Exploring the collaboration between FEniCSx and StarPU

Thomas Morin

Exploring the collaboration between FEniCSx and StarPU

Thomas Morin

14th December 2022

Version: First Draft

Exploring the collaboration between FEniCSx and StarPU

BY

Maelstrom team

Investigate cooperation between FEniCSx' DOLFINx computational back-end for the Finite Element Method and the StarPU task-based runtime system developed by STORM

14th December 2022

Thomas Morin

Exploring the collaboration between FEniCSx and StarPU

Internship report, 14th December 2022

Supervisors: Olivier Aumage
James D. Trotter

Maelstrom team

Bordeaux, Oslo

Thanks

Abstract

Motivated by the idea of having all the work chain parallelized for the Finite Element Method; we looked at the parallelization of the assembling phase on the FEM.

We have used the FEniCSx collection of tools for solving Partial Differential Equations; used by the Simula laboratory for the FEM part, and we try to use the StarPU runtime for parallelizing manycore architectures developed by the STORM team at INRIA Bordeaux

In this document, we explore different strategies for combining these tools.

Contents

List of Figures

1

Introduction

> *There is no wealth but life. Life, including all its powers of love, of joy, and of admiration.*
>
> **– John Ruskin –**

Today, modeling the world is based on Partial Differential Equations (PDEs). PDEs show us how a parameter influences a phenomenon. We find a lot of PDEs in many fields. Their goal is to model different aspects of the world. In particular, they focus on fields for modeling properties such as Physics, Financial sciences, Fluid mechanics, and Quantum mechanics.

A PDE is an equation for which the unknowns are multi-variable functions, and the relations on the equation are on the partial derivatives of the functions. They are expressed in a domain, called solving space. The PDEs are a generalization of the Ordinary Differential Equations (ODE). Often, they may have several solutions because, in contrast to ODEs, they have less restrictive boundary conditions. They are everywhere because they allow us to understand how different parameters evolve together like in complex systems. We can for example talk about the Heat Equation, which explains how heat is diffused in a system, or the Burger Equation, which is used for illustrating Adaptive Mesh Refinement codes. Another really important set of PDEs is Maxwell's equations, which is the foundation of classical electromagnetism. Among the most important questions concerning PDEs, we can talk about the existence and smoothness of solutions for the Navier-Stokes equations, which describe the motion of viscous fluid substances, and which is one of the Millennium Prize Problems defined in 2000.

Generally, it is admitted that there is no general fashion for solving PDEs. In fact, PDEs are divided into different types, each with its own particular solving methods.

Numerical methods are important tools for solving PDE. They include the Finite Element Method, the Finite Difference Method, and the Finite Volume Method. These methods use the same scheme of solving to approximate the solution. These methods approximate the solution by dividing the solving space into smaller pieces. Then, for each piece, the idea is to express the problem with fewer constraints which enables us to easily solve it. Finally, unifying all the smaller solutions gives an approximation of the solution on the global mesh. Unfortunately, these methods are resource-greedy.

However, for some years, we have seen supercomputers and manycores architectures appearing. With this type of machine, all these methods become more tractable and much more accurate. Thus, it allows us to use them and give us good approximations.

The problem is that computers become more and more complex and more and more difficult to program. To give an example, in June 2012, the first system in TOP500 was Sequoia - BlueGene/Q, with 17 petaflop/s with 1,572,864 cores. In June 2022, the first system in TOP500 is Frontier, with 8,730,112 cores and computing power of 1102 petaflop/s. Thus, in 10 years, the best computers are 65 times more powerful, with 5 times more cores and over 400 times more RAM (700 PB versus 1.6 PB). Similarly, Frontier has CPUs and GPUs, while Sequoia has only CPUs. It means that this new computer is heterogeneous, so balancing work is really difficult.

There are several methods to deal with this problem but a common one is the task programming and the Sequential Task Flow (STF) programming model. The idea of this method is to divide the work into small pieces, the tasks, and are executed by different process units. The idea of STF is that in addition to the work, pieces of data and access modes are also given to the tasks; and then the model ensures the coherency of data. An access mode associated with a piece of data is the way the piece of data is accessed. It specifies that a piece of data will be either read, written, or both read and written by a given task. There are also additional access modes, such as the commute one, but the most important are the three described before. For example, a task T_1 which is going to make a write access on a piece of data A has to be executed before any task submitted subsequently, which also makes access on A.

Finally, before giving our goal on this project, we are going to quickly present two different tools. The first is FeniCSx; a PDE solver which uses the Finite Element Method. The goal of FeniCSx is to describe our PDEs' problems in a high-level fashion (in Python or C++) and then all the resolution is made in the back-end with auto-generated code. The laboratory Simula of Oslo is one of the institutions involved in the FeniCS project. The second tool is StarPU; a runtime system that implements the STF model. This tool allows us to execute tasks in different types of process units only by using one of the predefined scheduling policies. It is developed at Inria & Inria centre at the University of Bordeaux, in the STORM team.

Finally, the main aim of this report is to present how it is possible to parallelize the assembly part of the Finite Element Method (FEM). For this, we would like to evaluate different strategies of parallelization. The aim is not to create a PDE solver and a runtime system, so we will use on one hand FeniCSx as a PDE solver,

and on the other hand StarPU as a runtime system. The challenge is to succeed to associate these two tools.

This project has been realized during an internship, with the tutorship of Olivier Aumage (INRIA, Bordeaux) and James D. Trotter (Simula, Oslo) in the context of the associated Team Maelstrom.

1.1 The Finite Element Method

In this section, we will present an overview of the Finite Element Method (FEM). This is only a summary, the reader interested in more detail is encouraged to read a numerical analysis lecture, like the one by Reddy [1], which is a good introduction to the FEM.

The FEM is a numerical analysis tool used to approximate in a given domain the solution of a partial differential equation. The idea of the FEM is to divide the domain into a collection of subdomains. For each subdomain, there is a set of equations that represents the contribution of the subdomain to the global domain. The idea is that if we solve all the equations, we will solve the initial problem. Moreover, each sub-equation is chosen to be simpler to solve (using linear systems) than the initial problem.

There are different phases for the FEM:

1. The first step is to make the domain discrete. This is the step described above.

2. Then, when there is a collection of subdomains, the goal is to create for each subdomain a linear system of equations, and so a sub-matrix per domain.

3. After, there is the assembly part. The aim is to create a global matrix from all the small local matrices corresponding to each subdomain problem. This matrix is sparse, which limits the amount of memory used; but it increases also the cost of potential load imbalance.

4. Finally, the last step is to solve this global matrix, thus resolving the initial problem.

1.2 The FEniCSx tools for the Finite Element Method

The FeniCSx tools are a collection of different tools for solving Partial Differential Equations. This is an open source computing platform. FEniCSx enables users to describe problems with high-level languages, like Python, and to solve them with efficient code. Also, this is an easy-to-use tool; but it allows powerful capacities for experienced and motivated programmers. FeniCSx is a new version of the FEniCS library [2], and the considered version of FEniCSx is the 0.5 version (August 2022).

To solve a problem with FEniCSx, there are several steps :

- Describe the problem with a high-level interface (in Python or C++). This is a mathematical step, with no need for a programming background; only a comprehension of the FEM.
- Compiling the described problem into low-level code. This step is managed entirely by FEniCSx. Otherwise, this is the most important part of FEniCSx: without it, it will be a classical solver.
- When the problem is compiled, you can now start to solve the problem. The first step is to assemble the matrix, to obtain the problem described in a numerical way.
- Finally, you can solve the problem by solving the assembled matrix.
- It is also possible to improve the code generated by FEniCSx, with a little programming background.

FEniCSx is not a unique tool, but it is composed of several tools :

- UFL for Unified Form Language [3]. This is a language defined for FEniCSx for the declaration of finite elements or variational forms.
- FFCx for FEniCS Form Compiler [4] is a compiler that takes UFL code and generates UFC output, a c++ interface of low-level functions. FFCx is defined for Python and C++.
- BASIX [5], which is a library with definitions for all Finite Elements. It is a domain-specific library, for evaluating functions, and having information on the topology of a mesh.

- DOLFINx, the C++ back-end which assembles the matrices and generates data for solving; and also for binding all the different parts of FEniCSx.

The solving part of the matrix generated by DOLFINx is managed by other libraries, like PETSc for example.

1.2.1 A benchmark code for having a concise code.

During this report, we may occasionally use a benchmark code instead of the real code of FENICSx. This is because FENICSx is a quite complex collection of tools, and so interactions between the different software and the different hardware could be tedious in a first approach.

To counter this, James D. Trotter has made a more concise code to isolate the assembly of the FEM; and we will use it sometimes. All the code has been tested, and the correctness of the results has been verified by hand in comparison to the results given by FENICSx.

Most of the time we have made, before making an implementation on FENICSx, a previous implementation on this benchmark code. If it looks promising, then it is included on FENICSx to have a real point of view.

1.3 A parallel programming model: the task programming model and the Sequential Task Flow (STF)

For using all the processing units of a computer, and for having parallelism, there are several ways.

The first one is to make several processes, and each process makes a section of the work. This fashion is not hard to do but could be really hard to optimize and make efficient because interactions between processes could be tedious.

Another way is to use loop parallelism. The idea is that most of the time, the iterations of a loop are independent; or with a bit of work, this is possible to make them independent. With this idea, the goal is to make blocks of iterations and

execute each block on a different processing unit. This could be easily used for parallelizing on GPUs but it could be tedious to use it when using several types of process units together, like CPUs and GPUs.

For this, we can use the task programming model. The idea is to divide computation into small pieces of work, called tasks. The notion of task is quite close to functions: the main difference is that when you submit a task, it will be executed later, but you do not know when. For example, the cilk programming language ([6]) is an efficient language to express some variants of the task programming model.

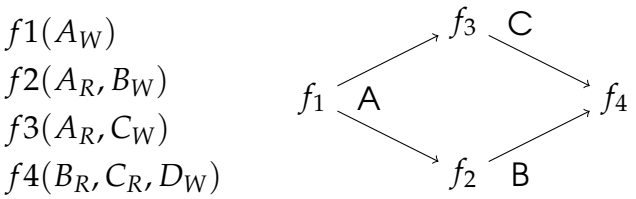
With the task programming model, your whole application will be defined by all the tasks and the interactions between them.

There are several types of interactions between tasks but we will restrict ourselves to talking about interactions with pieces of data. Thus, it is natural to talk about the Sequential Task Flow (STF) programming model, which is a way of using tasks. The idea of the STF is to define a task as a portion of code and also pieces of data. For each piece of data, you give memory access (ie if a piece of data will be accessed in a read mode, write mode, or read-write mode); and then dependencies between tasks will be computed according to the sequential flow of task submissions [7]:

- A task T which makes Read access on a piece of data D will be executed after all the tasks which make a Write Access on D, which have been submitted before T.
- A task T which makes Write access on a piece of data D will be executed after all the tasks which make a Write or a Read Access on D, which have been submitted before T.
- Consequently, submitting a set of tasks T_1;...; Tn which makes read access on a piece of data A ; between two tasks in write mode on A; allows the tasks T_1, ...T_n to be executed at the same moment, which generates parallelism.

There are other task models, more permissive, but the counterpart is the STF constraints allow the runtime to have more optimizations.

We can see on Figure 1.1a an example of code with the STF ; and in Figure



(a) Example of STF code **(b)** Corresponding STF DAG graph

Figure 1.1: Example of task graph generated from sequential source code.

1.1b the corresponding dependencies represented as a graph.

The name Sequential comes from the data dependencies: there is a guarantee of semantic equivalence of a compliant parallel STF execution with respect to the sequential execution of the tasks. The equivalence is only on the declared data dependencies. Also, the sequential order is determined by the order of the sequential submission of the tasks. It means that generally, you cannot have more than one processing unit which submits tasks; or you don't have any data dependency between the tasks submitted by different PUs.

The STF is a really powerful way to express parallelism because this is a contract with a runtime system (submit-and-forget parallelism). You do not have to manage by hand all the coherency of data if the dependencies are correctly expressed. Also, another advantage is, generally, the algorithms of the sequential application and the STF application are identical, or really similar.

Finally, to represent an application with STF, it is common to use Directed Acyclic Graph. On this type of graph, a node is a task, and an edge from a task A to a task B expresses that the task B depends on A. We can see in Figure 1.2 an example of a Cholesky factorization expressed in an STF way; and the corresponding DAG.

1.4 StarPU, a runtime system based on the STF

We have seen that a good way of making an application parallel is using the STF. Moreover, we have also seen that computers become more and more parallel. Nowadays, most computers have one or more accelerators, like GPUs or FPGAs; and commonly, they are divided into several memory spaces (NUMA nodes). All those elements mean that it is really complicated to program computers. For example, it could be tedious, if you have several different STF tasks,

```
for (k = 0; k < NT; k++) do
  POTRF (A[k][k]);
  for (m = k+1; m < NT; m++) do
    TRSM (A[k][k], A[m][k]);
  end for
  for (n = k+1; n < NT; n++) do
    SYRK (A[n][k], A[n][n]);
    for (m = n+1; m < NT; m++) do
      GEMM (A[m][k], A[n][k], A[m][n]);
    end for
  end for
end for
```

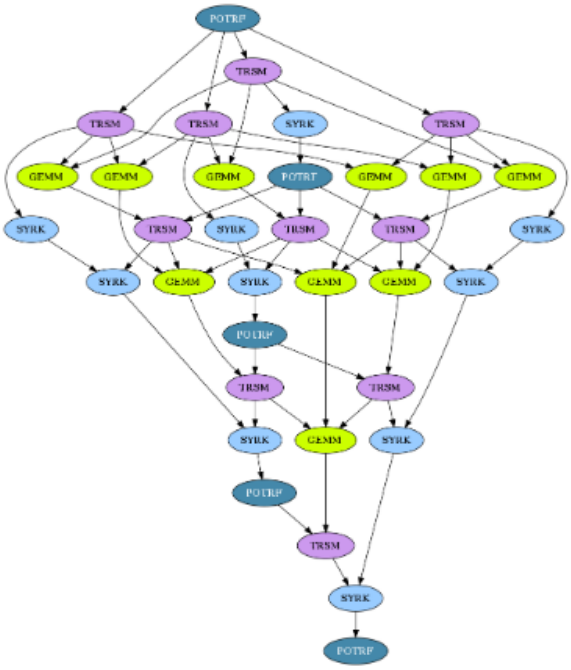


Figure 1.2: Tiled Cholesky factorization and the resulting task graph [8].

to know which processing unit should execute which task to optimize either the execution time or the energy consumption. To deal with this type of problem, we can talk about runtime systems, like StarPU. StarPU ([9]) is an implementation of the STF for heterogeneous multicore architectures. It means that it can efficiently use CPUs, GPUs, and also FPGAs. We call each process unit, without distinction, a worker.

Moreover, StarPU is an efficient runtime system. It means that more than giving a unified view of a heterogeneous system, it offers different schedulers for mapping and executing tasks onto a heterogeneous machine. Finally, with StarPU, the programmer does not have to deal with low-level problems, such as memory transfers, which are included on StarPU; or the architecture of the machine. For the case of changing the configuration of the machine, for example by only considering a part of the CPUs, StarPU will take it into account and try to execute the program in the most efficient way, without changing any line of code.

Also, using a new GPU or FPGA is really easy: the only requirement is to write the new version of the functions for the new units; without managing the data transfers, initialization, or other low-level details.

Finally, an important thing about StarPU is the scheduling engine. This is im-

portant because the goal of StarPU is to *"Make hardware-dependent decisions on behalf of the programmer"*. For this, there are different policies already programmed; and it is easy to create another policy. These policies came from a theoretical algorithmic corpus.

The first type of policy is the reactive one. The idea is to make work stealing, so attribute to each worker a queue of tasks; and when a worker has ended its queue, it will steal a task from another worker. There are two main policies, the first is **WS** (Work-Stealing), described as above; and the second is **LWS** (Locality Work-Stealing), which is a variant of WS, and the idea is a worker is going to steal tasks from neighbor workers.

The second type of policy is the anticipative one. The idea is to plan where this is the best choice to put a task. To do this, StarPU has performance models. This is a registration of the execution of the task according to the parameters, which is possible because a task in StarPU is not preemptible. Then, when submitting a task, StarPU will predict where the execution will be the best to minimize the execution time. There are several policies of this type. The most common is **DM**. It schedules tasks where their termination time will be minimal. A variant is **DMDA**, which takes also into account the data transfer times; and **DMDAR**; which takes into account the disponibility of buffers on the target worker.

1.5 Research Aim

We have seen that there are tools for approximate solutions of PDEs by the Finite Element Method, like FEniCSx. We have also seen that the FEM is divided into two major computation phases, the assembling phase, and the solving phase. The solving phase has been already studied; and there are some answers for parallelizing this phase. For FEniCSx we use the toolbox PETSC; which is already parallel. However, parallelization strategies for the assembling phase have not been studied a lot despite it could be really costly in computational time. The aim of this report is to explore different strategies for parallelizing this phase with the help of the runtime StarPU.

1.6 Difficulties and race conditions on the assembly part

Basically, the assembly part is a loop on all the entities of the mesh; and for each entity of the mesh, we have a global matrix in which we "put" the local contribution.

In the original algorithm, called cellwise, the idea is to make a loop on all the cells of the mesh, and then, for each cell, make a loop on all the vertices of the cell (called degrees of freedom).

Then for each degree of freedom, the idea is to do three different phases :

1. First, we have to recover the geometry of the cells, to know how and where the degree is going to contribute. We call this phase gather.

2. Second, we have to compute the contribution of the degree of freedom. We call this phase compute.

3. Finally, we need to add the local contribution to the global matrix. We call this phase scatter.

There are some important points to emphasize. The first point is that the first and the second phases induct no race conditions. It means that they can be performed in parallel without any problem: they only make read accesses on global memory and write only to local memory. The other important thing is each degree of freedom is going to write on a different (unpredictable in the general case) place on the matrix during the scatter phase. Moreover, it is possible for two different degrees of freedom to write at the same place during this phase. It means that there is a race condition in the scatter phase. So, we need to find solutions to make the assembly correct, and this is the most important part of this work. The difficulty is to make the scatter correctly because the other phases are not problematic in comparison.

1.7 Research Guidelines

Some ideas have been suggested for leading the project:

1. The atomic solution. The idea is basically to make the addition of the local contribution on the global matrix atomically, and so even if the addition is

more expensive, the operations ensure that they will be correct.

2. The buffer solution or "trade memory space for more parallelism". This is another way of working which could be interesting. The idea is to make additions using different matrices and after the end, make the addition of the buffers.

3. Using a different algorithm, the rowwise algorithm. Instead of iterating on all the cells and computing the degrees of freedom; the idea is to iterate on all the degrees of freedom, and for each one computing the cells and making the addition only for the part of the cell which has an influence on the degree of freedom. With this, you avoid the race condition because you consider sequentially all the modifiers of each degree of freedom.

4. The coloring solution. The idea is to divide the mesh into blocks and color the blocks. It means that two blocks have the same color if they don't have any common degree of freedom, and so they could be performed in parallel.

1.8 Dissertation Outline

This dissertation is presented as...

- Chapter 1: short brief of the subject; presentation of the different things to know, of the problem and introduction to the different solutions
- Chapter 2: explanation of the atomic solution, evaluation.
- Chapter 3: explanation of the buffer solution, why this is not a good solution.
- Chapter 4: explanation of the rowwise algorithm, a surprising problem.
- Chapter 5: explanation of the coloring solution.

1.9 Experimental environment

In this section, we will describe the environment used for our experiments.

For making experiments, we used two different platforms.

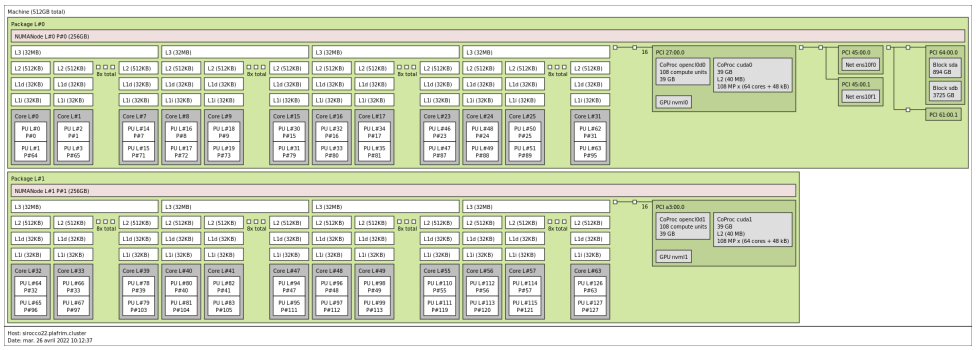


Figure 1.3: Topology of the machines sirocco23 and sirocco24.

The first is PlaFRIM. This is a computing platform used for experiments in Bordeaux. It is composed of several partitions. Each partition has its particularities, in terms of architecture. We have used the sirocco partition; which is composed of accelerated nodes; and the bora partition; which is composed of standard nodes.

On the sirocco partition, we have used the machine sirocco23 and sirocco24; with the same configuration :

- For the CPUs : 2x 32-core AMD Zen3 EPYC 7513 @ 2.60 GHz.
- For the GPUs : 2 NVIDIA A100 (40GB).
- For the memory : 512GB (10.6GB/core) @ 3200 MT/s.

We can see in Figure 1.3 the topology of this corresponding machine.

On the bora partition, we have used different machines; all of them have the same configuration :

- For the CPUs: 2x 18-core Cascade Lake Intel Xeon Skylake Gold 6240 @ 2.6 GHz.
- No GPUs.
- For the memory : 192 GB (5.3 GB/core) @ 2933 MT/s.

We can see in Figure 1.4 the topology of this corresponding machine.

The second platform we used is the Ex3 platform; a computing platform used at Simula (Oslo). There are also several partitions in this platform. The one we

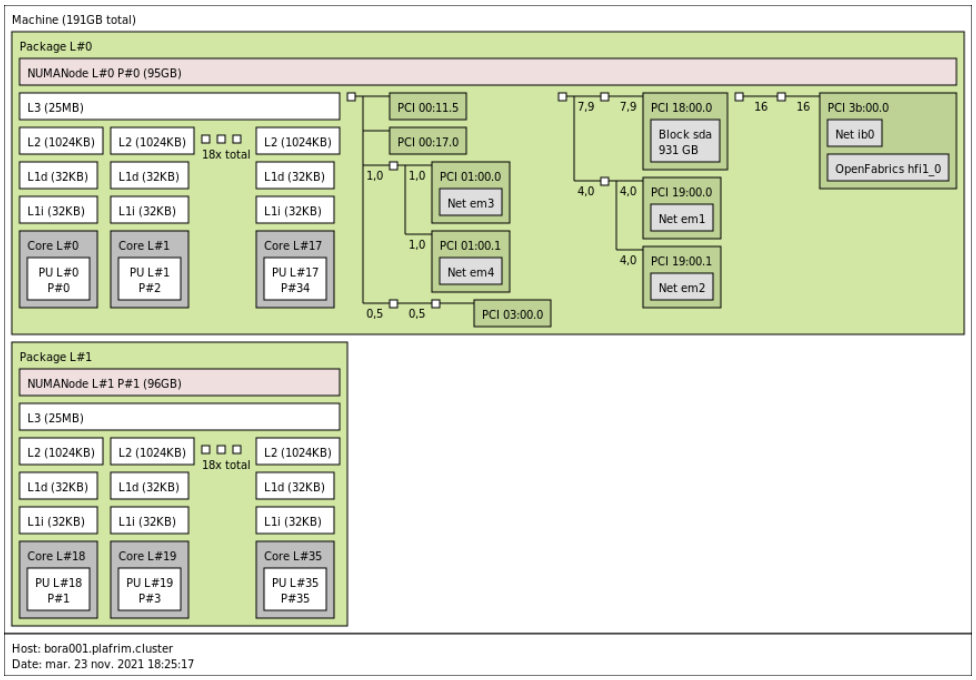


Figure 1.4: Topology of the machines of partition bora.

used is called the xeongold16q partition.

Its configuration is :

- For the CPUs: 2x 16-core Intel Xeon Gold 6130 @ 2.1 GHz.
- No GPUs.
- For the memory: 377 GB (11.8 GB/core).

We can see in Figure 1.5 the topology corresponding to this machine.

In brief, we have used these machines for all our experiments. Before each experience, we will give the name of the computer; and a quick brief about the type of machine. Finally, we have made all the experiments with multiple runs, but we will give the exact number at each experiment because it can differ according to the chapters.

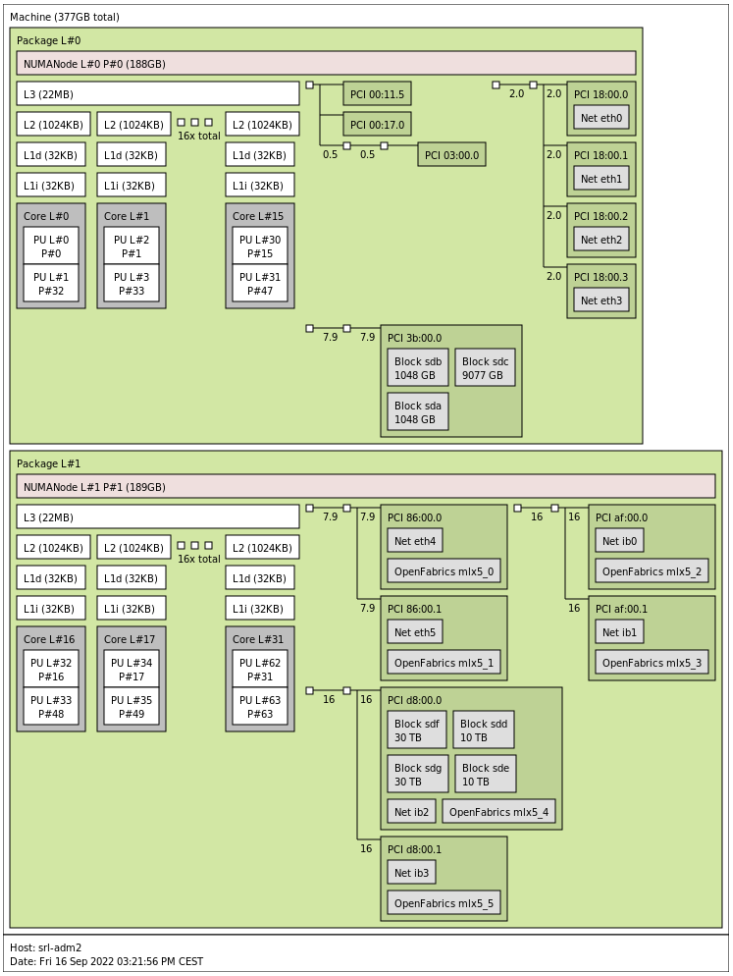


Figure 1.5: Topology of the machines of partition xeongold16q.

2

Parallelize the assembly part of the FEM by using the atomic solution

> *A genius is the man who can do the average thing when everyone else around him is losing his mind.*
>
> **– Napoleon –**

In this chapter, we will present the first solution to parallelize the assembling phase of the FEM.

2.1 Explanation of the atomic solution

The first solution is to use atomic operations to avoid racing conditions in the scatter phase. This is quite logical because our problem involves only additions. This solution is possible because in our case, the order of the addition is not major. It might be important because of the use of unassociative floating point arithmetic (for d_1, d_2, d_3 doubles, $(d_1 + d_2) + d_3 \neq d_1 + (d_2 + d_3)$) ; but this is outside the scope of the internship. Furthermore, the difference between these two additions is smaller than the accuracy generally expected for FEM. Thus, each addition is independent of the others, and so the matrix is, before the end of all phases, used as a big buffer.

With this, the natural idea that comes is rather than make simple additions to the matrix, it is possible to make atomic additions. It ensures that the value at the end will be correct, but it does not ensure that before the end, there is a global coherency on the matrix (i.e. before the end of the computation, we cannot say which cells have been considered or not). Cannot say which cells have been considered or not is not a problem, because the matrix is, before the end, only a buffer: we are not looking into it before the end.

The idea of using StarPU with this is quite simple: the first step is to divide the mesh into slices, and then push one task per slice. A slice is basically a set of cells. The division we choose for the moment is to attribute at each cell an index and give the first indexes to the first slice, then the second, ... For future work, it could be a good idea to use a mesh partitioner to equilibrate the work in the slices.

So, when a task is executed, all the assembly phases for the given slice are made at the same time. This is the simplest way to parallelize the assembly part, and when the scatter phase is done, instead of making the addition in the normal way, the addition is made by using std::atomic of C++.

2.2 Implementation of the atomic solution

Here, an implementation of atomic add on double in C++, because this is not native.

Listing 2.1: Implementation of atomic add on double in C++

```
static inline void atomic_addD(std::atomic<double> &d,
                               double d2) {
    double old = d.load(std::memory_order_consume);
    double desired = old + d2;
    while (!d.compare_exchange_weak(old, desired,
                                    std::memory_order_release,
                                    std::memory_order_consume)) {
                                    desired = old + d2;
    }
}
```

For the element of implementation, we also have to say that we can make two different versions of the code. The first version is the simpler one. This one uses direct pointers on global memory, without using the StarPU memory manager. This should be used only when memory is coherent. This is a prior version for testing.

The other version uses the StarPU memory handles to manage the memory accesses. This one is slower than the first because of the overhead of the memory management by StarPU, but this is an important version of the code because it will facilitate a heterogeneous parallelization or a port to distributed computing.

2.3 Prior remarks

This really simple solution should be sufficient if there is a good separation between the degrees of freedom: the atomic addition becomes slower if there are a lot of additions to the variable. If not, then it is approximately the same thing as making a normal addition. So, to have good performances, a good idea should be to make a good partitioning of the mesh.

Another remark is that we have to talk about the number of slices. If there are too many slices, tasks are small, and there is an overhead of using StarPU which is an important problem. If there are not enough slices, there is no overhead, but there is not enough parallelism, and so starvation.

Also, there is a difference between the code without atomic and with atomic: the global matrix on the normal code is a matrix of double; while on the code with atomic we need a matrix of *std::atomic<double>*. So, to deal with this, we need to create a temporary buffer of std::atomic variables for the atomic version; and then, in the end, make a copy of the atomic matrix on the normal global matrix [1]. It implies a little time overhead on it, because of the creation and the copy; and also a worse memory footprint. This is not a lot, but it can be something to know if you have several processes and a big matrix.

Finally, we have to be aware that for the moment, the code is not complete: we can only assemble normal vector cells. It means that to have a complete parallelization, we also need to implement changes in the other functions. The idea is that this is quite the same thing to parallelize, so what works for vectors will work for other types of assembling.

2.4 Testing the solution

We have implemented this solution on DolfinX with StarPU, and we would like to evaluate it. First, we have to discuss the number of slices. We have chosen a number of slices equal to five times the number of cores because it allows an important amount of tasks while avoiding generating too small tasks for StarPU.

Moreover, we would like to evaluate the overhead of using an atomic operation. For this, we have implemented the same algorithm; but instead of making the addition atomically, we make it in the usual way. It means that the computation result is not correct; this is just here to have some overhead measurement.

This warning made, we can present some results. We will do two different computations, one called hyper-elasticity and the other called "poisson". Those are demos benchmarks of assembly. For the hyper-elasticity example, we use a mesh of size 100x100x100. For the Poisson example, we use a mesh of size 4096x2048. We will present results for several versions: the sequential version:

[1] To avoid this copy, one can try to make a cast. This is, in the general case, a bad idea because it is possible to have alignment problems when using normal type as atomic; and also compilation problems. To avoid compilation problems, it is possible to use the third shape of the operator "new" (see here for more detail. To have a bigger explanation on using atomic operations on non-atomic data, you can see this discussion.

- The version with memory managed without StarPU, so with pointers on memory.
- The version with memory managed by StarPU, so with only the use of memory handles of StarPU.
- The incorrect version, so without the use of atomic operations. As said earlier, this is only a version for having a baseline.

For the parallel versions, we try on 1-2-4-8-16 and 32 cores.

The machine used is the xeongold16q partition of Ex3 (Simula); which is a non-accelerated node, so with only CPUs. We make 5 measurements for each try, and we make the mean of the execution times.

We compare also what we call "scalability", which is computed as $\frac{Execution\ time}{Execution\ time\ (1\ cpu)}$. This is how a version can speed up according to the number of used CPUs.

	hyper-elasticity		poisson	
	Ex. time (s)	scala.	Ex. time (s)	scala.
Sequential	2.426	1	2.344	1
With pointer (1 core)	3.514	1	3.36	1
With pointer (2 cores)	1.643	2.139	1.742	1.93
With pointer (4 cores)	0.914	3.85	0.955	3.51
With pointer (8 cores)	0.441	7.97	0.45	7.47
With pointer (16 cores)	0.28	12.56	0.273	12.31
With pointer (32 cores)	0.191	18.41	0.173	19.42
Without pointer (1 core)	3.549	1	3.509	1
Without pointer (2 cores)	1.916	1.85	1.854	1.89
Without pointer (4 cores)	0.873	4.06	0.880	3.98
Without pointer (8 cores)	0.497	7.14	0.519	6.76
Without pointer (16 cores)	0.257	11.95	0.273	12.85
Without pointer (32 cores)	0.169	21	0.154	22.78
Incorrect (1 core)	3.232	1	3.599	1
Incorrect (2 cores)	1.752	1.84	1.729	2.08
Incorrect (4 cores)	0.826	3.91	0.857	4.2
Incorrect (8 cores)	0.446	7.25	0.476	7.56
Incorrect (16 cores)	0.238	13.58	0.248	14.51
Incorrect (32 cores)	0.153	21.12	0.146	24.65

We can see several things. First, there is a big overhead caused by StarPU for this case. This is due to the important amount of interactions with other libraries. A thing we can say, but which is not given in the experiment is that making the sequential (initial) assembling but with StarPU enabled (it means no task is given to StarPU, but it is enabled) leads to an overhead equivalent at the overhead seen for making the assembly with StarPU on one core. So, initializing StarPU causes an overhead, because of the finding of the architecture, the allocation of some buffers, ... Thus, the seen overhead is due to StarPU itself and not from the algorithm.

Another thing we can see is that there is approximately no overhead in managing data with StarPU (the Without Pointer version) than without StarPU (the With Pointer version). Instead, it is a little more efficient to use StarPU for managing data. An explanation should be when StarPU manages data, it knows

the pieces of data used by a task, so it optimizes the execution of tasks to try to put two tasks with common data on the same core. As a result, we will now use StarPU to manage data, because of all the advantages of this.

Another point is that we do not have perfect scalability, namely the execution time with x CPUs is not equal to the execution time with 1 CPU divided by x. We can see on the column the division of the time when using one CPU by the time when using the current number of CPUs. What we see in the atomic cases is a dramatic decrease of scalability when we use more than 8 cores. The other point is that using atomic limits the scalability because the incorrect computation is a little faster than the atomic one, but the incorrect version is just a bit more scalable than the atomic one. Thus, probably the scalability is more a problem of memory accesses; or work repartition than a problem of using atomic operations: if the problem was totally from using atomic operations, we should see for the incorrect version perfect scalability, which is not the case.

All these experiments give us the idea of testing additional ideas to find a good strategy.

A point is that using atomic doubles is a good way to easily avoid race conditions for assembling a matrix; but this leads us to imperfect scalability, due to the structure of the atomics and also to bad memory accesses.

2.5 Chapter Summary

In this chapter, we have seen a first solution for parallelizing the assembly phase of the FEM, by using atomic operations for exporting the contribution of each cell.

We have seen it is a correct and easy-to-do solution, with a bit of parallelism. Also, using StarPU to manage data is an efficient way to optimize the execution of the tasks. We have also seen that it was not a perfectly scalable solution; due to the atomic operations, which are not really good when there are a lot of cores; but also due to memory accesses.

Effectively, the atomic operations do not explain all the problems because some experiments without atomic operations are also not perfectly scalable.

All these elements give us the desire of testing other solutions which use normal operations.

3

Why the buffer solution is not a good solution for parallelizing the assembly part

> *Hell is other people.*
>
> **– Jean-Paul Sartre –**

3.1 How to use buffers to limit the race conditions

What we have seen in the previous chapter is that atomic operations could be efficient, but there are problems when we use them. First, we have overhead due to the use of atomic operations instead of normal operations. It is known this type of operation is not scalable and implies overhead when there are a lot of cores trying to do the same operation. Second, we have an overhead due to the bad mapping of the matrix on the memory, because there are lots of cores that access the same variable.

A way to counteract this problem is to use different buffers for each worker. The idea is to create one matrix buffer per worker instead of having one global matrix, and each worker is going to add to this buffer. Therefore, the used buffer is not task-dependent but it is worker-dependent, according to the task executor. Then, at the end of the computation, we make an addition of all the buffers, called "reduction" to obtain the final global matrix.

This solution is allowed in our case, because, like for the atomic solution, we do not need any coherency of the matrix before the final reduction, and so before the addition of the buffers.

Like for the atomic solution, the idea is to divide the mesh into slices. However, we do not just push one big task for all the work, but we divide the work into the three given phases (gather, compute, scatter). The idea with this is that maybe StarPU will try to only put the scatter tasks on a few processing units, which would result in an economy of the number of buffers. The other point which ensures the correctness of the computation is that a StarPU task is not preemptible. It means that if a worker starts a task, before executing another task, it will finish the current task. This is an important point because if it was not the case, it could be possible that the addition on the buffer is not finished before executing another task, which is going to corrupt the memory.

3.2 Implementation of the solution

To implement this solution, we have used the buffers of StarPU and the StarPU REDUX functionality. It simplifies a lot the code and avoids managing the buffers by hand. The idea is to register a buffer on StarPU in a classical way, and then call the function starpu_data_set_reduction_methods to explain how the

reduction has to be made. Then, each time a task is using this piece of data in a STARPU_REDUX mode it takes a temporary buffer. This buffer is created the first time this piece of data is used in the given core. If a task uses this piece of data on a STARPU_REDUX mode on a core that has already allocated a temporary buffer, it will reuse this buffer. Finally, when the handle is unregistered, StarPU will reverse all the allocated buffers to the beginning buffer.

With this solution, it is possible, with a good scheduler, to prioritize the execution of this type of task on some cores and not all of them.

3.3 Testing the solution, and problems of this solution

This strategy has not been implemented on DOLFINx directly.

We can see in Figure 3.1 a trace of the execution on the benchmark code. We have tested to assemble this on a first order operation, for a unit mesh of size 100 x 100 x 200, on the computer sirocco24 of PlaFRIM (an accelerated node), and we use 32 CPUs. In this case, we only do one execution to analyze the trace; but we have seen this type of execution in different runs.

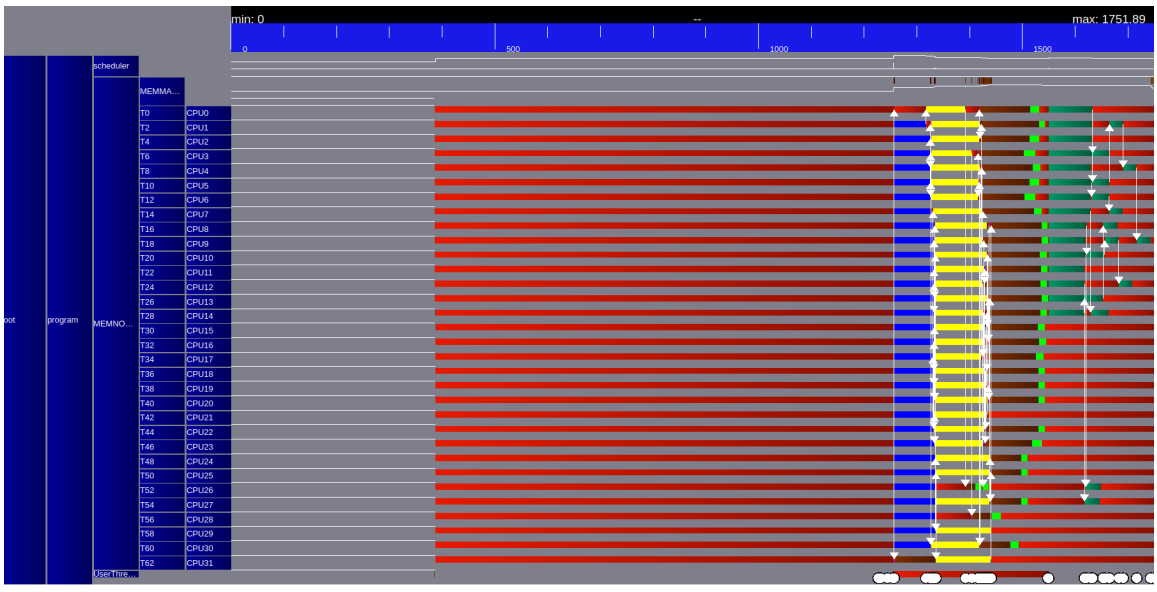


Figure 3.1: Trace of problematic buffered operations on the assembly benchmark code.

We can see some problems :

- The first major problem is the reduction: we see on the trace it takes a long time to reduce all the buffers on a single one: these are the last green

tasks on the trace. We see that it takes an important time, and a problematic point is this time depends on the number of workers used because we need one buffer per worker. The reduction is made in a $O(log(n\_workers) \cdot n\_elements)$ time. A remark is that this is probably useless to do so many buffers because most of the time, the tasks do not share any degree of freedom. There is probably an improvement to do by trying to color the mesh, or by only copying a portion of the matrix and not the whole matrix. Another remark is that with a smart scheduler, it is possible to not put a green task on each core; but only on some of them. With this, it limits the number of used buffers, and so limits the reduction time. It should be a good idea because the scatter tasks are short in comparison to the reduction tasks.

- Another problem we can see is due to the allocation of the buffers. First, the buffers could be big. On the given trace, each buffer has a size of several hundred MegaBytes. It has two consequences:
  - It takes lots of time to allocate all the buffers because there is one allocation by core so a lot of allocation. It implies a big memory contention; so long time to allocate all. This phase is the brown one on the trace.
  - The second consequence is problematic: if we have a lot of buffers, it means that we have a lot of memory allocated; and so we need an important amount of memory. The memory footprint is a big problem, and it is linear according to the number of workers. For the given example, the memory footprint is close to $300 \cdot 64 = 19200$ MB. As for the previous problem, being smarter in the scheduling of tasks allows better flexibility.
- The final problem is really different. In fact, this strategy will solve the problem in the CPU case; but if you use this strategy on the GPU case, this will clearly fail. The point is the buffer solution assumes that the worker which uses a buffer will not write on the buffer in parallel, and each write access will be sequential. For a GPU this is not the case: the idea is to submit a cell by GPU thread; so two cells will be treated in parallel; and if two cells share a degree of freedom and are computed at the same moment, it will result in a computation error. Thus, this solution is clearly unusable in the GPU

case and we will have to use another algorithm, or try to use the atomic solution.

On the other hand, we see that, only in terms of parallelism, and without consideration of the overhead, this is an efficient solution: executing 10 times the buffered version (without reduction between each phase) is a bit faster than executing 10 times the atomic version. This point shows us that this is not only a bad idea; because it allows faster parallelism, but it is not a scalable version.

With all of these points, we see that using buffers is not a scalable solution so we cannot make it in practice on DOLFINx, but with some smart schedulers it could be an efficient solution, or it could be a way to improve the parallelism of another solution.

3.4 Chapter Summary

We have seen in this chapter how we can use buffers for making assembly parallel. We have seen that it was a solution correct; but it presents some problems, because of the memory footprint, and the time of allocation and reduction of the buffers. One limiting factor is this solution could only be used for a CPU version; and not when using GPUs with many threads. The important point is that it could be an improvement of another solution, by using some buffers and not one by core.

We will now see how using another algorithm than the classical one could be a good way of doing it.

4

Using a different algorithm, designed for massive parallelism

> *I have a dream that my four little children will one day live in a nation where they will not be judged by the color of their skin but by the content of their character.*
>
> **– Martin Luther King –**

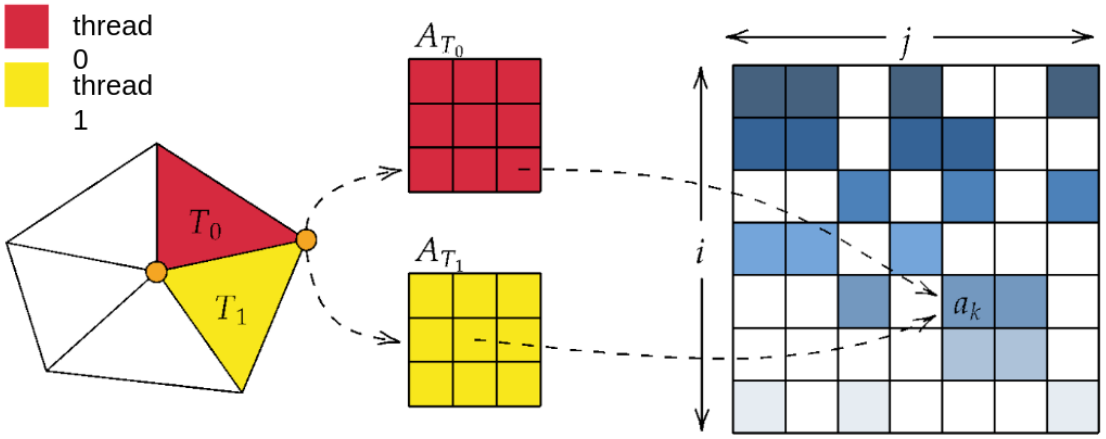


Figure 4.1: Illustration of the cellwise algorithm

4.1 Using a different algorithm: the rowwise algorithm

For the moment, the used algorithm is simple: the idea is to iterate on the cells of the mesh; for each cell determine a contribution; add for each degree of freedom (vertex) of the cell the corresponding contribution to the global matrix. The thing to know is this is the degree of freedom that determines where the contribution is going to be written. Thus, this algorithm implies by definition race conditions because if two cells are adjacent, they share a vertex, and so they will write to a common location. We can see in Figure 4.1 an illustration of the cellwise algorithm.

So, to avoid race conditions, instead of using low-level strategies, another solution is to change the algorithm a little. Instead of making a loop on the cells and finding the degree of freedom of each cell, the idea is to make a loop on the degrees of freedom and for each degree of freedom, find the cells which contribute to this degree of freedom and compute the partial contribution of the cell for this particular degree of freedom; and then make the addition. With this algorithm, we will avoid race conditions if one degree of freedom is managed by a single processor in a sequential way, because two different degrees of freedom will write at different places. This algorithm is described on [10].

Thus, with this algorithm, we avoid all race conditions; but the problem is that the loop will be bigger than looping on the cells: in the cellwise case, we consider only one time each cell. In the rowwise case, we consider each cell one time by degree of freedom, so we consider several times each cell; which

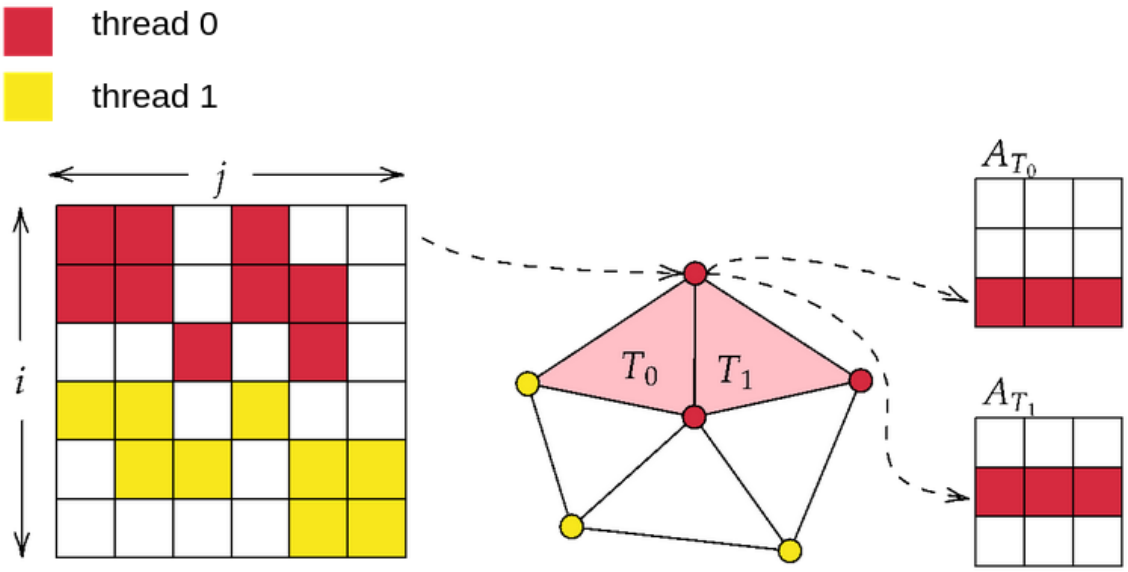


Figure 4.2: Illustration of the rowwise algorithm

means this is longer to compute. We can see at Figure 4.2 an illustration of this algorithm.

This could be a good alternative to the atomic version. The major point is that if there is much memory contention, (eg. if there are a lot of cores), using more costly loops will be faster than doing all the atomic. Another big point is that it is possible to use this algorithm on GPUs, unlike the buffer solution, because the algorithm is designed for avoiding race conditions. Indeed, this is possible because pushing one degree of freedom by GPU thread results in having one thread writing in one given location; so all the threads can write at the same moment. This is the main difference with the classical algorithm; because distributing the cells into the GPU's threads implies a share of the degrees of freedom between the different threads.

4.2 Implementation of the algorithm

We have tested this configuration only on the benchmark code because the algorithm was already implemented, and so the only thing was to port it with StarPU.

The used strategy is simple: we divide the mesh into slices. For each slice,

we create three tasks: One is for gathering the coordinates and the geometry of each cell of each degree of freedom of the slice; another is for computing for each degree of freedom the corresponding contribution, and the last is for scattering the contribution of each degree of freedom. An important thing is also that for implementing this strategy, we use temporary buffers, managed by StarPU, for writing and reading the results of each slice. It means that the gather task will write on a temporary buffer all the coordinates of the degrees of freedom of the slice, and the corresponding compute task will read this temporary buffer to make the computation, so there is a real consumption of data by the compute and scatter tasks. Hence, we improve the memory footprint: after a compute task, StarPU will free the consumed buffer. Thus, the transient memory footprint at a given point is nondeterministic: if you execute all the gather tasks; then all the compute tasks and finally all the scatter tasks, it results in a bad memory footprint because there is one buffer per chain of tasks at each moment. On the other hand, the faster you execute the scatter tasks, the faster you free buffers, and the better your memory footprint. To execute the faster possible scatter tasks, the used strategy is to give priority to tasks. By giving higher priority to scatter tasks (resp. compute) than to compute tasks (resp. gather), StarPU will prioritize the execution of scatter tasks if the used scheduler takes into account priorities.

4.3 Testing and evaluate the algorithm

We have tested this on several meshes; and several configurations. As we said earlier, we have only tested the benchmark code, and not on DOLFINx itself. We try this on the computer sirocco23, which is an accelerated node of PlaFRIM. We have first tried this just on CPUs, then just on GPUs, and finally on a heterogeneous configuration.

We have made, for each configuration, 5 runs.

4.3.1 The rowwise algorithm on CPUs

For comparing this, we have made experiments on the unit-mesh of size 100x100x200; so a mesh with 24 M elements to assemble. We have tried this on two different operators, the first order Laplacian operator, and the second order Laplacian

operator, which is more challenging.

We have also a comparison with an OpenMP version. The loop is simply parallelized by an *#pragma omp for* annotation; and does not use any task. The scheduling policy used is the static one.

To make the comparison, we have tried to make the assembly on all the 2^x for x in $[0..6]$, so we have made a strong scalability test. Also, we have tried three different scheduling strategies on StarPU; eager, LWS, and DMDAR. We look at the speed of assembling, which is the number of cells assembled per second.

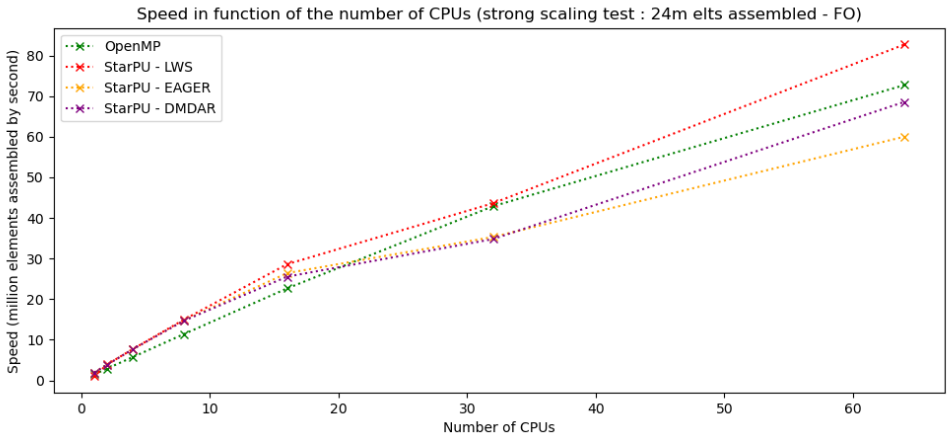


Figure 4.3: Speed of assembling the first order Laplacian operator according to the number of cores.

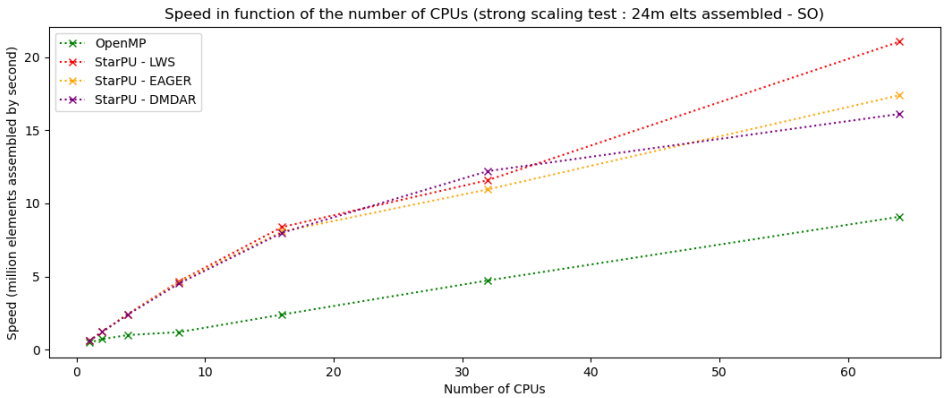


Figure 4.4: Speed of assembling the second order Laplacian operator according to the number of cores.

We can see in Figure 4.3 the speed for the first order operator and in Figure 4.4 the speed for the second order operator.

The first difference between the two operators is there is a scalability difference: for the best cases, it is 80 times better with 64 CPUs than 1 on the first order; and this is 30 times better for the second order. The difference between the two is explained by the memory accesses: the first order is really simple, and the major part of the job is memory access. With this, 64 CPUs optimize well the accesses because of the order of scheduling the tasks; which is not the case with the second order because there is more computation.

Another important thing is that this is faster to use tasks than OpenMP for loops. To confirm this is better to use tasks than OpenMP for loops, we have tried to use the same code used in StarPU with OpenMP, so with the use of OpenMP tasks. We have seen that using OpenMP tasks is faster than using OpenMP for loops, so it confirms that using a task paradigm in this case is a good way of parallelizing. The explanation of why this is better is quite unclear, but it is possible that this is due to memory accesses which are maybe optimized in such a way because there is less memory eviction by considering all the data for gathering and after all the data for computing.

Finally, one can like to see if there is a way to improve the parallelism. We can see in Figure 4.5, the trace of the execution of 5 times the assembly on 64 CPUs, that there is a few red, which means that there are few waiting times; so this is hard to make it better. Be aware that there is a need for a barrier between each iteration, so we can't avoid a bit of red between each iteration.

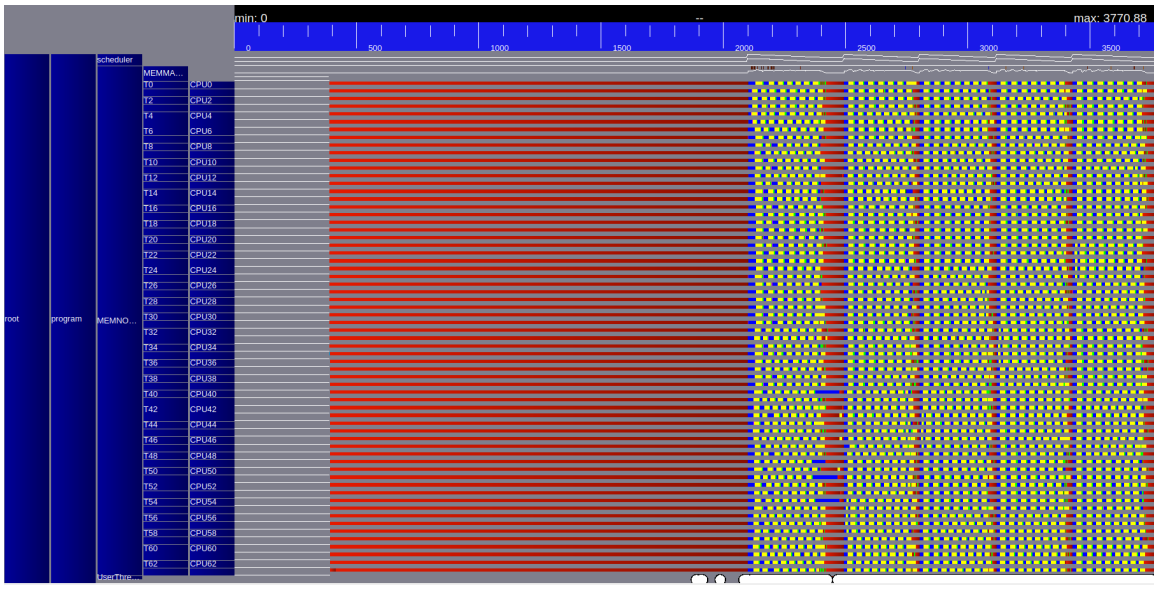
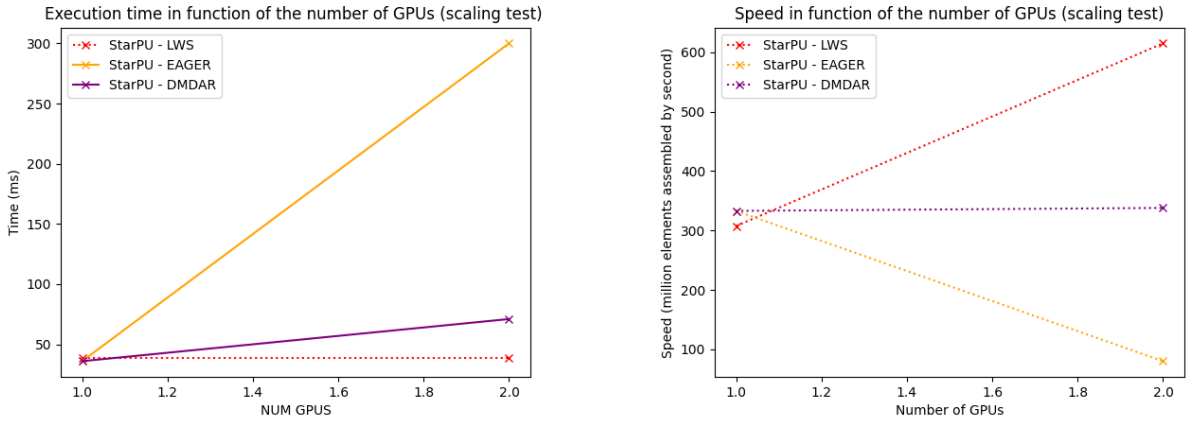


Figure 4.5: Trace of execution of the assembly with 64 CPUs and the LWS scheduler on the assembly benchmark code.



(a) Execution time of assembling 12M elts / GPU

(b) Execution speed of assembling 12M elts / GPU

Figure 4.6: Execution time and speed of assembling 12M elts / GPU on A100s GPUs.

This shows us that this is a good thing to try to parallelize things by using tasks with StarPU.

4.3.2 The rowwise algorithm on GPUs

Without a lot of pain, it is possible to try to use GPUs for making this computation. We have tried to use GPUs on the same computer, by deactivating the CPUs. There is so two NVidia A100 GPUs, and we assemble a matrix with the first order Laplacian operator, and we make a weak-scaling test. It means that we make an assembly for one and two GPUs, and we assemble 12 M elements by GPU, so 12 M for one GPU, and 24 M for two.

We can see in Figure 4.6 the results of this experiment. There are not a lot of things to say here, but something interesting is that using DMDAR is not good. This is because GPUs are really faster: 2 GPUs are ten times faster than 64 CPUs. The problem is that you need a lot of memory transfers to execute the task: as we can see in Figure 4.7, data transfers for assembly on GPUs are ten times slower than the execution of tasks. Thus, DMDAR is going to not put any work on one of the GPUs because DMDAR thinks that this is not worthwhile to do: it is better to put the task on one of the two GPUs if we pay for data transfers each time. But the fact is that we only pay the first time data transfers, because we need quite the same data between two different tasks of the same type, and so LWS is better than DMDAR on two GPUs because it will pay the first transfer and then put work on the two GPUS. Thereby, LWS is two times better than DMDAR, and as

fast as LWS on one GPU, because it will use all the resources when DMDAR only uses one GPU.

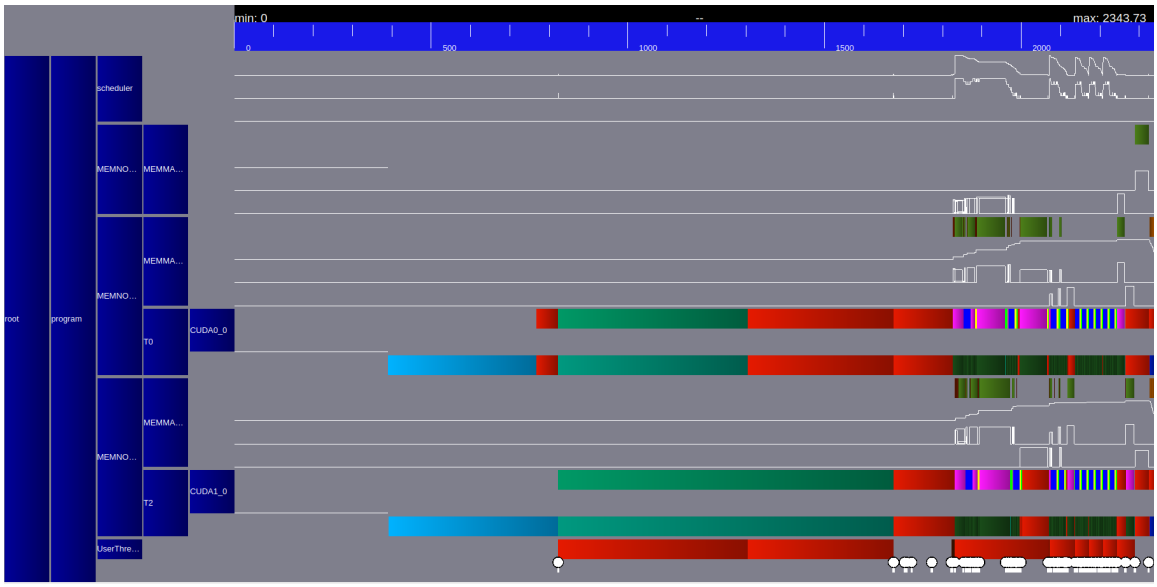


Figure 4.7: Trace of execution of the assembly with 2 GPUs and the LWS scheduler on the assembly benchmark code.

4.3.3 The rowwise algorithm on heterogeneous nodes

When we have the rowwise algorithm on CPUs and also on GPUs, this is possible to try to make it heterogeneous. The first step is to make slices and to let StarPU decides how to balance the work.

We can see in Figure 4.8 that this is not good. The point is that the slices for GPUs are too big to be well executed on CPUs.

A solution to that is to try to make different types of slices. The idea is to try to make small slices, which are going to be executed on the CPUs, and big slices, which are going to be executed on the GPUs We have now to know how many slices of each type we need to do. Experimentally, we can see that for this type of computation, one GPU is approximately 100 times better than one CPU. The idea is to use this approximation to determine what is the number of slices of each type.

But the thing we can see is that we cannot be better than only on GPUs; even with a really good load balancing.

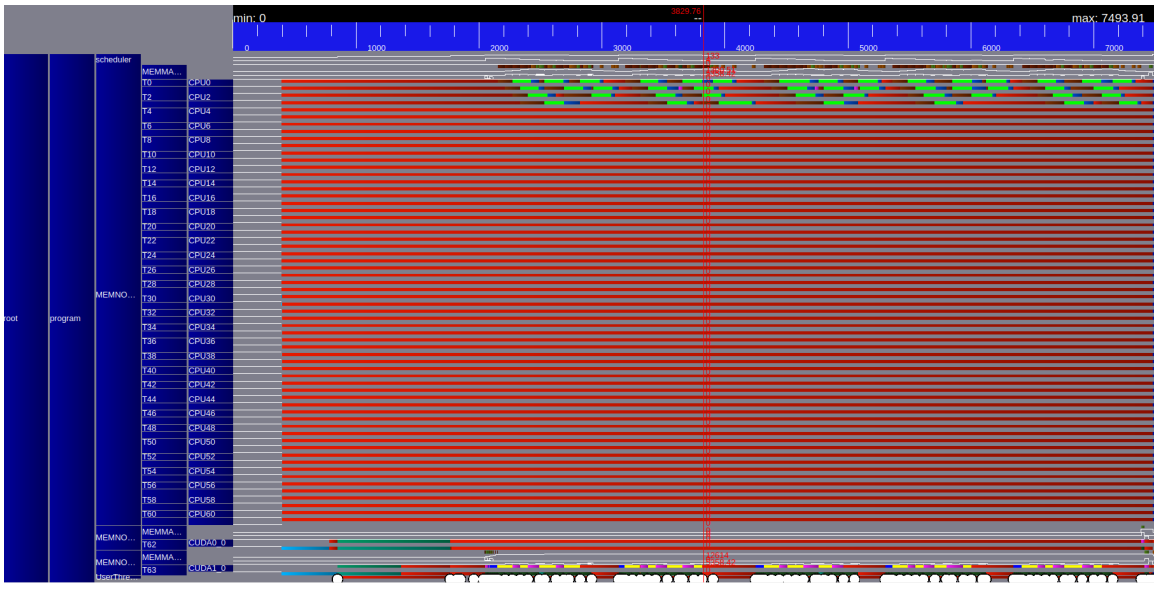


Figure 4.8: Trace of execution of the assembly with 64 CPUs; 2 GPUs with the DMDAR scheduler on the assembly benchmark code.

To explain this, we have made an experiment. The idea is to put all the work on the GPUs but to enable the CPUs. What we see is that this is two times or three times slower (depending on the number of enabled CPUs) than if the CPUs are not enabled. The major thing to understand is that there is no work executed by the CPUs, so the bad times are only due to StarPU. It means that when we go at the same speed with than without CPUs enabled, it is in fact two times faster than without working CPUs [1].

When we look further at the CUDA API calls; what we see is that the calls of CudaLaunchKernel are really slow when CPUs are enabled: when CPUs are enabled, those calls can take several hundred milliseconds; without it is something like 0.1ms. Looking again further, we see that without any work on the CPUs, the calls at the function _starpu_get_worker_task are really expensive, and when we put work on CPUs, we can divide by eight the number of calls of this function.

This function is called by the workers to decide which task they have to execute. Normally, they have to sleep if they have nothing to do, like when no task is put on CPUs. Here, the point is that the variable keep_awake is left to 1, which prevents workers to sleep, and so implies busy waiting. The first explanation of this is the big transfers between CPUs and GPUs because if there are

[1] A thing has to be noted. With P100 GPUs it is possible to be 1.1 times faster in a heterogeneous way than on an only-GPUs way. The difference with A100 GPUs is explained by the fact that P100 GPUs are really slower than A100 GPUs, so they are not 100 times better than CPUs.

transfers, there are things that change, so StarPU lets workers awake, but there is no proof of this for the moment.

Future work is to look at these calls to see what is going on, for making rowwise on heterogeneous nodes efficient.

4.4 What about managing computing resources in a clustered manner ?

Naturally, with this problem, a solution should be to cluster the computer. It means that instead of considering all the CPUs individually, we consider CPUs cores collectively by logical affinity groups. The idea is to let StarPU consider some CPUs, like one CPU by L3 cache, and then create OpenMP threads bound on the CPUs not used by StarPU. Then, when a work is submitted on a StarPU core, the core is going to subdivide and share the work among the cores of its logical group, by submitting some OpenMP work. This results in two-level hierarchical scheduling with StarPU being responsible for the coarse-level scheduling of tasks to logical core groups, and OpenMP being responsible for scheduling pieces of these tasks' work among the cores within a group.

In our case, it could be an interesting solution: as we said earlier, the overhead of using CPUs depends on the number of units managed. Thus, if you have fewer CPUs used, you have less overhead. Furthermore, because a CPU worker of StarPU is not only one CPU but several real CPUs, it means that one CPU worker is stronger than without clustering, and so the power of one CPU is closer to the power of one GPU.

We have tried this for the rowwise algorithm to make the CPUs better. The problem is that, again, GPUs make this type of computation too fast for CPUs; and also this is complex to have a good slice size.

It could be an idea to explore in future work.

4.5 Chapter Summary

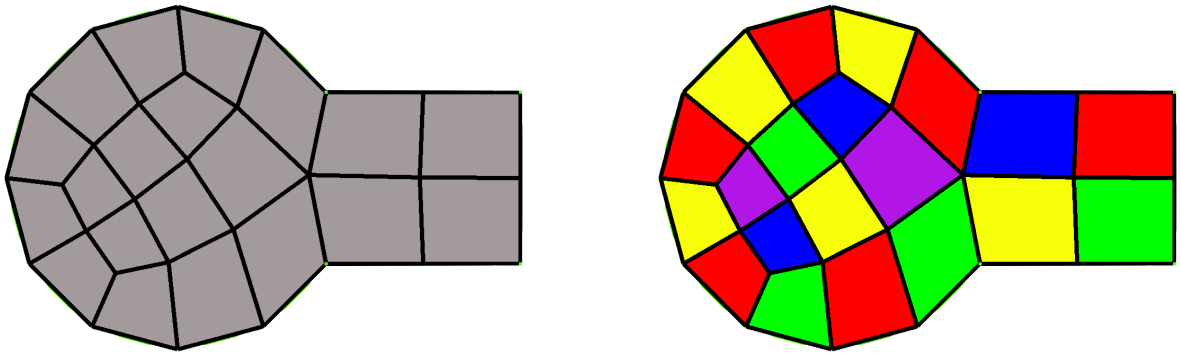
In this chapter, we have seen a new algorithm for assembling matrices with the FEM. This algorithm, the rowwise algorithm, is designed to avoid race conditions. We have seen this algorithm is quite efficient on CPUs, and also could be efficiently used on GPUs only. A surprising point is that the scheduler LWS, known to be limited, is the best of the predefined scheduling algorithm implemented in StarPU for this computation.

We have also seen that there are probably internal problems with StarPU for using efficiently this algorithm in a heterogeneous way, because of the availability of the CPUs which are making busy waiting, and using computer in a clustered fashion is also not an efficient idea.

5

Coloring the mesh to avoid race conditions

> *If you are going through hell, keep going.*
> **– Winston Churchill –**



(a) Example of FEM mesh uncolored.

(b) A valid coloration for this mesh.

Figure 5.1: A mesh and a possible correct coloration.

By thinking a little more about the problem, we see that we have a graph problem: we would like to avoid adjacent slices to be considered at the same time. When we say this, we think directly of the graph coloration problem. The goal of this part is to see how coloring the mesh can help us to avoid race conditions for the cellwise algorithm.

5.1 Trying to color the mesh to avoid the race conditions on the cellwise algorithm

This is natural to see our race conditions like a graph problem: if we divide our mesh into slices, like in the other chapters, we can attribute to each scatter task a color. The idea is that two slices can be of the same color only if they do not share any degree of freedom, and so they can be executed simultaneously. With this, the scatter part for each color could be performed without any race condition.

We can see in Figure 5.1a an example of mesh for the FEM; and a valid coloration for this mesh: for a given cell, each neighbor has a different color from the starting cell. One can see that we color in this example the cells of the mesh; even though we talk earlier about coloring slices. The point is that this is the same thing, we just, by simplicity, forget this detail. We can see in Figure 5.2 a coloration valid for a mesh, with cells separated by white lines and slices separated by black lines.

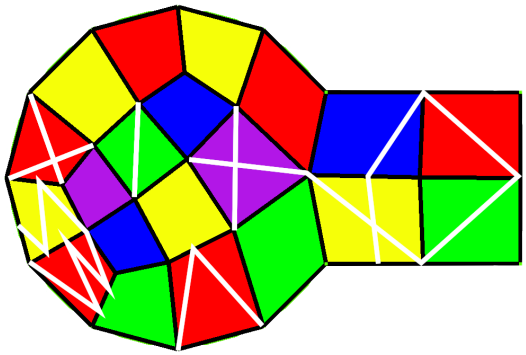


Figure 5.2: A slice coloration of the mesh of Figure 5.1a but with smaller cells.

To color this mesh, we have proceeded with an incremental approach, and the goal of the next sections is to describe the different steps.

5.1.1 Experiments

For the experiments in this chapter, we have used two different platforms: PlaFRIM and Ex3. When there is a consideration of the real DOLFINx, the experiments have been made on the partition xeongold16q of Ex3. When we use the test case, the experiments have been made on the partition bora. These two nodes are standard nodes.

For each experiment, we made 10 measurements and then the mean.

5.2 First version of the solution: one color by task

To do this, we have first to succeed in dividing each big task into three different tasks, one for each phase, like in the previous chapter. It would be bad to let the tasks big: the gathering phase and the computing phase could be performed in parallel because there are no race conditions on it, unlike the scatter phase.

So, the first version of the coloration is really simple, it corresponds to attributing one color by task, and so to make the scatter sequential. It is a form of valid coloration because each task has a different color, so each neighbor has a different color. One can say this is a stupid version, and this is the case, but this simply can lead us to some little important optimizations.

To implement this version, we have only put a piece of data in Read-Write access for the scatter tasks. We have tested this on DOLFINx directly, on the xeongold16q partition of Ex3.

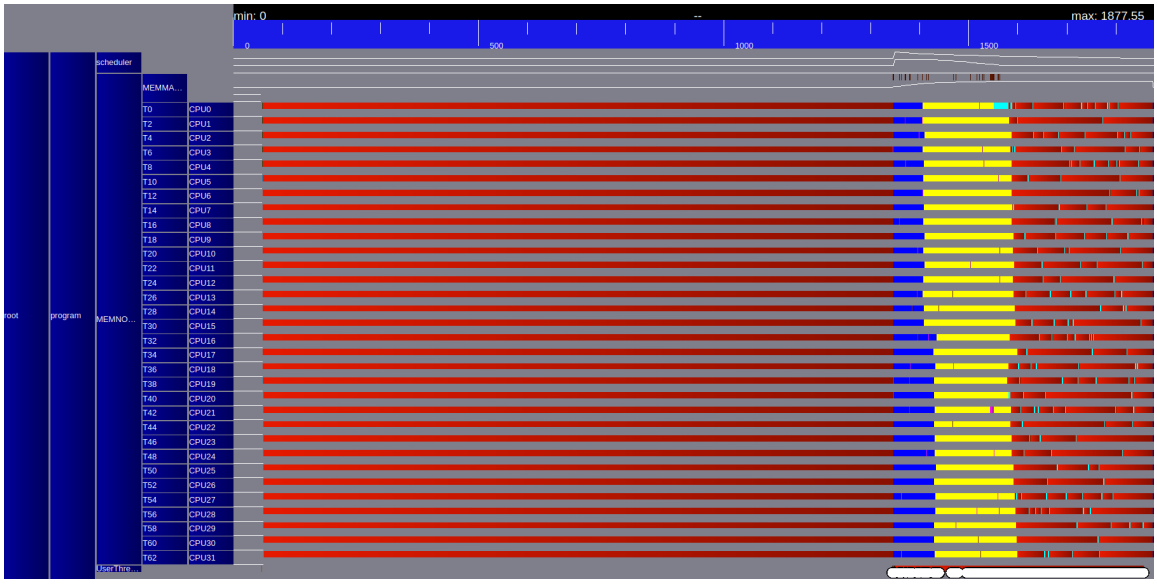


Figure 5.3: Trace of execution of the assembly with 32 CPUs with the LWS scheduler on DOLFINx; write accesses for scatter (scatter tasks are sequential)

We can see in Figure 5.3 the trace for this assembly, without atomic and so with sequential accesses on the buffers. What we can see on this trace are three different things :

- First, the scatter tasks (in light blue) are likely too small: they are less than one millisecond.
- Second, we have clearly a load balancing problem. All the scatter tasks are executed at the end, which is a problem because they could be performed in parallel with the other types of tasks. If you execute the tasks of scatter type when they arrive, you will have more parallelism. The idea to solve this is to use priorities and put bigger priorities on scatter tasks than on other tasks.
- Third, similarly to the second, it should be a good idea to put commute access on the global matrix, to permit tasks to be executed in an unordered way, so a scatter task will not be blocked if another scatter task is submitted before it is not ready.

We will first evaluate the respective influences of these modifications time after

time, in the case of Poisson with 32 CPUs. This is because, with 32 CPUs, we have the worst scalability on the atomic case.

Without any modification, we see that the execution time is 0.526 seconds. So, naively, this is three times slower to execute this assembly than the atomic one.

When we apply the first modification (make bigger tasks), we see that the execution time is 0.515 seconds. This is not really better than before, but we will keep this modification because bigger tasks imply fewer colors (with a smaller number of tasks, tasks will share fewer degrees of freedom; and so there are a smaller number of colors) and so fewer race conditions.

Then, when we do the second modification (they concern the second and the third point), we see that the execution time is 0.375 seconds, which is really better. This is far from perfect, but this is really faster than without priorities. We can see at Figure 5.4 that in this case, some of the scatter tasks are computed concurrently with other types of tasks (the turquoise ones on CPU2), which enabled more parallelism as we said.

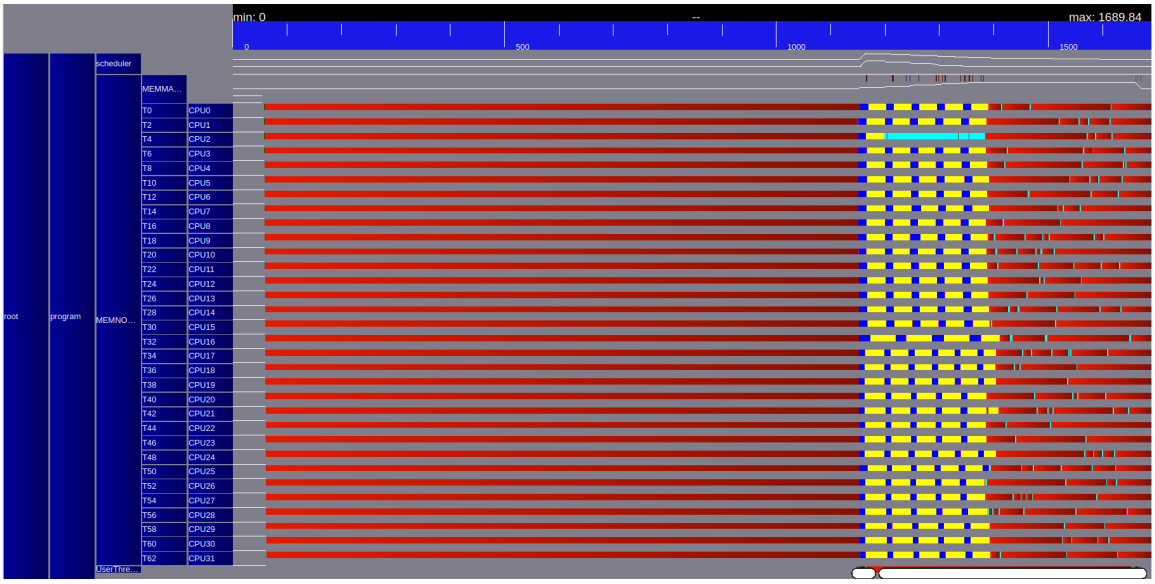


Figure 5.4: Trace of execution of the assembly with 32 CPUs with the LWS scheduler on DOLFINx; write accesses for scatter and priorities (scatter tasks are sequential)

Now, we can look at trying to make the scatter parallel with a better coloration.

5.3 Second version of the coloration: enabling several slices on the same color, use of barrier

We are going now to explore this coloration further. We will first see how we can color the mesh; then the problem we have to execute tasks with coloration, and finally a preliminary solution for solving this problem.

5.3.1 Coloration of the mesh and needed dependencies.

In this section, we keep the same things than on the first section, in terms of the division of the mesh, division of the work, and coloration. So, the idea is to only color scatter tasks, in the way that two scatter tasks of the same color can be executed simultaneously, and two scatter tasks of different colors will not be executed simultaneously.

The first point we have to see is "how we color the mesh". For this, we transform the mesh into a graph like this :

- A vertex of the graph is equivalent to a slice of the mesh.
- An edge of the graph relies on two vertices if and only if the two corresponding slices are adjacent on the mesh.

It is easy to construct the graph from the mesh, only by browsing several times the global matrix.

By doing this, we can see that our coloring problem is just a classical coloring problem on a graph; and so we can apply all the well-known relative theories.

Within the scope of this internship, we use the classical greedy algorithm *First-fit* (see Algorithm 1); but we can, if we want, use better algorithms.

Algorithm 1 The First-Fit algorithm for coloring vertex of a graph

Require: $G(V, E)$, an undirected graph
  for all $v \in V(G)$ **do**
    $col(v) \leftarrow minUsedColorByNeighs(v)$
  end for
Ensure: A valid coloration for each vertex of G

It is really important to notice that this algorithm does not give an optimal coloration, but only a valid coloration in less than $\Delta(G)+1$ colors, with $\Delta(G)$ the maximum degree of a vertex of G.

When our mesh is colored, we can now try to express what are the dependencies we need.

- First of all, we have the dependencies of the "chain" of tasks: we have, for each slice, a gather task, a compute task and a scatter task, and the scatter depends on the compute which depends on the gather. This is not hard to express, it suffices to use the classical STF dependencies.

- The second one is more tricky: we need to express the fact that two tasks of different colors cannot be executed at the same moment. This is a hard thing to express with the STF because this is "execution dependencies": there is a dependency at the execution because before the execution of a task, the others do not depend on it. The STF does not address this type of dependency, since it exclusively focuses on the task submission step.

We will now see the first idea to express this type of dependency, and we will see that this is not an optimal solution.

5.3.2 A preliminary solution: express dependency by a barrier

The first solution to express this dependency is to add dependencies at the submission to make them dependent. The simpler solution is to create a barrier between each color. To do this, we submit all the Gather and Compute tasks; and then, we submit color by color the scatter tasks, by adding between each color a barrier task. We can see at Listing 5.1 an STF pseudo-code for this expression. We only put the pieces of data which imply dependencies.

Listing 5.1: STF code for express coloring by barrier

```
int *data_gather[NB_TASKS], *data_compute[NB_TASKS],
    *data_barrier[NB_COLS] ;

for(int t = 0 ; t < NB_TASKS ; t++) {
    submit_task(gather, data_gather[t], WRITE) ;
    submit_task(compute, data_gather[t], READ,
        data_compute[t], WRITE) ;
}
```

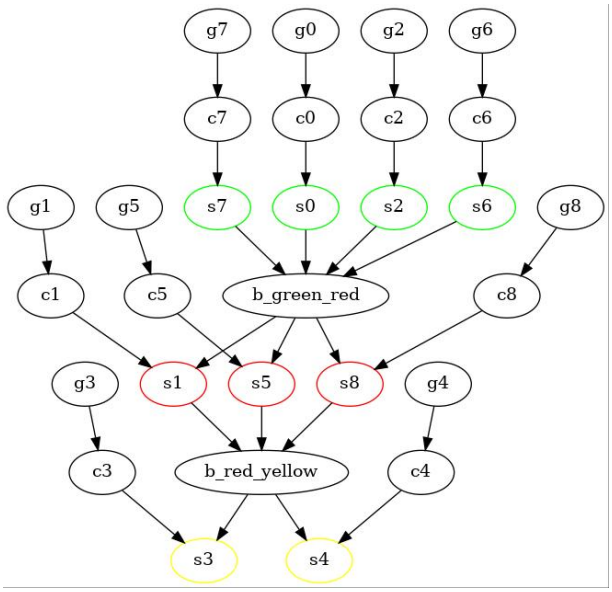


Figure 5.5: Example of DAG with coloration for NTASKS = 9 ; NCOLS = 3

```
int ** task_by_color = get_indexes_tasks_by_color() ;

for (int c = 0 ; c < NB_COLS ; c++) {
    for (int t = 0 ; t < nb_tasks_of_col(c) ; t++) {
        submit_task(scatter, data_compute[t], READ,
            data_barrier[c], READ) ;
    }
    if (c < NB_COLS - 1) {
        submit_task(barrier, data_barrier[c], WRITE,
            data_barrier[c+1], WRITE) ;
    }
}
```

With this solution, we lost a lot of flexibility because now we have introduced an order on the execution of the scatter tasks, and this is a problem. To illustrate the problem, we consider for example the case with 9 slices, and 3 colors, like in Figure 5.5. We see that if for example none of the red tasks (s1, s5, s8) are ready to be executed, but both c3 and c4 have been finished; with this expression, we cannot execute them, but with the dependencies actually required, it is possible to execute them. So, this solution is clearly non-optimal.

5.3.3 Evaluation of the solution

Now we have talked about this primary solution and about the non-optimality of this solution, we will try to evaluate whether this is an efficient solution or not.

We experimented on the same computer and configuration as before. It means that we try for the case Poisson 4096 x 2048 with 32 CPUs.

First, we have to notice that the complete time is really really longer: we have to color the mesh instead of just submitting the tasks. It implies a big time to compute only the coloration. We do not consider this time here because the only interesting thing is the time for assembling.

What we see is that, for 5 measurements, we have an execution time of 0.305 seconds. In comparison with the sequential version, we have a gain of almost a quarter of the execution time.

5.4 Third version of the coloration: enabling an unordered execution, simplification of the problem

We would like now to make a more efficient solution, by suppressing the barrier and making an execution by the coloration.

To do this, we will first explain the idea about how to make this unordered execution by going a little bit beyond the STF model; then show a simplification of the problem; next, describe the used algorithm for this solution, and finally evaluate this idea.

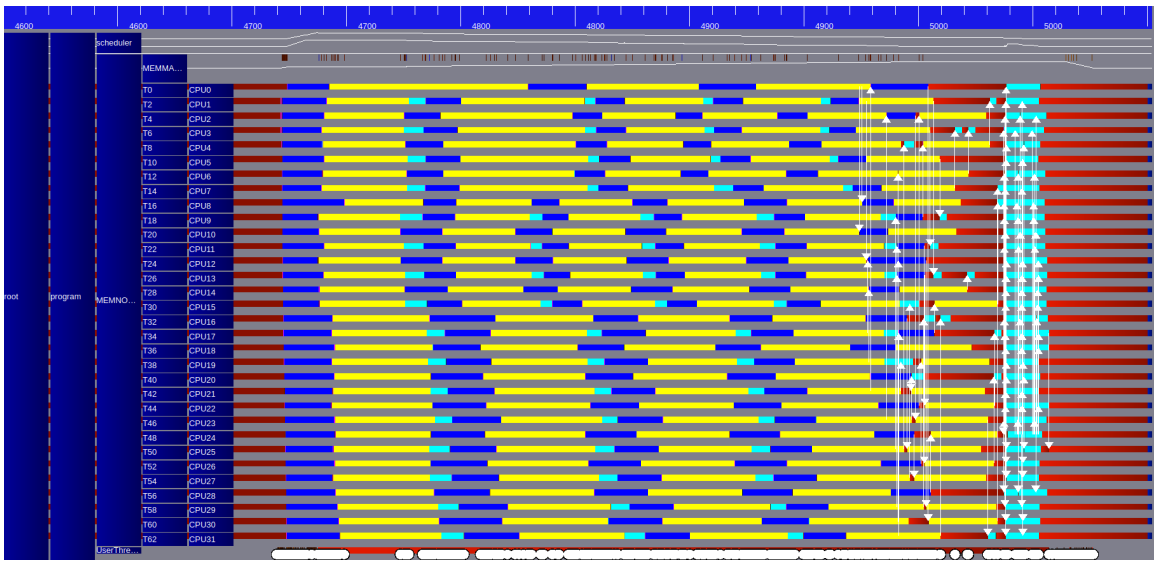


Figure 5.6: Trace of execution of the assembly with 32 CPUs with the LWS scheduler on DOLFINx ; coloration of scatter tasks with barrier

We can see on Figure 5.6 a trace for this execution.

We can first see that because of the new dependencies, the turquoise tasks are executed before: they allow the liberation of more tasks. We see also that there is again room for optimizations and improvements, with again a lot of sleeping time (in red). There are two things to gain: first, we have again a problem of barrier: we see the liberation of a lot of tasks at the same moment at the end, which means that we have a task that is blocking all the others. Second, we have maybe a problem with the amount of parallelism. If we manage to increase the amount of available parallelism, we can maybe gain by having less waiting time.

We will now discuss some strategies to have this.

5.4.1 How making this execution unordered

In the previous section, we have seen that this is not efficient to use a barrier, because this use implies an order in the execution of the tasks, which is a big problem. We have seen also earlier that there are two kinds of dependencies, the first between the tasks of the same chain, easy to express with the STF, and the other between the scatter tasks of different colors, hard to express with the STF. Furthermore, the problem is that it is nearly inexpressible with the STF: the STF implies an order between dependent tasks (one has to be executed before the other), and what we would like is a mutual exclusion during execution. We know that this is common, and it corresponds to the COMMUTE access mode on several runtime systems, but here this is a mutual exclusion between different sets of tasks, so really hard to do with the tools we have on STF.

Thus, an idea that comes to mind is "why we should use the STF for something which is not in the STF ?" For having these dependencies, we will have in a hand the classical STF dependencies, for the chain of tasks, and on the other hand, dependencies managed by hand.

The idea for this is to create different bags of tasks. There is a bag for the gather and the scatter tasks; and also one bag of tasks per color. We attribute to each color one bag, and we will put all the tasks of one color on the corresponding bag.

Then, during run time, we will distribute the different workers to the bags of tasks; to enable the execution of a single bag at any time, instead of multiple bags concurrently. It means that we have a referee algorithm that has the goal of dynamically managing the resources between the colors for :

1. Ensure there is only one color executing at the same moment.
2. Ensure the executing color is the one that enables more parallelism.
3. Ensure we never have a loss of parallelism because of a prior bad decision. This point seems repetitive to the last one, but the idea is we need a compromise between enabling more parallelism at the call, and also having good parallelism later.

We will talk later about this distribution, in the description of the algorithm. The point is that the referee can also play with the bag of gather-compute tasks to optimize the execution time.

For StarPU, a bag of tasks is called a context of execution. It is designed for our problem; it means that we can attribute exactly resources to one or several contexts. For distributing the resources, there is a tool called "context hypervisor", programmable for our needs; and callable everywhere on the program.

5.4.2 Go out of the FEM to simplify the problem

In order to experiment with this approach, we first implement it on a synthetic program that mimics the key FEM requirements in a simple setup.

On the other hand, we have a coloration problem; which is a quite common problem. It is sometimes natural to express dependencies with coloration.

So, the idea we have is to go out of the FEM to solve our problem. The advantages are having a more flexible problem (we can change parameters to change the time of the tasks, the number, ...); an easier problem (we do not need any prerequisite to understand it); and finally an independent problem (without any external library dependency).

We will now express the problem we have, and how we will solve it by an algorithm close to the one used in FEM.

Problem statement We have at the beginning a number of tasks $nbTasks$ and a number of areas $nbAreas$. From this, we create two arrays, an array $arrayProblem$, and an array $arraySolution$. We have also a factor $FACTOR\_TASK$. The size of $arrayProblem$ is $nbTasks \cdot FACTOR\_TASK$ and the size of $arraySolution$ is $nbAreas$.

We initialize the array $arrayProblem$ randomly ; we have also an array $whereToWrite$ of size $nbAreas$ which assigns to each number from 0 to $nbTasks$ a number from 0 to $nbAreas$.

Thus, our problem is a simplification, with only some readings, writings, and additions. We would like to, for each index i from 0 to $nbTasks$, computing the value v which is the sum of the indexes $i \cdot FACTOR\_TASK$ to $(i+1) \cdot FACTOR\_TASK$ of the array $arrayProblem$ and adding v at the index $whereToWrite[i]$ of the array $arraySolution$.

For now, the problem does not seem to resemble the FEM assembly problem. We will see the likeness come from the fashion of solving this problem.

Solving the problem Now, we will see how we solve the problem *at the manner* of the FEM.

The first thing is to have a division of the work in *tasks*; and with more details, a division of work in the chain of tasks. Thus, we will create $nbTasks$ chain of tasks, each one of size 3. We have, on each chain, three types of tasks :

- The tasks of type **A**. The aim of these tasks is to compute the value v, so reading on the array $arrayProblem$, and put the sum of the values of this array from $i \cdot FACTOR\_TASK$ to $(i+1) \cdot FACTOR\_TASK$ on a buffer, called *bufferReader*.
- The tasks of type **B**. The aim of these tasks is only to transfer the value on *bufferReader* to *bufferTransmitter*, another buffer. This task is only here to have a chain of size 3 of tasks.
- The tasks of type **C**. The aim of these tasks is to read on *bufferTransmitter* and to add this value at the index of $arraySolution$ correct for the given task.

One should notice that the duration of tasks is quite short. To overload this,

we make some calls to usleep on the tasks to synthetically control their duration. Also, to more realistically emulate the occurrence of consistency errors (if two tasks of type **C** writing in the same place are executing at the same time), we do several times the operation of add (and so subtract) on this task.

5.4.3 Description of the used algorithm

To solve the problem, we have used several strategies for making the dependencies of the tasks. We will now describe all of them.

First, we say that we make dependencies between each task of a given chain on each strategy.

For the first strategy, we use a classical one. It is called the "unsafe" strategy. The idea is only to submit all the tasks of type **C** without any dependency, and then see. This version is not correct; this is only a version to see what is the best to have. It means with this, we obtain a theoretical lower bound on the execution time of safe versions, so this strategy is only here to know what we can again gain.

The second strategy is also classical. This is the "barrier" strategy. It means we submit the tasks of type **C** color by color, with a barrier between each color. This version is similar to the version described in Section 5.3. This version is here to first evaluate the overhead of using barrier and second to make a more realistic comparison with the example of the FEM.

The third strategy uses the tools described in this section: we push each color of tasks into a context, and then we attribute with a hypervisor the resources to enforce the coherency of data. This strategy is called the "simple bags" strategy The used algorithm for the hypervisor is quite simple :

- First, we attribute all the resources to the first color.
- Second, each time color has ended; it transmits all the resources to the next color.

This is only a proof of concept.

The fourth strategy extends the third by using several additional buffers: we

have seen previously that using buffers could be a good idea; but only if we have not too many buffers. This strategy is called the "several bags" strategy. So the idea is to have n buffers and divide the resources in n. We attribute a $\frac{1}{n}^e$ of the resources to the n first colors, and each time color has completed its processing, it transmits its resources to the next color. The idea with this is to have a bigger amount of parallelism available at the same moment. For our example, we take $n = 3$.

The fifth strategy is an improvement of the fourth by making a dynamic distribution of the resources. The idea is to give the resources to the worker with the biggest number of tasks able to be executed. For this, we register in counters different information for each color: the total number of **C** tasks; the number of **C** tasks already executed; the number of **C** tasks ready but not executed; and the number of **C** tasks currently in execution. Thus, when the last **C** task of one color is finished, it triggers an explicit resizing of the resources, so the hypervisor will find what is the best to do. This strategy is called the "several bags dynamic" strategy.

5.4.4 Evaluation of the idea

To evaluate this, we will try the five strategies on a computer of the partition *bora* of *PlaFRIM*. We try several configurations, represented as $nbAreas$ - $nbTasks$; and several numbers of CPUs used to consider different things.

For all the buffered strategies, we have used three buffers.

We can see in Figure 5.7 the different times for the different strategies.

We can see several things. First, there is an overhead of using barriers over the baseline unsafe variant. Second, using the last strategy is quite good; and allows us to be really close to the version without security. It gives us the idea that this is an efficient way for executing the coloration problem.

Another important point to see is that using the 3^{rd} strategy is not better than using the barrier: this is normal because this is the same thing: we give colors without any concerns with the execution, and only one color could be executed at the same time. So, to see if this is only the buffers; or if using the 5^{th} strategy is a good idea, we will make an appendix of this Figure, to try the same strategy

conf.	40-83				400-835				4000-40000		4000-40135	
Nb CPUs	4	12	24	36	4	16	24	36	12	36	12	36
1^{st}	2.79	0.97	0.52	0.37	27.61	6.95	4.65	3.12	441.23	147.05	442.61	147.54
2^{nd}	3.00	1.24	0.90	0.71	28.01	7.33	4.95	3.44	442.24	148.32	443.04	148.79
3^{rd}	2.99	1.38	0.86	0.72	27.93	7.34	5.10	3.56	442.29	148.30	443.69	148.74
4^{th}	2.85	1.05	0.56	0.48	27.74	7.05	4.79	3.26	441.37	147.50	443.00	147.92
5^{th}	2.83	1.03	0.59	0.48	27.69	7.07	4.78	3.27	441.46	147.37	443.05	147.94

Figure 5.7: Time in seconds of coloration problem for different strategies, different configurations, and different numbers of CPUs.

config / strat	40-83		400-835		4000-40000		4000-40135	
Nb CPUs	4	12	4	16	12	36	12	36
$3^{rd} + 5^{th}$	2.98	1.30	27.96	7.39	442.26	148.27	443.70	148.78

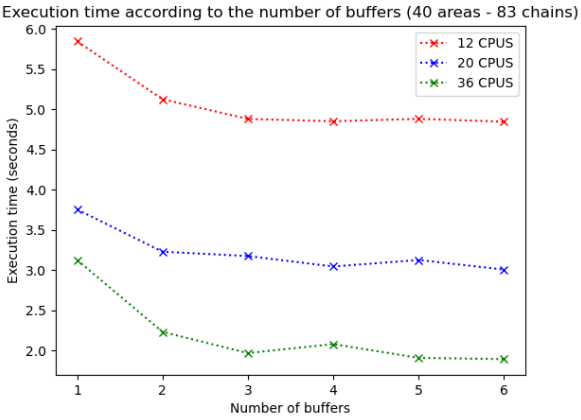
Figure 5.8: Appendix of Figure 5.7 ; mix of 3^{rd} and 5^{th} strategies.

as the 5^{th}, but with only one buffer. This appendix is available at Figure 5.8. We have not made all the experiments because the firsts were sufficient to give us an idea.

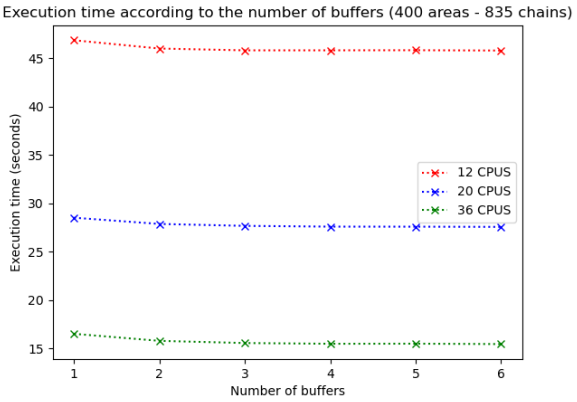
What we see is that this is not clear whether this algorithm is really better. Sometimes, we can gain a little, but we see the major gain is due to the use of several buffers.

Finally, we have seen that this is better to use several buffers, but we have not talked about the number for now. To evaluate the good number, we look at the execution time according to the number of buffers for several configurations. We can see in Figure 5.9 these times for the case $40 - 83$ and $400 - 835$. We have also looked at bigger cases, but the differences were not seeable, so we do not put them on the report.

We can see that after 3 buffers, there is no gain to increase the number of buffers, so we will keep this number. An interesting thing is that for the case $40 - 83$, using 36 CPUs with one buffer is close to using 20 CPUs, but having 2 or more buffers improves a lot the execution time. It is explained by the fact there is not enough parallelism to use all the CPUs, and so there is starvation, while when



(a) Evolution of the execution time according to the number of buffers (5^{th} strategy, 40 areas, 83 chains).



(b) Evolution of the execution time according to the number of buffers (5^{th} strategy, 400 areas, 835 chains).

Figure 5.9: Evolution of the execution time according to the number of buffers with different configurations.

it is possible to execute several colors at the same moment, it enables enough parallelism for all the cores.

5.4.5 Go back to the FEM

In the beginning, the main aim of this part is to describe the results of the assembly of the FEM.

So now, we will describe how to use all this work for trying to have better results on the assembly.

We will use the fifth strategy, described previously for making the assembly because it was the more promising version.

We evaluate this on the partition xeongold16q of Ex3. We have tried incremental experimentation.

First, we try Poisson experiment, for size 4096 x 2048. We have seen, by making 3 experiments by the number of cores, this :

Number of CPUs	4	8	16	32
Execution time	0.560	0.407	0.312	0.335

A remark is that for 4 and 8 cores, we are really faster than with the atomic

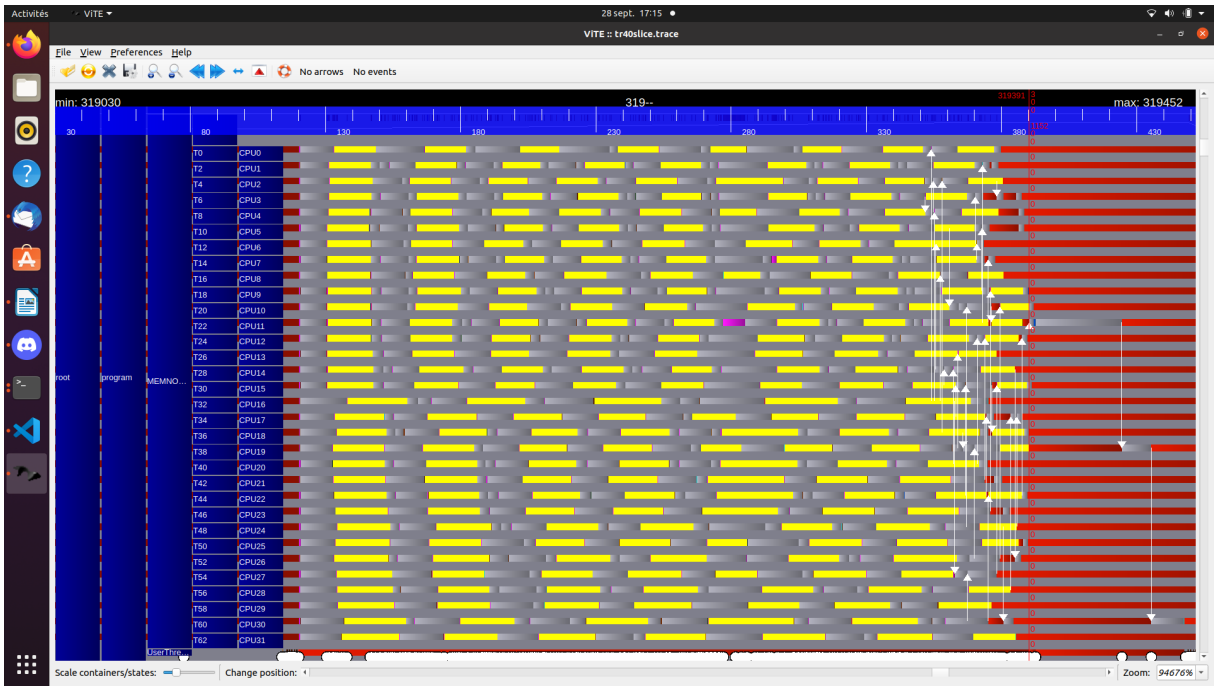


Figure 5.10: Trace of execution of assembly with 32 CPUs, Poisson case (4096 x 2048), xeongold16q partition

version; but for 16 and 32 cores, we are slower.

Also, when we make a lot of times the experiments, we see a big variability: it goes from 120 milliseconds to 420 milliseconds. We have for the moment one preliminary explanation:

We have three different groups of resources. When we finish a task of type "compute", we launch the attribution of the resources. It means we attribute to the color a group of resources. The problem can be that we attribute a group of resources to a color for which the "compute" task set has been executed by another group of resources. So, the problem is that we could have an eviction of data, and so dramatic data access.

We have several things which let us think that. First, we have variations of execution times, so the good execution times are the ones with good repartition, and the bad execution times are the ones with the worse repartition. Second, for the case with 4 and 8 CPUs, we do not have this type of problem, because there is a share of CPUs caches. Also, there is more luck to have a good repartition. Finally, when we look at the trace of the execution (Figure 5.10), we see that we have a really good repartition of the work, except for the final three last tasks, which constitute an unoptimized reduction. By comparison to the case "4" CPUs,

we only see that the tasks are longer.

Thus, a thing to do is probably to improve again the scheduling, for optimizing the data access, and so have good execution times for the coloration.

5.5 Chapter Summary

In this chapter, we have seen another way of assembling matrices, by coloring the mesh and executing in parallel only cells which have the same color (and so do not share any degree of freedom).

We have made an incremental approach, by doing different types of coloration. Also, we have tried to find a way of expressing with StarPU, and the limited constraints of the STF, dependency such as coloration.

We see this way of doing is efficient for a little number of CPUs but is not good for a big number of CPUs. A preliminary explanation is a problem of scheduling, which has to be explored further to be sure.

6

Conclusion

Knowledge is power.

– Sir Francis Bacon –

6.1 Different techniques to assemble matrices

In this report, we have seen several ways to assemble matrices in parallel, by using StarPU. We will now make a quick abstract of the several strategies.

The first strategy seen is the atomic strategy. The idea is to avoid race conditions of the operations only by using atomic operations. To sum up, the race conditions are only during a scattering phase, when all the cells are writing to the global matrix, which can cause some concurrent memory reads and writes. With this strategy, the solution is to make memory accesses atomic, and so protect the matrix.

The second strategy is the buffered one. The idea is, that each worker is going to write to a different global matrix, and at the end of the computation, we will reduce all these temporary matrices in the same buffer.

The third strategy is the use of a different algorithm, the rowwise algorithm. The idea is to see that the race conditions are due to the classical algorithm, because we distribute the cells, and the cells share degrees of freedom. Each degree of freedom is associated with one location in the global matrix, and so if two cells share the same degree of freedom, the computations associated with them will write at the same location. To avoid this, the aim is to use a different algorithm, the rowwise algorithm. With it, instead of distributing the cells, these are the degrees of freedom that are distributed, and then we recover the corresponding parts of each cell to write on the global matrix. Because a worker will compute a whole degree of freedom, it implies no race conditions.

Finally, the last strategy is the coloration strategy. The idea of this strategy is to divide the mesh into slices and to attribute to each slice a color. The idea of the color is that two slices have the same color if they do not share any degree of freedom, and so they can be executed in parallel.

6.2 Evaluation of the different strategies

All these strategies have been tested, and we can now conclude about the advantages and disadvantages of all these methods.

The worst is the buffering strategy. This is because of the allocation, reduction,

and deallocation time due to the use of one buffer per worker. The execution of these non-compute phases is so long that it forbid us to use it in the general case. But this is not something to forget, because it could be a good idea to use it, for example, to make better one given method.

We can also evaluate the atomic strategy. It is not inefficient, but it suffers from a lack of scalability, which makes it unusable with a lot of CPUs.

Finally, there are the two last solutions, the rowwise one and the coloration one. These strategies are really different, the first has the idea to use a different algorithm that is designed for avoiding race conditions; and the second uses the classical algorithm, but with a smart coloration, it allows us to have parallelism. The rowwise algorithm is quite efficient and could be executed quickly on a lot of CPUs. It is also efficient on GPUs, but there is an internal problem in StarPU that makes it inefficient in a heterogeneous fashion.

For the coloration strategy, we have seen this is inefficient for a big number of CPUs, probably because of a scheduling problem.

We have also to notice that the onlies applicable to a GPU context are the atomic one and the rowwise one.

6.3 Future work to improve these strategies

Finally, we can note some things to make faster these strategies.

For the rowwise one, we have to understand why there are so many calls to the function which let awake the CPUs, and so causes overhead.

Another point is to improve the coloration, to make it faster and better, and also to find other strategies for hypervising the resources, to improve the scheduling of tasks. Also, we need really to relate the hypervised resources and the executed resources, to be sure the attributed resources for executing scatter tasks of a given color are linked to the resources which have executed the compute tasks of this color, which is not the case for the moment.

References

[1] D. J N Reddy. *An Introduction to the Finite Element Method*. McGraw-Hill Education, 2005 (cit. on p. 5).

[2] M. S. Alnaes, J. Blechta, J. Hake et al. `The FEniCS Project Version 1.5'. In: *Archive of Numerical Software* 3 (2015) (cit. on p. 6).

[3] M. S. Alnaes, A. Logg, K. B. Ølgaard, M. E. Rognes and G. N. Wells. `Unified Form Language: A domain-specific language for weak formulations of partial differential equations'. In: *ACM Transactions on Mathematical Software* 40 (2014) (cit. on p. 6).

[4] R. C. Kirby and A. Logg. `A Compiler for Variational Forms'. In: *ACM Transactions on Mathematical Software* 32 (2006) (cit. on p. 6).

[5] M. W. Scroggs, I. A. Baratta, C. N. Richardson and G. N. Wells. `Basix: a runtime finite element basis evaluation library'. In: *Journal of Open Source Software* 7.73 (2022), p. 3982 (cit. on p. 6).

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul et al. `Cilk: An Efficient Multithreaded Runtime System'. In: *SIGPLAN Not.* 30.8 (Aug. 1995), pp. 207–216 (cit. on p. 8).

[7] Lamport. `How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs'. In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 690–691 (cit. on p. 8).

[8] Samuel Thibault. `On Runtime Systems for Task-based Programming on Heterogeneous Platforms'. Habilitation à diriger des recherches. Université de Bordeaux, Dec. 2018 (cit. on p. 10).

[9] Cédric Augonnet, Samuel Thibault, Raymond Namyst and Pierre-André Wacrenier. 'StarPU: a unified platform for task scheduling on heterogeneous multicore architectures'. In: *Concurrency and Computation: Practice and Experience*. Euro-Par 2009 best papers 23.2 (2011), pp. 187–198 (cit. on p. 10).

[10] James D. Trotter, Xing Cai and Simon W. Funke. 'On Memory Traffic and Optimisations for Low-Order Finite Element Assembly Algorithms on Multi-Core CPUs'. In: *ACM Trans. Math. Softw.* 48.2 (May 2022) (cit. on p. 33).