Received: 8 February 2020 | Revised: 20 April 2020 | Accepted: 17 May 2020
DOI: 10.1002/cpe.5905

SPECIAL ISSUE PAPER

Dynamic workload prediction and distribution in numerical modeling of solidification on multi-/manycore architectures

Kamil Halbiniak[1] | Tomasz Olas[1] | Lukasz Szustak[1] | Adam Kulawik[1] | Marco Lapegna[2]

[1]Department of Computer Science, Czestochowa University of Technology, Czestochowa, Poland

[2]Department of Mathematics and Applications, University of Naples Federico II, Naples, Italy

Correspondence
Kamil Halbiniak, Department of Computer Science, Czestochowa University of Technology, Dabrowskiego 69, 42-201 Czestochowa, Poland.
Email: khalbiniak@icis.pcz.pl

Funding information
Ministry of Science and High Education (Poland), Grant/Award Number: 020/RID/2018/19 (position 12); National Science Centre (Poland), Grant/Award Number: UMO-2017/26/D/ST6/00687

Summary
This work is a part of the global tendency to use modern computing systems for modeling the phase-field phenomena. The main goal of this article is to improve the performance of a parallel application for the solidification modeling, assuming the dynamic intensity of computations in successive time steps when calculations are performed using a carefully selected group of nodes in the grid. A two-step method is proposed to optimize the application for multi-/manycore architectures. In the first step, the *loop fusion* is used to execute all kernels in a single nested loop and reduce the number of conditional operators. These modifications are vital to implementing the second step, which includes an algorithm for the dynamic workload prediction and load balancing across cores of a computing platform. Two versions of the algorithm are proposed—with the 1D and 2D maps used for predicting the computational domain within the grid. The proposed optimizations allow increasing the application performance significantly for all tested configurations of computing resources. The highest performance gain is achieved for two Intel Xeon Platinum 8180 CPUs, where the new code based on the 2D map yields the speedup of up to 2.74 times, while the usage of the proposed method with the 2D map for a single KNL accelerator permits reducing the execution time up to 1.91 times.

KEYWORDS
Intel Xeon scalable and KNL processors, load balancing, multicore and manycore, numerical modeling of solidification, OpenMP, workload prediction

1 | INTRODUCTION

The phase-field method is a powerful tool for solving interfacial problems in materials science.[1] It has mainly been applied to solidification dynamics,[2] but it has also been used for other phenomena such as viscous fingering,[3] fracture dynamics,[4] and vesicle dynamics.[1] In our previous paper,[5] an application dedicated to the numerical modeling of alloy solidification, based on the phase-field and generalized finite difference methods, was used as a testbed in the suitability assessment of various programming environments for porting a real-life scientific application to hybrid shared-memory platforms with multicore CPUs and manycore Intel MIC accelerators. Two different cases were studied: with the static and dynamic intensity of computations. In the second case, calculations are performed using a carefully selected group of nodes that is growing in successive time steps. It was discovered that in this case the efficiency of utilizing computing devices falls significantly. Alleviating this limitation requires the development of algorithms enabling the dynamic (at runtime) balancing of workloads across available resources.

The main goal of this study is to ensure efficient data partitioning and workload distribution across computing resources for the solidification application with the dynamic computational intensity. To achieve this goal, we propose algorithms that allow us to manage computing resources during the application execution dynamically. These algorithms are based on predicting the amount of computations performed in the successive time step of a simulation.

The contributions of this article to areas of parallel computing and high performance computing simulations are as follows:

1. A two-step method for optimizing the performance of the parallel application for the phase-field modeling of alloy solidification on multi-/manycore architectures in the case of the dynamic intensity of computations. By reducing the number of conditional operators and modifying the selection criterion in the first step, this becomes possible in the second step to introduce an algorithm for the dynamic workload prediction and load balancing across cores of a computing platform, in successive time steps.
2. Two versions of the algorithm are proposed—with 1D and 2D computational maps used for predicting the computational domain within the grid. They allow reducing superfluous operations by adjusting the computational domain to the domain of simulation (or area of grain growth), as close as possible. These algorithmic solutions ensure a more efficient workload distribution and load balancing across available cores, using mechanisms provided by the OpenMP environment.
3. An experimental evaluation of the proposed optimizations shows that they allow increasing the application performance significantly for all tested configurations of computing resources. The highest performance gain is achieved for two Intel Xeon Platinum 8180 CPUs (56 cores totally), where the usage of 2D map yields the speedup of up to 2.74 times against the basic version, while for a single Intel KNL accelerator this solution permits reducing the execution time up to 1.91 times.

This article is organized in the following way. Section 2 presents related works, while Section 3 outlines the numerical model, as well as describes the basic version of the solidification application and provides the analysis of its characteristics. The proposed method of adaptation of the solidification application with the dynamic intensity of computations to multi-/manycore architectures is introduced in Section 4, while Section 5 describes details of developed algorithms for the workload prediction and load balancing. The next section presents the performance evaluation of the proposed approach on platforms with Intel Xeon CPUs and Intel KNL processors (accelerators), including a comparative analysis of the behavior of different versions of code. Section 7 concludes the article and addresses future works.

2 | RELATED WORKS

A fast grow of computing power[6] permits modeling complex solidification processes. Examples of this trend are the peta-scale phase-field simulation of dendritic solidification performed on the TSUBAME2.0 supercomputer powered by GPUs,[7] as well as large scale phase-field simulations of Ag-Al-Cu ternary eutectic solidification.[8] The approach proposed in the latter work was extended in Reference 9 and 10, which focused on optimizing the simulation performance, as well as automating the process of code generation. Thus, the presented research is a part of the global tendency to use modern computing systems for modeling the phase-field phenomena. There are many papers devoted to modeling dendritic solidification phenomena that use such approaches as cellular automata, finite element and finite difference methods.[11-13] The significant highlight of this work is the usage of the generalized finite difference method,[14] which allows us to model phenomena where the distribution of nodes in grids is diversified—concentrated in border areas of the inter-phase, and sparse in areas with a low diffusivity or already solidified.

In our previous works,[5,15-18] we dealt with porting and optimization of the phase-field simulations of alloy solidification on shared-memory computing platforms with Intel MIC accelerators, without significant modifications of the original application code. The last two works took advantages of using both CPUs and KNCs (or KNLs) for the parallel execution of the application, while in the rest of our papers, computational workloads were assigned to accelerators or CPUs only. Furthermore, the usage of different programming environments was studied—Intel Offload with OpenMP in References 15 and 18, OpenMP Accelerator Model in Reference 18, and hStreams with OpenMP in Reference 17. Based on these studies, it became possible[5] to carry out a suitability assessment of these environments for porting the solidification application to shared-memory platforms with multi-/manycore architectures.

This assessment revealed the satisfying efficiency of porting the application for modeling problems with the static intensity of computations. However, in the case of the dynamic intensity, the efficiency of utilizing computing resources drops significantly. In particular, for the static computational intensity, the usage of four KNL processors allows us to speed up the application about 3.1 times against the configuration with a single KNL. At the same time, the performance results achieved in the case of the dynamic intensity indicated pretty large room for further optimizations, as the utilization of four KNL devices gives only the speedup of about 1.9 times. Alleviating this shortcoming requires enabling the dynamic (at runtime) balancing of workloads across computing resources.

Providing appropriate load balancing across available resources plays a significant role in optimizing the overall performance of computations on parallel architectures.[19] It allows maximizing the performance of applications by keeping processor idle time and communication overheads as low as possible. Numerous mechanisms for ensuring efficient workload and data distribution have been developed.[20] In applications with constant workloads, *static* load balancing can be used as a pre-processor to the computation. A common approach to static load balancing is based on data partitioning.[21-24] However, the load balancing problem is not solved completely. It is especially evident in the scientific applications where the workloads are unpredictable or change during the execution. Such codes require programmers to develop dynamic strategies of load balancing that adapt the workload distribution based on changes monitored during the application execution. Dynamic algorithms (such as task scheduling and work-stealing) do not require a priori information about execution but may incur significant communication overheads due to data migration. Dynamic algorithms often use static partitioning for their initial step.[23]

One of the possible ways[20] to implement dynamic load balancing is the usage of software toolkits such as Zoltan[25] and ParMETIS.[26] In this work, dynamic load balancing is added directly to the application. Compared to using software toolkits, the main advantage of such an approach is its low overhead because load balancing is integrated directly with mechanisms provided by the OpenMP programming environment.[27] The concept of this integration was drafted in our previous paper,[28] where the algorithm for the dynamic workload prediction and load balancing with the 1D computational map was proposed, but without studying its implementation and efficiency. These topics are studied in this work, which also presents and investigates another algorithm for workload prediction and load balancing that uses the 2D computational map.

3 | NUMERICAL MODELING OF SOLIDIFICATION WITH DYNAMIC INTENSITY OF COMPUTATION

3.1 | Numerical model

In the numerical modeling problem studied in the article, a binary alloy of Ni-Cu is considered as a system of the ideal metal mixture in liquid and solid phases. The numerical model assumes the dendritic solidification process[11,29] in the isothermal conditions with constant diffusivity coefficients for both phases. It allows us to use the field-phase method defined by Warren and Boettinger.[30] In the model, the growth of microstructure during the solidification is determined by solving a system of two PDEs. The first equation defines the phase content ϕ:

$$\frac{1}{M_\phi}\frac{\partial \phi}{\partial t} = \varepsilon^2 \left[\nabla \cdot (\eta^2 \nabla \phi) + \eta\eta' \left(sin(2\theta)\left(\frac{\partial^2\phi}{\partial y^2} - \frac{\partial^2\phi}{\partial x^2}\right) + 2cos(2\theta)\frac{\partial^2\phi}{\partial x \partial y}\right)\right]$$
$$- \frac{1}{2}(\eta'^2 + \eta\eta'')\left(-cos(2\theta)\left(\frac{\partial^2\phi}{\partial y^2} - \frac{\partial^2\phi}{\partial x^2}\right) + 2sin(2\theta)\frac{\partial^2\phi}{\partial x \partial y} - \frac{\partial^2\phi}{\partial x^2} - \frac{\partial^2\phi}{\partial y^2}\right) - cH_B - (1-c)H_A - cor\ , \tag{1}$$

where M_ϕ is defined as the solid/liquid interface mobility, ε is a parameter related to the interface width, η is the anisotropy factor, H_A and H_B denote the free energy of both components, and *cor* is the stochastic factor which models thermodynamic fluctuations near the dendrite tip. The coefficient θ is calculated as follows:

$$\theta = \frac{\partial \phi}{\partial y} \Big/ \frac{\partial \phi}{\partial x}. \tag{2}$$

The second equation defines the concentration *c* of the alloy dopant, which is one of the components of the alloy:

$$\frac{\partial c}{\partial t} = \nabla \cdot D_c \left[\nabla c + \frac{V_m}{R} c(1-c)(H_B(\phi, T) - H_A(\phi, T))\nabla\phi\right],$$

where D_c is the diffusion coefficient, V_m is the specific volume, and *R* is the gas constant.

Transforming these formulas into differential equations is described in our previous paper.[18] The solutions of differential equations are then obtained[31,32] using the generalized finite difference method and explicit scheme of calculations with a small time step Δ (we select $\Delta t = 10^{-7}$ [s]). For approximating values of partial derivatives in Equations (1) and (2), the second-order Taylor expansion is adopted. In this numerical scheme, an *n*-point star (or stencil) is used to provide the "best" approximation of derivatives in the central point of the star.[18] As a result, this approach can be used for both regular and irregular grids.

The resulting computations[18] belong to the group of forward-in-time, iterative algorithms since all the calculations performed in the current time step (*t*+1) depend on results determined in the previous step *t*. The application code consists of two main blocks of computations, which are responsible for determining either the phase content ϕ or the dopant concentration *c*. In the model, the values of ϕ and *c* are determined for grid nodes distributed across a considered domain. In consequence, the values of derivatives in all nodes of the grid are calculated at every time step. In the application studied in this article, a 2D regular grid is used with nodes distributed uniformly across a square domain (Figure 1). Hence, each internal node of the grid has $n = 8$ neighbor grid elements participating in computations for the corresponding stencils.

In our previous work,[18] two different cases were introduced – with the static and dynamic intensity of computations. In the first case, the workload of computing resources is constant during the application execution, since a constant number of equations is solved. This assumption corresponds to modeling problems in which the variability of solidification phenomena in the whole domain has to be considered. In the second case, the model is able to solve differential equations only in part of nodes, which is changing during the simulation following the growth of microstructure. The use of a suitable selection criterion allows reducing the amount of computations significantly. The consequence is a significant workload imbalance since the selection criterion is calculated after the static partitioning of the grid nodes across computing resources.

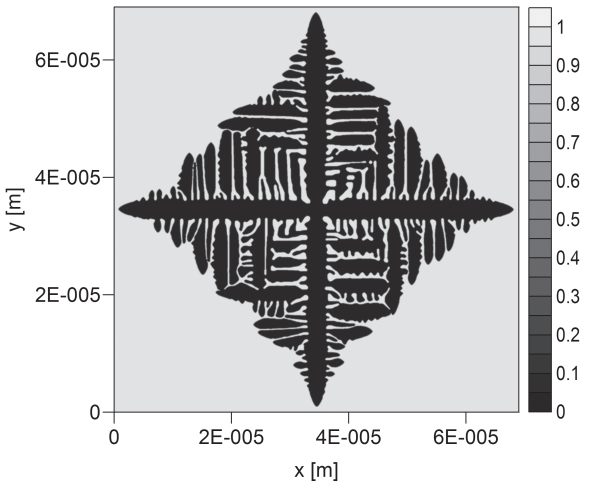

FIGURE 1 Phase content for the simulated time $t_S=2.75\times10^{-3}s$

3.2 | Basic version of solidification application with dynamic intensity of computations

Listing 1 shows the basic code for the computational core of the application implementing the numerical model of solidification with the dynamic intensity of computations. This listing corresponds to a single time step. The presented code allows the partial differential equations to be solved not only for regular, but also irregular grids. In this code, the memory management is organized according to the SOA (structure of arrays) layout with the computations executed on one-dimensional arrays. For example, `node_conc[i]` contains value of the dopant concentration for the *i-th* node, while `node_Dc[i]` corresponds to value of the diffusion coefficient for this node. The transformation to the SoA organization of data from the original array of structures (AoS) layout, which was used in the original code, corresponds to the first step of the design methodology proposed in work[5] and is out of the scope of this article. But it is worth emphasizing that this transformation is vital for developing an efficient code for the solidification application.

```
#pragma omp parallel {
   // Computations in boundary nodes for the dopant concentration
  #pragma omp for
   for(int i=0; i<grid_size; ++i) {
      if(node_isBoundary[i])
         Kernel1;
   }
   // Computations in internal nodes for the dopant concentration
   #pragma omp for
   for(int i=0; i<grid_size; ++i) {
      if(!node_isBoundary[i])
         Check_Condition_1;
         Kernel2;
   }
   // Computations in boundary nodes for the phase content
   #pragma omp for
   for(int i=0; i<grid_size; ++i) {
      if(node_isBoundary[i])
         Kernel3;
   }
   // Computations in internal nodes for the phase content
   #pragma omp for
   for(int i=0; i<grid_size; ++i) {
      if(!node_isBoundary[i])
         Check_Condition_2;
         Kernel4;
   }
   // Loop completing computations within a time step
   #pragma omp for
   for(int i=0; i<grid_size; ++i) {
      node_Fi0[i] = node_Fi[i];
      node_conc0[i] = node_conc[i];
      node_Dc[i] = Ds+fP(node_Fi[i])*(Dl-Ds);
   }
}
```

Listing 1: General scheme of executing a single time step for the basic version of the application with dynamic intensity of computations

All computations in the application are organized as five loops. Two of them, with kernels K_1 and K_3 are responsible for calculations executed in the boundary nodes, while the other two loops, with kernels K_2 and K_4, perform computations for the internal part of the grid. The last loop completes the execution within a single time step. In the basic version of the application shown in Listing 1, each loop iterates over all nodes of the grid. The selection of the boundary and internal nodes is performed using four conditional operators (lines 5, 11, 18, and 24).

Listings 2 and 3 present code snippets corresponding to kernels K_1 and K_2 that are responsible for determining the dopant concentration for the boundary and internal nodes of the grid. For a given grid node *i*, each of these kernels iterate over the neighbors `j` of the node. The indices `neighbor_idx` of these neighbor nodes are retrieved from the array `node_e[]` describing the geometry of the whole grid. This array is read from a configuration file before executing the application. The size of the array is the product of the total number of grid nodes (`grid_size`) and the maximum number of neighbors (`max_neighbors`). In the studied application, `max_neighbors` = 8.

```
const int offset = i*max_neighbors;
double d0(0.0), d2(0.0);
double z[max_neighbors];
/.../
for(int j=0; j<neighbors_count[i]; ++j) {
   // Stencil computations used to determine partial derivatives
   z[j] = (node_g2[offset+0]*node_cosAlf[i]+
          node_g2[offset+2]*node_cosBet[i])*node_hx[offset+j]+
          (node_g2[offset+1]*node_cosAlf[i]+
          node_g2[offset+3]*node_cosBet[i])*node_hy[offset+j]
   const int neighbor_idx = node_e[offset+j];
   d2 += node_conc0[neighbor_idx]*z[j];
   /.../
}
// Computations executed within nodes
/.../
node_conc[i] = ...;
```

Listing 2: Kernel K_1

```
 1  const int offset = i*max_neighbors;
 2  const int gOffset = i*25;
 3  double d1xCj(0.0), d1xDcj(0.0), d1xFj(0.0);
 4  double zx[max_neighbors];
 5  /.../
 6  for(int j=0; j<neighbors_count[i]; ++j) {
 7     // Stencil computations
 8     // 3 of all 15 stencils used in kernel K2
 9     zx[j] = 1/pow(node_h[offset+j],2*m)*
10            (node_g[gOffset+0]*node_hx[offset+j]+
11            node_g[gOffset+1]*node_hy[offset+j]+
12            0.5*node_g[gOffset+2]*node_hx[offset+j]*node_hx[offset+j]+
13            0.5*node_g[gOffset+3]*node_hy[offset+j]*node_hy[offset+j]+
14            node_g[gOffset+4]*node_hx[offset+j]*node_hy[offset+j]);
15     const int neighbor_idx = node_e[offset+j];
16     d1xCj += zx[j]*node_conc0[neighbor_idx];
17     d1xDcj += zx[j]*node_Dc[neighbor_idx];
18     d1xFj += zx[j]*node_Fi0[neighbor_idx];
19     /.../
20  }
21  // Computations performed within nodes
22  /.../
23  node_conc[i] = ...;
```

Listing 3: Kernel K_2

The calculations performed in these two kernels are focused around stencil computations. While kernel K_1 includes only a single stencil, 15 various stencils can be distinguished in the case of kernel K_2. Three of them are presented in Listing 3 in lines 16, 17, and 18. Each of these stencils updates a corresponding variable in the *i*th node utilizing values of a certain variable in its neighbor nodes and some temporary variable. In Listing 3, this temporary variable denoted as `zx[j]` is computed in lines 9-14 using only data that belongs to the *i*th node. As shown in Listings 2 and 3, some calculations are carried out outside the loop iterating over *j*. These calculations end with determining the value of the dopant concentration in the *i*th node for a given time step. The structure of kernels K_3 and K_4 is analogous; they contain a single stencil and five stencils, respectively. These two kernels are responsible for calculating the phase content.

The selection criterion affects the overall performance of the application decisively. In the basic version of the application, the selection criterion is checked during the execution of kernels K_2 and K_4, involving two conditional operators. As a result, six conditional operators are performed for a single time step. Moreover, the presence of the selection criterion leads to the necessity of analysis practically all grid nodes (excluding boundary ones), and not just nodes within the area of grain growth. For kernel K_2, the selection criterion is presented in Listing 4. This criterion is based on checking the absolute values of differences between values of both dopant concentration and phase content in a given node *i* and its neighbors. If none of these differences exceeds a given small threshold value 10^{-10}, then the value of the dopant concentration in the *i*th node remains unchanged. For kernel K_4, the selection criterion is specified analogously.

```
const int offset = i*max_neighbors;
// Calculating differences deltaD and DeltaP for a single stencil
double deltaD=0.0;
double deltaP=0.0;
for(int j=0; j<neighbors_count[i]; ++j) {
   const int neighbor_idx = node_e[offset+j]];
   deltaD += fabs(node_conc0[i]-node_conc0[neighbor_idx]);
   deltaP += fabs(node_Fi0[i]-node_Fi0[neighbor_idx]);
}
// Checking if the values of deltaD and DeltaP are non-zero
if((deltaD<1e-10) && (deltaP<1e-10)) {
   node_conc[i] = node_conc0[i];
   continue;
}
```

Listing 4: Check_Condition_1: selection criterion for kernel K_1

Summarizing, computations corresponding to kernels K_1-K_4 of the application are organized as two nested loops. The outer one iterates over the grid nodes, while the inner loop iterates over neighbors of each node. This loop corresponds to stencil computations used for the determination of partial derivatives. Since the indices of all neighbors of a given node are kept in the array `node_e[]` describing the grid, this solution allows the patterns of all 22 stencils to be determined during the application execution. Furthermore, the structure of these kernels enables their parallelization using **omp parallel for** directive of OpenMP[27] for the outer loop.

3.3 | Analysis of solidification application with dynamic intensity of computations

The solidification application with the dynamic intensity assumes that calculations are performed for a carefully selected group of nodes. The usage of the selection criterion permits reducing the amount of computations. At the same time, the intensity of computations increases significantly during the application execution, as is shown in Figure 2, which illustrates the growth of the simulation domain (black area) in successive time steps.

Figure 3 presents characteristics of the application during its execution. The upper plot shows the total number of grid nodes that are processed in successive time steps, while the bottom plot illustrates the execution time measured after completing subsequent time steps. The analysis of Figure 3a allows us to conclude that for the first 25% of the execution time, only up to 6% of the grid nodes are actually processed. For the next 50% of the execution time, the number of processed nodes does not exceed 26% of all grid nodes, and finally, it achieves just over 50%.

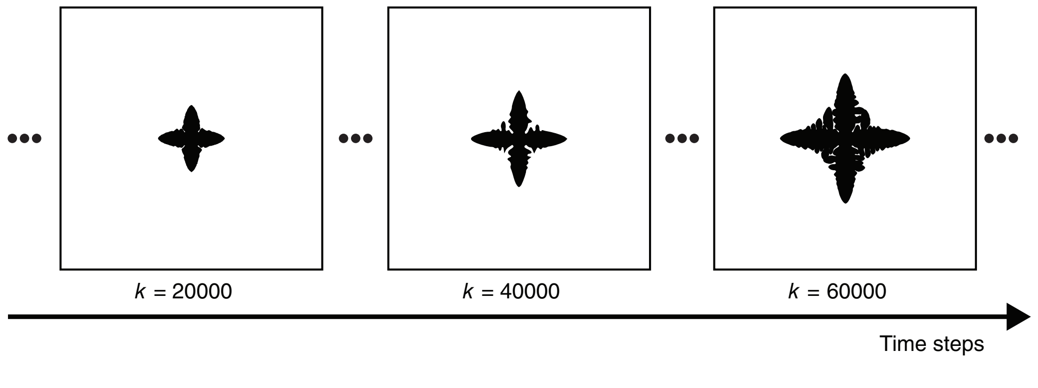

FIGURE 2 Growth of simulation domain (black area) for successive time steps

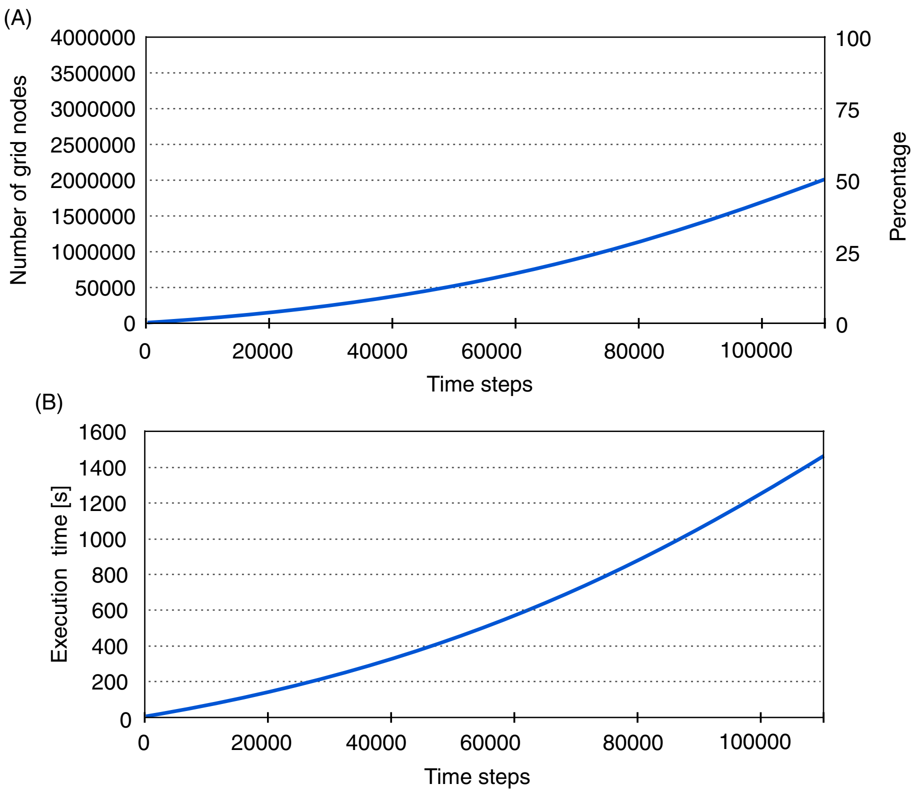

FIGURE 3 Characteristics of the solidification application with dynamic intensity of computations for grid of size $N = 2000$: A, number of grid nodes computed in successive time steps, and, B, total time of computations measured after completing successive time steps on a platform equipped with two Intel Xeon Platinum 8180 CPUs

A key way to improve the overall performance of the considered application is reducing superfluous operations, as well as providing an optimized workload distribution across computing resources. To this end, a significant modification of the application code is required as the first step. In fact, the previous subsection shows that six conditional operators are executed for a single time step. Moreover, this leads to the necessity of analysis of the whole grid, while only the nodes within the area of the grain growth should be considered in the optimal case. Thus, the application code after its transformation should allow reducing the number of conditional operators, and adjusting the domain of computation to the domain of simulation, which corresponds to the area of grain growth shown as the black area in Figure 2, as close as possible.

4 | OPTIMIZATIONS OF SOLIDIFICATION APPLICATION WITH DYNAMIC INTENSITY OF COMPUTATIONS FOR MULTI-/MANYCORE ARCHITECTURES

4.1 | General concept of adaptation

Based on the results of the application analysis presented in the previous section, we develop a method for optimizing the solidification application with the dynamic intensity of computations on multi-/manycore architectures with shared memory. This method includes two basic steps:

1. Modification of the application code using the *loop fusion* technique, in order to reduce the number of conditional statements. This modification is vital to implement the second step.
2. Introduction of an algorithm for predicting the domain of computations in successive time steps, in order to:

- reduce the amount of operations required to check the selection criterion;
- ensure an efficient workload distribution and load balancing across resources of the computing platform.

4.2 | Step 1: Modification of computational scheme

The first step of the proposed method includes the modification of the application source code using the *loop fusing* technique.[33] Listing 5 presents the modified code for a single time step of the solidification application. In contrast to the basic version of code (Listing 1), all workloads of the modified version are executed in a single nested loop. Such a solution allows us to reduce the number of conditional statements used to checking the selection criterion, as well as the number of conditional operators corresponding to selecting boundary and internal nodes (from four operators to a single one). Additionally, the copying of data, which is performed within the loop completing the execution of each time step, is replaced by swapping arrays using pointers.

```
#pragma omp parallel for
for(int i=0; i<grid_size; ++i) {
     Check_Condition;
    // Computations performed within a grid node
    if(node_isBoundary[i]) {
        // Execution of the kernels K1 and K3 corresponding to the boundary nodes
        Kernel1;
        Kernel3;
    }
    else {
        // Execution of the kernels K2 and K4 corresponding to the internal nodes
        Kernel2;
        Kernel4;
    }
}
// Completing computations with swaping arrays using pointers
swap(node_conc, node_conc0);
swap(node_Fi, node_Fi0);
swap(node_Dc, node_Dc0);
```

Listing 5: Solidification application with dynamic intensity of computations after the first step of the proposed method

This transformation of source codes also requires a suitable modification of the selection criterion. Listing 6presents the selection criterion for the modified code. It corresponds to the fusion of the selection criteria used for kernels K_2 and K_4 in the basic code. Moreover, due to removing the loop completing each time step, additional calculations are performed in the modified selection criterion (Line 14 in Listing 6).

```
 1  const int offset = i*max_neighbors;
 2  // Calculating differences deltaD and DeltaP for grid nodes within a single stencil
 3  double deltaD=0.0;
 4  double deltaP=0.0;
 5  for(int j=0; j<neighbors_count[i]; ++j) {
 6      const int neighbor_idx = node_e[offset+j]];
 7      deltaD += fabs(node_conc0[i]-node_conc0[neighbor_idx]);
 8      deltaP += fabs(node_Fi0[i]-node_Fi0[neighbor_idx]);
 9  }
10  // Checking values of deltaD and DeltaP
11  if((deltaD<1e-10) && (deltaP<1e-10)) {
12      node_conc[i] = node_conc0[i];
13      node_Fi[i] = node_Fi0[i];
14      node_Dc0[i] = Ds+fP(node_Fi[i])*(Dl-Ds);
15      continue;
16  }
```

Listing 6: Check_Condition: selection criterion after modification

In the application code shown in Listing 5, the selection criterion is still calculated for all nodes of the grid. However, this code permits us to introduce an algorithm for workload prediction. Such an algorithm is responsible for adjusting the domain of computations to the domain of simulation. In consequence, it allows ensuring more efficient workload distribution and load balancing across resources of a computing platform.

4.3 | Step 2: Prediction of the domain of computations

The second step of the proposed method embraces the prediction of the computational domain. It is responsible for determining the computational workload for successive time steps of the simulation. In practice, it allows adjusting the computational domain to the domain of simulation (black area in Figure 2). The computational domain refers to the grid area wherein the primary computations (including stencil workloads) are performed, as well as the selection criterion is checked. The prediction of the workload for the next time step (*t+1*) is based on the results of computations performed in the current step *t*. In practice, if values of variables in a grid node are computed in a given time step, this node and its neighbors are taken into consideration when predicting the computational domain for the next step.

Predicting the computational domain is vital for ensuring the efficient workload distribution across computing resources (cores). In the basic version of the application, the selection criterion is checked for the whole grid. It leads to an undesirable situation (Figure 4) when some cores spend most of the time on checking the selection criterion, while only a part of cores perform primary computations within grid nodes. The modification of the application code accomplished in the first step allows the usage of the algorithm for workload prediction, to resolve this problem. As a result, the selection criterion will be checked only for the nodes within the predicted domain of computation. This solution ensures a more efficient workload distribution across cores since each core will perform primary computations within the domain of simulation in successive time steps. Figure 5 illustrates two versions of the algorithm for workload prediction proposed in this work.

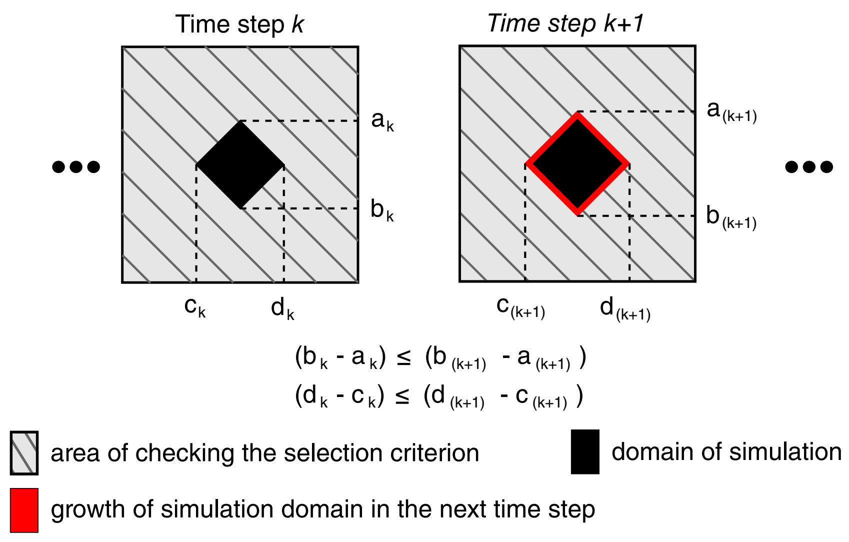
Time step *k*
Time step *k+1*
a_k
b_k
c_k
d_k
$a_{(k+1)}$
$b_{(k+1)}$
$c_{(k+1)}$
$d_{(k+1)}$
$(b_k - a_k) \le (b_{(k+1)} - a_{(k+1)})$
$(d_k - c_k) \le (d_{(k+1)} - c_{(k+1)})$
area of checking the selection criterion
domain of simulation
growth of simulation domain in the next time step

FIGURE 4 Areas of simulation and checking the selection criterion in successive time steps, for the basic version as well as the version obtained after the first step

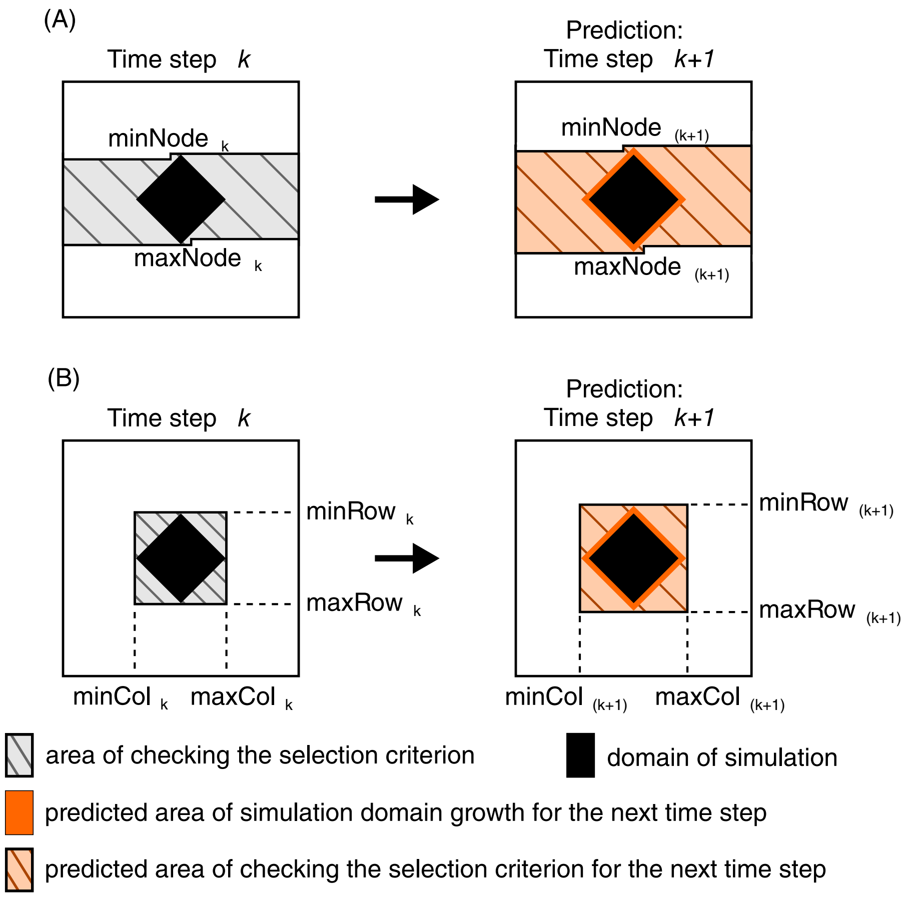

FIGURE 5 Predicting domain of computation in successive time steps with, A, 1D and, B, 2D maps

5 | ALGORITHMS FOR WORKLOAD PREDICTION AND LOAD BALANCING

5.1 | Prediction with 1D map

The first version of the workload prediction algorithm (Figure 5A) is based on the representation of grid nodes using a 1D array. Thus, the predicted area wherein the selection criterion is checked for a given time step is specified by two coordinates corresponding, respectively, to the beginning (`minNode`) and end (`maxNode`) of the area. Listing 7 presents the algorithm for determining these coordinates for the next times step (*t+1*) of simulation while executing the current step *t*. The analysis of Listing 7 permits us to conclude that overheads introduced by the prediction algorithm are negligible. In particular, the integer calculations performed in Lines 8-9 can be executed by integer ALU units of cores simultaneously with the primary computations performed by floating-point units. For the first time step (*t*=1), the area of checking the selection criterion includes the entire grid. In this case, `minNode = 0` and `maxNode = grid_size`. Then, starting from the second time step (*t*=2), the selection criterion is checked only within the area adjusted to the domain of simulation.

```
 1  int min_temp = minNode;
 2  int max_temp = maxNode;
 3  #pragma omp parallel for reduction(min:min_temp) reduction(max:max_temp)
 4  for(int i=minNode; i<maxNode; ++i) {
 5     const int offset = i * max_neighbors;
 6     // Checking the selection criterion
 7     Check_Condition;
 8     min_temp = min(i, min_temp);
 9     max_temp = max(i, max_temp);
10     // Primary computations
11     if(node_isBoundary[i]) {
12        // Execution of kernels K1 and K3 corresponding to the boundary nodes
13        Kernel1;
14        Kernel3;
15     }
16     else {
17        // Execution of kernels K2 and K4 corresponding to the internal nodes
18        Kernel2;
19        Kernel4;
20     }
21  }
22  if(min_temp - haloSizeTop < minNode)
23     minNode = min_temp - haloSizeTop;
24  if(max_temp + haloSizeDown > maxNode)
25     maxNode = max_temp + haloSizeDown;
26  // Completing computations with swapping arrays using pointers
27  /.../
```

Listing 7: Implementation of a single time step with workload prediction using 1D map

For parallelizing the execution of code with the 1D map, **omp parallel for** directive is used for the outer loop, which iterates over nodes of the grid. Ensuring efficient load balancing across available cores is based on the fact that this directive now embraces not the entire grid, but only the interval from `minNode` to `maxNode`. An essential role in adapting the way of distributing the workload across cores to properties of a particular computing platform plays the usage of a suitable scheduling clause. It is responsible for controlling the policy of assigning loop iterations to cores while executing **omp parallel for** directive. It was evaluated experimentally that the dynamic scheduling option with the size of chunks equal to the size of rows of the grid is the most efficient for all platforms studied in this article. This setup provides the best overall performance of computations—not only for the KNL processor but also for configurations with one and two CPUs.

5.2 | Prediction with 2D map

In the second version of the algorithm for workload prediction (Figure 5B), the computational domain is approximated using a rectangle which embraces the rows and columns of the grid. This rectangle is described by four coordinates:

$$\langle minRow, maxRow, minCol, maxCol \rangle, \qquad (3)$$

that correspond to the minimum and maximum index of row and column of the rectangular area, which includes both the domain of executing the primary computations and area of checking the selection criterion. The algorithm responsible for determining values of these coordinates is shown in Listing 8, with *minRow*=0, *minRow*=*mSize*, *minCol*=0, *maxCol*=*nSize* at the beginning of simulation, where *mSize* and *nSize* denote the number of rows and columns in the grid, respectively. Like the previous version, the usage of the 2D map introduces low performance overheads.

```
int minRow_temp = minRow;
int maxRow_temp = maxRow;
int minCol_temp = minCol;
int maxCol_temp = maxCol;
#pragma omp parallel for reduction(min:minRow_temp,minCol_temp) reduction(max:maxRow_temp,maxCol_temp)
for(int r=minRow; r<maxRow; ++r) {
   for(int c=minCol; c<maxCol; ++c) {
      const int offset = (r * nSize + c) * max_neighbors;
      // Checking the selection criterion
      Check_Condition;
      minRow_temp = min(r, minRow_temp);
      maxRow_temp = max(r, maxRow_temp);
      minCol_temp = min(c, minCol_temp);
      maxCol_temp = max(c, maxCol_temp);
      // Primary computations
      if(node_isBoundary[i]) { // Execution of kernels K1 and K3 corresponding to the boundary nodes
         Kernel1;
         Kernel3;
      }
      else { // Execution of kernels K2 and K4 corresponding to the internal nodes
         Kernel2;
         Kernel4;
      }
   }
}
if(minRow_temp - haloSizeTop < minRow)
   minRow = minRow_temp - haloSizeTop;
if(maxRow_temp + haloSizeDown > maxRow)
   maxRow = maxRow_temp + haloSizeDown;
if(minCol_temp - haloSizeLeft < minCol)
   minCol = minCol_temp - haloSizeLeft;
if(maxCol_temp + haloSizeRight > maxCol)
   maxCol = maxCol_temp + haloSizeRight;
// Completing computations with swaping arrays using pointers
/.../
```

Listing 8: Implementation of a single time step with workload prediction using 2D map

In the case of the 2D map, ensuring efficient load balancing is based on the fact that **omp parallel for** directive iterates over a subset of row indices in the range from `minRow` to `maxRow`. Each row includes only the grid nodes with column indices in the interval from `minCol` to `maxCol`. Similar to the 1D version of the map, an essential role in optimizing the overall performance plays selecting a proper scheduling clause. Again for all studied platforms, it was evaluated experimentally that the dynamic scheduling option yields the best performance, assuming the size of chunks equal to 1. This size corresponds to a single row of the predicted 2D domain.

5.3 | Comparison of the basic version and versions with 1D and 2D maps

Figure 6 shows the comparison of the number of grid nodes analyzed, using the selection criterion, by the basic version of the application (Listing 1), and versions with 1D and 2D maps (Listings 7 and 8, respectively). These numbers are calculated for the example illustrated in Figures 2 and 3. The amount of nodes participating in the primary computations in successive time steps of the simulation is also presented in Figure 6 (this amount is the same for all the versions). The introduction of workload prediction allows reducing significantly the amount of operations required by the selection criterion. This conclusion applies especially to the initial phase of the simulation.

For the basic version of the code (Listing 1) and the version shown in Listing 5, the selection criterion is calculated for all nodes of the grid, in successive time steps. At the same time, during the first 25% of the execution time, the amount of grid nodes analyzed by 1D and 2D maps do not exceed, respectively, 30% and 9% of all nodes. For the next 50% of the execution time, the number of analyzed nodes does not exceed 55% for the 1D map and 30% for the 2D map. In the final phase of the simulation, which includes only 2000 time steps, the selection criterion is checked for all nodes.

Figure 6 also shows that the usage of the 2D map allows significantly reducing the difference between the number of nodes analyzed with the selection criterion and the number of nodes participating in the primary computations against the 1D version of the map. For example, in the time step t=30 000, only 20% of nodes analyzed by the 1D version participate in the primary computations. At the same time, over 60% of examined nodes participate in the primary computations for the 2D map.

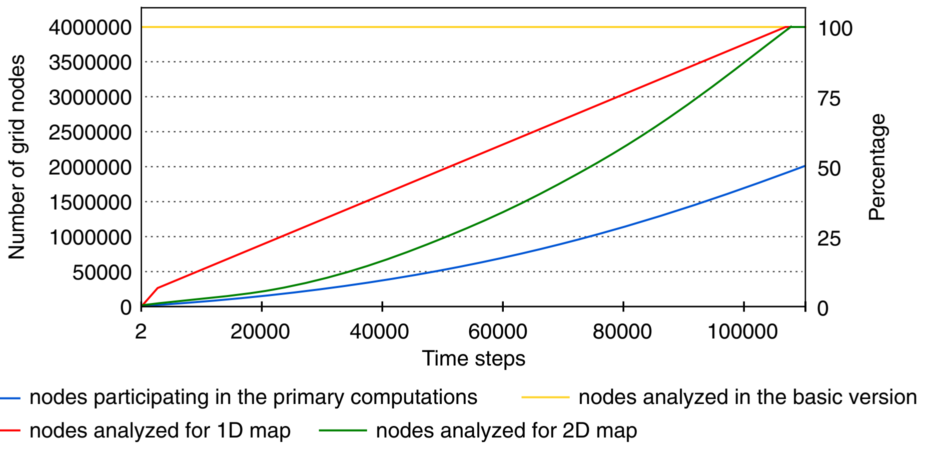

FIGURE 6 Comparison of the number of nodes analyzed by the basic versions of the application and versions with 1D and 2D maps, as well as the number of grid nodes participating in the primary computations in successive time steps

6 | PERFORMANCE RESULTS

This subsection presents performance results obtained for the proposed codes assuming the double precision floating-point format. The application that is tested corresponds to the example shown in Figure 2. It is executed on two platforms (Table 1):

1. SMP consisting of two Intel Xeon Platinum 8180 CPUs (Intel Xeon Scalable Processor architecture), totally with 56 cores;
2. single Intel Xeon Phi 7250F processor with the KNL architecture, consisting of 68 cores.

The KNL processor is used in the *quadrant* clustering mode, with the MCDRAM memory configured in the *flat* mode.[34] All the benchmarks are compiled using the Intel `icpc` compiler (ver. 19.0.1) with `-O3` and `-xMIC-AVX512` flags for the KNL processor, and `-xCore-AVX512 -qopt-zmm-usage=high` flags for Intel Xeon CPUs. To ensure the reliability of benchmark results, the measurements of the execution time are repeated r=10 times, and the median value of measurements is used finally.

Tables 2 and 3 present the total execution time achieved for the four versions of the application: (i) basic version (T_B), (ii) version corresponding to the first step of the proposed adaptation method (T_M), and versions with (iii) 1D (T_{1D}) and (iv) 2D (T_{2D}) maps. The tests are performed for 110 000 time steps, and two grid sizes: 2000×2000 and 3000×3000, with the following configurations of computing resources:

1. single KNL processor;
2. single Intel Xeon Platinum 8180 CPU;
3. two Intel Xeon Platinum 8180 CPUs.

TABLE 1 Specification of tested platforms

	Intel Xeon Platinum 8180 (SKL)	Intel Xeon Phi 7250F (KNL)
Number of devices	2	1
Number of cores per device	28	68
Number of threads per device	56	272
Base frequency [GHz]	2.5	1.4
(AVX frequency)	(1.7)	(1.2)
SIMD width [bits]	512	512
AVX peak for DP [GFlop/s]	3046,4	2611,2
LLC[a] size per platform [MB]	77	34
Memory size per platform	512GB DDR4	16GB MCDRAM
		96GB DDR4
Memory bandwidth [GB/s]	119.2	MCDRAM: 400+
		DDR4: 115.2

[a] LLC (last level cache) corresponds to aggregated L2 caches for KNL, and L3 cache for CPUs.

TABLE 2 Total execution times T_B,T_M,T_{1D},T_{2D} (in seconds) and speedups achieved for the following versions of the solidification application with the dynamic intensity of computations: (i) basic version, (ii) version obtained after the first step of the adaptation method, and versions with (iii) 1D and (iv) 2D maps, assuming the grid of size 2000 × 2000

Computing resources	T_B	T_M	S_M	T_{1D}	S_{1D}	T_{2D}	S_{2D}	S^f_{1D}	S^f_{2D}
1×KNL	1661	1078	1.54	923	1.17	872	1.24	1.80	1.91
1×SKL	1785	1001	1.78	789	1.27	740	1.35	2.26	2.41
2×SKL	1456	837	1.74	619	1.39	531	1.62	2.35	2.74

TABLE 3 Total execution times (in seconds) and speedups obtained for different versions of the solidification application with the grid of size 3000 × 3000

Computing resources	T_B	T_M	S_M	T_{1D}	S_{1D}	T_{2D}	S_{2D}	S^f_{1D}	S^f_{2D}
1 × KNL	3869	2540	1.52	2198	1.16	2078	1.22	1.76	1.86
1 × SKL	4061	2359	1.72	1908	1.24	1800	1.31	2.13	2.26
2 × SKL	3266	2005	1.63	1466	1.37	1297	1.55	2.23	2.52

Besides the execution time, both tables present also the following speedups:

- S_M=T_B/T_M—speedup achieved after the first step of the proposed method of adaptation;
- S_{1D}=T_M/T_{1D}—speedup obtained by the usage of the second step of the proposed method with the 1D map;
- S_{2D}=T_M/T_{2D}—speedup achieved by the usage of the second step of the proposed method with the 2D map;
- $S^f_{1D} = T_B/T_{1D}$—final speedup obtained as a result of using the proposed method with the 1D map;
- $S^f_{2D} = T_B/T_{2D}$—final speedup achieved as a result of using the proposed method with the 2D map.

The analysis of Tables 2 and 3 allows us to conclude that the usage of only the first step of the proposed adaptation method permits increasing the performance of simulation for all configurations of computing resources. For the smaller grid (2000 × 2000), the highest speedup of $S_M = 1.78$ is achieved for the configuration with a single CPU, while the lowest one equal to 1.54 times is obtained on the KNL processor. An analogous conclusion is also correct for the second grid (3000 × 3000).

Based on the total execution times T_{1D} and T_{2D} achieved for the versions with, respectively, 1D and 2D maps, it can be concluded that the 2D map gives a better performance than its 1D counterpart. For the smaller of grids, approximating the domain of simulation with the 1D map allows us to accelerate the application executed on the KNL processor by 1.17 times against the code obtained after the first step of the adaptation method, and 1.27 and 1.39 times for respectively one and two CPUs. For the 2D map, speedups of respectively 1.24, 1.35, and 1.62 times are achieved. Thus, the performance advantage of the 2D map is particularly visible for the configuration with two CPUs. For the greater of grids, the performance gain achieved by introducing the workload prediction step is slightly lower.

The last column of Tables 2 and 3 shows the speedup S^f_{2D} obtained by the usage of both steps of the proposed method of adaptation in comparison with the basic version of the application. The presented values of speedup mean that the proposed method reduces the execution time of simulation significantly. For the grid of size 2000 × 2000, the highest performance gain is achieved for two Intel Xeon CPUs, where the new code yields the speedup of about 2.74 times against the basic version. For configurations with a single KNL processor and single CPU, the developed implementation enables accelerating the simulation by about 1.9 and 2.4 times, respectively. In the case of the grid of size 3000 × 3000, the total performance gain is slightly lower.

Table 4 presents the comparison of the time spent on primary computations, time of checking the selection criterion, and time spent on building the map, for the version obtained after the first step of the adaptation method and versions with 1D and 2D maps. These results are shown for the grid of size 2000 × 2000 and two configurations of computing resources: (i) single KNL processor, and (ii) two Intel Xeon CPUs. For the KNL processor, the time spent on the primary (actual) computations practically does not depends on the version of the application. In consequence, the speedup achieved against the modified version is practically the result of applying the prediction algorithm. In the case of the modified version, checking the selection criterion takes 354 seconds. The introduction of the second step of the proposed method allows decreasing this time by about 162 seconds and 220 seconds for 1D and 2D maps, respectively. Taking into account a small overhead for building the map (7 seconds for KNL), the total time related to the selection criterion is 199 and 141 seconds for the versions with 1D and 2D maps. As a result, the selection criterion is executed 1.77 times and 2.5 times faster in comparison with the modified version of the code.

TABLE 4 Comparison of time spent on primary (actual) computations, time of checking the selection criterion, and time spent on building the map, for the grid of size 2000 × 2000

Version of application	Time spent on primary computations	Selection criterion			
		Time of checking T_C	Time of building map T_{MAP}	Full time $T_F = T_C + T_{MAP}$	Speedup $S = T_C / T_F$
1 × KNL: Listing 5	728	354	—	354	—
1 × KNL: 1D map	729	192	7	199	1.77
1 × KNL: 2D map	734	134	7	141	2.51
2 × SKL: Listing 5	401	438	—	438	—
2 × SKL: 1D map	375	247	2	249	1.76
2 × SKL: 2D map	359	172	2	174	2.52

For the configuration with two Intel Xeon CPUs, the contribution of checking the selection criterion to the overall execution time is considerably higher than for the KNL processor. It is especially visible for the version shown in Listing 5, when this checking takes 37 seconds longer than the time spent on actual computations. The usage of the workload prediction algorithm allows decreasing the time of checking the selection criterion from 438 to 247 seconds and 172 seconds for the 1D and 2D maps, respectively. In consequence, the speedup achieved for checking the selection criterion is practically the same as in the case of the KNL processor. An interesting effect observed for the configuration with two CPUs is the reduction of the time spent on actual computations with the introduction of the workload prediction. The usage of 1D and 2D maps increases the performance of executing the primary computations by about 6% and 10%, respectively. Most probably, the reason for this is a better utilization of the cache hierarchy of CPUs.

Finally, Figure 7 presents the aggregated time of checking the selection criterion for successive packages of *R* time steps each, achieved for: (a) Intel Xeon Phi 7250F accelerator, and (b) two Intel Xeon Platinum 8180 processors. The presented charts clearly confirm the effectiveness of the proposed workload prediction algorithm, as well as better efficiency of the 2D map as compared with the 1D version of the map.

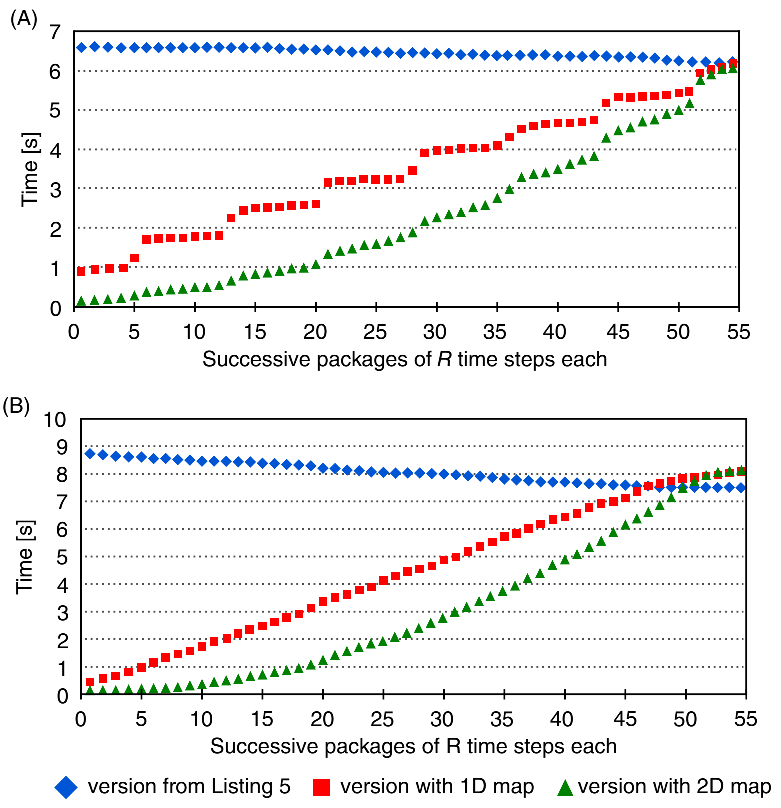

FIGURE 7 Aggregated time of checking the selection criterion for successive packages of *R* time steps each, achieved for, A, single Intel Xeon Phi 7250F processor and, B, configuration with two Intel Xeon Platinum 8180 CPUs

7 | CONCLUSIONS AND FUTURE WORKS

The main challenge of this work is the performance optimization of the solidification application with the dynamic computational intensity when calculations are performed using a carefully selected group of nodes. For this aim, a two-step method is proposed to increase the application performance for multi-/manycore architectures. In the first step, the *loop fusion* technique is used to permit executing all kernels in a single nested loop, as well as reducing the number of conditional operators performed within a single time step. These modifications are vital to implementing the second step, which includes an algorithm for the dynamic workload prediction and load balancing across resources of the computing platform, in successive time steps.

Two versions of the algorithm are developed—with 1D and 2D computational maps used for predicting the computational domain within the grid. They allow reducing superfluous operations by adjusting the computational domain to the domain of simulation (or area of grain growth), as close as possible. These algorithmic solutions ensure a more efficient workload distribution across available cores since each core will perform primary computations within the domain of simulation in successive time steps.

For parallelizing the code with the 1D map, **omp parallel for** directive is used for the outer loop, which iterates over nodes of the grid. Ensuring efficient load balancing across cores is based on the fact that this directive now embraces not the entire grid, but only an interval of nodes—from `minNode` to `maxNode`. A vital role in adapting the distribution of workload across cores to properties of a particular computing platform plays the usage of a suitable scheduling clause to control the policy of assigning loop iterations to cores. It is evaluated experimentally that the dynamic scheduling option with the size of chunks equal to the size of rows of the grid is the most efficient for all platforms studied in this article.

In the case of the 2D map, load balancing is based on the fact that **omp parallel for** directive iterates over a subset of row indices in the range from `minRow` to `maxRow`. Each row includes only the grid nodes with column indices in the interval from `minCol` to `maxCol`. Similar to the 1D version, an essential role in optimizing the overall performance plays selecting a proper scheduling clause. Again for all studied platforms, the dynamic scheduling option yields the best performance, but the size of chunks is now equal to 1. This size corresponds to a single row of the predicted 2D domain.

The achieved performance results show that the proposed optimization method allows increasing the application performance significantly for all tested configurations of computing resources, with a clear advantage of 2D map over 1D map. The highest performance gain is achieved for two Intel Xeon Platinum 8180 CPUs, where the new code based on 2D map yields the speedups of 2.74 and 2.52 times against the basic version, respectively for grids of size 2000×2000 and 3000×3000. At the same time, the usage of the proposed method with the 2D map for a single Intel 7250F KNL accelerator permits us to reduce the execution time about 1.91 and 1.86 times, respectively.

This work provides the basis for further development and optimization of the solidification modeling application with the dynamic intensity of computations. The primary direction of our future work is an extension of the proposed approach over the GPU accelerators of different vendors using the OpenCL framework.[35] Also, we are planning to explore CPU[36] and GPU[37] frequency scaling as a tool to optimize the energy efficiency of the application.

ACKNOWLEDGEMENTS

This work is supported by the National Science Centre, Poland under grants no. UMO-2017/26/D/ST6/00687 by the project financed within the program of the Polish Minister of Science and Higher Education under the name “Regional Initiative of Excellence” in the years 2019-2022 (project no. 020/RID/2018/19, the amount of financing 12 000 000 PLN).

ORCID

Kamil Halbiniak https://orcid.org/0000-0001-9116-8981
Marco Lapegna https://orcid.org/0000-0001-9953-1319

REFERENCES

1. Steinbach I. Phase-field models in materials science. *Modelling and Simulation in Materials Science and Engineering*. 2009;17(7):073001. http://dx.doi.org/10.1088/0965-0393/17/7/073001.
2. Provatas N, Elder K. *Phase-Field Methods in Materials Science and Engineering*. Weinheim, Germany: Wiley; 2010.
3. Folch R, Casademunt J, Hernandez-Machado A, Ramirez-Piscina L. Phase-field model for Hele-Shaw flows with arbitrary viscosity contrast II. numerical study. *Phys Rev E*. 1999;60(2):1734-1740.
4. Karma A, Kessler D, Levine H. Phase-Field Model of Mode III Dynamic Fracture. *Physical Review Letters*. 2001;87(4). http://dx.doi.org/10.1103/physrevlett.87.045501.
5. Halbiniak K, Wyrzykowski R, Szustak L, Olas T. Assessment of offload-based programming environments for hybrid CPU-MIC platforms in numerical modeling of solidification. *Simulat Modell Pract Theory*. 2018;87:48-72.
6. Hager G. *Wellein G*. Introduction to High Performance Computing for Science and Engineers: CRC Press; 2011.
7. Shimokawabe T, Aoki T, Takaki T, et al. Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. Paper presented at: Proceedings of the 2011 ACM/IEEE International Conference High Performance Computing, Networking, Storage and Analysis – SC’11; 2011.
8. Hötzer J, Jainta M, Steinmetz P, et al. Large scale phase-field simulations of directional ternary eutectic solidification. *Acta Materialia*. 2015;93:194-204.

9. Bauer M, Hötzer J, Jainta M, et al. Massively parallel phase-field simulations for ternary eutectic directional solidification. Paper presented at: Proceedings of the 2015 International Conference High Performance Computing, Networking, Storage and Analysis – SC'15; 2015.
10. Bauer M, Hötzer J, Ernst D, et al. Code generation for massively parallel phase-field simulations. Paper presented at: Proceedings of the 2019 International Conference High Performance Computing, Networking, Storage and Analysis – SC'19; 2019.
11. Adrian H, Spiradek-Hahn K. The simulation of dendritic growth in Ni-Cu alloy using the phase field model. *Archiv Mater Sci Eng*. 2009;40(2):89-93.
12. Choudhury A, Reuther K, Wesner E, August A, Nestler B, Rettenmayr M. Comparison of phase-field and cellular automaton models for dendritic solidification in Al-Cu alloy. *Comput Mater Sci*. 2012;55:263-268.
13. Zaeem M, Yin H, Felicelli S. Modeling dendritic solidification of Al-3%Cu using cellular automaton and phase-field methods. *Appl Math Modell*. 2013;37(5):3495-3503.
14. Trobec R, Korosec G. *Parallel Scientific Computing: Theory, Algorithms, and Applications of Mesh Based and Meshless Methods*. Cham, Switzerland: Springer; 2015.
15. Szustak L, Halbiniak K, Kulawik A, Wrobel J, Gepner P. Toward parallel modeling of solidification based on the generalized finite difference method using Intel Xeon Phi. 9573 of Proceedings of the 11th International Conference Parallel Processing and Applied Mathematics – PPAM; 2015, Lect. Notes in Computer Science:411-412.
16. Halbiniak K, Szustak L, Lastovetsky A, Wyrzykowski R. Exploring OpenMP accelerator model in a real-life scientific application using hybrid CPU-MIC platforms. Paper presented at: Proceedings of the 3rd International Workshop on Sustainable Ultrascale Computing Systems, NESUS 2016; 2016:11-14.
17. Szustak L, Halbiniak K, Kulawik A, Wyrzykowski R, Uminski P, Sasinowski M. Using hstreams programming library for accelerating a real-life application on intel MIC. 10049 of Proceedings of the International Conference Algorithms and Architectures for Parallel Processing - ICA3PP Lecture Notes in Computer Science; 2016:373-382.
18. Szustak L, Halbiniak K, Kuczynski L, Wrobel J, Kulawik A. Porting and optimization of solidification application for CPU-MIC hybrid platforms. *Int J High Perform Comput Appl*. 2018;32(4):523-539.
19. Laccetti G, Lapegna M, Mele V, Romano D. A high performance modified k-means algorithm for dynamic data clustering in multi-core CPUs based environments. 11874 of International Conference Internet and Distributed Computing Systems -IDCS 2019:89-99.
20. Devine K, Boman E, Heaphy R, et al. New challenges in dynamic load balancing. *Appl Numer Math*. 2005;52:133-152.
21. Lastovetsky A, Dongarra J. *High Performance Heterogeneous Computing*. New York, United States: Wiley; 2009.
22. Lastovetsky A, Reddy R. Data partitioning with a functional performance model of heterogeneous processors. *Int J High Perform Comput Appl*. 2007;21(1):76-90.
23. Lastovetsky A, Szustak L, Wyrzykowski R. Model-based optimization of EULAG kernel on intel Xeon Phi through load imbalancing. *IEEE Trans Parall Distrib Syst*. 2017;28(3):787-797.
24. Szustak L, Bratek P. Performance portable parallel programming of heterogeneous stencils across shared-memory platforms with modern intel processors. *Int J High Perform Comput Appl*. 2019;33(3):534-553.
25. Devine K, Boman E, Heaphy R, Hendrickson B, Vaughan C. Zoltan data management services for parallel dynamic applications. *Comput Sci Eng*. 2002;2:90-96.
26. G. Karypis, Schloegel K. Parallel graph partitioning and sparse matrix ordering library version 4.0.http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/manual.pdf. Accessed March 30, 2013.
27. OpenMP Application Programming Interface, version 5.0; 2018.https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.
28. Halbiniak K, Szustak L, Kulawik A, Gepner P. Performance optimizations for parallel modeling of solidification with dynamic intensity of computations. Paper presented at: Proceedings of the 13th International Conference Parallel Processing and Applied Mathematics – PPAM; 2019. 2020:370-384.
29. Takaki T. Phase-field modeling and simulations of dendrite growth. *ISIJ Int*. 2014;54(2):437-444.
30. Warren J, Boettinger W. Prediction of dendritic growth and microsegregation patterns in a binary alloy using the phase-field method. *Acta Metallurgica et Materialia*. 1995;43(2):689-703.
31. Benito J, Uren F, Gavete L. The generalized finite difference method. In: Àlvarez MP, ed. *Leading-Edge Applied Mathematical Modeling Research*. New York, United States: Nova Science Publishers; 2008:251-293.
32. Kulawik A. *The Modeling of the Phenomena of the Heat Treatment of the Medium Carbon Steel. Monographs*. Vol 281. Czestochowa: Wydawnictwo Politechnki Czestochowskiej; 2013 (in Polish).
33. Cardoso J, Coutinho J, Diniz P. Source code transformations and optimizations. *Embedded Computing for High Performance: Design Exploration and Customization Using High-level Compilation and Synthesis Tools*. San Francisco, United States: Elsevier; 2017:137-183.
34. Jeffers J, Reinders J, Sodani A. *Intel Xeon Phi Processor High Performance Programming*: Knights Landing Edition. Elsevier. 2016.
35. Halbiniak J, Szustak L, Olas T, Wyrzykowski R, Gepner P. Exploration of OpenCL heterogeneous programming for porting solidification modeling. *Concurr Comput Pract Exp*. 2020; under reviewing.
36. Haj-Yahya J, Mendelson A, Ben Asher Y, Chattopadhyay A. *Energy Efficient High Performance Processors Recent Approaches for Designing Green High Performance Computing*. New York, NY, Springer: Springer; 2018.
37. Tang Z, Wang Y, Wang Q, Chu X. The impact of GPU DVFS on the energy and performance of deep learning: an empirical study; 2019. arXiv:1905.11012.

How to cite this article: Halbiniak K, Olas T, Szustak L, Kulawik A, Lapegna M. Dynamic workload prediction and distribution in numerical modeling of solidification on multi-/manycore architectures. *Concurrency Computat Pract Exper*. 2021;33:e5905. https://doi.org/10.1002/cpe.5905