Scientific Programming 18 (2010) 139–151
DOI 10.3233/SPR-2010-0311
IOS Press

Overlapping communication with computation using OpenMP tasks on the GTS magnetic fusion code

Robert Preissl [a,*], Alice Koniges [a], Stephan Ethier [b], Weixing Wang [b] and Nathan Wichmann [c]
[a] *Lawrence Berkeley National Laboratory, Berkeley, CA, USA*
[b] *Princeton Plasma Physics Laboratory, Princeton, NJ, USA*
[c] *Cray Inc., St. Paul, MN, USA*

Abstract. Application codes in a variety of areas are being updated for performance on the latest architectures. In this paper we examine an application, which comes from magnetic fusion for performance acceleration with a particular emphasis on methods that are applicable for many/multicore and future architectural designs. We take an important magnetic fusion particle code that already includes several levels of parallelism including hybrid MPI combined with OpenMP. We study how to include new advanced hybrid models, which extend the applicability of OpenMP tasks and exploit multi-threaded MPI support to overlap communication and computation. Experiments carried out on Cray XT4 and XT5 machines resulting in a speed-up of up to 35% of the investigated GTS particle shifter kernel show the benefits and applicability of this approach.

Keywords: GTS, OpenMP task, communication overlap, hybrid MPI and OpenMP computing

1. Introduction – GTS, a massively parallel magnetic fusion application

The application chosen for this study is the Gyrokinetic Tokamak Simulation (GTS) code [13], which is a global 3D Particle-In-Cell (PIC) code to study the microturbulence and associated transport in magnetically confined fusion plasmas of tokamak toroidal devices. Microturbulence is a very complex, nonlinear phenomenon that is generally believed to play a key role in determining the efficiency and instabilities of magnetic confinement of fusion-grade plasmas [4]. GTS has been developed in Fortran 90 (with a small fraction coded in C) and parallelized using MPI and OpenMP with highly optimized serial and parallel sections; i.e., SSE instructions or other forms of vectorization provided by modern processors. GTS simulation runs have been conducted simulating a laboratory-size tokamak of 0.932 m major radius and 0.334 m minor radius confining a total of 2.1 billion particles using a domain decomposition of two million grid points on Cray's XT4 and XT5 supercomputers.

In plasma physics applications, the PIC approach amounts to following the trajectories of charged particles in self-consistent electromagnetic fields. The computation of the charge density at each grid point arising from neighboring particles is called the *scatter* phase. Prior to the calculation of the forces on each particle from the electric potential (*gather* phase) – we solve *Poisson's equation* for computing the field potential, which only needs to be solved on a 2D poloidal plane.[1] This information is then used for moving the particles in time according to the equations of motion (*push* phase), which is the fourth step of the algorithm.

2. The GTS parallel model

The *parallel model of GTS has three independent levels*: (1) GTS uses a one-dimensional (1D) domain decomposition in the toroidal direction (the long way around the torus). This is the original scheme of expressing parallelism using the Message Passing Inter-

*Corresponding author: Robert Preissl, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720, USA. Tel.: +1 510 486 6421; Fax: +1 510 486 4316; E-mail: rpreissl@lbl.gov.

[1] Fast particle motion along the magnetic field lines in the toroidal direction leads to a quasi-2D structure in the electrostatic potential.

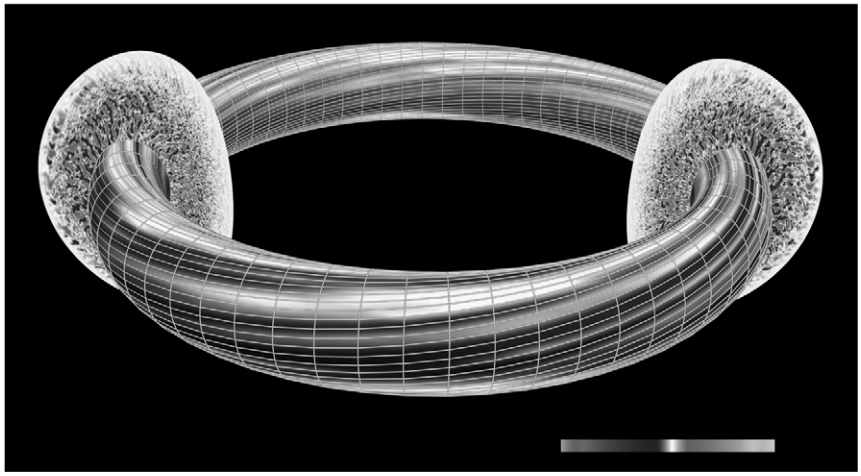

Fig. 1. GTS' toroidal domain decomposition with magnetic field lines and density fluctuations.

face (MPI) to perform communication between the toroidal domains. Particles can move from one domain to another while they travel around the torus – which adds another, a fifth, step to our PIC algorithm, the *shift* phase. This phase is of major interest in the upcoming sections. Only nearest-neighbor communication in a circular fashion (using MPI_Sendrecv functionality) is used to move the particles between the toroidal domains. It is worth mentioning that the toroidal decomposition is limited to 64 or 128 planes, which is due to the long-wavelength physics that we are studying. More toroidal domains would introduce waves of shorter wavelengths in the system, which would be dampened by a physical collisionless damping process known as Landau damping; i.e., leaving the results unchanged [4]. Using higher toroidal resolution only introduces more communication with no added benefit. (2) Within each toroidal domain, we divide the particles between several MPI processes, and each process keeps a copy of the local grid,[2] requiring the processes within a domain to sum their contribution to the total charge density on the grid at the end of the charge deposition or *scatter* step (using MPI_Allreduce functionality). The grid work (for the most part, the field solve) is performed redundantly on each of these MPI processes in the domain and only the particle-related work is fully divided between the processes. Consequently, GTS uses two different MPI communicators; i.e., an *intradomain communicator*, which links the processes within a common toroidal domain of the 1D domain decomposition and a *toroidal communicator* comprising all the MPI processes with the same intradomain rank in a ringlike fashion. (3) Adding OpenMP compiler directives to heavily used (nested) loop regions in the code exploits the shared memory capabilities of many of today's HPC systems equipped with multicore CPUs. Although of limited scalability due to the single-threaded sections between OpenMP parallel loops and also due to NUMA effects arising from the shared memory regions, this method allows GTS to run in a hybrid MPI/OpenMP mode. Addressing the challenges and benefits involved with hybrid MPI/OpenMP computing – i.e., taking advantage of the shared memory inside shared memory nodes, while using message passing across nodes – and applications of new OpenMP functionality (OpenMP tasking in OpenMP 3.0 [1]), is described in the next sections. These advanced aspects of parallel computing should be applicable to many massively parallel codes intended to run on HPC systems with multicore designs.

Figure 1 shows the grid of GTS following the magnetic field lines as they are twisting around the torus as well as the toroidal domain decomposition of the torus. The two cross sections demonstrate contour plots of density fluctuations driven by Ion Temperature Gradient-Driven Turbulence (ITGDT) [6], which is supposed to cause the experimentally observed anomalous loss of particles and heat at the core of magnetic fusion devices such as tokamaks. Dark and bright areas in the cross sections denote lower (negative) and higher (positive) fluctuation densities, respectively. These fluctuations attach to the magnetic field lines,

[2]Recently, research has been carried out to investigate different forms of grid decomposition schemes, ranging from the pure MPI implementation to the purest shared memory implementation using only one copy of the grid, and all threads must contend for exclusive access [7].

which is a typical characteristic of plasma turbulence in tokamak reactors.

In the following, we focus on one particular step of GTS – the *shifting of particles between toroidal domains* – and discuss how to exploit new OpenMP functionality, which will be substantiated with performance results on our Cray XT machines at NERSC at the end.

3. The GTS particle shifter and how to fight Amdahl's law

The shift phase is an important step in the PIC simulation. After the push phase, i.e., once the equations of motion for the charged particles are computed, a significant portion of the moved particles are likely to end up in neighboring toroidal domains. (Ions and electrons have a separate pusher and shift routines in GTS.) This shift of particles can happen to the adjacent or even to further toroidal domains of the tokamak and is implemented with MPI_Sendrecv functions operating in a ring-like fashion. The amount of shifted particles as well as the number of traversed toroidal domains depends on the toroidal domain decomposition coarsening (*mzetamax*), the time step resolution (*tstep*) and the number of particles per cell (*micell*); all of which can be modified in the input file processed by the GTS loader.

The pseudo-code excerpt in Listing 1 highlights the *major steps* in the original shifter routine. The most important steps in the shifter are iteratively applied and correspond to the following: (1) checking if particles have to be shifted, which is communicated by the allreduce call – lines 3–10 in Listing 1; (2) reordering the particles that keep staying on the domain – line 23 in Listing 1; (3) packing and sending particles to left and right by MPI_Sendrecv calls – lines 13–20 and lines 26–32 in Listing 1; and (4) incorporating shifted particles to the destination toroidal domain (the two loops at the end of the shifter) – lines 35–43 in Listing 1.

The shifter routine involves heavy *communication* due to the MPI_Allreduce and especially because of the ring-like MPI_Sendrecv at every iteration step in each shift phase, where several iterations per shift phase are likely to occur. In addition, intense *computation* is involved mostly because of the particle reordering that occurs after particles have been shifted and incorporated into the new toroidal domain, respectively. Note, that billions of charged particles are simulated in the tokamak causing approximately to the order of millions particles to be shifted at each shifter phase.

While most of the work on the particle arrays can be straight forward parallelized with OpenMP worksharing constructs on the loop level, a substantial amount of time is still spent in non-parallelizable (single-threaded) particle array work (sorting) and in the MPI communication, which is processed sequentially by the master thread in our hybrid parallel model. Figure 2(a) demonstrates in a high-level view the original MPI/OpenMP hybrid approach with its serial and parallel work sections at each MPI process implemented in GTS. Hence, the expected parallel speed-up for the shift routine – as well as of any other parallel program following this hybrid approach – is strictly limited by the time needed for the sequential fraction of this section the MPI task; a fact that is widely known as *Amdahl's law*.

The goal is to reduce the overhead of the sequential parts as much possible by *overlapping MPI communication with computation* using a new OpenMP feature. The notion of overlapping communication and computation in various ways has been described before [10,11] but we present here a new way based on the new functionality of the OpenMP tasking model. OpenMP version 3.0 introduces the task directive, which allows the programmer to specify a unit of parallel work called an *explicit task*, which express *unstructured parallelism* and defines *dynamically generated work units* that will be processed by the team [1]. In order to detect overlappable code regions and for preserving the original semantic of the code, we (manually) look for data dependencies on MPI statements and surrounding computational statements before code transformations are applied. Figure 2(b) gives an overview of the new hybrid approach where MPI communication is executed while independent computation is performed using OpenMP tasks. It can be easily seen from Fig. 2 that the runtime of our application following the new approach is reduced approximately (add OpenMP tasking overhead) by the costs of the MPI communication represented by the dashed arrow. Thus, the new runtime is denoted by $\max(\textit{computation time}, \textit{communication time})$. Below we will present three optimizations to the GTS shifter:

(1) We overlap the MPI_Allreduce call at line 9 from Listing 1 with the two loops from lines 14 and 18. We preserve the original semantics of the program since the packing of particles is independent on the output parameter of the MPI_Allreduce call. The transformed code segments are shown in Listing 2, where we used OpenMP tasks to overlap the MPI function call. Note, that shifting the MPI_Allreduce call below the two loops does not add extra overhead. Note, the program leaves that function in case of $sum\_shift\_p == 0$ and so, the packing statements

```
do iterations=1,N                                          1
!compute particles to be shifted
  !$omp parallel do                                        3
  shift_p=particles_to_shift(p_array);
                                                           5
!communicate amount of shifted
! particles and return if equal to 0                      7
  shift_p=x+y
  MPI_ALLREDUCE(shift_p,sum_shift_p);                      9
  if(sum_shift_p==0) { return; }
                                                           11
!pack particle to move right and left
  !$omp parallel do                                        13
  do m=1,x
    sendright(m)=p_array(f(m));                            15
  enddo
  !$omp parallel do                                        17
  do n=1,y
    sendleft(n)=p_array(f(n));                             19
  enddo
                                                           21
!reorder remaining particles: fill holes
  fill_hole(p_array);                                      23

!send number of particles to move right                   25
  MPI_SENDRECV(x,length=2,..);
!send to right and receive from left                      27
  MPI_SENDRECV(sendright,length=g(x),..);
!send number of particles to move left                    29
  MPI_SENDRECV(y,length=2,..);
!send to left and receive from right                      31
  MPI_SENDRECV(sendleft,length=g(y),..);
                                                           33
!adding shifted particles from right
  !$omp parallel do                                        35
  do m=1,x
    p_array(h(m))=sendright(m);                            37
  enddo
!adding shifted particles from left                       39
  !$omp parallel do
  do n=1,y                                                 41
    p_array(h(n))=sendleft(n);
  enddo                                                    43
enddo
```

Listing 1. Original GTS shifter routine.

right after the MPI_Allreduce call in the original code could be pointlessly executed. However, unnecessary computation is not the case because of $x == y == 0$ for each MPI process in case of *sum_shift_p* $== 0$.

The master thread encounters (due to statement at line 3 from Listing 2 only the thread with id 0 executes the highlighted regions) the tasking statements and creates work for the thread team for deferred execution; whereas the MPI_Allreduce call will be immediately executed, which gives us the overlap. Note, that the underlying MPI implementation has to support at least *MPI_THREAD_FUNNELED* as threading level in or-

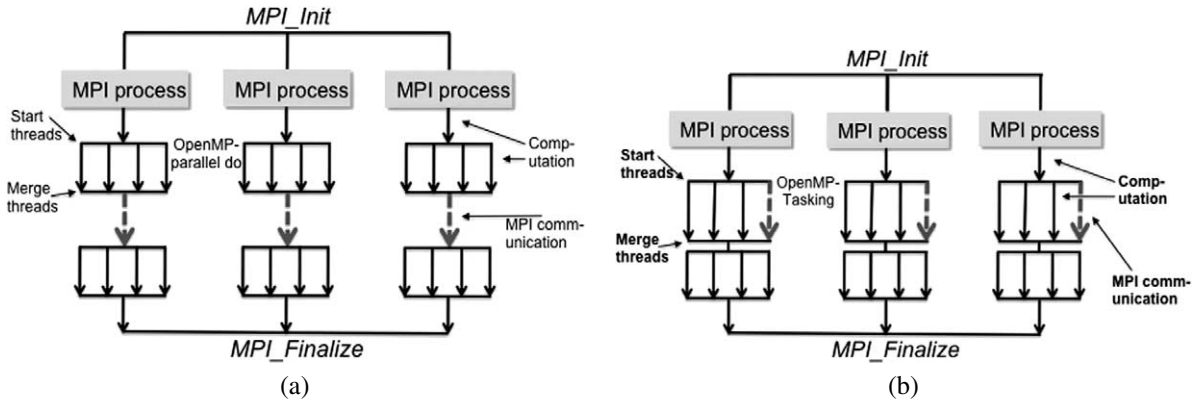

Fig. 2. Two different hybrid models in GTS using standard OpenMP worksharing (a) or the newly introduced OpenMP tasks to execute MPI communication while performing computation (b). (a) Original MPI/OpenMP hybrid model. (b) MPI/OpenMP hybrid model using OpenMP tasks to overlap MPI.

```
shift_p=x+y
!$omp parallel                                          2
!$omp master
  !$omp task                                            4
  do m=1,x
    sendright(m)=p_array(f(m));                         6
  enddo
  !$omp end task                                        8
  !$omp task
  do n=1,y                                              10
    sendleft(n)=p_array(f(n));
  enddo                                                 12
  !$omp end task
                                                        14
  MPI_ALLREDUCE(shift_p,sum_shift_p);
!$omp end master                                        16
!$omp end parallel
if(sum_shift_p==0) { return; }                          18
```

Listing 2. (1) Overlap MPI_Allreduce in the GTS shifter.

der to allow the master thread in the OpenMP model performing MPI calls.[3]

However, the presented solution in Listing 2 is heavily unbalanced (because of $x \neq y$; and the costs for the MPI_Allreduce call is usually lower than the time needed for the loop computation) and does not provide any work for more than three threads per MPI process. For this we subdivided the tasks into smaller chunks to allow better load balancing and scalability among the threads. This is shown in Listing 3 where the master thread generates multiple tasks with loops to the extent of *stride*. Listing 3 has now four loops because of the remaining computation in the two additional loops to the extent of (x *MOD stride*) and (y *MOD stride*) respectively.

(2) Applying similar tasking techniques enables us to overlap the computation intense particle reordering from line 23 of the original code in Listing 1 with communication intense MPI_Sendrecv statements from lines 26, 28 and 30 of Listing 1. Since the particle ordering of remaining particles and the sending or receiving of shifted particles is independently executed, the optimized code shown in Listing 4 does not change the semantics of the original GTS shifter.

[3]To determine the level of thread support from the current MPI library one can execute MPI_Init_thread instead of MPI_init.

```
integer stride=1000
!$omp parallel                                          2
!$omp master
!pack particle to move right                            4
  do m=1,x-stride,stride
    !$omp task                                          6
    do mm=0,stride-1,1
      sendright(m+mm)=p_array(f(m+mm));                 8
    enddo
    !$omp end task                                      10
  enddo
  !$omp task                                            12
  do m=m,x
    sendright(m)=p_array(f(m));                         14
  enddo
  !$omp end task                                        16
!pack particle to move left
  do n=1,y-stride,stride                                18
    !$omp task
    do nn=0,stride-1,1                                  20
      sendleft(n+nn)=p_array(f(n+nn));
    enddo                                               22
    !$omp end task
  enddo                                                 24
  !$omp task
  do n=n,y                                              26
    sendleft(n)=p_array(f(n));
  enddo                                                 28
  !$omp end task
  MPI_ALLREDUCE(shift_p,sum_shift_p);                   30
!$omp end master
!$omp end parallel                                      32
if(sum_shift_p==0) { return; }
```

Listing 3. (2) Overlap MPI_Allreduce in the GTS shifter.

In the new code from Listing 4 any thread in the team does the reordering (alone!) while the master thread takes care of the MPI statements (again, at least *MPI_THREAD_FUNNELED* has to be supported by the MPI library); which does not keep all the threads per MPI process busy (in case *OMP_NUM_THREADS* $\geqslant 3$), but still significantly speeds up the sequential code as we will demonstrate at the end of the section.

(3) The careful reader might have noticed that the code excerpt from Listing 1 only shows three MPI_Sendrecv while the original shift routine in Listing 1 depicts four of them. Since the three MPI_Sendrecv statements from Listing 4 are potentially more time consuming than the particle reordering (because of the middle MPI_Sendrecv of line 8 in Listing 4 sending a large array), we can overlap the fourth original MPI_Sendrecv of line 32 in Listing 1 with the data independent part of the remaining computation of the shifter, i.e., the loop from line 36 in Listing 1 by using, again, the newly introduced OpenMP tasking functionality. This results into the code excerpt from Listing 5, where the second last loop from line 36 in Listing 1 has been overlapped with the fourth MPI_Sendrecv of line 32 in Listing 1. Similar to the previous code optimization from Listing 3 the master threads creates multiple tasks for the loop from line 36 in Listing 1 in order to keep all the threads in the team busy while the master thread is responsible for sending and receiving data from neighboring MPI processes.

To sum up, by applying those three code transformations we are able to overlap all (iteratively called)

```
!$omp parallel                                                    1
!$omp master
  !$omp task                                                      3
  fill_hole(p_array);
  !$omp end task                                                  5

  MPI_SENDRECV(x,length=2,..);                                    7
  MPI_SENDRECV(sendright,length=g(x),..);
  MPI_SENDRECV(y,length=2,..);                                    9
!$omp end master
!$omp end parallel                                                11
}
```

Listing 4. Overlap particle reordering in the GTS shifter.

```
!$omp parallel
!$omp master                                                      2
!adding shifted particles from right
   do m=1,x-stride,stride                                         4
    !$omp task
    do mm=0,stride-1,1                                            6
      p_array(h(m))=sendright(m);
    enddo                                                         8
    !$omp end task
  enddo                                                           10
  !$omp task
  do m=m,x                                                        12
    !adding shifted particles from right
    p_array(h(m))=sendright(m);                                   14
  enddo
  !$omp end task                                                  16

  MPI_SENDRECV(sendleft,length=g(y),..);                          18
!$omp end master
!$omp end parallel                                                20

!adding shifted particles from left                               22
!$omp parallel do
do n=1,y                                                          24
  p_array(h(n))=sendleft(n);
enddo                                                             26
```

Listing 5. Overlap MPI_Sendrecv in the GTS shifter.

MPI functions from the original shifter routine of GTS from Listing 1. We are aware of the fact that for different parts of GTS or other MPI parallel applications such optimizations cannot always be applied due to complicated data dependencies. However, the aim of these code examples starting from Listing 3 to Listing 5 is to discuss these new optimization possibilities provided by OpenMP tasks. The presented techniques, i.e., overlapping (collective) MPI communication with computation, has not been the design incentive in the first place of the new tasking model, but we believe that it can play an important role in many of future HPC systems based on the hybrid MPI/OpenMP programming models. For the sake of completeness

```
shift_p=x+y
!$omp parallel                                                    2
!$omp do onthreads(1:omp,get_num_threads()-1) {
  do m=1,x                                                        4
    sendright(m)=p_array(f(m));
  enddo                                                           6
  do n=1,y
    sendleft(n)=p_array(f(n));                                    8
  enddo
}                                                                 10
!$omp onthreads(0) {
  MPI_ALLREDUCE(shift_p,sum_shift_p);                             12
}
!$omp end parallel                                                14
if(sum_shift_p==0) { return; }
```

Listing 6. Overlap MPI_Allreduce in the GTS shifter using OpenMP subteams.

we want to mention that non-blocking collective MPI communication, e.g., non-blocking allreduce communication (MPI_Iallreduce) are in the process of being standardized in the upcoming MPI 3.0 standard [8]. Nonblocking collective operations are already provided by libNBC [5], a portable implementation of non-blocking collective communication on top of MPI-1, which acts as the reference implementation for the proposed MPI 3.0 functionality currently under consideration by the MPI Forum. However, libNBC is restricted to a few HPC platforms and also exhibits some overhead as seen in previously performed research. In addition, we also see a benefit in using OpenMP tasking to overlap collective MPI communication regarding code portability since the optimized code will run on any system with MPI even if OpenMP support is not given, whereas libNBC is likely to be having made available on a new system, which might be difficult in a lot of cases. Finally, it should be remarked that also OpenMP tasking involves some extra overhead. Which approach – using OpenMP tasking or new MPI non-blocking collectives – performs best remains to be seen once the new MPI 3.0 version is available. Rabenseifner and Wellein [11] point out that the benefit is limited, mainly because the communication time can be hidden by parallelizing it to the numerical threads (which reduces the available threads for numerics by one). Therefore, without parallelizing communication with computation the maximum benefit ratio is $(2 - 1/n)$ on n threads. This maximum is achieved when the communication time and the computation time of the parallel region is identical in the case of non-overlapping. Hence, we use the concept of OpenMP work-sharing on all but 1 threads, which is responsible for the MPI communication. Once this thread doing MPI communication has finished it will join the other threads in the particle computation steps. We note that parallel to the development of the OpenMP tasking concept by Intel, Chapman et al. [2] have introduced another method to solve this problem, where the concept of OpenMP *subteams* has been introduced to allow work-sharing (e.g., loop scheduling) on a subset of threads. Listing 6 gives an example of using the proposed subteam functionality on the GTS particle shifter routine for overlapping the MPI_Allreduce call at line 9 from the original code shown in Listing 1. Here the master thread forms a single threaded team to execute the MPI call (line 12 Listing 6) and the other threads form a team to pack the particles into send buffers in parallel (line 3 Listing 6).

Using such OpenMP subteams facilitate another way to overlap computations and communication. However, this method lacks the flexibility provided by the OpenMP tasking concept since OpenMP subteams explicitly split up communication and computation between thread teams. As a result – in case the MPI communication takes less time than the particle work, which is mostly the case in GTS with its large particle arrays – the communicating thread(-team) remains idle while threads of the other team are performing computation. This is circumvented using the flexible OpenMP tasking approach, which causes the communicating thread to pick up other work from the task pool once finished with its MPI communication. Overlapping communication with computation using OpenMP subteams is appropriate for static code struc-

tures where the relative amount of data and computation does not vary and the number of participating threads and their roles remain the same [2]. Another benefit is the lower overhead implied as opposed to the OpenMP tasking method. However, using OpenMP subteams for computation/communication overlap is not optimal for dynamic application such as GTS, where the ratio of computation and communication can vary from MPI process to MPI process and between iteration steps, respectively.

In the next section we will present performance results of the above mentioned code transformations and compare them to the results gathered when executing the original code.

4. Performance results

The following experiments have been carried out at NERSC's Franklin – a Cray XT4 system having 9572 compute nodes with each node consists of a 2.3 GHz single socket quad-core AMD Opteron processor (Budapest) – and Hopper – a Cray XT5, which in the current phase I has 664 compute nodes each containing two 2.4 GHz AMD Opteron quad-core processors – machines. The second phase of Hopper, arriving in Fall 2010, will be combined with an upgraded phase 1 to create NERSC's first peta-flop system with over 150,000 compute cores. On Franklin we use the Cray Compiler Environment (CCE) version 7.2.1 and the Cray supported MPI library version 4.0.3 based MPICH2. On Hopper CCE version 7.1.4.111 and Cray MPICH2 version 3.5.0 is used.

4.1. Benefits and limitations of hybrid computing

Before we present runtime numbers of the OpenMP tasking optimizations, we want to address the benefits and limitations of the hybrid approach on the Gyrokinetic Toroidal Code (GTC) [3], another global gyrokinetic PIC code, which shares the similar architecture to the GTS code discussed in this paper, and uses the same parallel model. Therefore, the following study for GTC also applies to GTS. Figure 3 illustrates runtime numbers of four GTC runs using the same input parameters but varying the MPI/OpenMP ratio. All four runs are using the same number of compute cores on Hopper. Hence, the first group represents the runtime of GTC using a total of 192 MPI processes where each MPI process creates 8 OpenMP threads. Each group has eight columns reflecting the overall wall-

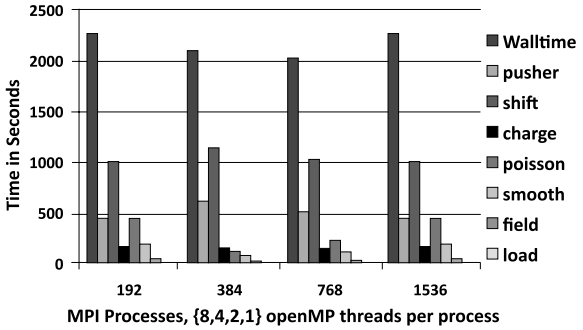

Fig. 3. Evaluation of MPI/OpenMP hybrid model with GTC on Hopper.

time, which is the aggregation of the remaining seven columns, i.e., the PIC steps in GTC. The second group depicts experiments with a total of 384 MPI processes with 4 OpenMP threads per MPI process and so forth. Figure 3 clearly demonstrates that the hybrid approach outperforms the pure MPI approach (the fourth group in Fig. 3) because of the less MPI communication overhead involved and better usability of the shared memory cores on the Hopper compute node. However, this picture also points out the limitations (using 8 OpenMP threads per MPI process performs similar to the pure MPI approach) to a certain number of OpenMP threads per MPI process due to NUMA and cache effects on the AMD Opteron system. In addition, Fig. 3 shows the impact of the shift routine to the overall runtime, which denotes in this experiment to an average of 47% – therefore, a step in the PIC method that is worth optimizing.

4.2. Performance evaluation of OpenMP tasking to overlap communication with computation

The diagrams shown in Fig. 4 present four GTS runs with different input files and domain decomposition executed on the Franklin Cray XT 4 machine. Figure 4(a) gives the breakdown of the runtime for the GTS shift routine with the torus divided up into 128 domains, where each toroidal section is further partitioned into 2 poloidal sections. The first two bars compare the overall runtime of the shifter using the optimized version (shown in dark gray) with the original one (light gray). The other three groups compare the runtime of the three previously introduced code pieces using OpenMP tasks with their original counterparts from Listing 1: "Allreduce" reflects the timing for the code shown Listing 3, "FillingHole" corresponds to the code from Listing 4 and "SendRecv" is

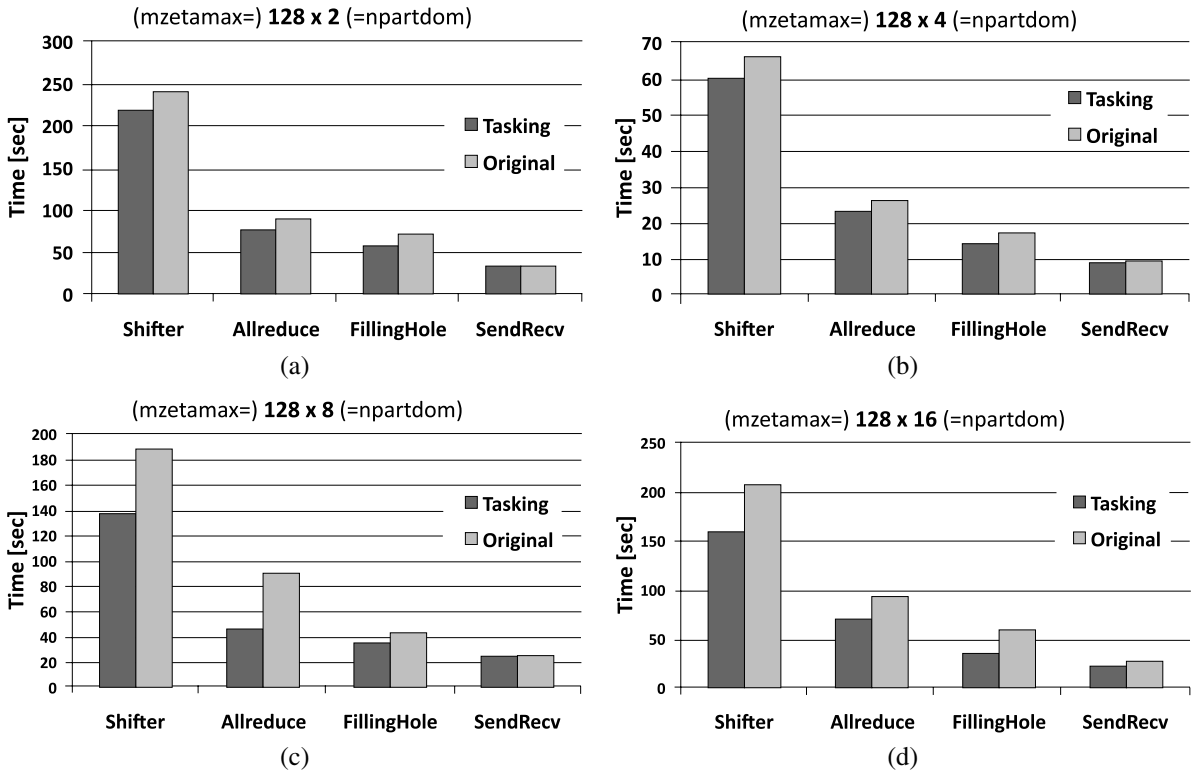

Fig. 4. Performance breakdown of GTS shifter routine using 4 OpenMP threads per MPI process with varying domain decomposition and particles per cell on Franklin showing that MPI communication can be successfully overlapped with independent computation using OpenMP tasks resulting in a maximum speed-up of the GTS shifter of 35% (c). (a) 128 toroidal domains, each having 2 poloidal domains. (b) 128 toroidal domains, each having 4 poloidal domains. (c) 128 toroidal domains, each having 8 poloidal domains. (d) 128 toroidal domains, each having 16 poloidal domains.

the measurement for Listing 5. Those three parts together with other computation on the particle arrays (as indicated at line 4 in the original code shown in Listing 1) add up to the numbers presented in the "Shifter" group. Besides that different input settings (e.g., varying the number of particles per cell) have been used to generate Fig. 4(a)–(d), the main difference is that the number of poloidal domains (*npartdom*) goes from 2 to 16. As indicated in the introduction of the parallel model of GTS in Section 2, all the MPI communication in the shift phase uses a *toroidal MPI communicator*, which is constant of size 128 in the four presented figures. However, as it can be seen from Fig. 4, it clearly makes a difference if particles are shifted in the 128-MPI-processes-toroidal-domain of a GTS run with an overall usage of 256 MPI processes (Fig. 4(a)) than in a 128-MPI-processes-toroidal-domain of a GTS run with a total of 2048 MPI processes (Fig. 4(d)). This is mainly because of the increasing latency for sending messages within larger toroidal communicators in a production run of GTS on the Cray XT4. The speed-up, or to put it in other words, the difference between the dark gray bar and the light gray bar, for each phase in the shifter is the time consumed by the MPI communication, which is overlapped in the newly introduced shifter steps (to simplify matters, neglecting the overhead involved with OpenMP tasking and assuming that the costs of loops workshared with traditional "omp parallel do" statements is the same as processing those loops workshared with OpenMP tasks.). Moreover, we can observe that the benefit of the "SendRecv" optimization (Listing 5) also depends on the number of MPI domains. While Fig. 4(a)–(c) show no or only marginal performance benefits, the speed-up due to the "SendRecv" optimization is about 18% in Fig. 4(d), which represents a 2048 MPI processes run. The tremendous speed-up due to the "Allreduce" optimization from Listing 4 (more than 100%) in the 1024 MPI processes run is pleasant, but is likely to be just a positive outlier and requires further investigation. The final GTS shifter routine, which incorporates all the previous discussed optimizations is shown in Listing 7.

```
do iterations=1,N

!compute particles to be shifted
  !$omp parallel do
  shift_p=particles_to_shift(p_array);

!overlap allreduce with filling of send buffers
  LISTING 3

!overlap filling of holes with three send-recv calls
  LISTING 4

!overlap adding of received data from right with the
! fourth send-recv call for receiving data from the
! left; plus add data received from left to array
  LISTING 5

enddo
```

Listing 7. Final optimized GTS shifter routine.

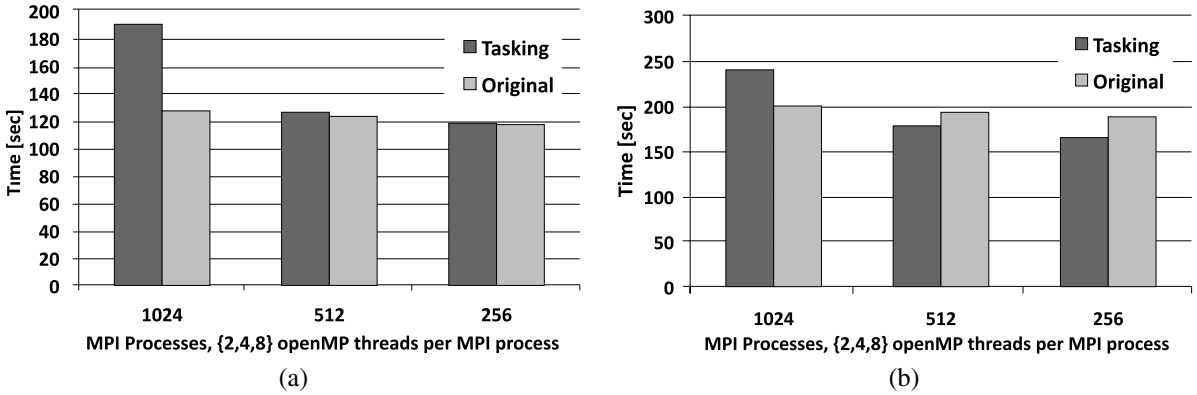

Fig. 5. Performance for overlapping execution of two consecutive MPI_Allreduce calls on Hopper. (a) Allreduce of 1 integer. (b) Allreduce of an array of 100 integers.

Next, we want to conclude our experiments with a discussion about the overlapping of MPI communication with consecutive, independent MPI communication.

4.3. Overlap communication with communication

Going one step further in reducing the time spent in the sequentially[4] executed MPI communication, we want to show early results of experiments with overlapping of MPI communication with other MPI communication succeeding in the control flow of the parallel program that is data independent on the preceding one. Examples in GTS are the consecutive independent MPI_Sendrecv statements in the shifter from above and four consecutive independent MPI_Allreduce calls in the ion pusher phase.

Figure 5 presents runtime comparisons of succeeding and independent MPI_Allreduce calls with varying messages sizes. Figure 5(a) and (b) show the time it takes with 1024 MPI process (2 OpenMP threads per MPI process), 512 MPI processes (4 OpenMP threads per MPI process) and 256 MPI processes (8 OpenMP threads per MPI process) to execute the code shown in Listing 8, which is highlighted in dark gray bars and compare it with the costs of processing the code from Listing 8 without OpenMP compiler support,

[4]In the hybrid MPI/OpenMP programming model the remaining cores are idle when one core executes an MPI command.

```
!$omp parallel                                                    1
!$omp master
do i=1,N                                                          3
  !$omp task
  MPI_Allreduce(in1,out1,length,MPI_INT,                          5
        MPI_SUM,MPI_COMM_WORLD,ierror);
  !$omp end task                                                  7
  MPI_Allreduce(in2,out2,length,MPI_INT,
        MPI_SUM,MPI_COMM_WORLD,ierror);                           9
enddo
!$omp end master                                                  11
!$omp end parallel
```

Listing 8. Overlap MPI_Allreduce with MPI_Allreduce.

i.e., without the overlap. Consequently, the number of used CPU cores is constant (==2048) in these experiments. Figure 5(a) reflects a run with MPI_Allreduce calls of just one integer variable whereas Fig. 5(b) shows results for MPI_Allreduce calls of an integer array of size 100. While no performance gain can be observed in the experiment with allreduces of size 1 (Fig. 5(a)), we can see a slight overlap in Fig. 5(b) for the 4- and 8-OpenMP-threads run. The run with 4 OpenMP threads is of major interest since it reflects the recommended MPI/OpenMP ratio for production runs on Hopper. However, we also see that no full overlap could be achieved, but we expect better threading support from upcoming MPI libraries. We are aware of the fact that 100% overlap is impossible to achieve due to the sequential nature of communication in a single network, but these early experimental data has already demonstrated that some (to the programmer invisible) steps of the MPI_Allreduce call can be successfully overlapped. Moreover, with optimal support of the MPI_THREAD_MULTIPLE threading level in MPI libraries such as already implemented in MPICH2 – where any thread can call MPI functions at any time – we expect a significant performance gain in (partially) overlapping more consecutive independent collective MPI function calls (e.g., the four consecutive independent MPI_Allreduce calls occurring in the ion pusher phase of GTS) in a hybrid programming model since future systems will have hardware support for multiple, concurrent communication channels per node [12]. Similar experiments to the one shown in Listing 8 have been conducted on Hopper with consecutive MPI_Sendrecv calls achieving similar same speed-ups.

5. Conclusion

Summing up, we have demonstrated that overlapping MPI communication with independent computation by the newly introduced OpenMP tasking model has a large potential, especially for massively parallel applications such as GTS scaling up to several thousands of compute cores. Consequently we believe that similar strategies can be applied to other massively parallel codes running on cluster equipped with multicore processors. As collective and/or point-to-point time increasingly becomes a bottleneck on future HPC clusters comprising thousands of multicore processors, using threading to keep the number of MPI processes per node to a minimum and to overlap – if possible – those MPI calls with independent surrounding statements is a promising strategy. Furthermore, we showed early experimental data of overlapping MPI communication with independent MPI communication, which we believe to be another valuable feature for future multicore HPC systems. Finally, we point out that the presented code transformations and data dependence analysis have been manually carried out and could be performed by automated source-to-source translating compilers such as the ROSE compiler framework [9] using static analysis techniques to guide subsequent code optimizations.

Acknowledgements

A majority of the work in this paper was supported by the Petascale Initiative in Computational Science at NERSC. Some additional research on this paper was supported by the Cray Center of Excellence at NERSC. Additionally, we are grateful for interactions

with John Shalf and Nicholas Wright and for the extended computer time as well as the valuable support from NERSC.

References

[1] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan and G. Zhang, The design of OpenMP tasks, *IEEE Trans. Parallel Distrib. Syst.* **20**(3) (2009), 404–418.

[2] B.M. Chapman, L. Huang, H. Jin, G. Jost and B.R. de Supinski, Toward enhancing OpenMPs work-sharing directives, in: *Euro-Par*, September 2006, pp. 645–654.

[3] S. Ethier, W.M. Tang and Z. Lin, Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms, *J. Phys. Conf. Ser.* **16**(1) (2005), 1.

[4] S. Ethier, W.M. Tang, R. Walkup and L. Oliker, Large-scale gyrokinetic particle simulation of microturbulence in magnetically confined fusion plasmas, *IBM J. Res. Dev.* **52**(1,2) (2008), 105–115.

[5] T. Hoefler, A. Lumsdaine and W. Rehm, Implementation and performance analysis of non-blocking collective operations for MPI, in: *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 2007, ACM, pp. 1–10.

[6] J.N. Leboeuf, V.E. Lynch, B.A. Carreras, J.D. Alvarez and L. Garcia, Full torus Landau fluid calculations of ion temperature gradient-driven turbulence in cylindrical geometry, *Phys. Plasmas* **7**(12) (2000), 5013–5022.

[7] K. Madduri, S. Williams, S. Ethier, L. Oliker, J. Shalf, E. Strohmaier and K. Yelicky, Memory-efficient optimization of Gyrokinetic particle-to-grid interpolation for multicore processors, in: *SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, New York, NY, USA, 2009, ACM, pp. 1–12.

[8] MPI-Forum, Collective communications and topologies working group, 2009, available at: https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/CollectivesWikiPage.

[9] D.J. Quinlan, ROSE: compiler support for object-oriented frameworks, *Parallel Proc. Lett.* **10**(2,3) (2000), 215–226.

[10] R. Rabenseifner and G. Wellein, Communication and optimization aspects of parallel programming models, in: *EWOMP*, September 2002.

[11] R. Rabenseifner and G. Wellein, Communication and optimization aspects of parallel programming models on hybrid architectures, *Int. J. High Perform. Comput. Appl.* **17**(1) (2003).

[12] M. Snir, A proposal for hybrid programming support on HPC platforms, 2009, available at: https://svn.mpi-forum.org/trac/mpi-forum-web/raw-attachment/wiki/MPI3Hybrid/MPI%2BOpenMP.pdf.

[13] W.X. Wang, Z. Lin, W.M. Tang, W.W. Lee, S. Ethier, J.L.V. Lewandowski, G. Rewoldt, T.S. Hahm and J. Manickam, Gyrokinetic simulation of global turbulent transport properties in tokamak experiments, *Phys. Plasmas* **13** (2006).