Computer Physics Communications 186 (2015) 48–64

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

MPI + OpenCL implementation of a phase-field method incorporating CALPHAD description of Gibbs energies on heterogeneous computing platforms

P. Gerald Tennyson [a,*], G.M. Karthik [b], G. Phanikumar [a]

[a] Department of Metallurgical and Materials Engineering, Indian Institute of Technology Madras, Chennai - 600036, Tamilnadu, India

[b] Department of Mechanical Engineering, National Institute of Technology Karnataka, Surathkal - 575025, Karnataka, India

ARTICLE INFO

Article history:
Received 8 January 2014
Received in revised form
13 August 2014
Accepted 24 September 2014
Available online 2 October 2014

Keywords:
Microstructure evolution
Phase-field modeling
MPI
OpenCL
GPU
Hybrid parallelization

ABSTRACT

Phase-field method uses a non-conserved order parameter to define the phase state of a system and is a versatile method for moving boundary problems. It is a method of choice for simulating microstructure evolution in the domain of materials engineering. Solution of phase-field evolution equations avoids explicit tracking of interfaces and is often implemented on a structured grid to capture microstructure evolution in a simple and elegant manner. Restrictions on the grid size to accurately capture the interface curvature effects lead to large number of grid points in the computational domain and render the simulation computationally intensive for realistic simulations in 3D. However, the availability of powerful heterogeneous computing platforms and super clusters provides the advantage to perform large scale phase-field simulations efficiently. This paper discusses a portable implementation to extend simulations across multiple CPUs using MPI to include use of GPUs using OpenCL. The solution scheme adapts an isotropic stencil that avoids grid-induced anisotropy. Use of separate OpenCL kernels for problem specific portions of the code ensure that the approach can be extended to different problems. Performance analysis of parallel strategies used in the study illustrate the massively parallel computing possibility for phase-field simulations across heterogeneous platforms.

1. Introduction

Phase-field method is an emerging technique for simulating microstructure evolution during phase transformations, which uses a non conserved order parameter to define the phase state of a system [1–6]. Microstructure evolution occurs by the minimization of a Ginzburg–Landau type free energy functional with respect to an order parameter ϕ along with other field parameters such as, temperature T and concentration c. The phase-field method is generally considered as an alternative technique to the more complex front tracking methods for moving boundary problems. It involves solution of a set of non-linear PDEs at every point in the spatial domain making it computationally demanding. Solution on domains with large number of grids is now possible due to increasing availability of computational power. Compute intensive applications are able to capitalize on multicore and superscalar CPUs for high performance computing. This has made distributed parallel architectures finding place in the computing roadmap for the next 30 years [7]. Several hybrid implementations are now becoming available to harness the potential of high performance heterogeneous platforms [8–12]. Large scale computations that are also memory intensive require use of multiple processors to be able to perform the simulations. In this study, we explore the combination of MPI library for communication along with OpenCL for GPU computation to perform large scale 3-D simulations. This attempt is the first such application of the MPI + OpenCL hybrid technique to phase-field method.

Voller and Porté-Agel [13] put forth the fact that the largest grid sizes that were used in simulation studies in a given year scaled according to Moore's law. However, literature pertaining to parallel implementation of the phase-field method is limited. One of the first parallel phase-field simulations was attempted by George et al. [14] to simulate an alloy dendrite growth in 3-D. Suwa et al. [15]

* Correspondence to: Tata Research Development and Design Centre, Pune-411013. India. Tel.: +91 4422574770; fax: +91 4422574752.
E-mail addresses: g.tennyson@tcs.com, geraldtennyson@gmail.com (P. Gerald Tennyson), gum.10m161@nitk.edu.in (G.M. Karthik), gphani@iitm.ac.in (G. Phanikumar).

http://dx.doi.org/10.1016/j.cpc.2014.09.014
0010-4655/

performed a MPI implementation of a multi-phase-field model for three dimensional grain growth. A shared memory parallelization of single component solidification was done by Xu et al. [16], which was further extended to study multiscale convection through distributed computing [17]. It was not until 2011, that AOKI Laboratories at Tokyo Tech GSIC was credited with running the first large scale computations on the growth of alloy dendrite clusters on the TSUBAME 2.0 [18,19,10,11]. A cluster of Al–Si alloy dendrites was grown on a grid of $4096 \times 6500 \times 10400$ and which achieved 2.0000045 PFlops for which Shimokawabe et al. [18] were awarded the 2011 Gordon Bell prize.

Though phase-field computations on the GPU were performed on a large scale and have shown the remarkable power of the GPU in speeding up the execution of phase-field codes, they were largely based on CUDA implementations and hence were constrained to run only on NVIDIA® cards [18,19,10,11]. OpenCL specification holds the following advantages: (i) open standard; (ii) cross platform and vendor independent [20–22] and (iii) allows low-level access to hardware [23,24,20,25,26]. Till date, there is no parallel implementation nor an evaluation of a phase-field technique using the OpenCL specification.

This paper discusses the following: (i) A brief description on the phase-field model used and realistic Gibbs energy functions employed for the relevant phases. Parameter relations while bridging these two methods are also highlighted. (ii) An OpenCL solver for solving the governing equations on single and multiple GPUs (MPI + OpenCL). (iii) Performance metrics for the solver across all platforms (CPU, GPU and multiple GPUs). This study highlights the portable usage of stencil based computation across heterogeneous platforms. The authors would like to emphasize that this is not the only solution method, but one that have been implemented by them. A comparison between OpenCL implementation across different GPU architectures (NVIDIA® and ATI Radeon®) has also been presented to demonstrate portability. Domain sizes are chosen such that memory limits are well within the specifications for each cards.

2. Phase-field model

An isothermal binary alloy phase-field model with anti-trapping current by Kim [27] is used for the simulations. Alloy solidification involves both heat and mass transfer. However, since these two processes are occurring at two different time scales, in most cases, it is sufficient enough to consider the slower of the two, namely, mass transfer. Heat transfer is usually two orders of magnitude faster than solute diffusivity and temperature evolves much rapidly compared to solute redistribution. Hence, thermal field can usually be treated as frozen or isothermal so that the process can be assumed to take place in a quasi-steady state.

The total free energy which must decrease during an isothermal process is given by,

$$F = \int_V \left[f(\phi, c, T) + \frac{\varepsilon_c^2}{2}(\nabla c)^2 + \frac{\varepsilon_\phi^2}{2}(\nabla \phi)^2 \right] dV \tag{1}$$

where $f(\phi, c, T)$ is the homogeneous free energy density which is a function of the order parameter ϕ, composition c and temperature T. ε_ϕ and ε_c are the gradient energy coefficients for phase-field and composition, respectively.

Although, simulating the evolution of complex morphologies is quite straightforward with this method, making quantitative predictions on experimentally relevant length and time scales has remained a major challenge. In such cases, the width of the diffuse interface δ and the characteristic time scale τ, being microscopic, will not be able to resolve a typical experimental length/time scales even with efficient algorithms. In the light of the above, the only way to proceed with the simulations would be increase the interface width and the time scale of the simulations. But the bloating of the diffuse width, though sounding computationally feasible, has its own inherent disadvantages, such as, (i) solute diffusion/surface diffusion along the arc length of the interface, (ii) interface stretching due to the moving curved interface thereby leading to a modification of the mass conservation equation at the interface and (iii) a discontinuity of the chemical potential across the interface leading to solute trapping [28–30]. The magnitude of these effects scales up with the interface width.

The first two of these effects can be nullified by a nonvariational approach which can be more computationally efficient than a strictly variational formulation [31]. In general, a nonvariational approach involves using different interpolation functions for the evolution equations: $p(\phi)$ for the phase-field and $h(\phi)$ for the diffusion equation and $s(\phi)$ for the composition profiles, thereby decoupling the phase-field equation from the diffusion equation. The third effect is the anomalous jump in the chemical potential (solute trapping) during low speed solidification. This is avoided by using an anti-trapping current, an additional solute current introduced, which pushes the solute out of the solidifying material.

The evolution equations to be solved for are:

$$\frac{\partial \phi}{\partial t} = M_\phi \left[\varepsilon_\phi^2 \nabla^2 \phi - W \frac{dg_w(\phi)}{d\phi} - \frac{dp(\phi)}{d\phi} f_D \right] \tag{2}$$

$$\frac{\partial c}{\partial t} = \nabla \cdot [(1 - h(\phi)) D_{\mathrm{L}} \nabla c_{\mathrm{L}}] + \nabla \cdot j_{at} \tag{3}$$

where f_D is the driving force for the phase transformation and j_{at} is the anti-trapping current given by,

$$f_D = \left(f^{\mathrm{S}}(c_{\mathrm{S}}) - f^{\mathrm{L}}(c_{\mathrm{L}}) + \frac{\partial f^{\mathrm{L}}(c_{\mathrm{L}})}{\partial c_{\mathrm{L}}} (c_{\mathrm{S}} - c_{\mathrm{L}}) \right) \tag{4}$$

$$j_{at} = \frac{\varepsilon_\phi}{\sqrt{2W}} (c_{\mathrm{L}}(\mathrm{x}, t) - c_{\mathrm{S}}(\mathrm{x}, t)) \alpha(\phi) \frac{\partial \phi}{\partial t} \frac{\nabla \phi}{|\nabla \phi|}. \tag{5}$$

Table 1
Redlich–Kister coefficients for the Al–Mg system in J/mol [34].

	v	a_v^ϕ	b_v^ϕ
ϕ = liq	0	−9019	4.794
	1	−1093	1.412
	2	494	0
ϕ = fcc	0	1593	2.149
	1	1014	−0.660
	2	−673	0

The solid and liquid compositions c_S and c_L, need to be solved iteratively at every time step and each spatial point through the solution of the following auxiliary equations,

$$c = s(\phi)c_S + (1 - s(\phi))c_L \tag{6}$$

$$\frac{\partial f^S}{\partial c_S}(c_S(x, t)) = \frac{\partial f^L}{\partial c_L}(c_L(x, t)). \tag{7}$$

$f(\phi, c, T)$ is the homogeneous free energy density which is a function of the phase-field ϕ, composition c and temperature T. f^S, f^L are the free energy densities of the solid and liquid phases, respectively. $\alpha(\phi)$ is a coupling coefficient for the non-variational formulation involving the interpolation functions $p(\phi)$, $h(\phi)$ and $h(\phi)$. Other model parameters are the double well potential $g_w(\phi)$, the energy barrier W, phase-field mobility M_ϕ and solute diffusivity D_L. Description of these model parameters and the formulation of the governing equations, numerical approaches are presented elsewhere [32].

3. Thermodynamic description and models

A thermodynamic description of a phase requires the assignment of Gibbs energy functions for each phase under consideration. In this context, CALPHAD provides a means to find required Gibbs energy functions through thermodynamic modeling. The CALPHAD method employs a variety of models to describe the temperature, pressure and composition dependencies of the Gibbs energy functions of various phases. In the present work, Al–Mg binary is chosen by considering two phases: liquid and fcc solid solution. The molar Gibbs energy for both the phases are modeled using random substitutional solution and is expressed as,

$$G_m = x_A{}^\circ G_A + x_B{}^\circ G_B + RT(x_A \ln x_A + x_B \ln x_B) + x_A x_B \sum_{v=0}^{n} {}^v L_{AB}(x_A - x_B)^v \tag{8}$$

where ${}^\circ G_{A,B}$ is the reference molar Gibbs energy of the elements A, B in their pure state, the third term is the ideal Gibbs energy of mixing and the fourth, excess contribution to the Gibbs energy arising due to the non ideal interaction between the elements. The Redlich–Kister coefficients for the binary excess model are expanded as ${}^v L_{AB} = a_v + b_v T$ for each phase. The Gibbs energy functions of the pure elements in their stable and metastable states are taken from the Scientific Group Thermodata Europe (SGTE) database [33] and the excess Gibbs energies for Al–Mg are taken from [34] and shown in Table 1.

3.1. Relating phase-field and CALPHAD

In most phase-field formulations, volume free energy densities, f, are used to describe the free energy functional F, which could be either Helmholtz or Gibbs energies. However, to evaluate the chemical contribution of the free energies in using the thermodynamic functions obtained through CALPHAD, molar Gibbs energy densities, g, are preferred [35]. Assuming molar volumes v_m to be the same for both phases, the volume free energy density f can be replaced with molar Gibbs energy densities g and the molar concentration c_i with x_i through,

$$x_i = \frac{n_i}{n_{tot}} \tag{9}$$

and

$$c_i = \frac{n_i}{V} = \frac{x_i}{v_m} \tag{10}$$

where, n_i and x_i are the amount and the mole fraction of the solute i, respectively and V, the total volume of the system. The free energy density f is related to Gibbs energy density g by

$$f(\phi, c) = \frac{1}{v_m} g(\phi, x_B). \tag{11}$$

Reformulating the phase-field and concentration equations incorporating this change of variables from c to x_B and f to g in Eqs. (2) and (3), we get,

$$\frac{1}{M_\phi}\frac{\partial \phi}{\partial t} = \varepsilon^2 \nabla^2 \phi - W\frac{dg_w(\phi)}{d\phi} - \frac{dp(\phi)}{d\phi}\frac{1}{v_m}\left(g^S - g^L + \mu(x_B^S - x_B^L)\right) \tag{12}$$

$$\frac{\partial x_B}{\partial t} = \nabla \cdot [(1 - h(\phi))D_L \nabla x_B^L] + \nabla \cdot \left(\alpha(\phi)\frac{\partial \phi}{\partial t}\frac{\nabla \phi}{|\nabla \phi|}\right). \tag{13}$$

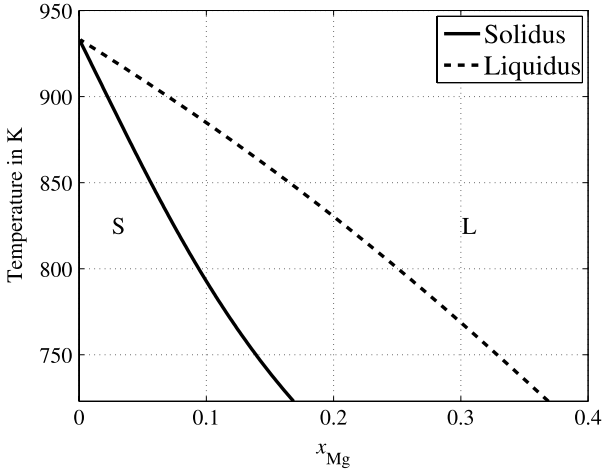

Fig. 1. Al-rich corner of Al–Mg, computed using CALPHAD description of Gibbs energies.

Here, μ is the slope of g^{L} with respect to x_B^L. The phase boundaries of the liquidus and solidus for the Al-rich corner of Al–Mg are calculated using Gibbs energy functions of the phases in question as shown in Fig. 1. For a given T and P, the unknowns x_B^L and x_B^S are calculated by the equal chemical potential condition given in Eq. (7).

The application of realistic thermodynamic functions is now possible by incorporating the above mentioned polynomial expressions for Gibbs energy as free energy density functions in the formulation. The authors believe that the simple nature of these functions enables a fast computation of realistic thermodynamic data as compared to the API route. The polynomial expressions in Eq. (8) and Table 1 are directly hard-coded into the program. The functions that contain these polynomials are kept modular for each system. The advantage of the entire source code being developed as part of this study enabled hard-coding of these functions to gain compile time optimizations by the compiler and lead to faster computation. This is important as the compilation time is negligible compared to the execution time for these programs.

4. Numerical approach

GPU performance benchmarks done for continuum CFD in CPU and GPU reported that implicit solvers gave a performance gain of 5X as compared to the 15X gain in using explicit solvers against the CPU framework [36]. This has encouraged the authors to explore similar performance gains using explicit solution scheme in phase-field method. Although, implicit approaches do take away the restriction on the time interval for numerical stability, parallelization of implicit schemes is often not straightforward. Also, use of explicit schemes renders the problem easily parallelizable across massively parallel architectures. In this study, an explicit Euler scheme was used for time discretization.

A finite difference approach was used to discretize the governing equations. An explicit forward difference in time and first order central difference in space for the gradients and a second order central difference for the Laplacians were used. In most conventional finite-differences used for the discretization of PDEs, an anisotropy is introduced into the numerical scheme, which comes from the directional bias of the error terms in the discretization. In order to preserve the symmetry in the simulation and mitigate the effect of grid anisotropy, a more generalized isotropic scheme for calculating the Laplacian was used [37]. A 9 point Laplacian for 2-D and a 27 point Laplacian for 3-D were utilized and their corresponding discretization schemes are given by Eq. (14) and Eq. (15), respectively. Computation of Laplacian using this scheme produces both faster as well as more rotationally symmetric results. Assuming uniform grid spacing (Δx) along all three directions for simplicity of expression for illustration, we can write the Laplacian as follows.

$$\nabla^2_{\phi} = \frac{1}{6\Delta x^2}[(\phi_{i-1,j+1} + \phi_{i+1,j+1} + \phi_{i+1,j-1} + \phi_{i-1,j-1}) + 4(\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j+1} + \phi_{i,j-1}) - 20\phi_{i,j}]. \tag{14}$$

A 27-point Laplacian, though discernibly large, can be expressed in terms of their positions on a cube. The 27 points can be split up as those that occupy the corners (**8**), edge centers (**12**), face centers (**6**) and body center (**1**).

$$\nabla^2_{27}\phi_i = \frac{3}{13\Delta x^2}\left[\sum_{j\in N_f}\phi_j + \frac{1}{2}\sum_{j\in N_e}\phi_j + \frac{1}{3}\sum_{j\in N_c}\phi_j - \frac{44}{3}\phi_i\right] \tag{15}$$

where, N_f are the face points on the cube, N_c are the corner points, N_e are the edge centered points and ϕ_i being the body centered point.

4.1. Discretization of the anti-trapping current and off-grid locations

In order to solve the diffusion equation (Eq. (3)) with the anti-trapping current, $\frac{\partial\phi}{\partial t}$ is required. Discretization of the anti-trapping current was done along similar lines as in the literature [38].

To calculate all the quantities at each grid point i, j, k with second order accuracy in space, it is required to calculate the gradients at the off-grid locations *i.e.* at half the cell size in each direction. The values of $h(\phi)$ in Eq. (13) are determined at the off-grid positions by averaging the values of the neighboring nodes in each direction.

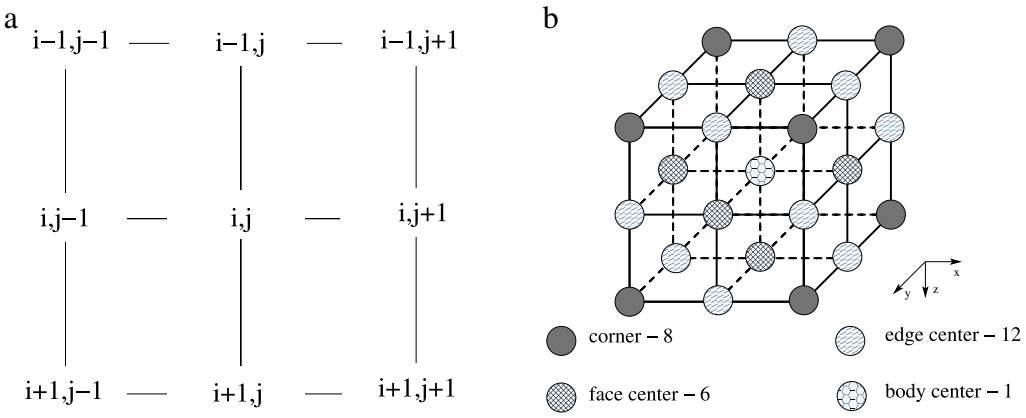

Fig. 2. (a) 9-point stencil and (b) 27-point stencil.

Table 2
Magnitude of the antitrapping current j_{at} as a function of interface width δ in comparison with one of the diffusion terms $(1 - h(\phi))\nabla c_L$.

Interface width δ	$\frac{(1-h(\phi))\nabla c_L}{\phi(1-\phi)}$	$\frac{j_{at}}{\phi(1-\phi)}$
600	0.0925	0.0089
700	0.1074	0.0103
800	0.1227	0.0118
900	0.1372	0.0133
1000	0.1528	0.0148

4.2. Stencils

The discretization of first and second order differentials and the requirement of gradient terms at the staggered grid locations reminds one of a similar method used for solving convection–diffusion problems [39]. The numerical techniques elucidated in Patankar's celebrated book [39] to solve problems of this kind include the application of Tri-Diagonal Matrix Algorithm (TDMA), Alternate Direct Implicit (ADI) method and iterative schemes to evaluate nodal and staggered grid values. These algorithms are array based, require considerable effort to convert to be massively parallel and the solvers are to be written for each type of platform used for implementation.

In this study, instead of solving the problem by taking the entire array as input, a stencil based solver is employed. A stencil is created at every node point in the domain depending on the dimension. In order to accommodate the points required for the gradients and the Laplacian, a 'Moore' neighborhood stencil is chosen, where the number of points in the stencil grid is 9 for 2-D and 27 for 3-D, shown in Fig. 2(a) and Fig. 2(b), respectively. A stencil computation is characterized by updating each point in a structured grid by an expression depending on the values on a fixed geometrical structure of neighboring grid points. In our context, a rectilinear grid is assumed, which from a hardware architecture point of view is favored due to the regularity of the data access pattern. This allows using the hardware prefetcher in an optimal way and streaming in the data from the main memory to the compute elements [40].

The practical advantage of using a stencil based solver is that the code can be readily ported across different platforms. In both cases (CPU and GPU), the stencil based solver is identical. This is because all the desired points for the gradients and the Laplacian are present within the stencil and as a result all the calculations can be performed within the stencil for that particular node point (i, j, k).

4.3. Effect of anti-trapping current on the tip velocity

Table 2 shows the magnitude of the anti-trapping current as a function of interface width. One of the diffusion terms is also given along for comparison. These numbers are normalized with respect to the total number of grid points in the interface region. We note that the anti-trapping current is only about 10% of one of the diffusion terms and is seen to be increasing in magnitude with the interface width. It should be noted that the magnitude of the antitrapping current though less, has a significant effect in neutralizing the effect of abnormal chemical potential jump as a result of increased interface width. This results in bringing the interface velocity close to the corresponding values of the sharp interface model. Fig. 6 shows the velocity invariance with interface width.

5. Parallelization

To parallelize this scheme, a two stage parallelization is done; a MPI communication of (i) ϕ_0 and c_0 values to calculate $\partial_t\phi$ and (ii) communication of the calculated $\partial_t\phi$ to calculate ϕ_n and c_n. Though this process involves communication in two stages for each time step, it is a fairly simple exchange limiting the number of ghost cells to just a singular array/plane. This procedure was adopted for both CPU and GPU. However, only the GPU version of the MPI + OpenCL implementation is discussed here. The CPU solver is used for comparison of execution times with the GPU.

5.1. Target platforms

The parallel implementation of the phase-field simulations was carried out on a CPU cluster (Virgo) and a GPU cluster (Libra) at IIT Madras, the details of which are provided in Tables 3 and 4. The number of CPUs that were employed for the simulations presented in this

Table 3
CPU cluster computing platform (Virgo).

System	IBM® x iDataPlex dx360 M4
Chip architecture	Intel® Xeon® E5-2670
Clock	2.6 GHz
Sockets per node	2
Cores per socket	8
Compute nodes	292
Threads per socket	16
DRAM capacity per node	64 GB
DRAM pin bandwidth	51.2 GB/s

Table 4
GPU platforms.

Platform	Cluster (Libra) NVIDIA® M2070 (Tesla)	Desktop ATI Radeon® HD5870 (Cypress)
Cluster compute nodes	8	1
GPUs/compute node	3	1
Compute units/GPU	14	20
CPU	Intel® Xeon® X5670	AMD Phenom® II X6 1090T
CPU frequency	2.93 GHz	3.2 GHz
Processing elements	448	1600
Core clock frequency	1.147 GHz	850 MHz
Local memory	48 KB	32 KB
Global memory	6 GB	1 GB
Global memory bandwidth	148 GB/s	153.6 GB/s
Peak performance		
(float)	1.03 TFlops	2.72 TFlops
(double)	515.2 GFlops	544 GFlops

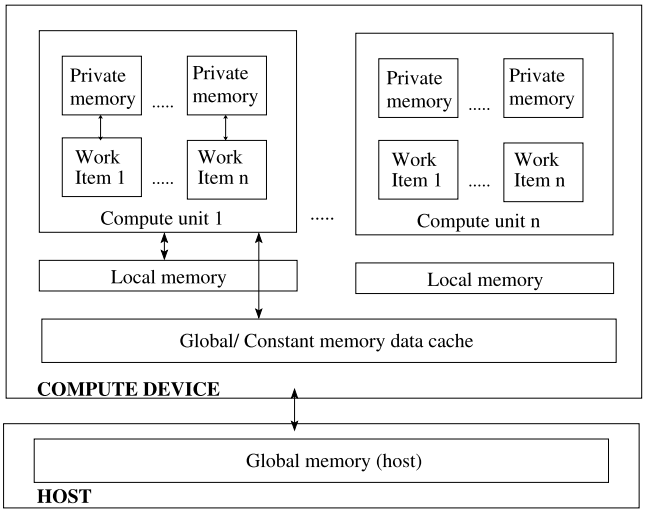

Fig. 3. Memory model of a typical GPU with reference to OpenCL terminologies.

study was a maximum of 64. Single GPU simulations establishing portability and performance across the architectures were done using ATI Radeon® HD5870 and NVIDIA® M2070. The multiple GPU simulations were carried out on the Libra cluster that contains a total of 24 NVIDIA® GPUs.

6. Parallel implementation on a single GPU

Implementation on a single GPU proceeds in a rather straightforward manner, where the entire computational domain is processed by stream cores or the processing elements. OpenCL implements a master–slave architecture, where the 'host' (an OpenCL enabled application) submits work to one or more devices. When a kernel is submitted for execution, an n-dimensional index space is defined. The application also defines the context of execution. Each context includes the list of target devices, associated command queues and memory accessible to the devices. OpenCL allows the application to distribute tasks to either the CPU or the GPU explicitly thus making the load-balancing to be implemented at the application level. In our context, two main buffers, $p0$ and p, and two intermediate buffers, c_{SL} and $\partial_t\phi$, were required for the solution scheme. $p0$ and p are arrays of structures holding the phase-field and concentration variables that are required to be solved and c_{SL} contains the variables c_S and c_L.

The memory model of a GPU depends on its architecture as shown in Fig. 3. Following is a basic outlook of the programming methodology as adopted by the authors on a GPU using OpenCL as the programming language. A simple parallelism approach method is followed which involves accessing the global memory cache and OpenCL being a low level programming language, it ends with the instruction set being quite large.

- Initialize the domain. Allocate memory for the required data structures.
- Identify the compute platforms where the processing needs to be done (CPU, GPU) and get their respective device ids.
- Create a context under which the instructions to be processed are executed. Different contexts can be created on different devices or the same context can be extended to different devices.
- Inside the context, a command queue is created in which the prioritization of instructions is input.
- Memory buffers are then created which hold the respective structures (phase-field, concentration, c_S, c_L), which can be designated as read only, write only and read write.
- The buffers are then queued up on the device memory.

  The sample code below demonstrates the querying of platforms, the devices present, creation of contexts, command_queues, buffers and writing buffers on to the device memory.

```
/*------- Create context and command_queue on the device -------*/
clGetPlatformIDs(1, &platform_id, &return_num_platforms);
clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, no_of_gpus, device_id,
&ret_num_devices);
context=clCreateContext(NULL, 1, &device_id, NULL, NULL, &err);
command_queue = clCreateCommandQueue(context, device_id, 0, &err);
```

```
/*------- Create buffers on the device -------*/
device_buffer1 = clCreateBuffer(context, CL_MEM_READ_ONLY,
array_size*sizeof(data_type), array_pointer1, &err);
device_buffer2 = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
array_size*sizeof(data_type), array_pointer2, &err);

/*------- Copy buffers from host to device -------*/
clEnqueueWriteBuffer( command_queue, device_buffer1, CL_TRUE, 0,
array_size*sizeof(data_type), array_pointer1, 0, NULL, NULL );
clEnqueueWriteBuffer( command_queue, device_buffer2, CL_TRUE, 0,
array_size*sizeof(data_type), array_pointer2, 0, NULL, NULL );
```

- In OpenCL, the .cl kernel file is read only during execution and hence the program has to be built during runtime and a buffer file has to be created to hold the errors, if any, which pop up during the compilation of the kernel file during runtime. This is shown below as a snippet.

```
#define MAX_SOURCE_SIZE (0x100000)
FILE *fp;
const char fileName[] = "solver_kernel.cl";
size_t source_size;
char *source_str;

/*-------- Load kernel source file ---------*/
fp = fopen(fileName, "rb");
source_str = (char *)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

/*----------- Create kernel from source -----*/
program = clCreateProgramWithSource(context, 1,
(const char **)&source_str, (const size_t *)&source_size, &err);

/*----- Build the program for device -------*/
clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
char buildString[100000];
clGetProgramBuildInfo(program, device_id,
CL_PROGRAM_BUILD_LOG, sizeof(char) * 100000, buildString, NULL);
```

- The kernel files are created which are nothing but the functions that are to be called from the host code and all the kernel arguments are also needed to be set as shown below.

```
/*------- Create kernels --------*/
kernel[0] = clCreateKernel(program, "solver_1", &err);
kernel[1] = clCreateKernel(program, "solver_2", &err);
.
.
```

```
kernel[n-1] = clCreateKernel(program, "solver_n-1", &err);

/*------- Set kernel arguments -------*/
clSetKernelArg(kernel[0], 1, sizeof(int), &arg1);
clSetKernelArg(kernel[0], 2, sizeof(int), &arg2);

clSetKernelArg(kernel[1], 1, sizeof(int), &arg1);
clSetKernelArg(kernel[1], 2, sizeof(int), &arg2);
.
.
```

- Execution of the program starts here and the host calls the required kernels and instructs the compute device to execute the instruction set in the respective order. Kernels are submitted for execution using `clEnqueueNDRangeKernel` specifying the dimensions *viz.*, `workdim` = 2, 3 for 2-D or 3-D, respectively.
- All kernel calls are set to execute with the command `clFush` and `clFinish`.

```
for(t=1;t<=ntimesteps;t++){
  err=clEnqueueNDRangeKernel( command_queue, kernel[0], workdim, NULL,
  array_size, NULL, 0, NULL, NULL );
  err=clEnqueueNDRangeKernel( command_queue, kernel[2], workdim, NULL,
  array_size, NULL, 0, NULL, NULL );
  .
  .
  err=clEnqueueNDRangeKernel( command_queue, kernel[n-1], workdim, NULL,
  array_size, NULL, 0, NULL, NULL );

  /*------Read buffer after specified time interval------*/
  if((t%saveInterval)==0){
  err=clEnqueueReadBuffer ( command_queue, buffer_on_device, CL_TRUE, 0,
    array_size*sizeof(data_type), buffer_to_be_read_into, 0, NULL, NULL );
  }
}
err=clFlush(command_queue);
err=clFinish(command_queue);
```

```
/*All the arguments declared as global reside in the device global memory
The local variables reside in the private memory
struct nodeData contains phi and c
struct cslnodeData contains cs and cl
struct gibbsEnergy contains all free energy variables
struct pdData contains phase diagram data (temperature, composition)
struct pfData contains phase-field data (W, epsilon)
struct grid contains domain data (nx,ny,nz,dx,dt)
*/

kernel void phidotsolver(global nodeData* p0, global cslnodeData* csl,
global gibbsEnergy* myGibbs, global pdData* myKKSpd,
global pfData* myKKSpf, global grid* mygrid, global float* phidot)
{
  int col = get_global_id(0);
  int row = get_global_id(1);
  int dep = get_global_id(2);
  /* Initialize all local variables*/

  /*---27 point stencil struct to hold the stencil points---*/
  nodeData stpc[27];
 /*p0 contains the phi and c values at the previous iteration*/

  /*------Solving only in the interior nodes-----*/
  if(row!=0 && row!=( ny - 1 ) && col!=0 && col!=( nx - 1 ) &&
     dep!=0 && dep!=( nz - 1 )){

 /*----------------Create stencil for phi and c ------------------------*/
 /*----- front face------*/
 /*----------------first row-------------------------//
 stpc[0]= p0[ ( dep - 1 )*nx*ny + ( row - 1 )*nx + col - 1 ];
 stpc[1]= p0[ ( dep - 1 )*nx*ny + ( row - 1 )*nx + col ];
```

```
 stpc[2]= p0[ ( dep - 1 )*nx*ny + ( row - 1 )*nx + col + 1 ];
 /*---------------second row-------------------------//
 stpc[3]= p0[ ( dep - 1 )*nx*ny + row*nx + col - 1 ];
 stpc[4]= p0[ ( dep - 1 )*nx*ny + row*nx + col ];
 stpc[5]= p0[ ( dep - 1 )*nx*ny + row*nx + col + 1 ];
 /*----------------third row-------------------------*/
 stpc[6]= p0[ ( dep - 1 )*nx*ny + ( row + 1 )*nx + col - 1 ];
 stpc[7]= p0[ ( dep - 1 )*nx*ny + ( row + 1 )*nx + col ];
 stpc[8]= p0[ ( dep - 1 )*nx*ny + ( row + 1 )*nx + col + 1 ];
 /*-------------------------------------------------*/
 /*--Similarly for the  middle face (dep) and the rear face (dep+1)---*/

 stpc[13] is the i,j,k position

 /*--- 27 point Laplacian----*/
 /*face centered atoms*/
 N_f = stpc[4].phi + stpc[10].phi + stpc[12].phi +
 stpc[14].phi + stpc[16].phi + stpc[22].phi;

 /*edge centered atoms*/
 N_e =  stpc[1].phi  + stpc[3].phi  + stpc[5].phi  + stpc[7].phi  +
 stpc[9].phi + stpc[11].phi + stpc[15].phi + stpc[17].phi  +
 stpc[19].phi + stpc[21].phi + stpc[23].phi + stpc[25].phi;

  /*corner atoms*/
 N_c =  stpc[0].phi  + stpc[2].phi  + stpc[6].phi  + stpc[8].phi  +
 stpc[18].phi + stpc[20].phi + stpc[24].phi + stpc[26].phi;

 Laplacian_phi = 3.0f*(fc_phi + ec_phi/2.0f + co_phi/3.0f -
                44.0f*stpc[13].phi/3.0f) / ( 13.0f*dx*dx );

 /*G_Liquid and G_Solid are the free energy of liquid and solid respectively
 calculated using Gibbs energy data from struct gibbsEnergy,
 dfliq_dc is the chemical potential,
 G_phi is derivative of free energy w.r.t phi,
 W is the double well potential obtained from struct pfData,
 gprime is derivative of the double well function g(phi) w.r.t phi,
 pprime is derivative of the interpolating function p(phi) w.r.t phi,
 Mobility is the phase-field mobility*/

 Driving_force = G_Liquid - G_Solid - dfliq_dc*( cliq - csol );
 G_phi  = W * gprime - pprime*Driving_force;
 phidot[dep*nx*ny + row*nx + col] = Mobility * ( epsilon^2 *
 Laplacian_phi - G_phi );

/* phi_new=phi_old + phidot * dt  */
  }
}
```

Code snippet 1: $\partial_t \phi$ solver depicting the usage of stencils and solver provided for isotropic Laplacian

7. Parallel implementation across multiple GPUs using MPI

The distributed memory parallel applications are written using the message passing model where each process has exclusive access to some amount of local memory and only indirect access to the rest of the memory [41].

7.1. Domain decomposition

For simulating large domains both in 2-D and 3-D it is often necessary to decompose the domain in order to bring down the memory overload and also the computation time as discussed earlier. A simpler option would be to distribute the load across multiple GPUs. There are many ways of splitting the data across the nodes. One is to create a single context and use different buffers on each of the GPUs in a compute node. This method cannot be scaled as it is restricted by the number of GPUs in a compute node. The other method is to tag each core of a CPU to a GPU by a MPI implementation. A MPI virtual topology is created and a 1-D domain decomposition is done as shown in Fig. 4. In order to use the GPUs on different compute nodes, the CPU cores need to be tagged manually to each GPU per compute node. Each MPI process is given a part of the domain with each MPI process holding two additional arrays functioning as ghost nodes. Before the start of each iteration, the boundary layers need to be transferred to adjacent nodes. This is done by a memory transfer from the device to host using `clEnqueueReadBuffer` to a 1-D array and then sending that array to the adjacent node. The adjacent MPI process receives the array on to a buffer and then writes the buffer to the main array through `clEnqueueWriteBuffer`. The MPI processes handling

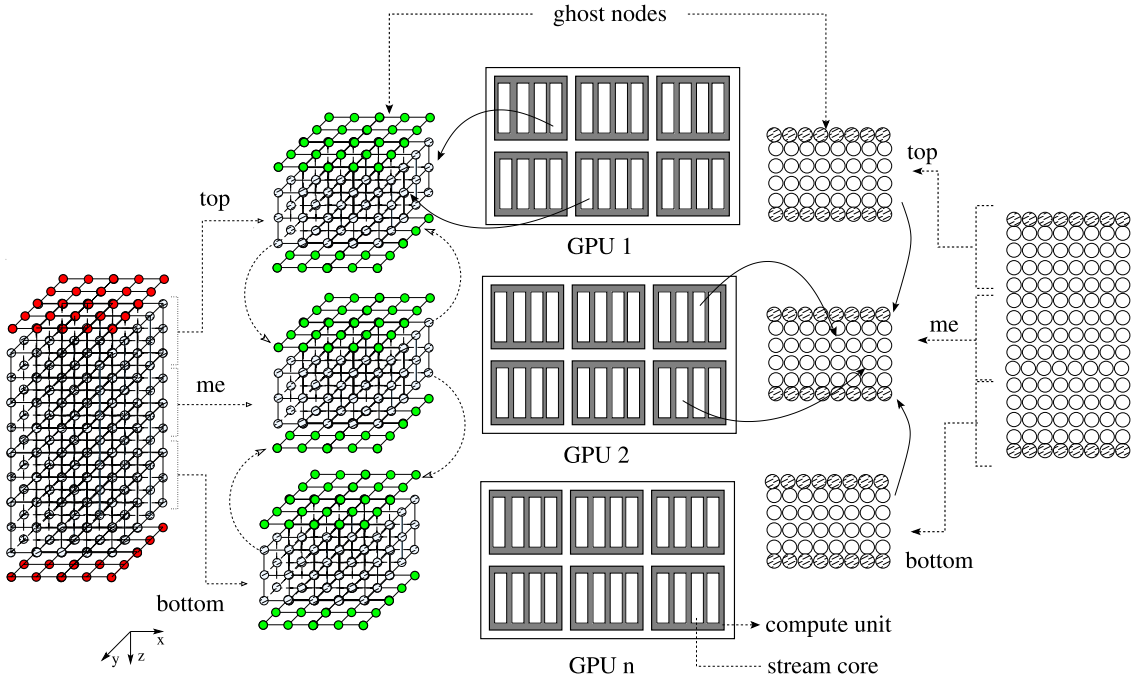

Fig. 4. An MPI+OpenCL implementation across multiple GPUs depicting 1-D domain decomposition employed for both 2-D as well as 3-D arrays.

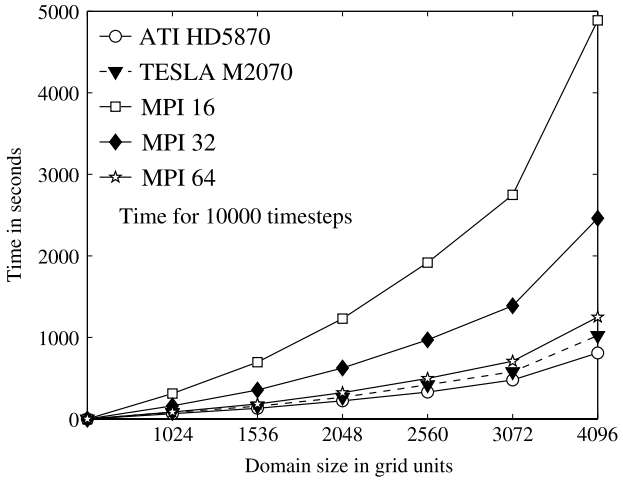

Fig. 5. Performance benchmark of the parallel phase-field solver on multiple CPUs and a single GPU.

the boundary segments of the domain map the boundary elements depending on the type of boundary condition. For example, a no-flux boundary condition is handled by mapping the values of $(n-1)$th row elements to nth row elements.

8. Performance analysis

The performance benchmark for the simulations run on multiple CPUs and a single GPU is shown in Fig. 5. The results for the benchmark pertain to the growth of a primary dendritic arm without any noise. This was to avoid the difference in schemes of the noise algorithm used for the calculations on the CPU and on the GPU. With the increase in the domain size (number of grid points), the GPUs throughput increased. Although the trend shows that GPUs performance to be in vast difference than CPUs, this is based on the results obtained through a naive implementation of the solver as discussed earlier. However, this difference can be minimized, by adopting various memory management techniques on the CPU itself. Fig. 6 shows the convergence of the tip velocities for the simulations executed across multiple CPUs and a single GPU confirming the equivalence of the two types of computation across the two platforms. The correctness of the results has been verified by comparing with relevant theoretical results and presented elsewhere [32].

8.1. Performance results from each device

The performance of the solvers on each of the devices *viz.*, AMD Phenom II X6 1090T, NVIDIA Tesla M2070 (single) and ATI Radeon HD 5870 (single) are shown in Fig. 7(a) and (b). Profiling was done using OProfile [42], by measuring the number of floating point instructions retired for the overall execution against different iterations on the CPU. This instructions count was used to calculate the

Algorithm 1: Code execution on multiple GPUs

Data: Structure $p_o(\phi, c)$,$p(\phi, c)$,$c_{SL}(c_S, c_L)$
Input: p_o
Output: p_n
Initialize the MPI environment;
Create buffers for the structures on the host;
Initialize and split the domain $p_o(\phi, c)$ across the processors;
Initialize OpenCL environment (clGetPlatformIDs, clGetDeviceIDs);
Create context and command queue for each of the devices;
Create buffers on the devices (clCreateBuffer);
Copy buffers from host to device (clEnqueueWriteBuffer);
Read kernel file (solverKernels.cl);
Create and build the kernels for execution;
Create kernels (clCreateKernel) and set the kernel arguments (clSetKernelArg);
Start timeloop;
foreach *timestep t* **do**
  Transfer elements of $p_o(\phi, c)$ at slice boundaries to ghost nodes between CPUs;
  → memory transfer from GPU–CPU (clEnqueueReadBuffer), CPU–CPU (MPI_Send/ MPI_Recv) and CPU–GPU (clEnqueueWriteBuffer);
  foreach Element *in grid* **do**
    Call c_{SL} kernel;
  end
  foreach Element *in grid excluding boundaries* **do**
    Call $\partial_t\phi$ kernel;
  end
  Apply boundary conditions for $\partial_t\phi$;
  Transfer elements of $\partial_t\phi$ at slice boundaries to ghost nodes between CPUs;
  foreach Element *in grid excluding boundaries* **do**
    Call (ϕ, c) kernel;
  end
  Apply boundary conditions for $p[\phi, c]$ boundaries;
  Call noise kernel;
  Swap buffers $p_o \longleftarrow p_n$;
end
MPI_Finalize();

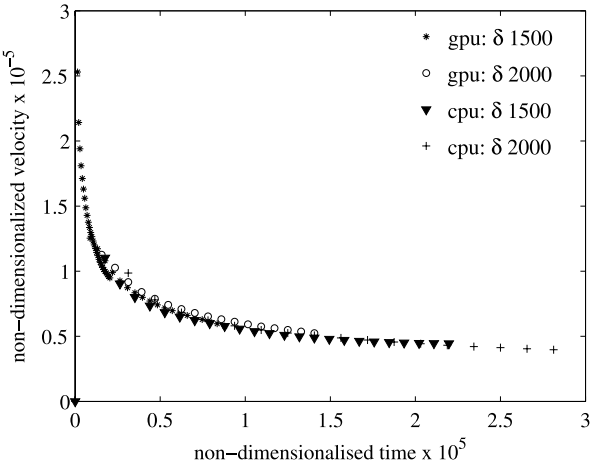

Fig. 6. Convergence of dendrite tip velocities from numerical codes executed on a GPU (OpenCL - NVIDIA® Tesla M2070) and multiple CPUs (MPI - 64 processors).

FLOPS for the respective devices having known the corresponding execution times. This method though an approximation, provides us with a better understanding of the overall performance of the kernels on the devices. The obtained measure of FLOPS on each of the devices also helps to possibly understand the efficiency of the kernels executing on the devices, shown in Fig. 7(b) by comparing simulations with the corresponding theoretical peak performance of each of the devices *viz*., CPU (55.55 GFLOPS), M2070 (1.03 TFLOPS) and HD5870 (2.72 TFLOPS). Although, the current kernel versions provide a good scaling, it clearly shows that the devices are still under utilized. This may be due to the naive implementation of the solver kernels, where device specific optimizations were not realized. In contrast, this implementation has achieved a significant speed up compared to CPU and also a good fraction of the respective theoretical peak performance on the GPU devices. The current implementation is an attempt to focus on a portable phase-field code that can be

executed across various computing devices using OpenCL. However, efforts are still underway to optimize the kernels to achieve maximum utilization of the devices.

8.2. Further analysis on scalability

Strong scaling analysis of the governing equations on multiple GPUs shows that 3-D domains scale up much more efficiently than 2-D domains as shown in Fig. 8(a) and (b). The ratio of time taken for communication including latency to the total time for computation is generally lower for a 3-D simulation. This is because of a large number of grid points assigned to the respective sub-domain.

Weak scaling results for 2-D and 3-D shown in Fig. 9(a) and Fig. 9(b), respectively, display a similar trend. In both cases, simulations on a 3-D domain were found to be closer to the ideal scenario.

A decrease in the overall execution time though ensured, the scalability analysis in Figs. 8 and 9, shows that it is always not possible to achieve theoretical scaling efficiency. This is mainly due to the time spent in communicating across processors between computations. In order to obtain the breakup in the overall execution times, it is necessary to profile the kernels to extract information on the computation times. OpenCL kernels can be profiled by enabling the CL_QUEUE_PROFILING_ENABLE option in clCreateCommandQueue and using event handlers *viz.*, clWaitForEvents and clGetEventProfilingInfo to retrieve timer information. The event handler holds the information on the initiation and completion of a kernel, which can be retrieved using clGetEventProfilingInfo. A code snippet for profiling clEnqueueReadBuffer and clEnqueueNDRangeKernel are depicted for brevity.

```
/*------- Kernel profiling -------*/
cl_event kerndone;
cl_ulong kern_start;
cl_ulong kern_end;
cl_mem d_p0;
int offset, count;
float *botsend;

offset =sizeof(float)*nx;
count=sizeof(float)*nx;
botsend  = (float *)malloc(sizeof(float)* nx );

command_queue = clCreateCommandQueue(context, device_id[choice],
CL_QUEUE_PROFILING_ENABLE, &ret);

clEnqueueReadBuffer(command_queue, d_p0, CL_TRUE, offset, count,
botsend, 0, NULL, &kerndone);
clWaitForEvents(1,&kerndone);
clGetEventProfilingInfo(kerndone, CL_PROFILING_COMMAND_START,
sizeof(kern_start), &kern_start, NULL);
clGetEventProfilingInfo(kerndone, CL_PROFILING_COMMAND_END,
sizeof(kern_end), &kern_end, NULL);
time_taken= kern_end - kern_start;
.
.
clEnqueueNDRangeKernel( command_queue, kernel[1], workdim,
NULL, globaldim, NULL, 0, NULL, &kerndone );
clWaitForEvents(1,&kerndone);
clGetEventProfilingInfo(kerndone, CL_PROFILING_COMMAND_START,
sizeof(kern_start), &kern_start, NULL);
clGetEventProfilingInfo(kerndone, CL_PROFILING_COMMAND_END,
sizeof(kern_end), &kern_end, NULL);
```

Code snippet 2: Profiling an OpenCL kernel

Timing information for the MPI communication can be obtained using MPI_Wtime(). However, it should be noted that the kernel execution times are in nanoseconds and the MPI_Wtime() is in seconds. Figs. 10 and 11 show kernel execution time as a measure of computation time and MPI_Wtime as a measure of time taken for communication. These figures also include the time taken for GPU–CPU and CPU–GPU memory copy which is negligible but could be dependent on the domain size and the size of the datatypes that are passed across. In the case of strong scaling, varying the domain size as a measure of the number of GPUs shows a slight increase in the memory copy time. The performance breakup shows that GPU–CPU memory copy consumes less time than the CPU–CPU communication time by the MPI API. In comparison, the analysis shows that the major part of the execution time is by the communication overhead of the MPI routines (MPI_Send and MPI_Recv).

9. Noise

During actual growth conditions, due to thermal/solutal fluctuations in the solidifying melt, perturbations set along the primary arm of the dendrite, which, if favored get amplified and grow into secondary arms/branches. To simulate this effect of the perturbations, numerical noise was introduced along the liquid side of the interface ($0.1 < \phi < 0.5$). Noise is introduced by generating random numbers between -1

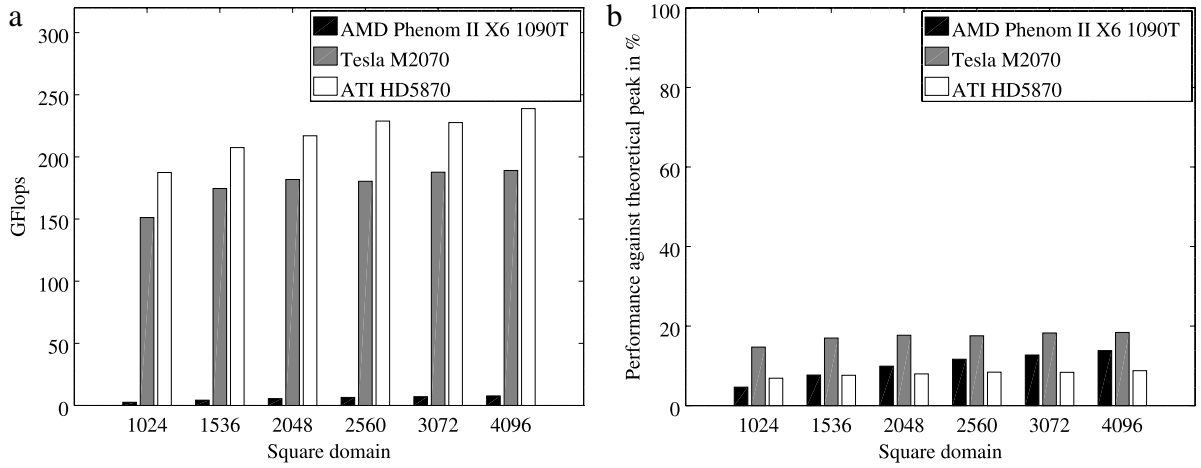

Fig. 7. (a) Performance of the solver kernels on AMD Phenom II X6 CPU, ATI Radeon HD 5870 and NVIDIA Tesla M2070 measured in GFLOPS. (b) Benchmark of the performance of the solvers against the theoretical peak of each device.

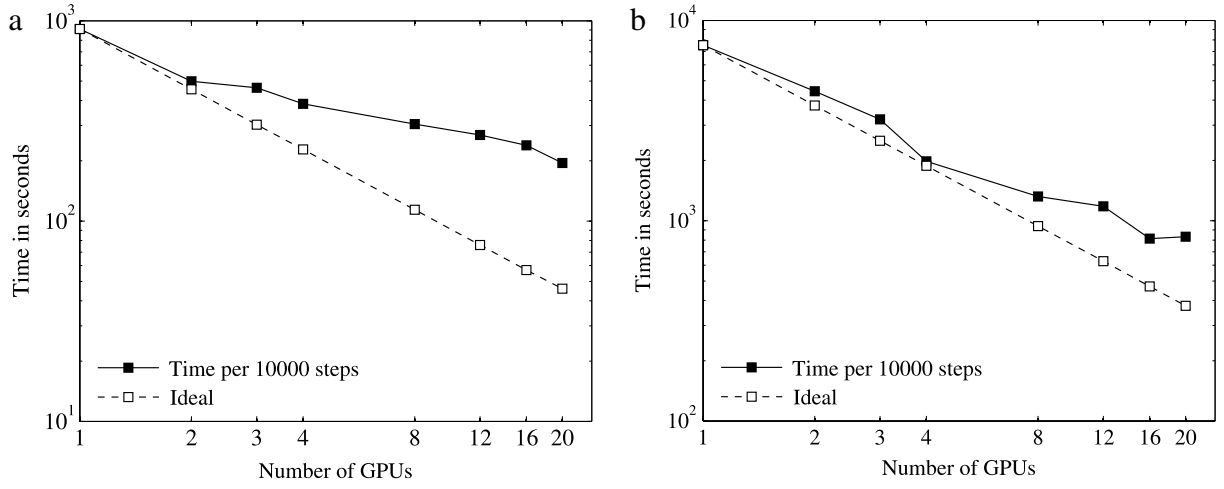

Fig. 8. Strong scaling analysis for (a) 2-D domain of length 4082× (4082/no. of GPUs) and (b) 3-D domain of length 482 × 482× (482/no. of GPUs).

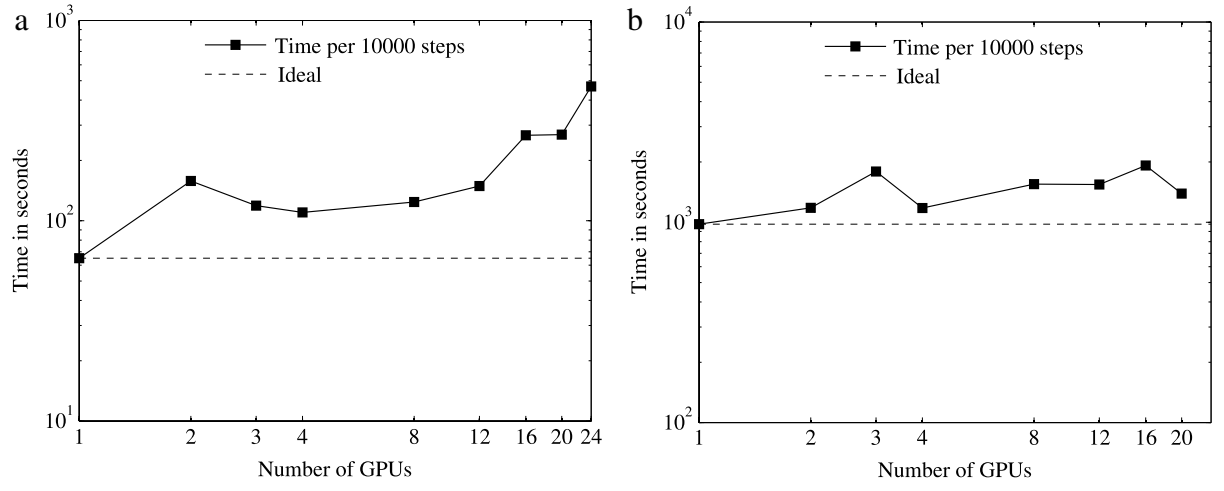

Fig. 9. Weak scaling analysis for (a) 2-D domain with per processor domain of 1024 × (1024 × no. of GPUs) and (b) 3-D domain with per processor domain of 256 × 256 × (256 × no. of GPUs).

and 1 then added to the interface region. Care should be taken that the noise added should not affect the growth of the tip. This is achieved by keeping the amplitude of the noise to be very minimal. A basic approach to add noise can be found in [43,44]. However, in the case of OpenCL, inbuilt random number generators are not available. The noise was generated by using a XOR shift random number generator written as a kernel. They generate the next number in their sequence by repeatedly taking the exclusive OR of a number with a bit shifted version of itself which makes them extremely fast on modern computer architectures. In order to compare the results of the simulations

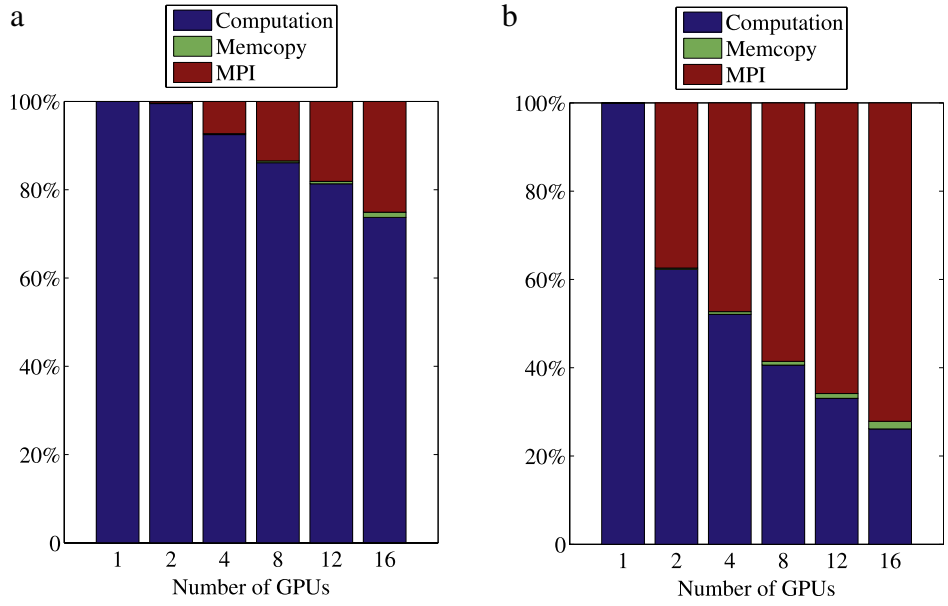

Fig. 10. Strong scaling analysis breakup of the overall execution time as a function of kernel execution time, memory copy time and communication time for (a) 2-D domain of length 4082 × (4082/no. of GPUs) and (b) 3-D domain of length 482 × 482 × (482/no. of GPUs).

with noise on both GPUs, two constant random integers were initialized within the kernel and a seed is generated using global_id(0). The parameters for every thread were obtained by using two random numbers which remain the same initialized in the kernel, the global ID of the thread and a heuristically chosen 32 bit number and then reduced to a float type number between −1 and 1. Introduction of noise along the liquid side of the interface is shown in Fig. 12.

```
kernel void noises( global nodeData *p, int nx, int ny ){
  int randomsx = 45896325;
  int randomsy = 56687692;
  int col = get_global_id(0);
  int row = get_global_id(1);
  int dep = get_global_id(2);

  float minr = -1.0f ;
  float maxr = 1.0f;
  float rnoise;
  int n;
  n=dep*nx*ny + row*nx + col;
  if ( (p[n].phi > 0.1f) && (p[n].phi < 0.5f) ){
    uint seed = randomsx + (362436069%n)*n;
    uint t = seed ^ (seed << 11);
    uint w = randomsy ^ (randomsy >> 19) ^ (t ^ (t >> 8));
    rnoise = w/1000000.0- w/1000000;
    rnoise = -1.0f + rnoise*2.0f;
    p[n].phi = p[n].phi + (0.001*rnoise );
  }
}
```

Code snippet 3: Noise and random number generation in OpenCL

10. Data handling and visualization

The individual chunks of output data from each processor are collected as unformatted binary files from the respective processors. Data stitching is done by reading the individual .bin files and writing them into a single (.mat) file, using `MatIO`, a library for writing and reading .mat files from `C` or `Fortran`, [45]. For 2-D, the output .mat files are visualized in Matlab®. For 3-D, however, Matlab® required a larger memory overhead for reading the data and it was found that **Mayavi** [46] and **ParaView** [47] offer a much simpler and an easier interface for importing and visualizing 3-D data. Mayavi and ParaView both sit on top of VTK (Visualization Toolkit, [48]), an open-source software system for 3-D computer graphics, image processing and visualization. VTK consists of C++ class libraries and several interpreted interface layers including Tcl/Tk, Java, and Python. Both these packages are open-source, multi-platform data analysis and visualization applications and known for quickly building visualizations. Example images of the dendrite simulations in 2-D and 3-D are shown in Fig. 13 and Fig. 14, respectively.

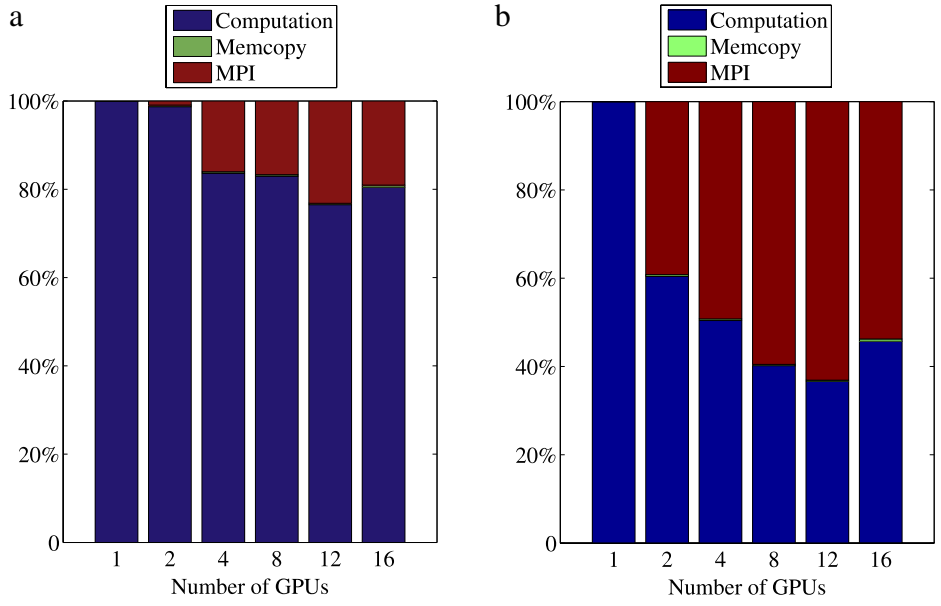

Fig. 11. Weak scaling analysis breakup of the overall execution time as a function of kernel execution time, memory copy time and communication time for (a) 2-D domain with per processor domain of 1024 × (1024 × no. of GPUs) and (b) 3-D domain with per processor domain of 256 × 256 × (256 × no. of GPUs).

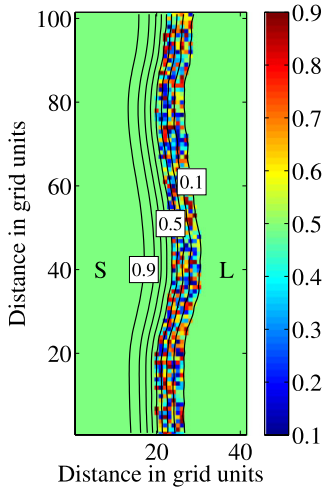

Fig. 12. Noise introduced along the liquid side of the interface given by the phase-field contours ($0.1 < \phi < 0.5$). S represents Solid and L, Liquid.

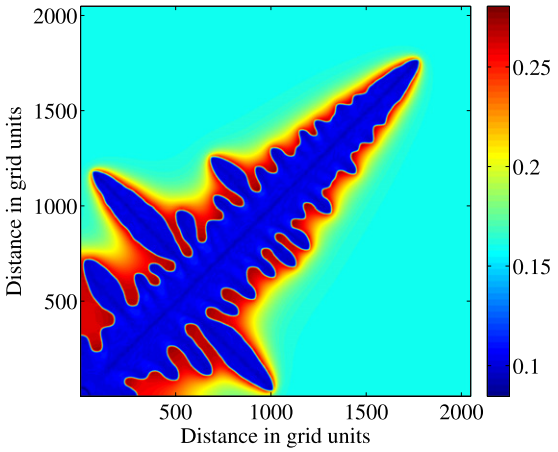

Fig. 13. 2-D Al–Mg alloy dendrite grown in a single GPU and visualized in Matlab®. The temperature of the domain was kept constant at 770 K and the initial composition of the liquid was $c_L = 0.18$. The dendrite was grown by placing a seed with solid composition $c_S = 0.1024$ in lower left corner. Noise introduced along the liquid side of the interface gives rise to the secondary arms.

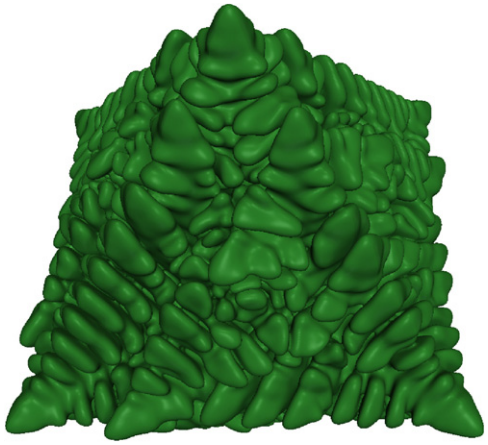

Fig. 14. Al–Mg dendrite grown in the first octant (X, Y, Z > 0) on a 1022 × 1022 × (87 × 12) with a spherical seed of solid composition $c_S = 0.1024$ nucleated at the origin and mirrored about the X = Y = Z = 0 planes taking 6800s for 10 000 timesteps. The temperature of the domain was kept constant at 790 K with the initial composition of the liquid at $c_L = 0.15$. The isosurface shown is at $\phi = 0.5$ and is visualized using ParaView.

11. Conclusion

A stencil based portable implementation of the phase-field equations was presented. A single solver based on the stencil approach could lead to satisfactory scale-up of performance. An OpenCL implementation of the phase-field equations was adopted to perform simulations on different GPU platforms. The two-level parallel (MPI + OpenCL) implementation of the solver across multiple GPUs allowed the simulations to be carried on large domains making realistic microstructure simulations possible.

Acknowledgments

The authors wish to thank the P.G. Senapathy Center for Computing Resources, IIT Madras for extending some of the computing resources used in this study and Prof. K.C. Hari Kumar for useful discussions on the thermodynamic aspects of the model.

Disclaimer

The authors are not in receipt of any funding for this work and have no conflict of interest in publishing this study. This work forms part of the doctoral thesis of P. Gerald Tennyson.

References

[1] L. Gránásy, T. Pusztai, J.A. Warren, Modelling polycrystalline solidification using phase-field theory, J. Phys.: Condens. Matter. 16 (2004) R1205–R1235.
[2] I. Steinbach, Phase-field models in materials science, Model. Simul. Mater. Sci. 71 (2009) 073001.
[3] M. Plapp, Remarks on some open problems in phase-field modelling of solidification, Phil. Mag. 91 (2011) 25–44.
[4] L. Chen, J. Shen, Applications of semi-implicit fourier-spectral method to phase field equations, Comput. Phys. Comm. 108 (1998) 147–158.
[5] B. Nestler, A.A. Wheeler, Phase-field modeling of multi-phase solidification, Comput. Phys. Comm. 147 (2002) 230–233. Proceedings of the Europhysics Conference on Computational Physics Computational Modeling and Simulation of Complex Systems.
[6] Z. Xu, H. Huang, X. Li, P. Meakin, Phase field and level set methods for modeling solute precipitation and/or dissolution, Comput. Phys. Comm. 183 (2012) 15–19.
[7] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, The Landscape of Parallel Computing Research: A View from Berkeley, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.
[8] P. Kegel, M. Steuwer, S. Gorlatch, dOpenCL: towards a uniform programming approach for distributed heterogeneous multi-/many-core systems, in: Parallel and Distributed Processing Symposium Workshops PhD Forum, IPDPSW, 2012 IEEE 26th International, pp. 174–186.
[9] A. Alves, J. Rufino, A. Pina, L. Santos, clOpenCL - supporting distributed heterogeneous computing in HPC clusters, in: I. Caragiannis, M. Alexander, R. Badia, M. Cannataro, A. Costan, M. Danelutto, F. Desprez, B. Krammer, J. Sahuquillo, S. Scott, J. Weidendorfer (Eds.), Euro-Par 2012: Parallel Processing Workshops, in: Lecture Notes in Computer Science, vol. 7640, Springer, Berlin Heidelberg, 2013, pp. 112–122.
[10] R. Aoki, S. Oikawa, T. Nakamura, S. Miki, Hybrid OpenCL: enhancing OpenCL for distributed processing, in: Parallel and Distributed Processing with Applications, ISPA, 2011 IEEE 9th International Symposium on, pp. 149–154.
[11] T. Aoki, S. Ogawa, A. Yamanaka, Multiple-GPU scalability of phase-field simulation for dendritic solidification, Prog. Nucl. Sci. Tech. 2 (2011) 639–642.
[12] T. Stefanski, N. Chavannes, N. Kuster, Hybrid OpenCL-MPI parallelization of the FDTD method, in: Electromagnetics in Advanced Applications (ICEAA), 2011 International Conference on, pp. 1201–1204.
[13] V. Voller, F. Porté-Agel, Moore's law and numerical modeling, J. Comput. Phys. 179 (2002) 698–703.
[14] W. George, J. Warren, A parallel 3D dendritic growth simulator using the phase-field model, J. Comput. Phys. 177 (2002) 264–283.
[15] Y. Suwa, Y. Saito, H. Onodera, Parallel computer simulation of three-dimensional grain growth using the multi-phase-field model, Mat. Trans. 49 (2008) 704–709.
[16] Y. Xu, T. Yang, J. McDonough, K.A. Tagavi, Parallelization of phase-field model for phase transformation problem, in: B. Chetverushkin, A. Ecer, J. Periaux, N. Satofuka, P. Fox (Eds.), Parallel Computational Fluid Dynamics 2003: Advanced Numerical Methods and Applications, pp. 213–218.
[17] Y. Xu, J. McDonough, K.A. Tagavi, Parallelization of phase-field model to simulate feezing in high-Re flow - Multiscale method implementation, in: J. Kwon, A. Ecer, J. Periaux, N. Satofuka, P. Fox, Parallel Computational Fluid Dynamics 2006: Advanced Numerical Methods and Applications, pp. 75–82.
[18] T. Shimokawabe, T. Aoki, T. Takaki, A. Yamanaka, A. Nukada, T. Endo, N. Maruyama, S. Matsuoka, Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer, in: Proceedings of 2011 SC - International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11.
[19] A. Yamanaka, T. Aoki, S. Ogawa, T. Takaki, GPU-accelerated phase-field simulation of dendritic solidification in a binary alloy, J. Cryst. Growth 318 (2011) 40–45.
[20] C. ÓBroin, L. Nikolopoulos, An OpenCL implementation for the solution of the time-dependent Schrödinger equation on GPUs and CPUs, Comput. Phys. Comm. 183 (2012) 2071–2080.
[21] M. Bach, V. Lindenstruth, O. Philipsen, C. Pinke, Lattice QCD based on OpenCL, Comput. Phys. Comm. 184 (2013) 2042–2052.
[22] R. Capuzzo-Dolcetta, M. Spera, A performance comparison of different graphics processing units running direct -body simulations, Comput. Phys. Comm. 184 (2013) 2528–2539.
[23] D.A. Augusto, H.J. Barbosa, Accelerated parallel genetic programming tree evaluation with OpenCL, J. Parallel Distrib. Comput. 73 (2013) 86–100.
[24] P. Rinaldi, E. Dari, M. Vénere, A. Clausse, A Lattice-Boltzmann solver for 3D fluid simulation on GPU, Simul. Model. Pract. Theory 25 (2012) 163–171.

[25] J. Habich, C. Feichtinger, H. Köstler, G. Hager, G. Wellein, Performance engineering for the lattice Boltzmann method on GPGPUs: Architectural requirements and performance results, Comput. Fluids (2012).
[26] K. Rojek, L. Szustak, Parallelization of EULAG model on multicore architectures with GPU accelerators, in: Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part II, PPAM'11, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 391–400.
[27] S.G. Kim, A phase-field model with antitrapping current for multicomponent alloys with arbitrary thermodynamic properties, Acta Mater. 55 (2007) 4391–4399.
[28] R.F. Almgren, Second-order phase field asymptotics for unequal conductivities, SIAM J. Appl. Math. 59 (1999) 2086–2107.
[29] A. Karma, Phase-field formulation for quantitative modeling of alloy solidification, Phys. Rev. Lett. 87 (2001) 115701:1–115701:4.
[30] B. Echebarriaa, R. Folch, A. Karma, M. Plapp, Quantitative phase-field model of alloy solidification, Phys. Rev. E 70 (2004) 061604: 1–22.
[31] A. Karma, W.-J. Rappel, Quantitative phase-field modeling of dendritic growth in two and three dimensions, Phys. Rev. E 57 (1998) 4323–4349.
[32] P.G. Tennyson, Phase-field modelling of microstructure evolution during solidification incorporating CALPHAD description of Gibbs energies (Ph.D. thesis), Indian Institute of Technology Madras, 2013.
[33] A. Dinsdale, SGTE data for pure elements, CALPHAD 15 (1991) 317–425.
[34] Y. Zhong, M. Yang, Z.-K. Liu, Contribution of first principles energetics to Al–Mg thermodynamic modeling, CALPHAD 29 (2005) 303–311.
[35] J. Eiken, B. Böttger, I. Steinbach, Multiphase-field approach for multicomponent alloys with extrapolation scheme for numerical application, Phys. Rev. E 73 (2006) 066122.
[36] S. Posey, A. Koehler, F. Courteille, GPU progress in sparse matrix solvers for applications in computational mechanics, in: 30th ORAP, 2012.
[37] A. Kumar, Isotropic finite-differences, J. Comput. Phys. 201 (2004) 109–118.
[38] A. Choudhury, Quantitative phase-field model for phase-transformation in multi-component alloys (Ph.D. thesis), Karlsruhe Institute for Technology, 2012.
[39] S.V. Patankar, Numerical Heat Transfer and Fluid Flow, Taylor and Francis, 2007.
[40] M. Christen, O. Schenk, Y. Cui, PATUS for convenient high-performance stencils: evaluation in earthquake simulations, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC'12, pp. 11:1–11:10.
[41] R. Hempel, D.W. Walker, The emergence of the MPI message passing standard for parallel computing, Comput. Stand. Interfaces 21 (1999) 51–62.
[42] Oprofile, http://oprofile.sourceforge.net/, 2013.
[43] R. Kobayashi, Modeling and numerical simulations of dendritic crystal growth, Physica D 63 (1993) 410–423.
[44] J.A. Warren, W.J. Boettinger, Prediction of dendritic growth and microsegregation patterns in a binary alloy using the phase field method, Acta Mater. 43 (1995) 689–703.
[45] C. Hulbert, Mat I/O library for writing .mat files from C/C++/Fortran, http://matio.sourceforge.net/, 2013.
[46] P. Ramachandran, Mayavi2: The next generation, in: EuroPython Conference Proceedings, https://svn.enthought.com/enthought/attachment/wiki/MayaVi/m2-paper-epc2005.pdf, 2005.
[47] Sandia National Laboratory, Kitware Inc., Los Alamos National Laboratory, Paraview opensource wiki, http://paraview.org/Wiki/ParaView, 2013.
[48] W. Schroeder, K. Martin, B. Lorensen, Visualization tool kit, http://www.vtk.org/, 2013.