KYBERNETIKA — VOLUME *46* (2010), NUMBER 2, PAGES 316–327

HYBRID PARALLELIZATION OF AN ADAPTIVE FINITE ELEMENT CODE

Axel Voigt and Thomas Witkowski

We present a hybrid OpenMP/MPI parallelization of the finite element method that is suitable to make use of modern high performance computers. These are usually built from a large bulk of multi-core systems connected by a fast network. Our parallelization method is based firstly on domain decomposition to divide the large problem into small chunks. Each of them is then solved on a multi-core system using parallel assembling, solution and error estimation. To make domain decomposition for both, the large problem and the smaller sub-problems, sufficiently fast we make use of a hierarchical mesh structure. The partitioning is done on a coarser mesh level, resulting in a very fast method that shows good computational balancing results. Numerical experiments show that both parallelization methods achieve good scalability in computing solution of nonlinear, time dependent, higher order PDEs on large domains. The parallelization is realized in the adaptive finite element software AMDiS.

Keywords: adaptive finite elements, parallelization, OpenMP, MPI

Classification: 65M60, 65Y05

1. INTRODUCTION

Nowadays, high performance computers are built together from multi-core architectures. Most of these computers have at least dual or quad-core CPUs, and the number of cores per CPU will increase over the next years. To make use of the whole performance that is provided by high performance computers, one has to parallelize software in two ways. First, the classical parallelization using domain decomposition and the message passing interface (MPI) must be used to partition the big computational domain into smaller parts. Each of these sub-problems is then computed on one CPU. The algorithms, which compute on the sub-problems, must be further parallelized based on shared memory concepts, e. g., OpenMP or PThread, to make use of the multiple cores of each CPU. To parallelize scientific software in a hybrid way, such that it makes use of multiple CPUs, i. e. multi-process parallelization, and of multi cores, i. e. multi-threading parallelization, is a challenging task.

Our concept of multi-process parallelization is not new and based on textbook techniques. We therefore concentrate on multi-threading parallelization for architectures with shared memory and an arbitrary number of cores. Our approach makes use of a multilevel mesh structure, has an efficiency of more than 80 %, as long as the memory bandwidth is not a limiting factor, and can make use of an arbitrary large number of cores. This makes it applicable not only for the current computer architectures, but also for future hardware concepts with dozens and hundred of cores per CPU.

Most multi-core parallelization concepts for finite element code consider solving the discrete problem only. In the 1d and 2d cases, this is also the most time consuming part with more than 90 % of the overall runtime. When going to the 3d case with millions of elements and highly adaptive meshes, assembling the matrices and vectors, and error estimation take a significant time of the simulation. In our 3d simulations, assembling and error estimation take up to 50 % of the overall solution time in sequential computations.

In this work, we consider only the parallelization of assembling the stiffness matrix and error estimation. Parallel direct and iterative solvers for the resulting large sparse matrices are considered, e. g., in the work of Davis [3] or Kotakemori and Hasegawa [7].

The parallelization concepts presented here are implemented in AMDiS (Adaptive MultiDimensional Simulations), a finite element toolbox for the solution of systems of partial differential equations (PDEs) that is written in C++. Problem formulations can be done on a high level of abstraction in a dimension independent way. Numerical issues are kept away from the user as far as possible. More information about AMDiS may be obtained from Vey and Voigt [14]. To parallelize the sequential code, we use the OpenMP programming interface for multi-threading parallelization and MPI for multi-process parallelization. A different parallel concept based on domain covering meshes has been considered by Vey and Voigt [13].

The remainder of this paper is organized as follows: Section 2 shortly describes adaptivity and the adaptive mesh structure used in AMDiS. In Section 3, we describe the multi-threading, and in Section 4 the multi-process parallelization concepts. Section 5 presents some numerical results to show the feasible speedups of our parallelization concepts on illustrating examples. In Section 6, we conclude our results and give an outlook for further research on this topic.

2. ADAPTIVE MESHES

Adaptivity is a standard technique in finite element codes to solve PDEs with a given error tolerance and with as least computing resources as possible. In AMDiS, the computation starts on a *macro mesh*, which is defined by a mesh file and a number of global refinements, both to be specified by the user. From this first mesh, a linear equation system is build (*assembling*) and solved with either an iterative or direct solver. Using this solution, the global error is approximated by local element-wise indicators (*error estimation*). If the global error estimation exceeds a given error tolerance, those elements with a high local error indicator are refined. In regions with a very low local error indicator, the mesh may be coarsened. The whole process, assembling, solution, error estimation and mesh adaption, is repeated, until a given error tolerance is achieved.

AMDiS meshes consist of simplicial elements which are lines in 1d, triangles in 2d and tetrahedrons in 3d. If an element has to be refined during the adaption

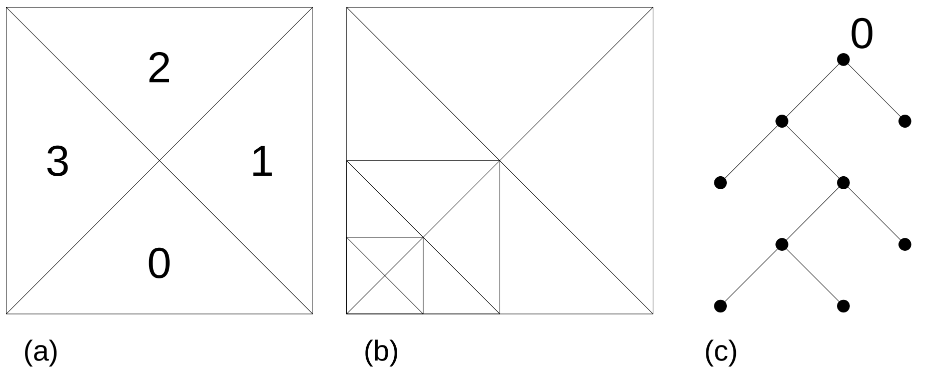

Fig. 1. (a) 2d macro mesh (b) some refinements of it (c) binary tree of macro element 0.

loop, it will be bisected into two elements of the same dimension. The refinement algorithm is described by Schmidt and Siebert [11] in more detail. The two new elements are called children of the original parent element. For each element of the macro mesh, a binary tree arises during adaption. In Figure 1, a triangular macro mesh consisting of four elements (a), some refinements of this macro triangulation (b), and the corresponding binary tree for macro element 0 (c) are shown. The representation of the mesh as a set of macro elements and the binary trees is the basis of our multilevel approach for parallelization. Using binary trees for mesh representation it is possible to traverse a mesh on different levels very efficiently. The coarsest level is the *macro level*. The finest level, also called the *leaf level*, represents the final mesh.

3. MULTI–THREADING PARALLELIZATION

The general idea of our parallelization approach is based on the fact that assembling and error estimation are operations that may be computed in parallel on different elements. The global error estimation is the sum of all local element-wise indicators. Therefore, it is possible to partition the set of all elements into n parts and assign each part to one core. Then each core computes the error estimation for its elements only. The final result is the sum of all locally computed error estimates. This addition can be done only sequentially, but its runtime is negligible in contrast to the parallel computation done before.

Parallel assembling of the stiffness matrix works in the same way than error estimation. The stiffness matrix is the result of adding all element matrices, which may be also computed independently from each other. Hereby, each core computes a private stiffness matrix for the elements that were assigned to it. At the end of the parallel computation, all private stiffness matrices have to be added to the global stiffness matrix. This operation can only be done in a sequential way to circumvent race conditions. In contrast to the addition of the local error estimation,

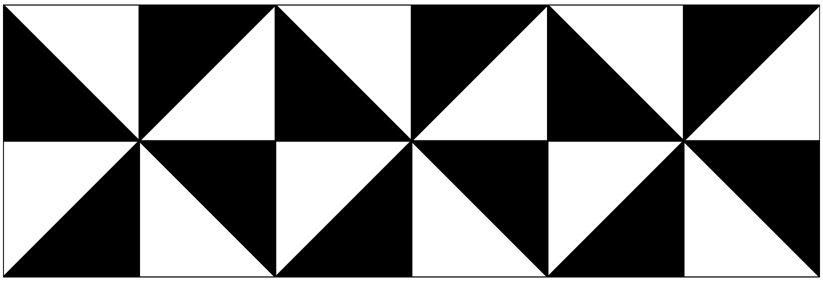

Fig. 2. A mesh with 24 macro elements is partitioned for two cores

this operation is more costly and cannot be negligible in the same way. For a small number of cores, we show in the Section 5 that the overall time to add all local stiffness matrices is quite short compared to the time for assembling them.

3.1. Parallel mesh traverse

The efficiency of a parallel algorithm for assembling and error estimation is based on a fast and almost uniform distribution of all elements to the available cores. A widely used approach is to use graph partitioning algorithms, e. g., ParMETiS [10], to partition a graph corresponding to the mesh into nearly equal sub-trees. Especially for large and adaptive meshes partitioning in this way takes more time than the operation to be done on the mesh. To make partitioning as fast as possible, we make use of the hierarchical mesh representation explained in the section before. Instead of partitioning the mesh on the leaf level, it is partitioned on the macro level.

The mesh partitioning is implemented in a “virtual” way, i. e. the operations that are performed on the mesh do not know anything about the mesh partitioning. The mesh traverse can be started either in a sequential or in a parallel way. In the later case, the current thread is forked to n threads. Each of them traverses only the corresponding sub-mesh and calls the operation to be done only on the private leaf elements. Using the parallel mesh traverse, all element-wise algorithms can be executed in parallel without changing the code:

```
#ifdef _OPENMP
  TraverseParallelMesh traverse;
#else
  TraverseMesh traverse;
#endif

do {
  foo(traverse.getElement());
} while (traverse.next());
```

If AMDiS is compiled with a compiler supporting OpenMP, the parallel mesh traversed is used automatically. In the case the compiler does not support OpenMP, the sequential mesh traverse is used. Because both classes implement the same interface, the code using the mesh traverse is the same in both cases.

Up to now, the only open question is how to partition the macro elements in a cheap way and to achieve a good load balancing, i. e. the number of leaf elements should be nearby equal on all cores. We have made the observation, that in almost all practical simulations the number of leaf elements is almost the same in all neighbouring macro elements. Under this assumption, no complicated partitioning algorithm has to be used. The parallel mesh traverse assigns all macro elements with the index i to core j, if $i \mod p_{\max} = j$, with $p_{\max}$ the number of all available cores. If the number of macro elements is much higher than the number of available cores, i. e. at least two magnitudes, this ensures a good partition of macro elements too the cores. A high number of macro elements is not a hard restriction. When using complex geometries, we need many macro elements to get a good triangulation of the geometry. Furthermore, we assume that when thinking about parallelization in large, the size of the geometry scales with the number of availble computing nodes. When using simple geometries, we still prefer to pre-refine the domain to enlarge the number of macro elements, altough the lower bound on the number of required macro elements to achieve a proper load balance is in the order of the number of cores that are used. But this way the load balancing is sufficient in most practical computation.

An example for this kind of partitioning is shown in Figure 2 for a 2d mesh with 24 macro elements. The advantage of the partitioning is that the sub-meshes are not continuous. This improves the leaf element distribution to the cores, if the mesh has some very local refinements, for example due to singularities in the solution.

3.2. Assembling

In the sequential version of AMDiS, for assembling the stiffness matrix and the load vector, the mesh is traversed on its leaf level and an assembling function is called for every element. To parallelize this loop, we use the parallel mesh traverse algorithm described above. Every thread creates a private matrix and vector and assembles its sub-mesh on them. Because the assembling procedure takes the same amount of time on every element, independently of the elements size or location, the parallel assembling on the sub-meshes scales perfectly, as long as all elements are perfectly partitioned to the available cores.

After all threads have assembled on their sub-meshes, the private matrices and vectors have to be added to the public stiffness matrix and load vector. This operation must be performed in a sequential way to avoid race conditions. For a small number of cores we have implemented it using OpenMP's critical section. That means, that all threads add their matrices to the global matrix one after the other. Up to 8 cores, the private to public matrix additions take less than 10 % of the assembling time. For higher number of cores, the percentage of the addition time will increase and therefore, the parallel efficiency will go down. To circumvent the problem, we have implemented a hierarchical addition of matrices. Hereby, always two threads are allowed to add their private matrices. Figure 3 shows an example

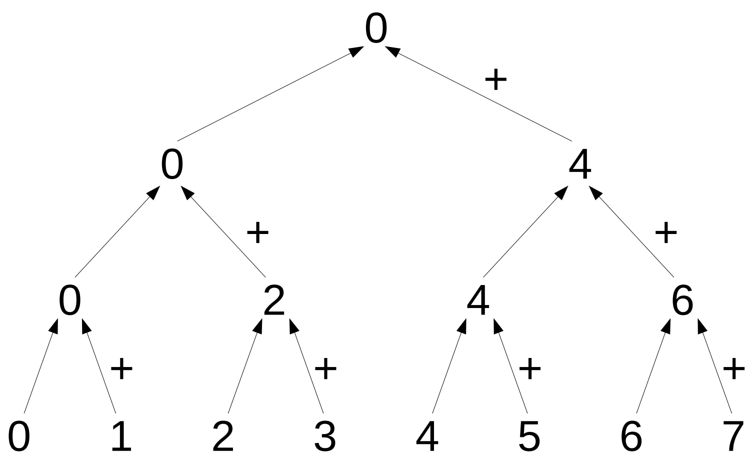

Fig. 3. Hierarchical matrix addition.

for eight threads. In the first step, threads 1 adds its private matrix to the matrix of thread 0, thread 3 to thread 2, and so on. All these additions on this level may be done in parallel. In the next step, thread 2 adds its matrix to thread 0 and thread 6 to thread 4. In the last step, the global stiffness matrix is contained in the master thread 0. It is clear, that the hierarchical matrix addition scales not linear, but instead logarithmically in the number of cores which leads to higher memory usage due to multiple copies of one matrix. In our simulations this not a cruical point because highly sparse matrices are used.

3.3. Error estimation

Parallelizing the error estimation loop is done in a very similar way to the assembling process. The mesh is traversed in parallel by all available cores. Each core evaluates some local estimators on all elements of its sub-mesh. Eventually, the result, which is just a floating point value, is added by all cores to a global estimation. Theoretically, using this approach the speedup and efficiency must scale linearly in the number of cores.

4. MULTI–PROCESS PARALLELIZATION

To make use of hundreds and thousands of computers connected by a fast network, we have parallelized AMDiS based on domain decomposition and distributed linear algebra. To avoid a bottleneck when partitioning large domain, we use the same idea as in multi-threading parallelization. The mesh is not partitioned on the leaf level, but on the macro level. Furthermore, each element on the macro level is weighted with the number of its leaves. Using ParMETIS [10], the partitioning of the macro mesh can be done very fast with a very good distribution of the elements to the

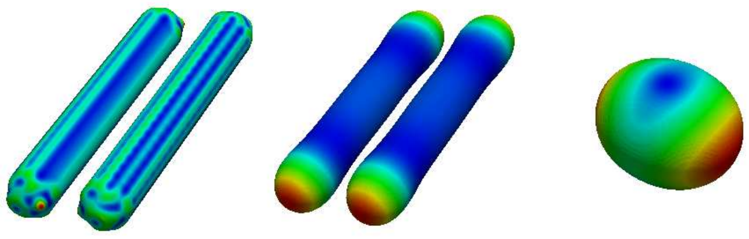

Fig. 4. Solution of the Cahn–Hilliard equation at $t = 0.003$, $t = 13.27$ and $t = 124.16$. The color indicates the local curvature of the solution.

processes.

Once the mesh is partitioned, AMDiS assigns to each degree of freedom at interior boundaries, i. e. at boundaries between process' sub-meshes, the owner of these degree of freedom. This is necessary to make the global indices of the degrees of freedom unique. Using the global indices for degrees of freedom, AMDiS uses PETSc [2] to assemble global matrices and vectors and to solve the linear system of equations.

5. NUMERICAL RESULTS

5.1. Multi-threading parallelization

As an example we use a phase-field model for surface diffusion, which has applications in nanotechnology [8]. The model follows from a free energy

$$\mathcal{F} = \int \frac{\delta}{2}|\nabla\phi|^2 + \frac{1}{\delta}f(\phi)\,\mathrm{d}\Omega \tag{1}$$

with $f(\phi) = 18\phi^2(1-\phi)^2$ and $\delta > 0$ a small parameter defining the width of the diffuse interface. The Cahn–Hilliard equation reads

$$\delta_t\phi = \nabla \cdot \left(B(\phi)\nabla\left(-\epsilon\Delta\phi + \frac{1}{\epsilon}f'(\phi)\right)\right) \tag{2}$$

with $B(\phi) = 36\phi^2(1-\phi)^2$ a degenerate mobility. We split the 4th order PDE (2) into a system of two second order equations

$$\delta_t u = \nabla \cdot (B(\phi)\nabla\mu) \tag{3}$$

$$\mu = -\epsilon\Delta\phi + \frac{1}{\epsilon}f'(\phi) \tag{4}$$

and discretize both using linear finite elements. Furthermore we use a semi-implicit time discretization, with a linearization of the double well potential

$$f'(\phi^{n+1}) \approx f''(\phi^n)\phi^{n+1} + f'(\phi^n) - f''(\phi^n)\phi^n \tag{5}$$

Details can be obtained from Rätz et al. [9]. Figure 4 shows the evolution of two nanobots which collide and evolve to a sphere.

Table 1. Runtime results for assembling.

cores	runtime [s]	matrix addition [s]	speedup	efficiency
1	6904.0	0.0 – 0 %	1.0	100 %
2	3612.2	83.67 – 2.3 %	1.91	95.5 %
4	1992.0	93.62 – 4.7 %	3.47	86.8 %
5	1856.5	102.98 – 5.5 %	3.72	74.4 %
6	1955.7	104.52 – 5.3 %	3.53	58.8 %
7	1667.2	114.18 – 6.8 %	4.14	59.1 %
8	1330.2	122.89 – 9.2 %	5.19	64.8 %

Table 2. Runtime results for error estimation.

cores	runtime [s]	speedup	efficiency
1	2452.9	1.00	100 %
2	1449.1	1.69	84.5 %
4	877.57	2.8	70.0 %
5	775.57	3.12	63.3 %
6	863.7	2.84	47.3 %
7	831.5	2.95	42.1 %
8	621.0	3.99	49.9 %

The simulation was done with adaptive timesteps and adaptive meshes. The overall number of computed timesteps is around 2500. We have used linear Lagrange finite elements, leading to meshes with up to 1.3 million degree of freedoms (DOFs) and around 4 million elements (tetrahedrons). The numerical results were done on a server with four AMD Opteron dual cores, 32 Gbyte RAM and SuSE Linux Enterprise 10.0.

Table 1 shows the runtime, the achieved speedup and the efficiency of parallel matrix assembling. The values are given for the first 50 timesteps. The third row specifies the time for adding the private matrices to the global matrix. Adding the private matrices one after the others takes less than 10 % of the overall assembling time. Using the hierarchical matrix addition does not lead to a significant fast addition for up to 8 cores. We expect this technique to be advantageous for larger number of cores per CPU.

The speedup is about 80 % for up to four cores, but breaks down for more cores. This adheres to the used hardware. Using four threads, each thread is set to one of the four CPUs. In the case of at least five threads, two threads have to share one dual core CPU. In this situation, the memory bandwidth is a limiting factor causing the worse speedup for more than four cores.

In Table 2, the runtime for error estimation of the first 50 timesteps is presented.

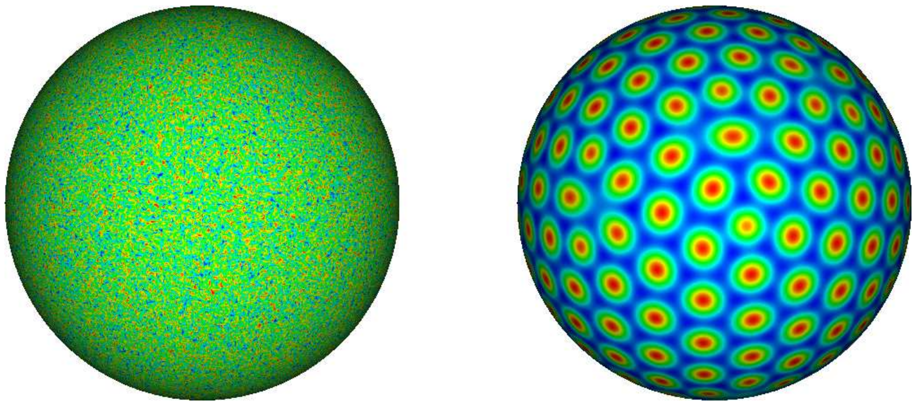

Fig. 5. Initial and final solution of the PFC equation on a sphere. The color indicates the density field.

The achieved speedup up to four cores is around 10 % worse than for assembling. This has only technical reasons, because we had to add some OpenMP barriers in our code because of some global variables. The same collapse of speedup can be observed for more than four cores, as for assembling, which again results from memory bandwith limitations.

5.2. Multi-process parallelization

The applicability and performance of our parallelization scheme is tested for a situation typical in scientific computing. We choose a nonlinear higher order PDE, the Phase Field Crystal model (PFC) which recently become popular in computational material science [5]. The PFC is a Phase Field model on atomistic scale. It can be derived as a approximation to classical density functional theory. The simplest dimensionless form of the free energy is

$$\mathcal{F} = \int \frac{1}{2}\psi \left(-\epsilon + (\Delta + 1)^2\right) \psi + \frac{1}{4}\psi^4 \, \mathrm{d}\mathbf{x} \tag{6}$$

which leads to the following conserved evolution law

$$\partial_t \psi = \Delta \left\{(-\epsilon + (1 + \Delta)^2)\psi + \psi^3\right\}, \tag{7}$$

where ψ is a rescaled density field of the underlying particles. For the connection to DFT and the relation of the involved numerical parameters to parameters of real materials, we refer to van Teeffelen et al. [12].

The resulting evolution equation is of 6th order, which already introduces some numerical difficulties. Furthermore, for the crystalline state the energy is minimized by a density field, which is peaked at the atomic positions. Due to the rapid spatial variations, fine computational grids have to be used in order to resolve the inter-atomic separation. This results in huge systems. Both aspects ask for an efficient

Table 3. Runtime results for MPI based parallelization.

CPUs	runtime [s]	speedup	efficiency	speedup	efficiency
2	16555	1.00	100 %	–	–
4	8948	1.85	92.5 %	–	–
8	3303	5.01	125.2 %	1.00	100 %
16	1766	9.37	117.1 %	1.87	93.5 %
32	853	19.4	121.2 %	3.87	96.7 %
64	415	39.89	110.8 %	7.95	99.3 %

numerical treatment of the equation on high performance computers. In order to solve the higher order PDE, we rewrite (7) as a system of three second order equations

$$v = \Delta\psi, \tag{8}$$

$$\partial_t\psi = \Delta u, \tag{9}$$

$$u = (1-\epsilon)\psi + 2\Delta\psi + \Delta v + \psi^3. \tag{10}$$

Furthermore, we use a semi-implicit time discretization, with a linearization of the derivative of the potential $(\psi^{n+1})^3 \approx 3(\psi^n)^2\psi^{n+1} - 2(\psi^n)^3$. A brief review of the finite element discretization of the PFC equation is stated in Backofen et al. [1].

For the numerical experiment we have computed the PFC equation on a surface mesh of a sphere with noise as initial condition. The equations have to be modified by replacing the Laplace operator Δ by its surface complement Δ_Γ the Laplace–Beltrami operator. Details on how to deal with such surface PDEs can be found in Vey and Voigt [14] as well as Dziuk and Elliott [4].

For discretization we have used 4th order Lagrange elements. The overall problem size is 10^6 degree of freedoms. Because of the three components of the PDE, the overall linear system consists of around $3 \cdot 10^6$ equations. The computation was done on the PC farm "deimos" at the Technical University Dresden. The specific computing nodes we used are AMD Opteron x85 processors equipped with 2 GB of memory. They are connected by an 4x Infiniband network. Table 3 shows the runtime results for computing 10 timesteps of the equation. Because we have used a globally refined mesh, the problem does not benefit from mesh adaptivity. The distributed linear system of equations is solved with PETSc's implementation of the TFQMR method with block jacobi preconditioning.

The third and fourth row show the speedup and the efficiency, respectively, which is related to the computation time of the 2 CPU results. Within the computation on two and four CPUs, there are much more cache misses than for the other computations where the problem sizes are smaller and thus fit better to the CPU's cache hierarchy. This explains the super linear speedup, when we compare the results for at least 8 CPUs to the computation with 2 CPUs. The two last rows of Table 3 show therefore the speedup and efficiency for computations on 16, 32 and 64 CPUs related to the computation time on 8 CPUs. Here we see nearly perfect speedup.

6. CONCLUSION

We have presented two parallelization techniques that make it possible to compute solutions of PDEs on large domains in 3d. Besides some technical trivia to improve the speedup of our OpenMP parallelization, we will combine both parallelization techniques together to parallelize all parts of the finite element method in both ways. The only reason why we have not done it till now is the absence of a distributed linear algebra package that makes use of both, MPI and OpenMP parallelization. In near future, the Matrix Template Library 4 [6], that we are using for sequential linear algebra, will be about to make use of both parallelization concepts.

ACKNOWLEDGEMENT

We acknowledge the provided computing resources at ZIH at TU Dresden. Axel Voigt was partly supported by DFG through Vo899/7-1 and SFB 609. Thomas Witkowski was partly supported by EU STRP 0/6447 and NSF DMR Award No. 0502737 within the project "MagDot". We thank the anonymous referees for their useful suggestions and comments.

(Received March 3, 2010)

REFERENCES

[1] R. Backofen, A. Rätz, and A. Voigt: Nucleation and growth by a phase-field crystal (PFC) model. Phil. Mag. Lett. *87* (2007), 813–820.

[2] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang: PETSc Web page. http://www.mcs.anl.gov/petsc (2009).

[3] T. A. Davis: Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method. ACM Trans. Math. Software *30* (2004), 2 196–199.

[4] G. Dziuk and C. M. Elliott: Finite elements on evolving surfaces. IMA J. Numer. Anal. *27* (2007), 262–292.

[5] K. R. Elder, M. Katakowski, M. Haataja, and M. Grant: Modeling elasticity in crystal growth. Phys. Rev. Lett. *88* (2002), 245701.

[6] P. Gottschling, D. S. Wise, and M. D. Adams: Representation-transparent matrix algorithms with scalable performance. In: ICS '07: Proc. 21st Annual Internat. Conference on Supercomputing 2007, pp. 116–125.

[7] H. Kotakemori and H. Hasegawa: Performance evaluation of a parallel iterative method library using OpenMP. In: ACM Proc. Eighth Internat. Conference on High-Performance Computing in Asia–Pacific Region 2005, pp. 432–437.

[8] B. Li, J. Lowengrub, A. Rätz, and A. Voigt: Geometric evolution laws for thin crystalline films: Modeling and numerics. Comm. Comput. Phys. *6* (2009), 433–482.

[9] A. Rätz, A. Ribalta, and A. Voigt: Surface evolution of elastically stressed films under deposition by a diffuse interface model. J. Comput. Phys. *214* (2006), 187–208.

[10] K. Schloegel, G. Karypis, and V. Kumar: Parallel static and dynamic multi-constraint graph partitioning. Concurrency and Computation: Practice and Experience *14* (2002), 3, 219–240.

[11] A. Schmidt and K. G. Siebert: Design of adaptive finite element software. (Lecture Notes in CSE *42*.) Springer, Heidelberg 2005.

[12] S. van Teeffelen, R. Backofen, A. Voigt, and H. Löwen: Derivation of the phase field crystal model for colloidal solidification. Phys. Rev. E. *79* (2009), 051404.

[13] S. Vey and A. Voigt: Adaptive full domain covering meshes for parallel finite element computations. Computing *81* (2007), 53–75.

[14] S. Vey and A. Voigt: AMDiS – adaptive multidimensional simulations. Comput. Visual Sci. *10* (2007), 57–67.

Axel Voigt, TU Dresden – Institut für Wissenschaftliches Rechnen, D-01062 Dresden. Germany.
e-mail: axel.voigt@tu-dresden.de

Thomas Witkowski, TU Dresden – Institut für Wissenschaftliches Rechnen, D-01062 Dresden. Germany.
e-mail: thomas.witkowski@tu-dresden.de