Hindawi Publishing Corporation
Scientific Programming
Volume 2016, Article ID 2360492, 14 pages
http://dx.doi.org/10.1155/2016/2360492

Research Article

A New Parallel Method for Binary Black Hole Simulations

Quan Yang,[1] Zhihui Du,[1] Zhoujian Cao,[2] Jian Tao,[3] and David A. Bader[4]

[1] *Tsinghua National Laboratory for Information Science and Technology, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China*
[2] *Institute of Applied Mathematics, Academy of Mathematics and Systems Science Chinese Academy of Sciences, Beijing 100190, China*
[3] *Center for Computation & Technology, Louisiana State University, Baton Rouge, LA 70803, USA*
[4] *School of Computational Science and Engineering, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA*

Correspondence should be addressed to Zhihui Du; duzh@tsinghua.edu.cn

Received 1 January 2016; Revised 25 May 2016; Accepted 6 June 2016

Academic Editor: Bormin Huang

Simulating binary black hole (BBH) systems are a computationally intensive problem and it can lead to great scientific discovery. How to explore more parallelism to take advantage of the large number of computing resources of modern supercomputers is the key to achieve high performance for BBH simulations. In this paper, we propose a scalable MPM (Mesh based Parallel Method) which can explore both the inter- and intramesh level parallelism to improve the performance of BBH simulation. At the same time, we also leverage GPU to accelerate the performance. Different kinds of performance tests are conducted on Blue Waters. Compared with the existing method, our MPM can improve the performance from 5x speedup (compared with the normalized speed of 32 MPI processes) to 8x speedup. For the GPU accelerated version, our MPM can improve the performance from 12x speedup to 28x speedup. Experimental results also show that when only enough CPU computing resource or limited GPU computing resource is available, our MPM can employ two special scheduling mechanisms to achieve better performance. Furthermore, our scalable GPU acceleration MPM can achieve almost ideal weak scaling up to 2048 GPU computing nodes which enables our software to handle even larger BBH simulations efficiently.

1. Introduction

The latest supercomputers [1] have significantly increasing number of computing nodes/cores, but many practical applications cannot achieve better performance on more computing resources because enough parallelism in the applications has not been explored. At the same time, many applications cannot take advantage of supercomputers equipped with accelerator such as GPU [2, 3] because the existing codes cannot run on GPU directly. We focus on the two problems and propose an efficient method to improve the performance of one kind of challenging scientific application (binary black hole simulations) on one typical large scale supercomputer, Blue Waters.

A binary black hole (BBH) system has two black holes in close orbit around each other. For the inspiralling BBH, the two black holes will move around each other. The number of orbits means the number of circles the black holes move. Mass ratio means the ratio of the mass of small black hole to the mass of the big black hole. BBH systems are important because they would be the strongest known gravitational wave [4] source in the universe. The most challenging and important subject in numerical relativity now is the simulation of these BBH systems. For gravitational wave detection, theoretical model for gravitational wave sources is essential for experimental data analysis. As an example, the theoretical model for BBH played an important role in GW150914 detection [5]. For ground based detectors of gravitational wave, such as LIGO, VERGO, GEO600, and KAGRA [6], the almost equal mass BBHs are the most important gravitational wave sources. For the space-based detectors, such as eLISA [7], the gravitational wave sources will include many large mass ratio BBH systems. When the mass ratio of BBH increases, the computational cost increases dramatically, roughly proportional to the fourth power of the mass ratio. Currently, the upper limit for the mass ratio of BBH which can be simulated by numerical relativity is 100. So, for the future space-based

detection of gravitational wave, improving the computational ability is quite important.

Adaptive mesh refinement (AMR) [8] is widely used in BBH simulation because of its simplicity and great reduction in total computing work. It is a natural method to divide each mesh into many submeshes and execute the submeshes of the same level in parallel to improve the simulation speed. We call it Submesh based Parallel Method (SPM) in this paper. Halo zones are needed for each submesh to keep the data from its neighbors. SPM can really achieve very good parallel performance when it only uses limited computing resources. But if the time saving by parallel executing the smaller submeshes cannot compensate the communication overhead to fill the halo zones, SPM cannot scale to more computing resources to achieve better performance. The size of submeshes cannot be too small because of the increasing communication overhead and the size of refined mesh cannot be very large because of the significant increase in total computation, so the number of parallel submeshes (*MeshSize*/*SubmeshSize*) cannot be too large. The latest supercomputers have more and more computing nodes/cores, but the SPM cannot take advantage of more computing resources to further improve its performance. When we try to handle some challenging BBH simulations with more orbits (≥20) and larger mass ratio (≥400), more parallelism must be explored to significantly improve the simulation performance to conduct the simulations in reasonable time.

We propose a novel Mesh based Parallel Method (MPM) to explore both the inter- and intramesh parallelism in BBH simulations. MPM will explore the parallelism among different mesh levels first (intermesh parallelism). Then, it employs SPM to explore the parallelism in each mesh (intramesh parallelism). MPM has two advantages. On one hand, for given submesh size, MPM can provide much more parallelism than SPM does. On the other hand, for given number of parallel tasks, MPM can assign more data and computation to each parallel task than SPM does. In other words, the computation-to-communication ratio of MPM can be larger than the SPM. So MPM has higher parallel efficiency than the SPM does.

We develop a new mesh partitioning algorithm to optimize the load and communication among all the parallel tasks. Our mesh partitioning algorithm is employed into our MPM to achieve balanced load and reduce communication as much as possible. In order to take advantage of the GPU capability to improve the performance, we rewrite the most computationally intensive solver codes in BBH simulations on GPU and employ different kind of GPU codes optimization methods (employing coalesced memory access and shared memory, reducing data copy between CPU and GPU, and removing unnecessary synchronization) to improve its performance.

Different kinds of experiments on performance evaluation have been conducted on the large scale Blue Waters supercomputer at National Center for Supercomputing Applications (NCSA). The experimental results show that significant performance improvement can be achieved with our scalable MPM. Because the sequential code is too slow, we select the performance of 32 MPI processes with SPM algorithm as the baseline. The existing method can achieve at most 5x speedup; our MPM can achieve 8x speedup. After we port and optimize the code on GPU, the existing method can achieve at most 12x speedup; our scalable MPM can achieve about 28x speedup. Furthermore, our scalable MPM + GPU acceleration method can achieve almost ideal weak scaling up to 2048 GPU computing nodes. This means that our method can handle very large problem on large number of computing resources efficiently.

The major contribution of our work lies in two aspects: First, the proposed MPM can enable the BBH simulation to take advantage of more computing resources of supercomputers to achieve better performance. Second, the proposed MPM has been enabled by GPU to further improve the performance of BBH simulation.

2. Problem Description

We will briefly introduce the BBH simulation problem and the numerical method used to solve the problem first. Then, a special mesh refinement method used in BBH simulation is given.

2.1. Equations for the Problem. In order to simulate BBH systems in general relativity, the basic idea is to solve Einstein's equations [9]. We adopt BSSN (Baumgarte-Shapiro-Shibata-Nakamura) [10] formalism which is widely accepted by the numerical relativity community. The BSSN formalism is a conformal-traceless "3 + 1" formulation of the Einstein equations. In this formalism, the space-time is decomposed into three-dimensional spacelike slices, described by three-metric γ_{ij}; its embedding in the four-dimensional space-time is specified by extrinsic curvature K_{ij} and the variables, lapse α, and shift vector β^i, which specify a coordinate system. G is the gravitational constant, and c is the speed of light. In numerical relativity, geometrical units are usually adopted which lead $G = c = 1$. In this paper, we follow the notations of [11]. The related equation description about the problem is as follows:

$$
\begin{aligned}
\partial_t \phi &= \beta^i \phi_{,i} - \frac{1}{6}\alpha K + \frac{1}{6}\beta^i{}_{,i},\\
\partial_t \widetilde{\gamma}_{ij} &= \beta^k \widetilde{\gamma}_{ij,k} - 2\alpha \widetilde{A}_{ij} + 2\widetilde{\gamma}_{k(i}\beta^k{}_{,j)} - \frac{2}{3}\widetilde{\gamma}_{ij}\beta^k{}_{,k},\\
\partial_t K &= \beta^i K_{,i} - D^2\alpha + \alpha\left[\widetilde{A}_{ij}\widetilde{A}^{ij} + \frac{1}{3}K^2\right],\\
\partial_t \widetilde{A}_{ij} &= \beta^k \widetilde{A}_{ij,k} + e^{-4\phi}\left[\alpha R_{ij} - D_i D_j \alpha\right]^{\mathrm{TF}}\\
&\quad + \alpha\left(K\widetilde{A}_{ij} - 2\widetilde{A}_{ik}\widetilde{A}^k_j\right) + 2\widetilde{A}_{k(i}\beta^k{}_{,j)}\\
&\quad - \frac{2}{3}\widetilde{A}_{ij}\beta^k{}_{,k},\\
\partial_t \widetilde{\Gamma}^i &= \beta^j \widetilde{\Gamma}^i{}_{,j} - 2\widetilde{A}^{ij}\alpha_{,j}\\
&\quad + 2\alpha\left(\widetilde{\Gamma}^i_{jk}\widetilde{A}^{kj} - \frac{2}{3}\widetilde{\gamma}^{ij}K_{,j} + 6\widetilde{A}^{ij}\phi_{,j}\right) - \widetilde{\Gamma}^j\beta^i{}_{,j}\\
&\quad + \frac{2}{3}\widetilde{\Gamma}^i\beta^j{}_{,j} + \frac{1}{3}\widetilde{\gamma}^{ki}\beta^j_{,jk} + \widetilde{\gamma}^{kj}\beta^i_{,kj}.
\end{aligned}
\tag{1}
$$

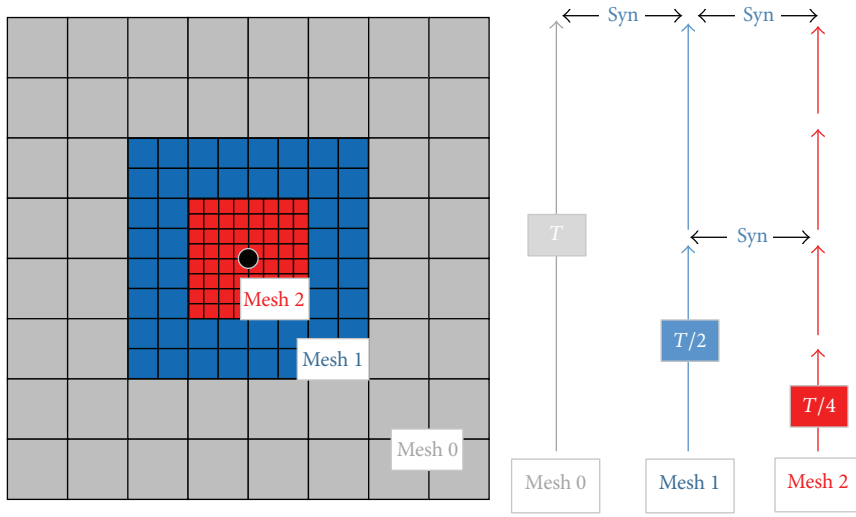


Figure 1: An example to show how different mesh levels are created to cover the black hole and how the different mesh levels are evolved with different time steps. Synchronization is necessary for neighbor mesh levels to exchange data.

Here, ρ, s, s_i, and s_{ij} are source terms which come from matter. For a vacuum space-time, $\rho = s = s_i = s_{ij} = 0$. In the above evolution equations, D_i is the covariant derivative associated with three-metric γ_{ij}, and "TF" indicates the trace-free part of tensor objects. For the time evolution, we use 4th-order Runge-Kutta (for short "RK" in the following parts) method. Spatial derivatives are computed using 4th-order finite differencing. The advection terms are approximated with an upwind finite differencing stencil, and other derivative terms are approximated with a centered stencil. More details regarding the equations and implementations can be found in [12, 13].

2.2. Mesh Refinement Method. Mesh refinement is a technique which allows a simulation to spend more time on the parts of the domain which are more interesting. The Berger-Oliger mesh refinement (or AMR) algorithm [8] is widely used in BBH simulations. We use Fixed Mesh Refinement (FMR) in this work instead of AMR because of the following two facts: (1) FMR will not introduce more computation than AMR in BBH simulations with the same accuracy; (2) FMR is easy to achieve load balancing and it is very critical to improve the performance in large scale parallel computing. The subcycling technology is also introduced because it can further reduce the total simulation steps significantly. For our FMR method with subcycling, the closer to the black hole, the finer the mesh, and the smaller the time step (typical in half), the more the evolution steps, but all the meshes at different levels will have the same grid points.

Figure 1 is an example of our FMR method with subcycling. For simplicity, we only show three mesh levels. They are the coarsest Mesh 0, the refined Mesh 1, and the finest Mesh 2. The finer mesh level will be closer to the black hole, but the coarser mesh level will cover larger region. All the mesh levels will have the same number of grid points. When Mesh 0 executes one simulation step with time step T, Mesh 1 will need to execute two simulation steps to catch up with Mesh 0 because its time step ($T/2$) is half of Mesh 0 and so on for the following finer mesh levels. After each simulation step, the mesh should synchronize with its neighbor mesh levels to exchange the boundary data or get the updated data covered by the refined mesh level.

Our FMR method with subcycling can achieve enough accuracy with significantly reduced total simulation steps because it only simulates the BBH evolution procedure with higher space and time resolution (more space points and smaller time step) when it is closer to the black hole area. In the following sections, we will introduce how we can accelerate its simulation performance on large scale supercomputers.

3. A Scalable Parallel Method

We first explain the pros and cons of the existing SPM parallel method. Then, the basic idea and the algorithm design of our new parallel are given to show how it can overcome the weaknesses of SPM by exploring more parallelism to improve the performance of BBH simulations.

3.1. Analysis of the Existing Method. The existing method divides each mesh level into submeshes and executes these submeshes of the same mesh level in parallel, but the calculation in different meshes will be executed in sequential order. This is the existing Submesh based Parallel Method (SPM).

Algorithm 1 is the pseudocode of the SPM; line (4) shows that when the total mesh levels is more than 1, it will call the *recursive_step* procedure to execute all the other mesh levels. The SPM is a very simple and natural way to simulate the BBH evolution. At the same time, it can achieve high parallel efficiency when only small or even moderate number of parallel tasks are employed (the computation-to-communication ratio is high because the size of submeshes can be big enough for limited number of parallel tasks).

```
(1)  procedure SPM_Evolution
(2)      Input: meshes and boundary conditions
(3)      evolve one step on current level
(4)      if (total_level > 1) call recursive_step(1)
(5)      analyze the results
(6)      regrid if needed
(7)      Output: values on all the mesh grid points
(8)  end procedure
(9)  procedure recursive_step(level)
(10)     for (i = 0; i < 2; i + +) do
(11)         evolve one step on current level
(12)         if (level < total_level − 1) then
(13)             call recursive_step(level + 1)
(14)         end if
(15)         exchange data with related processes
(16)     end for
(17) end procedure
```

Algorithm 1: Submesh based Parallel Method.

When larger scale supercomputers are available and we hope to further improve the BBH simulation performance with more computing resources, SPM often cannot achieve better performance on the latest supercomputers because it can only explore limited parallelism. SPM cannot achieve good parallel efficiency when the size of the submesh is very small. The halo zone is necessary for each submesh to exchange data with its neighbor submeshes and the overhead of communication to fill halo zone will increase quickly when the size of submesh is very small. So the number of parallel submeshes will be limited and this is the essential bottleneck which limits the SPM to achieve better scalability.

Figure 2(a) shows how different mesh levels in SMP method are evolved one by one in order. The green blocks are the calculation steps of different mesh level. The SPM will execute the Runge-Kutta calculation steps from RK1 to RK7 in sequential order even though some of them can be parallelized.

Another weakness of SPM is that when we employ GPU to accelerate the computationally intensive part of each parallel process, the computation time will be significantly reduced but the communication time will not change. This will lead to the quick decrease in parallel efficiency. The SPM can really employ GPU to improve the performance of each single parallel process, but it prevents the application from scaling onto more computing resources as well.

So the SPM only performs well in small or mediate scale parallel computing. It cannot scale to larger scale computing resources. When GPU computing resources are employed, the absolute performance can be improved at first but its scalability will become worse because of the lower computation-to-communication ratio. Another weakness of the SPM is that it cannot work together with GPU well. This is why we must design new scalable parallel algorithm for the challenging BBH simulations on larger scale supercomputers.

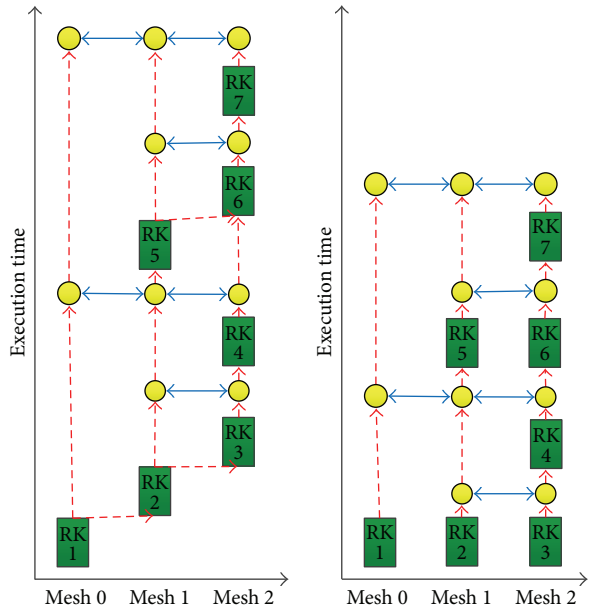
3.2. Mesh Based Parallel Method. The basic idea of our new parallel method is exploring the high level parallelism among all the mesh levels first. This means that we will remove all the control dependence among different mesh levels introduced by the recursive execution of SPM and allow all the meshes to evolve in parallel if only the necessary data is ready. Then, we will explore the low level parallelism in each mesh level if more parallelism is needed. We call this parallel method as Mesh based Parallel Method (MPM).

Figure 2(b) shows how the different Runge-Kutta calculations in Figure 2(a) can be executed in parallel way. For one big physical step T, Mesh 0, Mesh 1, and Mesh 2 have to execute Runge-Kutta calculation {RK1}, {RK2, RK5}, and {RK3, RK4, RK6, RK7}, respectively, based on our FMR method. All the evolution steps of the same mesh level must be executed in sequence because of the data dependence, but the evolution steps from different mesh levels can be executed in parallel. This is very different from the SPM. So our MPM will execute the steps as follows: parallel executing calculation {RK1, RK2, RK3} $\rightarrow$ syn $\rightarrow$ parallel executing calculation {RK4} $\rightarrow$ syn $\rightarrow$ parallel executing calculation {RK5, RK6} $\rightarrow$ syn $\rightarrow$ parallel executing calculation {RK7} $\rightarrow$ syn. But SPM has to execute calculation from RK1 to RK7 one by one in sequential order as follows: parallel executing calculation {RK1} $\rightarrow$ syn $\rightarrow$ parallel executing calculation {RK2} $\rightarrow$ syn $\rightarrow \cdots \rightarrow$ parallel executing calculation {RK7} $\rightarrow$ syn. In this way, the MPM can save about half the simulation time by executing more meshes in parallel.

Although the intermesh level parallel method was really employed alone in some package without subcycling technology, such as in [14], the inter- and intramesh level parallel methods together have never been used in packages/applications with subcycling technology. The subcycling technology can significantly reduce the total simulation steps without losing simulation accuracy, but it will make the parallel execution among different mesh levels very difficult because the communication and synchronization features among different mesh levels are very different from the features in the same mesh level.

To solve this problem, we design a general and highly efficient mechanism to implement both inter- and intramesh level communication. In our MPI [15] code implementation, all the data needed by the same MPI process will be collected first, then packed into one buffer, and sent to the corresponding process with nonblocking MPI communication function. At the same time, the corresponding MPI process will receive and unpack the data to get the halo zone data or prolongation/restriction data. In this way, we can combine the two communication patterns into one, reduce the total number of MPI operations, and allow MPI communication to overlap with computation.

Another problem is that, for large scale parallel computing, the unbalanced load can significantly harm the parallel effect. Our test also shows that the mesh partition method can greatly affect the performance of BBH simulations. So we design a new three-dimensional (3D) mesh partitioning algorithm to achieve two objects: load balancing and less communication.



(a) SPM executes different mesh levels one by one

(b) MPM executes different mesh levels in parallel

Figure 2: An example to show the difference between SPM and MPM. The red line is control flow and blue line is communication flow. The green blocks are the calculation steps with Runge-Kutta (RK) method and the yellow cycles are the communication operations between neighbor mesh levels.

```
(1)  procedure Mesh_Partition (Mesh, Procs, MinMesh)
(2)      Input: Mesh, Procs, MinMesh
(3)      tmp[] = 1
(4)      while (tmp[0] × tmp[1] × tmp[2] < Procs) do
(5)          select d to maximize ⌈Mesh[d]/tmp[d]⌉
(6)          if (⌈Mesh[d]/(tmp[d] + 1)⌉ < MinMesh[d]) break
(7)          tmp[d] + +
(8)      end while
(9)      for (d = 0; d ≤ 2; d + +) do
(10)         SubMesh[d] = ⌈Mesh[d]/tmp[d]⌉
(11)     end for
(12)     Output: SubMesh
(13) end procedure
```

Algorithm 2: 3D mesh partition algorithm.

Algorithm 2 is our method on how to divide a three-dimensional *Mesh* with given number of parallel processes *Procs* and minimum submesh size *MinMesh* as input parameters. The partition result *SubMesh* will be returned as the output result. The basic idea of our method is that the size of submeshes should be close to each other to keep load balancing. At the same time, we will try to make the shape of the submesh as cubic as possible to reduce the size of halo zone (so less data will be transferred). From line (3) to line (7), we calculate the maximum number of processors which will be used to divide the given *Mesh* in each dimension. The total number of processors cannot be larger than the given value *Procs* and the submesh size in each dimension cannot be smaller than *MinMesh*. Line (4) shows that we will divide the dimension with the largest submesh size first, so the shape of submesh cannot be far away from cube. Finally, we can get the size of the submesh which can meet all requirements (from line (8) to line (10)). It is easy to check that our 3D mesh partitioning algorithm can really find submeshes whose sizes are close to each other and whose shapes are close to cube enough.

After the mesh at different levels has been partitioned, Algorithm 3 provides the SPMD (Single Program Multiple Data) pseudocode of our MPM. In line (3), we first calculate the total number of evolution steps of the current mesh level. Then, from line (4) to line (9), all the evolution steps of the current level are executed. After each evolution step, line (6) shows that the related processes have to exchange data

```
(1)  procedure MPM_Evolution (Mylevel)
(2)      Input: meshes and boundary conditions
(3)      NSteps = 2^Mylevel
(4)      for (k = 0; k < NSteps; k++) do
(5)          evolve one step on current level
(6)          exchange data with related processes
(7)          analyze the results
(8)          regrid if needed
(9)      end for
(10)     Output: values on all the mesh grid points
(11) end procedure
```

Algorithm 3: Mesh based Parallel Method.

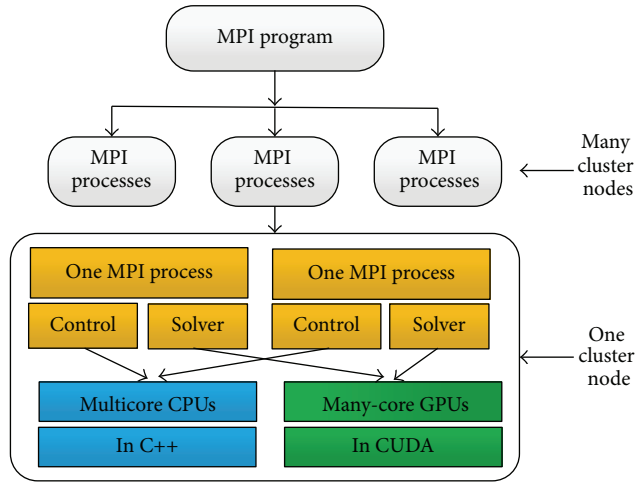


Figure 3: The software structure of our GPU enabled BBH simulation with proposed MPM.

with each other. It will include two kinds of communication patterns: (1) updating the halo zone area; (2) prolongation/restriction with neighbor mesh level. We will analyze the result after each simulation step (line (7)). If the black hole changed its position, we need to regenerate the mesh (line (8)).

4. Experimental Results

In this section, we first introduce the BBH simulation software which has integrated our MPM algorithm. Then, we describe the configuration of our hardware platforms and the default BBH simulation setup in our experiments. Next, we evaluate our GPU implementation to show that its result is correct and can match with the CPU result well. Finally, we show how our novel MPM can improve the performance of BBH simulations with or without GPU acceleration. Two special scheduling mechanisms are given to show that our MPM algorithm can work together with them to improve the performance of BBH simulations under two practical scenarios: (1) enough CPU resources are available, but no GPU resources are available; (2) only limited GPUs resources are available.

4.1. Simulation Software. All of the methods presented in this paper have been implemented in a practical BBH simulation software AMSS-NCKU [12, 16]. Figure 3 shows the structure of the AMSS-NCKU software system. It is an MPI program and its function can be divided into two parts, controler and solver. The control part is written in C++ and the solver is in CUDA. For a multicore cluster node, it may have one or multiple CPUs/GPUs. The user can decide how many MPI processes can be allocated for each cluster nodes. Multiple MPI processes in one node can share one GPU (we export *CRAY_CUDA_MPS* = 1 to enable multiple MPI processes on a single node to share one GPU on Blue Waters) or run exclusively on multiple GPUs. We redesign the mesh partitioning algorithm to enable the simulation control flow and the MPI communication mechanism to implement our MPM. At the same time, the new mesh partitioning algorithm can achieve load balancing and less communication cost. Nonblocking MPI communication is employed to overlap the computation and communication so better performance can be achieved. The solver of AMSS-NCKU is the most computationally intensive part and it has hundreds of Fortran functions. We port the hundreds of Fortran functions to GPU in CUDA and then a great effort is taken to optimize the large amount of CUDA codes on GPU to achieve better performance. The total source codes of AMSS-NCKU are more than 100 K lines.

4.2. Experimental Setup

4.2.1. Hardware Platforms. Most the experimental results provided in this paper are from Blue Waters supercomputer (https://bluewaters.ncsa.illinois.edu). Blue Waters is a Cray XE6/XK7 system consisting of more than 22,500 XE6 compute nodes (each containing two AMD Interlagos 16-core processors) augmented by more than 4200 XK7 compute nodes (each containing one AMD Interlagos 16-core processor and one NVIDIA GK110 "Kepler" accelerator) in a single Gemini interconnection fabric.

At the same time, we also test our program on other large GPU clusters, such as Mole-8.5 (http://www.top500.org/system/176899) which contains 362 compute nodes; each node has two Intel® Xeon® E5520 processors and 6 NVIDIA 2050 GPUs and SuperMike-II (https://www.cct.lsu.edu/supermike-ii) which contains 440 compute nodes; each node has two Intel Sandy Bridge Xeon® E5-2670 octacore processors and fifty nodes are equipped with two NVIDIA Tesla M2090 GPUs. Both of them use InfiniBand network.

4.2.2. Scheduling Methods. We employ three scheduling mechanisms to evaluate the different parallel methods. The first is Default Scheduling. For CPU jobs, it means that one MPI process is assigned to one CPU core so that one CPU computing node with K cores will be assigned to K MPI processes. For GPU jobs, it means that one MPI process is assigned to one GPU computing node so that one computing node with multiple CPU cores will have only one MPI process. For CPU jobs, the resource contention among different

MPI processes cannot be avoided and this will lead to performance reduction using Default Scheduling. For GPU jobs, the Default Scheduling will lead to low CPU and GPU utilization because it cannot fully explore the capability of all hardware resources. So we design and implement another two special scheduling mechanisms. They are Exclusive Scheduling for CPU jobs and GPU-Shared Scheduling for GPU jobs (we export *CRAY_CUDA_MPS* = 1 to enable multiple MPI tasks on a single node to share one GPU on Blue Waters or Titan supercomputer). Exclusive Scheduling means that one MPI process will be assigned to one independent computing node no matter how many cores it has. GPU-shared Scheduling means that more than one MPI process will share one GPU. In practical scenarios, sometimes the system is not equipped with GPU (or GPUs are not available temporarily), but it has many CPU computing nodes. Sometimes the system only has limited GPUs. We implement the Exclusive Scheduling and GPU-Shared Scheduling to do experiments under the two practical scenarios.

4.2.3. Default Application Configuration. The default configuration for all the experiments is as follows. A typical mesh size ($128 \times 128 \times 64$) which is widely used in the current BBH simulations is selected. The total number of mesh levels is 8. The execution time is the accumulated time of 5 big physical time steps.

4.3. Correctness Evaluation. The precision is very important for scientific applications. To evaluate the correctness of our GPU code, we do the following two kinds of evaluation tests. The first is directly comparing the difference of RHS calculation between CPU results and GPU results. Figure 4 shows how we conduct the experiments. We not only use the real data as inputs but also generate randomly hundreds of arrays as different inputs. For the same input, we will execute both the original CPU codes and the corresponding GPU CUDA codes. We develop a tool which can dump all the intermediate results at different position of both CPU and GPU codes. In this way, we can compare the difference between the CPU result and the GPU result. For all the different inputs, the maximal difference of the intermediate results between CPU and GPU is less than 10^{-14}. So the GPU results are very close to the CPU results for different inputs.

The second evaluation is using a typical application for binary black hole collision calculations (see Figure 5). Here, we use two spinless identical black holes head on to each other along y-axis. Initially, these two black holes separate $2.303M$, with M being the total mass of the two black holes. This configuration has been used for code testing by many numerical relativity groups. First, we compare the evolved dynamical variables to fourth-order Runge-Kutta time integration. The difference between CPU results and GPU results is less than 10^{-12} for the whole simulation process. Figure 5(a) gives an example of such comparison for evolved variable $\widetilde{\gamma}_{xy}$ on $z = 0$ plane at time $t = 140M$. These plots are from SuperMike-II.

Gravitational wave form is one of the most important physical quantities for numerical relativity study. In this work, we use Newman-Penrose scalar Ψ_4 to represent gravitational

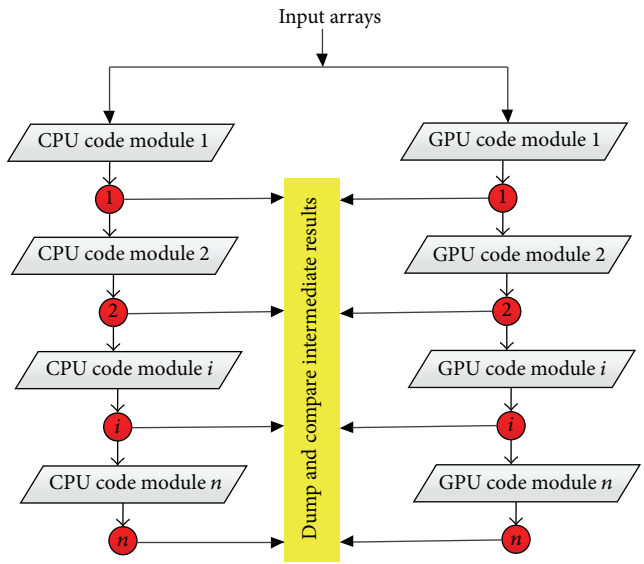


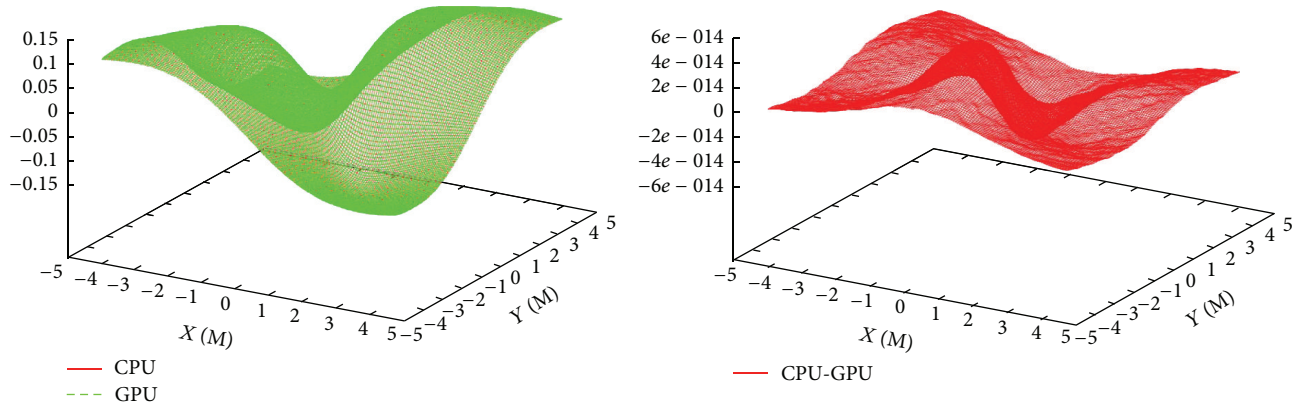
Figure 4: Different kinds of synthetic inputs are generated for the CPU and corresponding GPU codes. We can dump and compare the calculation results of CPU and GPU at different stages. In this way, we can analyze the numerical difference between CPU and GPU calculation for the same input.

wave. This numerical technique is widely used in numerical relativity community. Here, we have compared the gravitational wave form generated by CPU and GPU. The test simulation time is 120 M. In Figure 5(b), we can see that the gravitational waveforms generated from GPU and CPU BBH simulation on Blue Waters, SuperMike-II, and Mole-8.5 are almost identical to each other. All of these tests confirm the correctness of our GPU implementation for the numerical relativity code.

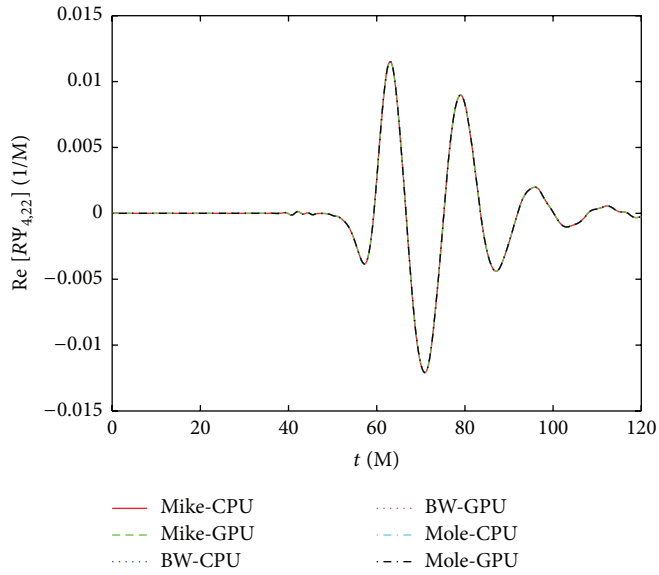
4.4. Performance Improvement with Our Scalable MPM Algorithm. In this section, we will investigate how our MPM algorithm can replace the existing SPM algorithm to achieve better performance with both CPU and GPU codes. The performance of two special scheduling mechanisms for practical scenario can be improved with our MPM algorithm too. Because the performance of sequential code is unacceptable, we use the execution time of 32 MPI processes of CPU-SPM code as the baseline (5718 s) to calculate the speedup. In this section, we will show the performance curve till the best performance is achieved.

4.4.1. Employing MPM Algorithm in CPU Version. Figure 6 shows how our MPM algorithm can use more parallel processes to improve the performance of BBH simulation. The new CPU version MPM algorithm is marked as CPU-MPM and the existing one is marked as CPU-SPM.

Figure 6(a) shows that the performance of CPU-SPM is even better than the performance of CPU-MPM when the number of parallel MPI processes is no more than 512. The reason is that MPM algorithm can explore more parallelism with the cost of some idle computing resources. SPM algorithm cannot scale to more MPI processes, but it



(a) Dynamic variables comparison shows that the CPU results are very close to GPU results



(b) The waveform comparison shows that the CPU results and GPU results from different platforms are very close to each other

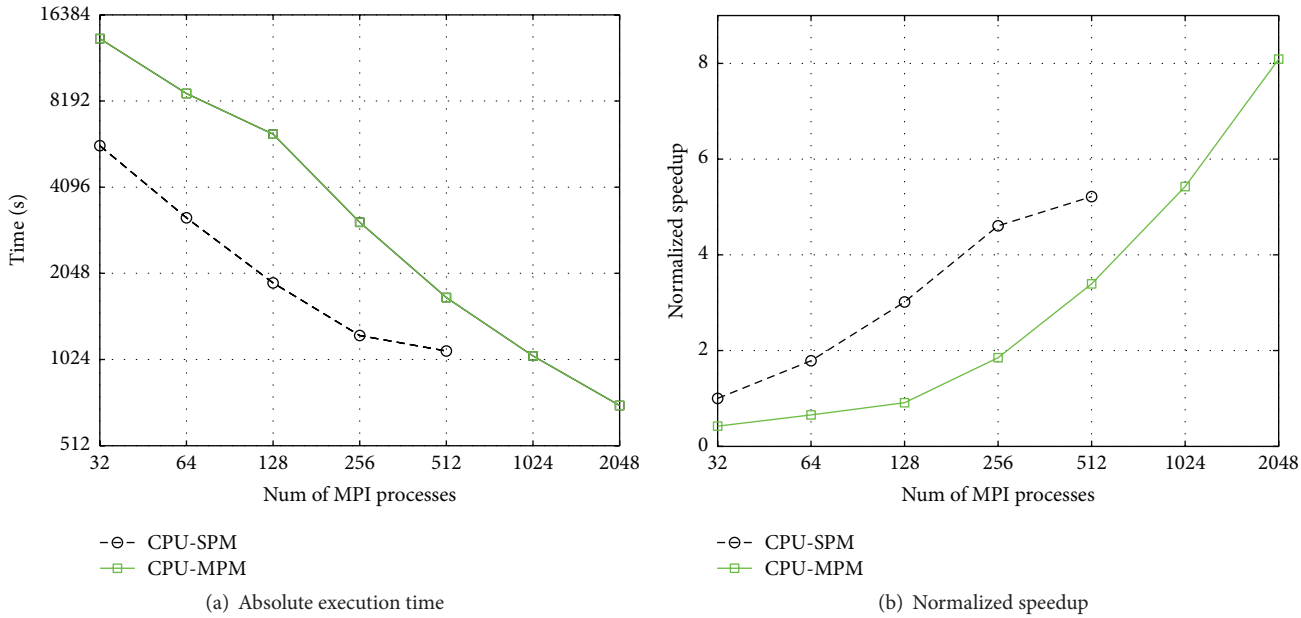
Figure 5: Correctness evaluation.

will keep all the computing resources busy. At first, when the number of MPI processes is small, SPM algorithm can fully take advantage of the given parallel resources and achieve higher performance. Since MPM algorithm will lead some computing resources to be idle when not all the mesh levels can be executed in parallel, the performance of MPM algorithm is lower than SPM. However, when more than 512 MPI processes are used, the parallel efficiency of SPM algorithm will become lower and lower because the total computation for each MPI process will become smaller and smaller. This is why the performance of SPM will become worse when more MPI processes are used. Compared with SPM algorithm, MPM algorithm can use more MPI processes to execute different mesh levels in parallel without significantly decreasing the computation for each MPI process. So the parallel efficiency of MPM is much better than SPM when more and more MPI processes are used. This is why the performance of SPM is better than the performance of MPM at first and finally MPM can scale to larger number of MPI processes to improve the performance of BBH simulation. The best performance of CPU-SMP with 512 MPI processes is 1096 s. The best performance of CPU-MPM with 2048 MPI processes is 707 s.

Figure 6(b) shows that CPU-SPM can achieve about 5x speedup, but CPU-MPM can achieve about 8x speedup compared with the baseline.

4.4.2. Employing MPM Algorithm in GPU Version. After we verify the effect of the MPM algorithm in CPU codes, we port and optimize the CPU codes onto GPU, replace the SPM algorithm in our GPU codes, and get the scalable GPU acceleration version. Figure 7 shows how our MPM algorithm can further improve the performance of our GPU codes. The GPU codes with MPM algorithm and SPM algorithm are marked as GPU-MPM and GPU-SMP, respectively.

Figure 7(a) shows that the performance of all our optimized GPU codes is significantly better than the performance



(a) Absolute execution time

(b) Normalized speedup

Figure 6: The strong scaling results of MPM algorithm and SPM algorithm in the CPU implementations.

of CPU-SPM. At the same time, the performance of GPU-SPM is also better than the performance of GPU-MPM when the number of parallel MPI processes is no more than 256 (the reason is similar to that in the CPU codes). The best GPU-SPM performance is 479 s with 256 MPI processes. Beyond 256 MPI processes, GPU-SPM cannot use more MPI processes to achieve better performance, but GPU-MPM can scale up to 1024 MPI processes (1024 GPUs) to achieve its best performance (203 s). The results show that employing GPU and the scalable MPM algorithm are the two major reasons to achieve the performance improvement.

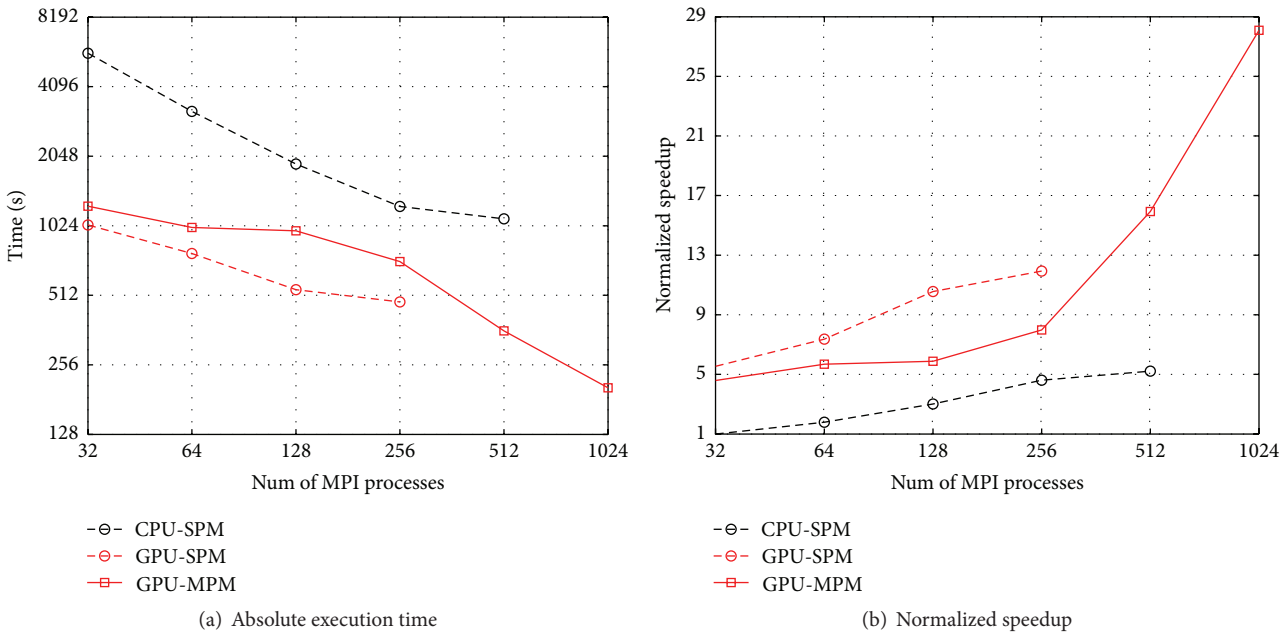
Figure 6(b) shows that GPU-SPM can achieve about 12x speedup but GPU-MPM can achieve about 28x speedup. The MPM algorithm can achieve about $479/203 \approx 2.36$x speedup for GPU codes but only about $1096/707 \approx 1.55$x speedup for CPU codes (see Section 4.4.1). This result shows that our MPM algorithm can work together with GPU codes better to achieve more significant performance improvement.

4.4.3. Employing MPM Algorithm in Two Special Scheduling Mechanisms. In this section, we will show how our MPM algorithm can work together with two special scheduling mechanisms, Exclusive Scheduling and GPU-Shared Scheduling, to improve their performance.

Exclusive Scheduling. Here each XE computing node will be assigned with only one MPI process and the results are shown in Figure 8. Figure 8(a) shows the execution time of the Exclusive Scheduling with the existing SPM algorithm (marked as CPU-SMP-Ex) and our MPM algorithm (marked as CPU-MPM-Ex). CPU-SPM-Ex can scale up to 512 XE computing nodes to achieve the best performance 807 s. CPU-MPM-Ex can even scale up to 2048 XE computing nodes to achieve its best performance 510 s. Exclusive Scheduling can really achieve better performance than the corresponding Default Scheduling for both SPM (1.36x speedup) and MPM (1.39x speedup) algorithm by avoiding resource contention.

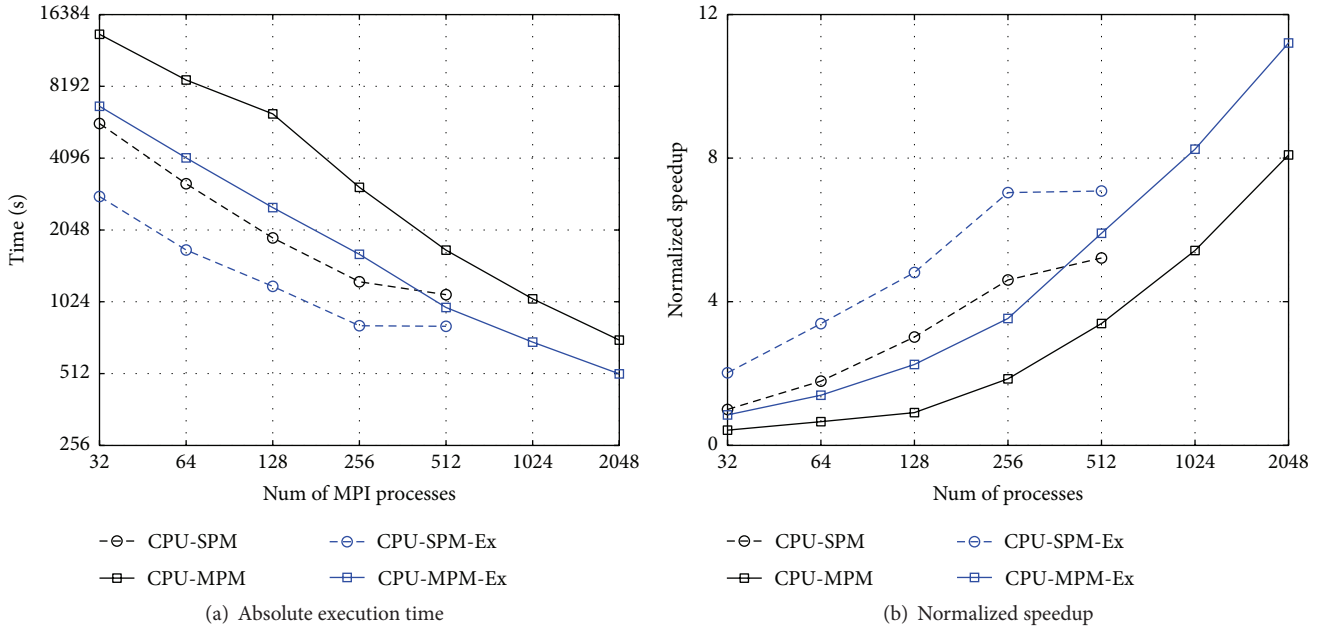
Figure 8(b) shows that CPU-SPM-Ex and CPU-MPM-Ex can achieve about 7x and 11x speedup, respectively. The best performance of CPU-MPM-Ex with 512 MPI processes is very close to the best performance of GPU-SPM with 256 MPI processes (256 XK computing nodes). It shows that, with enough CPU computing resources, our MPM algorithm can achieve about the same performance improvement as the GPU acceleration version with exiting SPM algorithm. Considering the major problem for large scale applications is that they cannot take advantage of more computing resources to improve their performance, MPM + Exclusive Scheduling is a feasible way to improve the simulation performance.

GPU-Shared Scheduling. When only limited GPU resources are available, we can employ GPU-Shared Scheduling to improve the performance of GPU jobs. Figure 9(a) shows the execution time of GPU-Shared Scheduling with the existing SPM algorithm (marked as GPU-SMP-S8) and the MPM algorithm (marked as GPU-MPM-S8) when one GPU (one XK computing node) is shared by 8 MPI processes here. The results show that when GPU-Shared Scheduling works together with the existing SPM algorithm, the performance will be even worse than the CPU codes with Default Scheduling because of its low computation-to-computation ratio. However, when GPU-Shared Scheduling works together with our MPM algorithm, GPU-MPM-S8 can execute 2048 MPI processes on 256 XK computing nodes (256 GPUs) using 476 s. Compared with 717 s of GPU-MPM's performance with 256 MPI processes (they have the same number of XK computing nodes but different number of MPI processes), about $717/476 \approx 1.5$x performance improvement can be achieved.



(a) Absolute execution time (b) Normalized speedup

Figure 7: The strong scaling results of MPM algorithm and SPM algorithm in the GPU implementations.



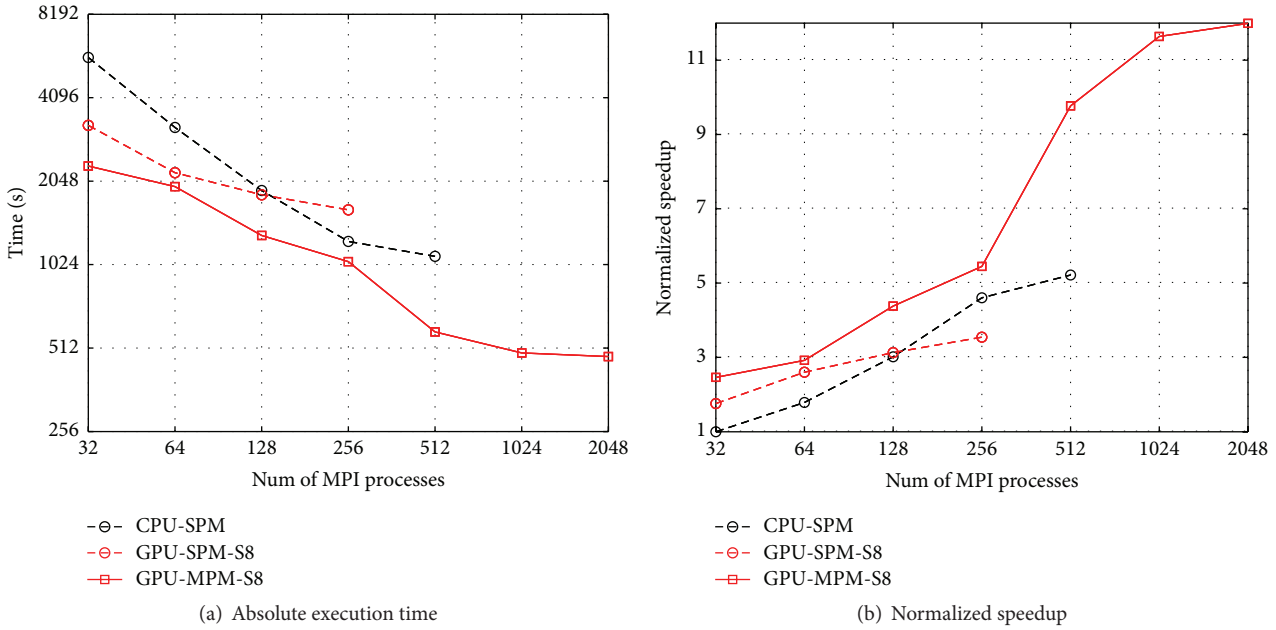
(a) Absolute execution time (b) Normalized speedup

Figure 8: The strong scaling results of MPM algorithm and SPM algorithm with Exclusive Scheduling.

Figure 9(b) shows that the GPU-MPM-S8 version can achieve about 12x speedup with 2048 MPI processes (256 XK computing nodes or 256 GPUs). This is better than the GPU-MPM version's 8x speedup with 256 MPI processes on 256 XK computing nodes. The results show that when the number of available GPUs is limited, our MPM algorithm can work together with GPU-Shared Scheduling to fully explore the capability of powerful GPU to improve the performance BBH simulations.

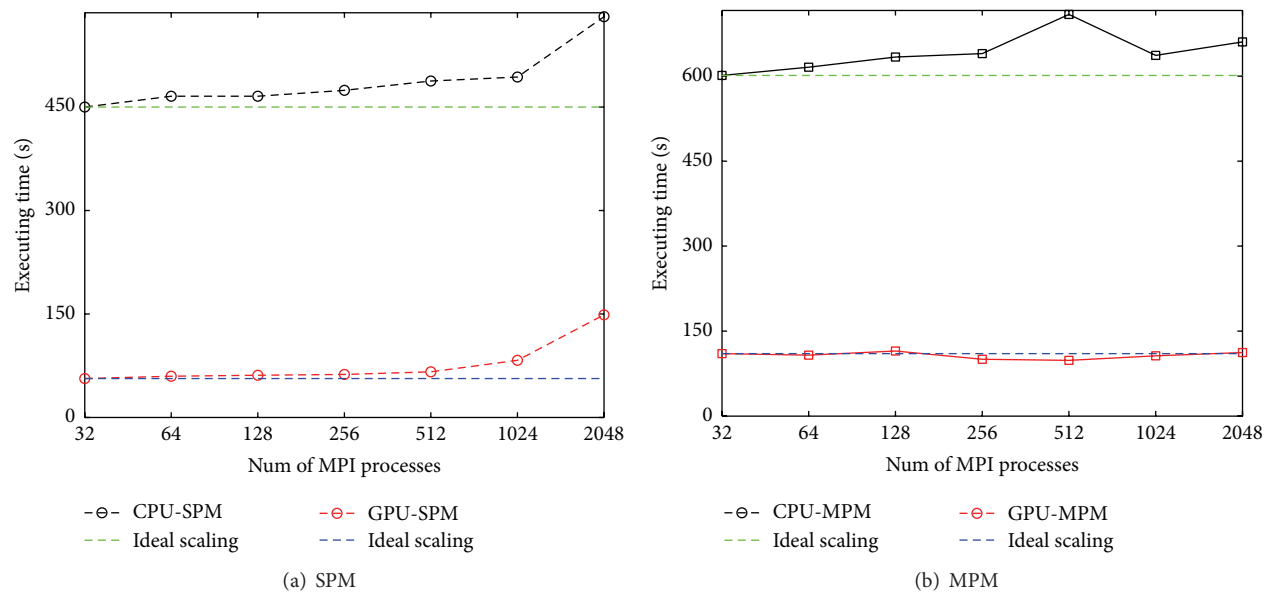
4.5. Weak Scaling of Our MPM Algorithm. To evaluate the weak scalability of our method, we start from mesh size 320 × 320 × 160 and use 32 MPI processes to solve the problem. When we double the number of MPI processes, we also double the mesh size. Only two mesh levels are employed to reduce the total experimental time. The total simulation time is 5 big physical time steps.

Figure 10(a) shows when the number of MPI processes is not large, the weak scaling results of SPM are good. But SPM



(a) Absolute execution time (b) Normalized speedup

Figure 9: The strong scaling results of MPM algorithm and SPM algorithm with GPU-Shared Scheduling.



(a) SPM (b) MPM

Figure 10: The weak scaling results of MPM algorithm and SPM algorithm.

cannot achieve good weak scaling result when more MPI processes are used. Figure 10(b) shows that the execution time of CPU-MPM has very little increase when the number of MPI processes increases from 32 to 2048. For the GPU-MPM curve, it is even flat and very close to the ideal weak scaling line. The good strong scaling result of our MPM is shown in Sections 4.4.1 and 4.4.2. This section shows that MPM especially the GPU-MPM version can achieve almost ideal weak scaling results. So our GPU-MPM can not only reduce significantly the execution time but also solve very large BBH simulation problem with more computing resources efficiently.

5. Related Work

5.1. BBH Simulation and the Related Codes. The first direct detection of gravitational wave [5] shows that the BBH simulation is essential for gravitational wave data analysis and the parameter estimation of the gravitational wave source. The matched filtering technique requires the accurate

gravitational waveforms, which can only be generated by BBH simulation to find the weak gravitional wave signal. The parameter estimation based on BBH simulation can tell us detailed information about the detected gravational wave. However, BBH simulation is very computationally intensive. Large mass ratio BBH simulation which is necessary for future space-based detectors is even challenging because the computational cost is about propotional to the fourth power of the mass ratio. Furthermore, delevoping a practical BBH simulation application is also a very challenging work because of its complexity.

Till now, there are only about 10 independent numerical relativity codes for binary black hole simulations [17] in the world. CCATIE, Einsetein Toolkit, LazEv, Lean, MayaKranc, and UIUC are based on Cactus platform. AMSS-NCKU, BAM, Hahndol, PU, SACRA, SpEC, and GRChombo [18] are based on different independent package. These codes employ different methods to handle two key problems, the numerical accuracy and computing efficiency. Since the computer hardware increases much faster, how to take advantage of the large scale compuer resources to significantly improve the performance of BBH simulation has become a critical problem for all the BBH simulation applications. The early work of Cao et al. 2008 [12] reinvestigation shows that AMSS-NCKU is in the state of art, so we select it as the baseline of our method.

5.2. Mesh Refinement Methods. Dubey et al. [19] give a detailed comparison on six publicly available, active, and existing at least half a decade adaptive mesh refinement (AMR) packages, BoxLib, Cactus, Chombo, Enzo, FLASH, and Uintah. They can be very different in the refine factor, time step ratio, language, and so forth because of the diverse requirements in specific domain areas and general functionality. Subcycling is often used to reduce the total amount of calculation when the grid points of coarse mesh level is not large. For structured AMR with subcycling, executing the submeshes of the same mesh level in parallel is a very genral and widely adopted idea to improve the performance, such as the method provided by Diachin et al. [20]. However, the SPM liked parallel method cannot scale to large scale computing resources because the communication among the submeshes will be too costly when the size of submesh is too small. Nonsubcycling will lead to an easy way to execute all the different mesh levels in parallel, such as PARAMESH [14]. However, this parallel method is too expensive if the number of grid points in finer mesh level is large. The two parallel methods have been developed independently and we have not seen the work to combine them together because they are used in very different scenarios. In this paper, we propose our MPM, which can explore the parallelism among the different mesh levels and the submeshes of one mesh level in BBH simulation. The advantage of the proposed MPM lies in that it can take advantage of the large scale powerful supercomputers to improve its performance.

5.3. GPU Optimization Methods. In general relativity studies, GPU has been used in solving the Teukolsky equation [21] and post-Newtonian evolution [22]. For numerically solving 3D Einstein's equations, some groups are trying to apply GPU to numerical relativity code [23]. Zink [24] implements a general relativistic evolution CUDA code on GPU and more than 20x speedup can be achieved. However, it can only run on one GPU with single precision. So it can not handle large scale problem and single precision is often unacceptable for many applications. There are also ongoing efforts from Cactus group. There are many cases to employ GPU to improve the performance of different applications [25–29]. Even though, for large applications, porting their CPU codes to GPU is an error-prone task, at the same time it is hard and tedious. So some compilers, such as [30, 31], are developed to transform CPU codes into GPU CUDA codes automatically. Lutz et al. work [32] can transform stencil computation onto multi-GPU automatically. Yang et al. [33] optimize the memory access of existing CUDA code. Considering thousands of Fortran codes will be rewritten in CUDA, we borrow those ideas and develop some compiler-like tool to make some regular code porting from CPU to GPU efficiently and automatically. This can really reduce the total work and avoid errors.

Optimization methods are very important for practical applications to achieve significant performance improvement on GPUs. There are excellent general suggestions and principals on GPU code optimization. From the view of hardware, NVIDIA's suggestions [34] are improving the hardware utilization and the throughput of both memory and instruction. For different applications, the performance bottleneck maybe different. Ryoo et al. work has great impact on GPU optimization [35] and they give three basic principles based on their work on optimizing different applications on GPU: improving the percentage of floating point instructions, reducing/overlapping the global memory access latency, and releasing the pressure of global memory bandwidth. Study on GPU speedup [36] indicates that the magic number of GPU speedup often has no meaning if the CPU codes are not optimized or parallelized. Only the optimized results can show the real difference between CPU and GPU codes. This is why we compare the best performance of CPU and GPU results.

6. Conclusion and Future Work

How to take advantage of the latest supercomputer to significantly reduce the execution time of large scale applications such as BBH simulations is a real challenging problem. In this paper, we propose a novel scalable parallel method (MPM) which can explore both the inter- and intramesh parallelism to gain more parallelism. So our MPM can take advantage of more computer resources efficiently to improve the performance of BBH simulations. The proposed MPM can improve the performance of both CPU and GPU BBH simulation codes. At the same time, it can work together with two practical scenarios to improve the real application's performance.

Our experimental results show that the existing SPM can only achieve 5x speedup for CPU codes and 12x speedup for GPU codes. But our MPM can achieve 8x speedup for CPU codes and 28x speedup for GPU codes. This results show

that it is necessary for us to take advantage of both hardware resources and the well designed algorithm to achieve better performance.

We are working together with the Cactus group and trying to integrate our method into their package to the whole community. Currently, our parallel mesh refinement method supports the maximum parallelism among all the meshes, but some computing resources will be idle during some simulation stages. In the next step, we will improve our method which can support the average parallelism of the meshes to improve the resource utilization. Furthermore, with smaller computing resources providing the same average parallelism, the communication overhead can obviously be reduced.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

This research is supported in part by National Natural Science Foundation of China (nos. 61440057, 61272087, 61363019, and 61073008), Beijing Natural Science Foundation (nos. 4082016 and 4122039), the Sci-Tech Interdisciplinary Innovation and Cooperation Team Program of the Chinese Academy of Sciences, the Specialized Research Fund for State Key Laboratories, and NSF Awards ACI-1265434 and ACI-1339745. Parts of experiments are conducted on HPC System Mole-8.5 constructed by Institute of Process Engineering, Chinese Academy of Sciences.

References

[1] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, Top500 supercomputing sites, 2015.

[2] J. D. Owens, D. Luebke, N. Govindaraju et al., "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[3] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010.

[4] R. Cowen, "Telescope captures view of gravitational waves," *Nature*, vol. 507, no. 7492, pp. 281–283, 2014.

[5] B. Abbott, R. Abbott, T. Abbott et al., "Observation of gravitational waves from a binary black hole merger," *Physical Review Letters*, vol. 116, no. 6, Article ID 061102, 2016.

[6] D. Blair, L. Ju, C. Zhao et al., "Gravitational wave astronomy: the current status," *Science China: Physics, Mechanics and Astronomy*, vol. 58, no. 12, Article ID 120402, pp. 1–41, 2015.

[7] P. Amaro-Seoane, S. Aoudia, S. Babak et al., "Low-frequency gravitational-wave science with eLISA/NGO," *Classical and Quantum Gravity*, vol. 29, no. 12, Article ID 124016, 2012.

[8] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of Computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.

[9] A. Einstein, "The foundation of the general theory of relativity," *Annalen der Physik*, vol. 354, no. 7, pp. 769–922, 1916.

[10] M. Campanelli, C. O. Lousto, P. Marronetti, and Y. Zlochower, "Accurate evolutions of orbiting black-hole binaries without excision," *Physical Review Letters*, vol. 96, no. 11, Article ID 111101, 2006.

[11] T. W. Baumgarte and S. L. Shapiro, *Numerical Relativity: Solving Einstein's Equations on the Computer*, Cambridge University Press, 2010.

[12] Z. Cao, H.-J. Yo, and J.-P. Yu, "Reinvestigation of moving punctured black holes with a new code," *Physical Review D—Particles, Fields, Gravitation and Cosmology*, vol. 78, no. 12, Article ID 124011, 2008.

[13] D. Hilditch, S. Bernuzzi, M. Thierfelder, Z. Cao, W. Tichy, and B. Brügmann, "Compact binary evolutions with the z4c formulation," *Physical Review D—Particles, Fields, Gravitation and Cosmology*, vol. 88, no. 8, Article ID 084057, 2013.

[14] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer, "PARAMESH: a parallel adaptive mesh refinement community toolkit," *Computer Physics Communications*, vol. 126, no. 3, pp. 330–354, 2000.

[15] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, vol. 1, MIT Press, 1999.

[16] P. Galaviz, B. Brügmann, and Z. Cao, "Numerical evolution of multiple black holes with accurate initial data," *Physical Review D—Particles, Fields, Gravitation and Cosmology*, vol. 82, no. 2, Article ID 024005, 2010.

[17] B. Aylott, J. G. Baker, W. D. Boggs et al., "Testing gravitational-wave searches with numerical relativity waveforms: results from the first numerical injection analysis (ninja) project," *Classical and Quantum Gravity*, vol. 26, no. 16, Article ID 165008, 2009.

[18] K. Clough, P. Figueras, H. Finkel, M. Kunesch, E. A. Lim, and S. Tunyasuvunakool, "Grchombo: numerical relativity with adaptive mesh refinement," *Classical and Quantum Gravity*, vol. 32, no. 24, Article ID 245011, 2015.

[19] A. Dubey, A. Almgren, J. Bell et al., "A survey of high level frameworks in block-structured adaptive mesh refinement packages," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3217–3227, 2014.

[20] L. F. Diachin, R. Hornung, P. Plassmann, and A. Wissink, *Parallel Adaptive Mesh Refinement*, vol. 143 of *Parallel Processing for Scientific Computing*, 2006.

[21] G. Khanna and J. McKennon, "Numerical modeling of gravitational wave sources accelerated by OpenCL," *Computer Physics Communications*, vol. 181, no. 9, pp. 1605–1611, 2010.

[22] F. Herrmann, J. Silberholz, M. Bellone, G. Guerberoff, and M. Tiglio, "Integrating post-Newtonian equations on graphics processing units," *Classical and Quantum Gravity*, vol. 27, no. 3, Article ID 032001, 2010.

[23] B. Brügmann, "A pseudospectral matrix method for time-dependent tensor fields on a spherical shell," *Journal of Computational Physics*, vol. 235, pp. 216–240, 2013.

[24] B. Zink, "A general relativistic evolution code on cuda architectures," in *Proceedings of the LCI Conference on High-Performance Clustered Computing*, Champaign, Ill, USA, 2008.

[25] Y. Liu, Z. Du, S. K. Chung, S. Hooper, D. Blair, and L. Wen, "Gpu-accelerated low-latency real-time searches for gravitational waves from compact binary coalescence," *Classical and Quantum Gravity*, vol. 29, no. 23, Article ID 235018, 2012.

[26] J. Michalakes and M. Vachharajani, "Gpu acceleration of numerical weather prediction," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS '08)*, pp. 1–7, Miami, Fla, USA, April 2008.

[27] R. S. Oliveira, B. M. Rocha, D. Burgarelli, W. Meira Jr., and R. W. dos Santos, "Simulations of cardiac electrophysiology combining gpu and adaptive mesh refinement algorithms," in *Bioinformatics and Biomedical Engineering*, pp. 322–334, Springer, 2016.

[28] H. Y. Schive, Y. C. Tsai, and T. Chiueh, "Gamer: a graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics," *The Astrophysical Journal Supplement Series*, vol. 186, no. 2, 2010.

[29] M. Schwarz and M. Stamminger, "Fast gpu-based adaptive tessellation with cuda," in *Computer Graphics Forum*, vol. 28, pp. 365–374, Wiley Online Library, 2009.

[30] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, article 54, 2013.

[31] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A script-based autotuning compiler system to generate high-performance CUDA code," *Transactions on Architecture and Code Optimization*, vol. 9, no. 4, article 31, 2013.

[32] T. Lutz, C. Fensch, and M. Cole, "PARTANS: an autotuning framework for stencil computation on multi-GPU systems," *Transactions on Architecture and Code Optimization*, vol. 9, no. 4, article 59, 2013.

[33] Y. Yang, P. Xiang, J. Kong, M. Mantor, and H. Zhou, "A Unified optimizing compiler framework for different GPGPU architectures," *Transactions on Architecture and Code Optimization*, vol. 9, no. 2, article 9, 2012.

[34] NVIDIA, Cuda c programming guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[35] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-M. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pp. 73–82, ACM, Salt Lake City, Utah, USA, February 2008.

[36] V. W. Lee, C. Kim, J. Chhugani et al., "Debunking the 100X GPU vs. CPU Myth: An evaluation of throughput computing on CPU and GPU," in *Proceedings of the 37th International Symposium on Computer Architecture (ISCA 2010)*, pp. 451–460, Saint-Malo, France, June 2010.