The Gyrokinetic Particle Simulation of Fusion Plasmas on Tianhe-2 Supercomputer

Endong Wang¹, Shaohua Wu¹, Qing Zhang¹, Jun Liu¹, Wenlu Zhang², Zhihong Lin³,⁴, Yutong Lu⁵,⁶, Yunfei Du⁵,⁶, Xiaoqian Zhu⁷

¹ State Key Laboratory of High-end Server & Storage Technology, Inspur, Jinan, China
² Institute of Physics, Chinese Academy of Science, Beijing, China
³ Department of Physics and Astronomy, University of California, Irvine, California, USA
⁴ Fusion Simulation Center, Peking University, Beijing, China
⁵ School of Computer Science, National University of Defense Technology, Changsha, China
⁶ National Supercomputer Center in Guangzhou, Sun Yat-sen University, Guang Zhou, China
⁷ Academy of Ocean Science and Engineering, National University of Defense Technology, Changsha, China

Abstract—We present novel optimizations of the fusion plasmas simulation code, GTC on Tianhe-2 supercomputer. The simulation exhibits excellent weak scalability up to 3072 31S1P Xeon Phi co-processors. An unprecedented up to 5.8x performance improvement is achieved for the GTC on Tianhe-2. An efficient particle exchanging algorithm is developed that simplifies the original iterative scheme to a direct implementation, which leads to a 7.9x performance improvement in terms of MPI communications on 1024 nodes of Tianhe-2. A customized particle sorting algorithm is presented that delivers a 2.0x performance improvement on the co-processor for the kernel relating to the particle computing. A smart offload algorithm that minimizes the data exchange between host and co-processor is introduced. Other optimizations like the loop fusion and vectorization are also presented.

Keywords—Fusion plasmas simulation; GTC; scalability; Tianhe-2; Xeon Phi

I. INTRODUCTION

Research on turbulent transport in fusion plasmas is important to provide physics foundations for the predictive extrapolations of plasma confinement properties from present-generation tokamak experiments to larger magnetic fusion devices such as the International Tokamak Experimental Reactor (ITER) [1], the largest device currently under construction in France [2], which is believed to the goal of magnetically confined fusion that can provide clean and sustainable supply of energy on earth [3]. High fidelity predictive numerical experiments plays an important role in understanding the physics foundations. However, gyrokinetic simulations are extremely compute-intensive since distribution functions in the five-dimensional phase space need to be accurately resolved [4].

A. State of the Art

The Gyrokinetic Toroidal Code (GTC) is a global particle-in-cell (PIC) application developed to address kinetic turbulence issues in fusion plasmas [5]. GTC is one of the most widely used fusion codes in the world fusion community. GTC is the key production code for the multi-institutional U.S. Department of Energy (DOE) Scientific Discovery through Advanced Computing (SciDAC) project, Gyrokinetic Simulation of Energetic Particle Turbulence and Transport (GSEP) Center, and the National Special Research Program of China for ITER. GTC has had a long and productive history of pushing forward the frontiers of plasma physics and advanced computing. GTC has published extensively in the most prestigious physics journals with over 5,000 citations and more papers in Science and PRL than any other PIC codes in the world fusion program [6-9].

The main GTC code has exhibited a well-established track record in supercomputing. GTC is the first fusion code to reach the tera-scale in 2001 on Seaborg computer at NERSC [10] and the peta-scale in 2008 on Jaguar computer at ORNL in production simulations [11]. GTC was also one of the first production code to run on GPU on both Tianhe-1A and Titan [12]. Recently, GTC was selected by Oak Ridge Leadership Computing Facility (OLCF) Center for Accelerated Application Readiness (CAAR) program for optimizing major application codes across all DOE-supported science research portfolio for GPU-based computers.

A multi-level parallelization using MPI/OpenMP has been designed in GTC to scale up to millions of cores and to take advantage of the memory hierarchy of current generation parallel computers [13, 14]. A “companion” version of the electromagnetic GTC code, the electrostatic GTC-P code is a modern, highly portable GTC code now operational on the top 7 supercomputers worldwide [15]. The GTC-P has been ported and optimized on different supercomputers such as IBM Blue Gene/P (BG/P) at Argonne National Laboratory, IBM Blue Gene/Q (BG/Q) of Mira at ALCF, Sequia at LLNL, the Cray XT4 at Lawrence Berkeley National Laboratory, et al [15, 16]. The scalability up to 131,072 BG/P and 32,768 XT4 cores were attained with as little as 512MB memory per core by
incorporating a new radial decomposition method, developed by Wang et al., that features a dramatic increase in scalability for the grid work and decrease in the memory footprint of each core [16]. Later, Kamesh et al. made further optimizations of the GTC-P, such as multi-level particle and grid decompositions, particle binning, and memory-centric optimizations. As a result, they delivered 1.22x, 1.35x, 1.77x, and 1.34x performance improvement on BG/P, the Cray XE-6, and Intel Cluster, and a Fermi Cluster respectively [17]. Then the GTC-P was weak scaling to full 98,304 nodes of Sequoia, and Intel Cluster, and a Fermi Cluster respectively [17]. The performance was increased from nearly 50,000 millions of particles per second per core to more than 100,000 MPST on 98,304 Sequoia nodes. GTC was also weak scaling to 32768 Fujitsu K nodes, and about 50,000 MPST was achieved [18].

Most of the above nontrivial performance achievements were running on leading homogeneous systems. On the top CPU+GPU heterogeneous system such as TH-1A in Tianjin, an efficient GPU parallel algorithm of GTC was developed [12]. The weak scaling to 3072 nodes of TH-1A was obtained with 2x-3x overall speedup comparing NVIDIA M2050 GPUs to Intel Xeon X5670 CPUs [12]. More recently, the GTC was ported to another top heterogeneous system, Titan, a Cray XK7 system with 299,008 CPUs and 18,688 K20 NVIDIA GPUs. The porting and optimizing of GTC on Tianhe-2, one of the fastest supercomputer in the world, is the next step. Tianhe-2 was launched in 2013. It has topped the TOP500 list six times [19]. Tianhe-2 is an Intel Xeon Phi accelerated system with 16,000 computing nodes. Each node consists of 2 CPUs and 3 Xeon Phis accelerators. The pioneer work of GTC on Xeon Phi was first reported by the 2013 Dongarra report on Top500 Supercomputers [20], followed by the porting of GTC-P on Stampede, a 6400 Xeon Phi accelerated cluster at TACC, in 2015 [21]. The Xeon Phi offers three programming modes: offload, native and symmetric. The native and symmetric mode were tested on Stampede [4]. On this system, the GTC-P weak scaling to 1024 nodes was demonstrated with a medium overall performance compared to other CPU only and CPU + GPU systems such as Titan, Blue Waters, Mira et al. [4], however the modification of the code is quite few, only about 6% lines of code changed. It is up to 150% lines of code changed on GPU system. The symmetric mode boosted the performance by 1.2x with respect to the CPU, although there was marked degradation in scalability for the number of nodes greater than 256, which is believed to be caused by the load imbalance and the network performance [4].

B. Major Achievements in This Paper

We have been porting and optimizing the production version of GTC on Tianhe-2 with the offload mode that provides much finer tuning of the code than native and symmetric modes. The weak scaling to 1024 nodes of Tianhe-2 is achieved with a linear scalability. The weak scaling takes advantage of full 3 Xeon Phis per node, thus delivers a linear scalability to 3072 co-processors. The performance improvement is up to 5.8x compared to the original GTC code running on Tianhe-2 CPUs.

All the previous optimizations of the GTC use a shift kernel with an iterative MPI communications to shift particles. This iterative method starts many MPI communications that are actually redundant. In order to fix this issue, we develop a direct MPI communication algorithm that simplifies the iteration to only once for particles exchanging. As a result, it reduces the time on MPI communications by removing the redundancy relating to the iterative MPI’s send and receive operations.

The ordering of the particles impacts the performance of the GTC simulation. A new parallel particle sorting algorithm targeting Multi- and Many-core system is developed in this paper, which was tested on Xeon CPUs and Xeon Phi accelerators in Tianhe-2. The performance of the kernel relating to the particle computing are improved by 2x on Xeon Phi co-processor.

C. Brief Introduction of the Paper

The sections below are organized as follows. Section 2 briefly overviews GTC. In section 3, we present the key concepts and innovations in details. We analyze the performance results on Tianhe-2 supercomputer in section 4, and we conclude in section 5.

II. THE GYROKINETIC TOROIDAL CODE

GTC solves the five-dimensional gyrokinetic Vlasov Poisson equations in full, global torus geometry to describe highly nonlinear plasma turbulence in magnetically-confined fusion experimental facilities. The PIC approach is incorporated in the GTC code to reduce the computation complexity from O(N^2) to O(N + MlogM), where N is the number of particles, and M is the number of grid points [22, 18]. The system geometry simulated in GTC is shown in Fig. 1. It is a torus with an externally-imposed magnetic field [17]. Particles interact with each other through a self-consisted field that is solved on the mesh in Eulerian coordinates. The field value is then used to force particles to move [23]. Depending on the physics being studied, the number of particles exceeds the number of grid points by 100 to 1000 [24]. At each time step, GTC executes nine kernels, which are listed below.

Charge: The GTC simulates two types of particles: iron and electron. So there are two charge kernels: Chargei for the iron, and Chargee for the electron. The charge of each particle (iron/electron) is distributed to the grid points surrounding the particle. A fraction of the particle’s charge is interpolated to the grids according to the distance from the particle position [22, 18].

Poisson/Field/Smooth: These three kernels perform the grid related operations. The gyrokinetic Poisson equation is solved in the Poisson kernel to obtain the electrostatic potential at each grid point. The field kernel computes the electronic field on the grid. The charge density and potential vectors are smoothed in the smooth kernel. Simulations often employ high particle densities, so the grid operations consume less than the particles computing.

Push (Pushi/Pushe): There are two Push kernels: Pushi for the ions, and Pushe for the electrons. The functionality of
these two kernels are quite similar. The force on each particle (iron/electron) is interpolated from the grid points that surround the particle. Then the particles are forced to move by solving the equations of motion with a second order low-storage Runge-Kutta scheme. This step requires scanning through particle arrays and indirect grid access in memory. As the grid locations surrounding each particle varies due to particle exchanges among MPI processes, the memory accesses of the field information attached to the grid points may have a giant leap in the field array, which increases the cache miss rate, and decreases performance.

Shift: There are two Shift kernels: Shifti for the iron, and Shifte for the electron, respectively. The functionality of these two kernels are identical: Sending and receiving particles among MPI processes. An iterative process is used in GTC. The iron and electrons that need to be sent to its neighboring MPI processes are picked and sent. For each iteration, it consists of five steps: 1) Packing the particles that need to be sent to its neighboring MPI processor; 2) Checking whether to exit the iteration. If the iteration has been performed more than once, an MPI reduction is conducted to compute the summation of the particles that need to be sent among all the MPI processes. If the summation is zero, it means there is no particle need to be sent for all the MPI processes. Then the iteration will be broke, and the shift kernel will return. 3) Sending and receiving particles by calling MPI’s sendrecv function; 4) Filling the holes left by sent particles with the remaining particles that reside at the end of the particles array; 5) Copying the received particles to the end of the particle arrays. Because particles are forced to move only one subdomain along the toroidal direction within one iteration, the shift kernel has to iterate $m_{toroidal}/2$ times if the particle need to be sent to the furthermost MPI process with the rank of $m_{toroidal}/2 + m_{toroidal}/2$, where $m_{toroidal}$ is the number of subdomains partitioned along the toroidal direction, and $m_{toroidal}$ is the rank of the current MPI process. As the periodic boundary condition is applied along the toroidal coordinate, a numerator of 2 is taken into account. The forth step, filling the holes is sequential in the GTC, which causes performance degradation in highly threaded environments, such as GPU and Xeon Phi.

The GTC code features multiple levels of parallelism. First, an one-dimensional domain decomposition is implemented in the toroidal dimension using MPI. In order to further increase MPI parallelism, a second level of decomposition, a particle decomposition, is introduced. Particles are divided, but fields are shared between MPI processes in the particle decomposition. Field solvers are parallelized using all MPI processes in the particle decomposition. In this way, each process owns a fraction of the total particles in the domain as well as a private copy of the field information attached on the local toroidal grid. The third level of parallelism is an intra-node shared memory partitioning (via OpenMP) of both particle and grid related work.

There are four loops in the GTC. The pushe and shifte kernels are in the innermost loop that accounts for up to 90% runtime of the GTC simulation. Our optimization focuses on the pushe and shifte kernels.

Fig. 1. GTC mesh for a single magnetic surface, showing equilibrium magnetic field lines (black), and values of perturbed electrostatic potential in linear regime (colors).

III. KEY CONCEPTS AND INNOVATIONS

The following design choices were made so as to address the network traffic caused by redundant MPI communications, the high cache miss rates caused by the irregular access of the field information, the inefficient floating point operations due to the low utilization of the vector units on CPU and Xeon Phi co-processors, and the PCI-E traffic caused by the large amount of data transmission between hosts and accelerators.

A. Innovations in the MPI Communication

In GTC, most of the MPI communications occur in the shift kernel. An iterative method that exchanges the particles among the MPI processes has been adopted for decades. Because the particles are forced to move across the subdomains along the toroidal coordination, the distance that the particles go may cover several subdomains. In the extreme case, the iteration have to conduct $m_{toroidal}/2$ times to send the particles from the origin to the destination.

MPI communications can be imbalanced in the iterative algorithm. The MPI processes that have particles went across multiple subdomains have to perform more iterations than the processes that only need to send the particles to their neighboring processes. This algorithm also performs additional computations and memory copies for the MPI processes that send particles across multiple subdomains. As a result, the runtime of each MPI process in the shift kernel may be highly imbalanced.

The imbalance can be clearly seen in Fig. 2. In this figure, the time spent on the MPI Reduction function was counted and plotted for all the 48 MPI processes that were executed on 16 nodes of Tianhe-2. In order to remove the influence of the pushe kernel, we inserted an MPI barrier before executing the shifte kernel. From Fig. 2 we can see that the time of each MPI process varies greatly. Nearly 21% of the MPI processes spent less than 10 seconds on this global MPI communication, while 31% of the MPI processes spent nearly 50 seconds. This imbalance will get worse when $m_{toroidal}$ and the number of particles increases. In order to address this issue, we develop a new algorithm that exchanges particles from origin to destination directly, therefore the original iterative MPI communication algorithm is simplified to once.
Algorithm 1 Pack and exchange particles among MPI processes. ‘zelectron’ is the particle array containing ‘me’ particles. ‘z_0’ and ‘z_1’ are locations for the left and right boundary.

1: msend(0:mtoroidal-1)=0
2: isum=0
3: for m=1,me do
4:  z=zelectron(3,m)
5:  if (z < z_0 or z>z_1) then
6:    id=floor(z/(2π)*mtoroidal)
7:    msend(id)=msend(id)+1
8:    isum=isum+1
9:    sendparticle(1,isum)=id
10:   sendparticle(2,isum)=m
11:  end if
12: end for
13: msendoffset(0)=0
14: for id=1,mtoroidal-1 do
15:  msendoffset(id)=msendoffset(id-1)+msend(id-1)
16: end for
17: msendcount = msendoffset(mtoroidal)-msend(mtoroidal-1)
18: msendoffset_1 = msendoffset
19: for m = 1, msendcount do
20:   id = sendparticle(1,m)
21:  msendoffset(id)=msendoffset(id)+1
22:  msendbuf(:,msendoffset(id)) = zelectron(:,sendparticle(2,m))
23: end for
24: for id=1, mtoroidal-1 do
25:  idest=mod(myrank_toroidal+id,mtoroidal)
26:  send msend(idest) to MPI process ranked idest.
27: end for
28: mrecvoffset(0)=0
29: for id=1,mtoroidal-1 do
30:  mrecvoffset(id)=mrecvoffset(id-1)+mrecv(id-1)
31: end for
32: mrecvcount=mrecvoffset(mtoroidal)-mrecv(mtoroidal-1)
33: for id=0,mtoroidal-1 do
34:  if(msend(id) > 0) then
35:    send msendbuf(:,msendoffset(id)+1:msendoffset(id)+msend(id)) to the MPI process ranked id
36:  end if
37: end for
38: for id=0,mtoroidal-1 do
39:  if(mrecv(id) > 0) then
40:    receive mrecvbuf(:,mrecvoffset(id)+1:mrecvoffset(id)+mrecv(id)) from the MPI process ranked id
41:  end if
42: end for

Fig. 2. Executing time of the MPI_Allreduce function in shift kernel.

Algorithm 2 Filling the holes, and append the receiving particles to the particle array.

1: isum=0
2: for m=me-msendcount+1, me do
3:  z=zelectron(3,m)
4:  if (z < z_0 or z>z_1) then
5:    isum=isum+1
6:    zelectron(:,sendparticle(2,isum))= zelectron(:,m)
7:  end if
8: end for
9: me=me-msendcount
10: for m=1,mrecvcount do
11:    zelectron(:,me+m)=mrecvbuf(:,m)
12: end for
13: me=me+mrecvcount

The algorithm includes four major steps as shown in Algorithm 1 and Algorithm 2. The first step, from line 1 to line 23 in Algorithm 1, packs the sent particles into a send buffer. In this step, exclusive scan is adopted to have the sent particles gathered continuously in the send buffer.

The second step, from line 24 to line 43 in the Algorithm 1, exchanges the particles among MPI processes. The number of the particles that need to be sent or received are communicated, then an exclusive scan is conducted to organize the locations that received particles need to be put in the receiving buffer. Then the particles are exchanged among the MPI processes. In contrast to the original iterative MPI communications, this new algorithm send and receive the particles from the origin to the destination directly.

The third step, from line 1 to line 8 in Algorithm 2, fill the holes that left by the sent particles. This step was executed concurrently with the second step so as to improve the performance. This is implemented by using two OpenMP threads, and each algorithm is controlled by one thread. The last step, from line 9 to line 13 in the Algorithm 2, adds the receive buffer to the end of the particle array.

In order to port the algorithm 1 and 2 to multi- and many core CPUs and Xeon Phi accelerators, the first, third, and the
final steps are parallelized by using exclusive scan in the multithread environment.

Fig. 3 presents the performance of the shifte kernel before and after optimization. These results were obtained by using the same datasets as the weak scaling study in section IV. We can see that the runtime of original shifte kernel increases with the number of nodes, while the it is nearly the same for the optimized kernel. The performance was greatly improved by the optimization. The speedup is 7.9x on 1024 nodes of Tianhe-2.

B. Innovations in the Particle Sorting

The ordering of the particles impacts the performance of the pushe and charge kernels. Bowers developed a sequential particle sorting algorithm for the PIC method that only traverses the input particle array twice [25]. His algorithm takes advantage of the fact there are only a relative small number of grids compared to the number of particles, but he didn’t test the algorithm with the real world software like GTC.

Marin developed an adaptive sorting interval mechanism that is able to automatically choose the sorting interval in the GTC simulation [26]. As a results, The execution time of GTC was improved by more than 20% on the Cray XT4 supercomputer.

Kamesh developed their own sorting algorithm that bins particles based on the radial coordinate, then bins them on the poloidal coordinate to deliver up to 1x performance improvement of the charge kernel on Nehalem-EX CPUs, although there was a marked degradation of the performance on Fermi GPU [27]. But Kamesh’s later research showed that the performance was improved on Kepler GPU with the same sorting algorithm [23]. In contrast to the Kamesh’s sorting method, we customize the algorithm developed by Bowers to GTC, and devise a parallel version in the multithread environment for the multi- and many-core systems.

Algorithm 3 describes the parallel sorting method in detail. To explain the algorithm: the first parallel region from line 2 to line 11 counts the number of particles in each grid cell. Each thread counts its own particles. The second parallel region from line 12 to line 20 conducts the exclusive scan to merge the count of each thread into a global array. The loop from line 22 to line 26 conducts an additional exclusive scan on this global array to get the offset that determines how sorted particle array should be organized. After this loop, ‘P(i)’ indicates the location in sorted particle array, where unsorted particle at the grid cell ‘i’ should go. In the third parallel region, the particle array is copied into an temporary array according to the global offset ‘P’ and their private offset ‘Poffset_iam’. At the end of the algorithm, the particles in the temporary array are copied back to the particle array.

This algorithm is well tailored to GTC as the grid cell location has been computed in the push kernel, thus it will be reused in the shift kernel. There are three types of the grid cell: radial, poloidal, and the inner or outer flux surface along the radial or poloidal coordinates. We perform sorting along these different grids. The sorting along the lower poloidal grid on inner flux surface delivered the best performance.

The runtime of the sorting algorithm is nearly the same as the pushe kernel, so the sorting interval should be chosen carefully to get the best performance. In this paper, the sorting was conducted after every eight executions of pushe kernel. Table I lists the performance improvements of the pushe kernel by using the sorting algorithm. As the particles are partitioned between hosts and co-processors, we list the wall clock runtime for both, and the unit is second. The speedup is about 2.0x for the kernel executing on Xeon Phi co-processor, while it is about 1.6x for the CPU.

C. Innovations in the Kernel Optimization

The pushe and shifte kernels are offloaded to Xeon Phis. In this offload mode, the MPI processes are only executed on CPUs, so the exchanged particle arrays need to be copied to co-processor before the kernel execution.

Then the processed particles need to be copied back to host before they are exchanged again by the MPI. The Xeon Phi co-processors are connected to the hosts by the PCI-E slots. So the data transfer rates between host and co-processors are limited by the PCI-E bandwidth.

The time spent exchanging particles is nearly equal to the runtime of pushe kernel, which greatly reduces the performance. In order to address this issue, a smart data exchanging algorithm is devised for GTC. The algorithm includes three steps: the first, the particle arrays are decomposed between CPU and Xeon Phi. Then the sub-arrays are copied to Xeon Phi before the main loop. These sub-arrays are hold by Xeon Phi through the whole execution of the GTC.

The second, in the main loop, the CPU and Xeon Phi conduct particle computations concurrently. Only the particles that need to be exchanged are copied back to CPUs. The particles transmission between hosts and co-processors is conducted in the shifte kernel before the MPI communications. The particles received from other MPI processes are decomposed between CPUs and Xeon Phis. Then only a part of the particles are offloaded to Xeon Phis. Because these offloaded particles are about 5% of the whole particles, it saves a lot of PCI-E bandwidth for data transfer.
Algorithm 3. Particle sorting. ‘zelectron’ is the particle array to sort containing me particles. ‘num_threads’ is the number of threads. ‘mgrid’ is the number of the grid cells.

1: P_iam = 0
2: Begin parallel
3:     iam = rank of the current thread
4:     delm=me/num_threads
5:     mbeg(iam) = 1+min(iam,i)*(delm+1)+max(0,(iam-i))*delm
6:     mend(iam) = mbeg+delm+(min((iam+1),i)/(iam+1))-1
7:     for m = mbeg(iam), mend(iam) do
8:         i = compute the grid cell location for particle m
9:         P_iam(i,iam) = P_iam(i,iam)+1
10: end for
11: end parallel
12: Begin parallel
13:     for i = 1,mgrid do
14:         Poffset_iam(i,0) = 0
15:         for iam = 1,num_threads-1 do
16:             Poffset_iam(i,iam) = Poffset_iam(i,iam-1)+P_iam(i,iam-1)
17:         end for
18:         P(i) = Poffset_iam(i, num_threads-1)+P_iam(i, num_threads-1)
19:     end for
20: end parallel
21: k = 0
22: for i = 1,mgrid do
23:     j = P(i)
24:     P(i) = k
25:     k = k+j
26: end for
27: Begin parallel
28:     iam = rank of the current thread
29:     for m = mbeg(iam), mend(iam) do
30:         i = compute the grid cell location for particle m
31:         Poffset_iam(i,iam) = Poffset_iam(i,iam)+1
32:         j = P(i)+Poffset_iam(i,iam)
33:         zsort(:,j) = zelectron(:,m)
34:     end for
35: end parallel
36: zelectron = zsort

TABLE I. THE EFFECTS OF PARTICLE SORTING ON THE RUNTIME OF THE PUSHE KERNEL

<table>
<thead>
<tr>
<th>Events</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executed on accelerator, particles are sorted</td>
<td>32</td>
</tr>
<tr>
<td>Executed on host, particles are sorted</td>
<td>128.0</td>
</tr>
<tr>
<td>Executed on host, particles are unsorted</td>
<td>203.8</td>
</tr>
</tbody>
</table>

The third, the particles are copied back to CPUs before the charge kernel. As charge kernel is invoked outside the innermost loop, the cost of this data copy is small. With this method, the performance of the data transfer is greatly improved compared to the direct data transmission between CPUs and Xeon Phi co-processors.

The vector unit of Xeon Phi is at least two times larger than modern CPU. Effectively utilizing the vector units of Xeon Phi is of critical importance to deliver good performance. The loops in the pushe kernel are merged into one loop to increase the computation density. Then the loop is internally partitioned into small blocks. The size of each block is tuned to fit into the L1 cache. These small blocks are also used to split the computations engaged in the indirect memory accesses. The compiler is unable to completely vectorize the unsplit loop because of the data dependencies within the indirect access pattern. By splitting these computations in different blocks, and using the temporary arrays to bridge these blocks, the whole loop can be vectorized.

Fig. 4 presents the performance of the pushe kernel before and after optimization. The results are obtained by using the same datasets as in the weak scaling study given in section IV. We can see that the performance of the pushe kernel is improved by 6.4x after optimization.

IV. SCALABILITY AND PERFORMANCE OF GTC ON TIANHE-2

In this section, we demonstrate the performance of GTC on the Tianhe-2 supercomputer using weak scaling and strong scaling of the particle sizes.

Tianhe-2 is a Xeon Phi accelerated system at the National Supercomputer Center in Guangzhou, China. Tianhe-2 consists of 16,000 computing nodes. Each node has two Xeon E5 2692 CPUs (12 cores, 2.2 GHz) and three Xeon Phi 31S1P co-processors (57 cores, 1.1 GHz) [28]. All the computing nodes are connected by a customized fat-tree topology.

The original GTC was running only with the CPUs, while the optimized code takes advantage of all the CPUs and Xeon Phis. We use one MPI process to control one Xeon Phi co-processor, therefore there are three MPI processes to be launched per node to fully utilize the Xeon Phis.

For each MPI process, there will be eight threads to be launched on CPU, and 220 threads on a Xeon Phi. As a result, there will be 24 threads for two CPUs and 660 threads for three Xeon Phi co-processors per node.

The particles are divided between CPU and co-processor. The ratio of particles on CPU to the particles on co-processor is finely tuned. The best performance is achieved with a ratio of 0.36. The PCI-E bandwidth for data transfer between host and co-processor is 8 GB/s. The achieved data transfer rate is 6.6 GB/s.

The floating point operations are counted manually for the GTC. All the FLOPS numbers measured in this paper refer to single precision.
Fig. 4. Runtime of the pushe kernel before and after optimization. The “Original” indicates runtime of the original pushe kernel, while the “Optimized” indicates runtime of the optimized pushe kernel.

Fig. 5. Weak scaling of the GTC from 32 to 1024 Tianhe-2 nodes. The number of particles are increased proportionally to the number of nodes.

TABLE II. SPEEDUP ON TIANHE-2 FOR THE WEAK SCALING

<table>
<thead>
<tr>
<th>Kernels</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32</td>
</tr>
<tr>
<td>pushe</td>
<td>6.3</td>
</tr>
<tr>
<td>shifte</td>
<td>4.1</td>
</tr>
<tr>
<td>GTC</td>
<td>5.3</td>
</tr>
</tbody>
</table>

A. Weak Scalability

A weak scalability study of GTC is presented in this section. Fig. 5 summaries the performance of all weak scaling runs. In contrast to the previous scaling studies that were carried out on Stampede [4], the optimized code scales linearly up to 1024 nodes without any marked degradation in scalability from 256 to 1024 nodes that was occurred on Stampede. It should be noted that there is only one Xeon Phi co-processor per Stampede node, so the previous scaling studies on Stampede only takes up to 1024 Xeon Phi co-processors. In this paper, we run GTC on the 3072 Xeon Phi co-processors on the 1024 nodes of Tianhe-2.

Table II lists the speedup of pushe, shifte and GTC for the weak scaling. The speedup of shifte is increased with the nodes, that means the new particle exchanging algorithm is well suitable for the GTC. We can predict the performance will be better on even larger nodes. The speedup of pushe and GTC drop slightly from 512 to 1024. Even in this case, 6.0x, and 5.5x performance improvements are attained for pushe and GTC, respectively.

B. Strong Scalability

A strong scalability study of GTC is presented in this section. The performance results are given in Fig. 6. In contrast to the weak scaling, the strong scaling tends to become a flat line on larger nodes. For the optimized code, the performance improvements from 512 to 1024 nodes are small, that is caused by the computation density of the Xeon Phi decreases with the nodes increase. As the computation density decreases, the idle time of the co-processors increase.

Table III lists the speedup of pushe, shifte and GTC for the strong scaling. The speedup tends to decrease as the number of the nodes increase. Because most of the computation are executed on the accelerators, the decreasing of the computation density make the performance of the accelerators get worse. However, a 2.2x performance improvement of GTC is attained on 1024 nodes.

V. CONCLUSION

We successfully optimized the GTC for the Tianhe-2 supercomputer, a Xeon Phi accelerated system that has ranked 1st on the TOP500 list six times. In contrast to the previous work carried out on Stampede [4], another Xeon Phi accelerated supercomputer, the weak scalability of the optimized GTC scales linearly up to 3072 Xeon Phi co-processors without any marked degradation, which is remarkable well in terms of scalability.

We devise a novel MPI communication algorithm for the GTC, which simplifies the original iterative scheme to a direct implementation. A 7.9x improvement in terms of MPI communication is achieved on 1024 nodes. A parallel particle sorting algorithm is developed that leads to 1.6x performance improvements for the kernel executed on the CPU, while it is 2x for the kernel executed on the Xeon Phi co-processor. All the loops relating to the particle computations are vectorized to effectively utilizing the vector units of CPUs and co-processors. These optimizations incorporated with a smart offload scheme that targets minimization of the data transmission between host and co-processors leads to an unprecedented up to 5.8x performance improvement on Tianhe-2 supercomputer.

Fig. 6. Strong scaling of the GTC from 32 to 1024 Tianhe-2 nodes. The number of particles are fixed with the number of nodes increased.
GTC is one of the most important physical-based turbulent simulations of tokamak fusion devices. The predictions made by the numerical simulations play an important role in extrapolating plasma confinement properties from present-generation tokamak experiments to larger magnetic fusion devices such as the ITER. The faster computing capability of the GTC can contribute to simulate much larger devices with reasonable time on the top supercomputers.

Our scalability study and performance results show that the GTC can be efficient and scalable for even larger simulations on Tianhe-2 or other Xeon Phi accelerated supercomputers. We expect that the GTC can readily utilize the power of the emerging exa-scale computer, which is needed for predictive simulation of fusion experiments such as ITER.

ACKNOWLEDGMENT

This research used the resources of Tianhe-2 supercomputer at the National Supercomputer Center in Guangzhou (NSCC-GZ), China. We wish to thank Depei Qian, Xiangfei Meng, and Yang Zhao for their exceptional support. Furthermore, the quick and competent help from Ying Zhong and Wen wan at NSCC-GZ were highly appreciated.

REFERENCES


TABLE III. SPEEDUP ON TIANHE-2 FOR THE STRONG SCALING

<table>
<thead>
<tr>
<th>Kernels</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushe</td>
<td>6.4</td>
<td>6.2</td>
<td>5.9</td>
<td>5.3</td>
<td>4.2</td>
<td>2.8</td>
</tr>
<tr>
<td>shifte</td>
<td>4.1</td>
<td>3.8</td>
<td>2.7</td>
<td>2.8</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>GTC</td>
<td>5.3</td>
<td>5.0</td>
<td>4.2</td>
<td>3.6</td>
<td>2.5</td>
<td>2.2</td>
</tr>
</tbody>
</table>