Parallelizing Finite Element Methods Assembly for Unstructured Meshes with D&C

Loïc THÉBAULT [a,1], Eric PETIT [a], Marc TCHIBOUKDJIAN [c], Quang DINH [b], and William JALBY [a,c]

[a] *PRISM - University of Versailles, France*
[b] *Dassault Aviation, Saint-Cloud, France*
[c] *Exascale Computing Research, France*

Abstract. Current algorithms and runtimes struggle to scale to a large number of cores and show a poor parallel efficiency on current HPC machines. Pure MPI codes waste IO, memory and communication. HPC users have to explore new paradigms to make an efficient use of the shared resources, such as hybrid MPI + threads. In this paper we propose and evaluate a Divide & Conquer, D&C, approach for the parallelization on a shared memory system of an unstructured mesh assembly into a sparse matrix. We compare D&C to the classic MPI domain decomposition and a state-of-the-art coloring approach. Our target application is an industrial CFD application developed by Dassault Aviation. Our implementation is based on the versatile Cilk runtime and standard MPI. The original Fortran code has been modified with minimum intrusion into the original pure MPI application. Preliminary results on the finite element assembly of the application are encouraging and show good locality and scalability characteristics and already improve the performance by 14% on a twelve cores Intel Xeon X5650 NUMA node.

Keywords. Divide and Conquer, Task, Cilk, Mesh Partitioning, CFD, FEM Assembly

1. Introduction

Current algorithms and runtimes struggle to scale to a large number of cores and show a poor parallel efficiency. The increasing number of cores and parallel units implies a lower memory per core, higher requirement for concurrency, higher coherency traffic and higher cost for coherency protocol. It results in a severe challenge for performance scalability. The manycore accelerators, such as the Intel Xeon Phi, exacerbate these issues. HPC users have to explore new paradigms.

In order to mitigate the node scalability issues, users can modify their application using hybrid MPI + threads to take advantage of the full topology of the machine and enhance the data and synchronization locality. In this paper, we propose and evaluate a new parallelization strategy based on the Divide and Conquer, D&C, principle [1], on unstructured problem. Our implementation relies on a task based runtime, Cilk [2].

[1]Corresponding Author E-mail: loic.thebault@prism.uvsq.fr

We demonstrate the potential of the D&C approach on the assembly step for Finite Element Methods, FEM. This step builds a matrix describing the linear system of equations from the mesh. The values in the sparse matrix are the result of the reduction operation on the neighboring elements. Due to the irregularity of the structure and the serialization of the reduction, achieving an efficient parallelization is challenging. The state-of-the-art approach for shared memory multicores parallelization is the coloring method. Recent works published on parallelizing FEM assembly codes are designed for GPUs [3,4], but can not be efficiently applied to shared memory systems of multicores.

We apply the D&C strategy to parallelize the finite element assembly of an industrial fluid dynamics code from Dassault Aviation. This code computes the effect of mesh deformation to optimize airplane structure. Our results outperform the original pure MPI implementation and the coloring method for shared memory parallelization and vectorization.

The paper is structured as follows:

- Section 2 presents the finite element assembly in applications using the coloring approach and our D&C proposal using Cilk and MPI.
- Section 3 presents related works on assembly of finite elements and different parallelization strategies.
- Section 4 presents the experimental setups and results of the FEM assembly parallelization of a Dassault Aviation application, called DEFMESH.
- Section 5 concludes and presents our future work.

2. FEM Assembly

Finite element assembly is common in most scientific applications using finite element methods. A simple FEM assembly illustration of a 2D regular mesh is presented in figure 1 with the associated sparse matrix. The non-zero values of the matrix corresponds to the edges of the mesh. These values are the reduction of neighboring elements contribution. In figure 1, $(N1, N2) = E1 + E2$. This matrix is sparse and symmetrical, thus we only store the non-zero values in the lower triangle. In 3D, the number of neighboring elements can be arbitrarily large.

To create parallelism, we must build independent domains to execute in parallel and synchronize them to handle shared edges. We consider two different approaches, a state-of-the-art coloring approach and our proposal of D&C strategy based on a topological recursive bisection.

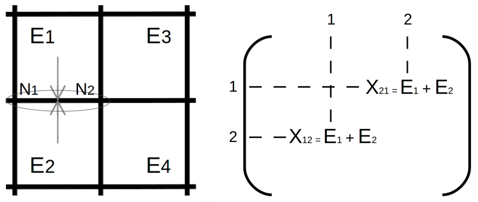
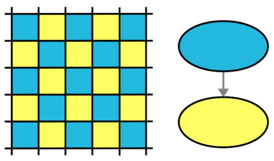


Figure 1. 2D FEM assembly example

2.1. *Coloring*

Coloring avoids race conditions by assigning a different color to the elements sharing an edge. It was originally targeting vector machines. Indeed elements of a same color do not contribute to the same edges and thus can be executed in parallel. Determining a minimal coloring is NP-complete, but creating an efficient coloring with a larger number of colors using heuristic is well-known in literature [5]. A pseudo-code and a simple 2D coloring example are given in figure 2.



(a) Coloring approach

```
NodeToElem = build_node_element (ElemToNode)
ElemToElem = build_element_element (NodeToElem)
ElemToColor = build_element_color (ElemToElem)
ColorToElem = build_color_element (ElemToColor)

For each color C in colors
  Parallel_For all elements E in C
    compute_element_contribution (E)
```

(b) Coloring pseudo-code

Figure 2.

Sorting the elements by color improves the spatial locality and enables vectorization. However, there is only one update per edge and per color. Furthermore each color covers the whole domain. For large domains, there will be no edge locality in cache between colors.

2.2. *Divide & Conquer*

The main idea of the D&C approach for shared memory parallelization is to enable task level parallelism while preserving good data locality and minimizing synchronization cost. When the size of the domain increases, we increase the number of tasks and threads instead of increasing the number of MPI domains and the amount of communication. This way, we take benefit of replacing communication by data sharing.

The rational is to recursively divide the work in two or more independent tasks and synchronize these tasks locally. This recursive approach has many advantages. First the recursive sharing naturally exposes high concurrency. As long as the data-set is large enough, it is possible to produce a deeper recursive tree to get more concurrency and therefore match the higher requirement of manycore systems. Furthermore, synchronizations are local: only nodes of a same parent in the recursive tree have to be synchronized. Finally, D&C approach improves the data locality by reordering the data as shown in section 2.2.1.

On a small number of cores, the scalability of the D&C parallelization is expected to be equivalent to the pure MPI. However, for a larger number of cores, D&C should continue scaling while pure MPI not. Furthermore, in the D&C approach, the code will benefit from the higher locality and reuse of elements. Therefore all the versions using the reordering should outperform the original code.

D&C consists of two parts described in following subsections. The first part is the recursive decomposition of each MPI domain and the second is the recursive execution of the FEM assembly step using Cilk.

2.2.1. Recursive Bisection

D&C is based on topological recursive bisections of the mesh. As illustrated in figure 3, the left and right sub-domains created by these bisections do not share any element and can be executed in parallel. However, the separator elements in the middle have nodes on both sides and must be processed after left and right sub-domains.

The load balancing between the sub-domains is important since it influences the depth of the recursive tree. A balanced tree will minimize the maximum depth and thus the number of synchronization on the critical path. Therefore, it is important to use a good partitioner to build equal sub-domains. In this study, we use the METIS graph partitioner [6].

In order to increase the data locality, we permute node and element arrays. As a result, nodes and elements can be consecutive in memory inside each sub-domain, improving intra-task locality. Secondly, since the tasks distribution is following the recursive domain decomposition, we store neighboring sub-domains and associated separator contiguously to improve locality between tasks. These permutations are applied to all MPI domains, therefore we renumber the nodes on the frontier so that exchanged data are consistent.

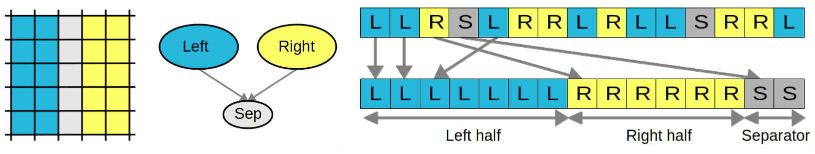
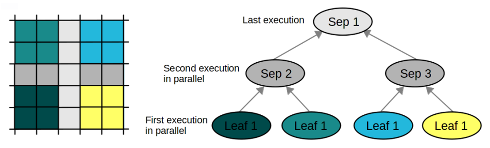


Figure 3. D&C approach

The proposed approach is then applied recursively to all sub-domains, providing a large amount of parallelism. As illustrated in figure 4, each leaf of the resulting tree is associated with an independent Cilk task which executes the FEM assembly process on its sub-domain. In order to optimize the locality with permutations whithout multiplying tasks, it is possible to stop the recursion before reaching the leaves.

The partitioning is topological: cuts are done on edges rather than on geometrical coordinates. This allows to only compute sub-domains once for a mesh and it is independent from the rest of the computation. Therefore, the partitioning is precomputed and the associated permutation is stored with the mesh. During the run, the application needs to apply the precomputed permutation before executing the recursive FEM assembly.



(a) D&C approach

```
function compute (subdomain)
  if Node is not a leaf
    spawn compute (subdomain.left)
    compute (subdomain.right)
    synchronize
    compute (subdomain.sep)
  else
    FEM_assembly (subdomain)
end
```

(b) D&C pseudo-code

Figure 4. D&C recursive execution

2.2.2. *Cilk Implementation*

Cilk [2] is a task based runtime originally developed by Frigo, Leiserson and Randal at MIT and now supported by Intel. It allows users to `spawn` and `sync` many parallel tasks. All these tasks are organized in queues and executed by *worker* threads. For load balancing, Cilk runtime uses a work-stealing scheduler. When a *worker* completes its queue, it can steal additional tasks from slower *workers*.

Once the recursive task tree as shown in figure 4 is constructed, the Cilk implementation is straightforward. Following the pseudo-code in figure 4, left and right subdomains of each node are split in two tasks and executed in parallel, as long as the recursion has not reached a leaf (domain size condition). These tasks have to be completed before launching the separator computation. When the recursion reaches the leaves, the original sequential FEM assembly routine is executed on their sub-domains.

When all elements are accessed in a regular loop, there is no need to use a recursive task tree. In this case, similarly to OpenMP, Cilk provides an easy way to parallelize loops using the `cilk_for` keyword. By substituting the original `for` with the `cilk_for` keyword, iterations of the loop are split recursively in several parallel tasks. For the DEFMESH application, we observe a better scalability with the `cilk_for` than with OpenMP `parallel for` pragma.

3. Related Work

Many choices are available to complete FEM assembly in parallel. All methods have their advantages and their limitations, depending for instance on the target architecture or the mesh size. Recent studies [3,4] investigate different ways to compute FEM assembly on CPUs and GPUs.

A first method, presented in [3], consists in creating parallelism between all the non-zero entries of the system of equations. This method has a very fine grain thread parallelism and thus can be applied only to GPUs. It stores, in parallel, the contributions from all elements in the global or local memory and then reduce in parallel these contributions to compute the non-zero values. By arranging the data layout, this method can provide good performance on GPUs. However, it suffers from bad load balancing since the number of neighboring elements used to compute non-zero values may vary. Due to the fast growing number of local copies, the approach is limited to relatively small test cases. Another variant consists in using one thread for the assembly of one non-zero value at a time. However, this approach leads to redundant computations of element contributions.

A second method [3] consists in assembling by mesh element. One thread is responsible for the whole assembly of one element at a time. It can be applied to both GPUs and CPUs. A coloring approach [7,5] can be used to launch several threads in parallel on independent elements. This way, each thread has a good load balance per color. The load balancing between colors has no influence since they are serialized. However, bad data locality can cause poor performance as explained is section 2.1.

Divide and Conquer paradigm and recursion in general have already been explored extensively for parallel algorithm [8]. M. Martone et al. [9] describe a matrix assembly approach for recursive sparse blocks (RSB) matrix on CPUs. Despite this format can provide good results in solver algorithms [10], according to authors results, FEM assembly on RSB matrix is memory-bandwidth bound [9]. Dongarra et al. [11] evaluate an asynchronous task based parallelism approach for well-known factorization algorithms and compared it to traditional fork-join approach. A recent investigation on hybrid programming integrating task based parallelism in MPI processes have been made [12]. The experiments done on a 1024 nodes cluster result in significant speed-up on several micro-benchmarks and strong scaling applications compared to pure MPI. However, recursive and D&C approaches are rarely employed on sparse algebra and unstructured problems because of their irregularity.

4. Experiments

To validate our approach, we apply the D&C algorithm using Cilk and MPI on the FEM assembly step of an application from Dassault Aviation called DEFMESH. We compare our results with the pure MPI original version of the FEM assembly step and with the MPI + OpenMP version using the coloring approach.

4.1. Dassault Aviation DEFMESH Application

The DEFMESH application is a computational fluid dynamics (CFD) code based on a finite element method (FEM). This volume mesh deformer is an important numerical module in Dassault Aviation aerodynamic optimization environment. It is also used in other simulations which may include surface variations of larger magnitude, such as in aero-elastic interactions or dynamics of moving bodies.

DEFMESH implements a three-dimensional elasticity-like system of equations from given surface data. These equations are solved by two different algorithms. The first one is a linear algorithm used when the magnitude of the surface data is small: the equations can be linearized into a system of linear equations. The second one is a non-linear algorithm where surface data of large magnitude are cut in a succession of small increments. The original equations are solved as a non-linear succession of linearized sub-problems, consisting of systems of linear equations.

Each linear system is described as a symmetric definite positive matrix. The systems are solved by a standard conjugate gradient (CG) algorithm.

DEFMESH implements two options for the definition of the elasticity-like operator for its system of volume equations. The first one is the Laplacian operator. It calls 3 times the CG algorithm at each linearized step. Each CG call computes the solution of a scalar system of linear equations. The second one is the Elasticity operator. This option implements the full 3D linear elasticity operator. It couples the 3 mesh coordinates, thus at each linearized step, the CG algorithm has to solve a 3 by 3 system of linear equations. The Elasticity operator permits mesh deformations of greater magnitude and smoother deformed meshes than the Laplacian operator.

The DEFMESH main kernel is decomposed in three steps. The first one is the FEM assembly, where mesh data are gathered into a CSR structure. The second step is the solver which works on this CSR structure and computes optimal displacements. The final step is the update of mesh coordinates using previously computed deformations.

We performed our experiments on an unstructured mesh from Dassault Aviation, called EIB and illustrated in the figure 5. It is composed of 6 346 108 elements and 1 079 758 nodes and it represents the displacements of a fuel tank along an airplane fuselage.

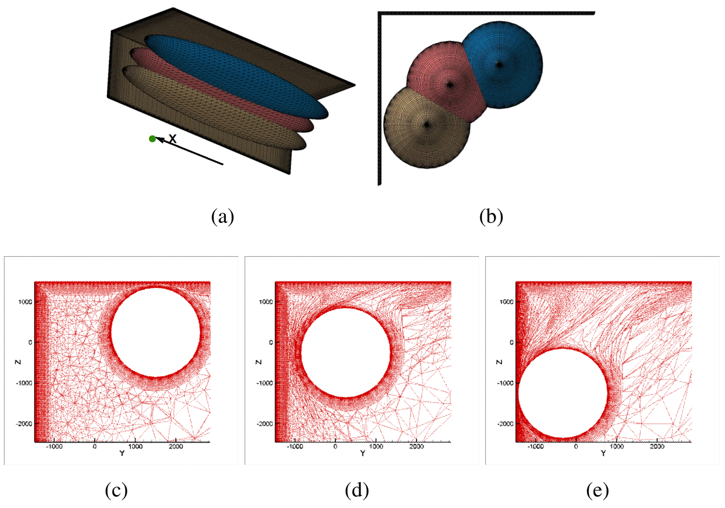


Figure 5. EIB fuel tank position optimization test case

4.2. Experimental Setup

In the following experiments, we compare three versions of the FEM assembly step from the DEFMESH application: the original one from Dassault Aviation called `Ref`, the new Divide & Conquer version called `D&C`, and the coloring version called `Color`. The `Ref` version uses MPI between the different blocks of the mesh and OpenMP to parallelize the sparse matrix-vector loop of the solver. The `D&C` version uses the recursive partitioning of each block of the mesh. In addition to MPI and OpenMP, this version also exploits a Cilk task parallelism in the FEM assembly. The `Color` version exploits OpenMP parallelism between all the elements of a same color in the FEM assembly.

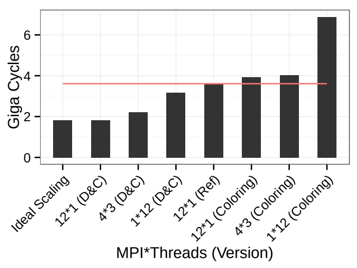
For each of these versions, we consider the Laplacian and the Elastic variants of DEFMESH. For both of them, the application needs 50 steps and the measures correspond to the average time of one iteration. We measure the FEM assembly step, where the D&C approach and the coloring have been applied. We also measure the solver to estimate the impact of the D&C data permutations on the rest of the code. We ensure that the numerical results and the number of solver iterations needed to converge are stable in all versions.

We present the results both in terms of execution times and of parallel efficiency. In all the graphics, the x axis represents the number of cores. It corresponds to the number of MPI processes multiplied by the number of threads. In the figures presenting execution time, the y axis corresponds to RDTSC cycles for the FEM assembly and to `MPI_Wtime` for the solver. Concerning efficiency, the y axis represents the parallel efficiency given by the following formula: $E_P = \frac{T_S}{P*T_P}$, where E_P is the efficiency on P processor, T_S the sequential time, and T_P the time on P processors.

For each figure, we use 1, 4, 8 and 12 MPI processes and 1 to 12 threads, OpenMP or Cilk, while the multiplication of MPI processes per thread is lower or equal to 12. This corresponds to the number of cores available. The number of Cilk threads used in the FEM assembly is equal to the number of OpenMP threads. We also make the number of METIS partitions and the number of Cilk tasks vary, but we have not observed significant changes in the results. According to the Cilk documentations, a minimum of 10 tasks per core is appropriate. Therefore we use 128 Cilk tasks, which is enough to fit in cache. We cut MPI domains in 512 partitions to improve locality.

The application has been compiled with Intel composers 13.1.1 (icc and ifort) and Intel MPI 4.1.0 and Intel Cilk+ runtime. The OpenMP affinity is set to scatter and Cilk threads are not pinned. We perform these experiments on twelve cores grouped in two sockets of 6 cores Intel Xeon X5650 clocked at 2.67 GHz. Each core has its own L1 (32KB) and L2 (256KB) caches and a shared L3 cache of 12MB. The main memory is divided in two NUMA nodes of 16GB.

4.3. FEM Assembly results



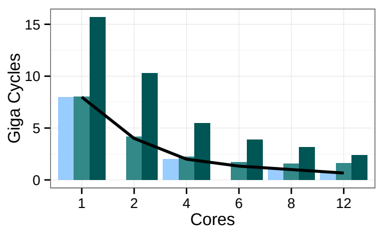
(a) All elasticity versions on 12 cores

Figure 6.

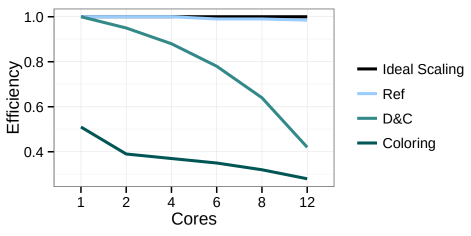
As we can see in figures 7(a) and 7(b), the `Color` version is the slowest one. The `D&C` version is very close to the `Ref` version, using pure MPI, but it starts to decrease after 6 cores. This is due to the remaining sequential part of the `D&C` code and the low intensity work in the Laplacian version. Indeed, separator elements are not parallelized and can represent an important proportion of the computation. According to the Amdahl's law, by increasing the number of cores, the proportion of the sequential part increases. This is confirmed on figure 7(b) showing the parallel efficiency.

In the Elasticity variant, illustrated in figures 7(c) and 7(d), the `D&C` version becomes equivalent to `Ref`. Since the amount of work of the FEM assembly step is larger than in the Laplacian variant, the proportion of sequential computation, including the separators, is lower. In this case the scalability improves. Furthermore, contrary to the `Ref` version, the data always fits in L2 cache with the `D&C` version. Unlike the original version that makes irregular accesses to the cells of the matrix, the `D&C` version works on matrix cells packed contiguously in small sub-matrix corresponding to the Cilk tasks. For any problem size, with `D&C`, we can increase the number of partitions so that that the tasks working-set will always fit in cache. As it can be seen from figure 7(c), the `Color` version is again the slowest.

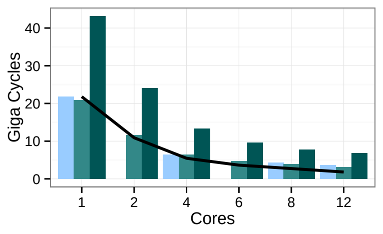
These first results are encouraging and we can reasonably expect to match the ideal scaling when the separators will be parallelized.



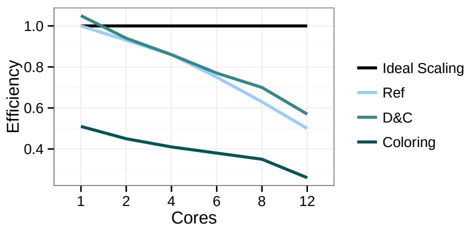
(a) Laplacian execution time



(b) Laplacian parallel efficiency



(c) Elasticity execution time

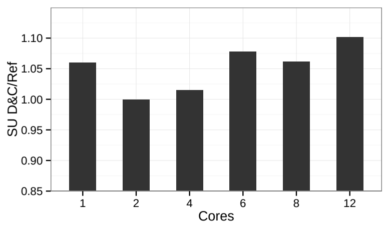


(d) Elasticity parallel efficiency

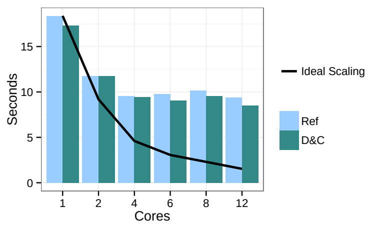
Figure 7. Assembly measures

4.4. Solver

We did not modify the solver part. However, we observe in figure 8 approximately 6% speed-up for the `D&C` version compared to the `Ref` version. This improvement is due to the better data locality enabled by the permutations explained in section 2.2.1. To show the impact of locality, we randomly permute the data and observe a significant degradation of performance. This result shows that the initial data already expose good locality. Moreover, the L3 cache misses occurring in the solver are 23% lower with `D&C` than with `Ref`.



(a) D&C solver speedup over Ref



(b) Solver execution time

Figure 8. Solver execution time and speed-up using D&C

5. Conclusion and Future Work

FEM assembly is the first step of many applications, such as seismic simulation, metal forming, or crash test simulations. Depending on the application, the FEM assembly can represent more than 80% of the execution time. In the Dassault Aviation DEFMESH application, the current D&C implementation is already faster than the pure MPI original version and overpasses in performance and scalability the state-of-the-art coloring approach. Even without exploiting the thread level parallelism, the D&C approach significantly improves the locality, the scalability, and the execution time. The pure MPI version with D&C has a perfect strong scaling for our experimental setup.

As a future work, we plan to parallelize the separator in the D&C FEM assembly and experiment on a larger number of cores. We strongly believe that D&C will provide good performances on new manycores such as Xeon Phi. Extension will focus on D&C friendly data structure definition and apply D&C on other parts of the FEM application.

Acknowledgments

This work has been supported by the ITEA2 project H4H, the French Ministry for Economy, Industry, and Employment, CEA, GENCI, Intel and UVSQ. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the CEA, GENCI, Intel, Dassault Aviation or UVSQ.

References

[1] Guillet Thomas and Tchiboukdjian Marc. Scalable and composable shared memory parallelism with tasks for multicore and manycore. http://www.teratec.eu/library/pdf/forum/2012/presentations/A2_05_FTeratec_2012_Tchiboukdjian_Guillet.pdf, 2012. Accessed: 2013-Jul-08.

[2] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[3] Cris Cecka, Adrian J Lew, and Eric Darve. Assembly of finite element methods on graphics processors. *International journal for numerical methods in engineering*, 85(5):640–669, 2011.

[4] GR Markall, A Slemmer, DA Ham, PHJ Kelly, CD Cantwell, and SJ Sherwin. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids*, 71(1):80–97, 2013.

[5] Charbel Farhat and Luis Crivelli. A general approach to nonlinear fe computations on shared-memory multiprocessors. *Computer Methods in Applied Mechanics and Engineering*, 72(2):153–171, 1989.

[6] George Karypis and Vipin Kumar. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[7] D. Komatitsch, D. Michea, and G. Erlebacher. Porting a high-order finite-element earthquake modeling application to nvidia graphics card using cuda. *Journal of Parallel and Distributed Computing*, 69(5):451–460, 2009.

[8] Ellis Horowitz and Alessandro Zorat. Divide-and-conquer for parallel processing. *Computers, IEEE Transactions on*, 100(6):582–585, 1983.

[9] Michele Martone, Salvatore Filippone, Salvatore Tucci, and Marcin Paprzycki. Assembling recursively stored sparse matrices. In *Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on*, pages 317–325. IEEE, 2010.

[10] Michele Martone, Salvatore Filippone, Pawel Gepner, Marcin Paprzycki, and Salvatore Tucci. Use of hybrid recursive CSR/COO data structures in sparse matrices-vector multiplication. In *International Multiconference on Computer Science and Information Technology - IMCSIT*, pages 327–335, 2010.

[11] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[12] Sanjay Chatterjee, Sagnak Tasırlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. Integrating asynchronous task parallelism with mpi. *Department of Computer Science, Rice University, Technical Report TR12-07*, 2013.