Parallelization and Optimization of Microstructure Simulations

Tobias Frodl[*], Britta Nestler[†], Michael Selzer[‡]
Faculty of Computer Science
Karlsruhe University of Applied Sciences
Moltkestr. 30
D-76133 Karlsruhe, Germany

Abstract

We describe the implementation of a parallel 3D simulator for the numerical computation of alloy solidification and microstructure formation in real metallic alloys and in other materials using the phase-field method. Naturally, simulating realistic and expedient experiments requires huge grids and thus, large amounts of memory and computation power which only parallel systems can offer. The numerical solver application is accomplished using a finite difference method on a rectangular three-dimensional grid with adaptive and parallel algorithms. In order to make the simulator simple to implement, portable and efficient, we have designed the solving algorithm to be able to adapt to a large number of even heterogeneous environments including dedicated scientific supercomputers for real-world simulations over university computer labs shared with students and standard computers for development and debugging. To support all these scenarios, several parallel computing approaches are combined for multiple dedicated computers with shared as well as disjoined memory. The simulation software uses a mix of self-implemented algorithms and protocols on top of well-established standard libraries for parallel applications such as OpenMP and MPI. Further, we developed additional optimization algorithms including intelligent timing and grid distribution, as well as compression algorithms to minimize network traffic.

1 Introduction

The characteristics of the microstructures which form during solidification of a metallic alloy play an important role for the mechanical properties of the resulting material and are hence of substantial interest to materials engineers. In these days, the research field of modeling and numerical simulation has become a powerful and valuable instrument to gain insight into the process of microstructure formation and to predict, determine and design materials with specific properties. The growth process of crystals is of inherently three-dimensional nature. The special emphases of our research work lies in the description of

[*]frto0014@hs-karlsruhe.de
[†]britta.nestler@hs-karlsruhe.de
[‡]semi0016@hs-karlsruhe.de

polycrystalline grain structures (see Fig. 1) and of phase transformations in multicomponent and multiphase systems under the consideration of mass and heat diffusion, convection, anisotropy and elasticity. The evolution process is described by a coupled set of partial differential equations of parabolic type for the physical fields such as e.g. the phase/grain states. The thermodynamically consistent derivation of the full set of equations is based on an entropy density functional and can be found in [1]. To give an impression of the type of equation and numerical problem to be solved, we give here - as a prototype - a simple example describing the growth of a single grain into an undercooled melt

$$\epsilon \frac{\partial \phi}{\partial t} = 2\epsilon\gamma \triangle \phi - f(\phi), \tag{1}$$

where ϵ is a small length scale parameter defining the thickness of the diffuse interface in the phase-field model and γ is the surface energy of the phase/grain boundary. The system variable $\phi(\vec{x}, t)$ depends on space and time and is an order parameter defining the local phase or grain state in the system. $f(\phi)$ is a nonlinear function of ϕ that may also depend on other system variables such as the the temperature $T(\vec{x}, t)$ or the concentration fields $c_i(\vec{x}, t), i = 1, \ldots, K$ for the alloy components.

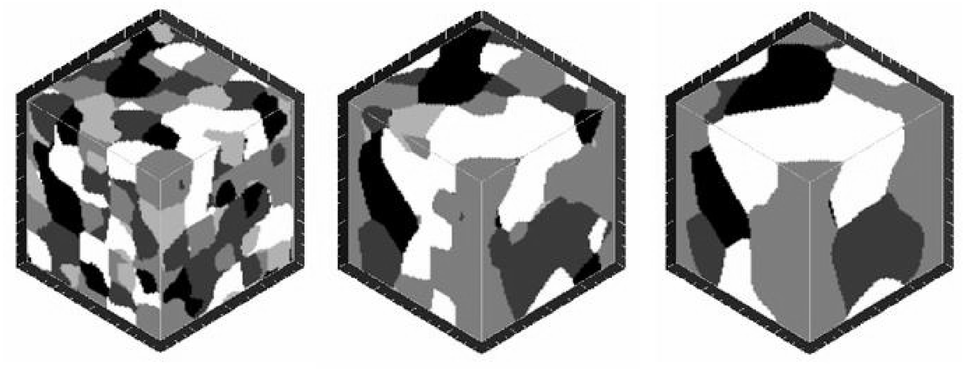


Figure 1: Simulation of the coarsening process of a polycrystalline grain structure at three subsequent times in 3D.

Due to the complexity of the multiple phase transitions and polycrystalline textures, the computation in three dimensions requires large computing power and the application of parallel programming (see also [2, 3]), adaptive finite element methods (see also [5, 6, 7]) and efficiency optimized algorithms. To solve the nonlinear partial differential equations of the type illustrated in Eq. (1) on a numerical grid of $i = 0, \ldots, N_x$, $j = 0, \ldots, N_y$, $k = 0, \ldots, N_z$ cells, we implemented a parallel finite difference algorithm for the discretized form of the differential Laplacian operator:

$$(\triangle\phi)_{i,j,k} = \frac{\phi_{i+1,j,k} - 2\phi_{i,j,k} + \phi_{i-1,j,k}}{(\delta x)^2} + \frac{\phi_{i,j+1,k} - 2\phi_{i,j,k} + \phi_{i,j-1,k}}{(\delta y)^2}$$

$$+\frac{\phi_{i,j,k+1} - 2\phi_{i,j,k} + \phi_{i,j,k-1}}{(\delta z)^2}. \quad (2)$$

We use the Euler's method to compute the time derivative at time t_{n+1} which employs first-order difference quotients:

$$\frac{\partial \phi}{\partial t} = \frac{\phi^{n+1} - \phi^n}{\delta t}. \quad (3)$$

The subscript n denotes the time level. All terms in the differential equation on the right hand side are evaluated at time t_n, so that the method is fully explicit

$$\phi^{n+1}_{i,j,k} \quad = \quad \phi^n_{i,j,k} - \delta t\Big(2\gamma(\triangle\phi)^n_{i,j,k} - \frac{1}{\epsilon} f(\phi^n_{i,j,k})\Big). \quad (4)$$

The discretized form shows that the computation of a new time step at a certain cell involves the values of the neighboring grid points.

For multiple dedicated computers with disjoined memory, the simulation grid is divided into multiple sub-grids. Each node computes one sub-grid and due to the dependencies of the numerical method on the neighboring cells, the boundaries of the sub-grid have to be exchanged in the network at each time step. Each sub-grid can be adaptively and automatically re-sized and freshly distributed during the simulation which allows optimizing the workload at the beginning of the simulation for each participating computer according to its abilities. Secondly, this process compensates for changes in utilization of each computer caused by other software running simultaneously. In addition to supporting multiple dedicated computers, the same solver application is able to utilize computers equipped with multiple processing units by assigning each processing unit an equal portion of the sub-grid and thus, introducing another stage of grid-division on a lower level. Both stages can be combined or employed separately from each other according to the amount of memory and power required for the simulation and the layout of the resources, respectively. In order to introduce parallelization efforts of this scale, several additional optimization algorithms need to be added to the software in order to keep the administration overhead of the overall simulation at a minimum and to increase the efficiency. This includes intelligent timing and grid distribution, as well as compression algorithms to minimize network traffic. Besides parallelization, several concepts such as result caching and adaptive grid resolution are implemented in the simulation code resulting in a much smaller memory-footprint compared to straight-forward textbook implementations of finite differences.

2 Parallelization

2.1 MPI Parallelization

MPI is a well developed standard, which can be found at http://www.mpi-forum.org/. It supports all necessary functionality to implement a distributed simulation program according to our agenda. MPI abstracts the underlying transport mechanism from the actual simulation model and hence speeds up the development and allows us to employ our solver on a wide range of hardware and operating systems without any modifications to the MPI-layer. All our tests were conducted with the LAM/MPI library (http://www.lam-mpi.org/)

The primary parallelization approach splits the three-dimensional simulation grid into multiple sub-grids as shown in Fig. 2. Each node gets assigned a specific part of the simulation space. To avoid adding another layer of complexity to the code, the simulation loop exclusively works on a local z-index, n_z as opposed to the global z-index N_z,used in the input and output data. By mapping the N_z to n_z for each node we are able to re-use the same code base as well as simulation description for serial as well as distributed simulations, effectively decoupling the simulation from the hardware executing it. Furthermore this approach allows us to add further optimization methods such as dynamic workload distribution as described in chapter 2.2.

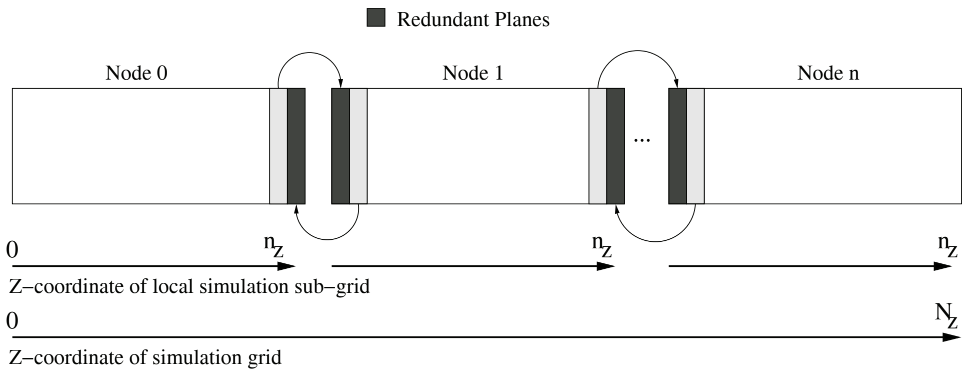
One issue that comes up with this sort of grid distribution is the absence of neighboring cells at the left- and rightmost planes of each sub-grid, which are required in order to evaluate the Laplacian $(\Delta\phi)_{i,j,k}$ as in Eq. (2). To solve this problem, we employ a method similar to the one described in [2]. Each distributed sub-grid allows room for two additional planes at its left and right side. These cells are handled and stored like regular cells while the simulation loop only takes cells $(0,0,1)$ to $(n_{xmax}, n_{ymax}, n_{zmax-1})$ into account. Again, through de-coupling the actual simulation from the data initialization and MPI-related code, we were able to keep this modification separate from the simulation code.

A MPI-based boundary exchange mechanism ensures that before each simulation time step commences, the outer planes of each node are copied into the excess space, provided by its immediate neighbor as shown in Fig. 2. The positioning of the two redundant planes now enables the algorithm to evaluate the complete simulation grid without any special conditions at the marginal cells.

One of the major disadvantages of the described boundary exchange method lies in the burst-like network traffic that is generated on data paths between the nodes whenever one time step is complete. The actual amount of traffic depends on the simulation grid's extent in the x and y dimension. Thanks to the current state of computer network technology, we have not experienced any major bottlenecks in our tests so far.

2.2 Dynamic Workload Distribution

Our first parallelization approach was based on a set of equal nodes, each responsible for a dedicated portion of the grid. This proofed to be inflexible for two reasons: The distributed storage of results which had to be concatenated after the simulation and an unbalanced distribution of the workload over the participating nodes.



Redundant Planes
Node 0
Node 1
Node n
...
0
n_z
n_z
n_z
Z–coordinate of local simulation sub–grid
0
N_z
Z–coordinate of simulation grid

Figure 2: The boundary exchange.

For some scenarios, like for instance a computing environment shared with other projects, a student computer pool or even a simulation case based on a very localized and expanding concentration, it is desirable to be able to dynamically allocate work to nodes with excess computing power and move work away from overloaded nodes. To obtain this goal a dynamic workload distribution process based on MPI has been integrated in the simulation program. For this distribution process we introduced a client/server structure with a single server implementing a central result storage and handling the distribution of workload chunks to the worker nodes.
For the sake of simplicity and due to the programmatically usually unpredictable course of simulations and thus the workload, an adaptive distribution algorithm was chosen: Initially, all nodes start with an equal share of the simulation grid. During the course of the simulations, each node w monitors the wall time t_w required to complete one time step of the allocated share of the simulation grid. Subsequently, all nodes synchronize with the server process and send their measured t_w. The server, implementing the statistical subsystem, now decreases or increases the number of assigned planes for each node according to its performance during the last time step. The newly found, optimized distribution is then passed to the nodes in form of a single scalar $c_{t+1}(n)$ denoting the number of grid-planes to be processed in the subsequent time step.
After each node has received its new assignment, the grid plane exchange process is initiated. During this step, the nodes decide which part of the currently assigned sub-grid needs to be passed to or taken over from a neighboring node. The server and thus the statistical subsystem are no longer involved. Fig. 3 shows how the process works. The variables $l(n)$ and $r(n)$ denote how many planes node n needs to be exchanged with its left $(n-1)$ and right $(n+1)$ neighbor. A positive number indicates that planes need to be accepted, a negative number indicates that planes need to be passed on. The variables $c_t(n)$ and $c_{t+1}(n)$ denote how many planes are currently in possession of node n and how many planes were assigned to the particular node for the subsequent time step. Since node 0 does not have a left neighbor,

$$l(0) = 0$$

and

$$r(0) = c_{t+1}(0) - c_t(0)$$

can be assumed. Since all planes given up by a node must be picked up by its immediate neighbor in order to keep the global grid in proper order,

$$l(n) = -r(n-1) \quad \forall \quad n \geq 1$$

needs to be fulfilled. With $l(n)$ obtained from the left neighbor and $c_{t+1}(n)$ assigned by the server process, $r(n)$ can be obtained through

$$r(n) = c_{t+1}(n) - c_t(n) - l(n),$$

which allows all nodes to compute the necessary shifts by means of subsequent passes of $r(n)$ from the node handling the leftmost sub-grid through to the node handling the rightmost sub-grid.

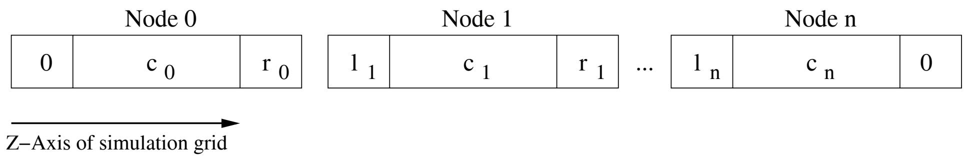


Figure 3: The grid plane exchange process.

For the actual data transfer the same functionality as for the regular MPI boundary transfer is utilized. Each plane that needs to be shifted is sent and confirmed individually by the involved nodes.

The most fundamental disadvantage of this solution is the possibility that due to a fast changing grid configuration, the increase in speed during the simulation is canceled out by the time required to transfer parts of the simulation grid between the nodes acting as workers. To overcome this problem, additional configuration parameters were introduced such as the definition of groups of workers which only perform plane exchanges within their own group. Secondly, the frequency of grid plane exchanges can be reduced to every nth time step.

2.3 OpenMP Parallelization

In addition to MPI for distributed processing, the simulation program uses OpenMP to implement another layer of parallelism below MPI which allows for utilizing the full capabilities of multi-processor systems.

OpenMP aims at shared memory parallelism. Usable in C, C++ and Fortran programming environments, OpenMP's primary aim is to provide a standardized abstraction of multi-threading libraries such as POSIX Threads or CMA Threads. It is designed to be very user friendly by providing a very limited set of commands on a high level. Thus the complexity of multi-threading on the operation system level is hidden. This allows the programmer to focus on algorithms rather than on the workings of the operating system. Furthermore, due to its abstraction from operation-system dependent multi-threading libraries, it makes software more portable to various platforms.

The concept of parallelization employed in the simulation software takes a straight-forward path: In the implementation, the three-dimensional simulation grid is regarded and stored as a single band of cells as opposed to the cuboid it represents. Without shared memory parallelization, the configuration of the subsequent time step is calculated for all cells one after another in the program's primary loop construct. If OpenMP is activated at compile time, the loop's workload is automatically split between all CPUs available to the simulation software utilizing an OpenMP loop level parallelism construct. This results in several cells being processed simultaneously.

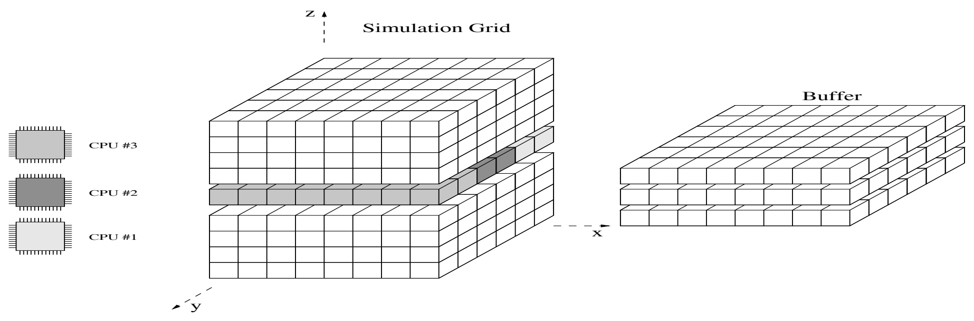


Figure 4: OpenMP parallelization approach with buffer planes.

Since the configuration of each cell in the grid depends on its six neighboring cells, the implementation must ensure that no cell is overwritten by one thread while another thread, responsible for another subset of the grid, depends on the original data. This race condition is solved by a buffering mechanism. As shown in Fig. 4, a synchronization between all participating threads occurs whenever a plane of the grid is fully solved. This condition assures that only cells in the current plane, the plane above and the plane below are accessed. These planes are copied to a separate buffer during each time step before the main calculation occurs. Additionally, this buffer is used to store temporary values required by the model equations. The construction of the buffer itself is again parallelized with OpenMP.

In conclusion, most of the shared memory parallelization is based on just two OpenMP constructs and a software design with parallelization in mind. All other aspects, such as probing the CPUs, balancing the workload and assigning grid chunks are handled transparently by the library back-end. To allow the software to run on various hardware configurations both MPI and OpenMP functionality can be excluded from the compiled program by means of two simple flags passed to the compilation script without the need for multiple code-bases.

3 Results and Conclusion

The simulation software, in particular the OpenMP-parallelized code has been subjected to several performance tests. So far, no in-depth testing has been performed on the MPI code with enabled dynamic workload distribution since this feature is still experimental and not yet fully stable. All tests were performed on a scientific computing cluster owned by the University of Karlsruhe. This computer features 112 nodes with two to eight 200MHz Power3 CPUs each and 1GB of memory per CPU.

Subject of our tests was measuring run-time on several three-dimensional grids with varying size and initial configuration. All tests were performed on the serial version as well as on the OpenMP-parallelized version using one, two, four and eight CPUs. In addition, a limited number of similar tests was performed on the MPI version.

Fig. 5 shows an extract of our OpenMP based test results. For this particular test, an alloy consisting of one component and two phases was used. The left diagram shows a simulation grid of $100 \times 100 \times 100$ cells. The diagram on the right shows a grid of $200 \times 200 \times 200$ cells. Both tests were conducted with one, two, four and eight CPUs.

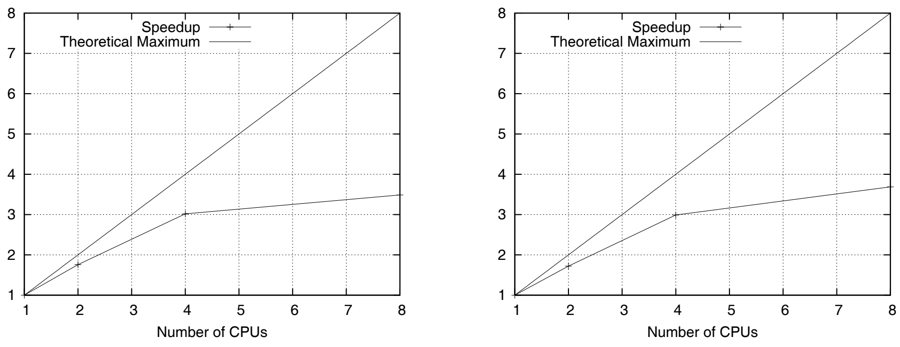


Figure 5: OpenMP Test Results.

Results show that the OpenMP-parallelized simulation software runs most efficiently on up to four CPUs. Depending on the size of the simulated cuboid and the filling, a substantial decrease in speed-up is usually experienced when more than four processors are utilized due to small number of cells assigned to each CPU before a synchronization occurs. So far, none of our tests exceeded cuboid dimensions of more than $200 \times 200 \times 200$ grid points. In recent times, our project group acquired a dedicated cluster based on Opteron 64-bit CPUs. We hope to conduct tests with substantially larger grids and a more in-depth analysis of the OpenMP, MPI and combined approaches in order to gain more knowledge about the efficiency of both paradigms in regard to finite differences as well as to compare our results to the work of William L. George and James Warren as presented in [2]. Special emphasis will be put on the question whether the OpenMP code can be replaced by executing two instances of the simulator on one node with MPI as only means of communication which would significantly reduce the complexity of the overall code-base.

Acknowledgments

The authors wish to thank the ”MFG Stiftung Baden-Wuerttemberg” for supporting this work through the ”Karl-Steinbuch-Scholarship” for Tobias Frodl. We also acknowledge the financial support of the state Baden-Wuerttemberg within the programme LARS, Grand No. lars-08.17.

References

[1] *Nestler, B., Garcke, H., Stinner, B.:* Multicomponent alloy solidification: Phase-field modelling and simulations, Phys. Rev. E 71, (2005) 041609-1 - 041609-6.

[2] *George, W. L., Warren, J. A.:* A Parallel 3D Dendritic Growth Simulator using the Phase-Field Method. J. Comp. Physics 177 (2002), S. 264-283.

[3] *Frodl, T.:* Parallelization of microstructure simulations on multiprocessor systems. Bachelor thesis (2004).

[4] *Vedder, C.:* Zentrale Datenablage und dynamische Gebietsverteilung. Studienarbeit (2005)

[5] *Provatas, N., Goldenfeld, N., Dantzig, J.:* Efficient Computation of Dendritic Microstructures using Adaptive Mesh Refinement. Phys. Rev. Lett. 80 (1998), S. 3308.

[6] *Provatas, N., Goldenfeld, N., Dantzig, J.:* Adaptive Mesh Refinement Computation of Solidification Microstructures using Dynamic Data Structures. J. Comp. Physics 148 (1999), S. 265.

[7] *Danilov, D., Nestler, B.: Microstructure characteristics in binary eutectic alloys: Adaptive finite element phase-field simulations*, Modern Physics Letters B (2005), submitted.