UPPSALA
UNIVERSITET

Implementation and Performance Studies of a Three-phase Model Solver

Alexandra Redlund, Gong Cheng

Project in Computational Science: Report

PROJECT REPORT

Institutionen för informationsteknologi

Abstract

This project considers the implementation of a highly parallel simulation of a three-phase problem. The model used is based on the Cahn-Hilliard equation, where the phases are controlled by the so-called concentration.

Our implementation is written in C++ based on a finite element package called deal.ii and a parallel framework Trilinos. We show that such an implementation can exhibit good parallel scaling by running it on the UPPMAX computer system on up to 1024 cores.

Comparing the performance of our implementation to that of a previous two-phase implementation, we observe that adding more phases does not impact performance.

Contents

1 Introduction

Multi-phase flows are physical systems where more than one fluid interact. These systems are very important to study since they are central in many technical applications. In this project we have studied three-phase flows.

Examples of such systems are three-phase flows in porous media where the phases interact and interfaces between them change as the liquids flow through the media cavities. Another example is production where water, oil and air are often mixed.

As a three-phase flow system evolves over time the existing interfaces between liquids can change at any time. When two bodies of different liquids separate to allow the third liquid to fill the gap, new interfaces between the two bodies and the third liquid are created. When two bodies of the same liquid merge, the interfaces to other liquids vanish. Any simulation of such systems needs to take these interfaces into account. However, because of the potentially large number of interfaces and that they can appear and disappear at any time, it becomes complicated to track them. In this project we have used a diffuse-interface phase-field model to implicitly track the interfaces between phases. Here the interfaces need not be known beforehand since they are not explicitly tracked, instead they are represented with a mathematical model that indicates where the interfaces are and that varies in space and time. Using this technique the effective thickness of the interfaces depend on the spatial resolution of the simulation. Thin interfaces are required to accurately simulate real physical systems. This forces us to use numerical methods of high resolution and accuracy.

The goal of this project has been to allow for the solution of such highly resolved systems by utilizing parallel computers. Using a previous two-phase C++ implementation and a three-phase MATLAB implementation we set out to create a parallel three-phase C++ implementation for high performance clusters. During the project we have had access to the UPPMAX computer system [1] for test runs of our program.

This report is structured as follows. First, in Section 2, the three-phase model is described and then in Section 3 the numerical solution method being used is explained. Section 4 shows the results of our numerical experiments and in Section 5 a discussion about the results is presented. Conclusions are found in Section 6.

2 Three-phase model

The model we use to describe the three-phase system represents each phase by a scalar variable, referred to as a concentration $c_i(\mathbf{x}, t)$. These concentrations are modeled such that they attain a value of 1 inside their own phase i and a value of 0 inside other phases. Thus, the phase at a given point in space and time can be derived from the values of the concentrations at that point. Over the interface between phases these concentrations vary rapidly but smoothly, from one value to another, creating a diffuse interface between the phases. These concentrations can be seen as mole fractions and obey the relations

$$\sum_{i=1}^{n} c_i(\mathbf{x}, t) = 1 \tag{1}$$

and

$$0 \leq c_i(\mathbf{x}, t) \leq 1.$$

It follows from (1) that the concentration of a given phase can be calculated from the other concentrations according to $c_n = 1 - \sum_{i=1}^{n} c_i$. Thus, in our case of three phases, we only need to calculate two concentrations since the third can be calculated from the first two. This simplifies the numerical simulations but what is more interesting with this model is that the interfaces are implicitly given. This eases numerical simulation significantly. First of all, we need not know the interfaces initially. We solve for the concentration from whatever our initial data is and can derive the position of the interfaces from this at each point in space and time. In this way, the complexity of the simulation is not dependent on the current state of the simulation as it is when explicitly tracking interfaces, interactions between them and how they are created, deformed, merged and destroyed.

The set of equations we have used that describe this process are called the Cahn-Hilliard equations. They are two coupled fourth order, time dependent partial differential equations one for each of the phases that model the process of phase separation and spontaneous forming of domains that are pure in each component. To write these down we first define another basic physical quantity, the so called chemical potential $\eta_i(\mathbf{x}, t)$. The chemical potential is a form of potential energy that changes during phase transitions and is released and absorbed during chemical reactions. Mathematically it is the partial derivative of the free energy with respect to the amount of a chemical substance. Using this, the Cahn-Hilliard equation can be written as

$$\frac{\partial \mathbf{c}}{\partial t} = \nabla \cdot \left(L\left(\mathbf{c}\right) \nabla \eta \right) \tag{2}$$

$$\left(L \nabla \eta\right)_i \cdot \mathbf{n} = 0. \tag{3}$$

Here the mobility matrix $L(\mathbf{x})$ is symmetric and positive semidefinite. To make sure mass is conserved and that (1) is satisfied L has to satisfy

$$L(\mathbf{c})\mathbf{e} = \mathbf{0}.$$

Using the chemical potential, equation (2) can be rewritten as a system of two coupled second order equations, see [6] for more information. The resulting version of the Cahn-Hilliard equations is

$$\eta_i - F'(c_i) + \frac{3\varepsilon}{4}\Delta c_i = 0, \tag{4}$$

$$L\Delta\eta_i - \frac{\partial c_i}{\partial t} - (\mathbf{u}\cdot\nabla)\, c_i = 0, \tag{5}$$

where η_i is the chemical potential, $F(c_i)$ is the Jacobian (nonlinear term), ε is the interface width, c_i is the concentration, L is the mobility matrix and $\mathbf{u}$ is the velocity vector. This is a nonlinear, coupled system of partial differential equations. The nonlinearity lies in $F(c_i)$ which is defined as

$$F_i(c) = \frac{4v_T}{v_i}\sum_{j\neq i}\left[\frac{1}{v_j}\left(\frac{\partial f_0}{\partial c_i} - \frac{\partial f_0}{\partial c_j}\right)\right],\ i = 1, 2\ , \tag{6}$$

where $f_0 = f_0^{(1)} + f_0^{(2)}$ is given by

$$f_0^{(1)}(\mathbf{c}) = \sigma_{12}c_1^2c_2^2 + \sigma_{13}c_1^2c_3^2 + \sigma_{23}c_2^2c_3^2 + c_1c_2c_3\,(v_1c_1 + v_2c_2 + v_3c_3)$$
$$f_0^{(2)}(\mathbf{c}) = 3\lambda v_1v_2v_3.$$

These are not only affected by the concentration but also the surface tensions between the phases, given by σ_{12}, σ_{13} and σ_{23}. We note that here a certain polynomial form of the function F is used , as described in [7].

We have looked at two different setups that give different characteristics to the solution. These setups are called partial and total spreading and are controlled by the chosen surface tensions, see [6]. Thus, the choice of initial condition affects $F(c_i)$.

3 Method

The Cahn-Hilliard equation in (4) and (5) is a coupled system of two second order partial differential equations [8] for both the concentration c and the chemical potential η. There is no analytical solution for these time dependent nonlinear system of partial differential equations. As a result, a numerical method is required to solve this problem.

3.1 Discreted Formulation

The system is discretized by the finite element method (FEM) in space with piecewise linear basis functions for the two unknowns c and η and a uniform square mesh. The backward Euler method is implemented in time domain for the stability of the numerical method, although it introduces a solution of a linear system for each time step.

The fully discretized system at time t_k is in the form

$$\begin{aligned} \mathcal{M}\eta^k - \mathcal{F}(c^k) - \gamma\mathcal{K}c^k &= 0 \\ \Delta t_k \omega \mathcal{K}\eta^k + \mathcal{M}c^k + \Delta t_k \mathcal{W}c^k - \mathcal{M}c^{k-1} &= 0 \ , \end{aligned} \tag{7}$$

where Δt_k is the time step that $t_{k+1} = t_k + \Delta t_k$, γ and ω are the constants governed by Peclet and Cahn number. The unknown variables are $c^k = [(c_1^k)^T, (c_2^k)^T]^T$ and $\eta^k = [(\eta_1^k)^T, (\eta_2^k)^T]^T$ where $c_j^k = \{c_{j,i}^k\}_{i=1}^N$ and $\eta_j^k = \{\eta_{j,i}^k\}_{i=1}^N$ are the corresponding unknown FEM vectors and they can be also written as the linear combinations of FEM basis functions $\varphi_i(x)$, $i = 1, 2, ..., N$ that $c_j(x,t) = \sum_{i=1}^N c_{j,i}^k \varphi_i(x)$, $\eta_j = \sum_{i=1}^N \eta_{j,i}^k \varphi_i(x)$, $j = 1, 2$. The index j represents the two phases to be solved. As a result, the matrices

$$\mathcal{M} = \begin{bmatrix} M & \\ & M \end{bmatrix}, \ \mathcal{K} = \begin{bmatrix} K & \\ & K \end{bmatrix}, \ \mathcal{W} = \begin{bmatrix} W & \\ & W \end{bmatrix}, \tag{8}$$

refer to block diagonal matrices with mass matrix M, stiffness matrix K and convection matrix W in FEM, respectively, reflecting the fact that we handle two phases.

3.2 Nonlinear term and Quasi-Newton method

The nonlinear term $\mathcal{F}(c^k) = (\mathcal{F}_1^T(c^k), \mathcal{F}_2^T(c^k))^T$ is a vector with elements

$$\mathcal{F}_{i,m}(c^k) = \int_\Omega F_i \left(\sum_{n=1}^N c_{1,n}^k \varphi_n(x), \ \sum_{n=1}^N c_{2,n}^k \varphi_n(x) \right) \varphi_m(x)\, dx, \ i = 1, 2 \ , \tag{9}$$

where $\varphi_m(x)$, $m = 1, ..., N$ are the FEM basis functions and $c^k = [c_1^k, c_2^k]^T$ are the unknown vectors of concentrations. $F_i(c)$ denotes the nonlinear term in (6).

In each time step, this nonlinear system is solved by a Quasi-Newton method [6] which simplifies the computation by using an approximate Jacobian matrix in the Newton iteration.

We denote $x^k = [c^k, \eta^k]^T$ and system (7) as $f^k(x^k) = 0$ for the time step at t_k. Considering the Newton's method for solving the nonlinear system, at

the sth iteration

$$x^{k,s+1} = x^{k,s} - \Delta x^{k,s},\ s = 1, 2, \ldots , \tag{10}$$

until the sequence converges. In another word, the nonlinear iteration will stop until $\Delta x^{k,s}$ is closed enough to 0. The value of $\Delta x^{k,s}$ is governed by the system

$$\mathcal{A}^{k,s}\Delta x^{k,s} = f^k(x^{k,s}) \ , \tag{11}$$

where $\mathcal{A}^{k,s}$ is the exact Jacobian of the system in (7) at time t_k. The Jacobian can be represented in matrix form that

$$\mathcal{A}^{k,s} = \begin{bmatrix} \mathcal{M} & -\mathcal{J}(\mathcal{F}(c^{k,s})) - \gamma\mathcal{K} \\ \Delta t_k\omega\mathcal{K} & \mathcal{M} + \Delta t_k\mathcal{W} \end{bmatrix} \ , \tag{12}$$

where $\mathcal{J}(\mathcal{F}(c))$ is the Jacobian matrix of function $\mathcal{F}(c)$. Thus, the linear system (11) is solved for each Newton iteration and the solution is updated by (10) until the difference $\Delta x^{k,s}$ is negligible.

However, the rate of convergence of Newton's Method could be rather slow and the computation is quite heavy since the system matrix of Newton method in (12) is recomputed for every iteration. In order to simplify the nonlinear system, a quasi-Newton method is implemented with a constant Jacobian for the nonlinear iteration in (11) such that the linear system is simplified (see [6]) as to solve

$$\mathcal{A}_0^k\Delta x^{k,s} = f(x^{k,s}) \ , \tag{13}$$

where $\mathcal{A}_0^k$ is an approximation of $\mathcal{A}^{k,s}$ obatained by neglecting the nonlinear term and convection term, namely,

$$\mathcal{A}_0^k = \begin{bmatrix} \mathcal{M} & -\gamma\mathcal{K} \\ \Delta t_k\omega\mathcal{K} & \mathcal{M} \end{bmatrix}. \tag{14}$$

The matrix $\mathcal{A}_0^k$ remains constant as long as the time step does not change. In this case, the matrix can be denoted as

$$\mathcal{A}_0 = \begin{bmatrix} \mathcal{M} & -\gamma\mathcal{K} \\ \Delta t\omega\mathcal{K} & \mathcal{M} \end{bmatrix}, \tag{15}$$

since a fixed time-step backward Euler method is implemented.

Consequently, the computation is to solve a system of constant matrix but with different right hand side $f(x^{k,s})$ which is evaluated by (7) for each nonlinear iteration.

3.3 Permutation of the block matrix

Considering the block structure (8) of the mass matrix and stiffness matrix in FEM, matrix $\mathcal{A}_0$ in (15) can be expanded as (16)

$$\mathcal{A}_0 = \begin{bmatrix} M & & -\gamma K & \\ & M & & -\gamma K \\ \Delta t\omega K & & M & \\ & \Delta t\omega K & & M \end{bmatrix} . \tag{16}$$

After performing a block permutation with a matrix P of the form

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} , \tag{17}$$

we obtain

$$P^T \mathcal{A}_0 P = \begin{bmatrix} A_0 & \\ & A_0 \end{bmatrix} = \begin{bmatrix} M & -\gamma K & & \\ \Delta t\omega K & M & & \\ & & M & -\gamma K \\ & & \Delta t\omega K & M \end{bmatrix} \tag{18}$$

and we see that the matrix (18) is block-diagonal and at time t_k for the sth nonlinear iteration the system in (13) can be divided into two independent sub-system with same matrix A_0 but different right hand sides $b_1^{k,s}$ and $b_2^{k,s}$ so that

$$A_0 \begin{bmatrix} \eta_i^{k,s} \\ c_i^{k,s} \end{bmatrix} = b_i^{k,s}, \ i = 1, 2 , \tag{19}$$

where

$$A_0 = \begin{bmatrix} M & -\gamma K \\ \Delta t\omega K & M \end{bmatrix} \tag{20}$$

and

$$b_i^{k,s} = \begin{bmatrix} M\eta_i^{k,s} - F_i(c_i^{k,s}) - \gamma K c_i^{k,s} \\ \Delta tk\omega K \eta_i^{k,s} + M c_i^{k,s} - M c_i^{k-1,0} \end{bmatrix} , \tag{21}$$

where $c_i^{k,s}$ and $\eta_i^{k,s}$ are the current value of the unknowns in the process of Quasi-Newton iterations, $c_i^{k-1,0}$ is the numerical solutions for the previous time step, and $F_i(c_i^{k,s})$ is the function described in (6).

3.4 Preconditioning

So far, by the above described algebraic simplifications, the size of system to be solved is decreased to a half of the original one. However, to achieve

a high accuracy without missing the interfaces, the space discretization in finite element method should be rather fine that the degrees of freedom are always too large and the linear system in (19) is hard to be solved. Therefore, an efficient solving algorithm is required for the linear system.

In [4], it is shown that the matrix A_0 can be further modified in order to simplify the solution process. Namely, the matrix A_0 is approximated by $\hat{A}_0$, where

$$\hat{A}_0 = \begin{bmatrix} M & -\gamma K \\ \Delta t\omega K & M + 2\sqrt{\beta}K \end{bmatrix} , \tag{22}$$

and $\beta = \Delta t\omega\gamma$. In turn, $\hat{A}_0$ can be factorized exactly as

$$\hat{A}_0 = \begin{bmatrix} M & 0 \\ \Delta t\omega K & M + \sqrt{\beta}K \end{bmatrix} \begin{bmatrix} I & -\gamma M^{-1}K \\ 0 & M^{-1}(M + \sqrt{\beta}K) \end{bmatrix} . \tag{23}$$

Computation shows, see also in [4], that the solution of the system

$$\hat{A}_0 \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \tag{24}$$

can be obtained by only solving system with $M + \sqrt{\beta}K$ twice and some additional vector operations. To see this, we consider the solutions that

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \hat{A}_0^{-1} \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} \frac{\gamma}{\sqrt{\beta}}(g_1 + g_2) \\ -g_2 \end{bmatrix} , \tag{25}$$

where

$$g_1 = (M + \sqrt{\beta}K)^{-1}(\frac{\sqrt{\beta}}{\gamma}f_1 + f_2) , \tag{26}$$

$$g_2 = (M + \sqrt{\beta}K)^{-1}(\frac{\sqrt{\beta}}{\gamma}f_1 - Mg_1) . \tag{27}$$

To summarize, the original nonlinear system is discretized in time by backward Euler method and formulated to a series of linear systems by Quasi-Newton method in (12) and (13). In one nonlinear iteration, the linear system s(13) with a 4×4 block size matrix is simplified as solving a one block size matrix twice with some matrix vector multiplications and vector updates. The matrix $M + \sqrt{\beta}K$ is symmetric and positive definite, it can be solved by the Conjugate Gradient Method (CG) and an algebraic multigrid method (AMG) as a preconditioner. As a result, the FEM problem in (7) is reduced to a quarter of the original size and the rates of convergence for both nonlinear and linear iterations are accelerated by optimized numerical methods.

3.5 Solution algorithm

Summing up all above, the solution algorithm can be described as three levels of nested loops as in Algorithm 1. The time loop in the outermost level is responsible for updating solutions for each time step. In the middle level, there is a nonlinear loop for the Quasi-Newton method. In each nonlinear iteration, two sub-systems with different right hand sides are solved by CG method with AMG preconditioner to accelerate the convergence.

Algorithm 1 The solution procedure

```
Initialize the grid
Assembly mass matrix M and stiffness matrix K
Set the initial values for c and η
for each time step do
    while nonlinear solution not converge do
        Assembly matrix to compute and update right hand side vectors
        for i = 1 to 2 do
            Solve g_1 and g_2 by CG method with AMG preconditioner
            Update c_i and η_i
        end for
    end while
    Update linear solutions of c and η
end for
```

In this way the task to solve system with $\mathcal{A}_0$ is reduced to a quarter of the original size which significantly improves the performance in both serial and parallel computation, especially for very large problem size.

3.6 Parallel Implementation

As Algorithm 1 shows, the main computation tasks in this solver are assembly of matrices, matrix-vector multiplications, vector operations and the preconditioned iteration method for the linear systems with $M + \sqrt{\beta}K$. The algorithm is implemented using deal.II(see [2]) which is an open source library for finite element applications in C++. It is used to set up all the required variables in FEM, such as the mesh, degree of freedom, the matrices and vectors. As the sparsity of mass matrix and stiffness matrix, the complexity of matrix assembly is $O(N)$ (or linear) to the problem size, and the memory cost is also linear.

Most of the deal.II matrices and vectors are wrapped by a framework for advanced parallel algorithm called Trilinos([3]), which takes care of data partitioning, load balance as well as for the CG iteration solver and the preconditioner. The framework based on Message Passing Interface (MPI)

Size	66564	264196	1052676	4202500	16793604	67141636	268500996
Partial Sp.	21/2	22/3	22/4	23/4	23/5	24/5	25/7
Total Sp.	41/3	35/3	32/4	30/4	28/5	27/5	27/5

Table 1: Number of nonlinear and inner CG iterations for one time step. ('Sp.' stands for spreading.)

provides complexity of $O(N)$ for distributing data into multi-cores as well as computing matrix-vector multiplication for sparse matrix[5].

The only solution procedure in the algorithm is for the linear system (26) and (27) by preconditioned CG solver. This method gives an almost constant number of iterations for CG method[4, 6] such that the complexity is almost linear. As a result, the total complexity of the algorithm is expected to be almost linear to the size of problem for both serial and parallel implementation.

4 Results

The numerical experiments are done on a UPPMAX cluster which has 160 nodes and each node has two 8-core Opteron 6220 processors running at 3 GHz. The test cases vary from 66564 to 268500996 degree of freedom in up to 1024 cores for both partial and total spreading condition.

4.1 Number of iterations

The number of iterations is not changed for the same problem with different number of cores in parallel. Table 1 illustrates the number of iterations for Quasi-Newton Method (left) with stopping criteria 10^{-6} and CG Method with AMG preconditioner (right) at a certain time step that $\Delta t = \frac{h}{10}$ where h is the space discretization parameter.

The partial spreading case turns out to be a simpler problem since the shape of the droplet and interfaces does not deform quite much. The whole domain does not contain any sharp angle and extreme values so that the nonlinear system is rather 'smooth' to be solved that the nonlinear iterations stay between 20 and 25 for all these cases. However, in total spreading, it takes twice as many iterations as for the partial spreading on the coarsest mesh. The iteration counts decrease as the mesh are finer, since the approximation of Jacobians becomes more accurate. Meanwhile, the iterations of the inner CG solver are all below 5 which leads to a nearly constant number of average iterations per time step as expected. As a result, the computation does not become much heavier when the mesh is refined.

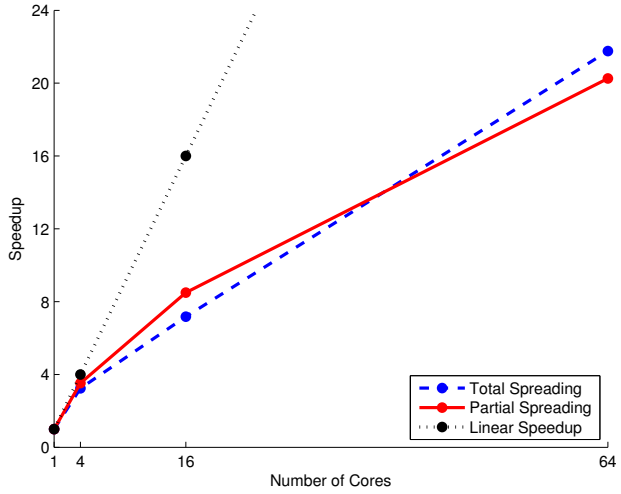

Figure 1: The speedup for problem size of 4202500.

4.2 Speedup

The speedup is defined as $S_p = T_1/T_p$, where p is the number of processors, T_1 is the time of sequential execution on one processor and T_p is the parallel execution with p processors. According to some pre-tests, the problem size is chosen as 4202500 which gives an acceptable sequential execution time (less than 3 hours) and the mesh size is not too large so that it resolves the interface well enough. Figure 1 illustrates the speedup of partial and total spreading cases by using up to 64 cores.

These two test cases show nearly the same speedup although the number of iterations and time cost in computation of partial spreading case is around 75% of those in total spreading. They achieved ideal speedup when expanded from 1 to 4 cores. However, in 16 cores, the speedup are only half of the ideal value since the communication becomes heavier than the computation when more cores are involved in this problem size. The speedup becomes even worse when the program is run on 64 cores and decreases to 1/3 of the ideal value.

The decrease of performance is due to the overhead of communication between cores in CG iteration solver and matrix vector multiplication which can not be totally overlapped by the computation.

Size	66564	264196	1052676	4202500	16793604
Number of Cores	1	4	16	64	256
Partial Spreading					
Average wall time (s)	1.67	2.39	4.64	9.09	10.99
Factor of increase		1.43	1.94	1.96	1.21
Total Spreading					
Average wall time (s)	3.18	3.65	6.97	10.43	13.03
Factor of increase		1.15	1.91	1.50	1.25

Table 2: Weak scalability tests (set 1), average wall time in seconds and factor of increase of time after one refinement of the mesh.

Size	264196	1052676	4202500	16793604	67141636
Number of Cores	1	4	16	64	256
Partial Spreading					
Average wall time (s)	7.13	11.23	21.67	33.89	54.56
Factor of increase		1.58	1.93	1.56	1.61
Total Spreading					
Average wall time (s)	11.49	16.46	31.63	42.94	64.30
Factor of increase		1.43	1.92	1.36	1.50

Table 3: Weak scalability tests (set 2), average wall time in seconds and factor of increase of time after one refinement of the mesh.

4.3 Weak scalability

The weak scalability tests are performed in two sets of different problem sizes each for total and partial spreading. The average wall time per time step and the factor of increase are shown in Table 2 and 3. Each of experiments ensures the same workload per core for sequential and parallel implementation. There are 66564 elements per core in set 1 and 264196 elements per core in set 2. The factor of increase is introduced to measure the execution time grows from one mesh refinement to the next. Ideally, this factor is expected to be close to 1 to show that the algorithm is able to achieve a good performance in parallel.

There are several test cases with the factors around 1.9 which contains all cases on 16 cores and a partial spreading case on 64 cores. One reasons is that the number of iterations is increased as the grid refined. However, this can not explain all of these observations. In total spreading, as shown in Table 1, the number of total iterations decrease from 32/4 to 30/4 for 16 cores in set 2 (which has a factor of 1.92). Therefore, another reason is considered, namely, that the architecture of the computing cluster also has large influence on the performance and it is discussed in the following.

	System Size					
Δt	66564	264196	1052676	4202500	16793604	67141636
$1.5h$	25 / 3	25 / 4	24 / 4	25 / 5	25 / 6	25 / 7
$h/4$	21 / 2	23 / 3	23 / 4	23 / 5	24 / 5	24 / 6
$h/10$	21 / 2	22 / 3	22 / 4	23 / 4	23 / 5	24 / 5

Table 4: Average number of nonlinear and inner CG iterations of partial spreading cases over 10 time steps.

	System Size					
Δt	66564	264196	1052676	4202500	16793604	67141636
$1.5h$	No convergence			53 / 5	44 / 6	39 / 7
$h/4$	597 / 3	44 / 3	39 / 4	35 / 5	32 / 5	30 / 6
$h/10$	41 / 3	35 / 3	32 / 4	30 / 4	28 / 5	27 / 5

Table 5: Average number of nonlinear and inner CG iterations of total spreading cases over 10 time steps.

5 Discussion

5.1 Choice of time steps

As mentioned in Section 3.4, the CG solver with an AMG preconditioner will provide almost constant number of iterations, shown in Table 4 and 5 with the same error tolerance for Quasi-Newton iterations. The first number represents the average nonlinear iterations over 10 time steps for solving a sub-system in (19) and the second number is the average CG iterations for solving the system of $M + \sqrt{\beta}K$ in (26) and (27).

The number of Quasi-Newton iterations also remains in most of the cases within acceptable bounds. For partial spreading, the choice of Δt has little effect on the number of iterations. However, in total spreading, it takes more nonlinear iterations for coarser mesh in both space and time domain. At $\Delta t = h/4$, where h is the mesh size, the problem in a size of 66564 takes ten times as many nonlinear iterations as $\Delta t = h/10$ to achieve the same stopping criteria. Furthermore, it never converge for total spreading at $\Delta t = 1.5h$ until a finer mesh is used.

5.2 Performance influenced by the architecture of HPC

The weak scalability results are further investigated by computing the average time cost for one inner linear iteration. Figure 2 shows the average execution time for one inner linear iteration with 65536 degree of freedom

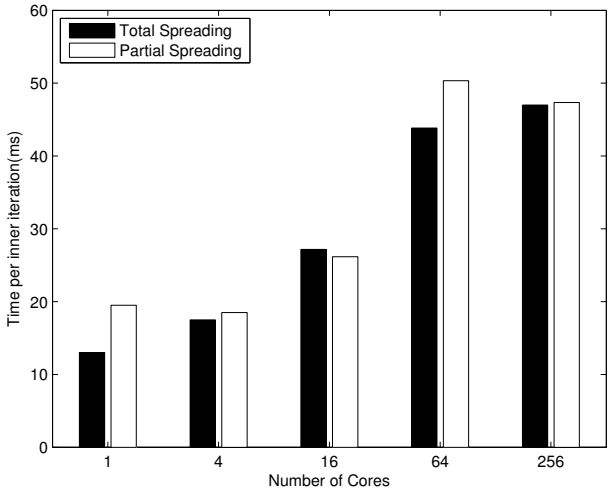

Figure 2: Time cost (ms) per inner linear iteration for the problem size of 65536 dof per core.

per core and Figure 3 is the time cost for 4 times larger problems.

There is no big difference in the execution time of one inner iteration between total and partial spreading. As a result, the extra time costs in total spreading cases are mainly due to the difficulties in solving the nonlinear system.

Considering the architecture of the computing cluster, which has 160 nodes and each node has two 8-core processors, the time cost for 1 and 4 cores are almost the same in these two figures since the program is still running on one processor. The parallel performance is nearly ideal and problem size fits in the local cache memory.

However, when the problem is expanded to 16 cores, the time cost is doubled although it is still in one node. These 16 cores are distributed in two processors on the same node that the communication between processors is much heavier than inside and the usage of cache and local memory gets less efficient. Also, it is the reason why in Table 2 and 3 we see a large decrease in performance all at 16 cores with factors around 1.9 after scaled up from 4 cores.

For the larger problem size (Figure 3), the time costs grow with low factors when the 64 cores are in used. As it only takes 4 nodes, the communication among nodes in such a simple network does not affect much the performance. Finally, when the work is spread among the network to 16 nodes (which contains 256 cores), the overhead of communication becomes more

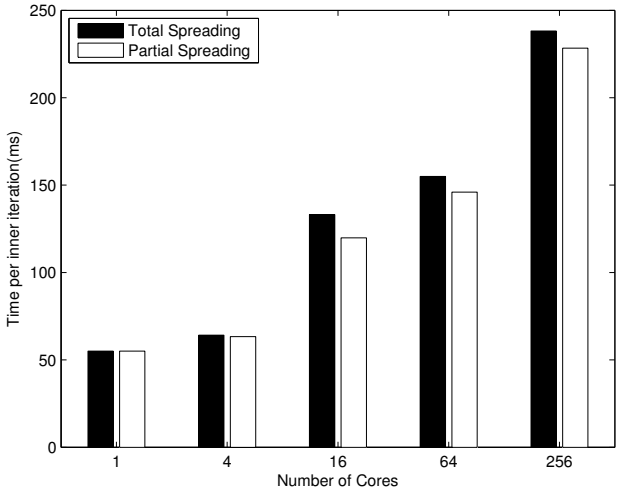

Figure 3: Time cost (ms) per inner linear iteration for the problem size of 262144 dof per core.

significant as the topology of network becomes complicated. On the other hand, in the smaller problem (Figure 2), the communication can not be overlapped by the computation even at 4 nodes as well as 16 nodes. However, by multiplying the number of iterations, the factors of increase in time are still acceptable to scale up the work.

5.3 Expand to multiphase problems

The results for number of iterations and the time costs reveal the fact that performance does not go down by adding one more phase from the two phase problems (see [4]). As the system of each phase is independent to each other in (18), the algorithm can be expanded to multiphase problems by adding the same block matrix in the block diagonal form. Meanwhile, the solving procedures do not have big difference among phases, even between total and partial spreading the average execution time per iteration are almost the same. All above combined with multiple right-hand-side methods will provide an improved algorithm for multiphase problems.

6 Conclusions

The scaling behavior of our test implementation demonstrates that our approach can utilize modern clusters. This shows promise for solving the large

problems that originate from real world scenario where such big clusters are required for computing results in reasonable time. Further, when comparing the behavior of our implementation to a previous implementation of the two-phase problem, we see similar behavior which indicates that this can be extended to multi-phase problems with more than three phases, provided that the approximation $\mathcal{A}_0$ of the Jacobian $\mathcal{A}$ preserves its good quality.

Acknowledgement

The computations were performed on resources provided by SNIC through Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX) under project p2009040.

References

[1] Uppsala multidisciplinary center for advanced computational science. http://www.uppmax.uu.se/.

[2] The deal.ii libray. http://www.dealii.org/, 2013.

[3] The trilinos libray. http://trilinos.sandia.gov/, 2013.

[4] Owe Axelsson, Petia Boyanova, Martin Kronbichler, Maya Neytcheva, and Xunxun Wu. Numerical and computational efficiency of solvers for two-phase problems. *Computers & Mathematics with Applications*, 65(3):301–314, 2013.

[5] Wolfgang Bangerth, Carsten Burstedde, Timo Heister, and Martin Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Transactions on Mathematical Software (TOMS)*, 38(2):14, 2011.

[6] Petia T Boyanova, Minh Do-Quang, and Maya Neytcheva. Efficient preconditioners for large scale binary Cahn-Hilliard models. *Comput. Meth. in Appl. Math.*, 12(1):1–22, 2012.

[7] Franck Boyer and Céline Lapuerta. Study of a three component Cahn-Hilliard flow model. *ESAIM: Mathematical Modelling and Numerical Analysis*, 40(04):653–687, 2006.

[8] Charles M Elliott, Donald A French, and FA Milner. A second order splitting method for the Cahn-Hilliard equation. *Numerische Mathematik*, 54(5):575–590, 1989.