A Parallel Multigrid Solver Based on Processor Virtualization Techniques *

Kai Wang[†] and Laxmikant V. Kale [‡]

Parallel Programming Lab
Department of Computer Science
University of Illinois at Urbana-Champaign,
Urbana, IL, 61801, USA

Abstract

We investigate the use of the processor virtualization technique in parallelizing the multigrid algorithm on high performance computers. By doing processor virtualization, we can achieve adaptive process overlapping, better cache performance, and dynamic load balance control. We use a neighbor based virtual processor to physical processor mapping strategy and dynamically changing the number of virtual processors with the multigrid levels to maximum the performance of the multigrid solver. A V-cycle multigrid solver, is accomplished based on Charm++, a message driven parallel language developed by Parallel Programming Lab at University of Illinois at Urbana-Champaign. Numerical experiments for solving convection-diffusion type equations on structured grids are reported to show the benefits we get from processor virtualization.

Key words: Processor virtualization, Charm++, MPI, multigrid algorithm, Convection-Diffusion problem.

AMS subject classifications:

1 Introduction

The multigrid method is known to be one of the most efficient techniques for numerical solutions of large classes of partial differential equations [4]. It has been successfully applied in many applications like fluid mechanics, chemical reactions in flows and flows in

*This work was supported in part by the National Science Foundation (ITR 0205611, DMR 0121695
[†]E-mail: wangkai@cs.uiuc.edu. URL: http://www.csr.uky.edu/∼kwang0.
[‡]E-mail: kale@cs.uiuc.edu. URL: http://charm.cs.uiuc.edu.

porous media [13]. For elliptic type problems, it offers a nice convergence rate independent of the grid size [4, 12, 27]. Besides a stand alone solver, the multigrid method can also be used as an effective preconditioner working with Krylov subspace methods for solving very difficult problems [3, 7].

In recent years, many efforts have been made to design multigrid solvers oriented to applications in different fields [8, 20]. The need for solving very large grid size problems arising from many important applications has been pushing the development of parallel multigrid solvers for massive parallel computers. How to accomplish an efficient and scalable parallel multigrid solver and take the advantage of the power of the modern computers are actively pursued.

A good parallel program depends not only on the underlying algorithm, but also on careful implementation. Theoretically models can be constructed to predict the ideal performance of multigrid methods [5]. However, many factors in practical implementation prevent us from getting the desired results as predicted.

The typical programming model like MPI uses a message passing mechanism [11], where one processor sends out a message, and another processor waits for the message coming. This mechanism is easy for user to control, but sometimes may produce CPU idle when a processor is blocked on receiving messages. To avoid the CPU idle, people have to design the parallel program carefully and utilize the processor waiting time, otherwise the program will show a poor behavior. Standard parallel multigrid algorithms require the grid data to be exchanged between two neighboring processors [8, 2]. A processor cannot go to the next step computation without receiving the information from all of its neighbors. That makes it difficult to insert jobs between the communication and computation to handle the processor idle.

The speed of a code on current computers depends heavily on how well the cache structure is exploited. Though strategies have been proposed to significantly optimize storage and access for caching in multigrid methods [9, 10, 24], overall performance remains far away from peak values.

Load imbalancing is another reason resulting an inefficient parallel multigrid solver. For multigrid solvers on structured grids, people achieve load balancing by investigating the solving problems, and assign roughly the same number of grid data to each processor [2] However, for unstructured grids, to divide and assign the computing jobs to each processor evenly is apparently a nontrivial task [1]. A seriously imbalanced load partition deteriorates the behavior of the final solver.

In this study, we try to develop a scalable multigrid solver by splitting the multigrid computation in each processor into many small independent jobs. Thus when one job is waiting for some data coming from the other processors, the other jobs may take the control of the CPU to do their computation. Therefore, automatic overlapping of communication and computation is achieved without revising the multigrid algorithm. Better CPU utilization can be expected. In addition, as the size of data involved in each computation is smaller compared to the original one, cache performance can also be improved. This strategy also provides us a convenient way to do the dynamically load balancing. We

can trace and statistic the load information in each processor during the run time and migrate some jobs from high loaded processor to under loaded processor dynamically.

This kind of strategy can be realized by the help of processor virtualization [18], which divides the program computation into a number of parallel parts according to the nature of the problem instead of the number of physical rocessors. Here each individual part is called a virtual processor. The processor virtualization techniques have been successfully employed in many applications which are notorious to be difficult to parallelize and got promising results [17, 21, 28].

In this paper, we investigate the using of processor virtualization techniques in parallelizing the multigrid algorithm. Several issues, the mapping from the virtual processors to physical processor, critical level, and data synchronization problems, which influence the efficiency of the multigrid solver are discussed. Specifically we design and test a prototype implementation to show processor virtualization does lead to a better scaled multigrid solver.

This paper is organized as follows. A brief introduction of the multigrid method is given in Section 2; In Section 3, we will discuss the problems encountered in parallelizing the multigrid methods; The processor virtualization concepts and its application in multigrid methods are explained in Section 4; Section 5 contains the technical details of the implementation of virtualization in multigrid methods. Section 6 gives some numerical results on a distributed memory machine to demonstrate the advantages of the strategy; Section 7 is some concluding remarks.

2 Multigrid methods

It is well known that the classical iteration methods like Jacobi or Gauss-sediel are good at reduce the oscillatory error components, but converge slowly for smooth error components [4]. Noticed that smooth error components may become oscillatory on a different spatial scale, i.e, a coarse grid, we can create a multiple grid structure so that various features of the error components can be resolved on appropriate spatial scales. Hence, after a few relaxation iterations (prerelaxation), we may compute the residual and project it to its coarse grid, where the smooth error components will become more oscillatory. The residual will be solved on this coarse grid and a new residual is computed and projected to the next level coarse grid. We can continue doing this until we reach the coarsest grid. Then the solution will be interpolated and added back to the current approximate solution on its fine grid as a correction. Some iterations (postrelaxation) are performed on this fine grid using the new solution as a initial guess and interpolated back the correction back to its fine grid. One V-cycle multigrid iteration is finished when the operation goes back to the finest grid. To reach the convergence, we need many such iterations usually.

There are several variants of the multigrid algorithm, which were designed to improve the roubustness and efficiency of the final solvers. According to the level generation, they can be categorized into geometric and algebraic [22]. According to the cycle pattern, they can be categorized into V-cycle, W-cycle, U-cycle, and full V-cycle [4, 8, 19]. According to

the coarsening direction, they can be categorized into full coarsening and semicoarsening [26, 5]. Each of them has its own merits and drawbacks. Since the processor virtualization techniques can be applied to almost all of these multigrid methods. Here we just emphasis on the standard V cycle algorithm to show the advantages brought by the processor virtualization techniques.

A $V(v_1, v_2)$-cycle multigrid algorithm is given as below [4]. Here the number in the parenthesis stands for the number of iterations in the prerelaxation and postrelaxation respectively.

Algorithm **2.1**

0. *Perform v_1 sweeps of prerelaxation iteration.*
1. *Compute residual.*
2. *Restrict residual from fine to its coarse grid.*
3. *Solve the residual equation by recursion.*
4. *Interpolate solution from coarse to its finer grid.*
5. *Update current solution.*
6. *Perform v_2 sweeps of postrelaxation iteration.*

Algorithm 2.1 is actually a serial version multigrid algorithm. Here we suppose that the grid levels have been constructed. In this paper, we are concerned about the parallel performance of multigrid methods. So in the next section, we will focus on talking about parallel multigrid methods, and the problems encountered in its real implementation.

3 Parallel Multigrid Methods

Usually the conventional parallel programming model parallelize an application based on the available physical processors. For multigrid algorithm, domain decomposition is the most convenient way to partition the data to each processor [25]. First a physical processor topology is constructed, then the finest grid data are distributed to processors accordingly. Two methods can be used here to store the coarse grid data. One way is to apply domain decompositions again to the coarse grid data. That means the coarse grid data need be redistributed across processors. This method retains the convergence rate of the serial algorithm with the cost of increasing communication in each level of grid computation. Another way is just let each processor holds the coarse grid information corresponding to its portion of fine grid. This kind of partition may deteriorate the convergence of the serial algorithm as only limited grid levels can be created, but only requires the communication of the boundary grid data among neighboring processors. For a 2D decomposition, the communication involves up to 7 neighbors. For a 3D decomposition, the communication involves up to 26 neighbors. A parallel version of the Algorithm 2.1 can be written as

Algorithm **3.1**

0. *Send and receive data before each prerelaxation.*
1. *Send and receive data before computing the residual*

2. *Send and receive data before Restriction.*
3. *Solve the residual equation by recursion.*
4. *Send and receive data before interpolation.*
5. *Update current solution.*
6. *Send and receive data before each postrelaxation.*

Here the prerelaxation and postrelaxation iteration is usually performed by some basic iterative methods like Jacobi, Gauss-sediel, or damped-Jacobi [23]. The restriction and interpolation can be done by a full weighting scheme and linear interpolation respectively [27]. All of these components are easy to be parallelized algorithmically. A theoretical model and analysis base on the model are introduced In the following discussion, we use a theoretical model to analysis the parallel performance of multigrid methods.

3.1 Theoretical model

A theoretical model has been constructed in [5] to predict the parallel performance for the semicoarsening multigrid methods. Here we give a model for standard multigrid algorithm 3.1.

The time T for each V-cycle multigrid iteration can be roughly modeled as the communication time T^c plus the computation time T^p.

$$T = T^c + T^p$$

Suppose a 3-D (D*D*D) decomposition is used in our parallel implementation, and each processor has a grid size of N^3 at the finest level, so the total problem size is $(DN)^3$ we can write the communication time T_l^c at multigrid level l as

$$T_l^c = 26\alpha + 6N_l^2\beta,$$

where a is the overhead for each communication, β is the time for each float point data transfer, and $N_l = 2^{-l} * N$ is the dimension of the grid size at level l.

The computation time T_p^l at level l is

$$T_l^p = N_l^3\gamma.$$

Here γ is the time to do one operation (relaxation, interpolation, or relaxation) on each node point, Obviously, for a m level multigrid algorithm, where $m = log_2(DN)$, N is the dimension of grid at each processor.

$$T^c = \sum_{l=0}^{l=m} T_l^c = 26m\alpha + 6\beta \sum_{l=0}^{l=m} 2^{-l} N_l^2 = 26log_2DN\alpha + 8N^2\beta,$$

$$T^p = \sum_{l=0}^{l=m} T_l^p = 8/7N^3\gamma.$$

So the time T can be approximately expressed as

$$T = 26log_2(DN)\alpha + 8N^2\beta + 8/7N^3\gamma. \quad (1)$$

From Eq. 1, we can see that when α, β, γ, and N are fixed, the CPU time spent on each multigrid iteration is affected by the number of processors. The more processor we use, the lower scaled efficiency we get.

Obviously, when we increase the problem size per processor, both the communication time (first two terms in Eq. 1) and the computation time (last term in Eq. 1 increase. However, noticed that the time of computation depends on N^3, and the time of communication depends on N^2, the scaled efficiency should be improved theoretically.

However, in the real implementation, we found that it is difficult to get a higher scaled efficiency when problem size per processor is large, sometimes, a decreasing scaled efficiency may even be observed. In Figure 1, we give the relationship between the number of processors and scaled efficiency when using a straightforward MPI implementation of the multigrid V(1,1) cycle algorithm introduced in 3.1 to do 10 iterations for a 3-D problems. We can see that the multigrid solver gets a scaled efficiency nearly 0.7 when the problem size in each processor is $32*32*32$. But when we go to a larger problem size, the scaled efficiency is not increasing too much as predicted. When we look at the processor utilization, we can see that the processor utilization for the large problem size is only about 60 percent, it is lower compared to the 90 percent of the problem size $32*32*32$.

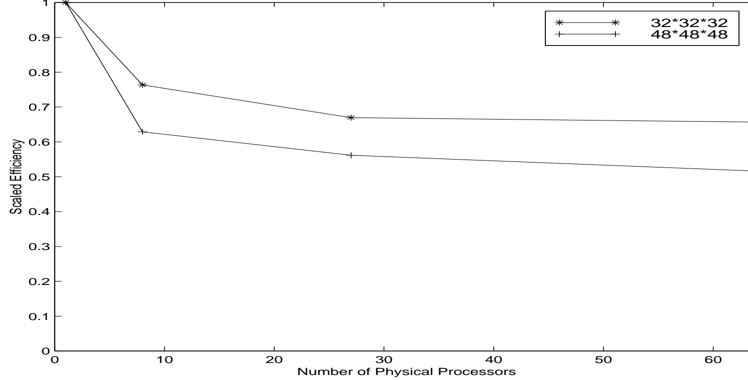

Figure 1: Scaled efficiency of an MPI version multigrid code.

Why this happens? It may come from many reasons. First is the cache performance. Large problem size worsen the memory locality, makes both the communication costs and computational cost increasing, and leads to an inefficient program. From the Algorithm 3.1 we can see that each processor can only go to the next step computation after got all the data from its neighbors. It is very possible that at some states the processor is idle and waiting for data coming which arises from different reasons including the different processor loads or powers. When the problem size is large, the time of communication is increased. That makes the processor idling serious, and heavily affects the final performance of the multigrid program. Especially when the number of processors is large, each processor may

need to communicate with more neighbors (in 3-D case, it is up to 26). The possibility of block for receiving is more seriously.

So careful implementation should be performed in developing. a successful parallel multigrid solver. That includes the concerns about the physical processor topology, cache locality, and load balancing. In the next section, we introduce why the processor virtualization techniques can achieve this goal.

4 Processor virtualization in multigrid methods

The concept of processor virtualization is first presented in the past decade. The basic idea in virtualization is to divide the program computation into a number of virtual processors. Then the program will be developed just like we have that many of physical processors instead of considering the real number of physical processors. The run time system takes the responsibility to map the virtual processors to physical processors. Usually the number of virtual processor should be large than the number of physical processors to guarantee each physical processor can hold at least one virtual processor.

Processor virtualization admits the most natural decomposition of the problem rather than being restricted by the physical machine. It will obvious bring software engineering benefits from the first glance. For example, the straightforward way to deal with a 3-D problem is to use a uniform 3D processor topology, which is impossible for some numbers of physical processors like 7, 11. Processor virtualization can do this by giving a number of virtual processors which can form a desired 3-dimension structure no matter the number of physical processors is prime or not.

The idea of processor virtualization is simple, but great benefits can be obtained, which are stated as follows.

- High processor utilization.

  Suppose each physical processor manages at least two virtual processors, if one of the virtual processors is blocked on receiving data, another virtual processor on the same physical processor can take the control to do its communication and computation. The adaptive overlap of communication and computation largely eliminates the probability of the physical processor idling, and increases processor utilization.

- Better cache performance.

  Compared to the original physical processor, each virtual processor handles a smaller set of data. So a virtual processor may have better memory locality in both the communication and computation. This blocking effect is the same strategy that many serial cache optimizations employ. So even a large set of grid data are assigned to the physical processor, the cache performance can be guaranteed by adjusting the number of virtual processor.

- Easy load balancing.

The load balancing strategies of parallel programming can be classified into two categories. Carefully divide the problems in the initial load distribution, or adjust the load during the run time. The first method can be employed on some problems with regular structure and stable communication and computation models. Like the multigrid methods or structured grid on a uniform domain, generally good load balancing can be achieved by assign each processor an approximately the same number of grid data. But for unstructured grid, evenly partition the problem domain may encounter some difficulties. An alternative way is to balance the load by tracing the program execution and marking the overloaded and underloaded processor. However, this strategy must deal with the questions like how to divide the overloaded job effectively? how to migrate the job efficiently? and the most important of all how to keep the data structure of the program completeness after the migration so that the program can run safely? By processor virtualization, these questions can be answered easily. As each virtual processor is an independent unit, we can migrate a few virtual processors from overloaded physical processor to underloaded physical processor directly. The whole algorithm structure does not need to be changed.

More benefits can be got by employing processor virtualization in parallel programming. Here we only mention those related to the multigrid implementation. For detail information, we refer readers to [18].

Charm++ is a C++ based programming system that supports the processor virtualization programming model. A virtual processor can be expressed as a chare object in Charm++, which contains methods can be invoked by by other chare object. That makes the communication between virtual processor possible. Contrasted to the message passing model of MPI scheduler, the Charm++ adopts a message driven execution model so that adaptive overlapping between communication and computation can be achieved naturally. Dynamic load balancing is supported by Charm++ by its integrated measurement and migration mechanisms. More detail of Charm++ programming model can be obtained from its website [6], or refer to paper [14, 15, 16].

5 Implementation issues

The success of a software depends on sophisticated implementations as heavily as on the novel underlying ideas. In this section, we discuss several implementation issues that need to be addressed to build an efficient multigrid solver for distributed memory parallel computers. Here we suppose the virtual processor structure is already constructed. Grid data, righthand sides, and solutions are distributed across the processors using domain decomposition.

5.0.1 Processor mapping

In processor virtualization, a virtual processor acts just like an independent physical processor. It has its own set of data, does its own computation, and gets needed information

from other virtual processors through message communication. When virtual processors are mapped to physical processors, these communication may be identified as the inner processor communication, which happens among the virtual processors in the same physical processor. or across processor communication, which happens among the virtual processors resided in different physical processors. The across processor communication is not only inefficient compared to inner processor communication, which actually triggers a memory copy, but also tend to cause the processor idling when there is a critical path existing.

According to the multigrid algorithm, a virtual processor only communicates with its neighbors. If the number of virtual processor and their spacial position are determined, the total number of virtual processor to virtual processor communication in each multigrid iteration will be fixed. In this case, different mappings yield different communication ratios, which is the number of inner processor communication divided by the number of across processor communication. The desired mapping should find the largest communication ratio in addition to a balanced load.

By default, the Charm++ uses a random mapping, and optimizes the processor load by some load balancing strategies thereafter. But generally it is difficult for a load balancing strategy to detect the communication model during the run time, and the migration of virtual processor is not a cheap operation. A reasonable solution is to give a good initial mapping. Figure 2 illustrates three different methods that map a $4 * 4$ virtual processor array to 4 physical processors. The virtual processors assigned to the same processor are colored as the same color. The message communication between two virtual processors with different colors is across processor message.

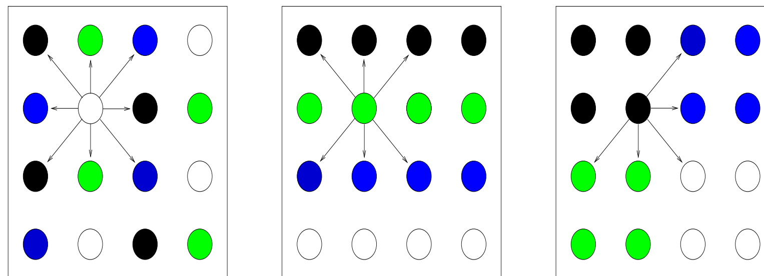

Figure 2: Across processor messages using different mapping strategies.

The mapping shown on the left figure is an extreme case of random processor mapping. We can see that in this case all the messages go to a different physical processor. The communication ratio here is 0. The mapping shown in the middle figure is a linear mapping, which puts the virtual processor numbered from 0 to 3 to physical processor 0, 4 to 7 to physical processor 2,...... in order. This strategy works better than random strategy, but not as good as the one shown on the right, which is the best mapping for this example.

To find the best mapping strategy is a very challenging topic. Here we give a block mapping algorithm, which try to avoid the neighbor virtual processor cutting during mapping by considering the topology of the physical processor. It is not the optimized solution for our problem, but can achieve our goal to some extent.

ALGORITHM **5.1**

0. *Suppose the domain decomposition does a n dimension decomposition.*
1. *Build a n dimension topology of the physical processors.*
2. *Build a n dimension topology of the virtual processors.*
3. *For the virtual processor on dimension i*
4. *Map it to the corresponding physical processor on dimension i by linear mapping.*
5. *End For*

Note here that this block mapping will produce the same mapping result as the linear mapping under 1 dimension decomposition. It will create the best mapping for the example shown in Figure 2

5.0.2 Data synchronization

To achieve automatic overlapping, message driven execution model is a have to. No virtual processor can hold the processor idle while it is waiting for its message. Instead, the virtual processor has a message waiting for it is allowed to continue. Howeve, message driven execution model leads to data asynchronous. So we must deal with the racing condition, which arises when a processor keep receiving messages from the other processors. There are two cases.

- Let processor A and B are neighbors, which means that there are message communication between each other. Suppose at iteration i, A sends out all the messages that the other processor needed and gets what it needs from the other processors, then it goes to the next iteration, where it will send and receive messages again. However, processor B may still stay at the ith iterations and wait for a message from the other processors. At this case, the new message from A may conflict with the old message from A.

  Usually doing a global reduction operation before each communication to make data synchronous is a straightforward solution, however the cost is too high, since a all to all message communication is needed. Noticed that in the multigrid algorithm, communications only happens between neighboring processors, which is usually a subset of the total processors, we only need to keep the data in neighbors to be synchronous. That can be done by sending a ready message to its neighbors, a virtual processor only begins its communication when it gets all the messages from its neighbors. This method is more efficient than a global reduction. However, the adding cost of the ready message communication is nontrivial, especially for some machines with a big message overhead. So a better way is to do the buffering. A data received in an appropriate time will be buffered until the right time comes.

- Another possibility is that processor A gets all its messages and is pushed to the next iteration by the message driven scheduler immediately without sending out any data the other processors needed. In this case, we can enforce the virtual processor to send a message to itself after sending all the data out, and a virtual processor will go to the next step computation only after it gets all the messages from the other neighbor processors including the message from itself.

The data communication algorithm can be written as follows.

ALGORITHM **5.2**

Function sendData()

0. *For each of its neighbors j*
1.   *Send the corresponding data to j.*
2. *Send a get data message to itself.*

Function recvData(data)

0. *Check synchronous.*
1. *If asynchronous*
2.   *Buffer the data.*
3. *Else*
4.   *Put the data into the corresponding positions.*
5.   *If receive all the data from its neighbors and*
    receive the message from itself
6.   *Begin computation.*
7.   *If there are buffered data*
8.   *Pop the data out.*
9.   *recvData(data).*
10.   *End if*
11.   *End if*
12. *End if*

5.1 Number of Levels

One problem in parallelization of multigrid methods is the happen of critical level, where a processor contains less data than needed to produce its coarse grid level. As the number of virtual processors is required to be larger than the number of physical processors, the critical level problem is more serious. Too few multigrid levels will deteriorate the convergence behavior of the multigrid solver. Therefore, here we define the number of multigrid levels as a parameter, and dynamically decrease the number of virtual processors between the critical level by migrating and combining the data in one virtual processor to its neighbor virtual processor.

This may lead to a phenomena that in some grid levels, where the number of virtual processors is less than the number of physical processors, some physical processors may be idle. This seems a waste of computer time, but our numerical results shows that this

strategy pays. It does gain a faster convergence speed by keeping a small number of iterations.

6 Numerical results

In this section, we show the performance of our multigrid solver based on the processor virtualization techniques introduced in previous sections for solving 3D convection diffusion equations.

Convection-diffusion problem. A three dimensional convection-diffusion problem (defined on a unit cube)

$$u_{xx} + u_{yy} + u_{zz} + \lambda\,(p(x,y,z)\,u_x + q(x,y,z)\,u_y + r(x,y,z)\,u_z) = 0 \qquad (2)$$

Eq. (2) is very important in computational fluid dynamics to model the transport phenomena. Here λ is so called Reynolds number. The convection coefficients are chosen as

$$\begin{aligned} p(x,y,z) &= x(x-1)(1-3y)(1-2z), \\ q(x,y,z) &= y(y-1)(1-2z)(1-2x), \\ r(x,y,z) &= z(z-1)(1-2x)(1-2y). \end{aligned}$$

Eq. (2) can be discretized by using 19-point fourth order compact difference scheme [27]. It is generally regarded that the resulting coefficient matrix tends to be ill-conditioned with a large Reynolds number.

The fine to coarse restriction and coarse to fine interpolation are done by using the full weighting scheme. We us 4-color gauss-sediel iteration to be the inner grid smoother and coarsest grid solver. The algorithm is realized by Charm++. The computation are carried out on a 208 dual-processor Linux cluster at the University of Illinois at Champaign-Urbana. Each processor is 1GHz pentium III with 1GB RAM. All the CPU times reported are measured in seconds.

Mapping strategies First we compare the performance results when using different processor mapping strategies in the multigrid solver. We try to solve a 3D convection diffusion problem using 32 virtual processors on 8 physical processors. The Reynolds number is chosen as 1000. The CPU times are reported on 100 multigrid iterations. The results are shown in Table 6, where 'Default' is the default mapping method of Charm++, 'Linear' stands for linear mapping, and 'Block' means the block mapping which is introduced in Algorithm 5.1.

From Table 6, we can see that the block mapping strategy offers the fastest running results. This because the block mapping strategy decreases the number of high cost across processor messages as well as the critical paths. In the following numerical tests, we let our code use the block mapping strategy.

Problem Size	Strategy	Time
32*32*32	Default	11.72
	Linear	9.98
	Block	8.37
64*64*64	Default	55.67
	Linear	46.38
	Block	43.89
128*128*128	Default	548.73
	Linear	451.68
	Block	372.08

Problem Size	level	iterations	Time
$32 * 32 * 32$	5	9	0.75
	4	16	1.31
	3	59	4.33
$64 * 64 * 64$	6	9	3.69
	5	17	6.35
$128 * 128 * 128$	4	58	22.41
	7	10	36.97
	6	17	60.10
	5	58	210.78

Number of Levels The number of multigrid levels greatly affects the convergence of the multigrid solver. For serial multigrid algorithm, the convergence can be improved by a precise coarsest grid solver, i.e., a direct solver, even with several multigrid levels. But it is difficult to find a good parallel coarsest grid solver. The 4-color Gauss-sediel solver used in our parallel multigrid implementation is not a accurate solver. So we had better to construct as many multigrid levels as possible. This can be demonstrated by Table 6, where we gives some tests for solving 3D Convection-diffusion equations with Reynolds number 10 by different multigrid levels. 8 physical processors are used using 32 virtual processors. 'iterations' in table stands for the number of multigrid iterations to decrease the relative residual by 10 orders of magnitude. Here we also use 32 virtual processors on 8 physical processors. The number of virtual processors will be changed so that we can create enough multigrid levels below the critical level, which is for example, 3, for the problem size $32 * 32 * 32$ From Table 6 we can see that even the migration of virtual processors may take time and make processors idle, but as it only happens in the grid levels with a small set of grid data, the costs are small and can be compensated by less number of multigrid iterations.

Degree of virtualizations The degree of virtualizations is the number of virtual processors in a physical processor. If the degree of virtualization is 1, it is the same as traditional parallel program. Usually the utilization of processors can be improved with

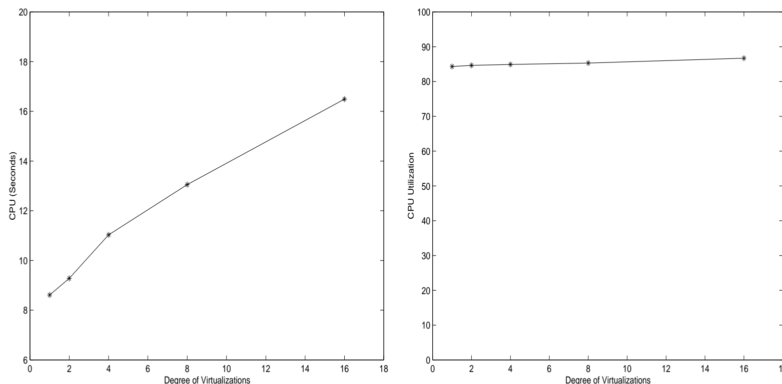

Figure 3: Left: Relationship between the degree of virtualizations and CPU time. Right: Relationship between the degree of virtualizations and processor utilization. Grid size=$32 * 32 * 32$. 100 iterations. 4 physical processors.

a higher degree of virtualizations. But as more virtual processors also bring additional computational and communication costs, it is not the case that the more virtual processors, the better. Figure 6 gives the performance of CPU time and processor utilization with the degree of virtualization increasing when solving a problem size of $32 * 32 * 32$ on 4 physical processor with 100 iterations. As the problem size is very small (each physical processor gets roughly 8000 grid data), the communication and computation can be finished in a very short time for each multigrid iteration. Neither processor utilization nor cache performance is the hurdle. More virtual processors can only make the program running slowly.

People care more about the parallel performance of their program when they are dealing with large problem size, where we do see the advantages of processor virtualization. Figure 6 gives another test for solving a larger grid size $64*64*64$ on 4 physical processors with 10 iterations. We can see a reduced CPU time with the degree of virtualization increasing from 1 to 4, corresponding to a clearly improved processor utilization. After that the CPU time will go up as the processor utilization will not be improved too much anymore.

How to choose the best degree of virtualization is a very important topic. We think that 4 virtual processors in each physical processor is generally not a bad choice when the problem size is large enough, this can also be illustrated from Figurefig:degree.

Code Performance Our parallel multigrid solver inherits the properties of standard multigrid solver like the grid independent convergence. It can solve the 3D convection-diffusion with Reynolds number up to 10000 in several hundred iterations (Table 6.

The convergence of our multigrid solver can be improved using more inner iterations, which is demonstrated in Table 6, where we try to solve the same problem in last column

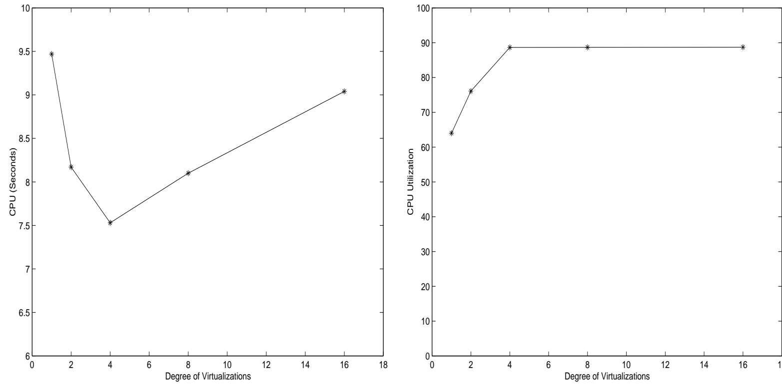

Figure 4: Left: Relationship between the degree of virtualizations and CPU time. Right: Relationship between the degree of virtualizations and processor utilization. Grid size=$64 * 64 * 64$. 10 iterations. 4 physical processors.

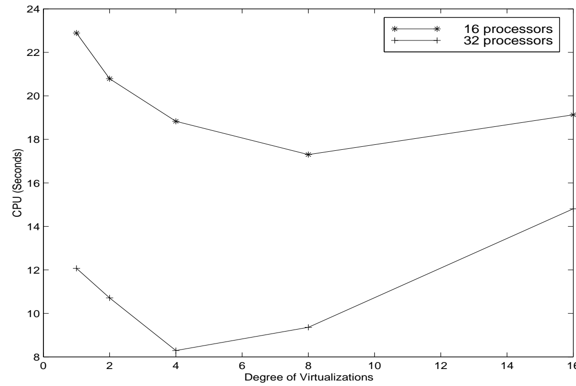

Figure 5: Relationship between the degree of virtualization and CPU time when solving a 3D Convection-diffusion problem. 10 iterations. Reynolds number = 100.

of 6 using different inner iterations.

Scalability test We do the scalability tests by fixing the grid size in each physical processor to be $48 * 48 * 48$. The results are reported in Figure 6, where different degrees of virtualizations are used. To be comparable, we also list the performance of a standard multigrid solver developed by MPI.

From Figure 6 we noticed that when the degree of virtualizations is 1, the performance of our solver is almost the same as that of the MPI version. Actually, their CPU times are almost the same, as when assigning 1 virtual processor to each physical processor make the solver do the same thing as the MPI version. When the degree of virtualization equals to 4, our multigrid solver gives the best scaled efficiency of all, which is up to 0.7 in 64 processors case. Compared to the scaled efficiency of the MPI version, we see that the

Reynolds Number	iterations	Time
0	9	15.75
1	9	15.86
10	10	17.13
100	16	27.89
1000	70	120.44
10000	274	472.64

Table 1: Solving 3D convection diffusion problems with different Reynold numbers. Inner iteration=1.

Smooth Iterations	iterations	Time
1	274	472.64
2	120	310.17
3	84	293.50
4	66	278.59
5	53	281.67

processor virtualization does shows its advantage.

7 Conclusion

In this paper, we propose to build scaled multigrid solver based on processor virtualization. Some implementation details are given for a developing a high efficiency multigrid solver. From the numerical results we can see that processor virtualization improved the performance of multigrid solver by improving the processor utilization and cache performance.

The numerical results are given based on structured grids, which do not have serious load balancing problems as long as we evenly divide the problem domain. On unstructured grids, the load balancing problem may be a dominant reason for the poor performance of multigrid solver. In that case, processor virtualization may shows its more benefits for its easy migration mechanism. So developing multigrid solver on unstructured grids will be our next step work.

As shown in our paper, initial processor mapping has important influence on our final multigrid solver. We give a block mapping strategy to show its advantage over the other processor mapping strategy. However, it is not our final conclusion, since the block mapping cannot produce the best mapping in some cases. Future work in this direction should focus on developing more effective mapping strategies and integrating them into the load balancing strategies.

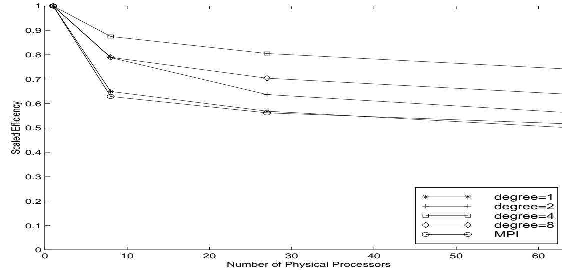

Figure 6: Comparison of scaled efficiency. Problem size in each processor= $48 * 48 * 48$. 10 iterations. Reynolds number = 100.

References

[1] P. Bastian, Load balancing for adaptive multigrid methods, *SIAM J. Sci. Comput.*, 19(4):1303-1321.

[2] A. Brandt, Multigrid solvers on parallel computers, in *Elliptic Problem Solvers*, M. H. Schultz, ed., Academic Press, New York, 1982, 39-83.

[3] D. Braess. On the combination of multigrid method and conjugate gradients, in *Multigrid Methods II*, W. Hackbusch and U. Trottenberg, eds, vol.1228 of Lecture Notes in Mathematics, 52-64, Springer-Verlag, 1986.

[4] W. Briggs, A multigrid tutorial, SIAM, Philadelphia, 1988.

[5] P. Brown, R. Falgout, J. Jones, Semicoarsening multigrid on distributed memory machines, *SIAM J. Sci. Comput.*, 21(5):1823-1834, 1999.

[6] Homepage of Parallel Programming Lab at Department of Computer Science, University of Illinois at Urbana-Champaign: http://charm.cs.uiuc.edu.

[7] W.Davids and G. Turkiyyah, Multigrid preconditioners for unstructured nonlinear 3D finite element models, *Journal of Engineering Mechanics*, 125(2):186-196, 1999.

[8] C. Douglas, A review of numerous parallel multigrid methods. In SIAM *News*, vol 12, May 1992.

[9] C. Douglas, Caching in with multigrid algorithms: Problems in two dimensions, *Paral. Alg. Appl.*, 9:195-204, 1996.

[10] C. Douglas, J. Hu, M. Kowarschik, U. Rude, C. Weiss, Cache optimization for structured and unstructured grid multigrid, *Elect. Trans. Numer. Anal.*, 10:21-40, 2000.

[11] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, 1994.

[12] M. Gupta and J. Zhang, High accuracy multigrid solution of the 3D convection-diffusion equation, *Applied Math. and Compu.*, 113(2-3):249-274, 2000.

[13] W. Hackbusch, Multigrid Methods and Applications, Springer and Verlag, 1985.

[14] L. Kale, B. Ramkumar, A. B. Sinha, and A. Gursoy, The CHARM parallel programming language and system: Part I – Description of language features, *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[15] L. Kale, B. Ramkumar, A. B. Sinha, and A. Gursoy, The CHARM parallel programming language and system: Part II – The runtime system, *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[16] L. Kale, S. Krishnan, Charm++: Parallel programming with message-driven objects, In *Parallel Programming Using C++*, G. V. Wilson and P. Lu, editors, MIT Press, 1996, 175-213.

[17] L. Kale, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, NAMD2: Greater scalability for parallel molecular dynamics, *J. of Comp. Phy.*, 151:283-312, 1999.

[18] L. Kale, The virtualization model of parallel programming: runtime optimizations and the state of art, In *LACSI 2002*, Albuquerque, October 2002.

[19] I. Llorente, M. Prieto-Matias, and B .Diskin, An efficient parallel multigrid solver for 3-d convection-dominated problems, ICASE Report No. 2000-29, NASA/CR-2000-210319, 2000.

[20] MGNet Homepage: http://www.mgnet.org/.

[21] M. Nelson, W. Humphrey, A. Gursoy, A .Dalke, L. Kale, R. Skeel, and K. Schulten, NAMD - a parallel, object-oriented molecular dynamics program, *International Journal Supercomputing Applications and High Performance Computing*, Winter 1996, Volume 10, number 4.

[22] J. Ruge and K. Stiiben, Algebraic multigrid(AMG), In *Multigrid methods, volume 3 of Frontiers in Applied Mathematic*, S. F. Mccormick, editors, 73-130, SIAM, Philadelphia, PA, 1987.

[23] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing, New York, NY, 1996.

[24] S. Sellappa, S. Chatterjee, Cache-efficient multigrid algorithm, In *Proceedings of the 2001 International Conference on Computational Science*, San Francisco, CA, 2001.

[25] B. Smith, P. Bjorstad, and W. Gropp, *Domain Decomposition*, Cambridge University Press, Cambridge, 1996.

[26] R. Smith, A. Weiser, Semicoarsening multigrid on a hypercube, *SIAM J. Sci. Stat. Comput.*, 13:1314-1329, 1992.

[27] J. Zhang, An explicit fourth-order compact finite difference scheme for Three Dimensional Convection-diffusion Equation, *Commun. Numer. Methods Engrg.*, 14:209-218, 1998.

[28] G. Zheng, G. Kakulapati, L. Kale, BigSim: A parallel simulator for performance prediction of extremely large parallel machines, In IPDPS, April, Santa Fe, New Mexico, 2004.