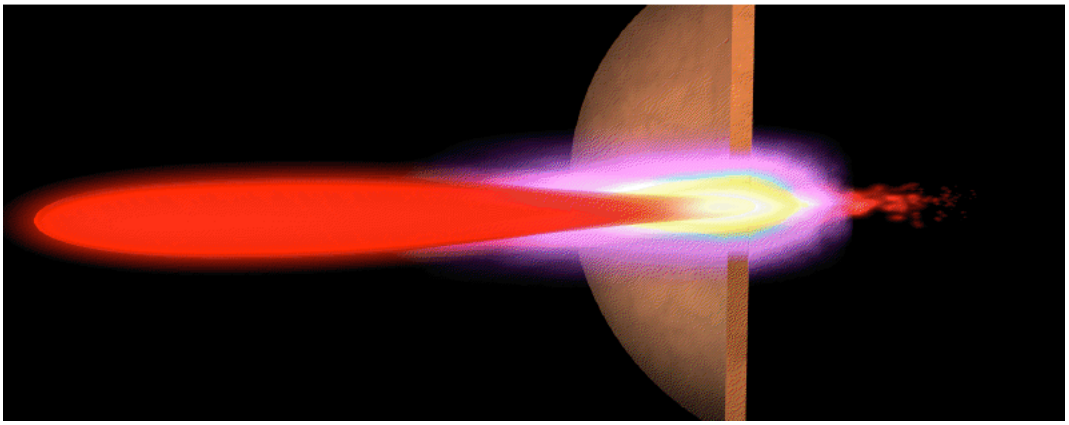
Computational Science and Engineering Petascale Initiative at LBNL



Progress Report

PI: Alice Koniges

June 28, 2010

Contents

1 Vision, Major Goals, Plan

The multi-core revolution provides opportunity for major increases in computational power over the next several years, if it can be harnessed. Effective use of multi-core platforms is essential for science problems that require petascale computing, but is also necessary to continue the performance growth associated with Moores Law at computing systems at all scales. This transition from simply increasing the single-processor and network performance to a different architectural paradigms forces application programmers to rethink the basic models of algorithm development and parallel programming from both the language and problem division standpoints. Additionally, advances in architecture and availability of resources are making the high performance computing (HPC) world more attractive to a variety of research areas that were not previously using advanced computing. Lawrence Berkeley National Laboratory (LBNL) is home to the NERSC Program, which serves the high-end computational needs of the Office of Science. As the Mission Computing Center for DOEs Office of Science, NERSC is tasked with helping over 3000 users to overcome the challenges of the multi-core revolution both through the use of new parallel constructs and languages and also by enabling a broader user community to take advantage of multi-core performance.

The Computational Science and Engineering Petascale Initiative at LBNL identifies key application areas with specific needs for advanced programming models, algorithms, and other support. The applications are chosen to be consistent with the current mission of the Department of Energy, with a particular focus on applications that benefit energy research, those supported by other ARRA (American Recovery and Reinvestment Act) funding and Energy Frontier Research Centers (EFRCs). The initiative pairs post-doctoral researchers with these high-impact projects at NERSC. The post-doctoral research program is centered at the Oakland Scientific Facility (OSF) where NERSC resides. Each post-doctoral researcher is assigned to one or more projects and works directly with the project scientists. The post-doctoral researchers funded under this program generally spend a third to half of their time at the OSF facility and the remainder their time embedded in the research project area, if the research project area scientists are located in commuting distance to the OSF. Alternatively, the post-docs travel for periods of time (1 - 3 weeks) to work directly at the research project facilities location. The post-docs are guided by NERSC staff experts in the computer science necessary for making these advances. The term of the post-doc assignments is generally 24 or 25 months.

The project started with funding at the beginning of the FY10. Subsequent recruiting efforts were begun to identify appropriate post-doctoral candidates and to pair them with one or more of the available projects. Since the start of funding, a total of 8 candidates have accepted positions. Since these are post-docs, some of them are finishing their thesis work. Thus even though they have been identified, their start dates will occur after completion of the requirements for the PhD. Thus as of the writing of this report, 4 of the 8 candidates have started work. In this document we focus primarily on the work of the candidates that have already started. We also give and indication of the focus of the remaining projects. Table 1 shows a summary of the project areas and the identified candidates.

An integral part of this program is to provide the post-doctoral researchers a productive environment for working with the application code teams. Despite the variety of application areas, there is considerable overlap in the methods and tools for achieving petascale application performance. Through interaction with the NERSC staff as well as the LBL computational science groups, considerable progress is already being made that has a significant benefit to NERSC staff and users. For example, material on hybrid programming options developed by one of the post-docs is already being included into tutorial presentations that will be given by NERSC staff.

Two poster presentations have already been made by the post-docs who have started work. Ad-

Petascale Hires and Projects

Specific Topic	Institute	Tentative PI-List	Candidate	More Details
Chemistry (EFRC)	UC Berkeley	Smit, Depaolo, Head-Gordon	Kim-HIRED/STARTED	Q-Chem parallelization
ITER, CS	PPPL	Tang, Ethier	Preissl-HIRED/STARTED	GTS for ITER-scale, Programming models
Inertial Fusion, Warm/Dense Matter	LBL/UCLA	Logan, Freidman, Bertozzi	Fagnan-HIRED/STARTED	ARRA-funded LBL NDCX-II modeling with ALE-AMR
Combustion EFRC	Princeton, LBL	Ju, Princeton, also Bell, Day LBL	Narayanyan-HIRED/Fall	EFRC modeling with AMR, DNS, and other methods
Nano-Control EFRC or Climate	LBL/LANL	Collela or Ng, LBL	Kavouklis-HIRED/Fall	Petascale Poisson Solver or Ice Sheet Model
CS/Bioinformatics	LBL, JGI	Strohmaier, LBL; Zhong Wang, JGI	Varbanescu-HIRED/Fall	Parallel Motifs (50%) Bio Liason (50%)
AMR Algorithms AutoParallel	LBL	Van Straalen, Chombo group, ATG	Offer in progress	Next generation AMR and source to source compilers
Accelerators, Next Generation Light Source	UC Berkeley	Wurtele	Austin-HIRED/STARTED	Modeling for Next Generation Light Source
Advanced Light Source	LBL	Marchesini	Maia-HIRED (80%)START July	Image Processing of ALS Data with GPUs
Earth Science	LBL	Newman	Maia-HIRED (20%)START July	Subsurface Geophysical Imaging with GPUs

Table 1: This table describes the projects and the post-doctoral assignments. In the case where a post-doc has not started, the project assignment may change depending on the time scale and other factors.

ditionally, these two post-docs made significant contributions to a paper presented internationally. This paper was awarded ”Best Paper,” at the Cray User’s group meeting in Edinburgh, Scotland in May 2010. Copies of the poster presentations and the ”best paper” are given in the appendices.

2 Chemistry EFRC Application

Many different scenarios have been written on achieving a substantial reduction in carbon emissions. In all these scenarios Carbon Capture and Sequestration (CCS) plays a significant role, as the predicted use of fossil fuels will continue to grow. There are two factors that determine the success of large-scale employment of CCS: (1) the uncertainties associated with the sequestration in geological formations and (2) the costs associated with carbon capture.

The Center for Gas Separations Relevant to Clean Energy Technologies is an EFRC headed by Prof. Berend Smit, and the focus of our first application area. One goal is to reduce energy costs associated with the separation of CO_2 from gas mixtures. The current technology has a parasitic energy of 30-40%, which implies a significant decrease in efficiency of power generation. Simple thermodynamic arguments show that the minimal parasitic energy to separate CO_2 from flue gasses is 3.5%. The chemical industry typically operates at 3-5 times the thermodynamic minimum, which suggests that the parasitic energy of carbon capture can be reduced by at least a factor of two. Figure 1 shows the graphical depiction of interaction between CO_2 molecules and metal-organic

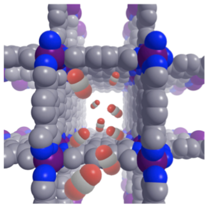


Figure 1: Carbon dioxide molecules represented by red-grey-red rods are inside a metal-organic framework, which selectively captures them

frameworks.

Q-Chem is a computational chemistry software package that is used for modeling related to the EFRC, for quantum chemistry calculations. Q-Chem includes Hartree-Fock (HF), density functional theory (DFT), coupled cluster (CC), configuration interaction, and Møller-Plesset perturbation theory. Many of these calculation methods provide researchers with the ability to accurately predict molecular equilibrium structures, through 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. This post-doctoral project is a collaboration with Prof. Martin Head-Gordon of UC Berkeley, the leader of the Q-Chem effort at Berkeley and the head of the EFRC, Prof. Berend Smit, also of UC Berkeley.

The calculation of interest is part of the second-order Møller-Plesset perturbation theory (MP2), which initially starts off with the mean-field HF approximation and treats the correlation energy via Rayleigh-Schrödinger perturbation theory to the second order. More specifically, we focus on a MP2 method that uses the resolution-of-the-identity (RI) approximation. The RI-approximation in 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 that can account for 80-90% of the correlation energy. Moreover, geometry optimizations using MP2 methods have generated equilibrium structures more reliable than HF, popular DFT alternatives. Unfortunately, there is 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 are exploring ways to cut down on the computational cost.

As an innovative solution, we are investigating using graphics processor units (GPU) to speed up the RI-MP2 energy gradient calculations. Compared to the traditional CPU (Central Processing Unit or standard processor), more transistors in GPUs are devoted to data processing as opposed to cache memory and flow control. Thus there is potential for massive parallelism within the GPUs for applications that can be easily formatted into the SIMD (single instruction, multiple data) instructions.

Portions of the CPU routine in this calculation have been converted to a CPU+GPU CUDA C routine. For our numerical simulations, we have used the new 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

Figure 2: Fermi GPU racks - Dirac cluster at NERSC

FX-5800 Quadro GPUs, with each GPU having 4GB of memory and 240 CUDA parallel processor cores. Recently the code has been ported to the new GPU cluster called Dirac, which consists of 44 Tesla C2050 Fermi GPUs. The racks for these GPUs can be seen in Figure 2.

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

Many of the details of this work are given in the paper in the appendix. In summary of this project to date, we have accelerated the Q-Chem RI-MP2 code by utilizing both the GPU and the CPU. This code is important in the research program of the Gas Separations EFRC. We identified a crucial step 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 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.

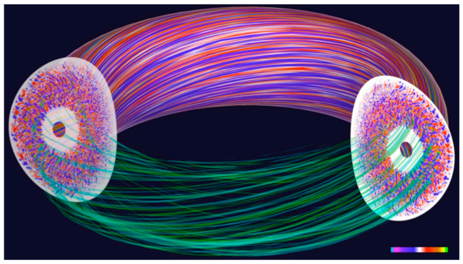


Figure 3: GTS' toroidal domain decomposition with magnetic field lines and density fluctuations

3 Fusion Application

Magnetic fusion as a viable energy source is finally on the horizon with the beginning of construction of the ITER facility and the planned follow-on demonstration fusion reactor (DEMO). From a computational science perspective, the goal of fusion simulation is to produce high-fidelity, full-device simulations to fully understand and predict the behavior of magnetically confined thermonuclear plasmas. The geometry of the device, and the nature and number of interacting species in a full-device calculation continue to make the problem one of utmost strain on computational resources.

3.1 GTS — A massively parallel magnetic fusion application

The fusion application chosen for the post-doc program is the Gyrokinetic Tokamak Simulation (GTS) code, 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. Our collaborators in this project are the research scientists in charge of this code effort, Drs. Stephane Ethier, and Weixing Wang, both of the Princeton Plasma Physics Laboratory. 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. 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. However, even with this relatively advanced programming model, the GTS code requires improvement and updates for new multicore architectures. It is also an excellent venue for trying out new programming models to determine if they can improve multicore performance. Finally, GTS requires updating to effectively model ITER scale reactors.

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 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).

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

3.2 The Parallel Model

The parallel model of GTS has three independent levels.: The first 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 Interface (MPI) to perform communication between the toroidal domains. In the second level of parallelism, particles can move from one domain to another while they travel around the torus. This *shift* phase is the one we have studied the most. Only nearest-neighbor communication in a circular fashion (using MPI_Sendrecv functionality) is used to move the particles between the toroidal domains. We note that the toroidal decomposition is limited to 64 or 128 planes, due to the long-wavelength physics being studied. 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. Using higher toroidal resolution only introduces more communication with no added benefit. Within each toroidal domain, we divide the particles between several MPI processes, and each process keeps a copy of the local grid, 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. A third level of parallelism is introduced by adding OpenMP compiler directives to heavily used (nested) loop regions. This hybrid programming model exploits the shared memory capabilities of many of today's HPC systems equipped with multicore CPUs. Although somewhat limited because of the single-threaded sections between OpenMP parallel loops and also due to NUMA effects arising from the shared memory regions, this method still affords additional speed-up for GTS. The initial work on this code through the Petascale initiative involves 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). Many details of this research are given in the paper in the appendix. These advanced aspects of parallel computing should be applicable to many massively parallel codes intended to run on HPC systems with multicore designs.

4 Neutralized Drift Compression Experiment (NDCX) Application

Construction is beginning on the second-generation Neutralized Drift Compression eXperiment (NDCX-II), a new high-current, modest-kinetic-energy accelerator at Lawrence Berkeley National Laboratory (LBNL). The machines ion beams will enable studies of the poorly understood "warm dense matter" (WDM) regime of temperatures around 10,000 K and densities near solid (as in the cores of giant planets). NDCX-II will also allow exploration of important issues in inertial-fusion target physics. These studies support the ultimate goal of using ion beams to heat deuterium/tritium fuel to ignition in a future inertial fusion energy power reactor (a role for which accelerators appear well suited). NDCX-II has received $11 million of funding from the American Recovery and Reinvestment Act. Construction began in July with completion of the initial 15-cell configuration anticipated in March 2012. Previous simulations of LBNL experiments did not include the effects of surface tension in the model. A scalable algorithm technology will be developed for this application that will allow for the inclusion of surface tension effects. This numerically challenging computational effort will focus on a new 3D simulation code, ALE-AMR. In addition, since the code is currently an MPI-only code additional parallelization options will be explored.

Currently at LBNL, fusion related experiments are studied using the NDCX-I ion accelerator that delivers a long-pulse beam (several microseconds) with a power density of $500(kW/cm)^2$ over a sampled spot size on the target of several hundred microns. With the addition of an imposed velocity tilt from an induction core, the NDCX-I can compress a portion of the long pulse to reach a power density of $25(MW/cm)^2$ over 2 nanoseconds. Under these conditions, the free-standing thin foil targets used in the experiments go through the melting and vaporization phases to reach temperatures up to 4000 K. Since the targets are thin foils of fractions of a micron in thickness we can model the target thermal dynamics using the equation of heat conduction for the temperature $T(x,t)$ as function of time and the spatial dimension along the beam direction; we also include cooling processes from energy flux from the surface of the foil due to evaporation, radiation, and thermionic (Richardson) emission.

During the first-generation NDCX-I experiments, the thin foils vaporized. Sensors indicate that the foil breaks up into droplets before vaporizing completely. There are two schools of thought on what causes the droplet formation, Rayleigh-Taylor Instability or the Marangoni effect which is driven by surface tension. The scientists at LBNL theorize that the Marangoni effect works on the timescale observed in the experiment, but the Rayleigh-Taylor instability would take too long. In order to verify the LBNL hypothesis, we must add higher order numerics to the ALE-AMR code to allow for the inclusion of surface tension. We have enlisted the expertise of applied mathematicians at the University of California, Los Angeles, (UCLA) to develop appropriate scalable algorithms. The second-generation Neutralized Drift Compression eXperiment (NDCX-II) will have a larger range of spatial and temporal scales and simulation will benefit from the use of adaptive mesh refinement (AMR), which is a central feature of ALE-AMR.

The ALE-AMR is a new fluid/solid mechanics code that combines AMR with Arbitrary Lagrangian Eulerian (ALE) hydrodynamics, developed by a team led by Alice Koniges. This code has the capacity to become a foremost modeling tool for the NDCX-II experiments, if appropriate scalable algorithms for handling the different material phases and surface tension and can be introduced. Traditionally, modelers for the NDCX project have not been NERSC users. In this post-doc project, we plan to make this code available to the NDCX project both through improved algorithms and high performance computing tuning. This code solves the fluid equations with an anisotropic stress tensor on a structured adaptive mesh using an ALE 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. 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. Most recently a new diffusion based model for heat conduction and radiation transport has been added to the code.

Figure 4 shows the result of an ALE-AMR calculation done where the surface of a thin metal foil (in red) on the left-hand side is heated by a laser. In this simulation, a hot plasma is ablated, which generates a strong shock that causes rear surface spalling of the foil. This figure also shows the different levels of mesh refinement in the lower half of the image. The darkest refinement regions correspond to the finest grids that are being generated to capture the spallation and plasma ablation. Similar simulations will be done in support of NNCX experiments where bulk heating of the foil is achieved by using ion beams. The proposed work of adding a surface tension model to the code will be important for these experiments.

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 an additional focus of this work.

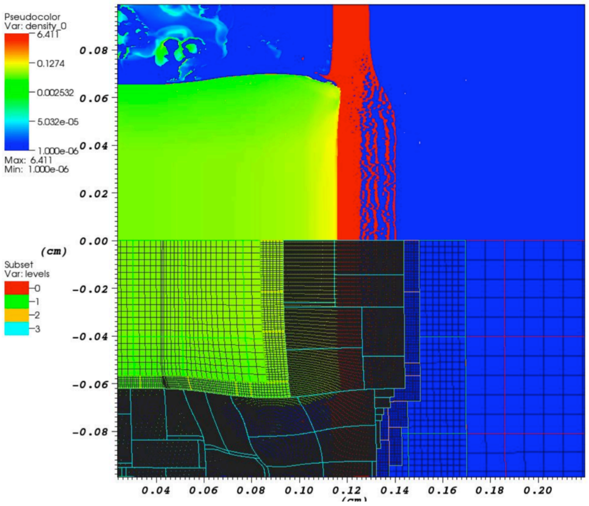


Figure 4: ALE-AMR simulation of a warm dense matter (WDM) metal foil target that has been heated on the left-hand side. Calculation shows the plasma ablation and spallation of the foil as well as the different levels of mesh refinement.

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.

5 Next Generation Advanced Light Source

LBL's Next Generation Light Source (NGLS) is a proposed facility for producing multiple tunable ultrafast (0.1 fs - 0.5 ps) laser pulses in the soft x-ray (1-100 nm) regime with bandwidths near the Fourier transform limit. These features of the NGLS will enable new experiments that cannot be done at other existing or proposed light sources. Unlike conventional x-ray sources, NGLS will provide a coherent X-ray laser, which can be used for single-shot X-ray diffraction experiments and nanoscale imaging an spectroscopy without encountering the 'phase problem' of traditional diffraction measurements. The availability of multiple tunable x-ray pulses permits two-color x-ray pump-x-ray probe experiments to track (via EXAFS and NEXAFS spectroscopies) energy transfer, charge separation and changes to the molecular timescale on picosecond and sub-picosecond timescales typical of these phenomena. The short pulse duration (100 as) will provide unprecedented time resolution, allowing direct observation of the dynamics of valence electrons that determine chemical reactivity.

Meeting the stringent design goals of the NGLS will require improvements to the electron gun, accelerator and free electron laser (FEL) undulator technologies that comprise the NGLS. The apparatus is illustrated in Figure 5. Simulation is essential to the development of these technologies because it is not feasible to construct a test device for each design change and simple models do not provide sufficient information about the brightness and emittance of the electron and laser beams. The goal of this postdoctoral project is to collaborate with Jonathan Wurtele and Ji Qiang from LBNL's Center for Beam Physics Theory Group to enhance the simulation tools being used to design the NGLS.

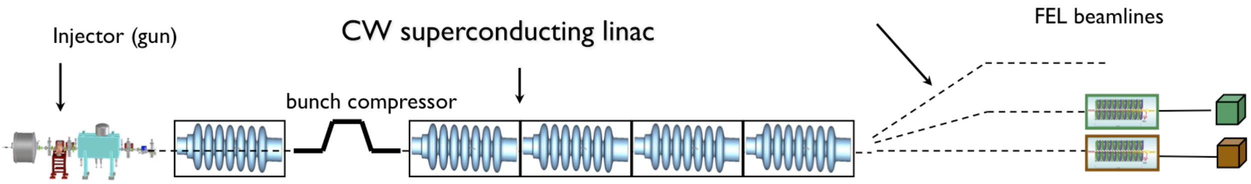


Figure 5: Diagram of NGLS apparatus.

Several codes are required to model the different components of the NGLS. The initial focus will be the Impact-T accelerator modeling code. Impact-T uses particle-in-cell (PIC) and integrated Green's function methods to solve Poisson's equation. The domain decomposition and particle field decomposition techniques used in Impact-T have been parallelized using MPI and Impact-T simulations that scale up to thousands of processors and billions of particles have already been demonstrated. Figure 6 shows the computed phase space for the electron beam.

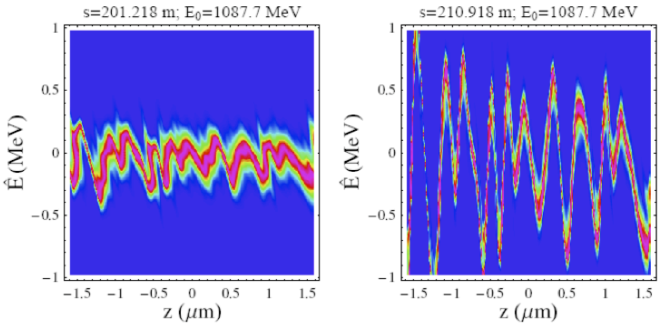


Figure 6: Longitudinal phase space diagrams computed with a Vlasov solver. The noise driven microbunching shown here is a source of instability in accelerators.

The first objective is to develop a parallel derivative free optimization algorithm that can be used in conjunction with the Impact-T accelerator modeling code to tune the NGLS accelerator design parameters. There is already a multitude of approaches to derivative free optimization; Our aim is to develop an optimization algorithm that combines the efficiency and robustness of Powell's BOBYQA trust-region approach with the computational scalability of the parallel direct search or particle-swarm approaches. The implementation of this optimization algorithm will enable a multi-level parallel strategy in which the parallel optimizer coordinates multiple, simultaneous parallel Impact-T runs, thereby providing an immediate route to using higher processor concurrencies. Because Impact-T can already run on several thousand cores, jobs using tens of thousands of cores are easily envisioned.

A second task, to be addressed in turn, is the design of an improved parallel solver for Poisson's equation, which contributes substantially to the simulation time for Impact-T. Specific strategies for addressing this interesting and worthwhile problem will be identified soon.

6 Upcoming Projects

In this next section we describe additional projects that have an identified post-doctoral researcher, but the post-doc has not yet started. Thus, after the post-doc arrives, the project will be better defined and work will begin. Because the post-doc has not actually started, the projects may take a different form once the researcher is on-site. Additionally, since the identified person accepting the post-doc must finish all degree requirements, there may be extenuating circumstances that will cause either a change in candidate, a change in project, or both.

6.1 Combustion EFRC

Recent concerns over energy sustainability calls for advanced engine technologies to achieve improved energy conversion efficiency and reduced emissions. These technologies include homogeneous charge compression ignition (HCCI) engines and low-temperature combustion (LTC) engines. However, the combustion and emission control of HCCI engines remain challenging because of the lack of fundamental understanding of the combustion regimes and ignition to detonation transition involving detailed chemistry with the negative temperature coefficient (NTC) combustion. Furthermore, the use of biofuels such as alcohols and biodiesel leads to new challenges in understanding HCCI combustion via kinetic coupling between hydroxyl and ester functional groups in biofuels with alkane molecules in conventional gasoline fuels.

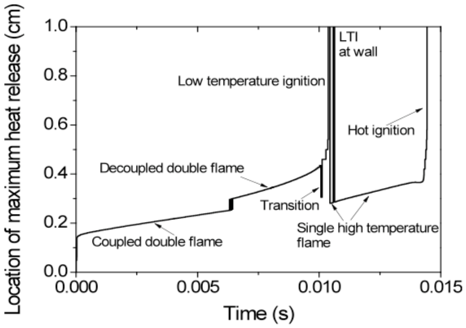


Figure 7: Transition of various flame regime of HCCI combustion of n-heptane/air mixtures at initial.

Understanding of combustion is an important area for enhancing our Nation's energy resources. As part of the Combustion Energy Frontier Research Center, a number of studies have been conducted to understand ignition and flame propagation in HCCI (Homogeneous Charge Compression Ignition) combustion using experiments and computations with simplified models. Different combustion regimes such as spontaneous ignition, flame deflagration, and detonation have been observed. The criteria for the occurrence of the different combustion regimes have been studied using hydrogen/air mixtures or global kinetic models. Recently, a direct simulation of one-dimensional HCCI combustion was conducted by the Princeton University and collaborators. Six different flame regimes were identified (Fig.7). Moreover, a singular dependence of the critical temperature gradient for ignition and acoustic wave coupling on initial temperature was identified near the NTC region. Unfortunately, the research is limited to one-dimensional laminar flow for n-heptane fuel only. The effects of concentration gradient, kinetic coupling between biofuel and alkanes, and turbulence-chemistry interaction on flame regimes, heat release rate, and ignition to detonation transition have not been examined. The post-doctoral researcher in this area will work to develop a parallel computing algorithm using a Dynamic Multi Time Scale (MTS) method integrated with adaptive chemistry, adaptive grids, and a second-order shock capturing scheme to enable direct numerical simulation of HCCI combustion for biofuel/gasoline surrogates using butanoate and h-heptane/iso-octane mixtures, and to investigate the impacts of thermal/concentration gradients, low temperature chemistry, kinetic coupling between biofuel and alkanes, and turbulence-chemistry interaction on combustion regimes, ignition to detonation transition, heat release rate, and unburned hydrocarbon emissions. The experimental and theoretical lead of this project is Prof. Yiguang Ju, of Princeton University. This post-doc will also work with members of the LBL AMR combustion research group led by Dr. John Bell to possibly incorporate AMR solution methods into this problem.

6.2 Biological Imaging with the Advanced Light Source

The Advanced Light Source (ALS), a division of Berkeley Lab, is a national user facility that generates intense light for scientific and technological research. As one of the world's brightest sources of ultraviolet and soft x-ray beams–and the world's first third-generation synchrotron light source in its energy range–the ALS makes previously impossible studies possible. Once specific problem in this area is using ALS for biological imaging. Important problems in this area include orientation, reconstruction, phasing and refinement. It may take on the oder of one million diffraction patterns per reconstruction. Several of the algorithms are highly compute intensive. The postdoctoral candidate identified for this project will work with Drs. C. Yang and S. Marchesini in developing a new class of orientation determination mehodes based on compressive sensing techniques that promise to outperform exisiting methods for nanocrystals.

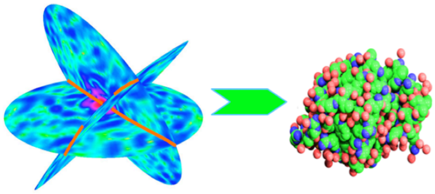


Figure 8: A full 3D diffraction data set is assembled from noisy diffraction patterns of identical particles in random and unknown orientations. Patterns are classified to group patterns of like orientation, oriented with respect to one another by correlation, and combined into a 3D reciprocal space. The image is then obtained by phase retrieval.

6.3 Subsurface Geophysical Imaging

EMGeo, the geophysical interpretation software for electromagnetic types of data developed by Drs. Gregory A. Newman and Michael Commer at Lawrence Berkeley National Laboratory, overcomes the technological problems involved in interpreting large geophysical data sets by exploiting 21st century computing powermassively parallel computing resourcesand combining that power with advanced electromagnetic measurement techniques, to provide a unique imaging capability, especially useful for hydrocarbon deposits. This capability has also proven very useful in solving other emerging problems, such as finding sources of alternative (specifically geothermal) energy and conducting optimally effective environmental remediation. Using sophisticated parallelization schemes, EMGeo can be scaled up to tens of thousands of computing processors, providing a unique advantage over comparable technologies in treating large-scale (industrial) 3D data sets (Commer and Newman, 2008). It enables investigators to see what is there, in the offshore subsurface, as never before, leading to new detection ability (both in area coverage and resolution), new efficiency, and new savings. In order to reduce typically high computing costs of large parallel computers and hence making EMGeo available to a broader range of users, the large potential of fast graphics computing hardware (GPUs) shall be exploited. In this post-doc project we will work on advanced implementations for GPU hardware. This development will ensure that EMGeo remains the technologically most advanced imaging package which is commercially available.

Subsurface geophysical imaging is

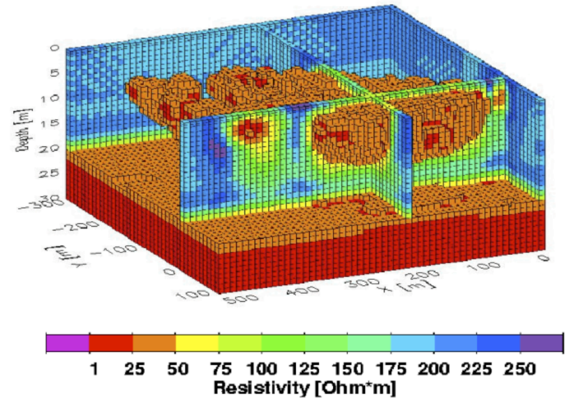


Figure 9: Three-dimensional resistivity image of a waste pit in Cologne, Germany, based on EMGeo technology. MT data acquired over the pit was imaged to show the pit extent and to verify that its waste products had not leached into the underlying groundwater and Rhine River.

6.4 Ice Sheet Modeling or Nano-Control EFRC

Beyond the popular media attention paid to global warming, a major risk to the environment is threats from abrupt climate change (ACC). The Intergovermental Panel on Climate Change (IPCC) has characterized several potential sources for abrupt change that would go beyond the threshold for societal intervention and adaptation. In response to these threats, the DOE has set up a program to study the mechanisms and risks of ACC. Fundamental to these studies is the use of a complicated system of computer models based on a detailed model of the Earth system known as the Community Climate System Model (CCSM). NERSC is a primary facility for users of the CCSM, which is a complicated set of computer models that span a variety of earth environments to create a long time period model of the earths climate. Recent improvements in the components have let to versions of the various models that are beginning to achieve petascale level performance on current architectures thus reducing the time to solution. However, as the machines include more and more cores, the continued scaling of these codes will require new software models that can handle the changes in the architectural environment.

Researchers at LBNL are pursuing an ice sheet model that is appropriate for a climate code, using the CHOMBO AMR framework. A post-doc candidate has been identified that has an appropriate background to work in this area. An alternative project for him involves the Non-Control EFRC. This is also work with the CHOMBO project. The candidate identified for this position will work on either one of two projects, or potentially both projects.

6.5 Parallel Motifs

Full applications can be immensely complicated pieces of code that may cloud our understanding of architecture, implementation, or programming models. Instead we can identify computational motifs that capture computation and communication patterns for a broad range of applications. This succinct set of well-defined algorithms and numerical methods at a high level of abstraction can be used to create minibenchmark codes from them that can drive future system design However, we will mostly specify the motifs independently of their actual implementations, so as to avoid optimizing for a particular implementation instead of specifying the general computational pattern.

Rather than engineering or evaluation benchmarks, motifs are better seen as challenging problems. The "problem" is to produce an efficient and scalable implementation of the specified numerical algorithm (e.g., a linear solver). Its "solution" includes code, which may require novel programming models and software infrastructure. We verify the solution through the problem's high-level mathematical definition. Motifs support evaluation of how classes of algorithms map onto architectures and programming paradigms. Thus, system architects can evaluate the performance and power impact of their design decisions through motifs. Similarly, we can evaluate or develop programming paradigms and supporting software such as operating systems, compilers, libraries, and runtime-systems based on them. Ultimately, a well-developed set of implementations will provide invaluable information for procurement. One of the post-docs projects is to work half-time in this area.

6.6 Algorithms for Biological Applications

Biological applications are varied, and biologist are starting to use the NERSC center more and more. One of the post-docs will also work half-time in this area. The goal will be to help the biological scientists with their computational needs and to introduce more high performance computing into this application area.

6.7 Next Generation Adaptive Mesh Refinement

The final area of study currently identified for post-doctoral research will be in the area of next generation AMR. The AMR method plays a role in in some of the post-doctoral projects, and different AMR libraries support different projects. The goal of this research will be to understand the differences in the different AMR implementations and to work on a next generation AMR that will use the best of the current packages and design new algorithms for multicore and upcoming architectures.

7 Personnel

Selecting a corps of post-docs who have demonstrated talent and expertise in their scientific domains and in high performance computing has been central to the early progress of the petascale initiative. Biographical information for the post-docs who have already started is given in this section.

7.1 Jihan Kim

Jihan Kim received his B.S. degree in University of California, Berkeley, and M.S. and Ph.D. in University of Illinois at Urbana-Champaign, all in electrical engineering. Under his advisor Prof. Jean-Pierre Leburton, Jihan conducted his research on numerical simulations of semiconductor quantum dots for quantum computers. He also did a summer internship at Sandia National Laboratory working on adding quantum effects to the device simulator code called Charon. Currently, Jihan works as a post-doctoral student at NERSC in Berkeley Lab and collaborates with Prof. Berend Smit and Prof. Martin Head-Gordon from UC Berkeley. His research focuses on using CUDA to accelerate quantum chemistry routines found in the computational chemistry package, Q-Chem.

7.2 Brian Austin

Brian Austin earned a B.A. in Chemistry from Reed College and a Ph.D. in Chemistry from the University of California, Berkeley. Brian's graduate research extended the feasibility of applying quantum Monte Carlo methods to large molecules by introducing several algorithms for linear-scaling wave function evaluation, wave function optimization, dynamic load balancing and parallel I/O. He also developed a more rigorous QMC treatment of electronic excited states Now a postdoctoral student at NERSC, Brian collaborates with LBNL's Jonathan Wurtele and Ji Qiang to simulate the Next Generation Light Source. His immediate goal is to develop a parallel derivative-free optimization library that can be coupled to parallel function evaluation code such as Ji Qiang's accelerator code, Impact-T, to create a multi-level parallel model.

7.3 Robert Preissl

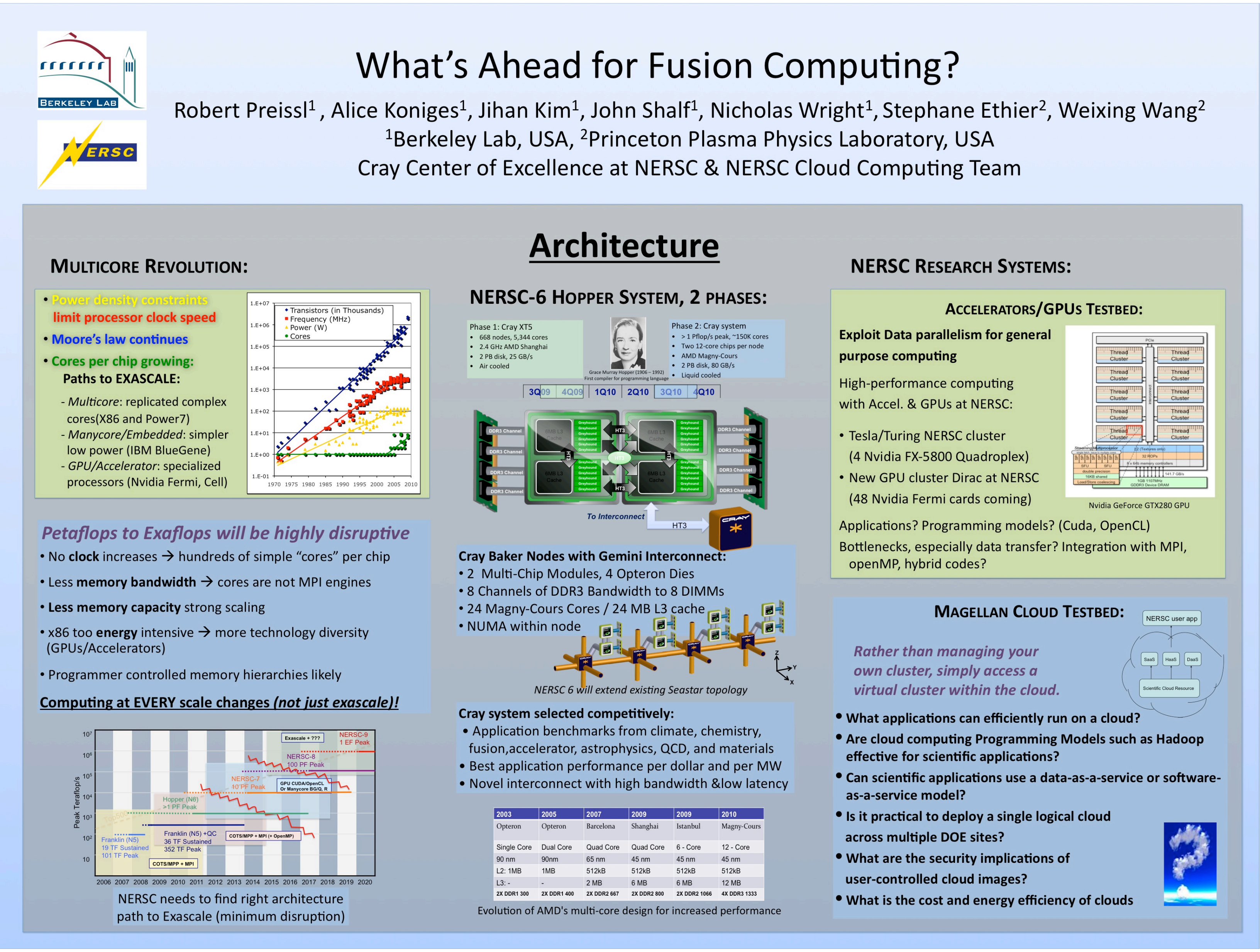
Robert Preissl completed his M.Sc. Degree in Applied Mathematics and earned a Ph.D. Degree in Computer Science at Johannes Kepler University of Linz, Austria. While pursuing his doctorate, Robert Preissl contributed to research on automated source-to-source transformation and program analysis of MPI parallel applications in collaboration with the Lawrence Livermore National Laboratory. At LBNL's NERSC division, Robert Preissl's post-doctoral studies deal with analyzing and optimizing parallel magnetic fusion simulations in cooperation with the Princeton Plasma Physics Laboratory. The main goal is to determine an optimal parallel programming paradigm and new algorithms to achieve Petascale performance which is crucial for the accuracy and precision of magnetic fusion simulations.

7.4 Kirsten Fagnan

Kirsten Fagnan received her Ph.D. in Applied Mathematics from the University of Washington, Seattle, for her thesis, ”High-resolution Finite Volume Methods for Exracorporeal Shock Wave Therapy” in March, 2010. Her background is in scientific computing, numerical analysis and partial differential equations. The work in her thesis has ties to the algorithms developed in the ALE-AMR code both in the form of the equations, as well as the use of Berger-Oliger style adaptive mesh refinement. Therefore, the NDCX modeling project is a good fit for her qualifications. As a graduate student, Fagnan ran calculations on the TeraGrid cluster and has experience with parallelization of scientific codes.

A Appendix: Posters

Poster presentations have already been prepared and given by post-doctoral researchers Jihan Kim and Robert Priessl in important conferences in their repective application areas. Miniature copies of the posters are given in this appendix. Jihan Kim made his presentation at MQM 2010 (Molecular Quantum Mechanics), and Robert Priessl made his presentation at the Sherwood Fusion Theory Conference.



What's Ahead for Fusion Computing?

Robert Preissl[1] , Alice Koniges[1], Jihan Kim[1], John Shalf[1], Nicholas Wright[1], Stephane Ethier[2], Weixing Wang[2]
[1]Berkeley Lab, USA, [2]Princeton Plasma Physics Laboratory, USA
Cray Center of Excellence at NERSC & NERSC Cloud Computing Team

Architecture

MULTICORE REVOLUTION:

- Power density constraints limit processor clock speed
- Moore's law continues
- Cores per chip growing:

Paths to EXASCALE:

- *Multicore*: replicated complex cores(X86 and Power7)
- *Manycore/Embedded*: simpler low power (IBM BlueGene)
- *GPU/Accelerator*: specialized processors (Nvidia Fermi, Cell)

Petaflops to Exaflops will be highly disruptive

- No **clock** increases → hundreds of simple "cores" per chip
- Less **memory bandwidth** → cores are not MPI engines
- **Less memory capacity** strong scaling
- x86 too **energy** intensive → more technology diversity (GPUs/Accelerators)
- Programmer controlled memory hierarchies likely

Computing at EVERY scale changes *(not just exascale)*!

NERSC needs to find right architecture path to Exascale (minimum disruption)

NERSC-6 HOPPER SYSTEM, 2 PHASES:

Phase 1: Cray XT5

- 668 nodes, 5,344 cores
- 2.4 GHz AMD Shanghai
- 2 PB disk, 25 GB/s
- Air cooled

Grace Murray Hopper (1906 – 1992)
First compiler for programming language

Phase 2: Cray system

- > 1 Pflop/s peak, ~150K cores
- Two 12-core chips per node
- AMD Magny-Cours
- 2 PB disk, 80 GB/s
- Liquid cooled

Cray Baker Nodes with Gemini Interconnect:

- 2 Multi-Chip Modules, 4 Opteron Dies
- 8 Channels of DDR3 Bandwidth to 8 DIMMs
- 24 Magny-Cours Cores / 24 MB L3 cache
- NUMA within node

NERSC 6 will extend existing Seastar topology

Cray system selected competitively:

- Application benchmarks from climate, chemistry, fusion,accelerator, astrophysics, QCD, and materials
- Best application performance per dollar and per MW
- Novel interconnect with high bandwidth &low latency

2003	2005	2007	2009	2009	2010
Opteron	Opteron	Barcelona	Shanghai	Istanbul	Magny-Cours
Single Core	Dual Core	Quad Core	Quad Core	6 - Core	12 - Core
90 nm	90nm	65 nm	45 nm	45 nm	45 nm
L2: 1MB	1MB	512kB	512kB	512kB	512kB
L3: -	-	2 MB	6 MB	6 MB	12 MB
2X DDR1 300	2X DDR1 400	2X DDR2 667	2X DDR2 800	2X DDR2 1066	4X DDR3 1333

Evolution of AMD's multi-core design for increased performance

NERSC RESEARCH SYSTEMS:

ACCELERATORS/GPUS TESTBED:

Exploit Data parallelism for general purpose computing

High-performance computing with Accel. & GPUs at NERSC:

- Tesla/Turing NERSC cluster (4 Nvidia FX-5800 Quadroplex)
- New GPU cluster Dirac at NERSC (48 Nvidia Fermi cards coming)

Nvidia GeForce GTX280 GPU

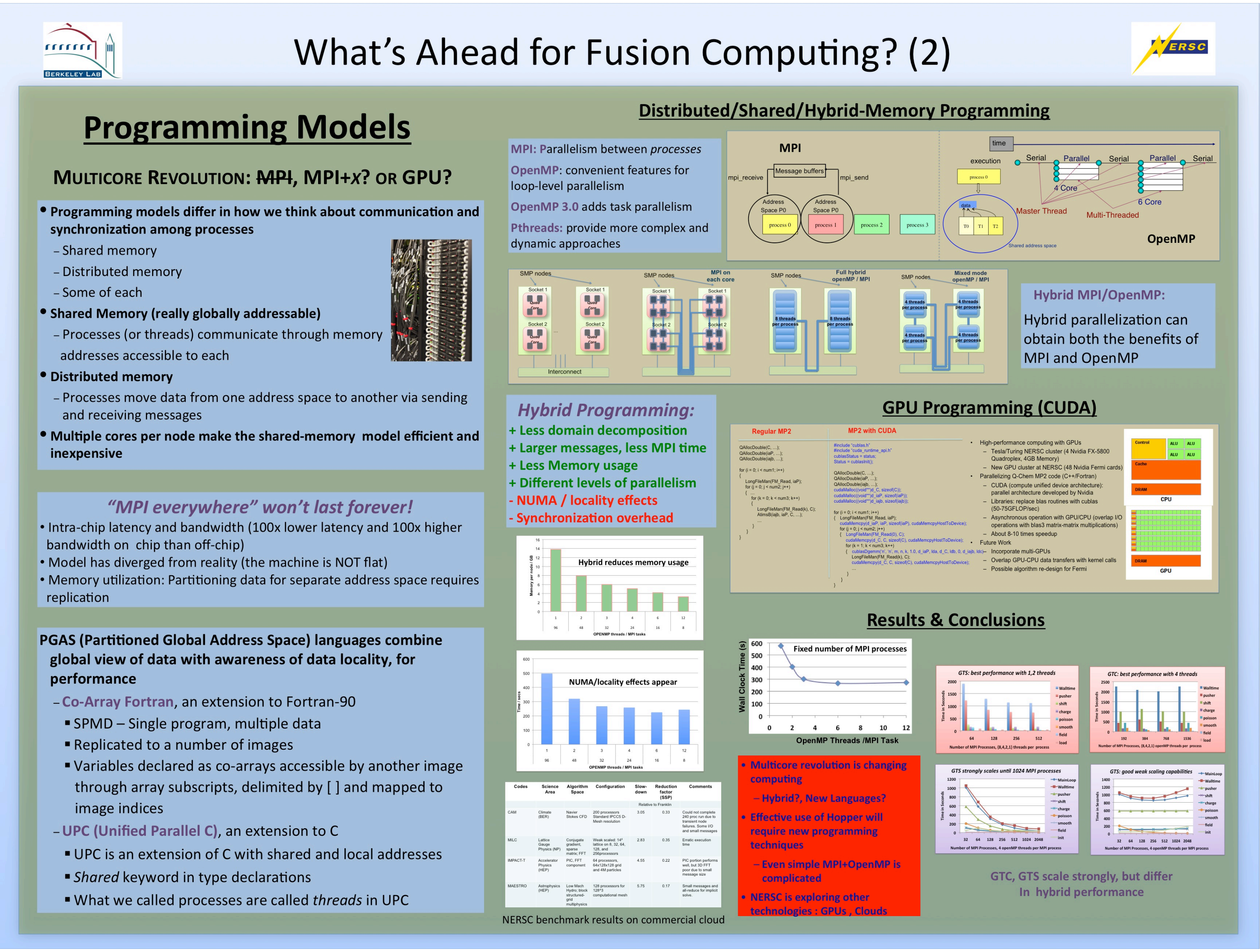
Applications? Programming models? (Cuda, OpenCL)

Bottlenecks, especially data transfer? Integration with MPI, openMP, hybrid codes?

MAGELLAN CLOUD TESTBED:

Rather than managing your own cluster, simply access a virtual cluster within the cloud.

- **What applications can efficiently run on a cloud?**
- **Are cloud computing Programming Models such as Hadoop effective for scientific applications?**
- **Can scientific applications use a data-as-a-service or software-as-a-service model?**
- **Is it practical to deploy a single logical cloud across multiple DOE sites?**
- **What are the security implications of user-controlled cloud images?**
- **What is the cost and energy efficiency of clouds**



What's Ahead for Fusion Computing? (2)

Programming Models

MULTICORE REVOLUTION: ~~MPI~~, MPI+*X*? OR GPU?

- **Programming models differ in how we think about communication and synchronization among processes**
  - Shared memory
  - Distributed memory
  - Some of each
- **Shared Memory (really globally addressable)**
  - Processes (or threads) communicate through memory addresses accessible to each
- **Distributed memory**
  - Processes move data from one address space to another via sending and receiving messages
- **Multiple cores per node make the shared-memory model efficient and inexpensive**

"MPI everywhere" won't last forever!

- Intra-chip latency and bandwidth (100x lower latency and 100x higher bandwidth on chip than off-chip)
- Model has diverged from reality (the machine is NOT flat)
- Memory utilization: Partitioning data for separate address space requires replication

PGAS (Partitioned Global Address Space) languages combine global view of data with awareness of data locality, for performance

- **Co-Array Fortran**, an extension to Fortran-90
  - SPMD – Single program, multiple data
  - Replicated to a number of images
  - Variables declared as co-arrays accessible by another image through array subscripts, delimited by [ ] and mapped to image indices
- **UPC (Unified Parallel C)**, an extension to C
  - UPC is an extension of C with shared and local addresses
  - *Shared* keyword in type declarations
  - What we called processes are called *threads* in UPC

Distributed/Shared/Hybrid-Memory Programming

MPI: Parallelism between *processes*

OpenMP: convenient features for loop-level parallelism

OpenMP 3.0 adds task parallelism

Pthreads: provide more complex and dynamic approaches

Hybrid MPI/OpenMP:

Hybrid parallelization can obtain both the benefits of MPI and OpenMP

Hybrid Programming:

+ Less domain decomposition
+ Larger messages, less MPI time
+ Less Memory usage
+ Different levels of parallelism
- NUMA / locality effects
- Synchronization overhead

Hybrid reduces memory usage
Memory per node / GB
0
2
4
6
8
10
12
14
16
1
2
3
4
6
12
96
48
32
24
16
8
OPENMP threads / MPI tasks
NUMA/locality effects appear
Time / secs
0
100
200
300
400
500
600
1
2
3
4
6
12
96
48
32
24
16
8
OPENMP threads / MPI tasks

Codes	Science Area	Algorithm Space	Configuration	Slow-down	Reduction factor (SSP)	Comments
				Relative to Franklin		
CAM	Climate (BER)	Navier Stokes CFD	200 processors Standard IPCC5 D-Mesh resolution	3.05	0.33	Could not complete 240 proc run due to transient node failures. Some I/O and small messages
MILC	Lattice Gauge Physics (NP)	Conjugate gradient, sparse matrix; FFT	Weak scaled: 14^4 lattice on 8, 32, 64, 128, and 256processors	2.83	0.35	Erratic execution time
IMPACT-T	Accelerator Physics (HEP)	PIC, FFT component	64 processors, 64x128x128 grid and 4M particles	4.55	0.22	PIC portion performs well, but 3D FFT poor due to small message size
MAESTRO	Astrophysics (HEP)	Low Mach Hydro; block structured-grid multiphysics	128 processors for 128^3 computational mesh	5.75	0.17	Small messages and all-reduce for implicit solve.

NERSC benchmark results on commercial cloud

GPU Programming (CUDA)

Regular MP2

```
QAllocDouble(C, …);
QAllocDouble(iaP, …);
QAllocDouble(iajb, …);

for (i = 0; i < num1; i++)
{
   LongFileMan(FM_Read, iaP);
   for (j = 0; j < num2; j++)
   {  …
      for (k = 0; k < num3; k++)
      {
         LongFileMan(FM_Read(k), C);
         AtimsB(iajb, iaP, C, …);
         …
      }
   }
}
```

MP2 with CUDA

```
#include "cublas.h"
#include "cuda_runtime_api.h"
cublasStatus = status;
Status = cublasInit();

QAllocDouble(C, …);
QAllocDouble(iaP, …);
QAllocDouble(iajb, …);
cudaMalloc((void**)d_C, sizeof(C));
cudaMalloc((void**)d_iaP, sizeof(iaP));
cudaMalloc((void**)d_iajb, sizeof(iajb));

for (i = 0; i < num1; i++)
{   LongFileMan(FM_Read, iaP);
    cudaMemcpy(d_iaP, iaP, sizeof(iaP), cudaMemcpyHostToDevice);
    for (j = 0; j < num2; j++)
    {   LongFileMan(FM_Read(0), C);
        cudaMemcpy(d_C, C, sizeof(C), cudaMemcpyHostToDevice);
        for (k = 1; k < num3; k++)
        {   cublasDgemm('n', 'n', m, n, k, 1.0, d_iaP, lda, d_C, ldb, 0, d_iajb, ldc);
            LongFileMan(FM_Read(k), C);
            cudaMemcpy(d_C, C, sizeof(C), cudaMemcpyHostToDevice);
            …
        }
    }
}
```

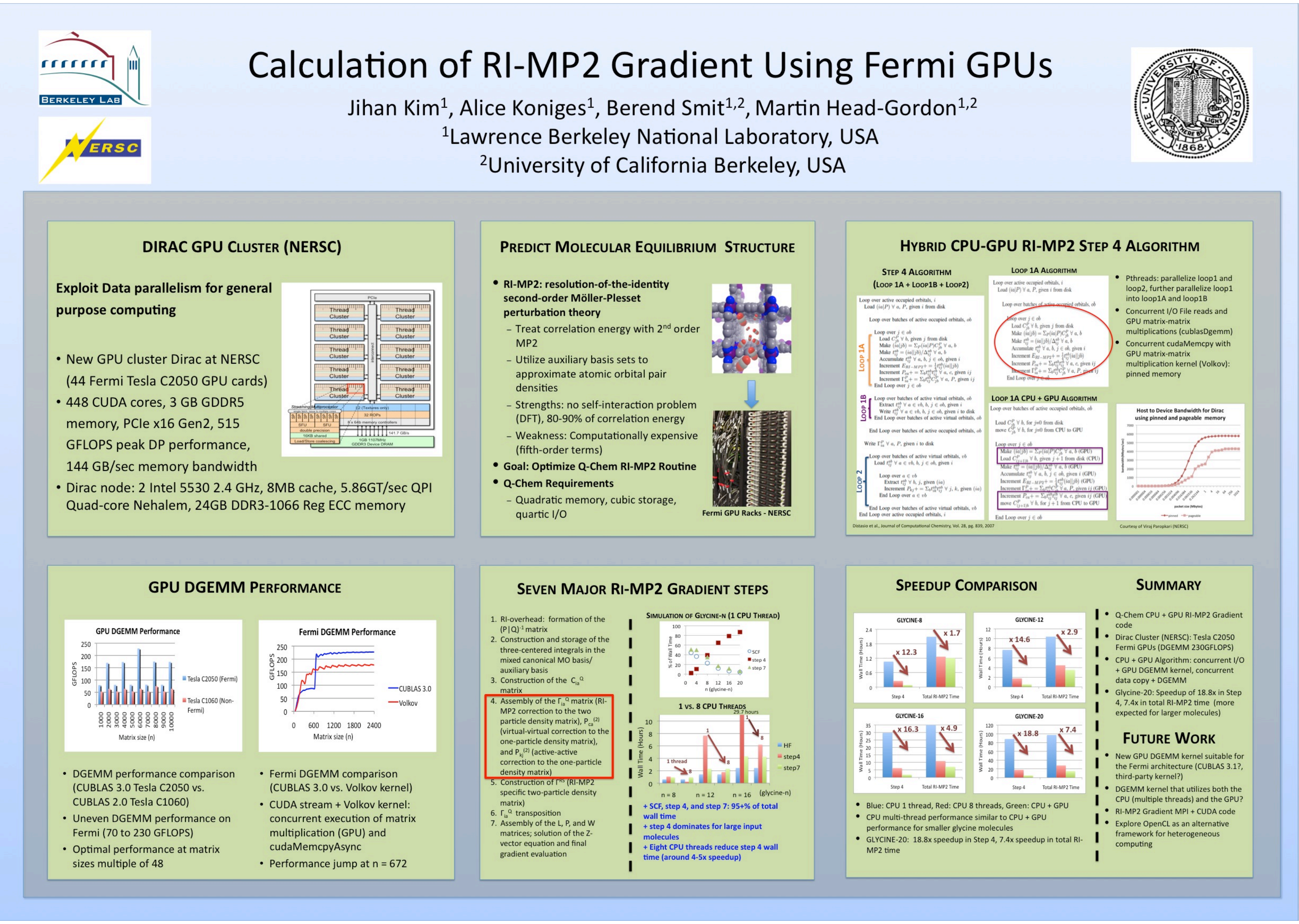
- High-performance computing with GPUs
  - Tesla/Turing NERSC cluster (4 Nvidia FX-5800 Quadroplex, 4GB Memory)
  - New GPU cluster at NERSC (48 Nvidia Fermi cards)
- Parallelizing Q-Chem MP2 code (C++/Fortran)
  - CUDA (compute unified device architecture): parallel architecture developed by Nvidia
  - Libraries: replace blas routines with cublas (50-75GFLOP/sec)
  - Asynchronous operation with GPU/CPU (overlap I/O operations with blas3 matrix-matrix multiplications)
  - About 8-10 times speedup
- Future Work
  - Incorporate multi-GPUs
  - Overlap GPU-CPU data transfers with kernel calls
  - Possible algorithm re-design for Fermi

Results & Conclusions

Fixed number of MPI processes
Wall Clock Time (s)
0
100
200
300
400
500
600
0
2
4
6
8
10
12
OpenMP Threads /MPI Task
GTS: best performance with 1,2 threads
GTC: best performance with 4 threads
GTS strongly scales until 1024 MPI processes
GTS: good weak scaling capabilities
Time in Seconds
Walltime
pusher
shift
charge
poisson
smooth
field
load
MainLoop
init
Number of MPI Processes, (8,4,2,1) threads per process
Number of MPI Processes, (8,4,2,1) openMP threads per process
Number of MPI Processes, 4 openMP threads per MPI process

- **Multicore revolution is changing computing**
  - **Hybrid?, New Languages?**
- **Effective use of Hopper will require new programming techniques**
  - **Even simple MPI+OpenMP is complicated**
- **NERSC is exploring other technologies : GPUs , Clouds**

GTC, GTS scale strongly, but differ In hybrid performance



Calculation of RI-MP2 Gradient Using Fermi GPUs

Jihan Kim[1], Alice Koniges[1], Berend Smit[1,2], Martin Head-Gordon[1,2]
[1]Lawrence Berkeley National Laboratory, USA
[2]University of California Berkeley, USA

DIRAC GPU CLUSTER (NERSC)

Exploit Data parallelism for general purpose computing

PCIe
Thread Cluster
Interconnect
Streaming Multiprocessor
SP
SFU
double precision
16KB shared
Load/Store coalescing
L2 (Textures only)
32 ROPs
8 x 64b memory controllers
141.7 GB/s
1GB 1107MHz GDDR3 Device DRAM

- New GPU cluster Dirac at NERSC (44 Fermi Tesla C2050 GPU cards)
- 448 CUDA cores, 3 GB GDDR5 memory, PCIe x16 Gen2, 515 GFLOPS peak DP performance, 144 GB/sec memory bandwidth
- Dirac node: 2 Intel 5530 2.4 GHz, 8MB cache, 5.86GT/sec QPI Quad-core Nehalem, 24GB DDR3-1066 Reg ECC memory

PREDICT MOLECULAR EQUILIBRIUM STRUCTURE

- **RI-MP2: resolution-of-the-identity second-order Möller-Plesset perturbation theory**
  - Treat correlation energy with 2nd order MP2
  - Utilize auxiliary basis sets to approximate atomic orbital pair densities
  - Strengths: no self-interaction problem (DFT), 80-90% of correlation energy
  - Weakness: Computationally expensive (fifth-order terms)
- **Goal: Optimize Q-Chem RI-MP2 Routine**
- **Q-Chem Requirements**
  - Quadratic memory, cubic storage, quartic I/O

Fermi GPU Racks - NERSC

HYBRID CPU-GPU RI-MP2 STEP 4 ALGORITHM

STEP 4 ALGORITHM (LOOP 1A + LOOP1B + LOOP2)

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

LOOP 1B
      Loop over batches of active virtual orbitals, $ob$
        Extract $t^{ab}_{ij}\ \forall a \in vb, b, j \in ob$, given $i$
        Write $t^{ab}_{ij}\ \forall a \in vb, b, j \in ob$, given $i$ to disk
      End Loop over batches of active virtual orbitals, $ob$

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

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

LOOP 2
    Loop over batches of active virtual orbitals, $vb$
      Load $t^{ab}_{ij}\ \forall a \in vb, b, j \in ob$, given $i$

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

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

Distasio et al., Journal of Computational Chemistry, Vol. 28, pg. 839, 2007

LOOP 1A ALGORITHM

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

LOOP 1A CPU + GPU ALGORITHM

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

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

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

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

- Pthreads: parallelize loop1 and loop2, further parallelize loop1 into loop1A and loop1B
- Concurrent I/O File reads and GPU matrix-matrix multiplications (cublasDgemm)
- Concurrent cudaMemcpy with GPU matrix-matrix multiplication kernel (Volkov): pinned memory

Courtesy of Viraj Paropkari (NERSC)

GPU DGEMM PERFORMANCE

- DGEMM performance comparison (CUBLAS 3.0 Tesla C2050 vs. CUBLAS 2.0 Tesla C1060)
- Uneven DGEMM performance on Fermi (70 to 230 GFLOPS)
- Optimal performance at matrix sizes multiple of 48

- Fermi DGEMM comparison (CUBLAS 3.0 vs. Volkov kernel)
- CUDA stream + Volkov kernel: concurrent execution of matrix multiplication (GPU) and cudaMemcpyAsync
- Performance jump at n = 672

SEVEN MAJOR RI-MP2 GRADIENT 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
3. Construction of the $C_{ia}{}^{Q}$ matrix
4. Assembly of the $\Gamma_{ia}{}^{Q}$ matrix (RI-MP2 correction to the two particle density matrix), $P_{ca}{}^{(2)}$ (virtual-virtual correction to the one-particle density matrix), and $P_{kj}{}^{(2)}$ (active-active correction to the one-particle density matrix)
5. Construction of Γ^{RS} (RI-MP2 specific two-particle density matrix)
6. $\Gamma_{ia}{}^{Q}$ transposition
7. Assembly of the L, P, and W matrices; solution of the Z-vector equation and final gradient evaluation

SIMULATION OF GLYCINE-N (1 CPU THREAD)

1 VS. 8 CPU THREADS

+ SCF, step 4, and step 7: 95+% of total wall time
+ step 4 dominates for large input molecules
+ Eight CPU threads reduce step 4 wall time (around 4-5x speedup)

SPEEDUP COMPARISON

- Blue: CPU 1 thread, Red: CPU 8 threads, Green: CPU + GPU
- CPU multi-thread performance similar to CPU + GPU performance for smaller glycine molecules
- GLYCINE-20: 18.8x speedup in Step 4, 7.4x speedup in total RI-MP2 time

SUMMARY

- Q-Chem CPU + GPU RI-MP2 Gradient code
- Dirac Cluster (NERSC): Tesla C2050 Fermi GPUs (DGEMM 230GFLOPS)
- CPU + GPU Algorithm: concurrent I/O + GPU DGEMM kernel, concurrent data copy + DGEMM
- Glycine-20: Speedup of 18.8x in Step 4, 7.4x in total RI-MP2 time (more expected for larger molecules)

FUTURE WORK

- New GPU DGEMM kernel suitable for the Fermi architecture (CUBLAS 3.1?, third-party kernel?)
- DGEMM kernel that utilizes both the CPU (multiple threads) and the GPU?
- RI-MP2 Gradient MPI + CUDA code
- Explore OpenCL as an alternative framework for heterogeneous computing

B Appendix: Papers

Two of the post-docs, Jihan Kim and Robert Priessl have made major contributions to a paper that was presented at the Cray User Group Meeting in Edinburgh Scotland in May 2010. This paper was awarded "Best Paper," by the organizing committee of the conference. A copy of this paper is given in this appendix. Additionally, first author papers by each of these post-doctoral researchers containing full coverage of the research material on each of these projects is being prepared for submission to refereed journals. An additional contribution to the paper comes from Kirsten Fagnan. As one of the first experiments at the start of her postdoc, Fagnan ran some experiments to investigate the memory usage of the ALE-AMR code. She ran the same calculation on different node and CPU configurations to come up with the plot in Figure 21. She is continuing to explore the scalability of the ALE-AMR code as well as the possibility of using a mixed MPI-OpenMP model within the code.

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
Stephan Ethier, Weixing Wang,
Princeton Plasma Physics Laboratory
Martin Head-Gordon,
University of California, Berkeley
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.

B.1 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.

B.2 Fusion Application

B.2.1 GTS — A massively parallel magnetic fusion application

The fusion application chosen for this study is the Gyrokinetic Tokamak Simulation (GTS) code [27], 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 [9]. 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[2]. 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.2.2 The 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 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 [9]. 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[3], requiring the processes within a

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

[3] 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,

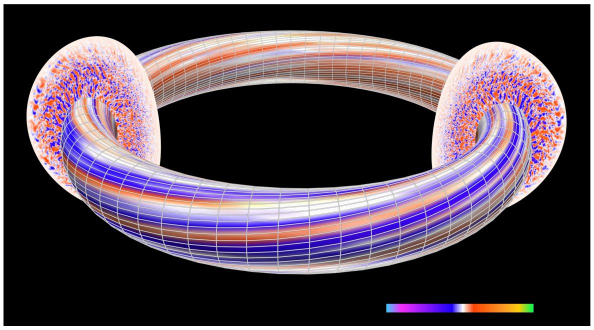


Figure 10: GTS' toroidal domain decomposition with magnetic field lines and density fluctuations

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 10 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) [17], 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.

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.

B.2.3 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

and all threads must contend for exclusive access [20].

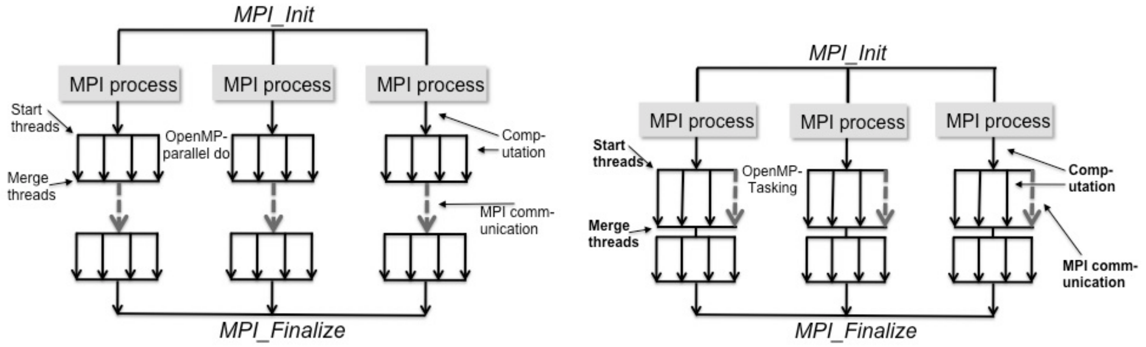
particles are likely to end up in neighboring toroidal domains. (Ions and electrons have a separate pusher and shift routines in GTS.) This shift of particles can happen to the adjacent or even to further toroidal domains of the tokamak and is implemented with MPI_Sendrecv functions operating in a ring-like 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 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.

```
do iterations=1,N
!compute particles to be shifted
  !$omp parallel do
  shift_p=particles_to_shift(p_array);

!communicate amount of shifted
! particles and return if equal to 0
  shift_p=x+y
  MPI_ALLREDUCE(shift_p,sum_shift_p);
  if(sum_shift_p==0) { return; }

!pack particle to move right and left
  !$omp parallel do
  do m=1,x
    sendright(m)=p_array(f(m));
  enddo
  !$omp parallel do
  do n=1,y
    sendleft(n)=p_array(f(n));
  enddo

!reorder remaining particles: fill holes
  fill_hole(p_array);

!send number of particles to move right
  MPI_SENDRECV(x,length=2,..);
!send to right and receive from left
  MPI_SENDRECV(sendright,length=g(x),..);
!send number of particles to move left
  MPI_SENDRECV(y,length=2,..);
!send to left and receive from right
  MPI_SENDRECV(sendleft,length=g(y),..);

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

Listing 1: Original GTS shift routine



(a) Original MPI/OpenMP hybrid model

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

Figure 11: 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).

```
!$omp parallel
!$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 2: Overlap MPI_Allreduce with MPI_Allreduce

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 11(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 the new OpenMP tasking functionality[4]. 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 11(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 11 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:

```
shift_p=x+y
!$omp parallel
!$omp master
  !$omp task
  do m=1,x
    sendright(m)=p_array(f(m));
  enddo
  !$omp end task
  !$omp task
  do n=1,y
    sendleft(n)=p_array(f(n));
  enddo
  !$omp end task

  MPI_ALLREDUCE(shift_p ,sum_shift_p);
!$omp end master
!$omp end parallel
if(sum_shift_p==0) { return; }
```

Listing 3: (1) Overlap MPI_Allreduce in 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,

[4] 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].

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 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[5].

```
integer stride=1000
!$omp parallel
!$omp master
!pack particle to move right
  do m=1,x−stride,stride
     !$omp task
     do mm=0,stride−1,1
        sendright(m+mm)=p_array(f(m+mm));
     enddo
     !$omp end task
  enddo
   !$omp task
  do m=m,x
     sendright(m)=p_array(f(m));
  enddo
   !$omp end task
!pack particle to move left
  do n=1,y−stride,stride
     !$omp task
     do nn=0,stride−1,1
        sendleft(n+nn)=p_array(f(n+nn));
     enddo
     !$omp end task
  enddo
   !$omp task
  do n=n,y
     sendleft(n)=p_array(f(n));
  enddo
   !$omp end task
  MPI_ALLREDUCE(shift_p,sum_shift_p);
!$omp end master
!$omp end parallel
if(sum_shift_p==0) { return; }
```

Listing 4: (2) Overlap MPI_Allreduce in the GTS shifter

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

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

is shown in Listing 4 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.

```
!$omp parallel                                  1
!$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

(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 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

```
!$omp parallel
!$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

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 [21]. Nonblocking collective operations are already provided by libNBC [12], 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.

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.

B.2.4 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 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.

Benefits & Limitations of hybrid Computing

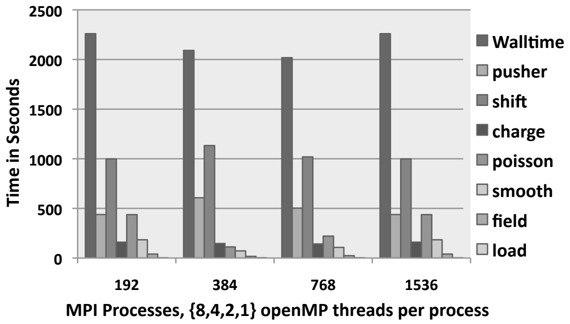
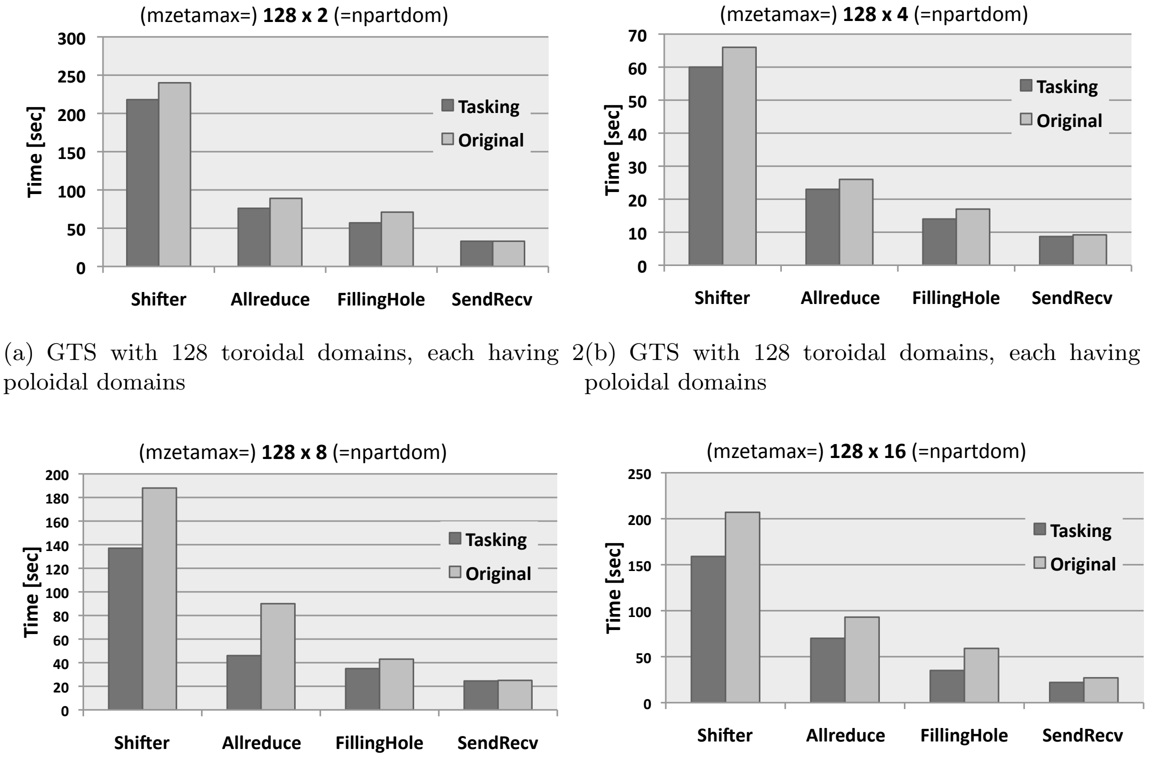


Figure 12: Evaluation of MPI/OpenMP hybrid model with GTC on Hopper.

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) [8], 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 12 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 12 clearly demonstrates that the hybrid approach outperforms the pure MPI approach (the fourth group in Figure 12) 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 12 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.

Performance Evaluation of OpenMP tasking to overlap communication with computation

The diagrams shown in Figure 13 present four GTS runs with different input files and domain decomposition executed on the Franklin Cray XT 4 machine. Figure 13(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



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

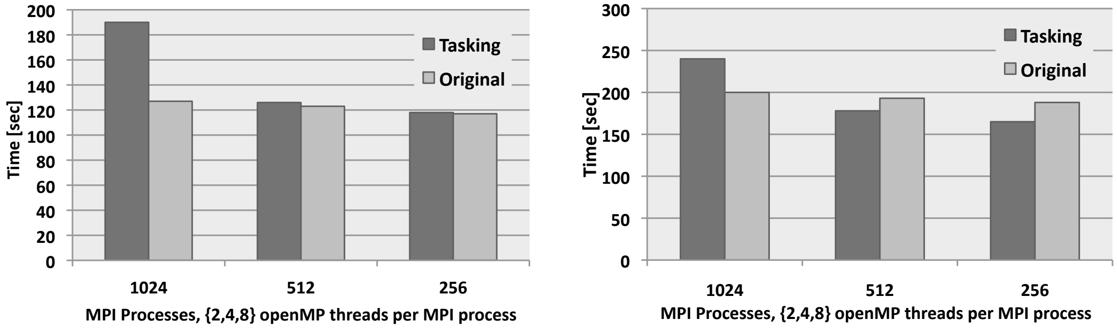
(b) GTS with 128 toroidal domains, each having 4 poloidal domains

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

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

Figure 13: 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.

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 13(a) to Figures 13(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 B.2.2, 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 Figure 13, 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 13(a)) than in a 128-MPI-processes-toroidal-domain of a GTS run with a total of 2048 MPI processes (Figure 13(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 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 13(a) to Figures 13(c) show no or only marginal performance benefits, the speed-up due to the "SendRecv" optimization is about 18% in Figure 13(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.



(a) Allreduce of 1 integer

(b) Allreduce of an array of 100 integers

Figure 14: Performance evaluation for overlapping execution of two consecutive MPI_Allreduce calls on Hopper.

```
!$omp parallel
!$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

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

Overlap communication with communication

Going one step further in reducing the time spent in sequentially[6] executed MPI communication, we want to show early results of experiments with overlapping of MPI communication with other

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

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 14 presents runtime comparisons of succeeding and independent MPI_Allreduce calls with varying messages sizes. Figure 14(a) and Figure 14(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 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 14(a) reflects a run with MPI_Allreduce calls of just one integer variable whereas Figure 14(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 14(a)), we can see a slight overlap in Figure 14(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 12. 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 [25]. Similar experiments to the one shown in Listing 7 have been conducted on Hopper with consecutive MPI_Sendrecv calls achieving similar same speed-ups.

B.2.5 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 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 [22] using static analysis techniques to guide subsequent code optimizations. The ROSE compiler framework will be introduced in more detail in section B.4.5 of the material modeling application.

B.3 Chemistry Application

B.3.1 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 [19]. More specifically, we focus on a MP2 method that utilizes the resolution-of-the-identity (RI) approximation [18], 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 [15] and can account for 80-90% of the correlation energy [14]. 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 [26]. 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.3.2 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.

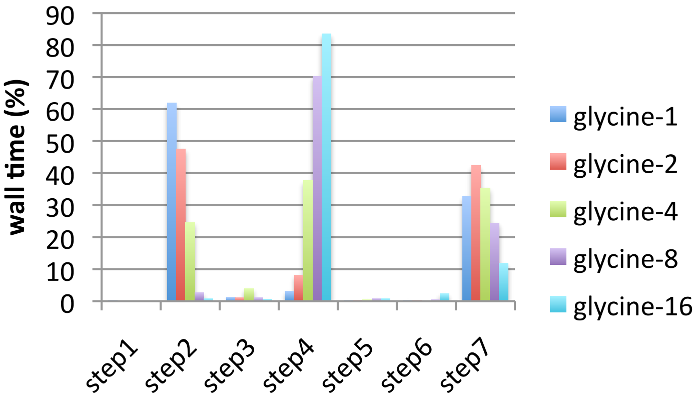


Figure 15: Percentage RI-MP2 wall time for glycine-n molecules with n = 1, 2, 4, 8, and 16.

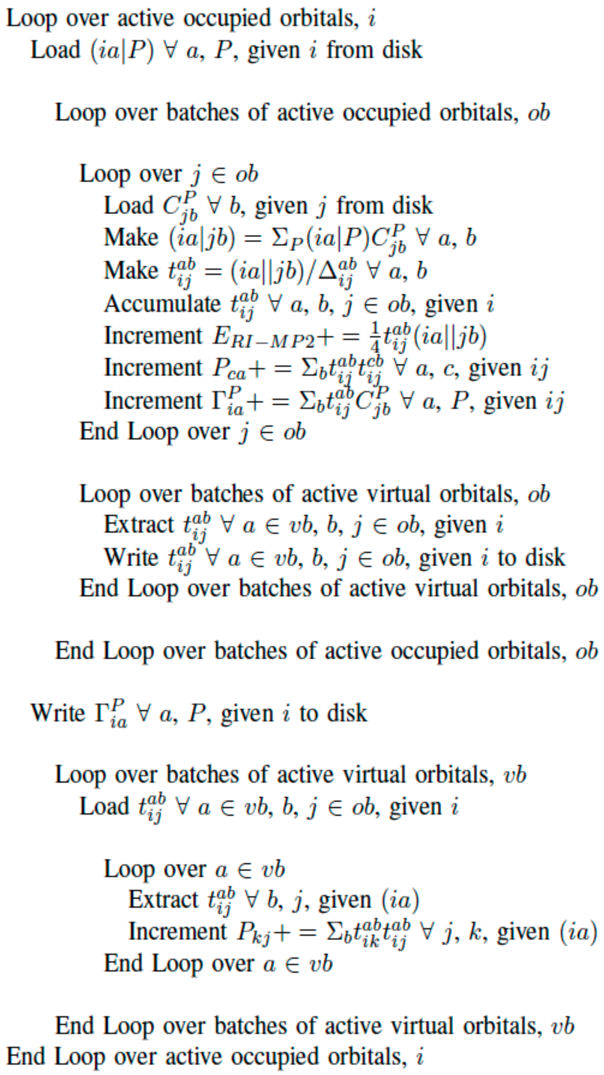
Table 2: step4, step7, and total wall time in seconds

	$n = 1$	$n = 2$	$n = 4$	$n = 8$	$n = 16$
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

Figure 15 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 2 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 [5], 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



```
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, $i$
```

Figure 16: Detailed look at the algorithm behind step 4

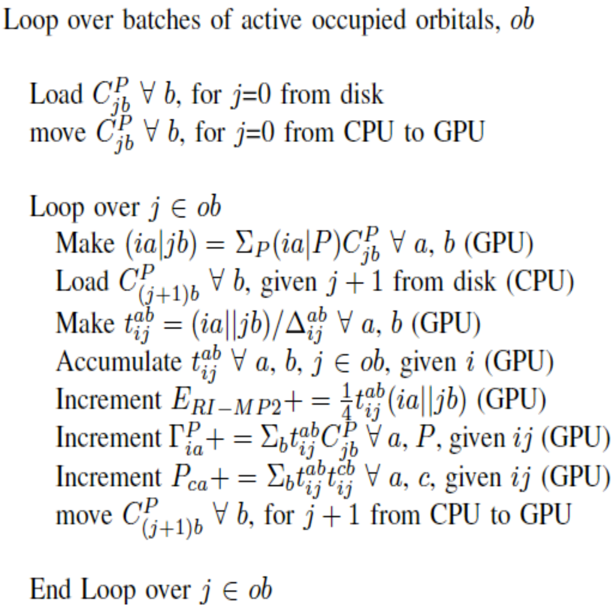
QuadroPlex 2200-S4, which contains four NVidia FX-5800 Quadro GPUs, with each GPU having 4GB of memory and 240 CUDA parallel processor cores.

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

The step 4 algorithm can be seen in figure 16. 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 17 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 $C^P_{jb}$ $\forall$ $b$, for $j$=0 from disk
  move $C^P_{jb}$ $\forall$ $b$, for $j$=0 from CPU to GPU

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

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

Figure 17: 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

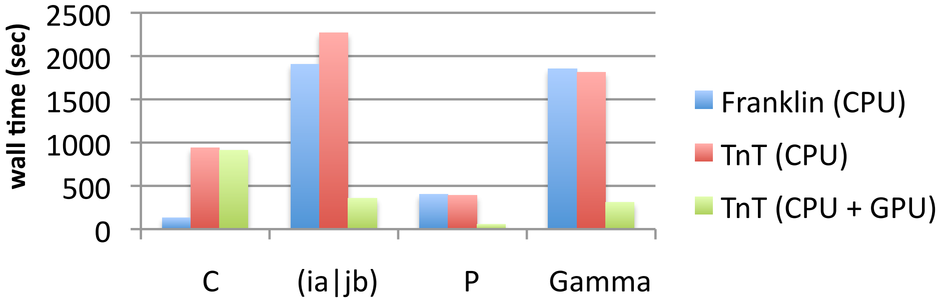


Figure 18: Four main routines wall times in step 4 for glycine-8 molecule

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 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.

B.3.3 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_{jb}^P, making $(ia|jb)$, making P_{ca}, and making Γ_{ia}^P.

From figure 18, 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_{jb}^P 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_{jb}^{P} load but from other I/O routines not seen from 18. 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.

B.4 Fluid/Materials Application

B.4.1 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 [16]. 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.4.2 Diffusion Solver Speed-up

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 [10]. The use of a FEM diffusion solver to model heat conduction and radiation transport is well studied [23] as is the integration of these physics modules into a hydrodynamic code [24]. 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.

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 [11] 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 [4] BiCG solver and the Euclid [13] 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

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

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

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

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 [23], and B is the blackbody intensity.

Parallel Issues

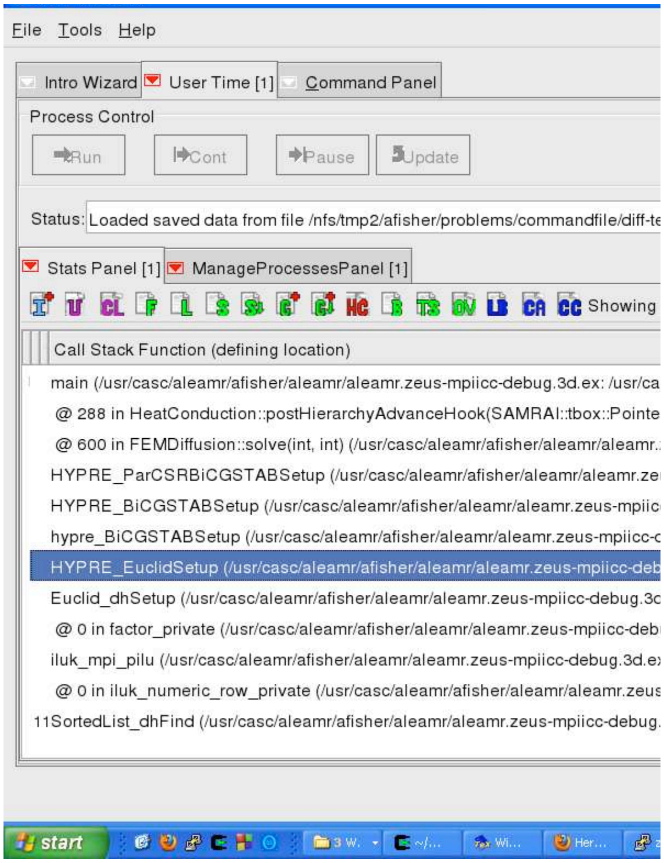
This diffusion solver and accompanying physics modules have been put through a variety of unit tests, accuracy checks, validation studies, and performance analyses some of which can be found in [10]. The solver performs well in all of these tests, however, when employed to solve larger parallel problems 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.

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

Table 3: 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.

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 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 19. This data turns out to be quite illuminating, as it seems that



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.
```

Figure 19: Hot call path obtained by the Open—SpeedShop tool. This represents a case where the program is running with the degraded performance issue.

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.

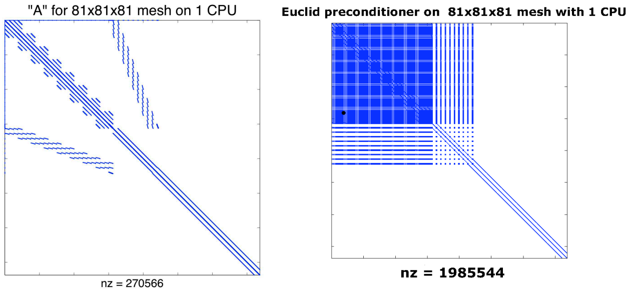


Figure 20: Sparsity plots of the A matrix and the corresponding matrix created by Euclid for preconditioning. Notice the large amounts of additional non-zero entries in the preconditioner.

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 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 4: 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.

B.4.3 Hybrid Parallelisation

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, [7] 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 [6]. 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 parallelisation.

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.

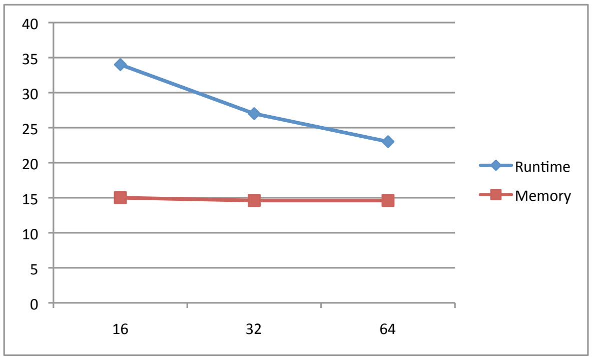


Figure 21: Looking at a fixed number of MPI tasks on a varying number of processors shows the potential of running OpenMP on the idle cores.

B.4.4 The ROSE Compiler Framework

The ROSE tool kit [22] 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.

B.4.5 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 translats 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 22 shows the lines of code and languages in ALE-AMR and the SAMRAI library.

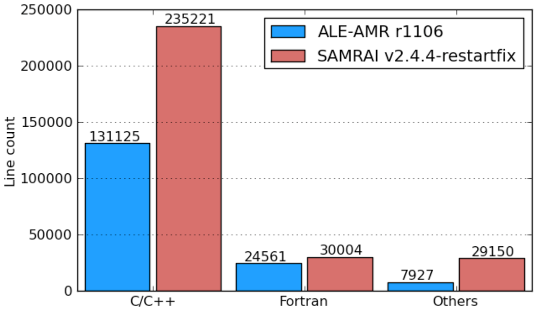


Figure 22: Breakdown of the programming languages used in the ALE-AMR code and those used in the SAMRAI library for particular releases of the codes.

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.

B.5 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.

B.6 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 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.

Appendix Specific 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] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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`.

[26] 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.

[27] 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.