Application Acceleration on Current and Future Cray Platforms

Alice Koniges, Robert Preissl, Jihan Kim, *Lawrence Berkeley National Laboratory*
David Eder, Aaron Fisher, Nathan Masters, Velimir Mlaker, *Lawrence Livermore National Laboratory*
Stephane Ethier, Weixing Wang, *Princeton Plasma Physics Laboratory*
Martin Head-Gordon, *University of California, Berkeley*
and **Nathan Wichmann**, *Cray Inc.*

ABSTRACT: Application codes in a variety of areas are being updated for performance on the latest architectures. We describe current bottlenecks and performance improvement areas for applications including plasma physics, chemistry related to carbon capture and sequestration, and material science. We include a variety of methods including advanced hybrid parallelization using multi-threaded MPI, GPU acceleration, libraries and auto-parallelization compilers.

KEYWORDS: hybrid parallelization, GPU's, multicore, optimization, applications

I. Introduction

In this paper we examine three different applications and means for improving their performance, with a particular emphasis on methods that are applicable for many/multicore and future architectural designs. The first application comes from from magnetic fusion. Here we take an important magnetic fusion particle code that already includes several levels of parallelism including hybrid MPI combined with OpenMP. In this case we study how to include advanced hybrid models that use multi-threaded MPI support to overlap communication and computation. In the second example, we consider a portion of a large computational chemistry code suite. In this case, we consider what parts of the computation are good candidates for GPU acceleration, which is one likely architectural component on future Cray platforms. Here we show performance implementation and improvement on a current GPU cluster. Finally, we consider an application from fluids/material science that is currently parallelized by a standard MPI-only model. We use tools on the XT platform to identify bottlenecks, and show how significant performance improvement can be obtained through optimizing library utilization. Finally, since this code is MPI-only, we consider if this code is amenable to hybrid parallelization and discuss potential means for including hybrid code via automatic hybridization tools.

II. Fusion Application

A. GTS — A massively parallel magnetic fusion application

The fusion application chosen for this study is the Gyrokinetic Tokamak Simulation (GTS) code [30], 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 [10]. 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. For this paper GTS simulation runs have been conducted simulating a laboratory-size tokamak of 0.932m major radius and 0.334m 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.

B. The Parallel Model

The *parallel model of GTS has three independent levels*: **(1)** GTS uses a one-dimensional (1D) domain decomposition in

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

the toroidal direction (the long way around the torus). This is the original scheme of expressing parallelism using the Message Passing Interface (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 [10]. 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 shared memory inside shared memory nodes, while using message passing across nodes — and applications of new OpenMP functionality (OpenMP tasking in OpenMP 3.0 [3]), 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) [18], which is supposed to cause the experimentally observed anomalous loss of particles and heat at the core of magnetic fusion devices such as tokamaks. Blue and red areas in the cross sections denote lower (negative) and higher (positive) fluctuation densities, respectively. These fluctuations attach to the magnetic field lines — a typical characteristic of plasma turbulence in tokamak reactors.

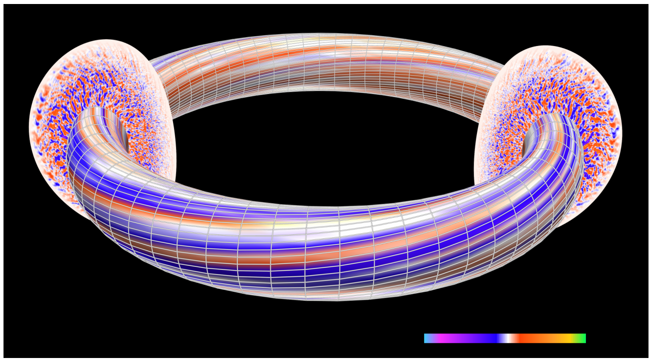


Fig. 1. GTS' toroidal domain decomposition with magnetic field lines and density fluctuations

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.

C. The GTS Particle Shifter & 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 to 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 to 20 and Lines 26 to 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 to 43 in Listing 1.

The shifter routine involves heavy *communication* due to the MPI_Allreduce and especially because of the ring-like

[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 [21].

```
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
}
```

Listing 1. Original GTS shift routine

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

```
!$omp parallel
!$omp master                                               2
do i=1,N
   MPI_Allreduce(in1,out1,length,MPI_INT,                  4
          MPI_SUM,MPI_COMM_WORLD,ierror);
   !$omp task                                              6
   MPI_Allreduce(in2,out2,length,MPI_INT,
          MPI_SUM,MPI_COMM_WORLD,ierror);                  8
   !$omp end task
enddo                                                      10
!$omp end master
!$omp end parallel                                         12
```

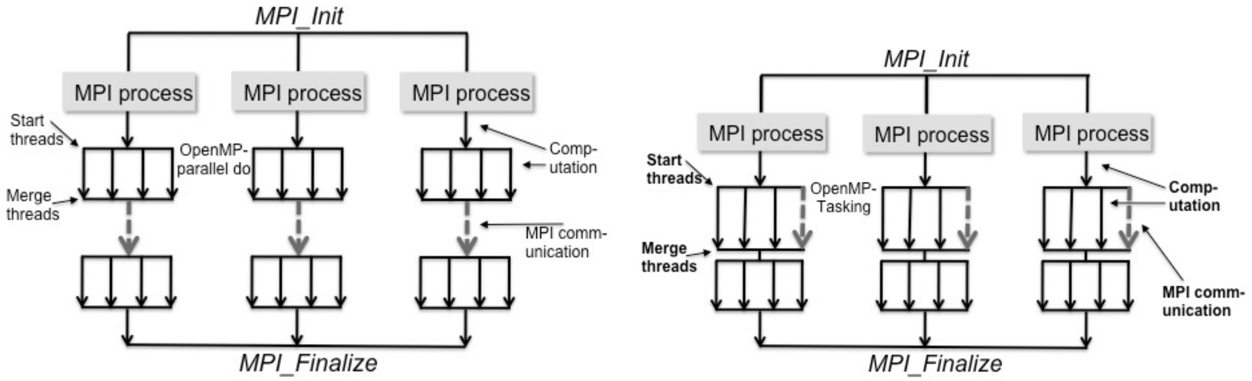
Listing 2. Overlap MPI_Allreduce with MPI_Allreduce

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 the new OpenMP tasking functionality[3] The notion of overlapping communication and computation in various ways has been described before[24], [25] but we present here a new way based on the new fuctionality of the OpenMP tasking model. 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 Figure 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. 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 3, 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 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

[3] 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 define *dynamically generated work units* that will be processed by the team [3].



(a) Original MPI/OpenMP hybrid model

(b) MPI/OpenMP hybrid model using OpenMP tasks to overlap MPI

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).

```
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 3. (1) Overlap MPI_Allreduce in the GTS shifter

case of $sum\_shift\_p == 0$.

The master thread encounters (due to statement at Line 3 from Listing 3 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 order to allow the master thread in the OpenMP model performing MPI calls[4].

However, the presented solution in Listing 3 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 4

```
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 4. (2) Overlap MPI_Allreduce in the GTS shifter

where the master thread generates multiple tasks with loops to the extent of $stride$. Listing 4 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

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

```
!$omp parallel                                  1
!$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 5. Overlap particle reordering in the GTS shifter

```
!$omp parallel
!$omp master
!adding shifted particles from right
    do m=1,x-stride,stride
      !$omp task
      do mm=0,stride-1,1
         p_array(h(m))=sendright(m);
      enddo
      !$omp end task
   enddo
   !$omp task
   do m=m,x
      p_array(h(m))=sendright(m);
   enddo
   !$omp end task

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

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

Listing 6. Overlap MPI_Sendrecv in the GTS shifter

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 5 does not change the semantics of the original GTS shifter. In the new code from Listing 5 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 $OMP\_NUM\_THREADS \geq 3$), 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 5 are potentially more time consuming than the particle reordering (because of the middle MPI_Sendrecv of Line 8 in Listing 5 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 6, 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 4 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) 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 4 to Listing 6 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 we want to mention that nonblocking 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 [22]. Nonblocking collective operations are already provided by libNBC [13], a portable implementation of nonblocking 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 nonblocking collectives — performs best remains to be seen once the new MPI 3.0 version is available. Rabenseifner and Wellein[25] point out that that the benefit is limited, mainly because the communication time can be hidden by paralelizing it to the numerical threads (which reduces the available threads for numerics by one). Therefore, without parallelizing communication with communication, 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

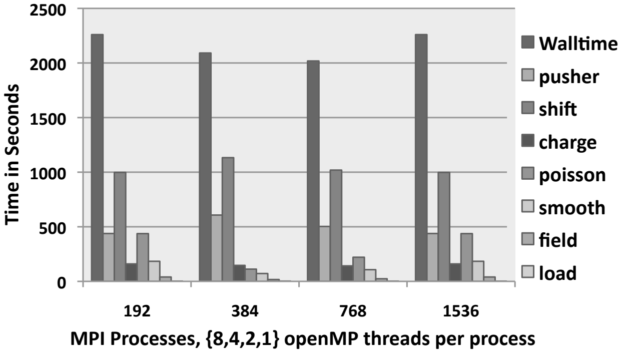


Fig. 3. Evaluation of MPI/OpenMP hybrid model with GTC on Hopper.

in the case of non-overlapping. To solve invoke the concept of OpenMP work-sharing on all but 1 threads, we use the new OpenMP task concept. We note that parallel to the development of the OpenMP task concept by Intel, Chapman, et al. have introduced another method to solve this problem.[4] Here, the concept of OpenMP *subteams* to allow work-sharing (e.g., loop scheduling) on a subset of threads.

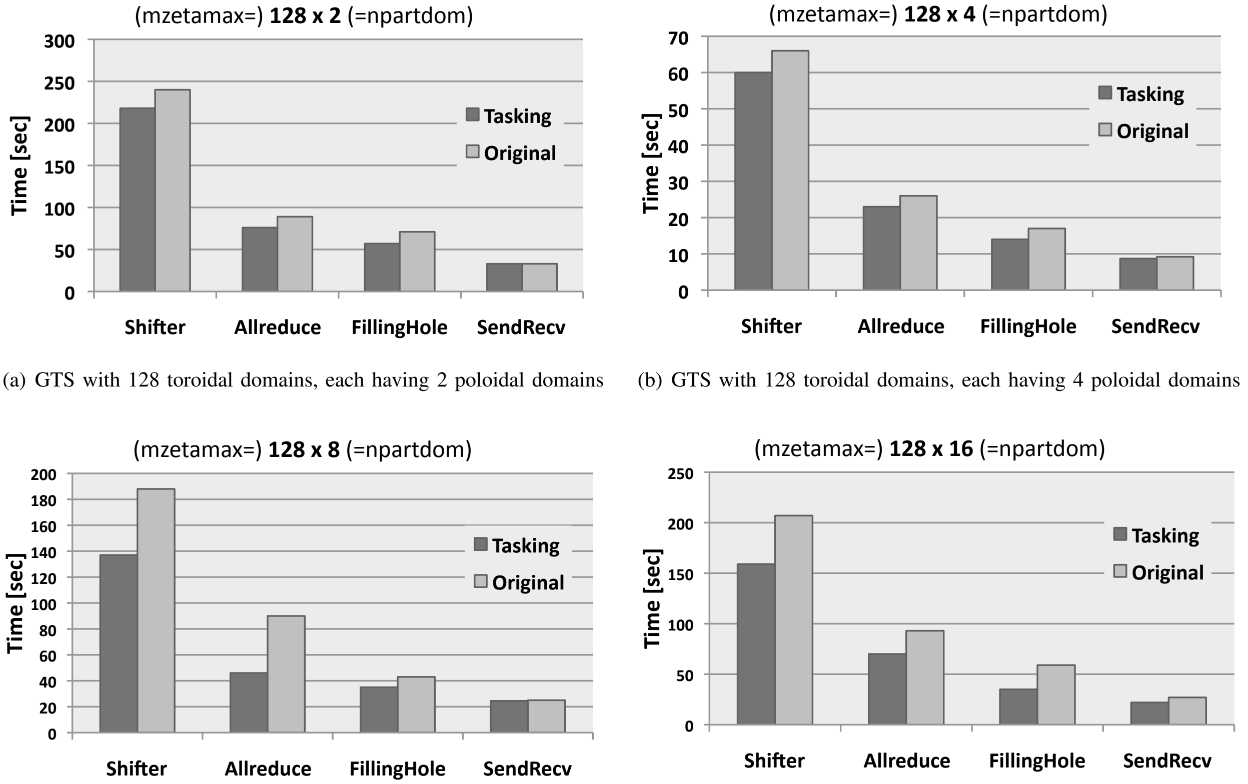
In the next section we will present performance results of the above mentioned code transformations and compare them to the results gathered when executing the original code.

D. 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 150000 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.

1) Benefits & 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) [9], 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 walltime, 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 Figure 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, Figure 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.

2) Performance Evaluation of OpenMP tasking to overlap communication with computation: The diagrams shown in Figure 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 4, "FillingHole" corresponds to the code from Listing 5 and "SendRecv" is the measurement for Listing 6. 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 Figures 4(a) to Figures 4(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 II-B, 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 Figure 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 (Figure 4(a)) than in a 128-MPI-processes-toroidal-domain of a GTS run with a total of 2048 MPI processes (Figure 4(d)). This is mainly because the MPI processes part of the toroidal MPI communicator in larger MPI runs of GTS are physically further away from each other than in a GTS run with fewer MPI poloidal domains; hence, causing more burden on the Cray Seastar interconnect to sending messages. 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



(a) GTS with 128 toroidal domains, each having 2 poloidal domains

(b) GTS with 128 toroidal domains, each having 4 poloidal domains

(c) GTS with 128 toroidal domains, each having 8 poloidal domains

(d) GTS with 128 toroidal domains, each having 16 poloidal domains

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.

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 6) also depends on the number of MPI domains. While Figures 4(a) to Figures 4(c) show no or only marginal performance benefits, the speed-up due to the ”SendRecv” optimization is about 18% in Figure 4(d) which represents a 2048 MPI processes run. The tremendous speed up due to the ”Allreduce” optimization from Listing 5 (more than 100%) in the 1024 MPI processes run is pleasant, but is likely to be just a positive outlier and requires further investigation.

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

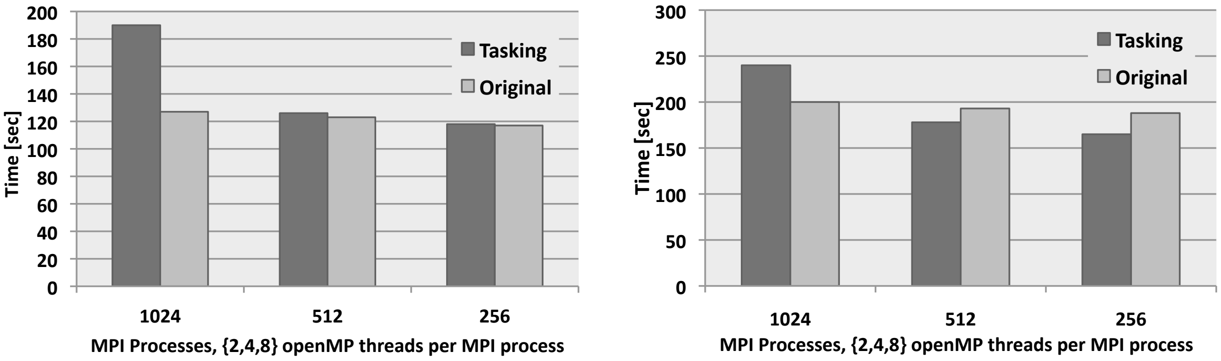
3) Overlap communication with communication: Going one step further in reducing the time spent in sequentially[5] executed MPI communication, we want to show early results of experiments with overlapping of MPI communication with other MPI communication succeeding in the control flow of the parallel 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.

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

Listing 7. Overlap MPI_Allreduce with MPI_Allreduce

Figure 5 presents runtime comparisons of succeeding and independent MPI_Allreduce calls with varying messages sizes. Figure 5(a) and Figure 5(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 7, which is highlighted in dark gray

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



(a) Allreduce of 1 integer

(b) Allreduce of an array of 100 integers

Fig. 5. Performance evaluation for overlapping execution of two consecutive MPI_Allreduce calls on Hopper.

bars and compare it with the costs of processing the code from Listing 7 without OpenMP compiler support, 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 Figure 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 (Figure 5(a)), we can see a slight overlap in Figure 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, which can been verified when looking at GTS performance results on Hopper in Figure 3. However, we also see that no full overlap could be achieved, but 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 [28]. Similar experiments to the one shown in Listing 7 have been conducted on Hopper with consecutive MPI_Sendrecv calls achieving similar same speed-ups.

E. 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 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 [23] using static analysis techniques to guide subsequent code optimizations. The ROSE compiler framework will be introduced in more detail in section IV-E of the material modeling application.

III. Chemistry Application

A. Introduction

Q-Chem is a computational chemistry software that specializes in quantum chemistry calculations, which includes Hartree-Fock (HF), density functional theory (DFT), coupled cluster (CC), configuration interaction, and Møller-Plesset pertubation theory. Many of these calculation methods provides researchers with the ability to accurately predict molecular equilibrium structures, which entails minimizing energy with respect to atomic positions. Due to the extreme reliability of these theoretical predictions, they can be considered suitable alternatives to experimental structure determination. In this paper, we focus on the second-order Møller-Plesset perturbation theory (MP2), which initially starts off with the mean-field HF approximation [1] and treats the correlation energy via Rayleigh-Schrödinger perturbation theory to the second order [20]. More specifically, we focus on a MP2 method that utilizes the resolution-of-the-identity (RI) approximation [19], in which the incorporation of the RI-approximation into the MP2 theory (RI-MP2) results in usage of auxiliary basis set to approximate charge distributions, subsequently reducing the computational cost of the MP2 method.

Compared to DFT, which is a popular alternative method used to conduct electronic structure calculations, RI-MP2 does not suffer from the self-interaction problem [16] and can account for 80-90% of the correlation energy [15]. Moreover, geometry optimizations using MP2 methods have generated equilibrium structures more reliable than HF, popular DFT alternatives and in some cases even CCSD. Unfortunately, there exists a fifth-order computational dependence on the system size when the MP2 and RI-MP2 theory is formulated in a basis of orthonormal set of eigenfunctions that diagonalize the Fock matrix. Comparatively, DFT methods can demonstrate nearly linear scaling for reasonably extended molecular systems, which is a major reason why DFT remains more popular. Thus, in order to obtain RI-MP2 geometry optimizations for large molecular systems in a reasonable time, we need to explore ways to cut down on the computational cost. In this work, we utilize graphics processor units (GPU) to speed up the RI-MP2 energy gradient calculations. Similar work has been conducted on the RI-MP2 energy calculation and a speedup of 4.3x has been observed in single point energy calculations of linear alkanes [29]. Compared to the CPU, more transistors in GPUs are devoted to data processing as opposed to cache memory and flow control. As such, there exists potential for massive parallelism within the GPUs and applications that can be easily formatted into the SIMD (single instruction, multiple data) instructions can benefit greatly from using GPUs.

B. GPU RI-MP2 gradient algorithm

In this section, we first explain the CPU algorithm used in Q-Chem, analyze the computational cost associated with the different steps of the current program, and finally provide an alternative GPU algorithm. In Q-Chem, the CPU RI-MP2 gradient code works under following constraints: quadratic memory, cubic disk storage, quartic I/O requirements, and quintic computational cost with respect to system size. We adhered to these constraints while optimizing for the computational cost. The initial RI-MP2 gradient algorithm (while omitting the self-consistent field (SCF) procedure) consists of seven major steps: (1) RI-overhead: formation of the $(P|Q)^{-1}$ matrix, (2) construction and storage of the three-centered integrals in the mixed canonical MO basis/auxiliary basis: $(ia|P)$ (3) construction of the C_{ia}^{Q} matrix, (4) assembly of the Γ_{ia}^{Q} (i.e. RI-MP2 correction to the two particle density matrix), $P_{ca}^{(2)}$ (i.e. virtual-virtual correction to the one-particle density matrix), and $P_{kj}^{(2)}$ (i.e. active-active correction to the one-particle density matrix), (5) construction of Γ^{RS} (i.e. the RI-MP2 specific two-particle density matrix), (6) Γ_{ia}^{Q} transposition, and (7) assembly of the L, P, and W matrices; solution of the Z-vector equation and final gradient evaluation.

Figure 6 provides proportional wall times for each one of the aforementioned steps for different glycine-n molecules for n = 1, 2, 4, 8, and 16 with cc-pVDZ correlation-consistent basis sets. All of these initial simulations were conducted on the Greta cluster, which consists of AMD quad-core Opteron processors. From the figure, we can see that as the system size increases, time spent in step 4 becomes proportionally larger. For example, for glycine-16 (115 atoms) input, 83% of the total RI-MP2 routine wall time is spent in step 4. Subsequently, we focus our effort to reduce the step 4 computation time. For large size molecules, step 7, which finalizes the gradient evaluation occupies the next largest step time and we list the total times in Table I for various glycine molecules.

Next, we further analyze what is actually happening in the step 4 portion of the code. Step 4 consists of assembly of Γ_{ia}^{Q}, $P_{ca}^{(2)}$, and $P_{kj}^{(2)}$ matrices, which are obtained from BLAS 3 matrix matrix multiplications and entail quintic computational efforts due to iterations over all i and j. In addition, there exists three quartic I/O steps, which are needed to construct the core quantities described above as unfortunately for large molecules, we cannot fit all the necessary data into CPU memory all at once and thus need to read and write into temporary files stored in the hard drive as the code progresses. Here is a more detailed look at the algorithm involved in this step [6], which is also included in the CPU RI-MP2 paper.

We have converted this step 4 CPU routine to CPU+GPU CUDA C routine. For our numerical simulations, we have used the Tesla/Turing GPU cluster at NERSC, which is a testbed consisting of two shared-memory nodes named Tesla and Turing. Each are Sun SunFire x4600-M2 servers with 8 AMD quad-core processors, 256 GB shared memory with the two nodes sharing an NVidia QuadroPlex 2200-S4, which contains four NVidia FX-5800 Quadro GPUs, with each GPU having 4GB of memory and 240 CUDA parallel processor cores.

For all simulations, we have used CUDA Toolkit and SDK v2.3. For matrix matrix multiplications, we initially used

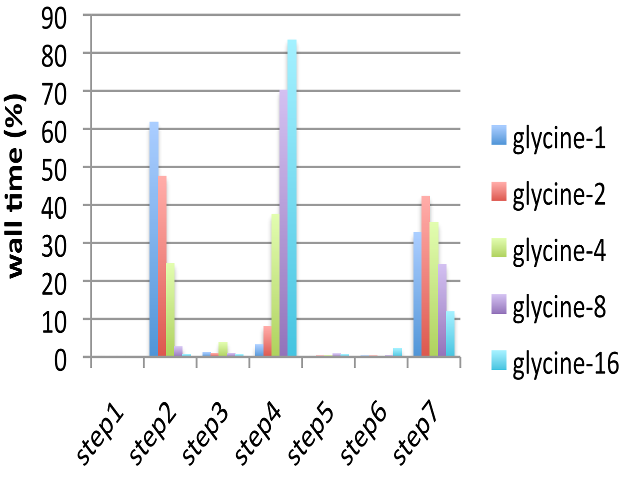


Fig. 6. Percentage RI-MP2 wall time for glycine-n molecules with n = 1, 2, 4, 8, and 16.

```
Loop over active occupied orbitals, $i$
 Load $(ia|P)$ $\forall$ $a$, $P$, given $i$ from disk

  Loop over batches of active occupied orbitals, $ob$

   Loop over $j \in ob$
    Load $C_{jb}^P$ $\forall$ $b$, given $j$ from disk
    Make $(ia|jb) = \Sigma_P (ia|P) C_{jb}^P$ $\forall$ $a$, $b$
    Make $t_{ij}^{ab} = (ia||jb)/\Delta_{ij}^{ab}$ $\forall$ $a$, $b$
    Accumulate $t_{ij}^{ab}$ $\forall$ $a$, $b$, $j \in ob$, given $i$
    Increment $E_{RI-MP2} + = \frac{1}{4} t_{ij}^{ab} (ia||jb)$
    Increment $P_{ca} + = \Sigma_b t_{ij}^{ab} t_{ij}^{cb}$ $\forall$ $a$, $c$, given $ij$
    Increment $\Gamma_{ia}^P + = \Sigma_b t_{ij}^{ab} C_{jb}^P$ $\forall$ $a$, $P$, given $ij$
   End Loop over $j \in ob$

   Loop over batches of active virtual orbitals, $ob$
    Extract $t_{ij}^{ab}$ $\forall$ $a \in vb$, $b$, $j \in ob$, given $i$
    Write $t_{ij}^{ab}$ $\forall$ $a \in vb$, $b$, $j \in ob$, given $i$ to disk
   End Loop over batches of active virtual orbitals, $ob$

  End Loop over batches of active occupied orbitals, $ob$

 Write $\Gamma_{ia}^P$ $\forall$ $a$, $P$, given $i$ to disk

  Loop over batches of active virtual orbitals, $vb$
   Load $t_{ij}^{ab}$ $\forall$ $a \in vb$, $b$, $j \in ob$, given $i$

    Loop over $a \in vb$
     Extract $t_{ij}^{ab}$ $\forall$ $b$, $j$, given $(ia)$
     Increment $P_{kj} + = \Sigma_b t_{ik}^{ab} t_{ij}^{ab}$ $\forall$ $j$, $k$, given $(ia)$
    End Loop over $a \in vb$

  End Loop over batches of active virtual orbitals, $vb$
End Loop over active occupied orbitals, $i$
```

Fig. 7. Detailed look at the algorithm behind step 4

the CUBLAS 2.0 library but later on switched to Vasily Volkov's GEMM kernel given that CUBLAS cannot be called with the asynchronous API. This is a big downside of the current CUBLAS library and accordingly it disallows us to concurrently copy data from CPU to the GPU (and vice versa) while using any of the CUBLAS matrix matrix multiplications. In our code, we are only interested in the double precision matrix matrix multiplications given that extra precision is important in most quantum chemistry calculations. Double-precision general matrix multiply subroutines (DGEMM) are considerably slower than the single-precision general matrix multiply subroutines (SGEMM) (at least for non-Fermi architecture GPUs) as we obtain a maximum value of around 75 GFLOPS using the GPU as opposed to reports of around 350 GFLOPS for SGEMM. For the CUBLAS DGEMM routine, performance numbers varied greatly depending on whether the dimensions of the input matrices were multiples of 16 or not. For example, upon multiplying a 4340 x 915 matrix A with 915 x 915 matrix B, we found the performance number to be 53.04 GFLOPS. On the other hand, upon multiplying a 4352 x 928 matrix A with 928 x 928 matrix B, we obtained 74.36 GFLOPS. In comparison, using Volkov's DGEMM kernel gave us smaller variation (73.50 and 74.77 GFLOPS for the aforementioned cases). It's unclear why the numbers vary so greatly in the CUBLAS DGEMM routine but we suspect it might be related to the fact that global memory loads and stores by threads of a half warp (16 threads) and accordingly, these transactions are not being properly coalesced in the CUBLAS DGEMM routine for matrices whose dimensions are not multiples of 16.

TABLE I
STEP4, STEP7, AND TOTAL WALL TIME IN SECONDS

	$n = 1$	$n = 2$	$n = 4$	$n = 8$	$n = 16$
step4	2.1	21.5	485.3	4993.8	80913.1
step7	21.6	112.1	455.6	1737.9	11532.5
total	66.0	264.7	1289.1	7102.4	96901.9

The step 4 algorithm can be seen in figure 7. Given that the number of active occupied orbitals is greater than the number of active virtual orbitals, the most computationally intensive part of the step 4 routine occurs during the loop over $j \in ob$. Most of this paper will concentrate on the algorithm inside this loop. Within the $j \in ob$ loop, the CPU code was transformed into a CPU + GPU code in a following way in our initial implementation. First, the matrix C_{jb}^P was read from hard drive for a given j. Afterwards, the matrix, which is stored as a one-dimensional vector, was transferred from the CPU to the GPU memory via the PCI Express using the cudaMemcpy CUDA kernel call. Because Tesla/Turing has a PCI Express 1.1 with only 8 lanes, the data transfer bandwidth peaked only at around 1.4 GB/sec. A new NERSC GPU cluster called Dirac is equipped with PCI Express 2.0 with 16 lanes so we expect the data transfer bandwidth to be much higher in this cluster (5 – 6 GB/sec). Unfortunately, Dirac is still undergoing its initial configurations and unavailable to users at this moment. Once the data is in the GPU, we call the DGEMM kernel and obtain $(ia|jb)$ with the matrix matrix multiplications. For subsequent operations inside the loop, we need not transfer the data stored in the GPU memory back to the CPU given that we can conduct all of our operations inside the GPU. In general, transferring data back and forth over the PCI Express lane is costly and should be avoided as much as possible. Fortunately in our program, we only need to transfer the GPU data back to the CPU at the end of the loop when our work is finished. At the end of our first implementation, the total computation cost inside this loop for a given iteration is as follows: T_{tot} = T_{read} + $T_{transfer}$ + T_{mm_1} + T_{mm_2} + T_{mm_3} + T_{rest}, where T_{read} is the time it takes to read the matrix from the hard drive to the CPU memory, $T_{transfer}$ is the time it takes to transfer matrix data from the CPU to the GPU memory, T_{mm_i} is the time it takes to conduct the i^{th} matrix matrix multiplication routine, and T_{rest} is the time it takes to conduct other operations within the GPU. For almost all input sizes, T_{rest} becomes trivial as it consists of less than 1% of T_{tot}.

From this initial implementation, we have made further optimizations in the CPU + GPU step 4 routine. First, we move the $j = 0$ C_{jb}^{P} file read routine and the $j = 0$ cudaMemcpy routine outside of its initial loop. Accordingly inside the loop, we can concurrently execute the first matrix matrix multiplication (i.e. Make $(ia|jb)$) in the GPU with the loading of the second $j = 1$ C_{jb}^{P} from the hard drive. This is possible because in CUDA, control is returned to the host (i.e. CPU) thread before the device (i.e. GPU) has completed its task, which allows programmers to overlap CPU work with GPU work. This feature comes in extremely handy especially when the GPU work is sufficiently long enough. Next, we switch the order in evaluation of P_{ca} and Γ_{ia}^{P} for a reason that will be explained subsequently. Because these two quantities are not dependent on one another, we can safely switch the order. Finally, we overlap evaluating P_{ca} with a copy routine that transfers the $j = 1$ C_{jb}^{P} from the CPU to the GPU, keeping in mind that this data was read from the file read routine that overlapped the first GPU matrix matrix multiplication. In order to conduct asynchronous copies, we have to use the CUDA driver API called cudaMemcpyAsync. We switched the order of the matrix matrix multiplications (P_{ca} and Γ_{ia}^{P} in order to avoid a data race condition that would have resulted from using the GPU data C_{jb}^{P} as both an input to a matrix matrix multiplication as well as a copied data from the CPU. It's important to note that in order to utilize cudaMemcpyAsync, we need to use page-locked host memory, which is a memory allocated on the host side via CUDA routine (e.g. cudaMallocHost). This memory should be conserved as too much usage results in overall degradation in performance. Figure 8 is a flowchart of the new CPU - GPU routine that summarizes the important algorithm. The portion of the pseudo-code only relevant to aforementioned discussion is shown here.

```
Loop over batches of active occupied orbitals, $ob$

  Load $C_{jb}^{P}$ $\forall$ $b$, for $j$=0 from disk
  move $C_{jb}^{P}$ $\forall$ $b$, for $j$=0 from CPU to GPU

  Loop over $j \in ob$
    Make $(ia|jb) = \Sigma_P (ia|P) C_{jb}^{P}$ $\forall$ $a$, $b$ (GPU)
    Load $C_{(j+1)b}^{P}$ $\forall$ $b$, given $j + 1$ from disk (CPU)
    Make $t_{ij}^{ab} = (ia||jb)/\Delta_{ij}^{ab}$ $\forall$ $a$, $b$ (GPU)
    Accumulate $t_{ij}^{ab}$ $\forall$ $a$, $b$, $j \in ob$, given $i$ (GPU)
    Increment $E_{RI-MP2} += \frac{1}{4} t_{ij}^{ab} (ia||jb)$ (GPU)
    Increment $\Gamma_{ia}^{P} += \Sigma_b t_{ij}^{ab} C_{jb}^{P}$ $\forall$ $a$, $P$, given $ij$ (GPU)
    Increment $P_{ca} += \Sigma_b t_{ij}^{ab} t_{ij}^{cb}$ $\forall$ $a$, $c$, given $ij$ (GPU)
    move $C_{(j+1)b}^{P}$ $\forall$ $b$, for $j + 1$ from CPU to GPU

  End Loop over $j \in ob$
```

Fig. 8. Step 4 CPU - GPU algorithm

At the end of our second implementation, the total wall time inside the loop for a given iteration is as follows: $T_{tot} \simeq$ max(T_{mm_1}, T_{read}) + T_{mm_2} + max(T_{mm_3}, $T_{transfer}$) + T_{rest}. We need not worry about the cost of initial T_{read} and $T_{transfer}$ for j = 0 case given that the total number of iteration inside the loop is large enough that this cost becomes negligible. As system size increases, the quintic matrix matrix multiplication calculations should dominate over the quartic I/O reads and transfers and accordingly, these costs will go away in principle. Unfortunately in Tesla/Turing, the lack of local scratch results in poor I/O performance ($100 - 150$ MB/sec in worst case) and subsequently, T_{read} becomes greater than T_{mm_1} for many of our input molecules. For relatively smaller molecules, the cache memory size is large enough that most of the data that has been read in the i^{th} (outermost loop) iteration is kept inside the cache, resulting in better I/O performance and $T_{mm_1} > T_{read}$. But for a system in which the size is not large enough such that the quintic computation does not dominate the wall time over the quartic I/O processes, the latter remains to be a problem on Tesla/Turing. One solution to combat for poor I/O performance is to conduct two different reads inside the loop with each of these reads loading one half of the matrix respectively. The second read can be overlapped with other the 2nd GPU routine inside the loop such that we can further hide the cost incurred by the CPU. Effectively, max(T_{mm_1}, T_{read}) + T_{mm_2} will become max(T_{mm_1}, T_{read_1}) + max(T_{mm_2}, T_{read_2}). There are some improvements in the performance numbers as file read is separated as such. We surmise that these problems will go away on the new Dirac cluster with improved I/O performance.

We can obtain a rough estimate and determine when T_{read} will be comparable to T_{mm_1} in a following way. The matrix C_{jb}^{P} has a dimension (NVirtbra, X), where NVirtbra = number of virtual orbitals and X = number of auxiliary basis functions. If we assign B to be the I/O bandwidth for a read operation in GB/sec, T_{read} = 10^9B / (8·NVirtbra·X). Furthermore, matrix $(ia|P)$ has dimension (NVirtbra, NVirtbra) and accordingly, the total number of FLOP in the matrix matrix multiplication is equal to 2(NVirtbra)(NVirtbra)(X). If we designate B_{flop} to be the matrix matrix multiplication GFLOPS, T_{mm_1} = $10^9 B_{flop}$ / (2·NVirtbra·NVirtbra·X). As a result, when $T_{read} = T_{mm_1}$, we have the following equality: $B = \frac{4B_{flop}}{NVirtbra}$. For mid to large size molecules such as glycine-8 and glycine-16 (58 and 115 atoms altogether), NVirtbra = 467 and 915 respectively. Given that our peak DGEMM numbers are around 75GFLOPS, we would need for input read bandwidth to be greater than 642MB/sec in glycine-8 and 327MB/sec in glycine-16 to avoid I/O being the bottleneck. For the new NVidia Fermi chips, the DGEMM performance expects to be larger and accordingly, we will need better I/O performance as well so that the I/O cost remains hidden.

Provided that we have excellent I/O available to us, there exists another additional room for improvement. The term $(ia|jb)$ represents the two-electron integral where indices

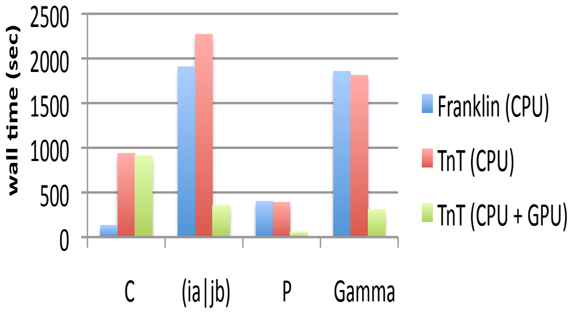


Fig. 9. Four main routines wall times in step 4 for glycine-8 molecule

represent virtual and active molecular orbitals. As such, $(ia|jb)$ is just a matrix transpose of $(ja|ib)$ and we can reduce the number of computation of these terms by half by storing the $(ia|jb)$ terms. Since these terms cannot be kept in memory due to their large size, we can overlap GPU routines with CPU routines that transfer data from the GPU to the CPU memory and finally to the hard drive to keep the CPU costs hidden. In practice, this can be done without incurring too much additional computational cost and would result in T_{mm_1} reducing to $0.5T_{mm_1}$. Unfortunately, our algorithm would require additional quartic I/O steps and exceed the cubic disk storage requirements, which might be problematic.

C. Results

For our results, we focus on the step 4 simulation time for glycine-8 molecule. We compare the results obtained from the Tesla and Turing (TnT) cluster with the ones obtained from the Franklin (Cray XT4) cluster, which consists of 2.3GHz single socket quad-core AMD Opterons. Specifically, we look at the four most time-consuming routines in step 4: loading C^P_{jb}, making $(ia|jb)$, making P_{ca}, and making Γ^P_{ia}.

From figure 9, we see that the I/O performance in Franklin is much better than in TnT (over 7 times faster) mostly due to existence of local scratch on Franklin. In the GPU routine, the load in C^P_{jb} is overlapped with the GPU making $(ia|jb)$ routine as mentioned in the previous section and thus, the bottleneck becomes the I/O CPU read in the case of glycine-8. Specifically, max(T_{read}, T_{mm_1}) = max(912.4, 357.8) = 912.4 seconds. Just by looking at the matrix matrix multiplication routines, there exists about 6 times improvement in the DGEMM performance going from CPU to GPU. The total step 4 wall time for simulations conducted on Franklin (CPU), TnT (CPU), and TnT (CPU + GPU) are 4945, 6542, and 1405 seconds respectively. The discrepancy in wall time between Franklin (CPU) and TnT (CPU) comes not ony from C^P_{jb} load but from other I/O routines not seen from 9. As it stands, there exists about 4.7x performance improvement in moving from CPU to the CPU + GPU routine on TnT. In the hypothetical situation where I/O bandwidth performance is as good in TnT as in Franklin, the CPU+GPU wall time would drop down to around 850 seconds, which would indicate around 5.8x improvement from the Franklin cluster and 7.7x improvement from the current TnT cluster.

In summary, we have accelerated the Q-Chem RI-MP2 code by utilizing both the GPU and the CPU. We identified step 4 as being the main bottleneck in the program and used concurrent file reads with GPU matrix matrix multiplications. We have also overlapped data transfer from the CPU memory to the GPU memory with additional GPU matrix matrix multiplications by using pinned memory. Overall in the TnT cluster, I/O file read times far exceed the matrix multiplication routines for mid to large size molecules. As seen from the results obtained from simulating glycine-8 on the Franklin cluster, which has local scratch, we expect the I/O read time to be cut to zero (due to the overlap with the GPU calculations) for clusters with better I/O.

IV. Fluid/Materials Application

A. The ALE-AMR fluid/solid mechanics application for material modeling

ALE-AMR is a new fluid/solid mechanics code that is used for modeling materials at a wide range of temperatures and densities [17]. This code solves the fluid equations with an anisotropic stress tensor on a structured adaptive mesh using an ALE (Arbitrary Lagrangian Eulerian) method combined with a structured dynamic adaptive mesh interface. Its basic method for combining ALE with AMR is based on an algorithm first suggested by Anderson and Pember [2]. Here, AMR stands for Adaptive Mesh Refinement. The structured adaptive mesh library provides much of the parallelism in ALE-AMR by dividing the work into patches that can be farmed out to various processors that communicate using MPI. Additional parallelism is provided by implicit solver libraries. How to best exploit the parallelism in these libraries is the major focus of this section.

The current version of ALE-AMR supports a variety of physics models that are introduced via operator splitting, and a new sophisticated algorithm for material failure and fragmentation. The code can model a variety of materials including plasmas, vapors, fluids, brittle and ductile solids, and the effective viscosity of most materials can be represented. Plans are underway to include surface tension effects. Most recently a new diffusion based model for heat conduction and radiation transport has been added to the code. The code is currently being used as a major component in the design of targets for the National Ignition Facility (NIF), which is the world largest laser. The code is also being applied to model experiments at the National Drift Compression Experiment (NDCX) in Berkeley and other high-energy facilities in France and Germany. Unlike the GTS code described earlier, this code does not already have OpenMP mixed into the MPI code. So the question for it is bifold: what ways are possible to speed up the code without changing the MPI parallel model and would the code benefit from a hybrid programming model such as MPI with OpenMP.

B. Diffusion Solver Speed-up

1) Introduction: Recent work on this code includes the development of heat conduction and radiation transport physics modules. These effects are important to many of the NIF target configurations that produce large temperature gradients in the target materials. Both of these physical effects are modeled using the diffusion equation which is discretized by a newly developed AMR capable Finite Element Method (FEM) solver [11]. The use of a FEM diffusion solver to model heat conduction and radiation transport is well studied [26] as is the integration of these physics modules into a hydrodynamic code [27]. However, the extension of these methods to AMR grids is novel, as such there are some interesting issues encountered in the parallel behavior of this approach.

In the following section we will give an introduction to the methods employed by the AMR capable diffusion solver recently introduced into ALE-AMR. This will be followed by a description of some parallel computation issues that we have recently experienced and an explanation of the approaches we used to debug these issues and improve the worst case performance drastically.

2) AMR Capable Diffusion Solver: To work with ALE-AMR a solver must be capable of operating on the multi-level, multi-processor, block structured, patch-based SAMRAI data representing the ALE-AMR field variables. The FEM, however, requires data in a single level composite mesh format. It is possible to use the SAMRAI data to form a fully connected composite mesh, however, this is not necessary. The hierarchical block structured nature of the SAMRAI data makes it possible to form a relatively simple mapping between the SAMRAI indices and the indices of a flattened composite mesh. This mapping can be formed without the need of creating and storing the composite mesh. The connectivity of most nodes in this mesh can be found trivially. The nodes and cells at coarse-fine interfaces, however, are significantly more complicated. Extra connectivity data about these special nodes and cells is stored to complete the composite mesh mapping.

At the beginning of an ALE-AMR simulation, the composite mesh mapping is formed on the initial grid. Whenever the grid changes through Lagrangian motion or AMR, the composite mesh mapping is updated to reflect the changed grid. Using this mesh mapping it is possible to obtain the global id numbers for all of the nodes in a given cell. However, the cells at the coarse-fine interfaces have extra nodes due to the refinement. Those extra nodes require basis functions to represent the solution within the cell and basis functions that maintain continuity across the coarse-fine interface are advantageous. We build on the transition element work found in [12] to create a family of elements suitable for our purposes.

Using the composite mesh mapping and this family of transition elements it is now possible to apply the FEM within the framework of ALE-AMR. We now turn our attention to the solution of the following diffusion equation.

$$\nabla \cdot \delta \nabla u + \sigma u = f \tag{1}$$

Applying the standard Galerkin approach yields the following linear system approximation

$$\begin{aligned} A\mathbf{u} + \mathbf{b} &= \mathbf{f} \\ A &= M_\sigma - K_\delta \\ (M_\alpha)_{ij} &= \textstyle\int_\Omega \alpha \phi_i \phi_j d\Omega \\ (K_\alpha)_{ij} &= \textstyle\int_\Omega \alpha \nabla\phi_i \cdot \nabla\phi_j d\Omega \\ \mathbf{b} &= 0 \end{aligned} \tag{2}$$

where M is the mass matrix, K is the stiffness matrix, and an insulating boundary yields $\mathbf{b} = 0$. The integrals are approximated over the elements with a family of mass lumping quadrature rules and the global mass and stiffness matrices are assembled using connectivity data obtained from the composite mesh mapping. We solve the resulting system of equations using the HYPRE [5] BiCG solver and the Euclid [14] preconditioner.

Both heat conduction and radiation transport can be modeled with relative ease using this diffusion solver. For heat conduction the equation can be time evolved implicitly by using the solver at each time step yielding

$$\begin{gathered} C_v \tfrac{T^{n+1} - T^n}{\Delta t} = \nabla \cdot D^n \nabla T^{n+1} - \alpha T^{n+1} \\ \delta = D^n,\ \sigma = -\alpha - \tfrac{C_v}{\Delta t} T^n,\ f = -\tfrac{C_v}{\Delta t} T^n \end{gathered} \tag{3}$$

where C_v is the specific heat, T is temperature represented at the nodes, D is the heat conductivity, and α is the absorptivity. The variables δ, σ, and f are the diffusion equation parameters from (1). Similarly the diffusion approximation to radiation transport can be implicitly time evolved yielding

$$\begin{aligned} \tfrac{E_R^{n+1} - E_R^n}{\Delta t} &= \nabla \cdot \lambda(\tfrac{c}{\kappa_r}) \nabla E_R^{n+1} + \tilde{\kappa}_p (B^n - c E_R^{n+1}) \\ C_v \tfrac{T^{n+1} - T^n}{\Delta t} &= -\tilde{\kappa}_p (B^n - c E_R^{n+1}) \\ \delta = \lambda(\tfrac{c}{\kappa_r}),\ \sigma &= -\tilde{\kappa}_p c - \tfrac{1}{\Delta t},\ f = -\tfrac{1}{\Delta t} - \tilde{\kappa}_p B^n \end{aligned} \tag{4}$$

where E_R is the radiation energy represented at the nodes, λ is a function used to impose flux limiting on the diffusion approximation, c is the speed of light, κ_r is the Rosseland opacity, $\tilde{\kappa}_P$ is a modification to Planck opacity which is used to linearize the equation as in [26], and B is the blackbody intensity.

3) Parallel Issues: This diffusion solver and accompanying physics modules have been put through a variety of unit tests, accuracy checks, validation studies, and performance analyses some of which can be found in [11]. The solver performs well in all of these tests, however, when employed to solve larger parallel problems in 3D, the solver performance often degrades to the point that it is unusable. To illustrate this problem we report some timing data we gathered while attempting to understand this problem on a $3D$ point explosion simulation with a uniform 2-level AMR mesh.

As this table shows, the solver performance can be reasonable in some situations as with the $27x27x27$ mesh, and be terrible in other situations as with the $81x81x81$ mesh. The performance also seems to be reasonable with only 1 CPU allocated to the problem, but becomes rapidly worse as more CPUs are added.

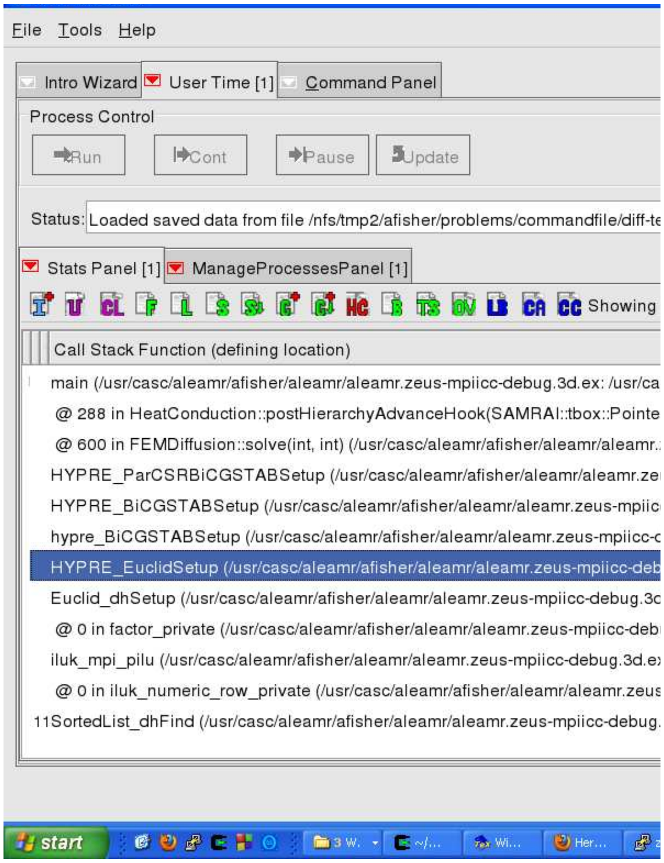
In order to better understand this problem, we use Open—SpeedShop, a performance analysis tool developed

	wall clock time (s)	
number of CPUs	27x27x27 mesh	81x81x81 mesh
1	21	73
2	15	420
4	9	816
8	7	960

TABLE II
WALL CLOCK TIMINGS OF THE ALE-AMR CODE SOLVING THE POINT EXPLOSION PROBLEM ON A 2-LEVEL AMR MESH. USING THE $27x27x27$ MESH, THE PERFORMANCE IS QUITE REASONABLE. HOWEVER, WHEN USING THE $81x81x81$ MESH WITH MORE THAN 1 CPU, PERFORMANCE IS SERIOUSLY DEGRADED.

by the Krell institute and recently installed at the NERSC facility. This tool instruments a code to gather data on how often a program is executing different areas of the code, as well as collecting data on characteristics like time spent in parallel communication. Using this data it is possible to gain insight into where a program is spending the most wall clock time and the resources that each part of the program consumes. Specifically, we ran ’usertime’ experiments in Open—SpeedShop and viewed the ’hot called path’ of the ALE-AMR both for normal and degraded performance. The hot call path is the call stack of the program that is most often encountered, and a good indicator of the code bottleneck. Below we provide hot call path data that we obtained from these experiments Figure 10. This data turns out to be quite illuminating, as it seems that in the degraded case the program is spending most of the wall clock time in HYPRE during the preconditioner formation. The normal case spends most of the time constructing and evaluating Jacobians which we expect to have a high computational cost in any FEM, and may be a future target for optimization in the ALE-AMR code.

These results suggest that we need to understand what is happening inside of HYPRE that is causing such performance degradation. Fortunately, HYPRE has some options that can give us a glimpse into how it is operating. We began enabling debug messages to get a better sense of what HYPRE is doing. This quickly told us that the solver iteration count was not changing significantly between the normal and degraded simulation cases. This implies that the time spent per HYPRE iteration is drastically different in the two cases. This leads us to consider the possibility that the systems being formed in the degraded case are in some way far more expensive to solve. In order to better understand this possibility we modified ALE-AMR to output the system A matrix and plot the sparsity pattern. We also set up Euclid to print out the matrix it is generating for the preconditioner. These sparsity plots show that the preconditioner matrix has a large amount of non-zero fill. This can be a manageable problem in serial since there is no communication to worry about. However, in parallel this large amount of non-zero fill can be devastating as many of the new non-zeros will require communication during the preconditioner formation.



Status: Loaded saved data from file /nfs/tmp2/afisher/problems/commandfile/diff-te

```
Call Stack Function (defining location)
main (/usr/casc/aleamr/afisher/aleamr/aleamr.zeus-mpiicc-debug.3d.ex: /usr/ca
 @ 288 in HeatConduction::postHierarchyAdvanceHook(SAMRAI::tbox::Pointe
 @ 600 in FEMDiffusion::solve(int, int) (/usr/casc/aleamr/afisher/aleamr/aleamr.
HYPRE_ParCSRBiCGSTABSetup (/usr/casc/aleamr/afisher/aleamr/aleamr.ze
HYPRE_BiCGSTABSetup (/usr/casc/aleamr/afisher/aleamr/aleamr.zeus-mpiic
hypre_BiCGSTABSetup (/usr/casc/aleamr/afisher/aleamr/aleamr.zeus-mpiicc-c
HYPRE_EuclidSetup (/usr/casc/aleamr/afisher/aleamr/aleamr.zeus-mpiicc-deb
Euclid_dhSetup (/usr/casc/aleamr/afisher/aleamr/aleamr.zeus-mpiicc-debug.3c
 @ 0 in factor_private (/usr/casc/aleamr/afisher/aleamr/aleamr.zeus-mpiicc-deb
iluk_mpi_pilu (/usr/casc/aleamr/afisher/aleamr/aleamr.zeus-mpiicc-debug.3d.ex
 @ 0 in iluk_numeric_row_private (/usr/casc/aleamr/afisher/aleamr/aleamr.zeus
11SortedList_dhFind (/usr/casc/aleamr/afisher/aleamr/aleamr.zeus-mpiicc-debug.
```

Fig. 10. Hot call path obtained by the Open—SpeedShop tool. This represents a case where the program is running with the degraded performance issue.

At this point, we must better understand how the Euclid preconditioner works in order to illuminate what is occurring in the degraded case. One method for the solution of a linear system with a matrix A is to decompose the matrix into lower and upper triangular parts L and U so that $A = LU$. Using this decomposition it is efficient to solve $LUx = b$ with a simple front and back solve technique. In $3D$ simulations the A matrix can be quite large with a sizable diagonal bandwidth, which makes computing and using this decomposition prohibitively expensive. This is due to the fact that in computing the LU decomposition, all of the zero values from the farthest diagonal band to the main diagonal will be filled with non-zeros, yielding L and U matrices with little sparsity.

The Euclid approach to this problem is to form Incomplete $\tilde{L}$ and $\tilde{U}$ (ILU) approximations that maintain some degree of sparsity. The simple front and back solve technique is then used to as the inversion operation of a preconditioner for A. This improves the conditioning of the matrix system thereby accelerating convergence to the solution. The trade-offs in this approach are between the computation cost of computing and applying the incomplete matrices and the rate of convergence to the solution. Generally, when $\tilde{L}$ and $\tilde{U}$ are closer to the actual L and U thus having less sparsity, the convergence is faster, but the cost of computing and applying $\tilde{L}$ and $\tilde{U}$ is higher.

It is now possible to consider remedies to the degraded

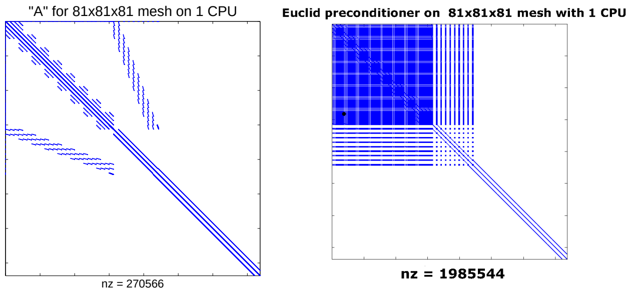


Fig. 11. Sparsity plots of the A matrix and the corresponding matrix created by Euclid for preconditioning. Notice the large amounts of additional non-zero entries in the preconditioner.

performance issue using this understanding of the ILU algorithm that is used by Euclid. The problem is caused by excessive non-zero fill in the degraded case, so altering the fill parameters in Euclid seems a fruitful path. As a first cut at this problem, we simply set Euclid to disallow any non-zero fill by using the 'level 0' option, forcing the sparsity of the $\tilde{L}\tilde{U}$ matrix to be the same as in the A matrix. This option may not be optimal in all cases as the preconditioner will be a more crude approximation to A and the HYPRE solver may need more iterations to converge. However, this approach should at least alleviate the excessive zero fill problem. To test this understanding, we re-ran the series of point explosion simulations on the $81x81x81$ 2-level AMR mesh that previously led to degraded performance.

num. CPU	wall clock time (s)
1	67
2	43
4	28
8	23

TABLE III
Wall clock timings of the ALE-AMR code solving the point explosion problem on an $81x81x81$ 2-level AMR mesh. In the multiprocessor case the runtime has been improved considerably (10x - 40x) by setting the Euclid preconditioner to avoid any non-zero fill.

This timing data shows that the degraded performance has been significantly improved and the problem now scales reasonably with the number of CPUs. Another run through Open—SpeedShop shows that the bottleneck in this case now resides in the Jacobian computation as was the case with non-degraded performance. These are both indicators that the this particular performance issue has been addressed, and barring any other issues, the diffusion solver is ready to run large $3D$ parallel simulations.

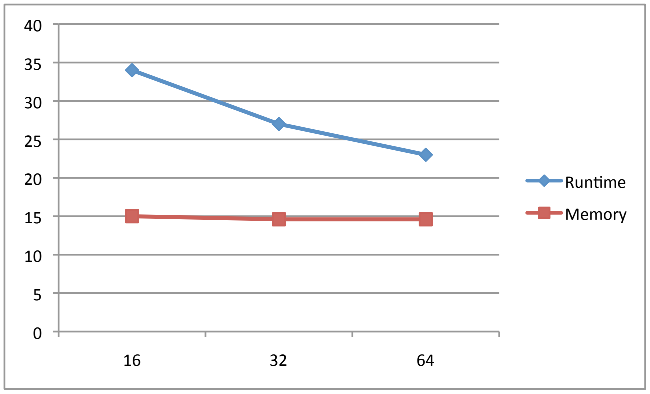


Fig. 12. Looking at a fixed number of MPI tasks on a varying number of processors shows the potential of running OpenMP on the idle cores.

C. Hybrid Parallelization

Adding an effective hybrid code model is not an easy task, and in this case consider what the benefits of such a model would be to ALE-AMR and how one would begin its implementation. One of the possible benefits, along with speed up from a hybrid model is memory consumption. A recent study, [8] in this proceedings shows that the memory reduction due to hybrid programming with MPI can be significant. This is likely to be more important for future architectural designs that have more memory limited cores. However, it is also known that sometimes adding a hybrid model to a code can actually slow the code down rather than improving the performance [7]. As part of choosing where to start adding hybrid code and to gauge its usefulness, we have performed the following simple experiments. We take some standard cases of running the ALE-AMR code with a fixed number of MPI tasks. We then look at how the code performs with the same number of MPI tasks, yet with more and more cores (or nodes). The idea is that if the code slows down as more cores are added, the OpenMP implementation would have to be extremely efficient to overcome the degradation. However, if the code actually speeds up when more nodes or cores (unused cores) are available, then this code is a good candidate for hybrid speedup. The hybrid benefit would then be a combination of the speedup attained by simply adding more cores (fixed number of MPI tasks) and the new OpenMP parallelization.

Adding the OpenMP hybrid model to an existing code can be a daunting task. Thus, we are exploring ways in which to make this process easier.

In considering utilizing shared memory parallelism in ALE-AMR, we first consider optimizing the SAMRAI (Structured Adaptive Mesh Refinement Infrastructure) software library. ALE-AMR utilizes SAMRAI for underlying functions of refinement/coarsening, load balancing, and MPI communication of mesh patch elements. While the overlying ALE-AMR codebase defines computationally intensive physics algorithms, it relies completely on SAMRAI for the interprocess tier, and its parallelization can yield considerable benefits to overall application performance.

D. The ROSE Compiler Framework

The ROSE tool kit [23] is a sophisticated and comprehensive infrastructure to create custom source-to-source translators developed at LLNL by Daniel J. Quinlan et al.. It provides mechanisms to translate input source code into an intermediate representation, called the Abstract Syntax Tree (AST), libraries to traverse and manipulate the information stored in the AST, as well as mechanisms to transform the altered AST information back into valid source code. The AST representation and the supporting data structures make exploiting knowledge of the architecture, parallel communication characteristics, and cache layout straightforward in the specification of transformations. Due to its efficient construction and (static) analysis capabilities of the intermediate representation, ROSE is especially well suited for analyzing large scale applications, which has been a central design goal for this compiler framework. In addition, ROSE is particularly well suited for building custom tools for program optimization, arbitrary program transformation, domain-specific optimizations, complex loop optimizations, performance analysis, software testing, OpenMP automatic parallelization and loop transformations, and (cyber-)security analysis. Further, a large number of program analyses and transformations have been developed for ROSE. They are designed to be utilized by users via simple function calls to interfaces. The program analyses available include call graph analysis, control flow analysis, data flow analysis (live variables, data dependence chain, reaching definition, alias analysis, etc.), class hierarchy analysis, data dependence and system dependence analysis. ROSE's automatic parallelization tool, autoPar, is capable of multithreading sequential C and C++ code by analyzing for-loops and amending them with OpenMP pragmas. autoPar operates on the source code build tree in place of the compiler, generating translated source files, and compiling and linking the executable.

E. First Autotuning attempts

Our initial attempts at automatically multithreading SAMRAI have been unsuccessful, and have uncovered several limitations in the current version of autoPar. The autoPar tool incorrectly translates class name and namespace scope resolution in SAMRAI's C++ code. This is not a complete surprise, especially considering that SAMRAI's more than 230 thousand lines of C++ code exploits many modern software design and implementation techniques. Since autoPar is an evolving part of ROSE, the ROSE development team has gladly accepted test cases resulting from these initial attempts, to further improve autoPar. Figure 13 shows the lines of code and languages in ALE-AMR and the SAMRAI library.

Using autoPar on SAMRAI is a reasonable starting point due to the fact that SAMRAI is a third-party software library to be used by client parallel applications. Designed to be general use code, it promises to be more easily parallelized than ALE-AMR. However, considering that SAMRAI is 230 thousand lines of C++ code while ALE-AMR is 130, it is worth investigating the possibility of using autoPar on ALE-AMR itself.

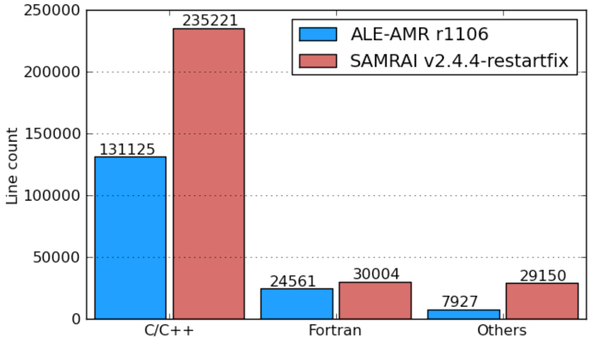


Fig. 13. Breakdown of the programming languages used in the ALE-AMR code and those used in the SAMRAI library for particular releases of the codes.

V. Conclusions

In this paper we show how significant performance improvement is possible on three different large application codes by a variety of techniques. We emphasize that these are real full application codes, and not reduced synthetic or otherwise adjusted benchmark codes. For the magnetic fusion code, GTS, we show that overlapping communication and computation is a very promising approach for a hybrid (MPI + OpenMP) code that is already optimized. For the quantum chemistry code, Q-Chem, we show the benefit of using GPU's for matrix matrix multiplications and overlapping data transfers from CPU memory to GPU memory with GPU computations. For the fluids/material science code, ALE-AMR, we show the importance of profiling matrix-solver libraries and studied options in adding threading (OpenMP) to this MPI-only code including issues associated with using an automated source-to-source translating compiler.

VI. Acknowledgments

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 (COE) at NERSC. Additionally, we are grateful for interactions with John Shalf, Mike Aamodt, and Nick Wright, and other members of the COE. Work by LLNL was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] N.S. Ostlund A. Szabo. *Modern Quantum Chemistry: An Introduction to Advanced Electronic Structure Theory*. Dover, 1989.

[2] R. W. Anderson, N. S. Elliott, and R. B. Pember. An arbitrary lagrange-eulerian method with adaptive mesh refinement for the solution of the euler equations. *J. Comput. Phys.*, 199(2):598–617, 2004.

[3] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009.

[4] Barbara M. Chapman, Lei Huang, Haoqiang Jin, Gabriele Jost, , and Bronis R. de Supinski. *In proceedings, W.E. Nagel et al. (Eds.): Euro-Par 2006*.

[5] E. Chow, A. J. Cleary, and R. D. Falgout. Design of the hypre preconditioner library. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (Yorktown Heights 1998)*, October 1998.

[6] R. Distasio, R. Steele, Y. Rhee, Y. Shao, and M. Head-Gordon. An improved algorithm for analytical gradient evaluation in resolution-of-the-identity second-order moller-plesset perturbation theory: aaplication to alanine tetrapeptide conformational analysis. *Journal of Computational Chemistry*, 28:839–856, 2006.

[7] A. E. Koniges et al. SC09 Tutorial.

[8] H. Shan et al. Analyzing the effect of different programming models upon performance and memory usage on cray xt5 platforms. This proceedings.

[9] S. Ethier, W. M. Tang, and Z. Lin. Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms. *Journal of Physics: Conference Series*, 16(1):1, 2005.

[10] 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):105–115, 2008.

[11] A. Fisher, D. Bailey, T. Kaiser, B. Gunney, N. Masters, A. Koniges, D. Eder, and R. Anderson. Modeling heat conduction and radiation transport with the diffusion equation in nif ale-amr. In *Proceedings of the Sixth International Conference on Inertial Fusion Sciences and Applications (San Francisco, 2009)*, September 2009.

[12] A. Gupta. A finite element for transition from a fine to a coarse grid. *International Journal for Numerical Methods in Engineering*, 12(1):35–45, 1978.

[13] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. Implementation and performance analysis of non-blocking collective operations for mpi. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.

[14] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM Journal of Scientific Computation*, 22(6):2194–2215, 2001.

[15] F. Jensen. *Introduction to Computational Chemistry*. Wiley, 1999.

[16] B. G. Johnson, C. A. Gonzales, P. M. W. Gill, and J. A. Pople. A density functional study of the simplest hydrogen abstraction reaction. effect of self-interaction correction. *Chemical Physics Letters*, 221:100–108, 1994.

[17] A. E. Koniges, N. D. Masters, A. C. Fisher, R. W. Anderson, D. C. Eder, T. B. Kaiser, D. S. Bailey, B. Gunney, P. Wang, B. Brown, K. Fisher, F. Hansen, B. R. Maddox, D. J. Benson, M. Meyers, and A. Geille. Ale-amr: A new 3d multi-physics code for modeling laser/target effects. *Journal of Physics*, (inpress), 2010.

[18] 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. *Physics of Plasmas*, 7(12):5013–5022, 2000.

[19] A. Komornicki M. Feyereisen, G. Fitzgerald. Use of approximate integrals in ab initio theory. an application in mp2 energy calculations. *Chemical Physics Letters*, 208:359–363, 1993.

[20] M.J. Frisch M. Head-Gordon, J.A. Pople. Mp2 energy evaluation by direct methods. *Chemical Physics Letters*, 153:503–506, 1988.

[21] Kamesh Madduri, Samuel Williams, Stéphane Ethier, Leonid Oliker, John Shalf, Erich Strohmaier, and Katherine 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*, pages 1–12, New York, NY, USA, 2009. ACM.

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

[23] Daniel J. Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[24] R. Rabenseifner and G. Wellein. *EWOMP 2002, Rome, Italy*.

[25] R. Rabenseifner and G. Wellein. Communication and optimization aspects of parallel programming models on hybrid architectures. *International Journal of High Performance Computing Applications*, 17(1), 2003.

[26] A. Shestakov, J. Harte, and D. Kershaw. Solution of the diffusion equation by finite elements in lagrangian hydrodynamic codes. *Journal of Computational Physics*, 76(2):385–413, 1988.

[27] A. Shestakov, J. Milovich, and M. Prasad. Combining cell and point centered methods in 3d, unstructured-grid radiation-hydrodynamic codes. *Journal of Computational Physics*, 170(1):81–111, 2001.

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

[29] L. Vogt, R. Olivares-Amaya, S. Kermes, Y. Shao, C. Amador-Bedolla, and A. Aspuru-Guzik. Accelerating resolution-of-the-identity second-order moller-plesset quantum chemistry calculations with graphical processing units. *Journal of Physical Chemistry A*, 112:2049–2057, 2008.

[30] 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.