Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Evaluating Component Assembly Specialization for 3D FFT

Jérôme Richard[a,†], Vincent Lanore[b,†], and Christian Perez[c,†,1]

[a] University of Orléans, France
[b] École Normale Supérieure de Lyon, France
[c] Inria
[†] Avalon Research-Team, LIP, ENS Lyon, France

Abstract

The Fast Fourier Transform (FFT) is a widely-used building block for many high-performance scientific applications. Efficient computing of FFT is paramount for the performance of these applications. This has led to many efforts to implement machine and computation specific optimizations. However, no existing FFT library is capable of *easily* integrating and automating the selection of new and/or unique optimizations.

To ease FFT specialization, this paper evaluates the use of component-based software engineering, a programming paradigm which consists in building applications by assembling small software units. Component models are known to have many software engineering benefits but usually have insufficient performance for high-performance scientific applications.

This paper uses the L^2C model, a general purpose high-performance component model, and studies its performance and adaptation capabilities on 3D FFTs. Experiments show that L^2C, and components in general, enables easy handling of 3D FFT specializations while obtaining performance comparable to that of well-known libraries. However, a higher-level component model is needed to automatically generate an adequate L^2C assembly.

1 Introduction

The Fast Fourier Transform (FFT) is an efficient algorithm to compute the Discrete Fourier Transform (DFT), a numerical operation widely used in several scientific domains (*e.g.*, molecular dynamics, signal processing, meteorology). FFT is used as a building block for many high-performance scientific applications that handle large amounts of data and that target distributed hardware architectures (supercomputers, clusters). The FFT part of those applications can contribute largely to the overall computing time.

Thus, efficient distributed FFT implementations have been the focus of a lot of research and many optimizations have been proposed. Many of those optimizations take advantage of specific hardware architectures and/or of the characteristics of the FFT to compute (*e.g.*, data size). Since hardware evolves rapidly, new optimizations are regularly devised. Consequently, FFT codes must often be tweaked to adapt to new architectures to maximize performance.

Adapting a FFT code for a specific use has a cost in terms of development time and requires a good knowledge of both the target platform and the FFT literature. It might also prove difficult for someone other than the writer of the original code (*e.g.*, when an external FFT library is used). Moreover, unless automated, adapting the code for a specific run (*e.g.*, for a specific data size or reservation size) is, in many cases, too costly. Some existing libraries (*e.g.*, FFTW codelet framework [1], OpenMPI MCA framework [2]) provide some forms of adaptation framework but, to our knowledge, none is able of easily integrating and automating new and/or unique optimizations.

A promising solution, being investigated, to easily handle FFT specialization is to use component-based software engineering techniques [3]. This approach proposes to build applications by assembling software units with well-defined interfaces; those units are called *components*. Syntax and semantics of interfaces and

[1]Corresponding author: christian.perez@inria.fr

assemblies are given by a *component model*. Such an approach allows for easy reuse of (potentially third-party) components and for high-level adaptation through the assembly. Also, some component models [4, 5] and tools allow for automatic assembly generation and/or optimization.

Component models bring many software engineering benefits but very few provide enough performance for high-performance scientific applications. Among them is L^2C [6], a low-level general purpose high-performance component model built on top of C++/FORTRAN and MPI. This paper focuses on studying its performance and adaptation capabilities on a 3D FFT use case. Our experiments and adaptation analysis show that it is possible to quite easily handle 3D FFT specializations (with high reuse, without delving into low-level code and with as little work as possible) while having performance comparable to that of well-known FFT libraries.

Section 2 presents the 3D FFT related work, in particular some common algorithms, optimizations and existing libraries. Then, Section 3 deals with component models and gives an overview of L^2C. Section 4 describes the assemblies of various flavors of 3D FFTs that we have designed and implemented with L^2C. Section 5 compares the 3D FFT L^2C assemblies with existing FFT libraries both in terms of performance and in terms of reuse/ease of adaptation. Section 6 concludes and gives some perspectives.

2 3D FFT

The Fourier Transform is a mathematical transformation used to convert signals from a spatial (or time) domain to a frequency domain or the other way round. The Fast Fourier Transform (FFT) is an efficient and widely-used algorithm to compute discrete Fourier transforms which is widely used in scientific computing. For example, FFT is used in the molecular dynamic package GROMACS [7]); it is also used to solve partial differential equations [8] or to accelerate multiplication of large integers [9].

2.1 Sequential FFTs

The FFT algorithm proposed by Cooley and Tukey in 1965 [10] is a well-known and efficient algorithm to compute the Discrete Fourier Transform (DFT) in $O(N\ log(N))$ time (where N is the input size) used in many FFT libraries.

Multidimensional FFT can be easily computed by applying unidimensional FFTs on each dimension. This can be done using multiple phases of unidimensional FFTs computation applied to the same axis interlaced by data transposition phases. In sequential implementation, this operation can be very expensive for large matrices.

2.2 Parallelization of 3D FFTs

Multidimensional FFTs have been parallelized to handle large matrices. Let us focus on 3D FFTs.

Existing methods to compute 3D FFT in parallel can be classified in two groups: those that use a global matrix transposition and those that use a binary exchange pattern. Using a global transposition is known to scale better on modern petascale supercomputers, even on a hypercube network where binary exchange has a natural mapping [11]. This paper thus focuses on the global transposition approach.

Let us introduce some notation and conventions related to global transposition. Let us consider a cube of data of size $N \times N \times N$ along axes X, Y and Z. This cube is stored in memory as a 3D array in row-major order. Thus, cells along the X axis are contiguous in memory and form lines which are stored contiguously along the Y axis. Those lines, in turn, form thin slabs which are stored contiguously.

The simplest way to compute a parallel 3D FFT using global transposition is to distribute data slices (called slabs) along the Z axis between processor elements (PEs) and to interlace computation and transposition phases. Thus, each PE store a block of size $N \times N \times n_z$ where $n_z \leq N$. Algorithm 1 and Figure 1 provide an overview of this first approach. This method is called slab decomposition or 1D decomposition [12] because data is split along one dimension (in this case, the Z axis). The data transposition can be achieved by using an all-to-all global exchange (*e.g.*, using `MPI_Alltoall` or `MPI_Alltoallv` collectives in MPI) followed by a local transposition. This approach is efficient up to N PEs at which point each PE has a slab of height 1. It is impossible to distribute data on more than N PEs with a 1D decomposition.

Algorithm 1: 1D decomposition scheme

```
   Data: XY-slabs of data in the spatial domain
   Result: XY-slabs of data in the frequency domain
1  Apply 2D FFTs on each local slab of data;
2  XZ slab transposition;
3  Apply 1D FFTs on $X$ axis of each slab of data;
4  XZ slab transposition;
```

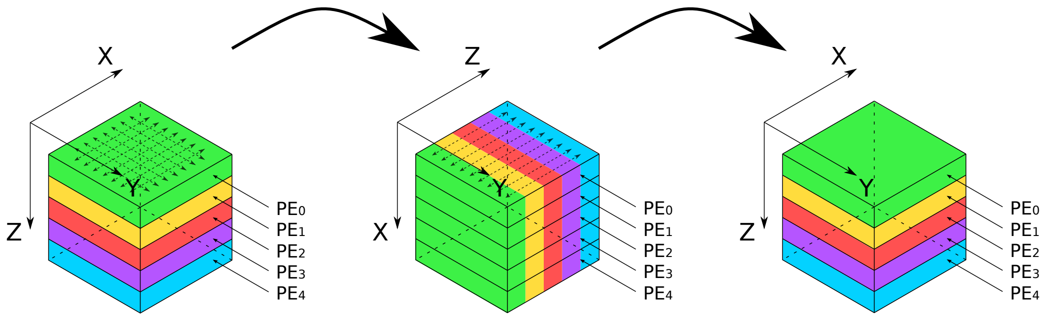


Figure 1: 1D decomposition scheme with an XZ transposition. Colors represent organization of data and dashed arrows represent FFT computation phases.

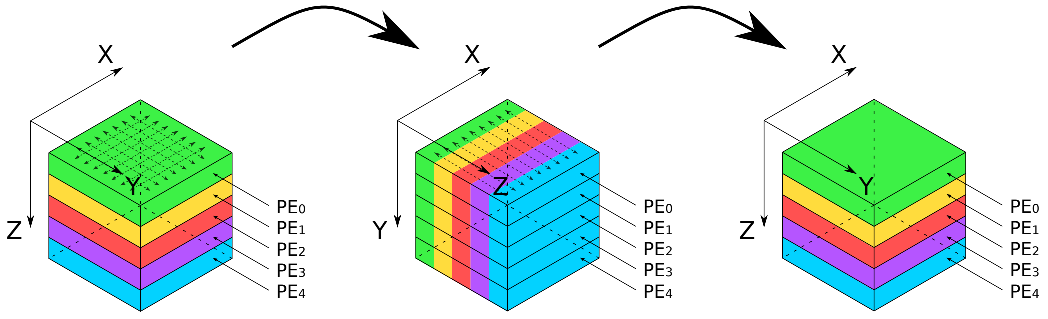


Figure 2: 1D decomposition scheme with an YZ transposition.

Note that it is possible to do either a XZ transposition as shown in Figure 1 or an YZ transposition as shown in Figure 2. Those two approaches will lead to different memory access patterns and thus can have different performance depending on the architecture.

The inherent scalability limit of the slab decomposition is a problem on modern supercomputers which can have millions of PEs [13]. To overcome this limitation, data can be distributed along two axes: Y and Z. Instead of slabs, data is distributed in pencils amongst PEs. This is called a pencil/2D decomposition [12]. An overview of such an algorithm is given in Algorithm 2 and Figure 3. Note that XY and XZ transpositions can be swapped in this algorithm. This approach can scale up to N^2 PEs (instead of N for the 1D decomposition).

Algorithm 2: 2D decomposition scheme

```
   Data: X-pencil of data in the spatial domain
   Result: X-pencil of data in the frequency domain
1  Apply 1D FFTs on each X-pencil of data;
2  XY pencil transposition;
3  Apply 1D FFTs on each X-pencil of data;
4  XZ pencil transposition;
5  Apply 1D FFTs on each X-pencil of data;
6  XZ pencil transposition;
7  XY pencil transposition;
```

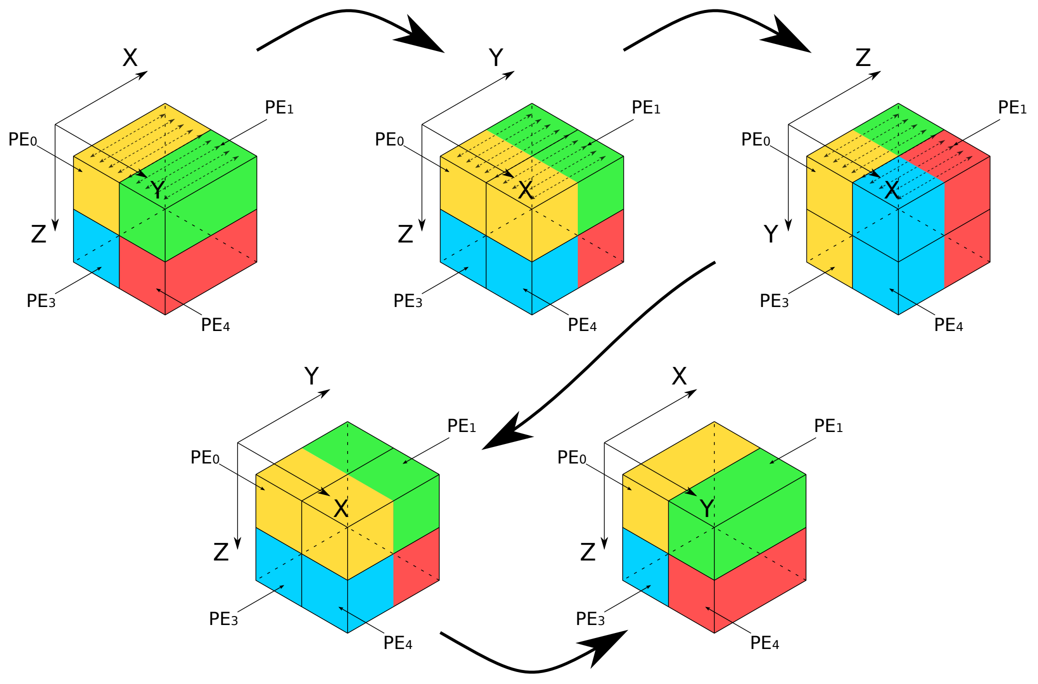


Figure 3: 2D decomposition scheme.

2.3 FFT Optimizations

Beyond the choice of 1D/2D decomposition discussed above, many optimizations can be done to the base FFT algorithm to take advantage of the diverse hardware architectures that are available or of the specific properties of the application.

First, it is possible to optimize the all-to-all exchange part of the algorithm. Multiple algorithms exist whose performance depends on multiple parameters such as message sizes, network topology, latency of network links, etc. For example, Bruck et al. proposed an algorithm that is particularly efficient for short messages [14]; Prisacari et al. proposed an algorithm that is particularly efficient for big messages in fat tree networks [15]. MPI implementations such as OpenMPI and MPICH select these algorithms either at launch or at runtime to try to maximize performance [16, 17].

On some hardware architectures, data distribution can have a significant effect on performance. For example, the native MPI collective `MPI_Alltoallv`, that is used for unevenly distributed data, is inefficient on Cray XT supercomputers [18]. Padding send/receive buffers and using a `MPI_Alltoall` instead (which works only for evenly distributed data) improves performance on Cray XT. Similarly, The Cray XT family supercomputers provides a specific shared memory system which can be used to enhance performance [19].

Another possible optimization is to overlap computations with communications. This optimization may introduce an overhead, modifies the communication pattern and, thus, does not always improve performance. Kandalla and al. [20] show that quasi-perfect overlapping can be achieved on 3D FFT using network offload; it results in a practical improvement of 23% compared to non-overlapped computations.

Some other optimizations take advantage of the actual application that use the FFT. Let us consider, for example, the case of an application that computes a convolution product on a cube of data. This can be done by computing a FFT, multiplying values by the convolution filter factors and computing an inverse FFT [21]. In this case, two transpositions can be avoided during the whole processing if 1D decomposition is used [18]; indeed, the last transposition in the 1D algorithm (see Figure 1) serves only to put the matrix back in its original orientation which is not required here.

2.4 FFT Libraries

2.4.1 Sequential libraries

A large amount of libraries, either open-source or commercial, provide sequential FFT implementations. The Fastest Fourier Transform in the West [1] (FFTW), developed by Matteo Frigo and Steven G. Johnson, is one of the most widely-used cross-platform libraries. This library is built to be fast on many hardware architectures. Examples of other open source libraries that compute FFT are Eigen [22] and the GNU Scientific Library [23] (GSL). Classical commercial libraries include the IBM Engineering and Scientific Software Library (ESSL) and the Intel Math Kernel Library (MKL).

2.4.2 Parallel libraries

Some FFT libraries provide multithreaded implementations and, in some cases, distributed implementations, usually MPI based. Many of these libraries use a well-known sequential FFT implementation for the sequential FFT part. FFTW and MKL, mentioned above, provide their own parallel implementations. Other libraries such as Parallel Three-Dimensional Fast Fourier Transforms [18] (P3DFFT) and 2DECOMP&FFT [19] are based on external FFT sequential implementations, such as FFTW or ESSL. Experiments of this paper have considered FFTW, P3DFFT, and 2DCOMP&FFT as reference FFT implementations to evaluate our approach (see Section 5):

FFTW Multiple variants of FFTW (version 3) exist: a threaded implementation which uses only one thread by default, a MPI version for distributed memory architectures (using slab decomposition) and a Cilk implementation [24] for shared memory SMP architectures. To achieve high performance, FFTW uses self-optimizing strategies: highly optimized pieces of C codes called codelets designed to compute small FFTs are assembled together to form a program during a planning phase. Codelets can be automatically produced from a high-level mathematical description of a DFT algorithm using a generator. The multi-threaded parallel version of FFTW is a special variant which supports parallel planning.

P3DFFT Developed by Dmitry Pekurovsky, P3DFFT [18] is an open-source FFT parallel library for distributed memory architectures. This library offers both 1D and 2D decompositions and was built to scale well on petascale platforms. P3DFFT can use different sequential FFT libraries like ESSL or FFTW. This library supports both real-to-complex and complex-to-real FFTs but it does not yet support complex to complex FFTs[2]. To improve performance, P3DFFT implements specific optimizations such as the replacement of `MPI_Alltoallv` by `MPI_Alltoall` for the Cray XT family supercomputers. These optimizations are implemented using conditional compilation. PD3FFT scales up to 65k cores.

2DECOMP&FFT Developed by Ning Li and Sylvain Laizet, 2DECOMP&FFT [19] is another open-source FFT parallel library for distributed memory architecture. But unlike P3DFFT, 2DECOMP&FFT supports both complex-to-complex, real-to-complex and complex-to-real. Like P3DFFT, some specific optimizations are implemented (*e.g.*, Cray XT optimizations) using conditional compilation.

2.4.3 Discussion

Because supercomputer architectures evolve rapidly, new specific optimizations must be implemented to maximize performance on every type of hardware. Each new optimization can introduce extra code and/or impact the whole structure of the application. Using conditional compilation leads to interleaving multiple pieces of specific optimized codes, thus decreasing readability and causing maintainability issues.

As there are many new hardware platforms and optimizations to implement, it requires a lot of effort for library developers to implement all of them. In practice, there is not a library that implements all optimizations. From the point of view of a user needing a specific set of optimizations that is not supported by an existing library, it means they must either re-develop a specific FFT code or delve into the code of an existing library to tweak it for their purpose.

Some parallel libraries rely on external sequential libraries, making a step toward separation of concerns. However, adapting these parallel libraries to use a new sequential library or new features of already supported sequential libraries can be a tricky work because their use is implemented in the core of these libraries.

[2]The implementation is in progress.

3 Component Models

3.1 Overview

Definitions The concept of software components was initially proposed by Douglas McIlroy in 1968 [25] and has since been the focus of a lot a research. Component-based engineering is a programming paradigm which proposes to compose software units called components to form a program. A component model is a programming model which defines components and component composition.

There is no widely-accepted definition of what a software component is. A classical definition has been proposed by Clemens Szyperski [3]: *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.* In many component models, interfaces are called ports and have a name. Most component models aim at ensuring the separation of concerns between component interfaces and their implementations for maximum application modularity.

To produce a complete application, components must be instantiated and then assembled together. This can be done by connecting interfaces between them. The actual nature of connections is defined by the component model and may largely vary from one model to another.

Benefits Component models specifically aim at separation of concerns and reuse of third-party software. Separation of concerns is achieved by separating the role of component programming (low-level, implementation details) from component assembly (high-level, application structure). Reuse of third-party components is possible because component interfaces are all that is needed to use a component; it is thus not necessary to be familiar with low-level details of the implementation of a component to use it.

Separation of concerns and reuse would allow to easily mix pieces of FFT codes from different sources to make specialized FFT assemblies. Thus, FFT adaptation would no longer require in-depth understanding of existing implementations (separation of concerns) or re-development of existing optimizations (reuse).

Existing Component Models This paper focuses on distributed component models. The CORBA Component Model [26] (CCM) and the Grid Component Model (GCM) [27] are notable examples of distributed models. However, they generate runtime overheads [28] that are acceptable for distributed application but not for HPC. Common Component Architecture [29] (CCA) is the result of an US DoE effort to enhance composability in HPC. However, CCA is mainly a process-local standard that relies on external models such as MPI for inter-process communication. As a consequence, such interactions do not appear in component interfaces. The Low Level Component (L^{2}C) Model [6] is a minimalist HPC Component Model built on top of C++/FORTRAN with no overhead at runtime. It provides primitive components, local connections (C++ and FORTRAN *uses/provides* ports), as well as MPI connections (components share an MPI communicator) and CORBA connections.

As this paper studies the use of L^{2}C to address the problem of adapting 3D FFTs, the next section presents the model in more detail.

3.2 L^{2}C Model

The L^{2}C model can be seen as an extension of modular compilation or as a low level component model that does not hide system issues. Indeed, each component is compiled as an object file. At launch time, components are instantiated and connected together according to an assembly description file or to an API.

L^{2}C supports various features like memory sharing, C++/FORTRAN procedure invocations, message passing with MPI, and remote procedure calls with CORBA. L^{2}C components can provide services thanks to *provides* ports and use services with *uses* ports. Services have to be declared as C++, FORTRAN or CORBA interfaces. Multiple *uses* ports can be connected to a unique *provides* port. A port is associated with an object interface and a name and communication between component instances is done by procedure calls on ports. L^{2}C also provides MPI port as a way to share MPI communicators between components. Components can also expose attributes used to configure component instances. The C++ mapping defines L^{2}C components as plain C++ classes with a few additional annotations to declare the component with its ports and properties. In the current version of L^{2}C, the FORTRAN mapping requires FORTRAN 2008 features.

An L^{2}C assembly can be described using a L^{2}C assembly descriptor file (LAD). This file contains a description of all component instances, their attributes values, and the connections between instances. Each component is part of a process and each process has an entry point (an interface that is called when the application starts). It also contains the configuration of MPI ports.

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

4 Designing 3D FFT Algorithms with L^{2}C

This section analyzes how L^{2}C can be used to implement distributed 3D FFT assemblies based on the use of global transpositions (see Section 2.2). To that end, we have first designed a basic 3D FFT assembly. Then, we have modified it to take into account some optimizations presented in Section 2.3. Optimizations are applied in three stages to highlight different component model features: *i)* replacing a component implementation with a more optimized implementation (Section 4.2), *ii)* using component attributes for heterogeneous platforms tuning (Section 4.3), and iii) global assembly adaptation to implement computation/communication overlapping and 2D decomposition (Section 4.4).

All the assemblies presented here have been implemented in C++/L^{2}C, and relevant assemblies are evaluated in Section 5.

4.1 Basic Assembly

4.1.1 Local Basic Assembly

Figure 4 displays an assembly that implements Algorithm 1. Let us detail it.

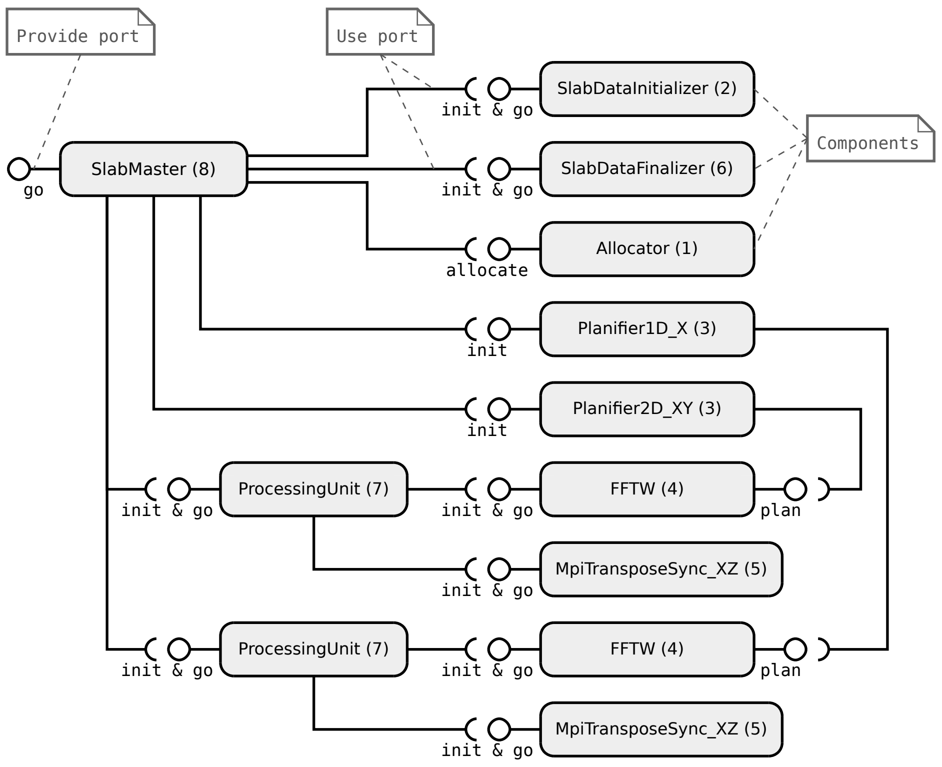


Figure 4: Local (one process) basic 3D FFT assembly using 1D decomposition.

From Algorithm 1, we have identified 8 tasks that can be mapped to components: 6 for the actual computation, and 2 for the control of the computation. The computation components implement the following tasks:

1. `Allocator`: allocate 3D memory buffers.
2. `SlabDataInitializer`: initialize input data.
3. `Planifier1D_X` and `Planifier2D_XY`: plan fast sequential FFTs.
4. `FFTW`: compute FFTs (wrapping of FFTW library).
5. `MpiTransposeSync_XZ`: transpose data between nodes.
6. `SlabDataFinalizer`: finalize output data by storing or reusing it.

The two control components implement the following tasks:

7. `ProcessingUnit`: group FFT computations and transpositions within a processing unit.
8. `SlabMaster`: control the global application structure.

Task 2 (`SlabDataInitializer`), 6 (`SlabDataFinalizer`) and 8 (`SlabMaster`) are specific to the 1D decomposition. Those tasks and Task 5 (`MpiTransposeSync_XZ`) are specific to a given parallelization strategy. Task 3 (`Planifier1D_X` and `Planifier2D_XY`) and Task 4 (`FFTW`) are specific to the chosen sequential FFT library.

All computation-oriented components except `Allocator` rely on memory buffers. They can use two buffers (*i.e.* an input buffer and an output buffer) or just one for in place memory computation. These buffers are initialized by passing memory pointers during the application startup. For this purpose, these components provide an `Init` port to set input and output memory pointers, but also to initialize or release temporary resources. When only a single buffer is needed, the input and output buffers are the same. These components also provide a `Go` port allowing to compute from input data and write the results inside the output buffer.

As FFT plans depend on the chosen sequential FFT library, Component `FFTW` (Task 4) exposes a specific `Plan` interface that is used by `Planifier1D_X` and `Planifier2D_XY` (Task 3). This connection is use to configure the FFT components after the planning phase.

The application works in three stages. The first step consists in initializing the whole application by allocating buffers, planning FFTs and broadcasting pointers and plans to component instances. The second stage drives the computation by invoking method calls on `Go` port of component instances to interlace FFT computation and communication. The last stage consists in releasing resources such as memory buffers. The whole processing (*i.e.* initialization, FFT planning, FFT computation, and finalization) starts via the `Go` port of the `SlabMaster` component.

The assembly has been designed to be configured for a specific computation of a 3D FFT. Buffers size and offsets are described as components attributes and are not computed at runtime. This enables to reduce application overhead.

4.1.2 Distributed Basic Assembly

Figure 5 describes a distributed extension of the local basic assembly presented in the previous section. The distributed version of the assembly is obtained by adding MPI port to `SlabDataInitializer` (Task 2), `MpiTransposeSync_XZ` (Task 5), `SlabDataFinalizer` (Task 6), and `SlabMaster` (Task 8). Furthermore, this assembly is duplicated on each MPI process. `MpiTransposeSync_XZ` instances of a same computation phase are interconnected through their MPI ports, so that they share an MPI communicator. It is also the case for `SlabMaster`, `SlabInitializer`, and `SlabFinalizer` instances. The resulting assembly is implemented as version called `L2C_1D_2t_xz` and it is the base assembly to evaluate reuse of other assemblies in Section 5.

4.2 Assembly Optimization by Replacing Components

The 3D FFT implementation can be turn more efficient by just replacing some component implementations, especially the transposition component, by more optimized ones. Component instances are easily replaced by other instances that expose the same interface. For example, it is possible to optimize the assembly

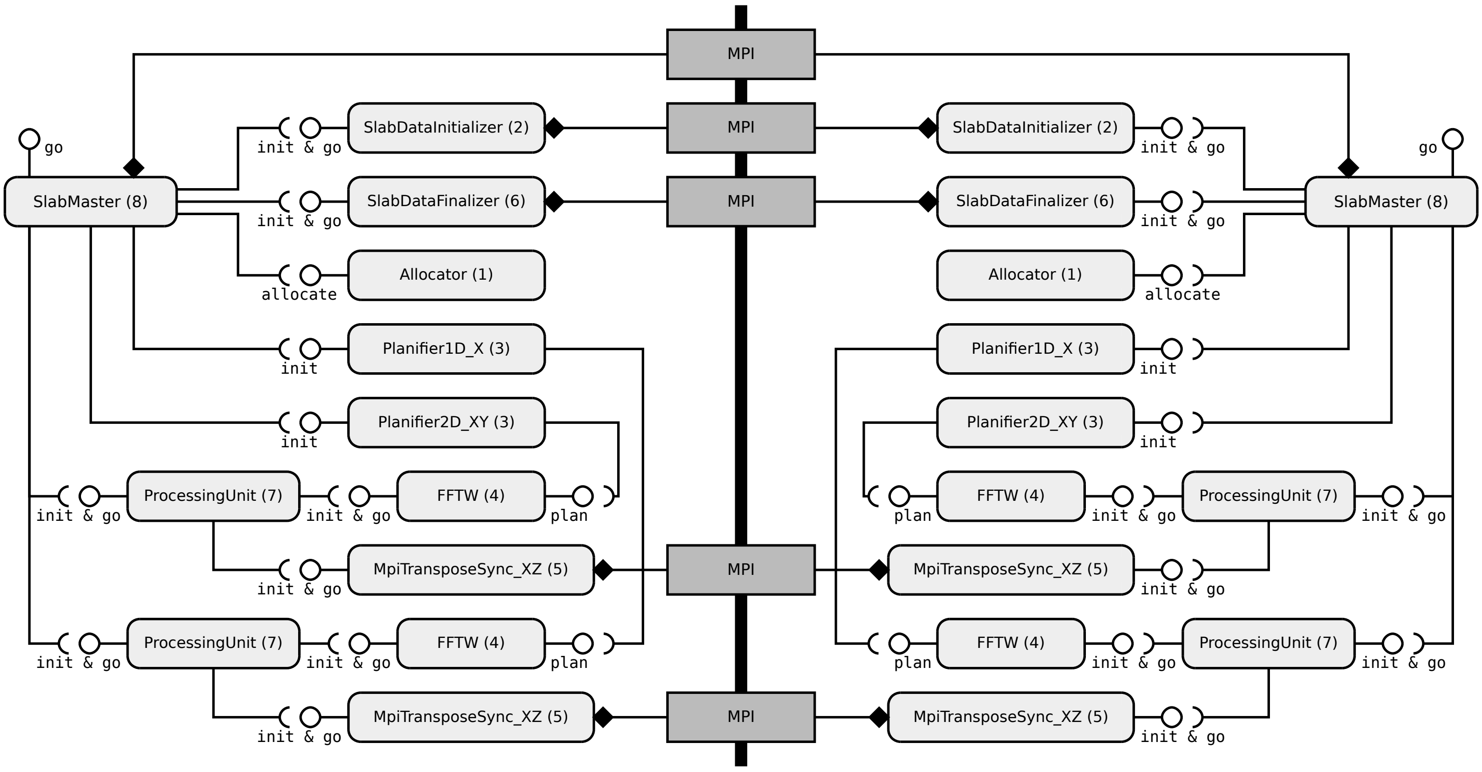


Figure 5: Distributed basic 3D FFT assembly for 2 MPI processes using 1D decomposition.

by replacing all MpiTransposeSync_XZ components by MpiTransposeSync_YZ components, and then by replacing all Planifier1D_X components by Planifier1D_Y components: those components compute 1D FFT on slabs along the Y axis as explained in Section 2.2 and presented in Figure 2. During the transposition, data is read in a more contiguous way in memory resulting in performance gain during the data transposition. However, the FFT library has to compute 1D FFT along a noncontiguous axis in memory resulting in a performance loss during the computation phase. Thus, this optimization is interesting only if the FFT computation performance loss is smaller than the transposition performance gain which is often the case since global transposition is very costly on large architectures. This optimization is implemented in assemblies L2C_1D1t_yz and L2C_1D2t_yz and evaluated in Section 5.

Another possible optimization consists in not computing local transpositions and applying 1D FFT along the Z axis of untransposed data. This avoids some memory copies but can only be applied for 1D decomposition when two transpositions are performed. This optimization is implemented in assemblies L2C_1D_2t_yz_blk and evaluated in Section 5

4.3 Assembly Optimization by Adapting Attributes

Component instance attributes can be set to take into account heterogeneous hardware architectures, such as for example the thin and large nodes of the Curie supercomputer. Indeed, when all nodes do not compute at the same speed and data is evenly distributed between nodes, the slower node limits the whole computation due to load imbalance. To deal with this problem, load balancing is needed and thus data must be unevenly distributed between nodes. A solution is to control data distribution through component attributes. A new transposition component must be implemented to handle uneven data distribution. Such a component has been implemented and the resulting assemblies (L2C_1DH_1t_yz, L2C_1DH_2t_yz_blk, and L2C_2DH_3t) are evaluated in Section 5 on heterogeneous scenarios.

4.4 Global Assembly Adaptation

Reducing the Number of Transpositions In many cases, the transposition phase is the main limitation; it is thus of interest to optimize it. As explained in Section 2.3, a transposition can be avoided in

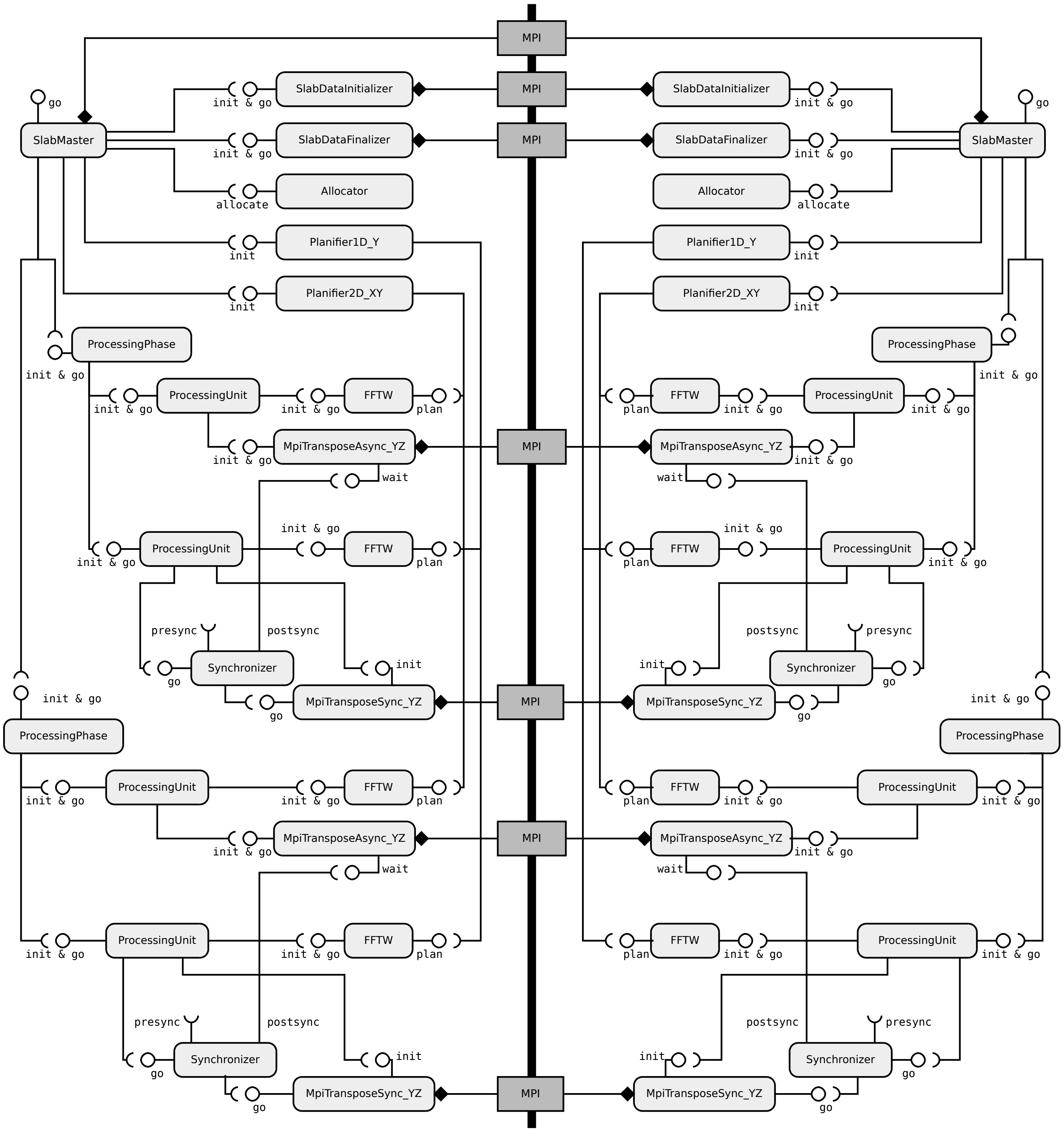


Figure 6: Overlap enabled (with 2 blocs per phases) distributed 3D FFT assembly for 2 MPI processes using 1D decomposition.

some cases using the 1D decomposition scheme (and up to two transpositions using a 2D decomposition). This approach enables to remove the final transposition in the application by adapting the assembly and by adding an attribute to the `Master` component: in each process, the second `ProcessingUnit` component instance connected to the `SlabMaster` via `Init` and `Go` ports is removed, and the transposition component is connected to it via the same port type; the `Go` and `Init` *uses* ports of the `SlabMaster` component are directly connected to the associated *provides* ports of components implementing the Task 4 of the last phase. As the `SlabMaster` behavior is different during the initialization depending on whether a final transposition is used or not (especially the final buffer differs), a boolean property is added to the `SlabMaster` to configure it. This assembly has been implemented and it is tested in Section 5 as assemblies which save one transposition (`L2C_1D_1t_yz`, `L2C_1DH_1t_yz`, `L2C_2D_3t` and `L2C_2DH_3t`). The current 2D decomposition assembly implementation computes one extra transposition. This extra transposition can be avoided using a similar assembly transformation but this optimization has not been implemented.

Computation/Communication Overlapping Computation/communication overlapping can be achieved by adding new components, replacing instances and adapting the assembly. Indeed, overlapping using 1D decomposition in our model is achieved by replicating `ProcessingUnit` and the associated FFT and transposition component instances. As the matrix has been distributed over more component instances, each FFT and transpose component instance is configured to deals with less data than the case without overlapping. It is achieved by adapting the properties that define the data size. A new `ProcessingPhase` component is introduced to route calls received from a `SlabMaster` instance to other instances such as `ProcessingUnit` instances. When a method is called on the `Init` *provides* port of the component, it calls the same method with the same parameters on every associated *provides* port connected to the `Init` *uses* port one after another. The same processing is done on the `Go` port. The interface of the transposition component is revised to allow asynchronous operations and synchronizations: the `Go` port now starts the transposition asynchronously and a `Wait` port is added to allow synchronizations between multiple instances. Developer-transparent synchronizations are achieved with a new component: an adapter called `Synchronizer` that can force instances to be synchronized before or after an interaction with the `Go` *provides* port when a method is called on its. An example of an assembly with overlapping is presented in Figure 6. This assembly is not implemented and is not tested in this paper.

2D Decomposition Because of the limitation of the 1D decomposition scaling described in Section 2.2, 2D decomposition assemblies are needed to achieve better performance. This can be done by adapting the assembly as displayed in Figure 7. The modifications have concerned the introduction of a new transposition component, and the replacement of three components: the `SlabMaster`, `SlabInitializer`, and `SlabFinalizer`. These three components are respectively replaced by `PencilMaster`, `PencilInitializer`, and `PencilFinalizer`. These new components provide two new MPI ports to communicate with instances which handle the pencil of the same processor row or on the same processor column. In this new assembly, two computing phases are also added and are managed by the `PencilMaster`. Because the 2D decomposition scheme introduces a XY transposition of distributed data not needed in the 1D decomposition scheme, a new transpose component has needed to be developed. However, the XZ transposition component can be reused from the 1D decomposition scheme. This assembly is implemented and tested in Section 5 as versions which use the 2D decomposition scheme (`L2C_2D_3t` and `L2C_2DH_3t`).

5 Performance and Adaptability Evaluation

This section evaluates the component based approach in terms of performance and adaptability of some assemblies described in the previous section. Performance and scalability are evaluated on up to 8192 cores on homogeneous and heterogeneous architectures. Adaptability relates to the easiness to implement the various optimizations, and how much code has been reused.

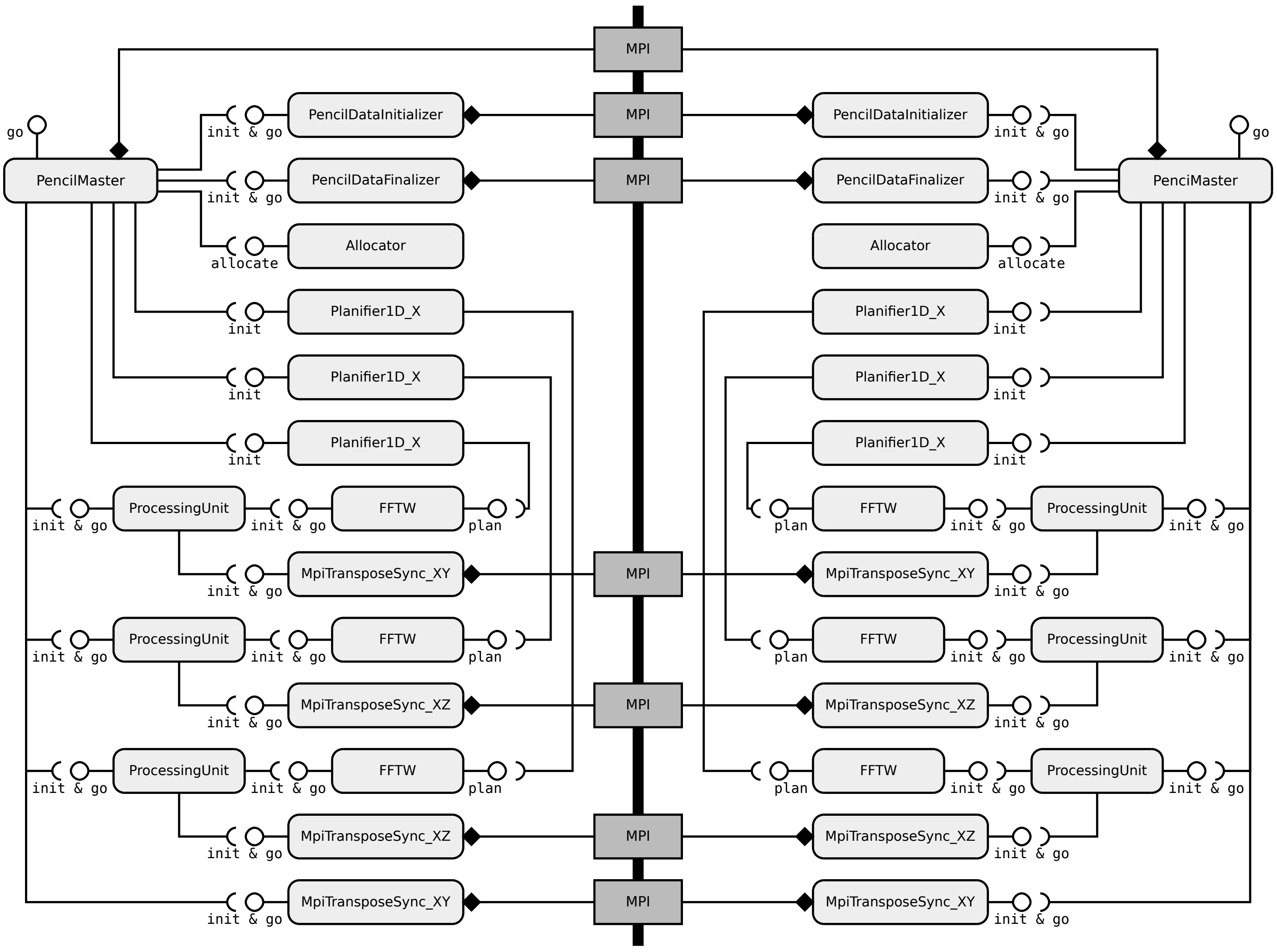


Figure 7: Distributed 3D FFT for 2 MPI processes which overlap communication and computation using 2D decomposition.

Cluster Name	# Nodes	CPU/ Node	Cores/ CPU	Cores/ Node	CPU Type	Freq. (GHz)	Network
Griffon	92	2	4	8	Xeon L5420	2.5	InfiniBand 20G
Graphene	144	1	4	4	Xeon X3440	2.53	InfiniBand 20G
Edel	34	2	4	8	Xeon E5520	2.27	InfiniBand **40G**
Genepi	72	2	4	8	Xeon E5420 QC	2.5	InfiniBand **40G**

Figure 8: Description of used Grid'5000 cluster.

Assembly Name	Decomposition	#Transposition	Heterogeneity Support
`L2C_1D_2t_xz`	1D	2	no
`L2C_1D_1t_yz`	1D	1	no
`L2C_1D_2t_yz`	1D	2	no
`L2C_1D_2t_yz_blk`	1D	2	no
`L2C_1DH_1t_yz`	1D	1	yes
`L2C_1DH_2t_yz_blk`	1D	2	yes
`L2C_2D_3t`	2D	3	no
`L2C_2DH_3t`	2D	3	yes
Library Name	Decomposition	#Transposition	Heterogeneity Support
`FFTW`	1D	2	not used
`FFTW_1t`	1D	1	not used
`2DECOMP_1D1t`	1D	1	not available
`2DECOMP_1D2t`	1D	2	not available
`2DECOMP_2D`	2D	2	not available

Figure 9: Assemblies and libraries used in the experiments.

5.1 Performance Evaluation

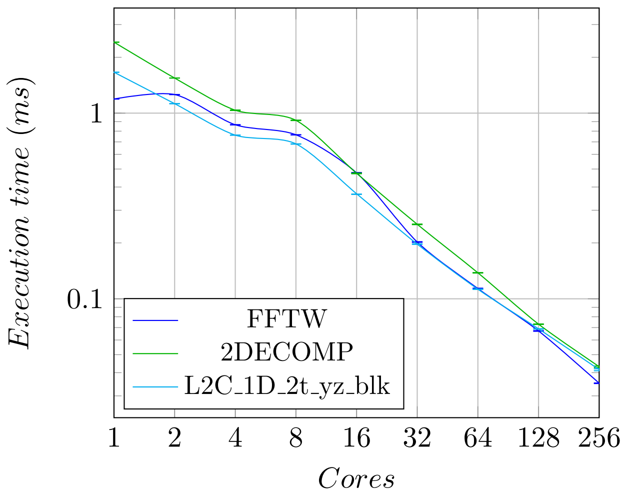
5.1.1 Experimental Setup and Methodology

Target Architectures A first group of experiments have been done on multiple clusters of the Grid'5000 experimental platform [30] (Section 5.1.2 and Section 5.1.3). These clusters are Griffon, Graphene, Edel and Genepi. Figure 8 details the hardware of each of these clusters. Heterogeneous tests have been done with the Genepi and the Edel clusters. Both clusters are connected to the same InfiniBand network. However, they have different processors which make them suitable for heterogeneous experiments.

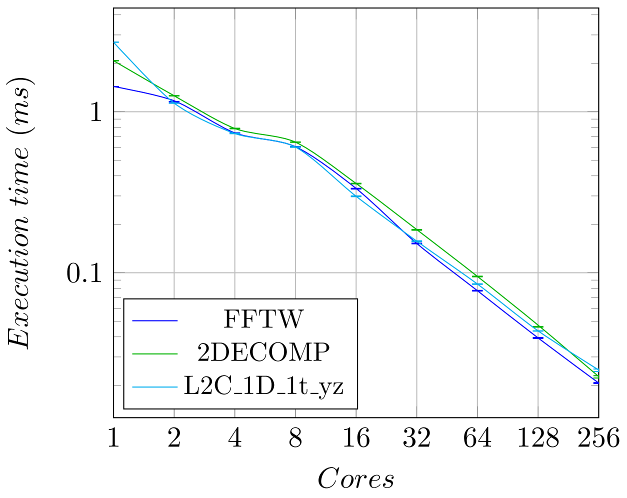
A second set of experiments focusing on scalability has been done on the supercomputer Curie (Section 5.1.4).

Algorithms and FFT Reference Implementation Figure 9 summarized the L^{2}C assemblies and reference FFT libraries that are used in experiments. All experiments involve complex-to-complex 3D FFTs. When it was supported, both 1D and 2D decomposition variants have been tested. For every 1D decomposition implementation, we have tested a version with a transpose at the end and a version without. We have not used L^{2}C assemblies with overlapping because their implementation is still ongoing.

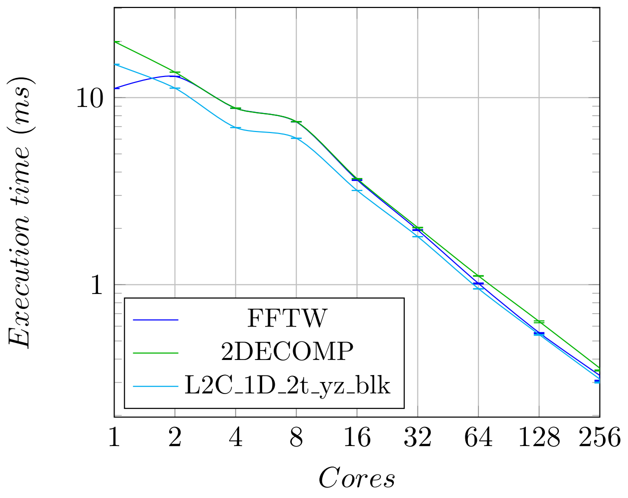
The FFT libraries used as reference are FFTW 3.3.4 and 2DECOMP 1.5. P3DFFT is not used because complex to complex 3D FFT are not yet implemented and this study focuses only on complex to complex computation. All libraries are configured to use a complex to complex 3D FFT without overlapping (as for L^{2}C assemblies) using FFTW sequential implementation and double precision floating point. All implementations use `FFTW_MEASURE` planning. The compiler used to compile all tests on Grid'5000 is `gcc` (version 4.7.2) and the implementation of MPI used is OpenMPI (version 1.8.1). On Curie, the Intel C++ Compiler (version 14.0.3) is used for the compilation and the MPI implementation is Bullxmpi (version 1.2.7.2) based on OpenMPI.



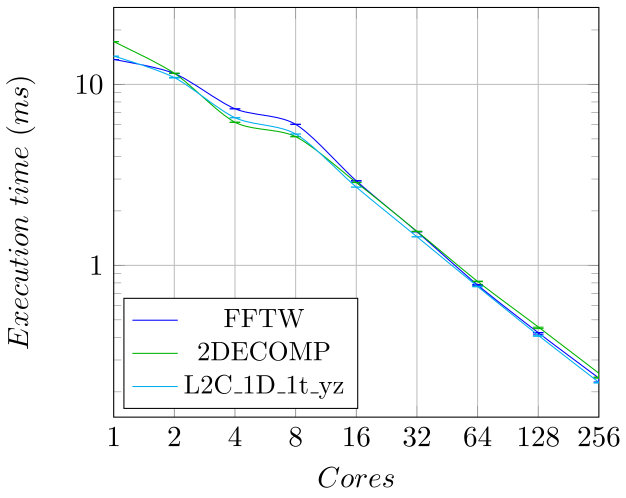
(a) Experiments for a 256^3 matrix size, and two transpositions.



(b) Experiments for a 256^3 matrix size, and one transpositions.



(c) Experiments for a 512^3 matrix size, and two transpositions.



(d) Experiments for a 512^3 matrix size, and one transpositions.

Figure 10: Execution time of complex to complex homogeneous 3D FFT on the Griffon cluster using 1D decomposition.

Experimental Evaluation Methodology Experiments have been done for homogeneous cases (Section 5.1.2), heterogeneous cases (Section 5.1.3), and scalability on homogeneous cases (Section 5.1.4).

Each experiment has been done 100 times and averaged. Error bars on plots correspond to the first and last quartile. They are almost imperceptible without zooming. All input matrices are cubic (256^3 or 512^3). The size of input matrices and the number of processes are a power of two.

In the heterogeneous case and using 1D decomposition, data are distributed along the Z axis and the block size on this axis depends on the node performance: on each node, the computing time needed to apply a 3D FFT (using all cores) is collected and then the block sizes are set in a inversely proportional way to the computation time. Using 2D decomposition, the same operation is applied along the Z axis and the Y axis remain the same as the homogeneous case in our experiments.

5.1.2 Homogeneous Test

Figure 10 presents the results for the Griffon cluster up to 256 cores for a 1D decomposition, for matrices of size 256^3 (Figure 10a and Figure 10b) and 512^3 (Figure 10c and Figure 10d) with and without an extra transposition.

Overall, performance results of L^{2}C, FFTW and DECOMP are similar. We note a few exceptions on Grid'5000:

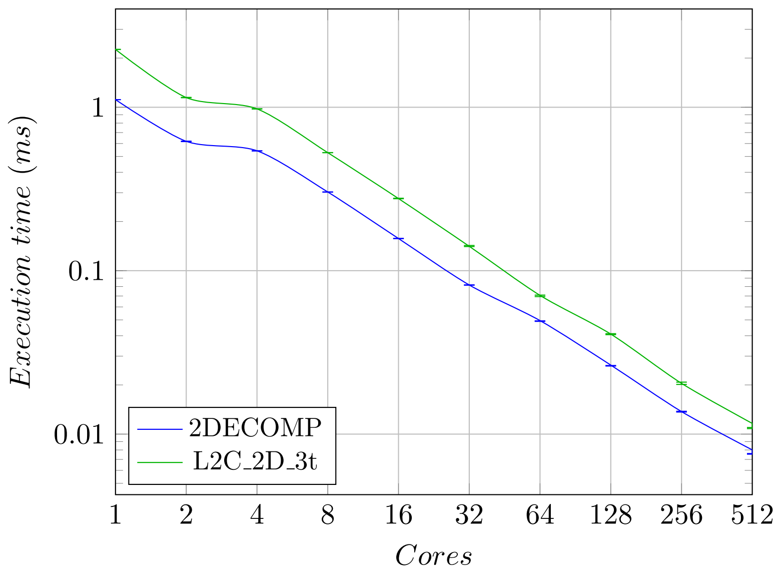


Figure 11: Execution time of 256^3 complex to complex homogeneous 3D FFT on the Graphene cluster using 2D decomposition and saving transpositions

- on 1 core (sequential) FFTW is consistently better;
- 2DECOMP has slightly worse performance than L^{2}C and FFTW for the 256^3 matrix;
- FFTW is 20% faster than both L^{2}C and 2DECOMP for 256 cores, the 256^3 matrix, and two transpositions.

The observed performance variability of the FFTW library is due to the selection of a fast algorithm during the planning phase. When the planner uses the `FFTW_MEASURE` mode, FFTW uses an heuristic to find a fast algorithm but does not necessarily find the fastest. This can be solved by using the `FFTW_EXHAUSTIVE` mode which computes an exhaustive exploration of all possible algorithms variations but it can take a large amount of time.

Figure 11 presents the results on the Graphene cluster up to 512 cores for a 2D decomposition (`L2C_2D_3t` and `2DECOMP_2D`), and for a matrix size of 512^3. Results show that 2DECOMP is always faster than the L^{2}C assembly from 1 core (102% faster) up to 512 cores (45% faster) mainly due to an extra[3] transposition done by `L2C_2D_3t`. Still, `L2C_2D_3t` scales up to 512 cores.

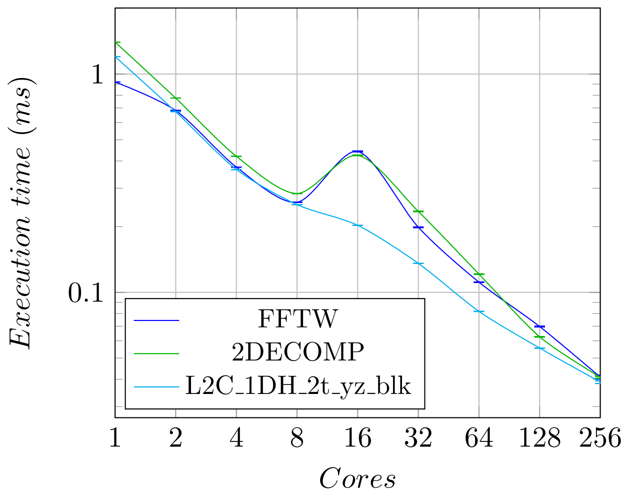
5.1.3 Heterogeneous Test

Figure 12 presents the results on the clusters Edel and Genepi up to 256 cores for 1D decomposition, a matrix size of 256^3, with and without extra transposition. Results up to 8 cores correspond to the homogeneous case on one Edel node. From 16 cores and up, half the cores are from Edel nodes and the other half from Genepi nodes. The Edel cluster is overall faster than the Genepi cluster.

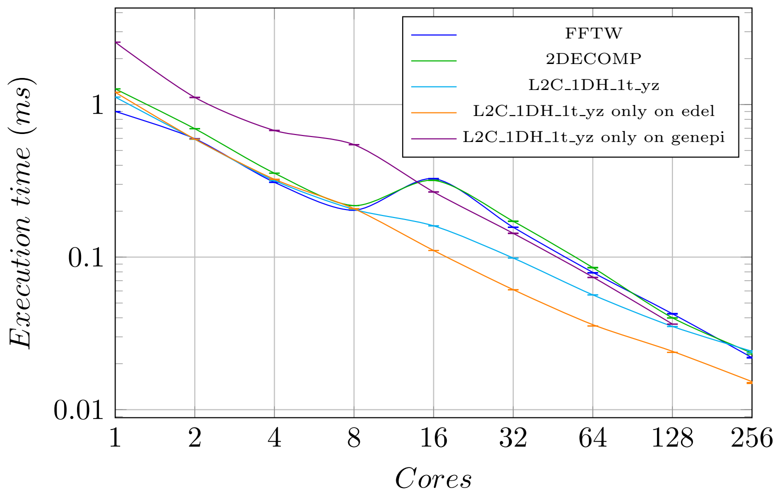
We observe that from 8 to 16 nodes 2DECOMP performance decreases. That is because 2DECOMP does not balance its load and is thus limited by the speed of the slowest cluster.

In the case of L^{2}C assemblies, Figure 12b shows that their performance is somewhere between the performance obtained only on Edel (the fast cluster) and the performance obtained only on Genepi (the slow cluster). It means the heterogeneous L^{2}C assembly successfully takes advantage of both clusters and is not limited by the speed of the slowest one. As the number of cores increases to 256, performance of heterogeneous L^{2}C assemblies gets closer to 2DECOMP and full-Genepi L^{2}C. That is because 1D decomposition is used here: the height of the slabs decreases down to 1 and rounding prevents an efficient load balancing. Full-Genepi and full-Edel performance are reported here but they have a similar behavior in term of scalability. This is especially because the scalability of the application is mainly limited by all-to-all

[3]It can optimized as explained in Section 4.4.



(a) with a 256^3 matrix and using two transpositions



(b) with a 256^3 matrix and using one transpositions

Figure 12: Execution time of complex to complex heterogeneous FFT on Edel and Genepi using 1D decomposition scheme

communications. All-to-all communications are as fast on the Edel cluster as on the Genepi cluster because the two clusters have the same interconnection network (*i.e.* InfiniBand 40G).

Overall, heterogeneous L^{2}C assembly is as fast or faster (up to 117% on 16 cores with extra transposition) than 2DECOMP because it supports heterogeneous data distribution.

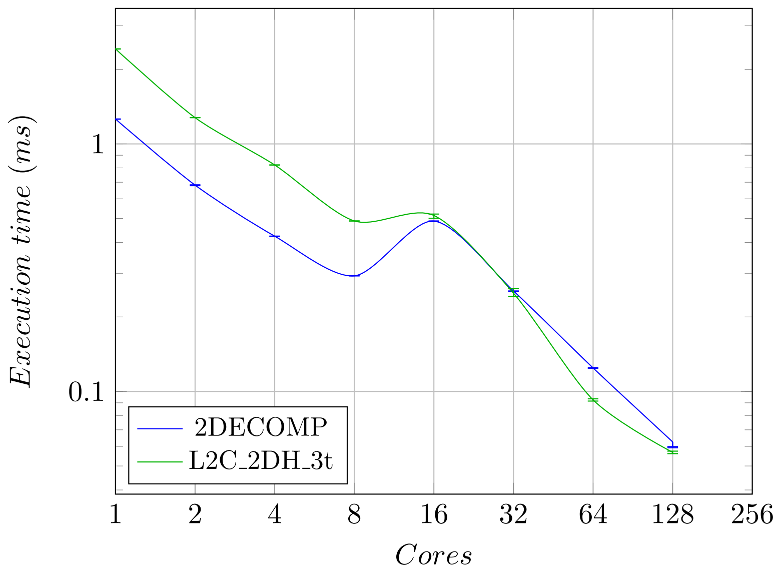


Figure 13: Execution time of 256^3 complex to complex heterogeneous FFT on Edel and Genepi using 2D decomposition scheme and saving one transposition

Figure 13 presents the results on Edel and Genepi from 1 to 128 cores, for 2D decomposition, matrix size of 256^3, without extra transpositions for 2DECOMP, and with only one extra transposition for `L2C_2DH_3t`. The performance of the heterogeneous `L2C_2DH_3t` assembly has a behavior similar to that of the 1D version but with two main differences. First, the `L2C_2DH_3t` assembly is little bit slower on 16 core than 8. This seems to be due to a bad data distribution between nodes. Then, the raw performance of the `L2C_2DH_3t` is lower than that of 2DECOMP on one node (Figure 11). The heterogeneity advantage it is not enough to overcome the lack of optimizations already mentioned in the homogeneous case. Still, on 64 and 128 cores `L2C_2DH_3t` is respectively 34% and 10% faster.

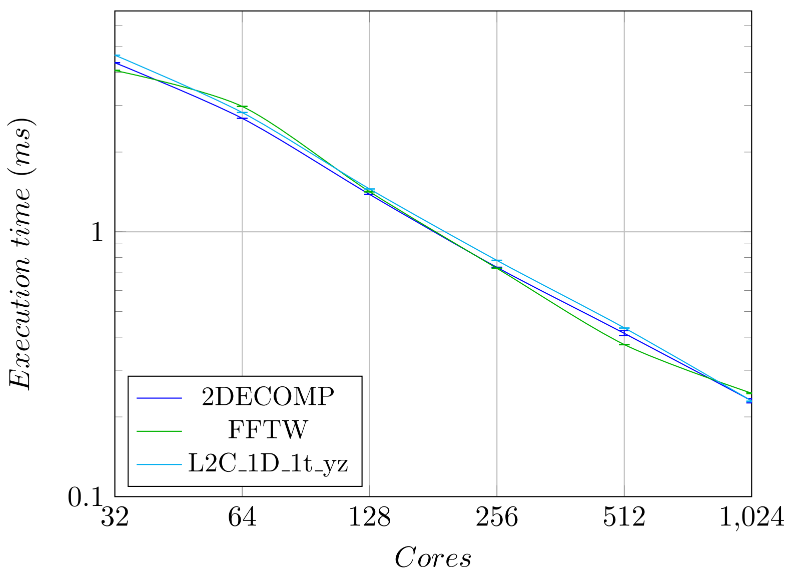


Figure 14: Execution time of 1024^3 complex to complex homogeneous FFT on Curie using 1D decomposition scheme and saving transpositions

5.1.4 Scalability

Figure 14 presents the results obtained with the thin[4] nodes of the Curie supercomputer from 32 to 1024 cores for a 1D decomposition, for matrices of size 1024^3 without an extra transposition. Performance results of L^2C, FFTW and DECOMP are similar than on Grid'5000 and L^2C has slightly worse performance than 2DECOMP.

Figure 15a presents the results for thin nodes of the Curie supercomputer from 256 to 8192 cores for a 2D decomposition, for matrices of size 1024^3 with limited extra transpositions. The performance of the `L2C_2D_3t` assembly has a behavior similar to that of Grid'5000 homogeneous experiments. So, L^2C scales well up to 8192 cores. Figure 15b presents the results of the same experiment but using a bigger matrix size (4096^3) from 2048 to 8192 cores. Performance results are similar: they confirm that the assembly scales using larger matrices.

Beyond the limit of 8192 cores, the current implementation of L^2C takes to much time to deploy the assembly on nodes and the amount of memory needed to perform the deployment become too large. This is mainly due to the assembly file size that increases with the number of core. We plan to evaluate more specifically performance bottleneck to optimize the implementation to deal with more cores. Main issues are to keep memory footprint low and deployment time nearly constant with respect of the number of cores.

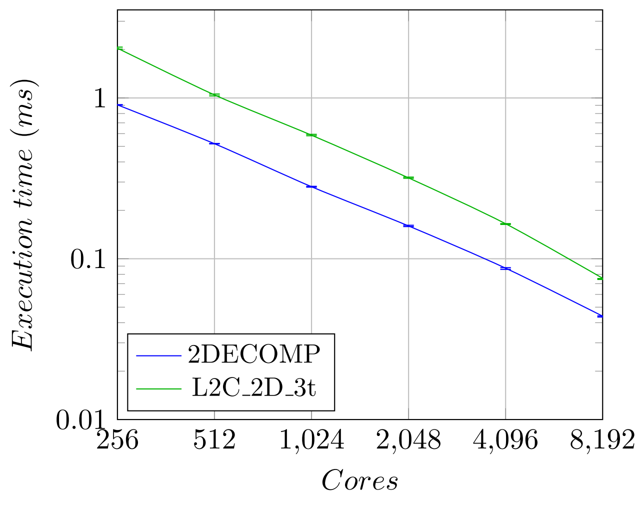
5.2 Adaptability Evaluation

This section evaluates the adaptability of L^2C component based approach compared to a selection of reference FFT3D libraries.

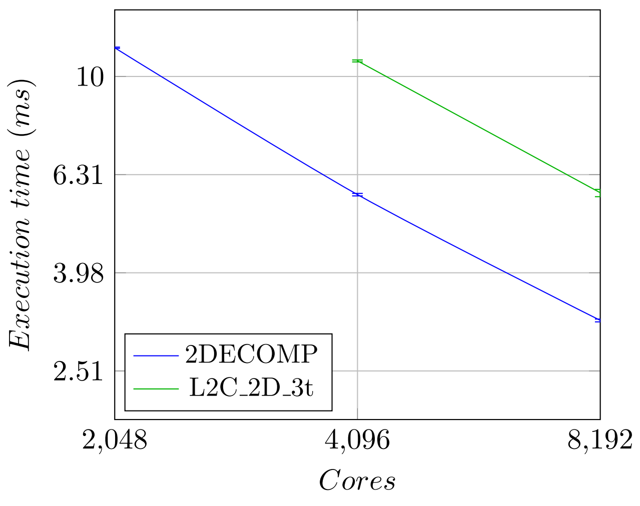
Optimization Comparison Table 16 shows a comparison of implemented or possible optimizations of existing libraries and L^2C assemblies. The assemblies can be adapted to perform all shown optimizations. To perform specific optimizations for Cray XT supercomputers, the assembly can be adapted to use specific transpositions components which either add padding to data and use a `MPI_Alltoallv` or perform a global exchange using the shared memory offered by these supercomputers. To perform a global exchange of data between nodes, FFTW uses a multiple transposition algorithm (with different complexities) and selects the fastest during the planning time.

Reuse Table 17 shows code reuse (in terms of number of lines of C++ code) between some of L^2C assemblies. Reuse is the amount of code that is reused from the assemblies list higher in the table. Overall,

[4]Intel height-core dual processor so 16 cores per node.



(a) with a 1024^3 matrix



(b) with a 4096^3 matrix

Figure 15: Execution time of complex to complex homogeneous FFT on Curie using 2D decomposition scheme and saving transpositions

	FFTW	P3DFFT	2DECOMP	Assembly
1D decomposition	Yes	Yes	Yes	Yes
2D decomposition	No	Yes	Yes	Yes
Load balancing (hetero. nodes)	Yes *(manually)*	No	No	Yes *(by adapting the assembly)*
Communication overlaping	No	Yes *(limited to fixed block sizes and direction)*	Yes	Implementation ongoing
Padding to avoid `MPI_Alltoallv`	No	Yes	Yes	Possible *(with new transposition components)*
Use shared memory of Cray XT	No	Yes	Yes	Possible *(with new transposition components)*
Global exchange method	Several MPI transposition algorithms	Based on MPI all-to-all exchange	Based on MPI all-to-all exchange	Multiple versions possible *(MPI all-to-all version implemented)*

Figure 16: Comparison of implemented/possible optimizations between existing libraries and the assembly

Version	C++ Lines of code	Reused code
`L2C_1D_2t_xz`	927	-
`L2C_1D_1t_yz`	929	77%
`L2C_1D_2t_yz`	929	100%
`L2C_1D_2t_yz_blk`	1035	69%
`L2C_1DH_1t_yz`	983	80%
`L2C_1DH_2t_yz_blk`	1097	72%
`L2C_2D_3t`	1067	87%
`L2C_2DH_3t`	1146	69%

Figure 17: Total number of lines for the various versions of the 3D FFT application.

our L^{2}C implementations are much smaller than 2DECOMP or P3DFFT (respectively 11570 and 8118 lines of FORTRAN code); that is also because 2DCOMP and P3DFFT implement more features.

Since our components are medium-grained and have simple interfaces (see Section 4), modifying an assembly for one PE is only a matter of changing a few parameters, connections and adding/removing instances. This process does not involve any modification in low-level code. It is done at architecture level and it is independent of possible changes made to the component's implementations.

With L^{2}C, assembly descriptions need to be rewritten for each specific hardware. As it is fastidious and error-prone, such descriptions should be automatically generated. This is currently done by a group of programs (*i.e.* scripts). But adaptation of these programs can be tedious and their maintainability remains an issue. A higher level component model that aims at automating assembly generation can increase the maintainability of assembly generation and ease development of new assembly. This is one of the purposes of HLCM [4]. But the HLCM implementation is in development phase: it is not yet ready to generate assemblies for a large number of cores.

5.3 Evaluation Summary

With respect to performance, experiments show that:

- L^{2}C assemblies scale up to 8192 cores;
- L^{2}C assemblies benefit from load balancing on heterogeneous architectures while 2DECOMP is limited by the slowest cluster;
- 1D decomposition is competitive with 2DECOMP and FFTW;
- 2D decomposition is slower (not fully optimized) but it scales and it benefits from load balancing.

These results are encouraging but the core count is still low compared to target architectures in the literature (*e.g.*, 65k cores for P3DFFT[18]). To remedy that, we are currently improving the L^{2}C deployment phase to handle larger experiments on supercomputers such Jade and Curie.

With respect to adaptation, that is the main goal of this work, components enable lightweight and specialized assemblies. Several optimizations from the literature have been implemented, taking advantages of code re-use, component replacement in assemblies, and component attribute tuning. Other optimizations require the implementation of new components. The specialization process allows to reuse most of the base components (69% to 100% reuse) without any modification.

6 Conclusions and Perspectives

To achieve adaptability of high performance computing applications on various hardware architectures, this paper has evaluated component-based implementations and specializations of classical algorithms for 3D FFT. 3D FFT algorithms have been modeled and then optimized using component models features (component replacement, component attributes tuning, and assemblies).

The experimental results obtained on Grid'5000 clusters and on the Curie supercomputer show that L^{2}C assemblies can be competitive with existing libraries in multiple cases using 1D decomposition scheme. It is consistent with previous results obtained on a simpler use case [6]. Using an HPC oriented component model does not add overhead while providing higher software engineering features. However, assemblies based on 2D decomposition still require more optimizations. Re-usability results show that components enables the writing of optimized applications by reusing parts of other versions.

Results are encouraging, but more work on L^{2}C and HCLM implementations is needed to allow building highly scalable component-based applications as well as assemblies using the 2D decomposition scheme. We plan to perform larger experiments on Curie and Jade supercomputers, to measure the scalability of assembly implementations and the limits of the approach. We are also working on assemblies that support communication/computation overlapping.

7 Acknowledgment

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement n^o PRACE-2IP 283 493. Some experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] M. Frigo and S.G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, February 2005.

[2] Jeffrey M Squyres and Andrew Lumsdaine. The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms. In *Component Models and Systems for Grid Applications*, pages 167–185. Springer, 2005.

[3] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition edition, 2002.

[4] Julien Bigot and Christian Pérez. High Performance Composition Operators in Component Models. In *High Performance Computing: From Grids and Clouds to Exascale*, volume 20 of *Advances in Parallel Computing*, pages 182 – 201. IOS Press, 2011.

[5] M. Bozga, M. Jaber, and J. Sifakis. Source-to-Source Architecture Transformation for Performance Optimization in BIP. *Industrial Informatics, IEEE Transactions on*, 6(4):708–718, Nov 2010.

[6] Julien Bigot, Zhengxiong Hou, Christian Pérez, and Vincent Pichon. A low level component model easing performance portability of HPC applications. *Computing*, November 2013.

[7] Sander Pronk, Szilárd Páll, Roland Schulz, Per Larsson, Pär Bjelkmar, Rossen Apostolov, Michael R. Shirts, Jeremy C. Smith, Peter M. Kasson, David van der Spoel, Berk Hess, and Erik Lindahl. GROMACS 4.5: a High-Throughput and Highly Parallel Open Source Molecular Simulation Toolkit. *Bioinformatics (Oxford, England)*, 29(7):845–854, April 2013.

[8] R. C. Le Bail. Use of Fast Fourier Transforms for Solving Partial Differential Equations in Physics. *J. Comput. Phys.*, 9(3):440–65, 1972.

[9] Daniel Guinier. The Multiplication of Very Large Integers Using the Discrete Fast Fourier Transform. *SIGSAC Rev.*, 9(3):26–27, June 1991.

[10] James Cooley and John Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[11] Anshu Arya. Optimization of FFT Communication on 3-D Torus and Mesh Supercomputer Networks. Master's thesis, University of Illinois, Illinois, 2013.

[12] Roland Schulz. 3D FFT with 2D decomposition. CS project report http://cmb.ornl.gov/Members/z8g/csproject-report.pdf, April 2008.

[13] Top500. Top 500 Supercomputer. http://www.top500.org/.

[14] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, and Derrick Weathersby. Efficient Algorithms for All-to-All Communications in Multi-Port Message-Passing Systems. In *SPAA*, pages 298–309, 1994.

[15] B. Prisacari, G. Rodriguez, C. Minkenberg, and T. Hoefler. Bandwidth-optimal All-to-all Exchanges in Fat Tree Networks. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, pages 139–148. ACM, Jun. 2013.

[16] Jeffrey M. Squyres and Andrew Lumsdaine. The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms. In Vladimir Getov and Thilo Kielmann, editors, *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, pages 167–185, St. Malo, France, July 2004. Springer.

[17] Rajeev Thakur and Rolf Rabenseifner. Optimization of Collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19:49–66, 2005.

[18] Dmitry Pekurovsky. P3DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions. *SIAM J. Scientific Computing*, 34(4), 2012.

[19] N. Li and S. Laizet. 2DECOMP&FFT – A highly scalable 2D decomposition library and FFT interface. In *Cray User Group 2010 conference*, Edinburgh, 2010.

[20] Krishna Chaitanya Kandalla, Hari Subramoni, Karen A. Tomko, Dmitry Pekurovsky, Sayantan Sur, and Dhabaleswar K. Panda. High-performance and scalable non-blocking all-to-all with collective offload on InfiniBand clusters: a study with parallel 3D FFT. *Computer Science - R&D*, 26(3-4):237–246, 2011.

[21] R. Agarwal and J. Cooley. New Algorithms for Digital Convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 25(5):392–410, 1977.

[22] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[23] Brian Gough. *GNU Scientific Library Reference Manual - Third Edition.* Network Theory Ltd., 3rd edition, 2009.

[24] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996. (An early version appeared in the *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, July 1995.).

[25] M. D. McIlroy. Mass-produced Software Components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[26] Juergen Boldt. The Common Object Request Broker: Architecture and Specification. July 1995.

[27] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. GCM: a grid extension to Fractal for autonomous distributed components. *Annales des Télécommunications*, 64(1-2):5–24, 2009.

[28] Nanbor Wang, Kirthika Parameswaran, Michael Kircher, and Douglas C. Schmidt. Applying Reflective Middleware Techniques to Optimize a QoS-Enabled CORBA Component Model Implementation. In *COMPSAC*, pages 492–499. IEEE Computer Society, 2000.

[29] Bernholdt D.E., Allan B.A., Armstrong R., Bertrand F., Chiu K., Dahlgren T.L., Damevski K., Ewasif W.R., Epperly T.G.W, Govindaraju M., Katz D.S., Kohl J.A., Krishnan M., Kumfert G., Larson J.W., Lefantzi S., Lewis M.J., Malony A.D., McInnes L.C., Nieplocha J., Norris B., Parker S.G., J. Shende Ray, T.L. S. Windus, and S Zhou. A Component Architecture for High Performance Scientific Computing. *International Journal of High Performance Computing Applications*, May 2006.

[30] Frédéric Desprez, Geoffrey Fox, Emmanuel Jeannot, Kate Keahey, Michael Kozuch, David Margery, Pierre Neyron, Lucas Nussbaum, Christian Pérez, Olivier Richard, Warren Smith, Gregor Von Laszewski, and Jens Vöckler. Supporting Experimental Computer Science. Rapport de recherche RR-8035, INRIA, July 2012.