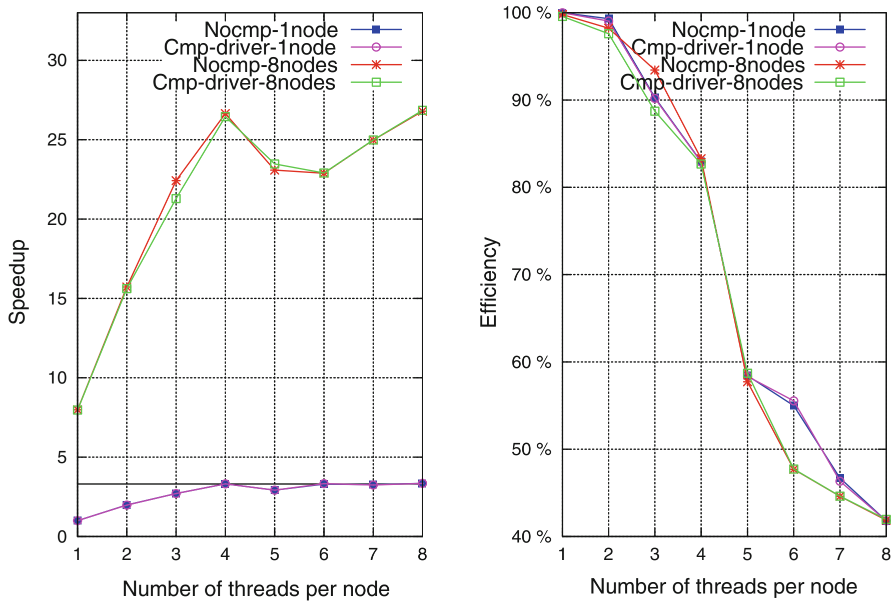

Fig. 9 Jacobi speedup and efficiency up to 8 threads per node and up to 8 nodes

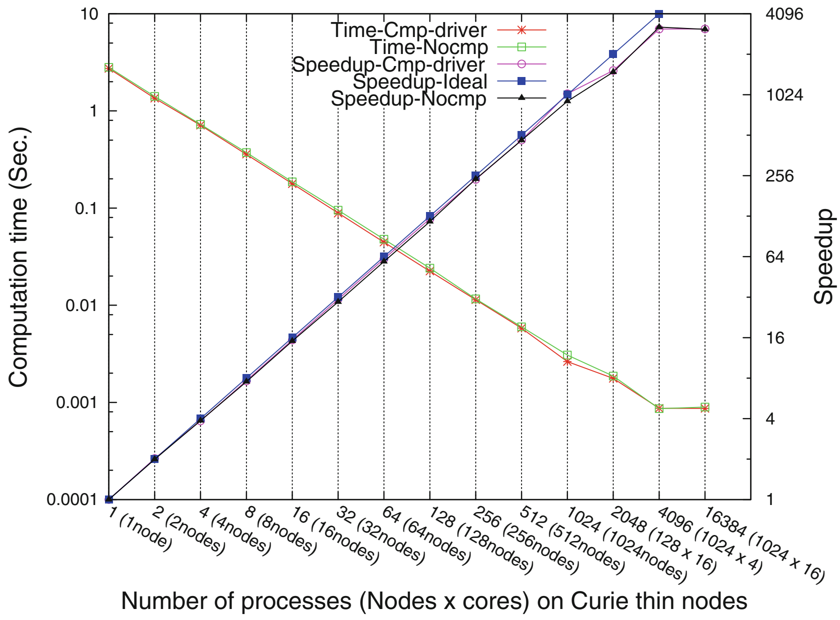

Fig. 10 Jacobi strong scalability with a fixed size (16, 384 × 16, 384) on Curie thin nodes

5.4 Performance results on curie thin nodes

Experiments done with thin nodes of the Curie supercomputer confirm the results obtained with the Griffon cluster. As there is little difference between the driver version and connector version, this section just reports experiments with the native and driver versions.

Figures 10 and 11 report respectively the results for strong and weak scalability on Curie thin nodes. For both versions (MPI native and L^2C version) of the Jacobi application, the strong and weak scalability results are good due to the intrinsic good parallelism of the Jacobi benchmark.

In most cases, the performance of the component version is slightly better than that of the native MPI version by 0.2–7 %. This is mainly due to the pre-initialization of the `DataInitializer` component. Otherwise, the maximal measured overhead of the component model version is 2.7 %.

5.5 Cyclomatic complexity

The cyclomatic complexity is a measurement of the complexity of a code [15]. It measures the number of linearly independent paths through the source code. Table 3 shows the cyclomatic complexity of non component and component versions. 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 to native code. It is due to the integration of many functions into

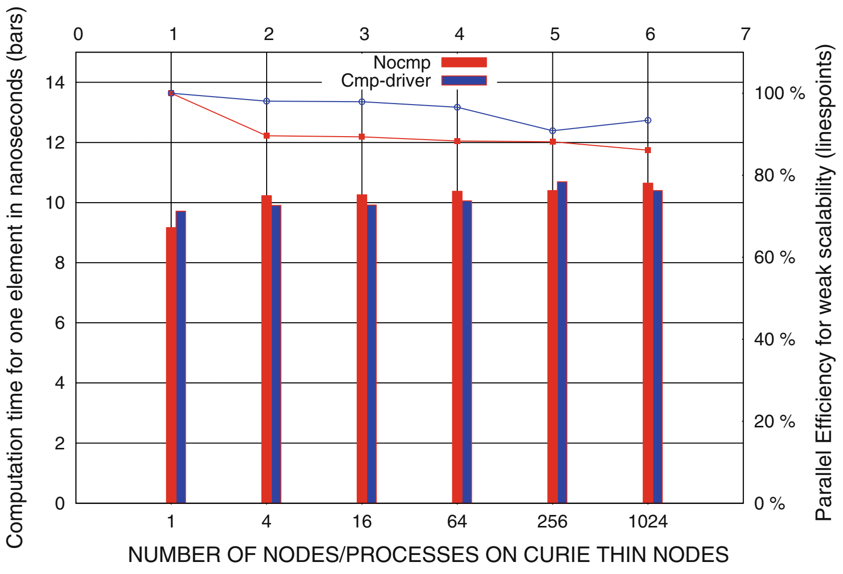

Fig. 11 Jacobi weak scalability with fixed size (640 × 640 elements per node, one process per node) on Curie thin nodes

Table 3 Cyclomatic complexity of the native codes and of the component based codes

Version	Native—no component	Driver version	Connector version
Sequential	28	32	8
Multithread	76	41	26
MPI	55	22	13

one big component. It confirms modularity helps reducing complexity if separation of concerns is adequately handled.

6 Conclusion

For the development and adaptability of HPC applications on various hardware resources, this paper has proposed and evaluated the L^2C model easing performance portability. L^2C supports multiple kinds of interactions between components with very low runtime overhead if any. Its evaluation has been conducted on some typical hardware architectures with respect to a well-known Jacobi application. The experimental results obtained on a GRID’5000 cluster and on the Curie supercomputer demonstrate that L^2C makes it possible to implement separation of concerns between domain specific code and hardware specific code. With code written to fully take advantage of the L^2C features, it eases performance portability to different HPC hardware architectures by enabling the manipulation of the application structure. Moreover it increases code reuse, without degrading performances.

Though components bring many advantages, they also introduces a requirement: turning existing code into a component based code. In addition of defining and implementing components, one also needs to describe their assembly for each specific hardware architecture. As it is fastidious and error prone, such assembly description should be automatically generated. This is one of the purposes of HLCM [13]. During an assembly generation, algorithms that determine the best assembly for a given architecture can be supported.

L^{2}C supports C++, FORTRAN, MPI, and CORBA. Ongoing work aims to show its benefits when applied to CHARM++ on FFT codes. However, more work is needed to integrate GPGPU, as well as applications with a dynamic structure. An important challenge is to efficiently adapt applications to runtime conditions. It would be particularly useful when dealing with faults or with cloud environments where adaptation is very important.

Acknowledgments This work has been supported by the PRACE-2IP project of the European Community's Seventh Framework Programme (FP7/2007–2013), with grant agreement n°RI-283493, in particular to access the Curie supercomputer. This work has also used the French GRID'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER, several Universities, and other funding bodies (see https://www.grid5000.fr).

References

1. Szyperski C (2002) Component software: beyond object-oriented programming. Addison-Wesley, Boston
2. Barbara C, Gabriele J, van der Ruud P (2007) Using OpenMP: portable shared memory parallel programming (scientific and engineering computation). MIT Press, Cambridge
3. Wienke S, Paul Springer P, Terboven C, An May D (2012) OpenACC—first experiences with real-world applications. Euro-Par (2012) Parallel Processing, Rhodes Island, Greece
4. Kale LV, Bohm E et al (2008) Programming Petascale applications with charm++ and AMPI. Petascale computing: algorithms and applications. Chapman & Hall CRC Press, Boca Raton
5. Coarfa C, Dotsenko Y, Mellor-Crummey J, Cantonnet F, El-Ghazawi T, Mohanti A, Yao Y, Chavarra-Miranda D (2005) An evaluation of global address space languages: co-array fortran and unified parallel C. In: Proc. of the 10^{th} ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP'05). ACM, New York, pp 36–47
6. Du P, Weber R, Luszczek P, Tomov S, Peterson G, Dongarra J (2012) From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming. Parallel Comput 8:391–407
7. Augonnet C, Thibault S, Namyst R, Wacrenier P-A (2011) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr Comput Pract Exp Spec Iss Euro-Par 2009 23:187–198
8. Benkner S, Pllana S, Larsson J (2011) PEPPHER: efficient and productive usage of hybrid computing systems. IEEE Micro 31(5):28–41
9. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Boston
10. Object Management Group (2008) Common object request broker architecture specification, version 3.1, part 3: CORBA component model
11. Baude F, Caromel D, Dalmasso C, Danelutto M, Getov V, Henrio L, Pérez C (2009) GCM: a grid extension to FRACTAL for autonomous distributed components. Spec Issue Ann Telecommun Softw Compon Fract Initiat 64(1):5
12. Allan BA, Armstrong R, Bernholdt DE, Bertrand F, Chiu K, Dahlgren TL, Damevski K, Elwasif WR, Epperly TGW, Govindaraju M, Katz DS, Kohl JA, Krishnan M, Kumfert G, Larson JW, Lefantzi S, Lewis MJ, Malony AD, McInnes LC, Nieplocha J, Norris B, Parker SG, Ray J, Shende S, Windus TL,

Zhou S (2006) A Component Architecture for High-Performance Scientific Computing. Int J High Perform Comput Appl 20(2):163–202
13. Bigot J, Pérez C (2011) High Performance Composition Operators in Component Models, vol. 11, pp. 182–201. doi:10.3233/978-1-60750-803-8-182
14. Bigot J, Pérez C (2010) Enabling connectors in hierarchical component models. INRIA, RR-7204
15. McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng 4:308–320