Application Software Case Studies in FY06 for the Mathematical, Information and Computational Sciences Office of the U.S. Department of Energy

October 10, 2006

DRAFT

Contents

Contact Information **2**

FY06 Joule Software Effectiveness **3**

0.1 Annual Milestone 3

0.2 Q2 Information 3

0.2.1 Problem description 3

0.2.2 Algorithm description 3

0.2.3 Machine description 4

0.2.4 Data 4

0.3 Q4 Information 5

0.4 Results 5

0.4.1 DCA-QMC 7

0.4.2 Enzo 7

0.4.3 MADNESS 8

0.4.4 ScalaBLAST 8

DCA-QMC **10**

0.5 Q2 Information 10

DRAFT

0.5.1 Problems 10
0.5.2 Algorithm 15
0.5.3 Machine 17
0.5.4 Data 17
0.6 Q4 Information 17
0.6.1 Algorithm 17
0.6.2 Data 20
0.7 Results 20

Enzo **23**

0.8 Q2 Information 23
0.8.1 Problems 23
0.8.2 Algorithm 27
0.8.3 Machine 32
0.8.4 Data 32
0.9 Q4 Information 46
0.9.1 Algorithm 46
0.9.2 Data 50
0.10 Results 50
0.11 Algorithm for nonequilibrium ionization and radiative cooling (NONEQ) 56
0.12 Algorithm for equilibrium ionization and radiative cooling (GADGET) 57

MADNESS **58**

0.13 Q2 Information 58

DRAFT

0.13.1 Algorithm 59
0.13.2 Machine 60
0.13.3 Data 61
0.14 Q4 Information 61
0.14.1 Algorithm 62
0.14.2 Data 68
0.15 Results 70

ScalaBLAST **72**

0.16 Q2 Information 72
0.16.1 Problems 73
0.16.2 Algorithm 77
0.16.3 Machines 79
0.16.4 Data 80
0.17 Q4 Information 91
0.17.1 Algorithm 91
0.17.2 Data 98
0.18 Results 109

Bibliography **111**

DRAFT

Contact Information

K. J. Roche (rochekj@ornl.gov), Computer Science and Mathematics Division, Future Technologies Group, Oak Ridge National Laboratory, One Bethel Valley Road, P.O. Box 2008, MS - 6173, Oak Ridge, TN 37831-6173, U.S.A

DOE Program Management Distribution
Barb Helland, `barbara.helland@science.doe.gov`
Daniel Hitchcock, `daniel.hitchcock@science.doe.gov`
Fred Johnson, `fjohnson@er.doe.gov`
Michael Strayer, `michael.strayer@science.doe.gov`

Application Contacts
DCA-QMC: T. Schulthess (schulthesstc@ornl.gov), Computer Science and Mathematics Division, Computational Materials Science, Oak Ridge National Laboratory, One Bethel Valley Road, P.O. Box 2008, MS - 6164 Oak Ridge, TN 37831-6164, U.S.A
ENZO: M. Norman (mnorman@cosmos.ucsd.edu), University of California, San Diego,Center for Astrophysics and Space Sciences, 424 SERF Building, 9500 Gilman Drive, La Jolla, CA 92093, U.S.A
MADNESS: R. Harrison (harrisonrj@ornl.gov), Computer Science and Mathematics Division, Computational Chemical Sciences, Oak Ridge National Laboratory, One Bethel Valley Road, P.O. Box 2008, MS - 6367 Oak Ridge, TN 37831-6367, U.S.A
ScalaBLAST: T. Straatsma (tps@pnl.gov), Biological Sciences Division, WR Wiley Environmental Molecular Sciences Laboratory, Molecular Sciences Software, Pacific Northwest National Laboratory, P.O. Box 999, K1-92 Richland, WA 99352, U.S.A

Technical Team (as of end of Q2)
(dca-qmc) Thomas Maier, (enzo) James Bordner, Robert Harkness, Pascal Paschos, (madness) George Fann, Rebecca Hartman-Baker, William Shelton Jr., Sin-Ichiro Sugiki, (scalablast) Christopher Oehmen, Jarek Nieplocha

DRAFT

FY06 Joule Software Effectiveness

0.1 Annual Milestone

(SC GG 5.23.2) Improve Computational Science Capabilities: Average annual percentage increase in the computational effectiveness (either by simulating the same problem in less time or simulating a larger problem in the same time) of a subset of application codes within the `Scientific Discovery through Advanced Computing (SciDAC)` and base program efforts. FY06 Target: $\geq 50\%$

0.2 Q2 Information

In the second fiscal quarter, the application experts stated problems to study on target machines with their existing application code base at that time. Application benchmark runs of the problems on the machines were conducted at that time. The following information was gathered for each problem that was exercised.

0.2.1 Problem description

- description of physical problem to be constructed, computed, or recognized
- defense of why the problem is significant

0.2.2 Algorithm description

- description of the problem's complexity
- description of a procedure to compute the problem

DRAFT

0.2.3 Machine description

- description of the machine used to compute the problem

0.2.4 Data

- application software source codes at end of Q2
- build information on target machine at the end of Q2
  - makefile
  - screen dump of the make
  - environment - special variables, os, compiler, library versions, etc.

- runtime information at the end of Q2
  - batch script for each problem's job submission
  - preferably utilize $\geq 50\%$ of the available machine hardware
  - input data for program
  - environment - special variables, os, scheduler, etc.
- measured observables at the end of Q2 (minimal set)
  - total wall time to generate, compute, and accept results for the problems
  - total wall time should exceed 1 hour on preferably $\geq 50\%$ of the available machine hardware
  - measures of problem specific metrics of performance

Q2 Information	**Problem**	**Algorithm**	**Machine**	**Data**
sec[0.4.4] DCA-QMC	sec(0.5.1)	sec(0.5.2)	sec(0.5.3)	sec(0.5.4)
DCA-QMC	sec(0.5.1)	sec(0.5.2)	sec(0.5.3)	sec(0.5.4)
sec[0.7] ENZO	sec(0.8.1)	sec(0.8.2)	sec(0.8.3)	sec(0.8.4)
ENZO	sec(0.8.1)	sec(0.8.2)	sec(0.8.3)	sec(0.8.4)
sec[0.12] MADNESS	sec(0.13)	sec(0.13.1)	sec(0.13.2)	sec(0.13.3)
sec[0.15] ScalaBLAST	sec(0.16.1)	sec(0.16.2)	sec(0.16.3)	sec(0.16.4)
ScalaBLAST	sec(0.16.1)	sec(0.16.2)	sec(0.16.3)	sec(0.16.4)
ScalaBLAST	sec(0.16.1)	sec(0.16.2)	sec(0.16.3)	sec(0.16.4)
ScalaBLAST	sec(0.16.1)	sec(0.16.2)	sec(0.16.3)	sec(0.16.4)

Table 1: The baseline information collected at the end of Q2 for the FY06 case studies.

DRAFT

0.3 Q4 Information

The application experts state and demonstrate their enhancements on the original target platforms. Pointers into the text are provided in Table[2].

Q4 Information	Problem	Algorithm	Machine	Data
sec[0.4.4] DCA-QMC	sec(0.5.1)	sec(0.6.1)	sec(0.5.3)	sec(0.6.2)
sec[0.7] ENZO	sec(0.8.1)	sec(0.9.1)	sec(0.8.3) da	sec(0.9.2)
sec[0.12] MADNESS	sec(0.13)	sec(0.14.1)	sec(0.13.2)	sec(0.14.2)
sec[0.15] ScalaBLAST	sec(0.16.1)	sec(0.17.1)	sec(0.16.3)	sec(0.17.2)

Table 2: The information collected at the end of Q4 for the *enhanced* FY06 case studies. The list excludes some results for clarity but represents an enhancement for each application accomplished this FY.

0.4 Results

In this report a problem refers to a question whose language is derived from physical reality and theory. An instance of a problem is a list of values for each of the parameters that define it. The computational complexity of the problem is the relationship between values of the input symbols and the number of operands managed at any instant during execution of the procedure and the number and types of operations to be performed.

The process of constructing and computing algorithms for a problem using other algorithms is called reduction. The notion of reduction is recursive and to be practical assumes special properties for the algorithms. Informally, effective procedures are finitely describable *and* consist of discrete steps, each of which can be executed mechanically. Note that the notion of an *effective* reduction does not imply correctness nor convergence to it. The interpretation of a computed physical result is required to assure that. Such a complete analysis is not found here.

- performance enhancement
  - $T_{Q4}\{p_i, a_l := e(a_j), m_k, data_{i,l,k}\} \leq \frac{1}{2} T_{Q2}\{p_i, a_j, m_k, data_{i,j,k}\}$
    - $T_{Q4}\{p_i, a_l, m_k, data_{i,l,k}\}$ is the time for machine m_k to compute problem p_i with algorithm a_l at the end of the fiscal year
    - $T_{Q2}\{p_i, a_j, m_k, data_{i,j,k}\}$ is the time for machine m_k to compute problem p_i with algorithm a_j at the end of the second fiscal quarter
    - $a_l := e(a_j)$ is the performance enhanced version of algorithm a_j
    - both a_l and a_j compute problem p_i on machine m_k
  - discuss how a_l is different from aj
  - provide the data

    * application software source codes at end of Q4
    * build information on target machine at the end of Q4
        · makefile
        · screen dump of the make
        · environment - special variables, os, compiler, library versions, etc.
    * runtime information at the end of Q4
        · batch script for each problem's job submission
        · preferably utilize $\geq 50\%$ of the available machine hardware
        · input data for program
        · environment - special variables, os, scheduler, etc.
    * measured observables at the end of Q4 (minimal set)
        · total wall time to generate, compute, and accept results for the problems
        · total wall time should exceed 1 hour on preferably $\geq 50\%$ of the available machine hardware
        · total cpu time per processor
        · measures of problem specific metrics of performance

- scientific enhancement
  - $T_{Q4}\{p_l := e(p_i), a_m := e(a_j), m_k, data_{l,m,k}\} \sim T_{Q2}\{p_i, a_j, m_k, data_{i,j,k}\}$
    * $T_{Q4}\{p_l, a_m, m_k, data_{l,m,k}\}$ is the time for machine m_k to compute the enhanced version of problem p_i with the enhanced algorithm a_m at the end of the fiscal year
    * $T_{Q2}\{p_i, a_j, m_k, data_{i,j,k}\}$ is the time for machine m_k to compute problem p_i with algorithm a_j at the end of the second fiscal quarter
    * $p_l := e(p_i)$ is the science enhanced version of p_i; e.g. it is two times more accurate, can take twice as many time steps, the rate of convergence doubled, etc.
    * $a_m := e(a_j)$ is the enhanced version of algorithm a_j that accounts for the enhancement
  - discuss how p_l is different from p_i -include notion of *factor of two*
  - discuss how a_m is different from aj
  - provide the data
    * application software source codes at end of Q4
    * build information on target machine at the end of Q4
        · makefile
        · screen dump of the make
        · environment - special variables, os, compiler, library versions, etc.
    * runtime information at the end of Q4
        · batch script for each problem's job submission
        · preferably utilize $\geq 50\%$ of the available machine hardware
        · (enhanced)input data for program

        - environment - special variables, os, scheduler, etc.
    - measured observables at the end of Q4 (minimal set)
        - total wall time to generate, compute, and accept results for the problems; should be approximately equal to the time measured in Q2
        - total wall time should exceed 1 hour on preferably $\geq 50\%$ of the available machine hardware
        - total cpu time per processor
        - measures of problem specific metrics of performance

- new result
    - $T_{new} := T_{Q4}\{p_i, a_j, m_k, data_{i,j,k}\}$
        - collect the information found in the *Q2 Baseline Information* described above

Speedup, γ. Some applications report the effectiveness of their optimization in terms of a speedup factor, call it γ. This means that the time to compute the results with the original code is γ times as long as the code that has been enhanced for performance. That is $T_o = \gamma T_1$ and thus the assertion becomes $\frac{T_o - T_1}{T_o} = \frac{\gamma T_1 - T_1}{\gamma T_1} = \frac{\gamma - 1}{\gamma}$. Clearly, when $\gamma \geq 2$ then the constraint is satisfied.

0.4.1 DCA-QMC

The DCA-QMC,Dynamical Cluster Approximation-Quantum Monte Carlo technique, treats local interactions inside a finite cluster explicitly and otherwise uses mean-field theory for the interation. The mapping from a microscopic lattice to the cluster is done by coarse-graining the reciprocal lattice. The coarse-grained many-body Green functions and interactions are computed to define an effective problem of a periodic cluster of size N_c coupled to a mean-field host. The QMC solves the effective cluster problem, which yields a new estimate for the mean-field host. This procedure is repeated until self-consistency is achieved. The primary

Problem	System	Enhancement
Pairing interaction, 2D Hubbard Model (sec0.5.1)	Cray X1E [0.5.3], *phoenix.ccs.ornl.gov*	**performance**(Tables[6,9]), $\sim 74.03\%$

metric is the *time to compute the Green function update per time-slice.* The performance enhancement comes directly from comparing he original and enhanced times to compute *problem one*(0.5.1) on machine(0.5.3) with the original algorithm(0.5.2) and the enhanced algorithm(6). The times are found in tables[6,9] respectively.

0.4.2 Enzo

Enzo simulates the detailed dynamics and thermodynamics of the cosmic plasma under the influence of dark matter and dark energy, as well as ionizing radiation backgrounds to study

galaxy formation in the universe. Cosmological structure formation is inherently nonlinear, multidimensional, and involves a broad range of physical processes operating on a range of length- and time-scales. Large scale numerical simulation is the only means of studying it in any detail.

A performance metric of primary interest is *time/processor/physical timestep*.

Problem	System	Enhancement
High redshift galaxies, amr (sec0.8.1)	IBM Power5 (0.8.3), *bassi.nersc.gov*	**performance**(Tables[8,7]), $\sim 66.35\%$

The results from running the original algorithm(0.8.2) may be referenced in Figure[19] and Table[8]. The enhanced results come from running AMR_FINAL(0.9.1) and may be referenced in Figure[16] and Table[7].

0.4.3 MADNESS

MADNESS is a general framework for scientific simulation that provides new capabilities for rapid computation with guaranteed precision enabled by new mathematical tools for multiresolution analysis in multiwavelet bases and efficient computation in many-dimensions through the use of separated representations.

Problem	System	Enhancement
project nuclear copper potential into wavelet basis(sec0.13)	Cray XT3, *jaguar.ccs.ornl.gov*	**performance**, $\gamma = 4.4$ (see[23])

Table 3: A scalable speedup,γ, is asserted for the baseline problem stated for the **MADNESS** code. The projection with adaptive refinement to a precision of 10^{-3} into the $k = 9$ polynomial basis of the full nuclear potential arising from a cube of 4096 copper atoms arranged on a body-centered cubic lattice was computed.

0.4.4 ScalaBLAST

ScalaBLAST performs homology detection and sequence alignment to identify relationships between molecular machinery of organisms which have been well-characterized and those which have been newly sequenced. It is a hybrid application that capitalizes on the scalability of query scheduling approaches, and at the same time shares the target database using the Global Array toolkit which provides a software-enabled shared memory interface to both true shared memory and distributed memory architectures.

Problem	System	Enhancement
(sec0.16.1)	MPP2, *mpp2.emsl.pnl.gov*	**new result**, see(**??**)

Table 4: FY06 Joule performance enhancement for the ScalaBLAST code. The measure of performance is *time/query/processor*.

DRAFT

New result At the turn of FY06, MPI BLAST(v1.4.0) was the accepted state-of-the-art in high performance sequence alignment. The primary limitation found with MPI BLAST is memory usage. Figure[33] illustrates the memory footprint of MPI BLAST, which grew linearly on the master node in time during execution of the large ENV benchmark. This trend continued until all the memory of the master node was consumed. Figure[34] illustrates the corresponding drop in free memory for the master node during this time. Once memory is consumed on the master node, swapping begins, as illustrated in Figures[35,36]. At this point, the calculation has essentially halted -prior to producing any output.

The two key capability limitations to performing very large BLAST searches that were addressed by ScalaBLAST are 1) inability of query scheduling applications to handle databases too large to fit in-core without swapping and 2) poor scaling of database parsing methods. Modifications (please see sec[0.17.1]) to ScalaBLAST addressed these issues head on.

Compared with the failure of MPI BLAST to produce any output for either of the benchmark runs on both Jacquard and MPP2, ScalaBLAST succeeded in delivering a high-throughput of BLAST results per-processor on both platforms and for both benchmarks essentially without regard for number of processors involved in the calculation-the same as ideal scaling for both problems, as illustrated in Table [28].

DRAFT

DCA-QMC

0.5 Q2 Information

0.5.1 Problems

Understanding the mechanism behind high-temperature superconductivity Superconductivity is, from an energy point of view, usually mentioned in the context of power transmission. The fact that metallic wires have non-zero electrical resistance currently imposes constraints on how electric power is used and distributed. For example, in order to minimize energy losses, alternating currents (ac) at high voltage are used in transmission, while the majority of power sources and consumption operate with direct currents (dc) and at low voltage. Availability of wires with zero resistance that can sustain large enough electrical currents, would allow to directly connect power sources to end-users and open new avenues to power generation. For example, photovoltaic energy sources could be concentrated in remote locations while making the generated power available in major urban agglomerations. These and many other very appealing applications in electronics and medicine seem to explain the continued interest in superconductivity, a state of matter with zero electrical resistance. But the implications on science and technology are potentially much broader. The superconducting state is one of few known macroscopic quantum states, where of the order of 10^{23} electrons are coherent (or entangled in the terminology of quantum computing). In high temperature superconductors (HTSC), this state persists up to 150 degrees Kelvin. Understanding the mechanism how this state forms could, for example, fundamentally impact the way quantum computing and quantum teleportation is approached.

The discovery in 1986 by Bednorz and Müller of superconductivity with transition temperatures above 30 K in Ba doped $LaCuO_2$, a poor conductor ($LaCuO_2$ is an insulator), marked the beginning of a new era in condensed matter research. Within a few years new transition metal oxides were discovered with superconducting temperatures as high as 150 K - Bednorz and Müller were awarded the Nobel Prize in 1987). Despite two decades of intense research, the pairing mechanism in high temperature superconductors (HTSC) remains a mystery and represents one of the most

DRAFT

important outstanding problems in condensed matter science today.

Conventional superconductors are well described by the Bardeen-Cooper-Schrieffer (BCS) theory. In this theory, superconductivity results from an attractive interaction between electrons mediated by the exchange of lattice vibrations or phonons. By forming Cooper-pairs, the electrons can lower their potential energy, condense into a coherent macroscopic quantum state and conduct electricity without resistance. While conventional superconductors are well understood, the pairing mechanism in HTSC is believed to be of entirely different nature. Strong correlations between electrons were realized to play a crucial role in HTSC. Hence, models describing itinerant correlated electrons, in particular the two-dimensional Hubbard model, are believed to capture the essential physics of HTSC. With only the basic ingredients of a strongly correlated electron system, i.e. a hopping integral t that allows electrons to move between neighboring sites, and a mutual Coulomb repulsion U electrons feel when residing on the same site, the Hubbard model attempts to capture the physics of electrons moving in the CuO_2 planes of HTSC. With more than a thousand publications per year for the last ten years, the Hubbard and related models are among the mostly studied models in condensed matter physics. Despite this huge effort and its simplicity, however, this model has remained unsolved and the question of whether it contains enough ingredients to describe HTSC continues to be one of the most important problems in condensed matter theory.

A recent concurrence of new algorithmic improvements over the past decade that resulted in the DCA/QMC code [8], and the continued delivery of increasingly more powerful computing capabilities have enabled systematic studies to solve the two-dimensional (2D) Hubbard model for the HTSC cuprates [20]. The DCA method [1, 3, 4, 5, 6] (for a review see [7]) takes advantage of the short length-scale of spin correlations observed in optimally doped HTSC to map the original infinite size system onto a cluster self-consistently embedded in a mean-field host. It retains the dynamics of short-ranged correlations within the cluster, while approximating longer-ranged correlations on the mean-field level. Translational invariance of the original system assures that the quantity describing the mean-field host can be self-consistently determined from the solution of the cluster problem. The complexity of the original problem with an infinite number of degrees of freedom is thus reduced to a self-consistent finite-size cluster problem with N_c degrees of freedom. The remaining cluster problem may then be solved numerically by a number of techniques including the QMC method used in this project.

The DCA/QMC method has been recently utilized to perform a systematic study of superconductivity in the 2D doped Hubbard model [20]. In the study, superconductivity is indicated by the divergence of the pair-field susceptibility. Systematic calculations of this quantity for different cluster sizes show, for the first time, evidence for superconductivity in the 2D Hubbard model at finite temperatures, if large enough clusters are considered. Based on these results, the 2D Hubbard model has enough ingredients to describe HTSC.

Given these results, there is the unique opportunity to gain insight in the mechanism of high-temperature superconductivity. In this project the success of these calculations is leveraged to examine the question of what is causing the electrons to pair in the 2D Hubbard model. To this end, large-scale computer simulations will be used to examine the structure of the irreducible contribution to the pair-field susceptibility, the particle-particle irreducible vertex function for various values of the interaction strength U. This quantity can be viewed as the effective pairing interaction and thus is responsible for pairing. In addition, the corresponding quantities for the magnetic and charge susceptibilities, i.e. the magnetic and charge irreducible particle-hole vertices, will be studied. The ouput will be valuable information about the dominant excitations in the system. Further, an exact decomposition will be made of the particle-particle vertex function into a fully two-fermion irreducible vertex and magnetic and charge exchange channels. The hope is to pinpoint the dominant contribution to the pairing interaction.

On the weak coupling (perturbative) level, the decomposition is well understood. The dominant part of pairing interaction is found to come from the magnetic exchange channel , so that the exchange of antiferromagnetic spin fluctuations can be held responsible for pairing (see e.g. [21, 22]). Recent DCA/QMC calculations in the weak to intermediate coupling limit confirm this picture [**?**]. However, the relevant regime to describe the HTSC is the intermediate to strong coupling regime where U is of the order of the bandwidth. Here, the DCA/QMC simulations will provide *exact* results for the decomposition of the pairing interaction. When completed, the study will provide, the first time, information on what is causing the electrons to pair in the 2D Hubbard model at intermediate coupling, and hence with information on the mechanism of high-temperature superconductivity.

Significance of Research A complete theoretical understanding of HTSCs, in particular the pairing mechanism, could lead to the ability to design and synthesize room-temperature superconductors, which would have tremendous technological implications. Evidence for the importance of this problem is further given by the fact that among the 125 most important problems in science as listed in the Science magazines anniversary edition [23] is "What is the pairing mechanism behind high-temperature superconductivity?". Despite two decades of studies, this remains an unsolved problem.

Many different techniques, from analytical to numerical have been applied to study superconductivity in the Hubbard model. Weak coupling theories for the Hubbard model based on perturbative approaches assume that the Coulomb repulsion U between electrons is weak compared to their kinetic energy. Results from these methods show evidence that the exchange of antiferromagnetic spin fluctuations can mediate pairing with the correct symmetry [22]. However, these approaches suffer from the large magnitude of U necessary to describe HTSC, rendering these approaches neither helpful nor valid. Many theorists have therefore turned to numerical approaches to close the gap between the model defined by its Hamiltonian and its properties. A large

body of work has been devoted to a direct (numerically) exact solution of finite-size systems using exact diagonalization or Quantum Monte Carlo (QMC) methods (for a review see [24]). Exact diagonalization, however, is severely limited by the exponential growth of computational effort with system size, while QMC methods suffer from what is known as "the sign problem". The size and temperature limitations faced by these finite size approaches has prevented these studies to actually observe a superconducting transition in the Hubbard model (see [21]). Another difficulty of these methods arises from their strong finite-size effects, often ruling out the reliable extraction of low-energy scales, which are important to capture the competition between different ground states often present in correlated electron systems.

The proposed tool in this project, the DCA/QMC approach, does not face these limitations. The approach is valid for any strength of the Coulomb interaction U, and thus is well suited to address the question of superconductivity in the relevant intermediate to strong coupling regime. In addition, the DCA sign problem is significantly less severe than that encountered in finite size simulations, as illustrated in Fig. 1 where the DCA average sign is compared to that of a corresponding finite size QMC calculation by White *et al.* [25]. As a result, exploring larger systems and lower temperatures has become possible.

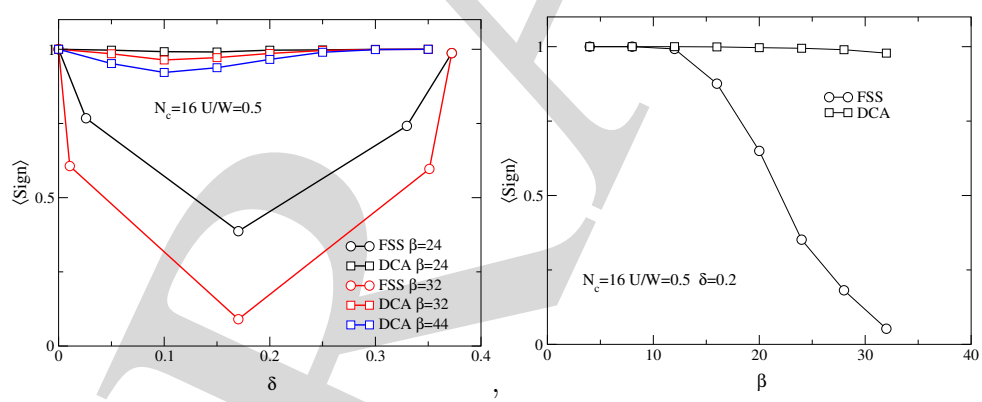

Figure 1. A comparison of the average sign for DCA and finite size simulations (FSS) of the 2D Hubbard model when the Coulomb interaction is one-half of the bare bandwidth $U = 0.5W$ for a cluster/lattice size $N_c = 16$ as a function of doping δ (left) and as a function of inverse temperature $\beta = 1/T$ (right).

Problem One:The structure of the pairing interaction in the 2D Hubbard model In this study we will examine the question of what is causing the electrons to pair in the 2D Hubbard model. To this end, we will use large-scale computer simulations to systematically examine the structure of the pairing interaction in the 2D Hubbard model. The effective pairing interaction is given by the irreducible particle-particle vertex which is obtained by inverting the Bethe-Salpeter equation for the vertex function. We will study this quantity for various values of the interaction strength U. In addition, we will examine the corresponding quantities for the

magnetic and charge susceptibilities, i.e. the magnetic and charge irreducible particle-hole vertices, which will yield valuable information about the dominant excitations in the system. We will further use an exact decomposition of the particle-particle vertex function into a fully two-fermion irreducible vertex and magnetic and charge exchange channels. This will allow us to pinpoint the dominant contribution to the pairing interaction.

In this case study, we examine the evolution of the eigenvectors of the particle-particle kernel of the Bethe-Salpeter equation with temperature, at fixed filling $\langle n \rangle = 0.95$ and Coulomb interaction $U = 8t$ for a cluster size of $N_c = 24$. This study is representative of the runs required to complete this project. Due to the large value of U, the time step $\Delta\tau \propto 1/\sqrt{U}$ has to be chosen small enough to generate reliable results. The numerical expense of the DCA/QMC code scales as $\mathcal{O}((N_c N_l)^3)$ where the number of time steps $N_l = \beta/\Delta\tau$ and $\beta = 1/T$ is the inverse temperature. Hence, large scale computations on a significant portion of the X1E are necessary due to the large values of N_c and N_l required in this study.

Problem Two:Groundstate energies in the 2D Hubbard model Numerical studies of the 2D Hubbard model have shown that this model can exhibit antiferromagnetism, stripes, pseudogap behavior as well $d_{x^2-y^2}$-wave pairing. In addtion, this model appears to be delicately balanced between these nearly degenerate phases. Slight increases in doping can tip the balance from antiferromagnetism to a striped phase or a superconducting phase. Small variations in U can lead to a transition from a striped phase to superconductivity. Due to the delicate balance between these phases, different numerical techniques often find different groundstate phases for the same model. Renormalization group techniques describing the physics of a finite size system, e.g., find striped phases, while the DCA/QMC technique favors d-wave superconductivity. The fact that these techniques find different groundstates may also reflect the influence of different boundary conditions or different cluster shapes and aspect ratios. Furthermore, the remarkable similarity of this behavior with the range of phenomena observed in the cuprates provides strong evidence that the Hubbard model containes the esential physics of the problem.

Here we will study the collection of different phases within a single numerical technique, the DCA/QMC approach, and calculate their respective groundstate energies for different fillings and values of the Coulumb interaction U. This will allow us to decide which phase is favored in a particular parameter regime. This study will require long runs averaging over many samples generating a small statistical error to resolve the small energy differences between different phases.

In a case study, we will perform simulations at fixed cluster size $N_c = 4$, Coulomb interaction $U = 8t$ and filling $\langle n \rangle = 0.95$ down to very low temperatures to allow for the extrapolation of the energy to zero temperature. Since the low temperatures necessitate the use of small time-steps $\Delta\tau$ and therefore large N_l, this study will require a significant portion of the resources on the X1E.

0.5.2 Algorithm

The essential nature of the dynamical cluster approximation [1, 3, 4, 5, 6] (for a review see [7]) is that correlations within a finite size cluster are treated explicitly, while those at longer length-scales are treated at the mean-field level. The mapping from the lattice to the cluster is accomplished by coarse-graining the reciprocal space of the lattice. The coarse-grained many-body Green functions and interactions are then used to define an effective problem of a periodic cluster of size N_c coupled to a mean-field host. The QMC method may then be used to solve the effective cluster problem, which yields a new estimate for the mean-field host. This procedure is repeated until self-consistency is achieved.

QMC is itself inherently parallel because it consists of a stochastic Markov-chain walk. Measurements are made periodically along this walk. The code performs several independent, shorter Markov-chain walks on different processors and averages the results of each walk to obtain the final result of the calculation using MPI. Apart from the fraction of the walk required to achieve equilibrium, the result is an almost perfect parallel speedup as an increasing number of processors are applied to a problem. This arises because only an extremely small amount of communication between processors is required – first to initialize the Markov-chain walks and then to collect the data for averaging at the end of the Markov process (even this averaging can be done in parallel using MPI calls).

The central quantity of the DCA/QMC code is the single-particle Green function G_c of the effective cluster problem, which is a matrix of size $N \times N$ [8]. Here $N = N_c N_l$ where N_l is the number of "time-slices" in the time direction. The majority of the CPU time is spent in the inner loop of the QMC simulation. The Hirsch-Fye algorithm sweeps through the HHS decoupling fields and proposes local changes in the HHS fields

$$s_n \rightarrow s'_n = -s_n. \tag{1}$$

The transition probability is calculated from the ratio of the old Green's function G_c to the new Green function G'_c.

$$G_{c_\sigma} {G'_{c_\sigma}}^{-1} = I + (I - G_{c_\sigma})(e^{V'_\sigma - V_\sigma} - I)\,. \tag{2}$$

where the matrix I is the identity and the matrix V is diagonal with finite elements $V = \alpha\sigma s_n$. Since the change is local in both space and time, $(e^{V'_\sigma - V_\sigma} - I)$ has only one finite element. So

$$R = \det\left[G_{c_\sigma} {G'_{c_\sigma}}^{-1}\right] = 1 + (1 - G_{c_{\sigma,nn}})(e^{-2\alpha\sigma s_n} - 1)\,. \tag{3}$$

If R is greater than a random number between zero and one, then the change is accepted, and the Green's functions must be updated using

$$G'_{c_\sigma} = G_{c_\sigma} + (G_{c_\sigma} - I)(e^{V'_\sigma - V_\sigma} - I)G'_{c_\sigma} \tag{4}$$

Eq. (4) is iterated twice to obtain an updating equation for G_c

$$G'_{c_{\sigma,ij}} = G_{c_{\sigma,ij}} + \left(G_{c_{\sigma,in}} - \delta_{in}\right)\left(e^{-2\alpha\sigma s_n} - 1\right) G_{c_{\sigma,nj}}/R \tag{5}$$

This equation is the inner loop of the code where a major part of the CPU time is spent. It is a vector outer product followed by a matrix update, $\mathbf{G}'_c = \mathbf{G}_c + \mathbf{a} * \mathbf{b}$ where a and b are two vectors of dimension N. The operation may be computed with a call to the BLAS[9] subroutine *dger()* which performs a double-precision rank-one matrix update representing $O(N^2)$ operations. Each iteration requires N such calls, however, resulting in $O(N^3)$ operations.

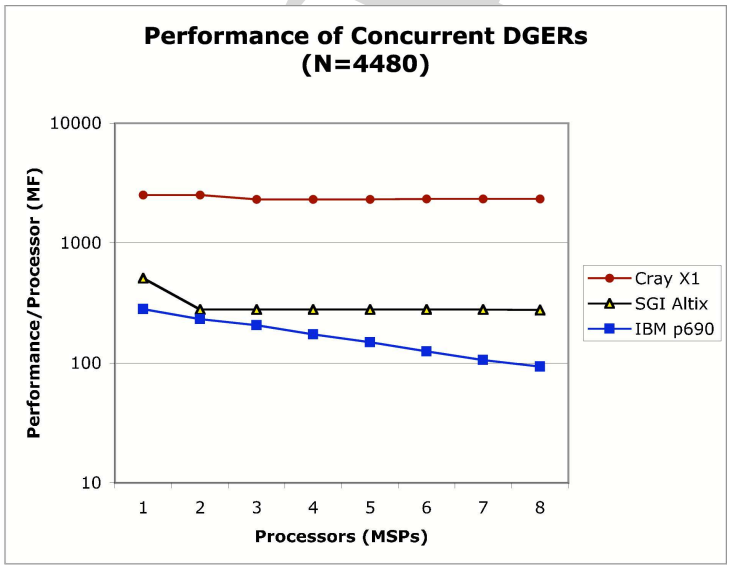

Figure 2. Per-processor performance of concurrent DGER calls using $N = 4480$ matrices.

In Figure[2] the vendor optimized *dger()* was used on each system. The figure shows the performance for a matrix of size $N = 64 \times 70 = 4480$, which is representative of large DCA runs. Separate *dger()* instances were run concurrently across increasing numbers of processors (MSPs), mimicking the processes of a Monte-Carlo simulation. The Cray X1E memory system is able to maintain performance and efficiency with added processors, while the IBM P690 steadily degrades. The SGI Altix degrades going from one to two processors because memory bandwidth is shared between processor pairs. The Cray X1E maintains 8 times the performance and 4 times the efficiency of the other systems. Despite the Monte-Carlo nature of the DCA/QMC algorithm, the Cray X1E also has an important scalability advantage over systems with weaker processors. Each DCA/QMC process has a significant fixed start-up cost, which favors splitting the Monte-Carlo iterations across fewer, faster processors.

Analysis code In addition to the QMC code described above, an analysis code has been developed to analyze the output of the DCA/QMC code. For large data sets, this code is run on machines with a large amount of shared memory, such as ORNL's SGI Altix (Ram). This code takes the irreducible quantities produced by the DCA/QMC simulation, including the vertex functions, and uses them to calculate a variety of lattice response functions including the magnetic and pair-field susceptibilities, which are needed in the analysis proposed here.

0.5.3 Machine

The Cray X1E (http://info.nccs.gov/resources/phoenix) at the National Center for Computational Sciences (NCCS) was utilized to establish both the Q2 baseline and the Q4 enhancements. Practically speaking, the system hardware description was static during the fiscal year.

0.5.4 Data

Q2 Collected Data	Problem One(0.5.1)	Problem Two(0.5.1)
Source codes	yes	yes
Build information	yes	yes
Runtime information	yes	yes
Total walltime	yes	yes

Table 5: Information collected during Q2 for the computed DCA-QMC problems. Please see sec[0.2.4] for description.

0.6 Q4 Information

The problems and target machine remained unchanged throughout the fiscal year in this study. Please see section[0.5] for details. The algorithm was modified. A description now follows of the changes.

0.6.1 Algorithm

Delayed Green's function Updating The subroutine *dger()* is inefficient as it computes only two floating point operations per memory access. Both floating point operations can be done in the same clock cycle, but many clock cycles are required

Identifier	problem size(a.u.)	processors	Q2 Time(h)	Weighted Time(h)
101	240	8	3.2	25.6
102	240	8	3.13	25.04
103	240	8	3.08	24.64
104	240	8	3.10	24.8
105	240	16	1.52	24.32
106	240	32	0.72	23.04
107	288	64	0.51	32.64
108	384	64	1.13	72.32
109	480	128	1.75	224
110	576	128	2.97	380.16
111	768	256	4.13	1057.28
112	864	256	4.52	1157.12
113	1056	512	3.85	1971.2
				5042.16

Table 6: Times to compute *problem one*(0.5.1) on machine(0.5.3) with algorithm(0.5.2) measured during the second fiscal quarter.

Identifier	problem size(a.u.)	processors	Q2 Time(h)	Weighted Time(h)
122	576	128	2.05	262.4
123	704	256	1.88	481.28
124	800	256	2.43	622.08
125	1056	512	2.71	1387.52
				2753.28

Table 7: Times to compute *problem two*(0.5.1) on machine(0.5.3) with algorithm(0.5.2) measured during the second fiscal quarter.

DRAFT

for the memory access. Thus Eq. (5) is limited by the speed of the cpu-memory bus -e.g. how fast G_c can be moved in and out of memory.

However, the Green function updating may be delayed, until a sufficiently large number of changes, call this number nb, have been accumulated. Only after nb changes in the HHS fields is the whole Green function matrix updated. If these nb accepted changes are labeled sequentially, then Eq. (5) becomes

$$G^{nb}_{c_{\sigma,ij}} = G^{1}_{c_{\sigma,ij}} + \sum_{N=1}^{nb-1} [G^{N}_{c_{\sigma,in(N)}} - \delta_{in(N)}][e^{-2\alpha s_{n(N)}} - 1] G^{N}_{c_{\sigma,N(n)j}} / R^N \tag{6}$$

where the sum on n is over the indices of the accepted changed field and G^N_c and R^N represent the corresponding Green's functions and Metropolis ratio before the change has been accepted. Again, this may be written as a series of vector outer products. If one identifies

$$A^N_{\sigma,i} = \left(G^{N}_{c_{\sigma,in(N)}} - \delta_{in(N)}\right)\left(e^{-2\alpha s_{n(N)}} - 1\right)/R^N \tag{7}$$

$$B^N_{\sigma,j} = G^{N}_{c_{\sigma,n(N)j}} \tag{8}$$

then

$$G^{nb}_{c_{\sigma,ij}} = G^{1}_{c_{\sigma,ij}} + \sum_{N=1}^{nb-1} A^N_{\sigma,i} B^N_{\sigma,j} \tag{9}$$

This can be collapsed into a matrix product of slender rectangular matrices which may be performed efficiently with the BLAS call *dgemm()*.

Since each A^N and B^N depends on G^N_c, this step is pointless unless some way may be found to calculate A^N and B^N for each N quickly; i.e. which scales like $\mathcal{O}(N)$. This is possible, since Eqs. (7) and (8) don't require the entire Green function, but rather only one column or row. This efficient updating algorithm relies on the ability to calculate and store (in A^N and B^N) the Nth row and column of the updated Green function, and delay the full Green function update. To do this, the algorithm is re-written in terms of the columns and rows

$$A^N_{\sigma,i} = \left(col^N_{\sigma,i} - \delta_{in(N)}\right)\left(e^{-2\alpha s_{n(N)}} - 1\right)/R^N \tag{10}$$

$$B^N_{\sigma,j} = row^N_{\sigma,j} \tag{11}$$

$$col^M_{\sigma,i} = col^1_{\sigma,i} + \sum_{N=1}^{M-1} A^N_{\sigma,i} B^N_{\sigma,n(N)} \tag{12}$$

$$row^M_{\sigma,i} = row^1_{\sigma,i} + \sum_{N=1}^{M-1} A^N_{\sigma,n(N)} B^N_{\sigma,j} \tag{13}$$

Furthermore, the row and column information is needed only to calculate the corressponding A^N and B^N, and may be discarded afterwards. A similar equation is used to calculate the diagonal elements of G_c^N needed for the calculation of R^N.

$$diag_{\sigma,i}^M = G_{c_{\sigma,ii}}^1 + \sum_{N=1}^{M-1} A_{\sigma,i}^N B_{\sigma,j}^N \tag{14}$$

Together, Eqs. (9)–(14) form a very efficient Green function update. Eqs. (10) and (11) only require $\mathcal{O}(N)$ operations, Eqs. (12) and (13) require only $\mathcal{O}(N \times nb)$ operations. Of the equations involved in the Green function update, only Eq. (9) has $\mathcal{O}(N^2)$ operations; however, they have many FLOPs per memory access.

Note that nb must not be too large. As nb approaches N, Eqs. (12) and (13) will begin to scale like $\mathcal{O}(N^2)$. I.e. the cpu time required to account for the dependences noted above will eventually excede that required to update the Green's function, Eq. (9). So, as nb approaches N, this process is likely to become less efficient. Fortunately, an $1 \ll nb \ll N$ is sufficient to overcome the caching problem in the Green's function update. Furthermore, Eqs. (12) and (13) will always have many more FLOPs than memory accesses, so this algorithm is expected be superior to the original implementation of Hirsch and Fye.

0.6.2 Data

Q4 Collected Data	Problem(0.5.1)	Problem(0.5.1)
source codes	yes	yes
build information	yes	yes
runtime information	yes	yes
total walltime	yes	no

Table 8: Information collected during Q4 for the computed DCA-QMC problems. Please see sec[0.2.4] for description. Only the algorithm changed during the fiscal year. Only one of the benchmark problems was rerun with the enhanced code. The results of that enhanced run are reported.

0.7 Results

Scientific results of first physical problem The results of this case study contributed to a complete characterization of the effective pairing interaction that leads to superconductivity in the 2D Hubbard model of cuprate high-Tc superconductors. This study allowed us to examine the frequency and momentum dependence of the eigenfunction that corresponds to the leading eigenvalue of the effective pairing interaction in the 2D Hubbard model for a specific set of parameters as a function of

Identifier	problem size(a.u.)	processors	Q4 Time(h)	Weighted Time(h)
101	240	8	2.87	22.96
102	240	8	2.82	22.56
103	240	8	2.78	22.24
104	240	8	2.79	22.32
105	240	16	1.33	21.28
106	240	32	0.64	20.48
107	288	64	0.39	24.96
108	384	64	0.65	41.6
109	480	128	0.64	81.92
110	576	128	0.94	120.32
111	768	256	0.98	250.88
112	864	256	1.17	299.52
113	1056	512	0.70	358.4
				1309.44

Table 9: Measured time to compute *problem one*(0.5.1) on machine(0.5.3) with the enhanced algorithm(0.6.1).

temperature. The momentum dependence of this eigenfunction was found to vary as $cos(kx) - cos(ky)$ and its frequency dependence is determined by the spin 1 particle-hole continuum. Given these results together with the information of additional runs in different parameter regions, we were able to conclude [2] that the pairing interaction is attractive for electrons located on nearest-neighbor sites in a spin singlet state and that it is retarded on a time scale set by the antiferromagnetic spin flcutuations. The strength of the pairing interaction measured by the size of the leading eigenvalue peaks when the strength of the Coulomb repulsion between electrons is similar to the width of the electronic band. The pairing strength is also found to increase as holes are removed from the sytem.

Optimization of Fourier-transform by Kronecker products In our implementation of the Hirsch-Fye quantum Monte Carlo algorithm, we periodically perform measurements of two-particle correlation functions. This entails a double Fourier-transform of the single-particle Green's function matrix from space-time to reciprocal space and frequency. In the Q2 version of the code, both Fourier-transforms were performed simultaneously in a single unitary transform F. With the number of time slices N_l and the number of spatial sites N_c, F can be represented as an $N \times N$ matrix where $N = N_l * N_c$. Hence, this Fourier-transformation scaled as $\mathcal{O}(N^3)$. In the optimized version of the code, this combined Fourier-transform F is replaced by a Kronecker-product, $F = kron(A, B)$. Here, A is a $N_l \times N_l$ matrix which transforms from time to frequency, while B is a $N_c \times N_c$ matrix which transforms from real space to reciprocal space. The result is a Fourier-transformation which scales as $O((N_c + N_l) * (N_c * N_l)^2)$, i.e. much better than the original $O((N_c * N_l)^3)$. The

improvement in speed is documented by the reduction in time spent in the subroutine *meas()*:

```
Q4:
 cpu  ,time=   2523.1184966814162
 geod ,time=   2.6031148318584201
 date ,time=   817.6950566194721
 meas ,time=   1694.0317702389382
 put  ,time=   3.9090832300885268
 sumup,time=   1.02522124052484287E-4

Q2:
 cpu  ,time=   13874.31194784071
 geod ,time=   2.6029983185844685
 date ,time=   8551.5687524601908
 meas ,time=   5199.7460792477796
 put  ,time=   105.49880338938056
 sumup,time=   9.62212379818083718E-5
```

The runtime improvement from the delayed updating reflects itself in the time spent in *date()*.

Performance enhancement Table[6] and Table[9] refer to the original and enhanced times to compute the dca-qmc *problem one*. A comparison of the total weighted runtimes asserts the Joule software effectiveness metric for this application.

Problem	System	Enhancement
Pairing interaction, 2D Hubbard Model (sec0.5.1)	Cray X1E, *phoenix.ccs.ornl.gov*	**performance**, $\sim 74.03\%$

Table 10: FY06 Joule performance enhancement for the DCQ-QMC code.

DRAFT

ENZO: Incite and Novel Computational Impact on Theory and Experiment

0.8 Q2 Information

0.8.1 Problems

The universe is homogeneous and isotropic on scales exceeding one billion light years, but on smaller scales it is clumpy, exhibiting a hierarchy of structures which includes individual galaxies, groups and clusters of galaxies, and superclusters of galaxies. Understanding the origin and cosmic evolution of these structures is the goal of *cosmological structure formation–CSF*. It is now understood that CSF is driven by the gravitational clustering of dark matter, the dominant mass constituent of the universe. In addition, the rate at which structure develops in the universe depends upon its expansion rate, which in turn is influenced by dark energy whose existence was revealed in 1998 by the discovery that the expansion rate is accelerating. The quantitative study of CSF is thus a direct route to studying two of the most mysterious substances in modern physics: dark matter and dark energy.

The part of the universe that astronomers can see is made up of ordinary baryonic matter. Thus, in order to make contact with observations, we must simulate the detailed dynamics and thermodynamics of the cosmic plasma—mostly hydrogen and helium—under the influence of dark matter and dark energy, as well as ionizing radiation backgrounds. Such simulations are called hydrodynamic cosmological simulations, and are the type we consider here. CSF is inherently nonlinear, multidimensional, and involves a broad range of physical processes operating on a range of length- and time-scales. Large scale numerical simulation is the only means we have of studying it in any detail.

In this study we will apply the ENZO hydrodynamic cosmology code developed at UC San Diego to simulate two problems in cosmological structure formation. The

DRAFT

first is to simulate the distribution and evolution of the diffuse intergalactic plasma on large scales. The second is to simulate the formation and evolution of galaxies which condense out of this plasma in the same volume. The two simulations together will paint a complete picture of the history of baryons in the young universe, as well as put new constraints on the mass of the dark matter particle. While the physical equations we solve are the same in both cases, the spatial and temporal discretizations are quite different, and therefore present two very different challenges for analysis and optimization. The first experiment, henceforth called the "Unigrid" experiment, employs a large, uniform 3D Cartesian mesh for discretizing the equations of hydrodynamic cosmology. The second experiment, henceforth called the "AMR" experiment, employs structured adaptive mesh refinement (SAMR) to provide higher spatial and temporal resolution in galaxy-forming regions. While the first presents a static work load to the computer, the second is highly dynamic and increasing. ENZO has been designed to operate in both modes, although how efficiently is the subject of this study.

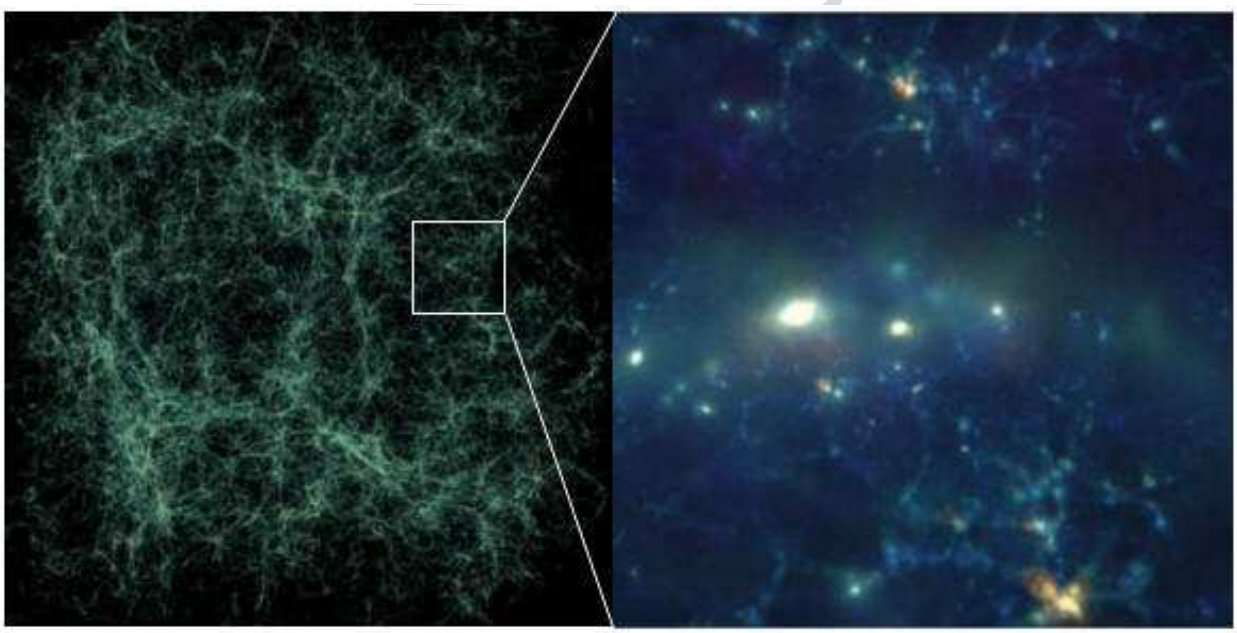

Figure 3. ENZO simulation of galaxy formation and the intergalactic medium. **Left**: Baryons in 80 Mpc survey volume simulated with 1 billion cells and particles on SDSC Blue Horizon. **Right**: Zoom in on young galaxies using adaptive mesh refinement (AMR). Runs similar to these form the baseline runs for this study.

To simulate the structure of the diffuse intergalactic medium, a spatial dynamic range of 10^3 per dimension in 3D is a bare minimum, but 10^4 is ideally needed given the range of important scales. For reasons analogous to turbulence simulations (uniform sampling), we prefer to use uniform grids for this application. The current state-of-the-art using ENZO in uniform grid mode is a 2048^3 simulation currently running on NERSC's Seaborg system under this year's INCITE award. When completed,

it will be the largest, most comprehensive simulation of the intergalactic medium ever carried out. This calculation requires 4 TB of memory, and we estimate it will consume 1.4 M CPU-hours. While there is nothing to be done about the memory requirement, we hope that through optimization we can increase ENZO's execution rate and thereby reduce the CPU time.

To simulate the formation of collapsed structures like galaxies or galaxy clusters, a bare minimum is a spatial dynamic range of 10^4 per dimension, but 10^5 is really needed. This would allow, for example, resolving the internal structure of every galaxy (10 kiloparsecs) simulated within a volume the size surveyed by the Sloan Digital Sky Survey (1 gigaparsec). Anything less either under-resolves galaxies or under-samples large scale structure. Thus, 10^5 dynamic range is a fidelity threshold in cosmological simulations of galaxy formation and large scale structure. This resolution is only needed where galaxies form, not everywhere in space, and therefore adaptive mesh refinement (AMR) methods are ideally suited to achieving this.

The current state-of-the-art is an ENZO AMR simulation with a root grid of size 512^3 and 7 levels of refinement by two, for a spatial dynamic range of $512 \times 2^7 = 6.5 \times 10^4$. We have found that the primary limitation for successfully running simulations of this kind is the memory-per-node, with total system memory being of secondary importance. The reason for this is that AMR places dynamic demands on node CPU and memory resources despite our careful attention to load balancing. Nodes with a large amount of shared memory are more forgiving to the occasional memory requirement spikes that occur. For this reason, clusters with large memory nodes are vastly more attractive than small memory nodes. For this reason we have chosen the POWER5 system Bassi at NERSC to carry out our numerical experiments. A second limitation in carrying out large AMR simulations efficiently is traversing the grid hierarchy efficiently when the number of grid patches becomes very large. The above mentioned simulation generates over $350,000$ unique grid patches at all levels of resolution. We hope to speed this up through analysis and optimization.

Generating Initial Data So far, we have described how the dark matter and baryonic species are evolved in an expanding universe. Initial conditions are completely specified by assuming the matter fluctuations are linear in amplitude at the starting time (redshift), and can be described as a Gaussian random field with a given power spectrum. Both of these assumptions characterize to a high degree of accuracy the temperature anisotropies observed in the cosmic microwave background radiation field as mapped by the Wilkinson Microwave Anisotropy Probe (WMAP). We literally read our initial conditions from the sky. The cosmological model thus deduced is well described by a cold dark matter dominated universe with dark matter, baryon, and dark energy fractions of 0.23, 0.04, and 0.73 times the closure density for a Hubble constant of 72 km/sec/Mpc. This model is referred to as the ΛCDM concordance model, as it is consistent with all known measurements of cosmic structure at a variety of epochs.

Validating the Science The validity of the concordance model parameters and matter power spectrum is now firmly established observationally; i.e., the inputs to our simulations are not in serious doubt. The outputs consist of realizations of cosmic structure formation including baryonic fields which can be compared to observations at various epochs of cosmic evolution. Since looking out into space corresponds to looking backward in time, the time history of the observed universe can be directly compared to the time history of our simulations. These comparisons are both structural and statistical.

The distribution of intergalactic gas is observationally probed along lines of sight to distant quasars by means of high resolution absorption spectra. In particular, the so-called Lyman alpha forest spectra probe the distribution of neutral hydrogen along path lengths of hundreds of Megaparsecs through the intergalactic medium. There are many hundreds of such high resolution spectra now available. Our simulations of the diffuse intergalactic medium are analyzed by creating synthetic absorption spectra to imaginary quasars at different redshifts. The statistical properties of our synthetic spectra are compared to the statistical properties of the observed spectra. He have shown that the two agree for the concordance model parameters for a specific value for the ultraviolet radiation background intensity and spectral shape ([59]). Our unigrid experiment will extend the range of path lengths over which the data is compared, and thereby measure the matter power spectrum on scales below that obtained by the WMAP satellite.

The distribution of high redshift galaxies is not as well determined as the intergalactic gas, since the galaxies must be detected in emission, not absorption. They are much fainter that high redshift quasars, and not as many have been detected. Still, some information is known about high redshift galaxies from limited surveys, such as their mass, luminosity, star formation rates, and 2-point correlation function. Our AMR simulation is being done in anticipation of much larger surveys in the future which will improve these measurements substantially. We will validate the results of our simulation by computing 1-point and 2-point statistics of the dark matter and galaxy distributions and comparing them with available data.

Problem One: Structure of the Intergalactic Medium (Unigrid) We simulated the evolution of a volume of the universe 154 Megaparsecs ($\approx$ 500 Mega-light-years) on a side assuming the concordance model parameters ($\Omega_m = 0.27$, $\Omega_b = 0.044$, $\Omega_\Lambda = 0.73$, $h = 0.72$, $\sigma_8 = 0.9$, $n_s = 1$). Initial conditions for dark matter and baryonic density and velocity perturbations were generated using the routine `EnzoInit` for redshift=60 (approximately 10 million yr after Big Bang (ABB)) and evolved with ENZO in unigrid mode to redshift 3 (approximately 5 billion years ABB). A total of 1024^3 dark matter particles were evolved using a force mesh of 1024^3 cells. The same mesh was used to evolve the 6 baryonic fluid species concentrations, temperature, and velocity assuming the ionizing radiation background given by Madau, Haardt & Rees (2001). The simulation was run on 512 processors of NERSC Bassi.

Problem Two: High Redshift Galaxies (AMR) The physical design of AMR baseline calculation was in all respects identical to the unigrid calculation with the exception of the size of the initial grid and the number of dark matter particles evolved. Instead of a 1024^3 uniform grid and 1024^3 dark matter particles, the initial data was generated for a 512^3 grid and 512^3 particles. Whereas in the unigrid calculation the spatial resolution was fixed, in the AMR run we allowed the mesh to refine locally to a maximum depth of four levels, where each level is refined by a factor of two relative to its parent. The cosmological parameters, power spectrum, and physical domain size were kept the same as before. In addition, the same random seed was used to compute the Gaussian random field of density fluctuations, which means that the pattern of fluctuations is identical in both simulations. The reason for doing this is so that we can compare structures computed at different mass and spatial resolution between the two simulations. The unigrid simulation will posses better mass and spatial resolution in the diffuse intergalactic gas, while the AMR simulation will possess 8 times poorer mass resolution (512^3 particles versus 1024^3 particles) in collapsed objects (galaxies), but 8 times better spatial resolution (1024 versus 512×2^4).

0.8.2 Algorithm

Equations The hydrodynamical equations for mass, momentum and energy conservation in an expanding FRW universe with comoving coordinates are

$$\frac{\partial \rho_b}{\partial t} + \vec{\nabla} \cdot (\rho_b \vec{v}_b) + 3\frac{\dot{a}}{a}\rho_b = 0, \tag{15}$$

$$\frac{\partial(\rho_b v_{b,i})}{\partial t} + \vec{\nabla} \cdot [(\rho_b v_{b,i})\vec{v}_b] + 5\frac{\dot{a}}{a}\rho_b v_{b,i} = -\frac{1}{a^2}\frac{\partial p}{\partial x_i} - \frac{\rho_b}{a^2}\frac{\partial \phi}{\partial x_i}, \tag{16}$$

$$\frac{\partial e}{\partial t} + \vec{\nabla} \cdot (e\vec{v}_b) + p\vec{\nabla} \cdot \vec{v}_b + 3\frac{\dot{a}}{a}(e+p) = \Gamma - \dot{E}, \tag{17}$$

where ρ_b, p and e are the baryonic density, pressure and specific internal energy defined in the proper reference frame, $\vec{v}_b$ is the comoving peculiar baryonic velocity, ϕ is the comoving gravitational potential that includes baryonic plus dark matter contributions, $a = 1/(1+z)$ is the cosmological scale factor, and $\dot{E}$ and Γ are the microphysical cooling and heating rates.

The equations for collisionless dark matter in comoving coordinates are

$$\frac{d\vec{x}_d}{dt} = \vec{v}_d, \tag{18}$$

$$\frac{d\vec{v}_d}{dt} = -2\frac{\dot{a}}{a}\vec{v}_d - \frac{1}{a^2}\vec{\nabla}\phi. \tag{19}$$

The baryonic and dark matter components are coupled through Poisson's equation for the gravitational potential

$$\nabla^2\phi = 4\pi G a^2(\rho - \bar{\rho}), \tag{20}$$

DRAFT

where $\rho = \rho_b + \rho_d$ is the total density and $\bar{\rho} = 3H_0^2\Omega_0/(8\pi G a^3)$ is the proper background density of the universe.

The cosmological scale factor $a(t)$ is given by Einstein's equation

$$\frac{da}{dt} = H_0 \left[\Omega_M(\frac{1}{a} - 1) + \Omega_\Lambda(a^2 - 1) + 1\right]^{1/2} \tag{21}$$

where $\Omega_M = \Omega_b + \Omega_d$ is the density parameter including both baryonic and dark matter contributions, $\Omega_\Lambda = \Lambda/(3H_0^2)$ is the density parameter attributed to the cosmological constant Λ, and H_0 is the present Hubble constant.

In addition to the usual hydrodynamic equations (15) – (17), we must also solve equivalent mass conservation equations for the densities ρ_i of each of the nine separate atomic and molecular species that we track

$$\frac{\partial \rho_i}{\partial t} + \vec{\nabla} \cdot (\rho_i \vec{v}_b) + 3\frac{\dot{a}}{a}\rho_i = \pm \sum_j \sum_l k_{jl}(T)\rho_j \rho_l \pm \sum_j I_j \rho_j, \tag{22}$$

where the signs of each term on the right–hand–side depend on whether the process creates or destroys the species ρ_i. The $k_{jl}(T)$ are rate coefficients for the two body reactions and are functions of the gas temperature T. Explicit analytic fits for these coefficients over a broad range of temperatures, and a general discussion of the relevant chemical reactions, can be found in Abel et al. (1996). In all, we include 28 rate coefficients, one for each of the chemical reactions shown in Appendix A. The I_j in equation (22) are integrals due to photoionizations and photodissociations

$$I_j = \int_{\nu_{0,j}}^{\infty} 4\pi\sigma_j(\nu)\frac{\mathcal{I}(\nu)}{h\nu}d\nu, \tag{23}$$

where $\mathcal{I}(\nu) = \mathcal{F}(\nu)/4\pi$ is the intensity of the radiation field, $\mathcal{F}(\nu)$ is the flux, $\sigma_j(\nu)$ are the cross-sections for the photoionization and photodissociation processes, and $\nu_{0,j}$ are the frequency thresholds for the respective processes. We note that the nine equations represented by (22) are not all independent. The baryonic matter is composed of hydrogen and helium with a fixed primordial hydrogen mass fraction of f_H. Hence we have the following three conservation equations

(24) hydrogen nuclei: $$\rho_H + \rho_{H^+} + \rho_{H^-} + \rho_{H_2^+} + \rho_{H_2} = \rho_b\ f_H,$$

(25) helium nuclei: $$\rho_{He} + \rho_{He^+} + \rho_{He^{++}} = \rho_b\ (1 - f_H),$$

(26) charge conservation: $$\rho_{H^+} - \rho_{H^-} + \frac{1}{2}\rho_{H_2^+} + \frac{1}{4}\rho_{He^+} + \frac{1}{2}\rho_{He^{++}} = m_H\ n_e,$$

where n_e is the number density of free electrons and m_H the proton mass.

To complete the set of equations (15) – (17), we must also specify the equation of state appropriate for an ideal gas

$$e = \frac{p}{\gamma - 1} = \frac{k_B T}{\gamma - 1}\sum_{i=1}^{9} n_i, \tag{27}$$

DRAFT

where $\gamma = 5/3$ is the ratio of specific heats for the baryonic matter, k_B is boltzmann's constant, T is the gas temperature, and n_i are the number densities for each of the different species. We also need to provide the necessary cooling $\dot{E}$ and heating Γ functions to the right-hand-side of equation (17)

$$\dot{E} = \dot{E}_{Comp} + \sum_j \sum_l \dot{e}_{jl}(T)\rho_j\rho_l, \tag{28}$$

$$\Gamma = \sum_j J_j(\nu)\rho_j, \tag{29}$$

where $\dot{E}_{Comp}$ is the Compton cooling (or heating) due to interactions of free electrons with the cosmic microwave background radiation, and $\dot{e}_{jl}(T)$ are the cooling rates from two-body interactions between species j and l. The $J_j(\nu)$ are integrals due to photoionizing and photodissociating heating

$$J_j(\nu) = \int_{\nu_{0,j}}^{\infty} 4\pi\sigma_j(\nu)\mathcal{I}(\nu)\frac{(h\nu - h\nu_{0,j})}{h\nu}d\nu. \tag{30}$$

We include a total of fourteen processes in the cooling function and three processes for heating. The physical mechanisms and mathematical expressions for each process are given in [49].

Discussion Matter in the universe is of two basic types: ordinary "baryonic" matter composed of nucleons and electrons out of which stars and galaxies are made, and non-baryonic "dark" matter of unknown composition, which is nevertheless known to be the dominant mass constituent in the universe on scales of galaxies and larger. ENZO self-consistently simulates both components, which evolve according to different physical laws and therefore require different numerical algorithms. The equations solved by ENZO follow.

Baryonic matter is evolved using a finite volume discretization of the Euler equations of gas dynamics cast in a frame which expands with the universe including energy source and sink terms due radiative heating and cooling processes, as well as changes in ionization state of the gas [49]. We use the Piecewise Parabolic Method (PPM), which is a higher-order Godunov scheme developed by Colella and Woodward for ideal gas dynamics calculations. The species abundances for H, H+, He, He+, He++, and e- are solved out of equilibrium by integrating the rate equations including radiative and collisional processes. Radiation fields are modeled as evolving but spatially homogeneous backgrounds using published prescriptions.

Dark matter is assumed to behave as a collisionless phase fluid, obeying the Vlasov-Poisson equation. Its evolution is solved using particle-mesh algorithms for collisionless N-body dynamics [58]. In particular, we use the spatially second order-accurate Cloud-in-Cell (CIC) formulation, together with leapfrog time integration, which is formally second order-accurate in time. Dark matter and baryonic matter interact only through their self-consistent gravitational field. The gravitational potential is

computed by solving the Poisson equation on the uniform or adaptive grid hierarchy using Fast Fourier transform and multigrid techniques. In generic terms, ENZO is a 3D hybrid code consisting of a multi-species hydrodynamic solver for the baryons coupled to a particle-mesh solver for the dark matter via a Poisson solver.

Matter evolution is computed in a cubic domain of length $L = a(t)X$, where X is the domain size in comoving coordinates, and $a(t)$ is the homogenous and isotropic scale factor of the universe which is an analytic or numerical solution of the Friedmann equation, a first order ODE. For sufficiently large L compared to the structures of interest, any chunk of the universe is statistically equivalent to any other, justifying the use of periodic boundary conditions. The speed of Fast Fourier Transform algorithms and the fact that they are ideally suited to periodic problems make them the Poisson solver of choice given the large grids employed—1024^3 or larger.

CSF simulations require very large grids and particle numbers due to two competing demands: large boxes are needed for a fair statistical sample of the universe; and high mass and spatial resolutions are needed to adequately resolve the scale lengths of the structures which form. For example, in order to adequately simulate the internal structure of galaxies and simultaneously describe their large scale distribution in space (large scale structure), a dynamic range of 10^4 per spatial dimension and 10^9 in mass is needed *at a minimum*, as discussed above.

The Enzo Code The need for higher resolution than afforded by uniform grids motivated the development of ENZO. ENZO uses structured adaptive mesh refinement (SAMR, [50, 52]) to achieve high resolution in gravitational condensations. The central idea behind SAMR is simple to describe but difficult to implement efficiently on parallel computers. While solving the desired set of equations on a coarse uniform grid, monitor the quality of the solution; when necessary, add an additional, finer mesh over the region that requires enhanced resolution. This finer (child) mesh obtains its boundary conditions from the coarser (parent) grid or from other neighboring (sibling) grids with the same mesh spacing. The finer grid is also used to improve the solution on its parent. As the evolution continues, it may be necessary to move, resize or even remove the finer mesh. Even finer meshes may be required, producing a tree structure that can continue to any depth.

To advance our system of coupled equations in time on this grid hierarchy, we use a recursive algorithm. The `EvolveLevel` routine is passed the level of the hierarchy it is to work on and the new time. Its job is to march the grids on that level from the old time to the new time:

DRAFT

```
EvolveLevel(level, ParentTime)
begin
    SetBoundaryValues(all grids)
    while (Time < ParentTime)
    begin
        dt = ComputeTimeStep(all grids)
        PrepareDensityField(all grids, dt)
        SolveHydroEquations(all grids, dt)
        SolveRateEquations(all grids, dt)
        SolveRadiativeCooling(all grids, dt)
        Time += dt
        SetBoundaryValues(all grids)
        EvolveLevel(level+1, Time)
        RebuildHierarchy(level+1)
    end
end
```

Figure 4. ENZO AMR algorithm

Inside the loop which advances the grids on this level, there is a recursive call so that all the levels above (with finer subgrids) are advanced as well. The resulting order of timesteps is like the multigrid W-cycle.

Before we update the hyperbolic gas dynamics equations and solve the elliptic Poisson equation, we must set the boundary conditions on the grids. This is done by first interpolating from a grid's parent and thing copying from sibling grids, where available. Once the boundary values have been set, we solve the Poisson equation using the procedure `PrepareDensityField` and evolve the hydrodynamic field equations using procedure `SolveHydroEquations`. The multispecies kinetic equations are integrated by procedure `SolveRateEquations`, followed by an update to the gas energy equation due to radiative cooling by procedure `SolveRadiativeCooling`. The final task of the `EvolveLevel` routine is to modify the grid hierarchy to the changing solution. This is accomplished via the `RebuildHierarchy` procedure, which takes a level as an argument and modifies the grids on that level and all higher levels. This involves three steps: First, a refinement test is applied to the parent grids of the current level to determine which cells need to be refined. Second, rectangular regions are chosen which cover all of the refined regions, while attempting to minimize the number of unnecessarily refined points. Third, the new grids are created and their values are copied from the old grids (which are deleted) or interpolated from parent grids. This process is repeated on the next refined level until the grid hierarchy has been entirely rebuilt.

DRAFT

0.8.3 Machine

Please refer to the webpage `http://web-dev.nersc.gov/nusers/resources/bassi/`.

0.8.4 Data

Q2 Collected Data	Problem One(0.8.1)	Problem Two(0.8.1)
Source codes	yes	yes
Build information	yes	yes
Runtime information	yes	yes
Total walltime	yes	yes

Table 11: Information collected during Q2 for the computed Enzo problems on machine(0.8.3). Please see sec[0.2.4] for description.

Data and Analysis: *Problem One(unigrid)* The simulation proceeded in two steps because we had to restart the job. The table below summarizes the two runs, referred to as UB_1 and UB_2.

Run	**Timesteps**	**Redshifts**	**Wall clock (sec)**	**CPU-hrs**
UB_1	1 to 569	60 to 10	24889	2881.47
UB_2	565 to 1806	10 to 3	61497	7015.82

Table 1. Summary of the unigrid baseline runs.

Figures 5 and 6 below show the accumulated wall-clock-time versus timestep broken down by major code regions, as well as the total wall-clock time, for runs UB_1 and UB_2. The function of the major code regions is summarized in Table 2. A unigrid run in ENZO is treated as an AMR run with no levels of refinement. This means the algorithm is as given in Figure 4, but there is no recursion. Also, the fast sibling neighbor search in `boundary update 1` is omitted. Roughly 50% of this is for the hydrodynamic update of the baryonic fields. The next largest is the radiative cooling calculation, which is subcycled relative to the hydrodynamic timestep. The third most costly step is a boundary update step which exchanges ghost zone information and moves dark matter particles.

DRAFT

Code Region	Function	Routines Called	Comments

Boundary update 1*	Collects ghost zone data for grid patches and moves dark matter particles	`SetBoundaryConditions()` calls
		`grid::SetExternalBoundaryValues()`
		`grid::InterpolateBoundaryFromParent()`
		`grid::CheckForOverlap()`
		`FastSiblingLocatorInitialize()`
		`grid::FastSiblingLocatorAddGrid()`
		`grid::FastSiblingLocatorFindSiblings()`
		`grid::FastSiblingLocatorFinalize()`
			First occurrence in Figure 4, also does fast neighbor search

Self-gravity*	Assigns mass to mesh, solves Poisson equation, calculates accelerations	`PrepareDensityField()` calls
		`DepositParticleMassField()`
		`PrepareGravitatingMassField()`
		`grid::CheckForOverlap()`
		`grid::SolveForPotential()`
		`grid::CopyPotentialToBaryonField()`
		`grid::ComputeAccelerationField()`
		`grid::InterpolateAccelerations()`
			FFT used for unigrid and AMR root grid; multigrid used for subgrids

Hydrodynamics	Solves multispecies Euler equations	`grid::SolveHydroEquations()`	Method: PPM
Rate equations	Integrates rate equations for primordial gas chemistry	`grid::SolveRateEquations()`	Method: BDF (Anninos et al. 1997) Subcycled.
Radiative cooling	Updates energy equation with radiative heating and cooling	`grid::SolveRadiativeCooling()`	Subcycled.

Boundary update 2*	Collects ghost zone data for grid patches and moves dark matter particles	`SetBoundaryConditions()` calls
		`grid::SetExternalBoundaryValues()`
		`grid::InterpolateBoundaryFromParent()`
		`grid::CheckForOverlap()`
			Second occurrence in Figure 4

Rebuild hierarchy*	Controls the AMR grid hierarchy	`RebuildHierarchy()`	
Total	Sum of the above plus time spent doing I/O		

Table 2. Description of code regions instrumented in ENZO. A region with an asterisk(*) invokes message-passing.

DRAFT

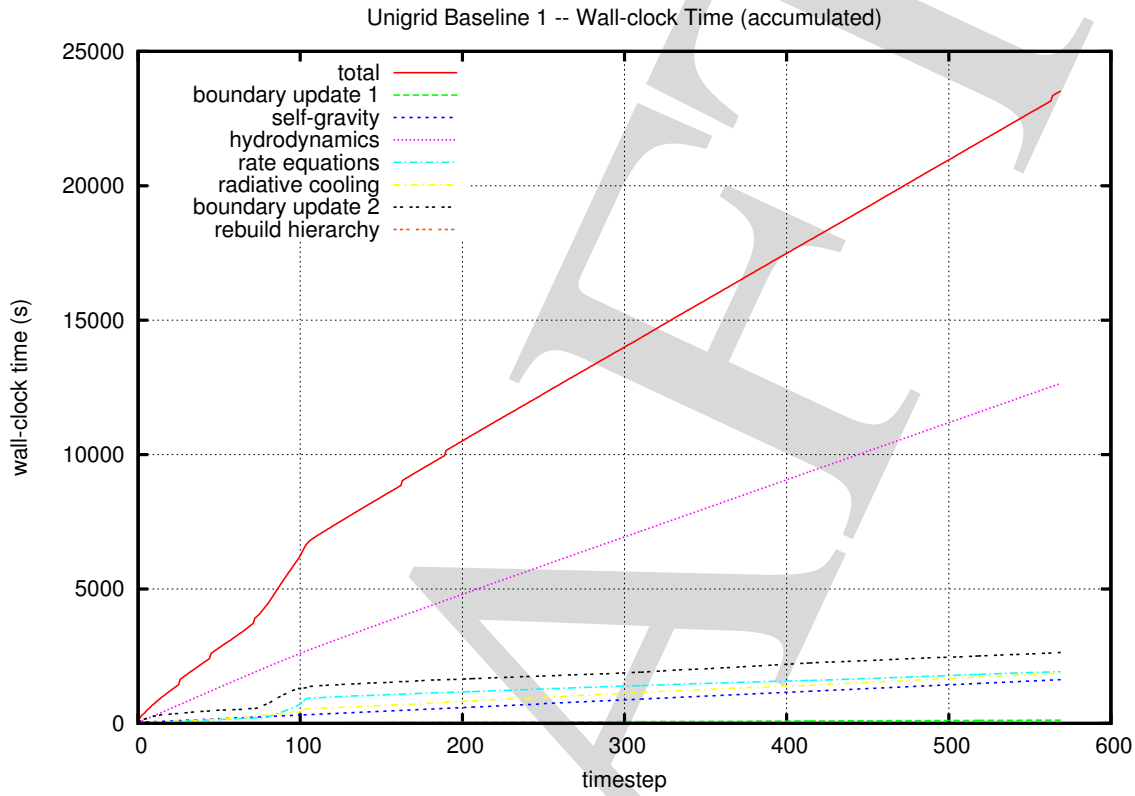

Figure 5. Cumulative wall-clock time versus timestep number for unigrid baseline run UB_1 broken down according to major code regions.

DRAFT

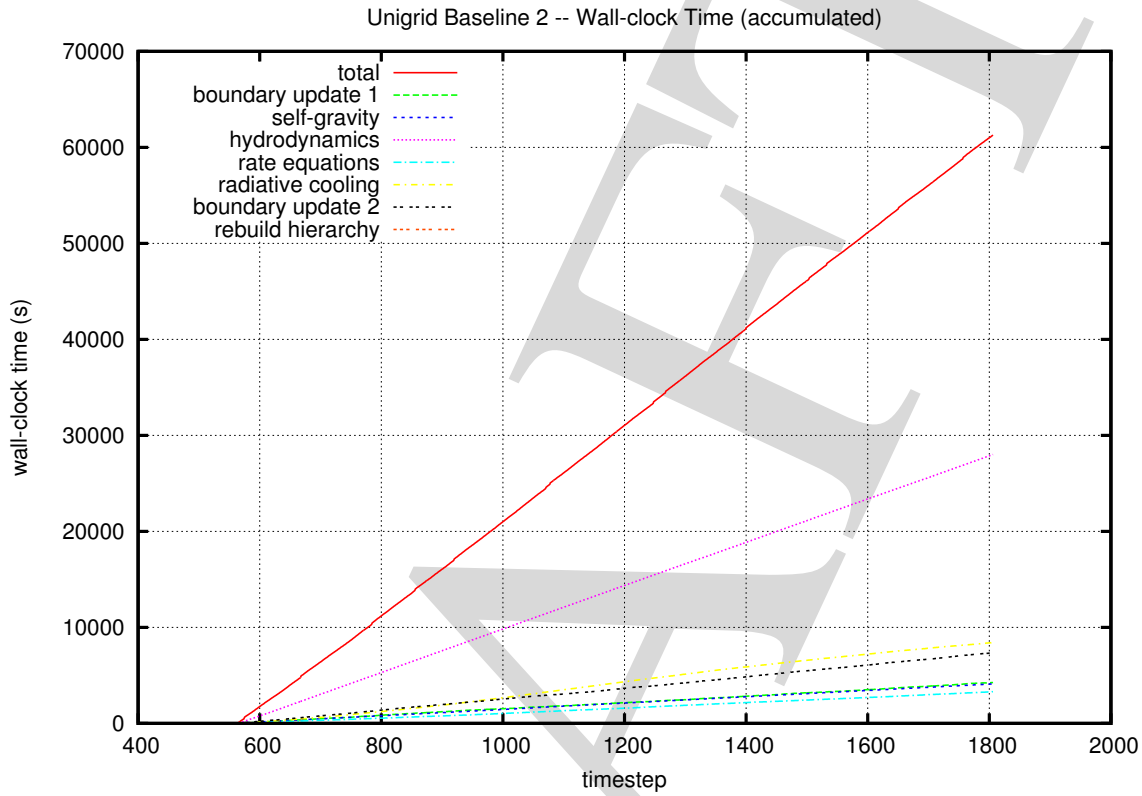

Figure 6. Cumulative wall-clock time versus timestep number for unigrid baseline run UB_2 broken down according to major code regions.

Figures 7 and 8 show the wall-clock time per timestep broken down by code region for runs UB_1 and UB_2. Except for I/O which causes spikes in the total time, the wall-clock time is roughly constant with timestep after an initial transient between timesteps 70-100. The origin of this transient is unknown. It is also not understood what causes the fluctuations in the timing curves for `boundary update 2`, `rate equations`, and `radiative cooling`. The curves for the gas and dark matter dynamics do not show these oscillations. Looking at Figure 7, we see that `radiative cooling` roughly doubles in cost from time step 550 to 1100, thereafter declining. This is probably physical, as the gas begins to be ionized at timestep 291 and becomes nearly fully ionized at timestep 1100. At maximum, the radiative cooling calculation accounts for (8 sec./45 sec) $\approx$ 15% of the cost of the calculation. `Hydrodynamics` and `radiative cooling` account for (30 sec/45 sec) $\approx$ 66% of the total run time. Self-gravity accounts for only about (3 sec/45 sec) $\approx$ 7% of the total run time.

DRAFT

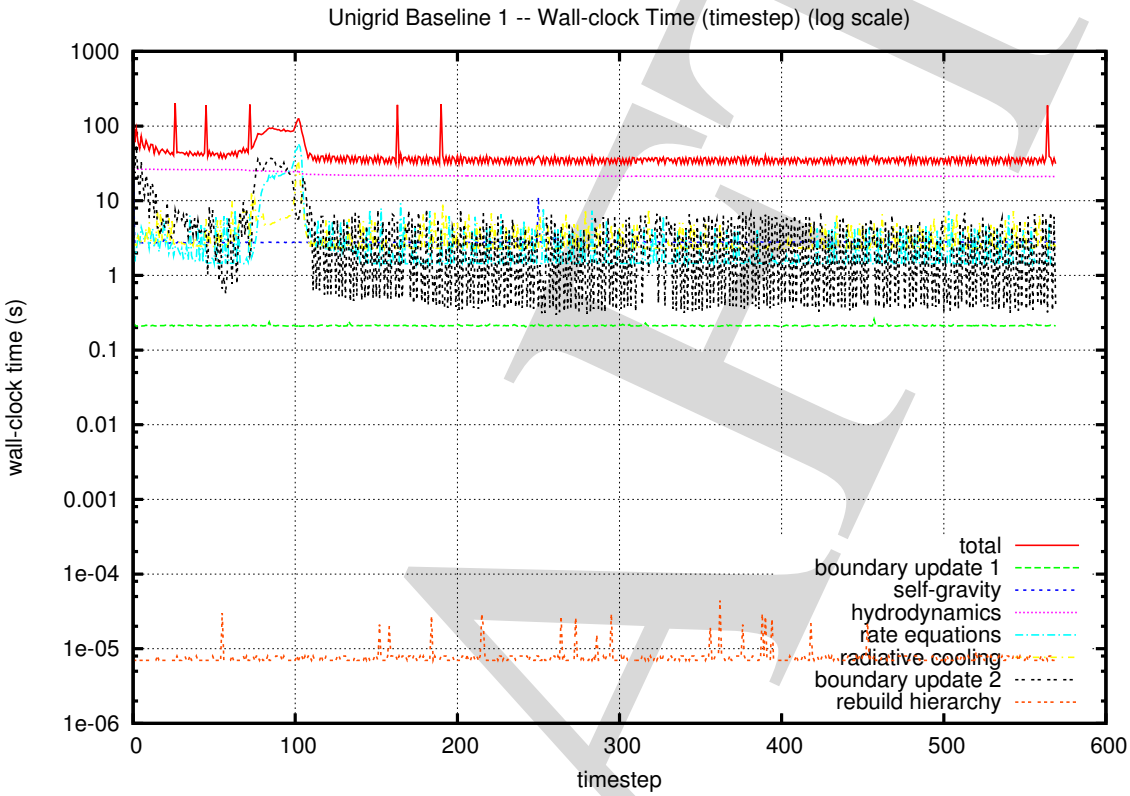

Figure 7. Wall-clock time versus timestep number for unigrid baseline run UB_1 broken down according to major code regions (Table 2). Spikes in total time correspond to I/O events.

DRAFT

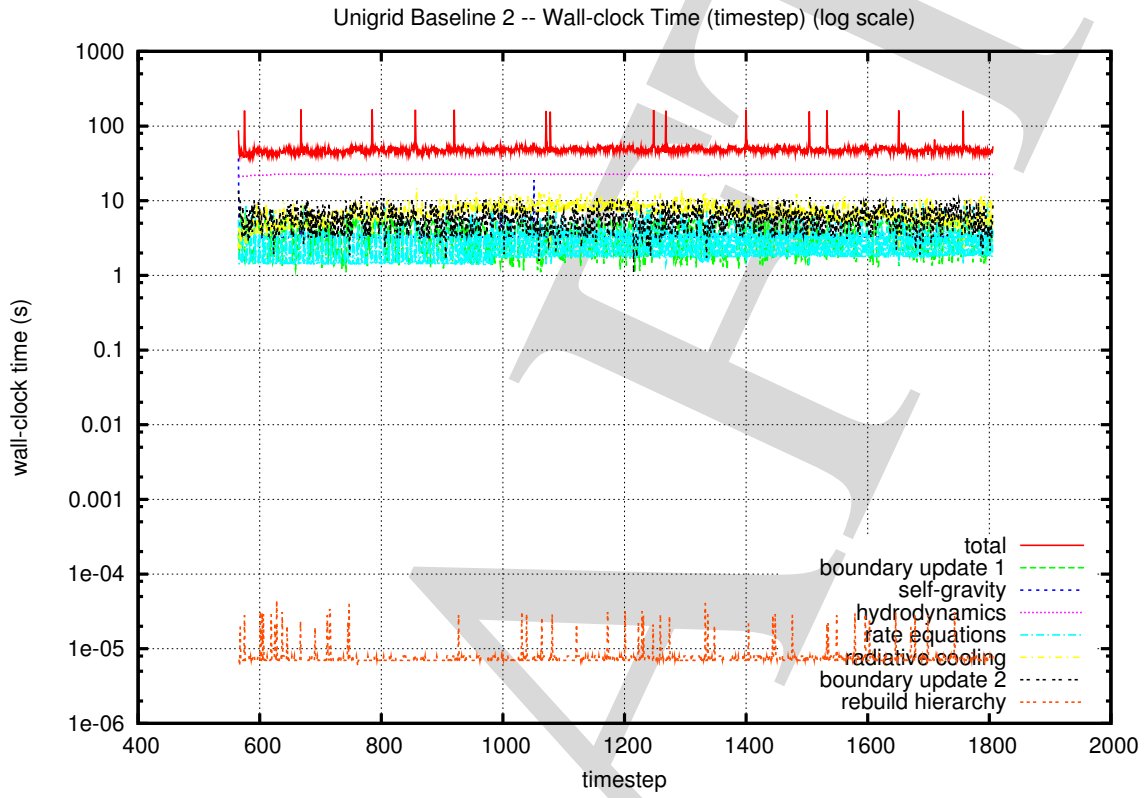

Figure 8. Wall-clock time versus timestep number for unigrid baseline run UB_2 broken down according to major code regions (Table 2). Spikes in total time correspond to I/O events.

Figure 9 shows the CPU time per processor for baseline run UB_2 (UB_1 is similar, and not shown here, but available on the website `http://jbpc.ucsd.edu/ jbordner/ nersc_baseline/`). It would appear that one processor per node is doing about 9% more work than the other seven, but otherwise the workload is roughly uniform across processors.

DRAFT

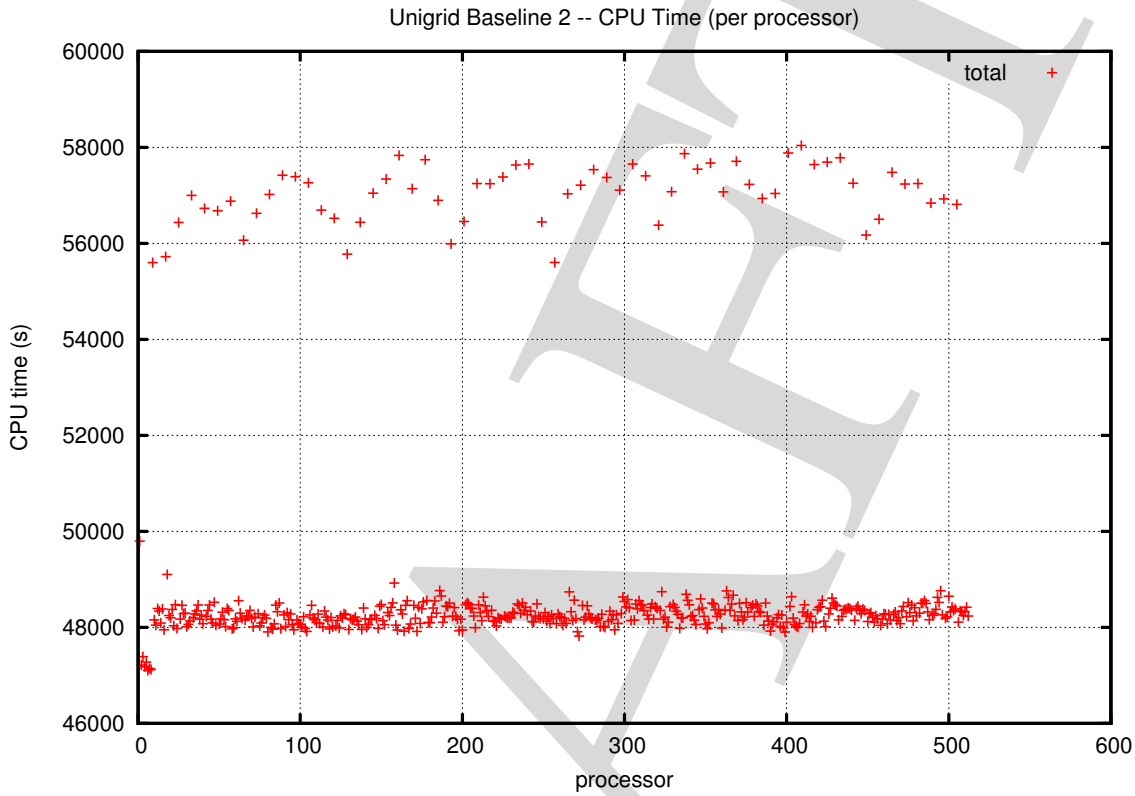

Figure 9. CPU time per processor for run UB_2.

Data and Analysis: *Problem Two(amr)* The computational consequences of using AMR are many. First, we use AMR in CSF to refine regions of the domain undergoing gravitational collapse, which increase in number and complexity as the simulation evolves. Thus both the memory and CPU requirements grow dramatically from the start to the end of the calculation (Fig. 11). While the total number of dark matter particles remains fixed, they are distributed among a growing number of refined grid patches as time goes on. A particle is assigned to the finest resolution grid patch which contains it. The AMR baseline calculation generates $26,000$ grids by the end of the calculation at all levels of resolution (Fig. 10). This is to be compared with 512 grids for the statically decomposed base grid at the start of the calculation. The state data for the baryonic fields grows with the number of grids, and therefore is not a constant. Since the number of grids generated greatly exceeds the number of assigned processors, each processor advances on average $26,000/512 = 51$ grids by the end of the calculation.

Second, the work load is not uniformly distributed. Gravitational clustering produces some regions which have more matter concentrated in them than others. This will translate into more grids and particles in some processors than others. We use a

simple load balancing scheme to redistribute the work load, but the load balancing is not perfect. Moreover, dynamic load balancing is achieved by messaging entire grids with their state data to under-loaded processors, increasing the communication overhead.

Third, we use hierarchical timestepping, which means that a grid at level L is updated roughly 2^L times as frequently as the base grid. The work per processor per root grid timestep is therefore

$$W = \sum_{L=0}^{4} (N_cell_L \times W_{cell} \times 2^L + N_part_L \times W_{part} \times 2^L),$$

where N_cell_L and N_part_L are the number of mesh cells and dark matter particles at level L, W_{cell} and W_{part} is the computational work to update one baryonic cell's density, temperature, velocity, and species concentrations, and dark matter particle's position and velocity, respectively.

Fourth, as particles move through space, they are passed from one grid to another using message passing. In addition, each grid is surrounded by several rows of ghost zones which store boundary values interpolated from parent grids, or copied from sibling grids. These boundary updates must be done before every update step on every level of the hierarchy. They involve a neighbor search in order to determine which grids are contributing boundary data, and finally a message passing step to actually gather the data. In the plots below, the time for all these operations are collected under the rubric `boundary update 1, 2`.

Finally, the solution of the Poisson equation is more complicated than in the unigrid case. As in the unigrid case, once mass is assigned to the root grid, the Poisson equation is solved using 3D Fast Fourier Transforms assuming periodic boundary conditions. To calculate the potential within a subgrid, we first assign the particle mass to mesh, add in the baryon mass which already defined on the mesh, and then solve the Poisson equation with isolating boundary conditions using multigrid relaxation where the potential boundary values interpolated from parent grids, or copied from adjacent sibling grids. Gathering the boundary data and solving the Poisson equation iteratively on each subgrid increases the cost of the Poisson solve dramatically as the number of grids grows.

Processors	**Timesteps**	**Redshifts**	**Q2 Time(s)**	**Weighted Time(h)**
512	1 to 153	60 to 6.4	14130	2010

Table 3. Summary of the AMR baseline run.

We now evaluate the relative cost of these AMR-related steps. Table 3 summarizes

our AMR baseline run AMR_FAST. It ran for a total of 14130 wall-clock seconds on 512 processors (64 nodes) of NERSC bassi. Of that, 12926 was spent in computation, the remainder (1204 s) in start-up and clean-up.

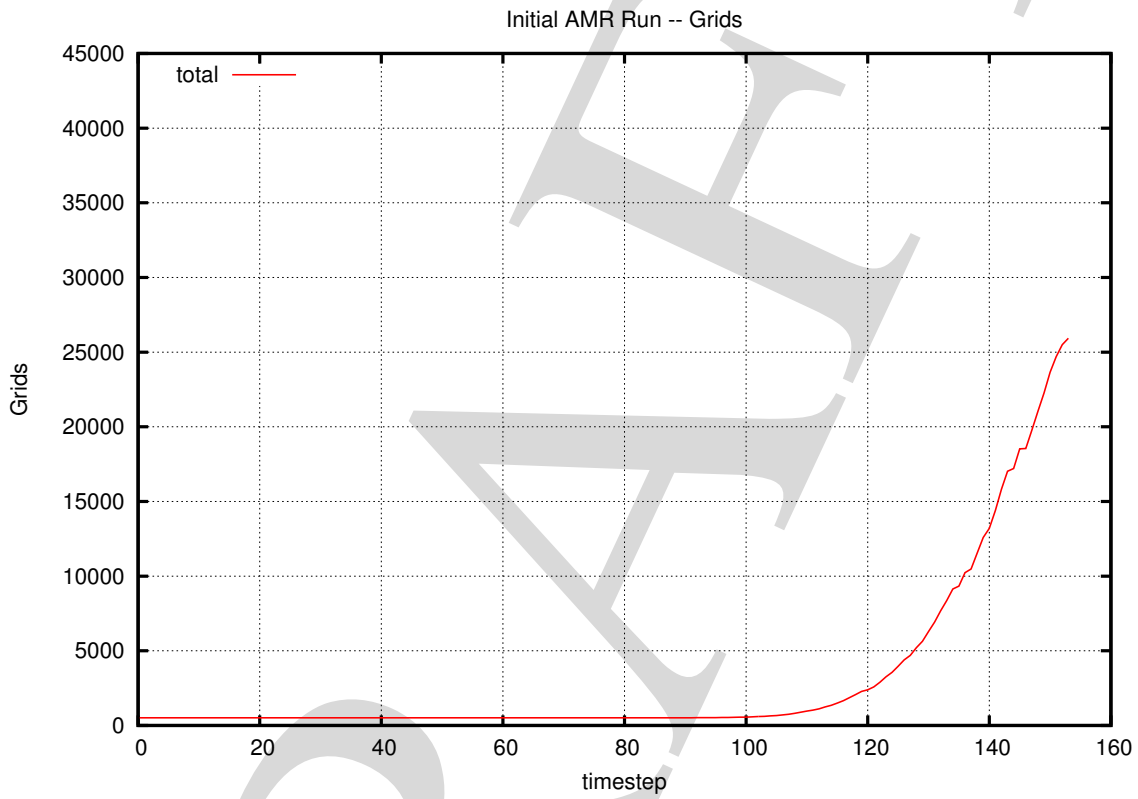

Figure 10. Number of grid patches versus root grid timestep in the AMR_FAST baseline run.

Figure 11 shows the accumulated wall-clock time for run AMR_FAST, broken down by code region. The accumulated time is roughly linear in the number of timesteps during the early unigrid phase, but skyrockets once mesh refinement begins. Figure 12 shows the same data, but the vertical axis is log-scaled. Table 4 shows the accumulated time at the end of the calculation, broken down according to code region, and their percent contribution to the total cost. The largest item in Table 4 is "other" at 33%, which is time spent on things other than computation. We believe this time is dominated by MPI barrier synchronization. If this is the case, then this is an obvious area for optimization.

DRAFT

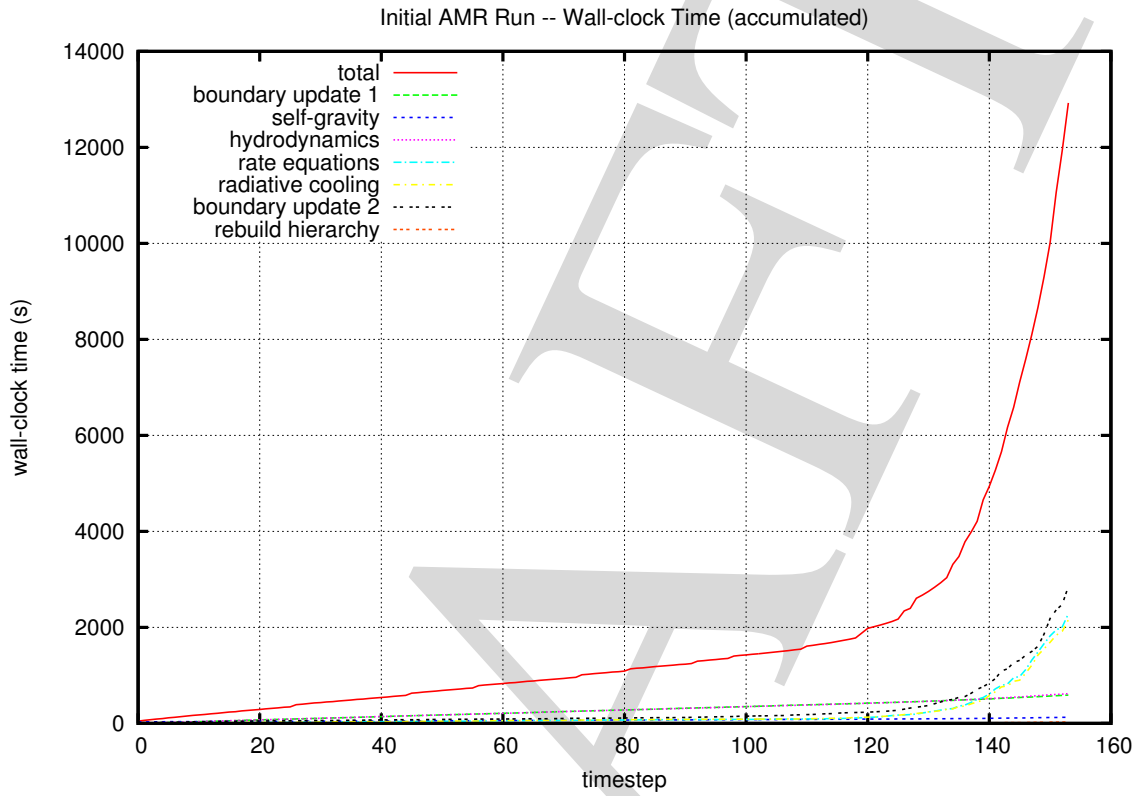

Figure 11. Accumulated wall-clock time for run AMR_FAST, broken down by code region. Accumulated time is roughly linear in the numberof timesteps during the early unigrid phase, but skyrockets once mesh refinement becomes significant at timestep 104.

Region	Time (s)	Percentage of total
Boundary update 1	589	4.6
Self gravity	128	1
Hydrodynamics	620	4.8
Rate equations	2282	17.6
Radiative cooling	2155	16.7
Boundary update 2	2853	22.1
Rebuild hierarchy	30	0.2
Other	4269	33
Total	12926	100

Table 4. Breakdown of time spent in various code regions for AMR_FAST baseline.

Of the remaining 67%, the most time-consuming physics calculations are `rate equations` and `radiative cooling`, which together account for 34.3% of total. Prior to the onset of mesh refinement (timestep 88), hydrodynamics dominates the cost of the physics

portion of the calculation, just as in the unigrid baseline. However, by the end of the calculation, hydrodynamics accounts for only 4.8% of the total runtime. The significant thing to notice is that the cost of `rate equations` and `radiative cooling` increases much faster than the hydrodynamics cost after mesh refinement begins, even though the number of cells to be updated is the same. A possible explanation for the disproportionate time spent in these routines is that the multispecies chemistry and cooling, which is solved out of equilibrium using a subcycling technique, is subcycling more and more as the mesh is refined. This is another area where optimization might be needed.

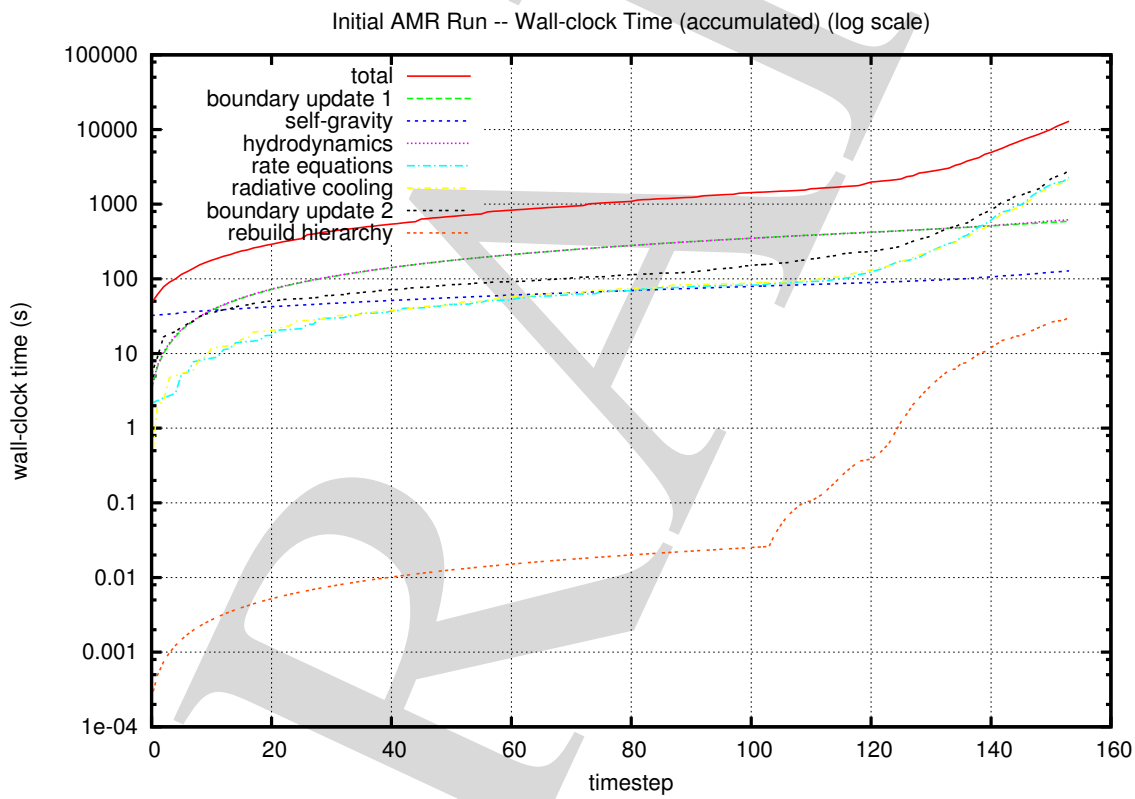

Figure 12. Same as Fig. 11, but log(accumulated wall-clock time).

Figure 12 shows the time spent in the various code regions as a function to root grid timestep. The jump in cost for procedure "Rebuild Hierarchy" at timestep 104 signals the onset of significant mesh refinement. At first, there are only the 512 root grid tiles–one per processor–but by timestep 153 there are nearly 26,000 at four levels of resolution. The cost to advance the root grid by one timestep remains is dominated by hydrodynamics until timestep 104, when it is overtaken by `boundary update 2`, `rate equations` and `radiative cooling`. Together, they account for about 56% of the total cost. Routines `hydrodynamics`, `boundary update 1`, and `self-gravity` together account for only about 10% of the total cost. Rebuilding the AMR grid

hierarchy is a negligible cost, at about 0.2%.

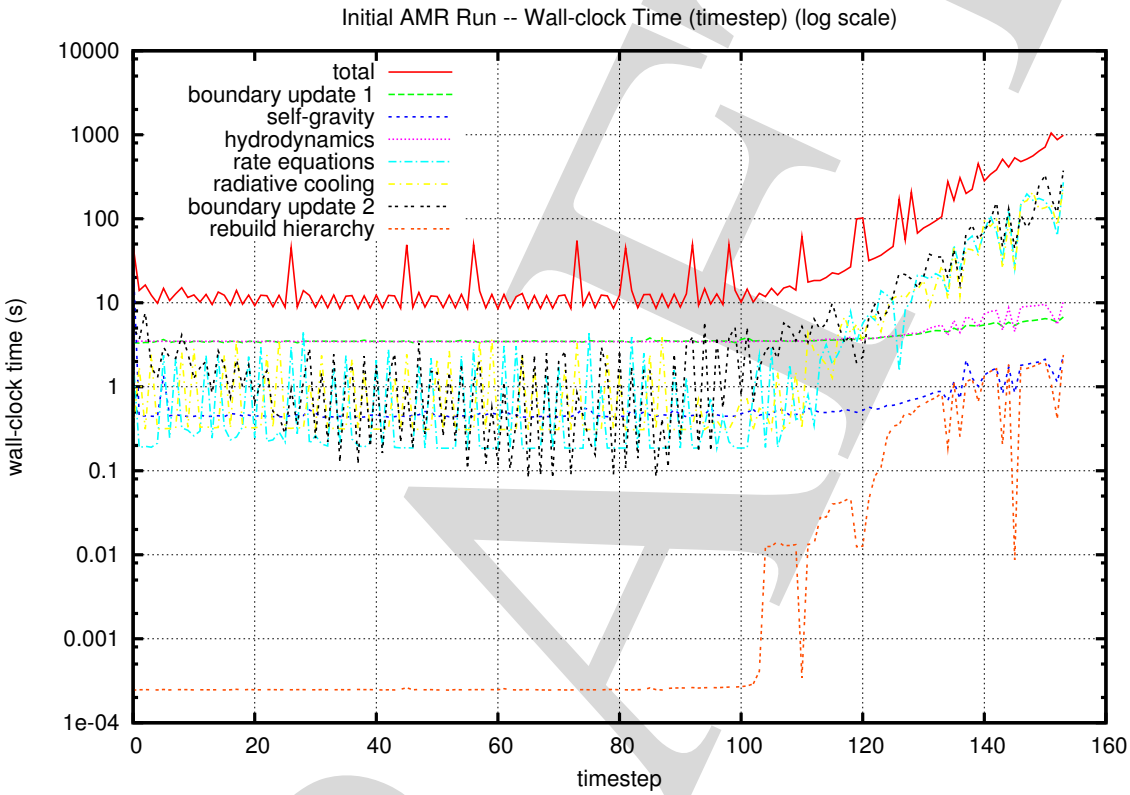

Figure 13. Same as Fig. 12, but log(wall-clock time per timestep). The jump in the cost of Rebuild Hierarchy at timestep 104 indicates the onset of significant mesh refinement.

DRAFT

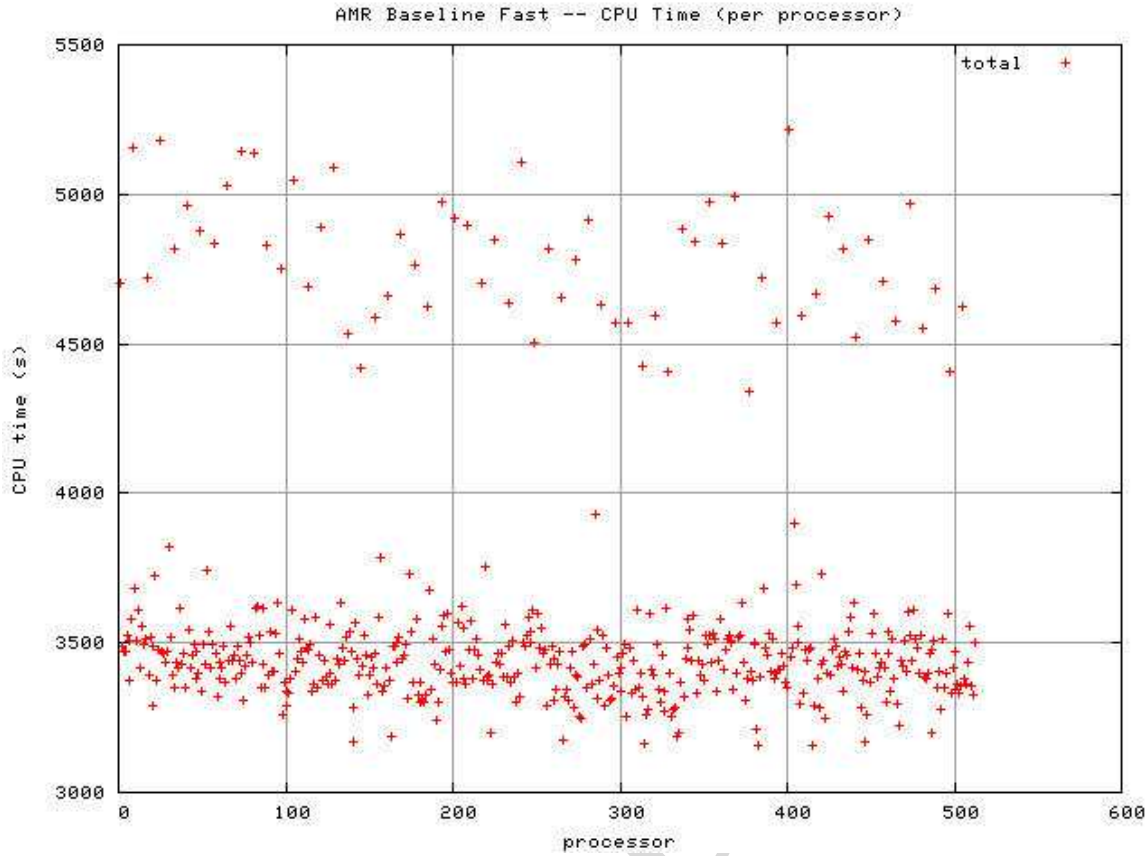

Figure 14. CPU time per processor for run AMR_FAST.

Figure 14 shows the total CPU time for run AMR_FAST as a function of processor. There is considerably more spread between the largest and smallest time compared with the unigrid run (5200/3200)=1.63 versus (58000/48000)=1.21, which may be due to load imbalances.

Summary of Communication Costs Finally, in Table 5, we summarize the communications costs for our baseline runs. We see that for the unigrid runs communications accounts for about 12% of the total cost of the run. The number is higher for the AMR baseline run. Communication becomes a larger and growing fraction of the total cost once mesh refinement is well advanced. The dominant contributions to this cost are boundary updates and self-gravity.

	UB_1	UB_2	AMR_FAST
Boundary update 1	40	553	64
Set time step	0	0	2
Self-gravity	607	1915	52
Boundary update 2	2557	3743	2839
update from finer grids	1	2	5
rebuild hierarchy	0	0	8
Total MPI time	3427	6898	3684
Percent of total time	13.8%	11.2%	28.5%

Table 5. Breakdown of communication costs for baseline runs.

Discussion and Opportunities for Future Optimizations Based on these results, we now discuss where optimization efforts might be most profitable. With regard to unigrid simulations, nearly 60% of the time is spent in three baryonic physics routines: `hydrodynamics`, `rate equations` and `radiative cooling`. Further analysis of where the time is spent in hydrodynamics may identify regions that can be sped up or replaced with faster approximations. We know, for example, a lot of time is spent in the iterative Riemann solver used by the PPM algorithm. A faster, approximate Riemann solver might be profitable. We might also save a substantial amount of time if we replaced our iterative non-equilibrium species solver routine with a direct solution of the equilibrium ionization equations–a valid approximation for the state of the intergalactic medium after reionization, and one that is already implemented in ENZO. We also subcycle the energy equation in code region `radiative cooling`. We might find some savings there through further analysis. Of the remaining 40%, we note that 19% is spent in the `boundary update` routines. It may be possible to eliminate one of these calls per timestep, since it would seem to be redundant in the case of unigrid runs. Another source of inefficiency is the $\sim 20\%$ load imbalance among processors (Fig. 6). The source of this imbalance is unclear, but might have to do with additional functions assigned to the master processor in each node. We request assistance from NERSC specialists on this machine-specific issue.

The playing field is more complex in the case of the AMR baseline run. In addition to the above mentioned optimizations, improved load balancing would speed things up significantly since there is a 60% disparity between fastest and slowest processor (Fig. 11). Some of this scatter surely has to do with additional work being done by the master processor on each node, but the majority is probably due to true load imbalances arising from dynamic mesh refinement. As noted above, we are puzzled by the growth of the cost of regions `rate equations` and `radiative cooling` relative to `hydrodynamics` as mesh refinement proceeds. We need to investigate whether the number of subcycles is growing and becoming too onerous; if so, switching to the equilibrium solver might be a substantial win. With 22% of the runtime,

`boundary update 2` should also be scrutinized for potential improvement. While these calls are absolutely essential to the correctness of the algorithm, we might find upon analysis of the memory access patterns that certain optimizations are possible, including buffering, pipelining, and cache blocking. Finally, we need to investigate what is responsible for the 30% of the runtime hidden within "other". If it is mostly due to synchronization barriers, then this would be revealed with message-passing graphing packages like Jumpshot. While we could do this kind of analysis ourselves, we would greatly appreciate getting assistance from someone at NERSC expert in this kind of analysis to save us time.

0.9 Q4 Information

Given the compressed timeframe of this project (6 months), we chose to focus on optimizing the AMR simulation (AMR_FAST) as it was both forward–looking in terms of where our science is going, and also it presented more opportunities for optimization. As noted above (cf. Table 4), three routines come to dominate the cost of a timestep at the end of our baseline run: `boundary update 2`, `rate equations`, and `radiative cooling`. As shown in Figure 13, the cost per timestep for these routines increases between 2 and 3 orders of magnitude while the cost for `hydrodynamics` increases only by a factor of three. Since the cost of hydrodynamics scales with the number of active zones, we see that the cost of `boundary update 2`, `rate equations`, and `radiative cooling` is increasing disproportionately with the number of zones.

The cause for the rapid increase in `rate equations` and `radiative cooling` is that both these routines are subcycling more and more as time goes on.

The problems and target machine remained unchanged throughout the fiscal year in this study. Please see section[0.8] for details. The algorithm was enhanced through a physical reduction in complexity. A description now follows.

0.9.1 Algorithm

Referring to the pseudocode0.11 , we see that we iterate these two routines `itrate` and `itcool` times per grid per level, up to a maximum of 10,000 times depending upon the ratio of the hydrodynamical time to the chemical timescale $\Delta t_{hydro}/\Delta t_{chem}$. This ratio can become very large in the dense gas produced by gravitational collapse where recombination times are short. Fig. 15 shows the distribution of zones and zone-iterations versus processor for last timestep of AMR_FAST. The red symbols are the zones. We can see that the zone count is reasonably well load balanced. The zone-iterations for the rate equations and radiative cooling steps are shown by blue and green symbols, respectively. We see a load imbalance has developed, particularly in

rate equations. Averaging over all grids and levels, the effective number of iterations for the last three root grid timesteps of AMR_FAST are given in Table 6. We see the iteration count for solving the rate equations is climbing rapidly, as suspected.

Timestep	$\langle itrate \rangle$	$\langle itcool \rangle$
151	7.99	1.22
152	9.66	1.23
153	10.77	1.26

Table 6. Effective number of iterations for the last 3 timesteps of AMR_FAST.

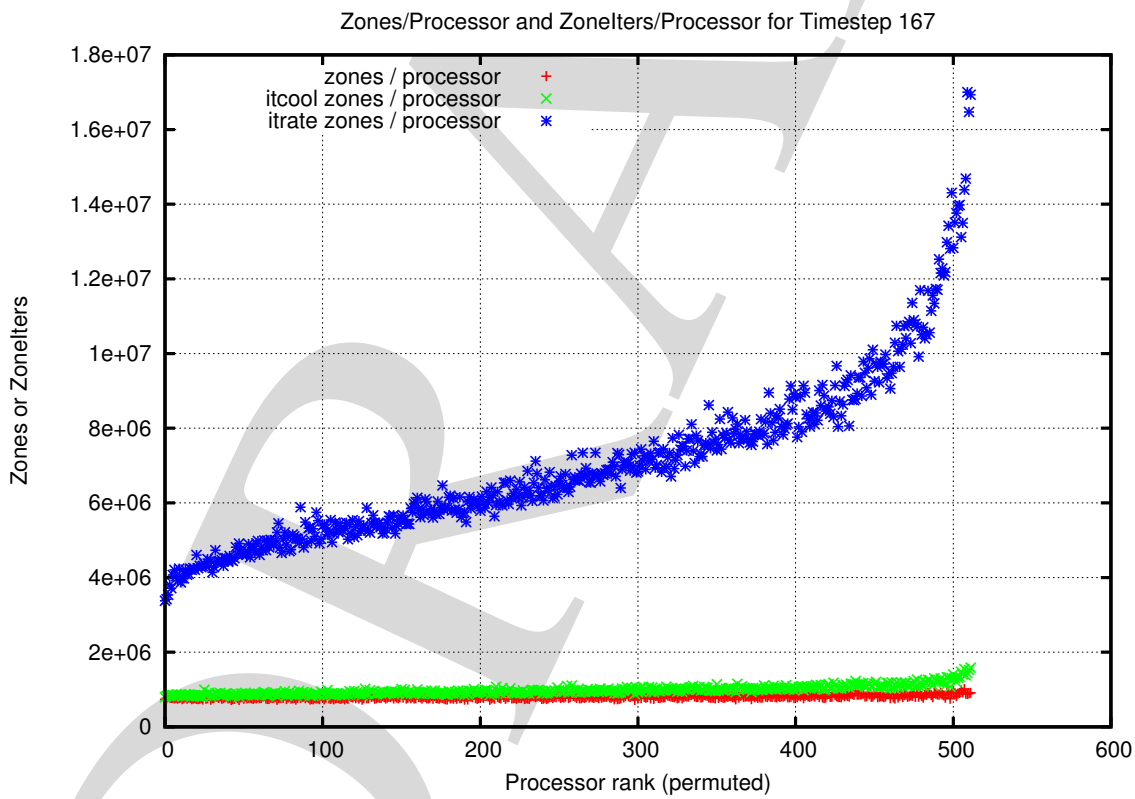

Figure 15. Distribution of zone-iterations versus processor for last timestep of AMR_FAST.

DRAFT

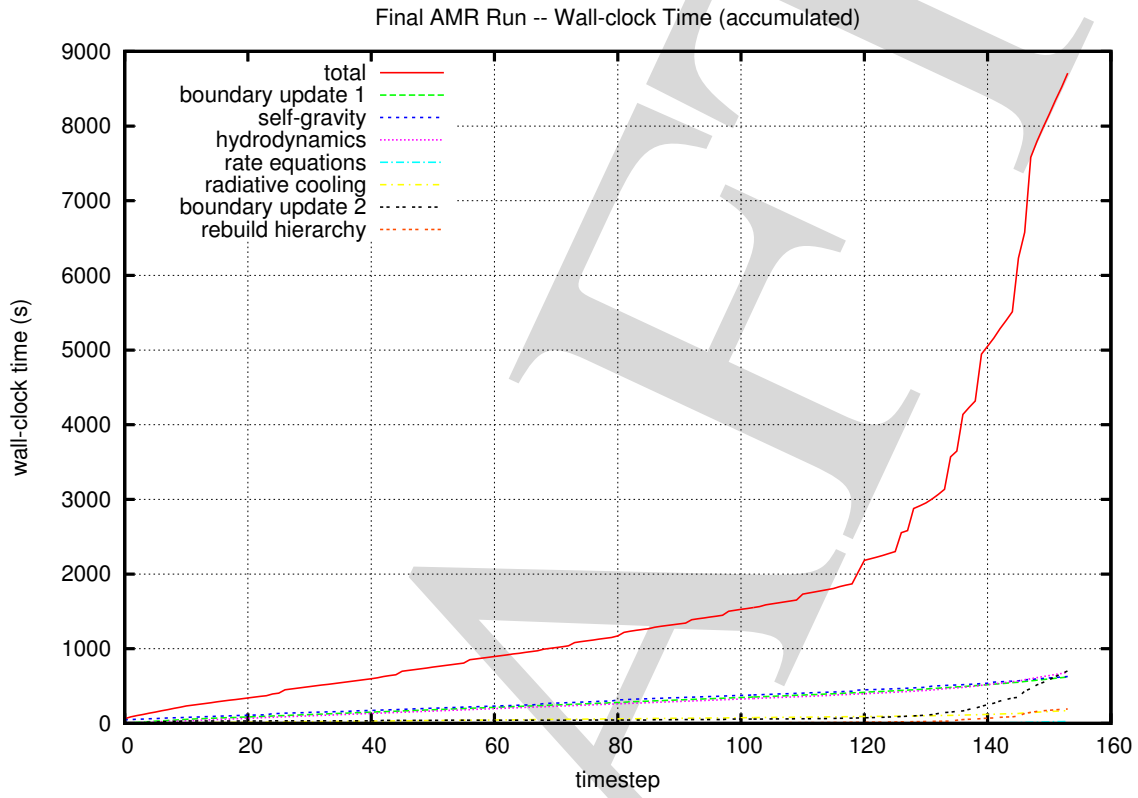

Figure 16. Accumulated wall-clock time for run AMR_FINAL, broken down by code region. This is to be compared with Fig. 11.

DRAFT

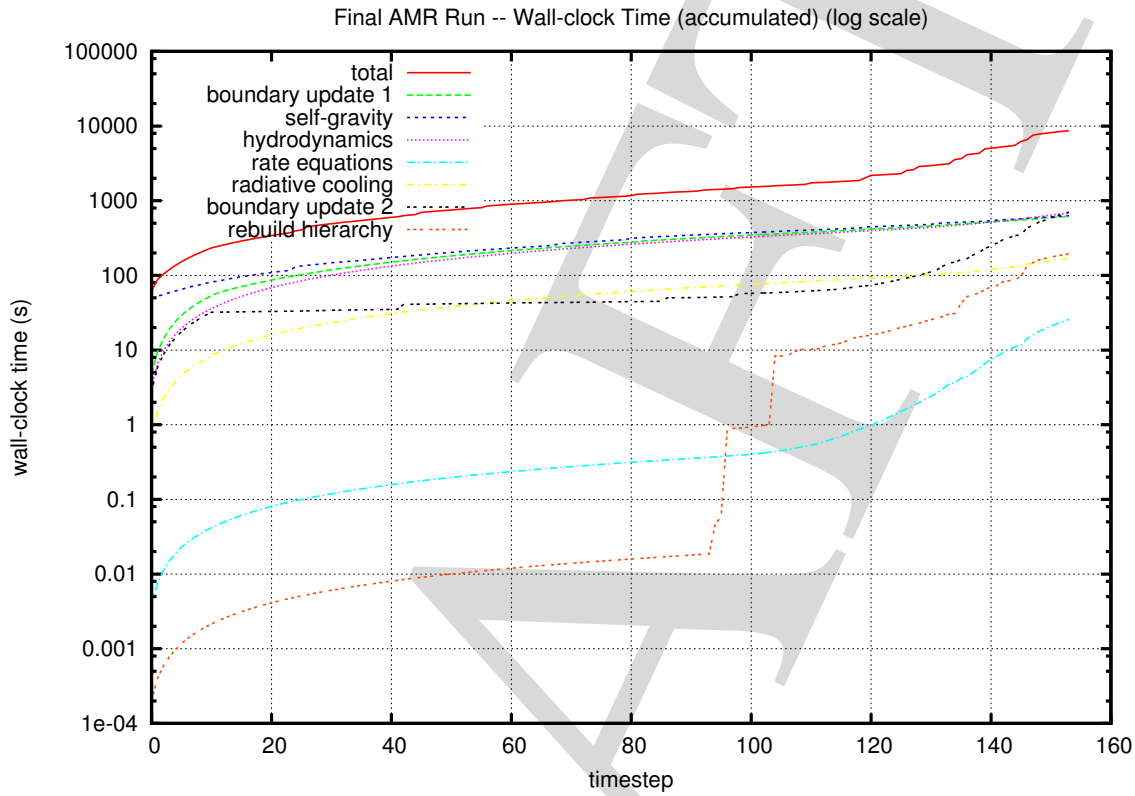

Figure 17. Same as Fig. 16, but log scaled. This is to be compared with Fig. 12.

Obviously, one could attempt to load balance the `rate equations` step. Even better, however, is to eliminate the iterations altogether by assuming the species concentrations are in ionization equilibrium. While this is a poor approximation in the low density gas, it is a very good approximation in high density regions–the very regions that are giving us a large iteration count. The reason is that the recombination time is very short in high density gas, and thus ionization equilibrium is easily established.

With this motivation, we have rerun the AMR_FAST baseline calculation substituting an equilibrium ionization and radiative cooling algorithm (hereafter GADGET cooling) for our nonequilibrium solver (hereafter NONEQ cooling). The GADGET cooling algorithm solves for the equilibrium abundance of our primary chemical species and then evaluates the cooling rates using them. The algorithm is described in detail in [60]. The structure[0.12] shows the pseudocode for GADGET cooling. Far fewer iterations are required to solve the system of six nonlinear equations to find the equilibrium concentrations than to step the nonequilibrium ionization equations along in the high density gas. We refer to this new calculation as AMR_FINAL.

DRAFT

0.9.2 Data

Q4 Collected Data	Problem(0.8.1)	Problem(0.8.1)
source codes	yes	yes
build information	yes	yes
runtime information	yes	yes
total walltime	no	yes

Table 12: Information collected during Q4 for the computed Enzo problems. Please see sec[0.2.4] for description. Only the algorithm changed during the fiscal year. Further, only one of the benchmark problems was rerun with the enhanced code, *problem two*. The results of that enhanced run are reported.

Figs. 16 and 17 show the cumulative wall-clock time for AMR_FINAL broken down according to code region; i.e., identical to Figs. 11 and 12, respectively.

Region	**Time (s)**	**Percentage of total**
Boundary update 1	621	7.1
Self gravity	629	7.2
Hydrodynamics	685	7.9
Rate equations	26	0.3
Radiative cooling	171	2
Boundary update 2	702	8.1
Rebuild hierarchy	193	2.2
Other	5681	65.2
Total	8708	100

Table 7. Breakdown of time spent in various code regions for AMR_FINAL.

Table 7 shows the accumulated time at the end of the calculation, broken down according to code region, and their percent contribution to the total cost. The total run time is $T_{Q4} = 8708s$.

0.10 Results

The total run time for AMR_FINAL is $T_{Q4} = 8708s$, down from $T_{Q2} = 12926s$. The ratio T_{Q4}/T_{Q2}=0.674, greater than the desired result ≤ 0.5. At face value, we are not in compliance. However, a careful look at the components of the total reveal that things are somewhat better. For reasons unknown to us, the cost of the gravity calculation increased fivefold in AMR_FINAL relative to AMR_FAST.

This is not related to mesh refinement as can be seen by comparing the cost per timestep in Fig. 18 for AMR_FINAL with same in Fig. 13 for AMR_FAST. Before mesh refinement begins at timestep 102–during the unigrid phase of the calculation–we see that `self-gravity` is uniformly more costly in AMR_FINAL per timestep by about a factor of five. We have checked code, input parameters, compiler flags, and environment variables, and find them the same for both runs. We suspect but don't know for certain that something changed on Bassi to cause this increase. To check this, we reran AMR_FAST using the executable we used for AMR_FINAL, but engaging the NONEQ model. This run we will refer to as AMR_RERUN. To be clear, the only difference between AMR_RERUN and AMR_FINAL is the value of one parameter flag which switches between NONEQ cooling and GADGET cooling. The executable was the same in both cases. The results are shown graphically in Figures 19 and 20, and Table 8.

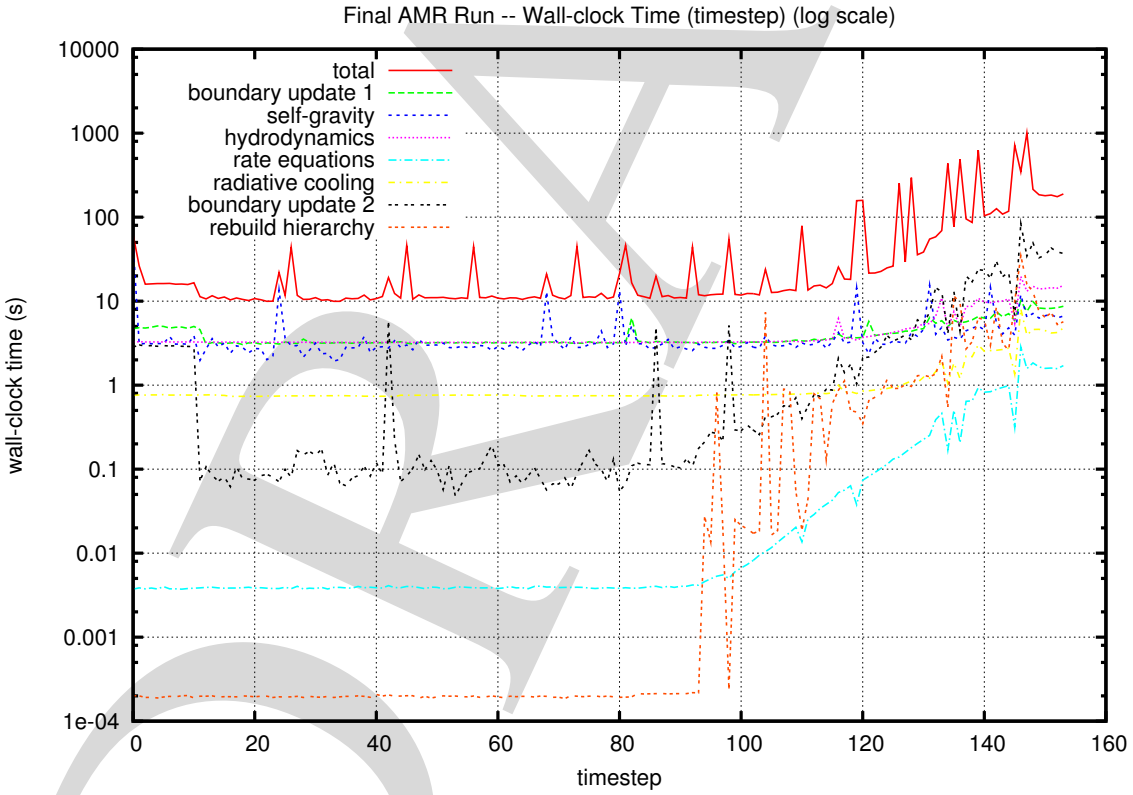

Figure 18. Cost per timestep for different code regions for run AMR_FINAL. This is to be compared with Fig. 13.

DRAFT

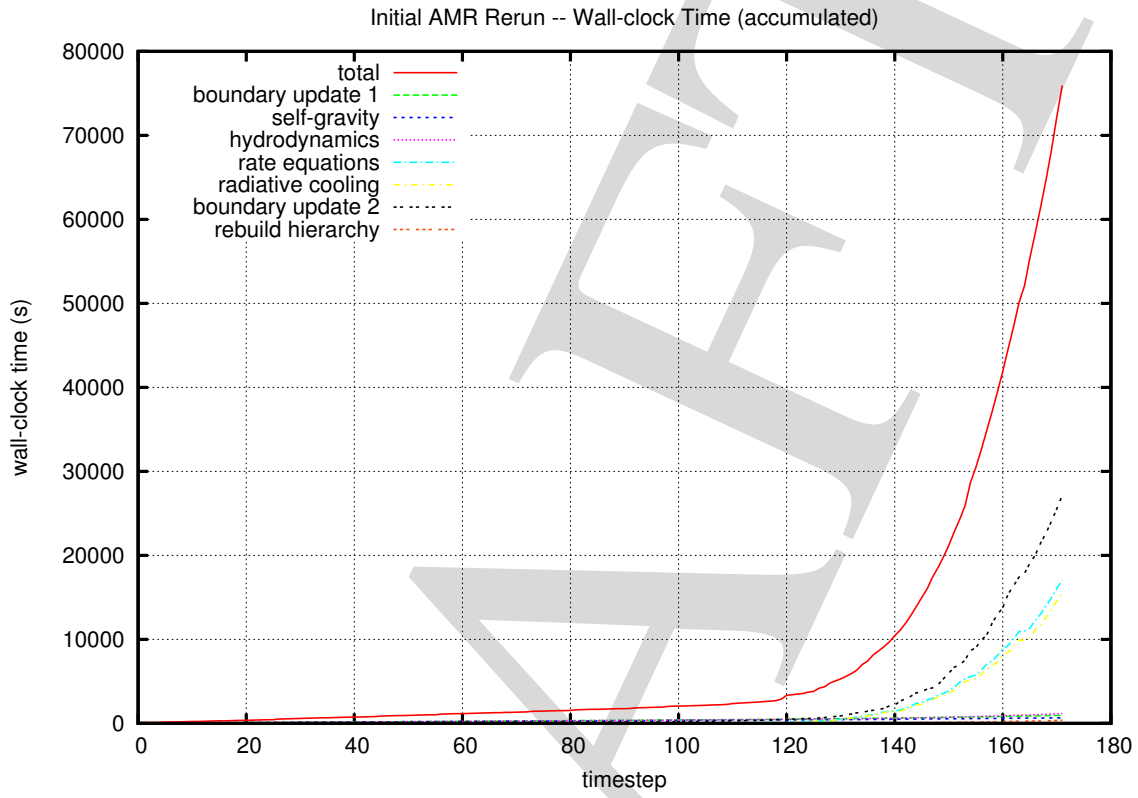

Figure 19. Accumulated wall-clock time for run AMR_RERUN, broken down by code region.

DRAFT

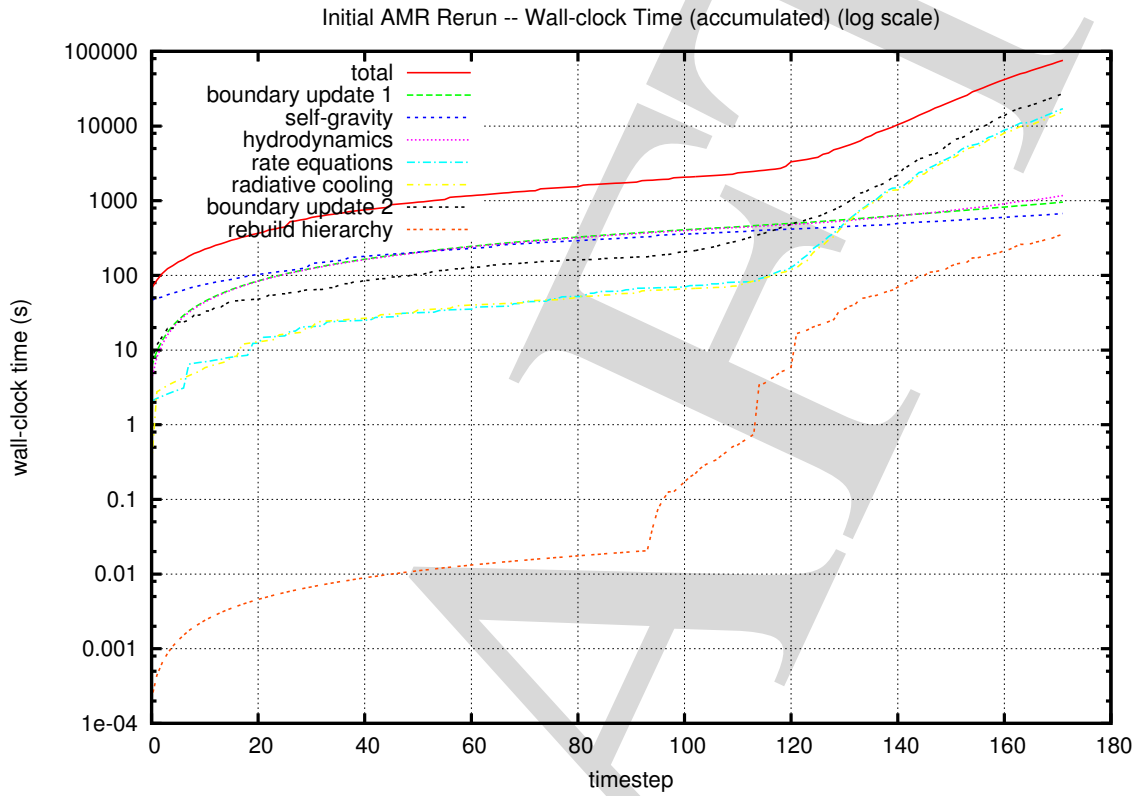

Figure 20. Same as Fig. 19, but log scaled.

Region	Time (s)	Percentage of total
Boundary update 1	749	2.9
Self gravity	561	2.2
Hydrodynamics	787	3
Rate equations	5309	20.5
Radiative cooling	4946	19.1
Boundary update 2	7518	29
Rebuild hierarchy	151	0.6
Other	5857	22.7
Total	25879	100

Table 8. Breakdown of cumulative time spent in various code regions for calculation AMR_RERUN to timestep 153.

Now we see quite a different story. The ratio T_{GADGET}/T_{NONEQ}=8708/25879=0.34–almost a factor 3 speedup. The run using equilibrium cooling is three times as fast as the run using the nonequilibrium model. This happy result is due to the fact that our rerun of the baseline calculation took roughly twice as long as the Q2 basline: 25879

s for AMR_RERUN versus 12926 s for AMR_FAST. Table 9 compares these two runs side by side. We see that virtually every part of the calculation is taking longer in AMR_RERUN than in AMR_FAST. Many code regions are taking more than twice as long, not just `self-gravity`. We suspect something fundamental has changed on Bassi between Q2 and Q4, perhaps the compiler. It is also possible something has changed in ENZO, however, the factor of three speedup is real since we used the same executable. Technically, we failed to meet the 2× speedup as strictly defined, but we have achieved a 3× speedup if we compare the two algorithms at the same instant in time. We note that the speedup will only increase the longer we integrate. The baseline was terminated at timestep 153 for budgetary not scientific reasons. A real application would integrate for a 1000 timesteps or more. We expect that the equilibrium model would be at least 10 times faster than the nonequilibrium model for a full science run. The incorporation of the equilibrium model into ENZO will enable new, previously unfeasible calculations to be carried out. From this standpoint, the OSCAR Joule Metric exercise has been useful for advancing our science capability.

Region	**AMR_RERUN**	**AMR_FAST**
Boundary update 1	749	589
Self gravity	561	128
Hydrodynamics	787	620
Rate equations	5309	2282
Radiative cooling	4946	2155
Boundary update 2	7518	2853
Rebuild hierarchy	151	30
Other	5857	4269
Total	25879	12926

Table 9. Cumulative times for AMR_RERUN and AMR_FAST through timestep 153.

Performance enhancement A comparison of the total weighted runtimes asserts the Joule software effectiveness metric for this application. The results from running the original algorithm may be referenced in Figure[19] and Table[8]. The enhanced results come from running AMR_FINAL and may be referenced in Figure[16] and Table[7].

Problem	**System**	**Enhancement**
High redshift galaxies, amr (sec0.8.1)	IBM Power5, *bassi.nersc.gov*	**performance**, ∼ 66.35%

Table 13: FY06 Joule performance enhancement for the Enzo code. A metric of primary interest is *wall time/processor/physical timestep*.

DRAFT

DRAFT

0.11 Algorithm for nonequilibrium ionization and radiative cooling (NONEQ)

```
main()

  EvolveHierarchy()

  for level in hierarchy(W-cycle)

    EvolveLevel(level)
      ⋮
    for grid in level
        ⋮
      SolveRateEquations(grid)

        solve_rate(grid)

        for k along grid z-axis
            for j along grid y-axis
                for iter = 1 to itrate(grid,j,k)
                    for i along grid x-axis
                        solve rate equations for grid zone (i,j,k)
                    end for i
                end for iter
            end for j
        end for k

      SolveRadiativeCooling(grid)

        multi_rate(grid)

        for k along grid z-axis
            for j along grid y-axis
                for iter = 1 to itcool(grid,j,k)
                    cool1d_multi(grid,j,k)
                        for i along grid x-axis
                            solve radiative cooling equations for grid zone (i,j,k)
                        end for i
                end for iter
            end for j
        end for k

        ⋮
    end for grid in level
      ⋮
  end for level in hierarchy
```


DRAFT

0.12 Algorithm for equilibrium ionization and radiative cooling (GADGET)

```
main()
  EvolveHierarchy()

  for level in hierarchy(W-cycle)

    EvolveLevel(level)
        ⋮
    for grid in level
          ⋮
      SolveRadiativeCooling(grid)

        GadgetCalculateCooling(grid)

        for k along grid z-axis
            for j along grid y-axis
                for i along grid x-axis
                    GadgetEquilibriumCooling(grid,i,j,k)
                        for iter = 1 to itgadget(grid,i,j,k)
                            solve equilibrium cooling equations for grid zone (i,j,k)
                        end for iter
                end for i
            end for j
        end for k

          ⋮
    end for grid in level
        ⋮
  end for level in hierarchy
```

Figure 17. Algorithm for equilibrium ionization and radiative cooling (GADGET).

DRAFT

MADNESS

0.13 Q2 Information

MADNESS is a general framework for scientific simulation that provides new capabilities for rapid computation with guaranteed precision enabled by new mathematical tools for multiresolution analysis in multiwavelet bases and efficient computation in many-dimensions through the use of separated representations. It is under very active development as part of the DOE, OBES SciDAC project Advanced Methods in Electronic Structure (R.J. Harrison, P.I.) and the Scientific Application Partnership (SAP) Advanced Mathematics for Electronic Structure (G.I. Fann, PI), in close collaboration with Prof. G. Beylkin, U. Colorado. After successful prototyping in both Python and C++, a new production version of MADNESS is now being prepared for emerging petascale architectures anticipated to have 100,000+ processors.

While MADNESS is a general purpose framework with emerging applications in fusion, climate science, nuclear structure, accelerator dynamics, and image processing being explored, it is still primarily motivated by application to chemical problems, and to this end has also been incorporated into the premier DOE parallel chemistry code, NWChem. The protoype code successfully demonstrated for a wide range of chemical computations (ground and excited state energies, Hartree-Fock and density functional theory, analytic derivatives w.r.t. atomic positions, and optical rotation dispersion) the controlled elimination of basis set error while maintaining fast computation.

The major project software goals for this year are to complete the production version of MADNESS and to tune it for efficient and reliable exploitation of the nations largest supercomputers. Using this framework we shall commence construction of a very wide range of tools for chemistry and material science. We anticipate the following outcomes:

- a *two-fold speed-up* of the current most expensive step in the new production code -the projection of the nuclear potential into the wavelet basis.
- a *ten-fold increase in the size of systems* that may be studied with density func-

DRAFT

tional theory compared to the prototype Python code. Such an increase will constitute an entirely new scientific capability by enabling reliable simulation of huge, all-electron molecules free of basis set error. This will require application to a system with at least 1000 atoms and 6,000 electrons. To make this benchmark well defined, we propose the study of both a fragment of DNA and a section of a carbon nano-tube.

Problem: Copper Lattice The test input used to assess the current state of MADNESS for the Q2 baseline describes the positions of copper atoms in a lattice and computes the times to project the nuclear potential into the wavelet basis (labeled construction), to transform to and from the wavelet basis (compression and reconstruction, respectively) and to add the potential to a copy of itself (nearly identical in cost to that of compression).

0.13.1 Algorithm

The new computational tools promise fast ($\mathcal{O}(N)$ or $\mathcal{O}(NlogN)$) computation with guaranteed precision in three, six, and higher dimensions. The combination of multiresolution and separation representations enable fast application of many physically important integral operators which enables direct solution of integral equations rather than poorly conditioned differential equations that involve operators of high-norm with associated loss of precision.

DRAFT

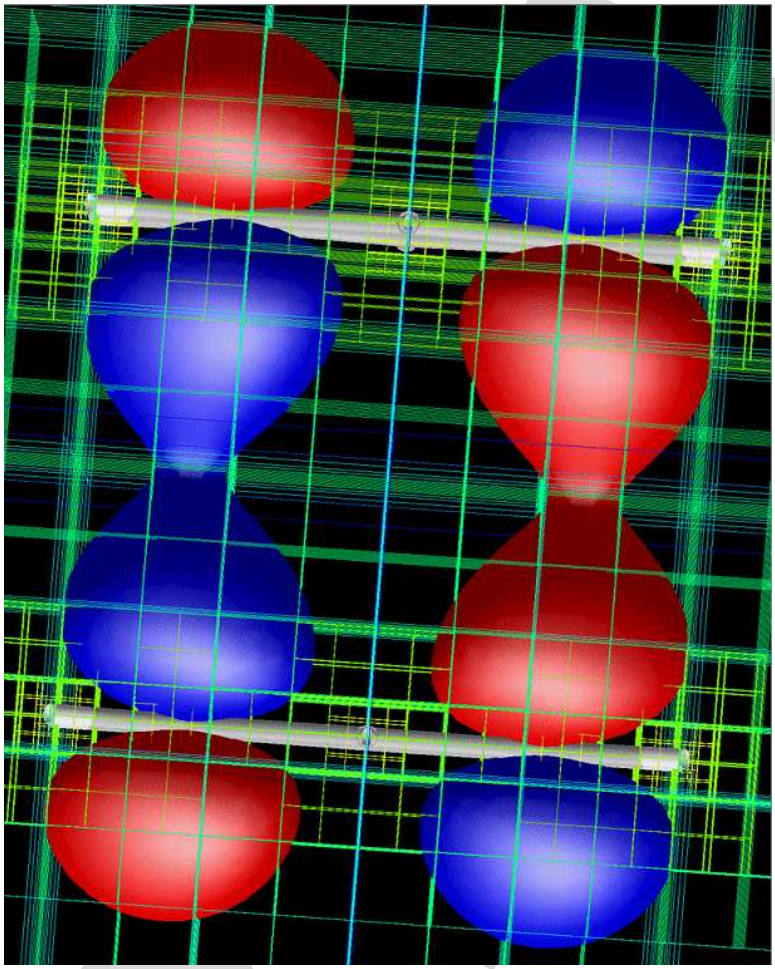

Figure 21. A molecular orbital of $(C_6H_6)_2$ demonstrating the automatic adaptive refinement.

In addition to low algorithmic operation count and efficient sequential and parallel execution, scientific applications enjoy a very high level of composition in Python and C++ that is ultimately intended to be akin to that of Maple or MATLAB.

The details behind the Q2 algorithms that were benchmarked and the subsequent enhancements in Q4 are described in section[0.14.1].

0.13.2 Machine

The Cray XT3 (http://info.nccs.gov/resources/jaguar) at the National Center for Computational Sciences (NCCS) is being utilized for the MADNESS study.

0.13.3 Data

Q2 Collected Data	Problem One(0.13)
Source codes	yes
Build information	yes
Runtime information	yes
Total walltime	yes

Table 14: Information collected during Q2 for the computed MADNESS problem. Please see sec[0.2.4] for description.

The capabilities of the production code are not yet complete and essential operations for scientific applications are still being developed (notably the differentiation and integral convolution operators). Thus, the initial performance benchmarking is of existing capabilities, whereas the final goals include both performance enhancements and demonstration of new scientific capabilities.

MADNESS Copper Lattice Exercises					
Oper::PEs	**256**	**512**	**1024**	**2048**	**4096**
construct f	8210.01	4486.95	2586.41	1562.33	982.30
compress f	2.95	1.61	.93	.56	.35
reconstruct f	2.94	1.59	.92	.56	.35
construct g	8209.58	4486.70	2586.47	1562.48	982.66
add f + g	6.30	3.43	1.97	1.20	.75
Total Time [s]	16432.20	8980.5	5176.84	3127.21	1966.47

Table 15: The Q2 benchmark information. The time for each task in seconds is reported for a scaling study over the number of processing elements (PEs).

Excellent parallel scaling is observed for all steps from 256 to 4096 processors, but the only 52% efficiency is disappointing. The numerical algorithms are designed to deliver much higher efficiency and we currently ascribe the poor overall performance to poor load balance but much more analysis is necessary. The construction step is infrequently performed but presently very expensive. Fast summation of the nuclear potential might resolve this issue, but mathematical analysis has yet to be performed. The reconstruction and compression steps are much faster, but are repeated often.

0.14 Q4 Information

The problems and target machine remained unchanged throughout the fiscal year in this study. Please see section[0.5] for details. The algorithm was modified. A description now follows of the changes.

The goals of the MADNESS project broadened significantly over the last nine months to include a new, portable, and general purpose parallel programming model in addition to the original goals of deploying multi-resolution methods on massively parallel computers and applying them to electronic structure methods. This was in response to three stimuli:

1. Involvement with the DARPA High-Productivity Computing Systems (HPCS) language evaluation project which emphasizes enhancements to the programming environment.
2. Involvement in an NSF computer science project and researchers at Ohio State University who are using MADNESS to motivate their research into new parallel programming paradigms.
3. The needs of MADNESS itself and its anticipated end-users.

As a result of this change in scope, the full electronic structure functionality is not yet complete, though it is shown that a direct consequence of the new development is that new functionality is now far, far easier to implement both correctly and efficiently. A relevant familiar adage is *measure twice, cut once*.

We start by describing the technical challenges in the original distributed-memory message-passing version (used for the Q2 baseline) by comparing with the simplicity of a shared-memory parallel approach, and then we examine the new parallel programming model. The improved productivity is demonstrated, and finally, performance data is presented for the new version of MADNESS on the ORNL Cray-XT3.

0.14.1 Algorithm

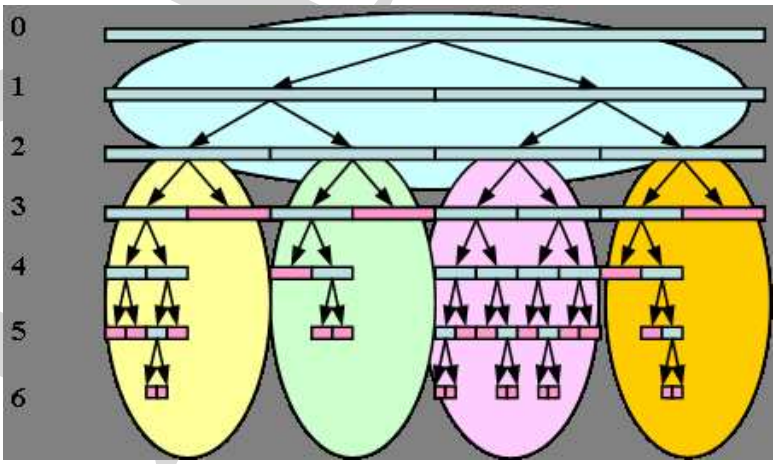

Figure 22. Recursive tree structure maintained by MADNESS

MADNESS implements its adaptive refinement using a tree; in three dimensions this is an oct-tree since $2^3 = 8$. In Figure[22] is shown a representative binary tree (e.g., for a function in one dimension). Recursive algorithms are very powerful at performing operations on trees.

```
Function walk_down(node)
    Operation(node)
    For each child of node:
        walk_down(child)
```

Figure 23. Pseudo-code for a function to walk down a tree to apply an operation on every node in the tree.

The function *walk_down()*[23] calls itself, which is the characteristic of recursive algorithms. It should also be apparent that the operation is being applied while walking down the tree - i.e., an operation is first applied to a node, and then to its children, and so on. Applying an operation while walking up the tree[24], i.e., first to a node's children and then to itself, is just as easy.

```
Function walk_up(node)
    For each child of node:
        walk_up(child)
    Operation(node)
```

Figure 24. Pseudo-code for a function to walk up a tree to apply an operation on every node in the tree.

If the work on each child is completely independent, then we can spawn activities on different processors to do this work. The shared-memory programming language Cilk provides the keyword spawn to describe just this, e.g., in [25] is a parallel version of *walk_up()* highlighting where the additional keyword was introduced.

```
Function walk_up(node)
    For each child of node:
        spawn walk_up(child)
    Operation(node)
```

Figure 25. Parallel *walk_up()*.

Our shared-memory parallel version of MADNESS was almost literally this easy, ignoring the details of all the math and data structures behind this pseudo-code. But on a distributed memory computer, things get a lot more complicated. First, a child, parent or sibling node may no longer be on the same processor, and so the data structure must keep track of what lives where. When a node changes its own state (commonly as a result of adaptive local refinement) it must notify its relatives about this so that they share a consistent view of the world. Herein lays a more major problem - how does the relative even know to look for a message when it does not know that the remote state changed? Most other adaptive mesh codes only change such state at regular, pre-scheduled times and so such notifications can be handled all at one time and the overhead of the implied global synchronization is acceptable. However, a crucial strength of MADNESS is very fluid changing of the adaptive refinement and we cannot afford the overhead of global synchronization.

The first approach was to send a message whenever the state could have changed. This results in potentially unnecessary message traffic, but worse, it also requires traversing parts of the tree that might otherwise be dormant, e.g., in order to let a remote great-great-grandchild know that nothing of interest to it has changed. It is this additional logic that blows the complexity of the code way up, and it technically arises from using a single-program multiple-data model to maintain a consistent distributed data structure.

Consider [26] and [27] -the actual message passing code is about 5-10x longer still due to mundane details that have been elided for clarity.

```
Function refine(node)
    For each child of node
        get_data(child)
    If (need to refine)
        For each child of node
        spawn refine(child)
```

Figure 26. Parallel Cilk code to adaptively refine a tree.

DRAFT

```
Function refine(node)
    If (node is remote parent)
        For each child of node
            _Receive message with flag
            _IF (flag is true)
            __refine(child)
    Elseif (node is active)
        For each child of node
            _get_data(child)
        flag = if we need to refine
        For each remote child of node
            _Send message with flag
        For each local child of node
            _refine(child)
    Elseif (node is inactive)
        For each remote child of node
            _Send message with flag=false
        For each local child of node
            _refine(child)
```

Figure 27. The initial message passing version of [26].

As messy as this appears, one need to write and debug it once. *Refinement* is actually one of the easier tasks to do since it stays within the tree structure, works on only one function, and the computation occurs where the data resides. *Multiplication* of two functions is a harder example. The two functions may have very different adaptively refined meshes and there are mathematical rules that prescribe how to bring one function down to the same level as a more deeply refined function so that multiplication can occur. This can require asking a remote parent for data which may in turn have to ask its parent, and so on. *Differentiation* is harder still since instead of moving data up/down the tree, it requires values from neighboring boxes at the same level in the tree (i.e., siblings). These may again reside on different processors and even worse, due to the adaptive refinement, they may not even exist which requires additional logic and communication to figure out how to proceed. *Integral operators and higher-order differential operators* have to search even further out at the same level. The communication between nodes also causes excessive synchronization which inhibits scalability. There are many other operations that we need to implement, and it would be very valuable for less experienced programmers to be able to extend MADNESS. Finally, if we are to cope with problems large enough to require petascale computers, we only have enough space to store local connectivity information about the tree - global knowledge is by definition not scalable.

There are three components to the MADNESS solution. The first was to develop a

portable, distributed memory version of Cilk (in standard C++). Essentially, each process has a queue of tasks and any process can put tasks into the queue of any other process. A task can be either ready to run, or waiting for some other task to complete first. These dependencies are expressed naturally using the same mechanism as Cilk (futures; see below). The approach is portable since it presently uses MPI+polling, but when more powerful mechanisms become available on our largest computers (e.g., active messages) it can switch to those. This capability means that we can move computation around the machine just as easily as we can move data. This sets the stage for eventual implementation of dynamic load balancing through work stealing.

The second technique was to factor most of the local/remote logic out of the code by using futures. A *future* is the possibly yet-to-be-evaluated result of a computation that was spawned for parallel execution. Trying to read the value of a future waits if necessary for the computation to complete, and then returns the value. A future can also be asked (probed) if it is ready - if not, you can do some other task instead of idly waiting. In the MADNESS distributed memory version, futures can be either local or remote and indeed can change from one to the other as a byproduct of the calculation. The important thing is that the consumer of the future does not know or care where the data is coming from, merely that it will be there when requested.

Finally, the complexity of walking up and down the tree, recurring coefficients down as necessary to adjust adaptive refinement in one or more functions can be eliminated by borrowing a concept from the standard C++ library - *iterators*. MADNESS now provides a standard set of iterators that relieve the programmer from these complex and error prone tasks. In [28] is the pseudo code that a user would write for the multiplication of two adaptively refined functions (f and g) using the new framework.

```
class TaskMult :  public TaskLeaf<TaskMult>
private:
    Future a, b;
public:
    TaskMult(Tree node,Function f,Function g,Function result)
        a = f.get(node)
        b = g.get(node)
    bool probe()
        return a.probe() and b.probe()
    bool predicate()
        return a.exists() and b.exists()
    void apply()
        multiply(node, a.get(), b.get())
```

Figure 28. MADNESS multiplication of two adaptively refined functions.

Again, some mundane details have been elided for clarity, but the actual code is not

much longer and critically contains no if tests, no details about where data resides, and no code to handle the recursion of coefficients down the tree and/or between processors. The constructor of the class assigns to two futures the result of possibly remote calculations to get/make the necessary coefficients in each function. The *probe()* method is called by the task scheduler to inquire if the task is ready to execute, which is the case if its futures are ready. The *predicate()* method is called when the task is actually running to determine if refinement is necessary. The *apply()* method is invoked to actually perform the sequential task of multiplying two blocks of coefficients. All of the details of recursion and tree traversal are contained in the base class $TaskLeaf$ which is the MADNESS iterator that applies operators to coefficients at the leaves of the tree. In turn, much of the logic to do with where data resides has been moved out of both $TaskLeaf$ and $TaskDiff$ into the $Future$ class. The $Future$ class is used extensively throughout the code and $TaskLeaf$ is reused multiple times for operators such as differentiation, refinement, multiplication, squaring, and so on.

The iterator proceeds by first placing all initially known tasks in the task queue (local and remote). If several independent operations on functions are to be performed all of these tasks can be scheduled. Then, the tasks scheduler is started and computation begins. As execution proceeds additional local or remote tasks and futures can be generated. E.g., to refine a sub-tree resident on a neighboring processor and to get the generated coefficients, a future is associated with a remotely scheduled task that will eventually perform the refinement. To avoid delaying other workers, remotely generated tasks are assigned higher priority than locally generated tasks.

In addition to exposing nearly all available parallelism, this approach has numerous other advantages. It is very suitable for multi-core processors. Many of the communication buffers can be pre-allocated which is essential to eliminate bottlenecks and to realize maximum transmission speed. Much of the communication, hardware permitting, can be overlapped with useful computation. The more expensive tasks can be migrated to idle processors which enables system wide load balancing. Many enhancements are possible including more flexible prioritization of tasks, richer mechanisms for expressing and managing dependencies, construction of near-optimal local/regional/global task schedules, tree partitioning for static load balancing, and work stealing/diffusion for dynamic load balancing. A large literature and software base exists on many of these topics and we are benefiting greatly from our collaboration with computer scientists at ORNL, OSU and elsewhere.

In adopting a heavily object-oriented approach to programming, one is always concerned with the "abstraction overhead." By hiding details and complexity from the human in order to make it easier to program, we may also be hiding information that the compiler needs for optimization. Object-oriented methods are also notorious for introducing lots of additional procedure calls that slow execution and further hinder optimization. Fortunately, modern C++ and recent compiler technology provide powerful solutions in the form of templates and inlining. This is not the place to discuss these topics, but with care they can be employed to eliminate the performance penalty of abstraction while still realizing the benefits. Profiling MADNESS reveals

that its execution time is strongly dominated by useful work and the abstraction penalty is at most a few percent, and much less in high accuracy calculations.

0.14.2 Data

Q4 Collected Data	Problem(0.13)
source codes	yes
build information	yes
runtime information	yes
total walltime	yes

Table 16: Information collected during Q4 for the computed MADNESS problem set. Please see sec[0.2.4] for description.

Processors	T_project+refine	T_compress	T_reconstruct
512	1020	0.077	0.178
1024	585	0.044	0.111
2048	309	0.022	0.061
4096	159	0.011	0.032

Table 17: Timing results for $k = 9$, $tolerance = 1.e-3$ for the Q4 implementation of the MADNESS operations for : project and refine, compress, reconstruct. The runs utilized the the enhanced algorithm(0.14.1).

Processors	T_multiply	T_square	T_differentiate
512	0.092	0.057	0.26
1024	0.052	0.032	0.147
2048	0.027	0.015	0.096
4096	0.0142	0.0064	0.099

Table 18: Timing results for $k = 9$, $tolerance = 1.e-3$ for the Q4 implementation of the MADNESS operations for : multiply, square, differentiate. The runs utilized the the enhanced algorithm(0.14.1).

Presented in Tables[21,22] are the results showing the computation rate in GFLOP/s, the execution time of each step, and the speedup assuming the execution rate on 64 processors is ideal. There is not sufficient data yet to explain the poor scaling of differentiation which a simple performance model suggests should scale well for the target function.

Finally, a note with regard of the use of GFLOP/s to compare the computation rates. The number of floating point operations is independent of the number of processors and the time taken to perform these dominates the calculations. Thus, it is a useful

Processors	T_project+refine	T_compress	T_reconstruct
512	1953	0.161	0.315
1024	1086	0.091	0.193
2048	570	0.047	0.101

Table 19: Timing results for $k = 7$, $tolerance = 1.e-5$ for the Q4 implementation of the MADNESS operations for : project and refine, compress, reconstruct. The runs utilized the the enhanced algorithm(0.14.1).

Processors	T_multiply	T_square	T_differentiate
512	0.169	0.096	0.406
1024	0.089	0.052	0.224
2048	0.045	0.0255	0.133

Table 20: Timing results for $k = 7$, $tolerance = 1.e-5$ for the Q4 implementation of the MADNESS operations for : multiply, square, differentiate. The runs utilized the the enhanced algorithm(0.14.1).

Ops::Procs	512	1024	2048	4096
project+refine	223	389	735	1432
compress	451	771	1424	2645
reconstruct	202	318	564	1023
multiply	503	3757	1530	2731
square	523	905	1776	3341
differentiate	472	810	1168	1099

Table 21: Floating point performance (GFlops) for $k = 9$, $tolerance = 1.e-3$ for the Q4 implementation of the MADNESS operations : project+refine,compress, reconstruct, multiply, square, differentiate. The runs utilized the the enhanced algorithm(0.14.1).

Ops::Procs	512	1024	2048
project+refine	231	416	793
compress	348	620	1178
reconstruct	179	293	549
multiply	424	0.052	1516
square	500	895	1716
differentiate	301	531	863

Table 22: Floating point performance (GFlops) for $k = 7$, $tolerance = 1.e-5$ for the Q4 implementation of the MADNESS operations : project+ refine, compress, reconstruct, multiply, square, differentiate. The runs utilized the enhanced algorithm(0.14.1).

DRAFT

measure of how "fast" one component of the code is executing relative to another since they all employ similar sequential kernels. It is not valid to compare these rates to the unattainable peak speed of the machine for many reasons. Trivially, we can easily construct algorithms that hit a high fraction of this speed and yet have much longer execution times. We also do not yet know the theoretical peak floating point execution rate of these specific algorithms on this machine, and furthermore at least some useful work does not involve floating point operations. More relevantly, the memory, the processors and the communication fabric are all very expensive machine components and it is not meaningful to focus on just the floating-point units since they are certainly not the price limiting component and are probably neither the performance limiting component. Development of detailed performance models is a key future step which will inform machine design/purchase decisions by enabling a cost-benefit analysis of investments in key hardware or software systems.

0.15 Results

Performance enhancement The performance benchmark established in Q2 was projection with adaptive refinement to a precision of 10^{-3} into the $k = 9$ polynomial basis of the full nuclear potential arising from a cube of 4096 copper atoms arranged on a body-centered cubic lattice. The adaptive refinement starts at level 4 and continues until the accuracy criterion is satisfied which is at level 16. This corresponds to the finest resolved length scale being about 100,000x smaller than the simulation cell. The projected function is then compressed and reconstructed which correspond to the forward and reverse fast-wavelet transform which are very important operations and analogous to the fast-Fourier transform. The old and new timings in seconds are presented in the Tables[15,17].

Processors	project+refine	compress	reconstruct
512	4.398970588	20.88584416	8.952303371
1024	4.421213675	21.059	8.309918919
2048	5.056084142	25.61036364	9.150557377
4096	6.178	32.25090909	10.9781875

Table 23: Speedup comparison of the Q4 versus the Q2 MADNESS implementation of the operations : project and refine, compress, reconstruct.

Problem	System	Enhancement
project nuclear copper potential into wavelet basis(sec0.13)	Cray XT3, *jaguar.ccs.ornl.gov*	**performance**, see[23]

Table 24: A scalable speedup is asserted for the baseline problem **MADNESS** code. The projection with adaptive refinement to a precision of 10^{-3} into the $k = 9$ polynomial basis of the full nuclear potential arising from a cube of 4096 copper atoms arranged on a body-centered cubic lattice was computed.

Discussion The new times are uniformly faster by factors ranging from 4.4 to 32.3 on the same number of processors. It should first be recalled that the old hardware was based upon a single-core 2.4 GHz opteron processor, whereas the new hardware uses a dual-core 2.6 GHz opteron. Thus, a speed-up of 1.083 could be ascribed to processor speed enhancement, but in practice this would be offset by the two cores in the new system competing for the same the bandwidth to memory available to the original single core.

For the projection + refinement operation, about a factor of 2 can be ascribed to improved use of compiler optimizations and manual optimizations such as unrolling, etc., and another factor of two to improved load balance by adjusting the distribution of data. It is observed that the new code scales better to larger number of processors. The speedup in going from 512 to 4096 processors was 4.6 whereas in the new code it is 6.4. This is solely due to the enhanced programming model described above.

For compression, the speedup is much larger. This is because, in addition to the optimizations discussed above, the old code was computing to unnecessary precision and the new code (in the MADNESS spirit of fast algorithms with guaranteed precision) computes more closely to the requested precision whereas the old code kept unnecessary coefficients. Again, better parallel scaling is seen in the new code.

The reconstruction operation should be running as fast as compression, but it is over a factor of 2 slower. It is presently hypothesized that this is due to poor load balance, and in particular from giving too much work to processes that own nodes near the top of the tree which hinders fan out of parallel work down the tree. A new work distribution algorithm is under development to resolve this and related issues.

Science enhancement MADNESS performed calculations to a 100x greater precision, i.e., to a precision of 10-5. For the corresponding times see Table[19].

The additional two digits of precision increase the cost of the calculation by just a factor of about 2. This is a result of the high-order polynomial basis providing rapid convergence once sharp features have been resolved by adaptive refinement, and corresponds to just 25accurate is still substantially faster than the old code.

DRAFT

ScalaBLAST

0.16 Q2 Information

High performance sequence alignment for biological applications Homology detection and sequence alignment are the two most fundamental computational tasks applied to biological problems in the so-called post-genomic era because they make it possible to identify relationships between molecular machinery of organisms which have been well-characterized and those which have been newly sequenced. To make these connections, newly sequenced organisms must be compared to all known sequences. But the rate at which new sequences are completed continues to grow exponentially and, correspondingly, the volume of known sequences increases in kind resulting in an explosion of sequence data. Driven by this revolution of information entirely new techniques and fields of biology have emerged (informatics-driven biology), such as genome context mining and reconstructing the 'tree of life', for which comparisons between many whole organisms are required. The importance of these fields to DoE is underscored by programs such as the Joint Genome Institute (JGI), genomics:GTL, the Shewanella Federation, and the biological areas covered in programs such as SciDAC, through which it is hoped that research in biosystems can realize its potential to revolutionize environmental cleanup, energy production, and national security. But the scope of problem which can be addressed using current bioinformatics tools is in many cases the limiting step. Analyzing even a modest collection of sequences may take weeks or longer on a conventional workstation. The real problems of interest are thousands or millions of times this size and so are considered intractible. We have developed ScalaBLAST as a high-performance extension to BLAST optimized for very large sequence homology and alignment calculations on high-end supercomputers enabling the next generation of informatics-driven computational biology.

This report is a performance evaluation of baseline sequence alignment code MPI BLAST 1.4.0 for the beginning of FY06 and (will become) an assessment of improvement by the end of FY06 of a newly developed highly scalable sequence alignment code, ScalaBLAST, to be used as a part of the OMB software effectiveness metric. The biology target applications are both driven by science problems that will require very large (1 million) queries to be performed against the nonredundant protein (nr)

database as provided by NCBI as of February of 2006 for the baseline measurement and the nr database of May 2006 for the FY06 improvement metric. Each query task will involve homology detection and sequence alignment against identified homologs from the nr database and reporting of these results in the standard NCBI BLAST output format. Both baseline and improvement metrics will be taken from ScalaBLAST running on 320 (50MPI BLAST is the accepted state-of-the-art in high performance sequence alignment [[64]]. Developed and distributed by Los Alamos (http://mpiblast.lanl.gov/), MPI BLAST is an open-source extension to the NCBI toolkit. MPI BLAST allows a user to break up a sequence database into fragments for running in a parallel environment. ScalaBLAST is an extension to the NCBI BLAST executable and confers scalability through query scheduling and advanced memory management on a large query set. ScalaBLAST has previously been optimized for high-performance distributed memory architectures with very low latency (Quadrics ElanIV) interconnect and cache-coherent NUMA platforms (SGI Altix). We have observed an approximate 3-fold slowdown in performance when running over higher-latency interconnects such as Infiniband, Myrinet or Gigabit Ethernet because ScalaBLAST has not yet been optimized for these higher latency interconnects. Since these interconnects are more representative of what is used in university and NIH/DOE laboratory bioinformatics pipeline environments, the impact of achieving the goal of this OMB performance metric is substantial-success in this metric will make the scalability and efficiency of ScalaBLAST accessible to the worldwide community of biology investigators. For this OMB performance metric, we propose to address the high-throughput related issues with MPI BLAST primarily by tackling optimization issues regarding task partitioning and memory management to more effectively hide latency associated with I/O and memory access. We will not attempt to improve the performance of the internal BLAST algorithm itself. Modifications to the core BLAST algorithm are beyond the scope of this study and are the subject of ongoing research by many groups. The following is a more detailed discussion of the motivation for the science drivers, algorithms and implementation of the code base, input and run-time parameters, and the performance metric we will use to drive code enhancements for ScalaBLAST.

0.16.1 Problems

Background and Motivation Genetic code is based on DNA, which can be represented as a single string of characters (A,C,G &T). Genes are regions of DNA encoding for proteins, which can each be represented as a single string of characters as well. Homology is a measure of the similarity between two sequences and alignment is the physical mapping from one sequence to a different but homologous sequence. BLAST, one of the most widely used computational tools in biology, is a sequence alignment (and homology detection) algorithm distributed by the National Institute of Health (NIH) National Center for Biotechnology Information (NCBI). BLAST represents a quantum leap in efficiency over previous homology and global alignment algorithms such as Needleman-Wunsch [[65]]. These global alignment methods search over the

entire space of possible alignments for the 'global' optimal alignment. For two given sequences of length n and m respectively, these methods require nxm memory and perform on the order of nxm operations. Currently the collection of unique proteins contains over 3 million sequences. The most modest of the grand challenge problems in sequence analysis would involve performing sequence alignments of each of these entries against every other entry. Using the global alignment methods, one would expect to need at least 3 million X 3 million double precision numbers in a single array (72 TB of memory) to perform this calculation without repeated file access. The other straightforward approach would be to step through the entire 1.7 GB database 3 million times (once for each sequence), putting a heavy load on file I/O.

While finding the global optimal alignment improves the statistics of homology detection, it comes at a computational price. As outlined above, the memory and computational requirements for global alignment methods are unwieldy for grand-challenge sized problems in computational biology. To address this problem, local alignment methods such as Smith-Waterman [[66], [67]], and FASTA [[68]] were developed. These local alignment methods do not produce a single global optimal alignment, but rather identify stretches of subsequences which align statistically, but which can be separated by gaps or mismatched regions. BLAST was a further improvement in the area of local alignment methods because it improved the computational run time significantly at the expense of sensitivity. BLAST is a heuristic method employing the notion of breaking sequences into words and finding local alignments of those words (or sequence substrings) among the collection of sequences of known biological function [[69], [70]]. BLAST is much faster than FASTA, Smith-Waterman or Needleman-Wunsch and is adequately sensitive for many applications [[71], [72]]. Because of its speed and availability through the NCBI website (`http://www.ncbi.nlm.nih.gov/BLAST`), BLAST has become one of the most fundamental and widely used computational tools for the post-genomic era biologist. Because of its central importance in computational biology, enhanced sequence alignment has been the target of many efforts making use of advanced computing capabilities. Specialized hardware [[73],[74]], service center architectures [[75]-[77]], POSIX thread-based applications [[69], [70]], shared memory applications [[78]], heterogeneous cluster applications [[79]], MPI-based distributed memory applications [[64]], and hybrid software/hardware applications [[80]] have all been developed to increase the throughput of sequence analysis. Work has also been done in managing the database sequences both during the query by using an alternative hash-table representation to approximate the database [[81]], and during the reporting of results by employing parallel I/O to exploit concurrency between processors after the alignments have been calculated [[82]]. In general, approaches that center around parallelizing the sequence database (either sharing it or modifying its image) have the advantage that larger databases can be maintained in-core during a run, limiting the need for constant file access. However, they tend to scale poorly, especially in the case of MPI-BLAST [[82], [83]]. On the other hand, approaches that center around parallelizing the query sequences (query scheduling) tend to scale well, but require repeated file access to the same database continuously through a run. Since the sequence databases are growing exponentially (see www.ncbi.nlm.nih.gov/Genbank/genbankstats.html),

the overhead from repeated I/O will continue to degrade performance. ScalaBLAST [[83]] was developed as a hybrid application with the intent of capitalizing on the scalability of query scheduling approaches, and at the same time sharing the target database using the Global Array toolkit [[84]-[86]] which provides a software-enabled shared memory interface to both true shared memory and distributed memory architectures. ScalaBLAST has been shown to scale nearly linear to the thousand processor range and because of database sharing in a global array object, ScalaBLAST keeps the entire database in-core throughout the run. Benchmark runs on an itanium 2 cluster over a quadrics elanIV interconnect (MPP2) demonstrated that ScalaBLAST processed over 448,000 queries from the PFAM [[87]] sequence database in 3.5 hours using 1500 processors. This opens the door for much larger grand challenge runs in which millions or more sequences are analyzed in a short time. However, ScalaBLAST has not yet been optimized on higher latency networks much more common in both academic and government laboratory facilities, and has been shown to have a factor of three slower run-time or worse. The aim of this study is to demonstrate significant speedup of ScalaBLAST on an opteron cluster over an infiniband interconnect-the NERSC Jacquard machine-in the context of two real science drivers for computational biology which fall into the grand challenge category. These problems were chosen because they both require on the order of 1 million queries to be performed, but with different search and output characteristics.

The first problem, relating to the set of environmental sequences representing many full microbial genomes collected from locations such as the Sargasso Sea, requires a standard BLAST search to be performed for each sequence against the entire nr database under the default conditions for protein searching. This is a good representative test case of the sort of search most users would want to perform. The average size of the sequences is close to the average size of proteins in general and the volume of output expected and the amount of computational work done per sequence is representative of what one would expect for any collection of complete proteins.

The second problem involves finding exact matches of very small protein fragments (peptides) used in the high-throughput proteomics facility at PNNL for the organism Salmonella T. This is a highly skewed scenario in which the sequences are extremely short and for which only the single best scoring exact match is retrieved from the nr database. This problem was chosen because it should provide a reverse map of peptides back to possible parent proteins (proteins which contain the exact peptide substring). To make sure optimizations made for the first representative problem are relevant to a wider range of BLAST problems, this second problem was chosen with a different balance between search and output time and a different frequency of hits expected from each query. Demonstrating sufficient speedup on both these problems with a single code base will ensure that optimizations resulting from this study are applicable to a wide range of computational biology problems.

Problem One: Environmental sequences The revolutionary technology of shotgun whole genome sequencing, 4.similar to the process used to complete the human

genome sequence, 4.was applied to a collection of evolutionarily primitive of organisms 4.from sites such as the Sargasso Sea [[88]]. These organisms have been sequenced revealing a collection of 1.2 million previously unrecorded proteins used in the baseline calculation, and more than 1.6 million proteins in the FY06 performance improvement metric. Much of initial effort in analyzing these biological populations has focused on the genetic and phylogenic profile of the samples, which gives hints about the evolutionary relationship between the organisms and which has revealed new insight into our origins. We propose to extend this by performing an exhaustive search of all predicted protein sequences from the so-called 'environmental' samples against the nonredundant protein database to identify relationships between well-characterized sequences and the environmental sequences. A search of this magnitude is important in identifying possible functions of these proteins and for enabling the downstream analysis of these genes using advanced tecnhiques such as genome context mining. Additionally, as the nr database continues to be updated, repeated exhaustive searches of the environmental samples are expected to give new information. But the computational bottleneck of performing the searches is prohibitive using conventional sequence alignment applications. We will demonstrate the baseline effeciency and scalability MPI BLAST as compared to the FY06 improvement provided by development of ScalaBLAST using this test case employing at least 50

Problem 2: Protein map for the Accurate Mass Tag (AMT) database High-throughput proteomics (HTP) is a cornerstone of modern systems biology research. This technological advancement makes it possible to measure the entire protein content (molecular machinery) of a biological sample in one experimental pipeline- a process which would have taken years or lifetimes using conventional analysis tecnhiques. Mass-spectrometry based HTP relies on identifying fragments of proteins (peptides) by producing extremely accurate measures of the mass-to-charge ratio of these peptides. In fact, one method of identifying peptides is by comparing them to a database of the mass to charge ratio of known peptides (AMT database). We propose to characterize the peptides in a production AMT (public Salmonella data) by identifying proteins in the nonredundant (nr) protein database which have exact subsequence matches. This would provide a valuable map from peptides back to protein sequences in a variety of organisms.

Description of Biological Systems The NCBI nonredundant protein database will be used for both the test problems. The February 2006 draft of this database, used in the baseline MPI BLAST performance measurements contains 3,284,262 sequences with 1,125,694,017 characters for an average protein length of 343 characters. This database contained all the sequences in the nonredundant database as distributed by NCBI as of February 2006. The binary sequence file associated with this database (nr.psq) has a total size of 1.1 GB. For the FY06 performance metric, improvements on the BLAST throughput attained by ScalaBLAST will be performed on the May 2006 draft of the database containing 3,658,078 protein sequences and a total of 1,256,998,427 characters for an average protein length of 344 characters.

For the baseline assessment of MPI BLAST, the first test problem (environmental sequence search), the collection of 994,617 environmental sequences as distributed by NCBI in March 2006 contains a total of 194,022,063 characters, for an average sequence length of 195 characters per sequence. The total size of the input file was 247MB. These were aligned against the nr database (as described above) using all default arguments for the standard NCBI blastp executable. Run-time parameters are discussed in detail in the Parameter Selection section below. The default values were chosen to get alignments representative of the default way in which BLAST and hence ScalaBLAST would be used by most users. The output of this first test problem was a collection of similar sequences for each of the environmental sequences, providing the impetus to perform further processing (beyond the scope of this project) to mine for features and patterns within this set of homologs for each protein. For the FY06 performance improvement attained by ScalaBLAST, the first test problem was performed on a more recent version of the environmental sample collection as provided by the Joint Genome Institute. This updated version contained 1,618,944 protein sequences from over 400 microbial genomes with a total of 532,358,124 characters for an average protein size of 329 amino acid residues. Since this test problem was of significant interest to JGI, we chose different output and filtering options (as discussed below), but since MPI BLAST failed to produce any output in the baseline tests related to the size of the input file (rather than as a consequence of filtering options or output format), the test between FY06 improvement gain in ScalaBLAST vs. MPI BLAST baseline is still valid.

For the second test problem (Accurate Mass Tag database peptide matches), a collection of 1,418,939 peptides was selected with a total of 34,209,722 characters and an average length of 24 characters. No peptides were shorter than 10 characters. The total input file size was 83MB. These sequences compose the public AMT database for Salmonella at the Pacific Northwest National Laboratory high-throughput proteomics facility. These were aligned against each of the nr database sequences (as described above) using a parameter set chosen to select only a single exact match for each peptide. The default values are discussed in greater detail in the Parameter Selection section below. Identical input files, target database and command-line options were used on both the baseline runs on MPI BLAST and the FY06 performance improvement runs for ScalaBLAST.

0.16.2 Algorithm

Underlying Algorithms of BLAST Sequence Alignment The core algorithm of BLAST is character matching. The measure of homology between two sequences is calculated by finding the maximal segment pair (MSP), which is the subsequence pair of identical lengths which have the highest similarity score. Two completely identical sequences will have a maximal segment pair equal to the sequences. For one sequence which is an exact subsequence of another, their maximal segment pair is equal to the subsequence itself. But often in nature the maximal segment pair is not

as straightforward. Some amino acids can be commonly replaced with another with little penalty to the survivability of the organism. But some amino acids are so fundamentally different (polar vs. nonpolar; aromatic vs. acidic) that the spontaneous mutation underlying the mismatch is much less likely to occur and be propagated.

BLAST uses the notion of a scoring matrix to quantify the frequency (and hence the inverse of the penalty) associated with all amino acid substitutions or nucleotide substitutions in the case of genes. A protein scoring matrix can be thought of as a 20x20 field in which each of the 20 amino acids is represented along the vertical and horizontal axis. The i,j th entry contains a relative measure of the likelihood of seeing amino acid i propagate to amino acid j in an evolutionary related protein. This matrix is constructed based on statistical measures of the frequency of mutations in nature so that unlikely substitutions are heavily penalized and likely substitutions or an exact match are highly rewarded.

BLAST uses a small piece of the query sequence (a word whose size can be changed by the user) to locate a region matching the target database sequence. When a matching region is found, BLAST attempts to extend the match by adding more words to the alignment. BLAST proceeds in this way until the best-scoring collection of MSP's has been located. The final score for each candidate alignment can also include a gap initiation penalty and a gap extension penalty. The collection of disjoint MSP's are considered to represent evolutionarily conserved subregions of the sequences which are connected by mismatched sequences or sequence gaps.

For each alignment reported from the sequence database for a given query, two associated metrics are reported: bitscore and expect (E) value. Bitscore is calculated using the number of matches or near matches (matches with low penalty in the scoring matrix). Higher bitscore suggests a higher degree of homology for a given query, so results are given sorted by bitscore for each query. E value is a measure of the number of random hits one would expect to find for a given bitscore in a database of the size of the target database. A large E value suggests that the sequence is more likely to have occurred by chance and an E value close to 0 lends more evidence to the argument that the match is not by chance. For a more detailed discussion on the statistics used by BLAST to describe homology, see www.ncbi.nlm.nih.gov/BLAST/tutorial/Altschul-1.html.

By default, query sequences are screened for low complexity regions. Low complexity is characterized by segments containing some types of repeating motifs, or of stretches dominated by a single character. These stretches can create homolog artifacts when the low-complexity region is the basis of a match. They are automatically masked by BLAST to prevent this anomaly.

Computational Approach of MPI BLAST MPI BLAST was developed as a database parsing method in which a large sequence database is preformatted into a collection of smaller database fragments [[64]]. At run-time, database fragments

are distributed among processors responsible for calculating the partial results of each query against their local database fragment. This method has been shown to give superlinear speedup when the database fragments are small enough to fit in-core for large databases [[89]]. This has been chosen as our baseline for the OMB software effectiveness metric because MPI BLAST is the accepted state-of-the-art in high performance sequence alignment at the start of FY06. For this study, we will be using MPI BLAST 1.4.0.

MPI BLAST Code Overview Using MPI BLAST requires preformatting the target database into a predetermined number of fragments using the executable mpiformatdb. When finished, the user can select the target database fragments on which to run the queries. According to the MPI BLAST 1.4.0 documentation, a maximum of 250 database fragments can be created with mpiformatdb. One may expect that for a database like nr, one should use as many fragments as possible for the 1000 processor run. However, we have used small test cases to estimate the best configuration for running MPI BLAST 1.4.0 for the baseline measurement of the OMB software effectiveness metric. We have identified two competing limitations of MPI BLAST 1.4.0. On the one hand, there is a performance penalty for using many more processors than database fragments. While the code does handle this case, parallel efficiency drops off rapidly. For example, on a test query of 99 sequences against the nr database parsed into 3 fragments we observed a speedup of only 37 times when running on 64 processors. One would expect this to be significantly worse when running on 1000 processors. At the other end of the spectrum, there is a penalty for using more database fragments on a given number of processors. For example, on the same test query of 99 sequences, using 64 processors against 3 vs. 63 database partitions, using 63 database partitions increased the run time by a factor of 6.4. These two run-time limitations to MPI BLAST motivated the design of ScalaBLAST in the first place. But since ScalaBLAST was not yet available and was not yet fully functional at the outset of FY06, we must compare our improvement in performance during FY06 to a baseline performance of MPI BLAST as it existed at the outset of FY06. Since the limitation in scalability (50% dropoff in efficiency as 64 processors) for our test problem appears less pronounced than the penalty for using more database fragments, we use 3 database fragments for the baseline measurement with the intention of best representing performance of MPI BLAST against the nr database.

0.16.3 Machines

Machine One The HP/Linux Itanium-2 over Quadrics in the Molecular Science Computing Facility at the Pacific Northwest National Laboratory (http://mscf.emsl.pnl.gov/hardware/conf is being utilized for the current study.

Machine Two Concurrently, the AMD Opteron Cluster over InfiniBand at NERSC (http://www.nersc.gov/nusers/resources/jacquard/) is being utilized to study the same problem base.

0.16.4 Data

Q2 Collected Data	Problem One(0.16.1)	Problem Two(0.16.1)
Source codes	yes	yes
Build information	yes	yes
Runtime information	yes	yes
Total walltime	yes	yes

Table 25: Information collected during Q2 for the computed ScalaBLAST problems. Please see sec[0.2.4] for description.

Description of Test Problems and Performance Metrics The test problems were designed to probe the throughput of MPI BLAST and ScalaBLAST on large-scale BLAST calculations. The key metric is therefore sustained throughput defined as number of queries per processor completed per minute of wallclock time consumed by the application (normalized to a database size of 1 million proteins). For the ScalaBLAST benchmark runs performed on Jacquard, a 6-hour run-time was selected (to conform to Jacquard queue policy) during which ScalaBLAST processed as much of the query list as possible using 320 processors. For the ScalaBLAST benchmark runs performed on MPP2, the ENV benchmark was performed until completion of all 1.6 million sequences (for a total of just over 18 hours broken up over two runs) and the STP benchmark was run for 8.5 hours. The nr database is a representative database containing the full list of proteins sequenced to date with duplicates omitted. Demonstrating throughput against nr is a good indication that an application can drive high-throughput BLAST calculations against a typical protein database. To gague the impact of system performance on throughput from the standpoint of I/O, the two benchmarks were chosen to flex I/O in different ways. First, the ENV benchmark uses the 'hit table' output format, common in many large-throughput tasks because it omits the details of the alignments themselves and rather focuses on the scores. This option was requested by the Joint Genome Institute collaboration aspect of this test, in which the microbial genome database (ENV) results are being used by JGI in their upcoming version of the IMG database. Hence this output is a relevant and representative format for real-world users. This output format in ScalaBLAST has very similar run-time performance, scaling and machine utilization to the full output format as output is a tiny fraction of the overall run-time on the systems tested. Second, the STP benchmark uses the full output, but only reports the single top hit for each query. This forces output of the alignment and statistics as well as header and query information, but is not as extensive as the default output options, which force 500 alignments to be printed.

Parameter Selection Run-time options used for the baseline performance assessment of MPI BLAST for the ENV benchmark were all taken as the default options, whereas for the FY06 improvement assessment of ScalaBLAST the command line used on MPP2 was as follows: -m 8 -F 'm S' -e 1e-5 -b 2000. On Jacquard, the filter string 'm S' was not interpreted at run-time by csh, so only the 'm' option was used. We tried several variants on this including environment variables, etc but we did not find a workaround for this. For the second problem (AMT database), the MPI BLAST baseline and the ScalaBLAST OMB software effectiveness benchmark on both MPP2 and Jacquard both used the following command-line arguments : -e 20000 -W 2 -M PAM30 -G 9 -E 1 -v 1 -b 1 and the same nr database as the first problem.

Baseline Performance and Scaling (MPI BLAST) For all of the large benchmarks attempted on both systems (MPP2 and Jacquard), MPI BLAST failed to produce any output. Several key limitations in MPI BLAST which are addressed in ScalaBLAST combine to produce this behavior. First the scaling is poor, regardless of the number of databases being used (see fig. 1). Even though MPI BLAST is intended to be used on systems where the number of database partitions is greater than or equal to the number of processors, it was prohitive in these benchmarks to use any more than 3 database fragments. Second, large input files cause MPI BLAST to stall in the query distribution phase. We have used NWPerf, a performance monitoring tool on MPP2 to reconstruct a detailed profile of the run-time behavior of MPI BLAST for the ENV benchmark. Using this set of results, it can be shown that a memory usage limitation caused by large input files causes the master node to run out of memory and begin swapping, essentially stalling the entire application. This benchmark ran for over 5 hours on 1000 processors on MPP2 producing no output, with the performance characteristics illustrated in the following figures.

DRAFT

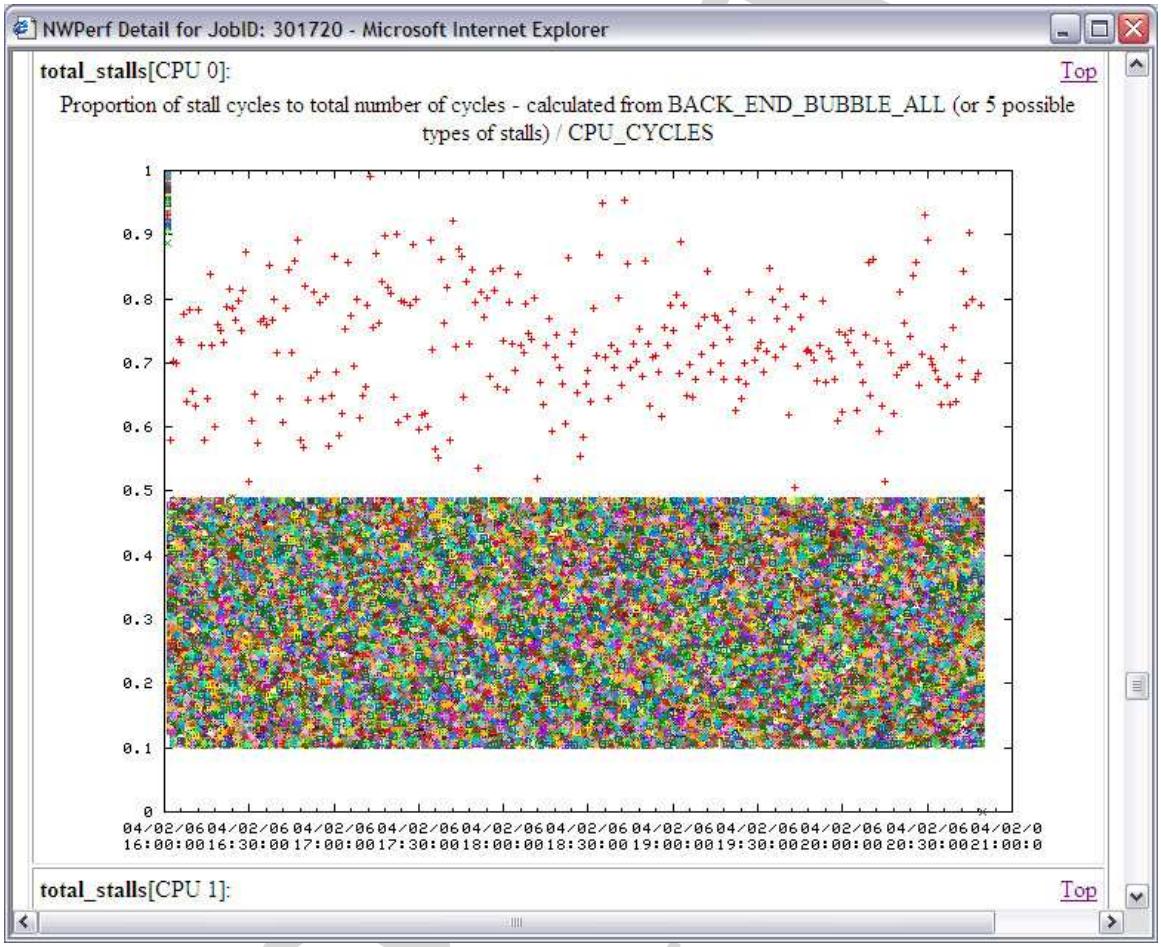

Figure 29. Output of NWPerf for the MPI BLAST ENV test which produced no output running on 1000 processors on MPP2 after several hours. For the duration of the run, the master node is essentially 80% stalled.

DRAFT

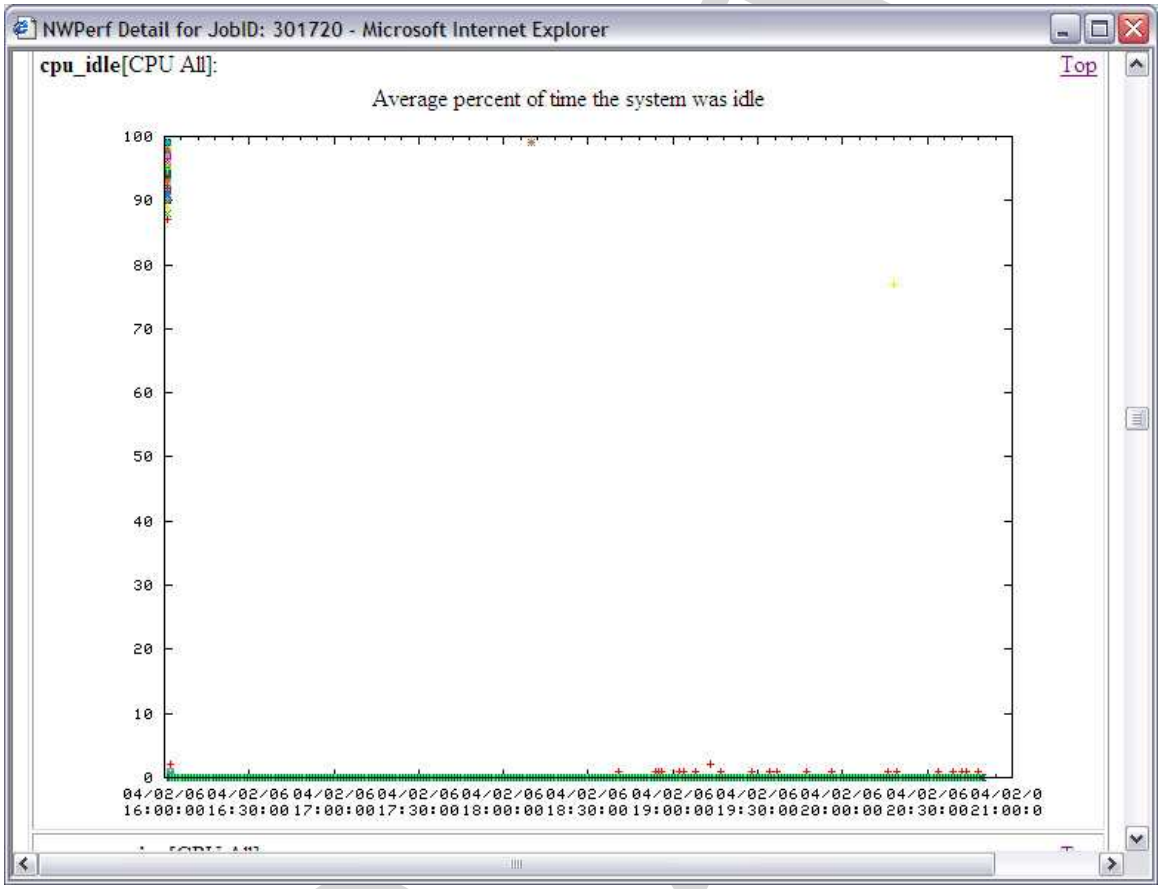

Figure 30. NWPerf output illustrating that virtually no time on any processor was spent idle. MPI BLAST ENV test on MPP2 resulted in a spin-wait state for all processors just after startup

First, looking at hardware stalls (Figure[29]), one can see that the master node remains at a high stall rate (80essentally spin-waiting. Figure[30] illustrates that this stall rate is not a consequence of idle processors, as virtually all of the processors have an idle rate near 0, and are spending all their time in user space (Figure[**??**]). Figure[32] illustrates that the floating point operation rate is very low for all processors, but that the master node is performing a small number of flops throughout the job.

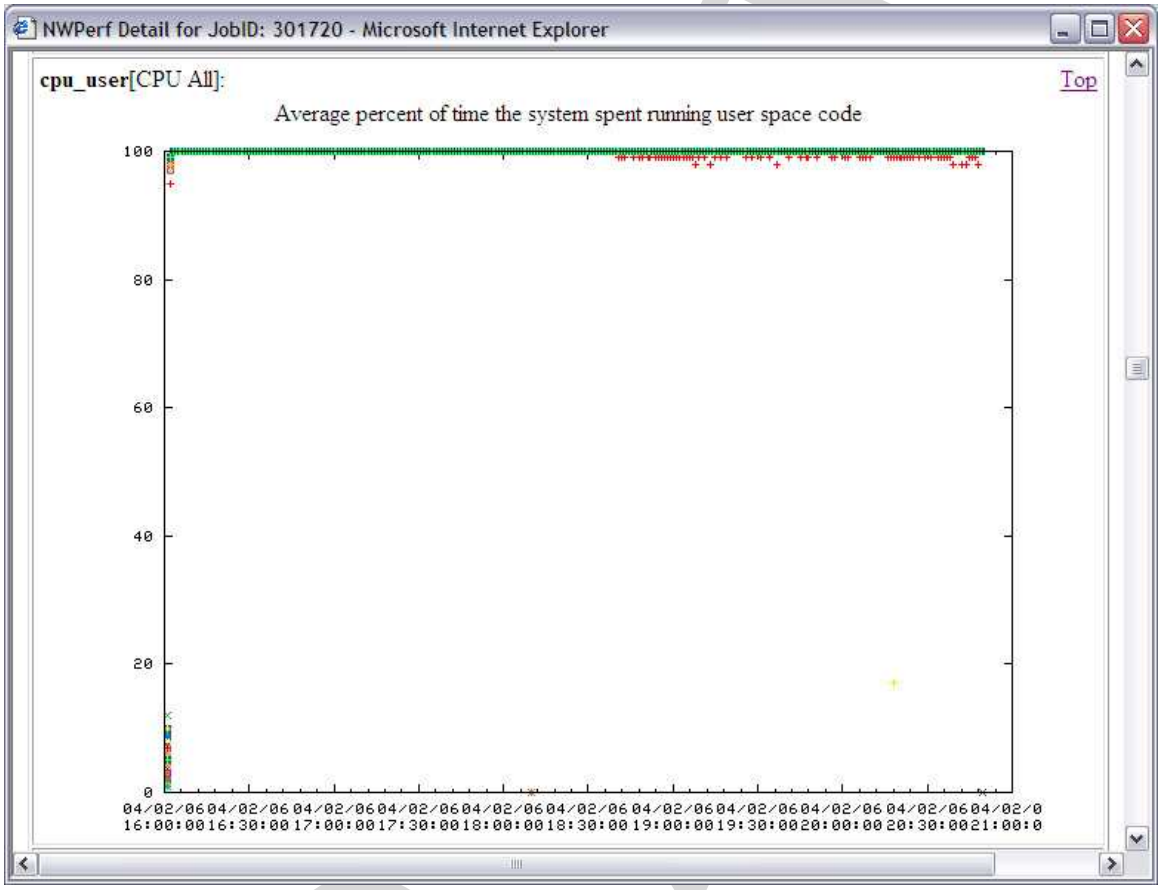

Figure 31. NWPerf output for MPI BLAST ENV run on 1000 processors shows all cpu's almost completely dedicated to user space tasks (not blocking on I/O)

DRAFT

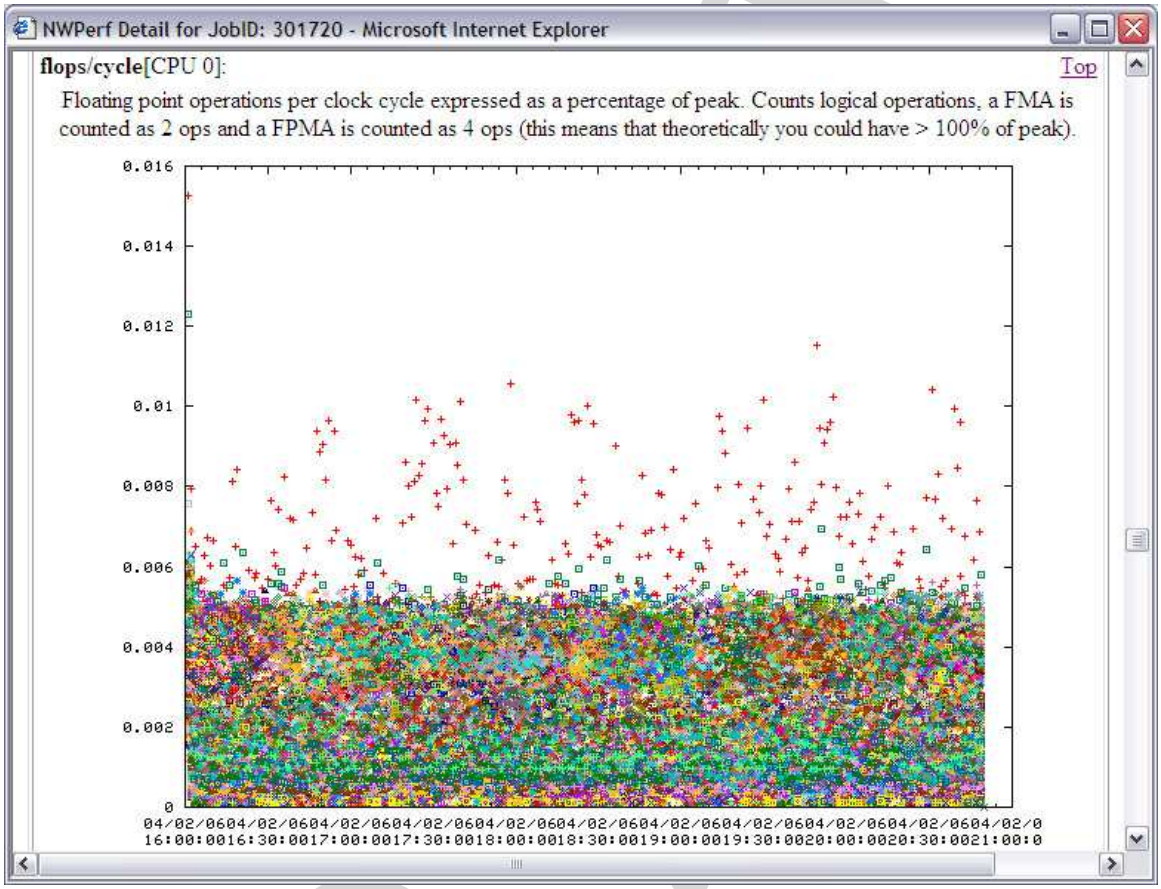

Figure 32. NWPerf output for ENV run of MPI BLAST on 1000 processors illustrates very little floating point operations occur during the run, even on the master processor (red dots above).

Baseline Communication, Memory Utilization and I/O Overhead (MPI BLAST) The real crux of the limitation we found with MPI BLAST is memory usage. Figure[33] illustrates the memory footprint of MPI BLAST, which grew linearly on the master node in time during execution of the large ENV benchmark. This trend continued until all the memory of the master node was consumed. Figure[34] illustrates the corresponding drop in free memory for the master node during this time. Once memory is consumed on the master node, swapping begins, as illustrated in Figures[35,36]. At this point, the calculation has essentially ground to a halt-before any BLAST output has been produced.

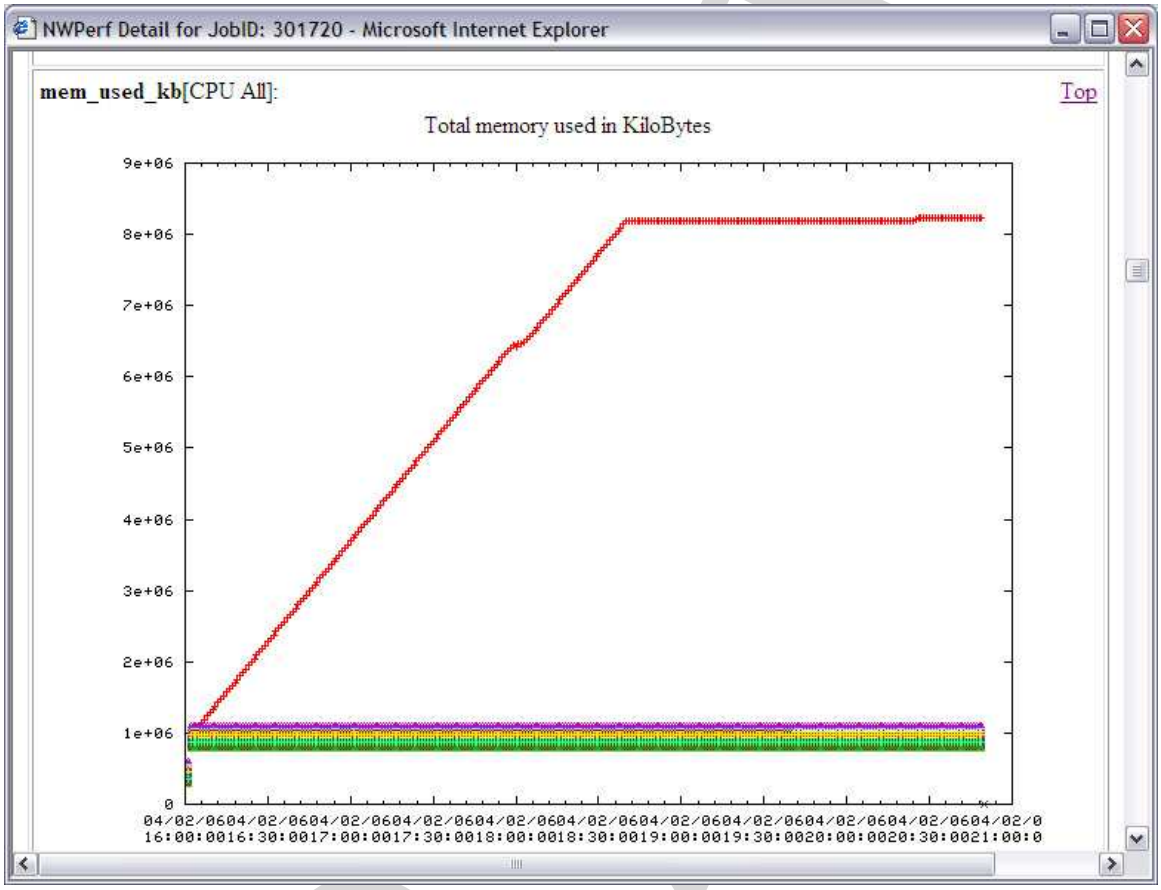

Figure 33. NWPerf results from 1000 processor MPI BLAST run of ENV benchmark problem on MPP2 illustrates memory usage for master node rises to total memory per node (8GB) before swapping begins.

DRAFT

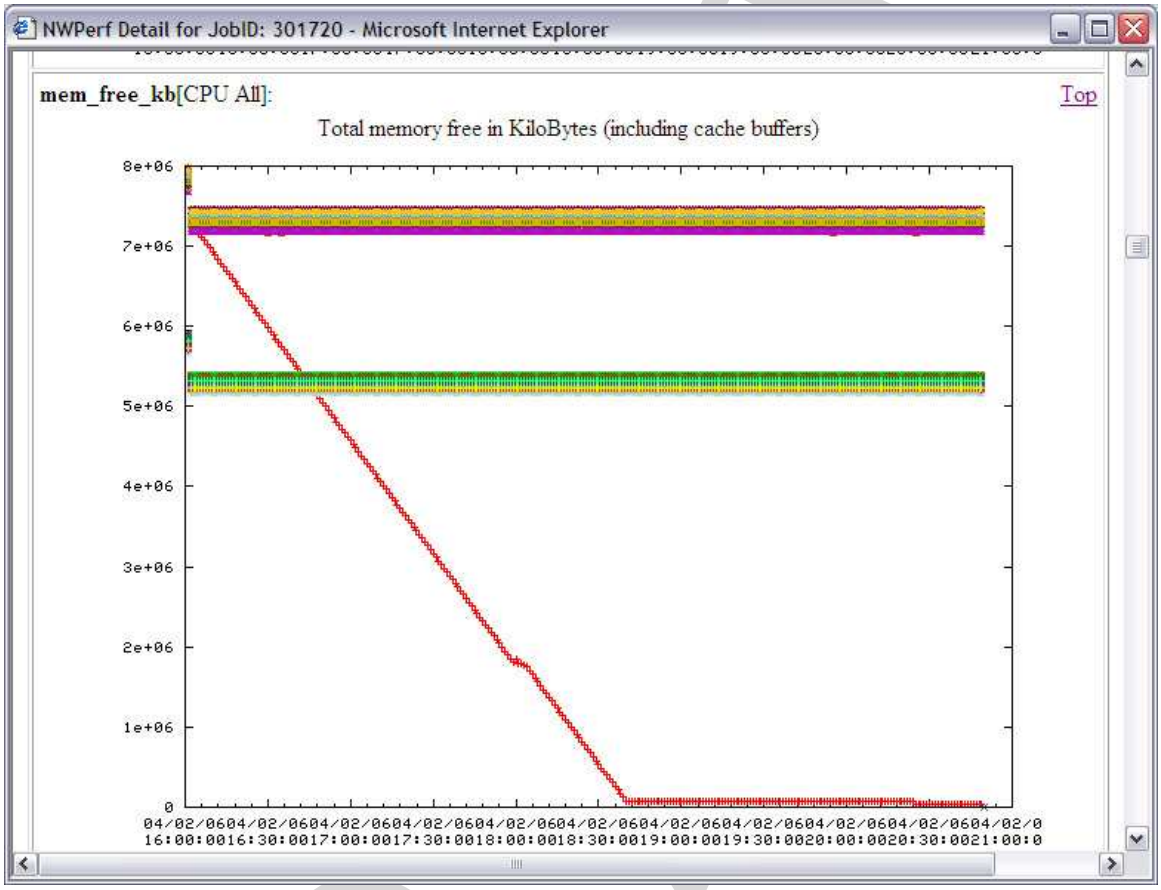

Figure 34. NWPerf results for ENV benchmark of MPI BLAST on 1000 processors of MPP2. Free memory drops to 0 as memory is consumed by the application during the query assignment phase.

DRAFT

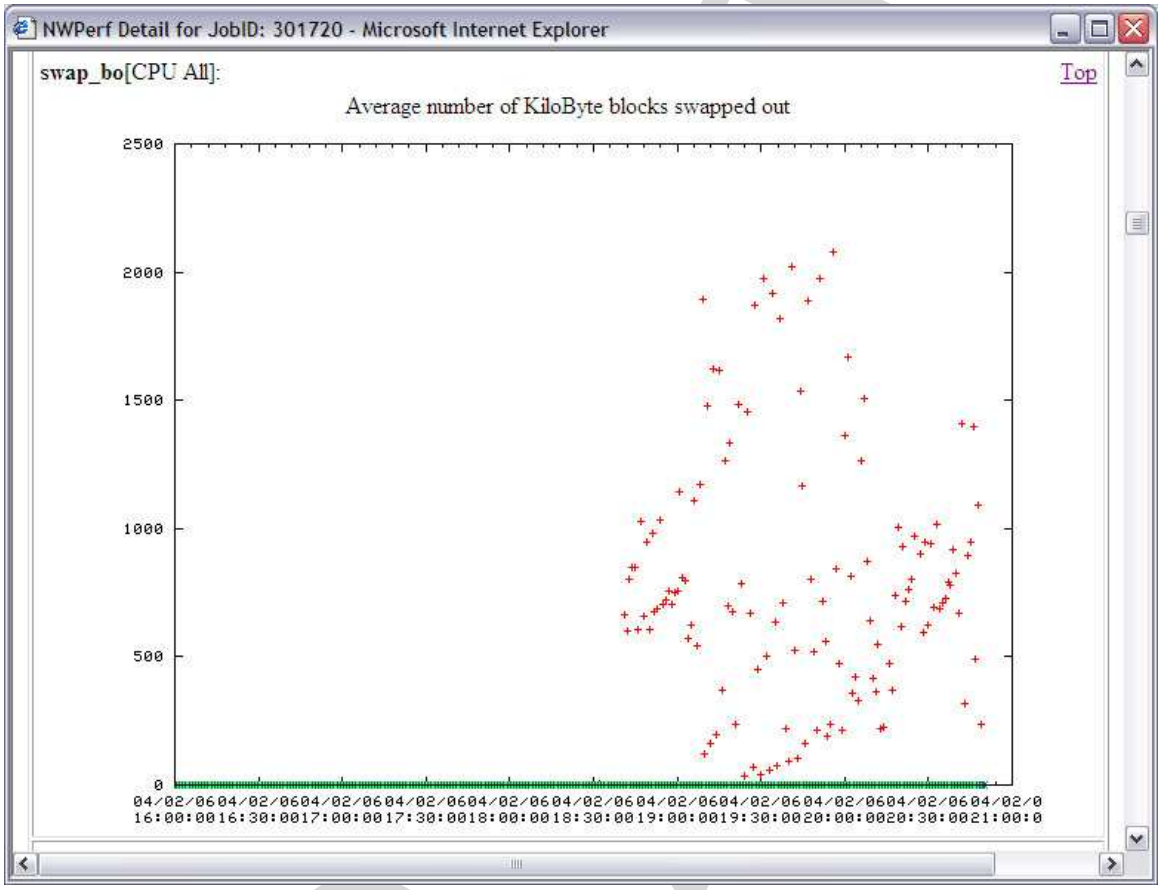

Figure 35. NWPerf results for MPI BLAST ENV run on 1000 processors of MPP2 illustrates processor 0 (the master node) begins swapping out bytes around the same time it runs out of memory.

DRAFT

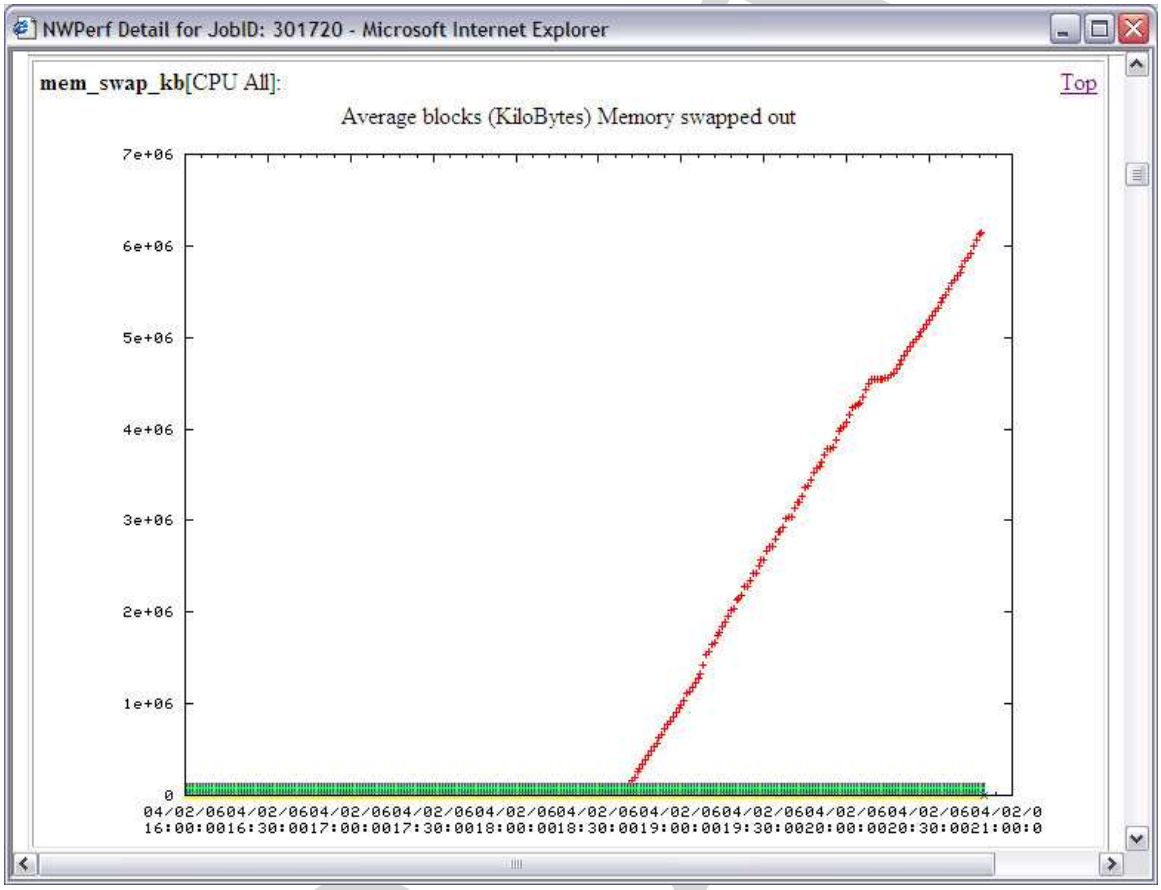

Figure 36. NWPerf output for 1000 processor MPP2 run of ENV benchmark for MPI BLAST reveals the memory swapping problem which begins after the node's full memory is consumed.

DRAFT

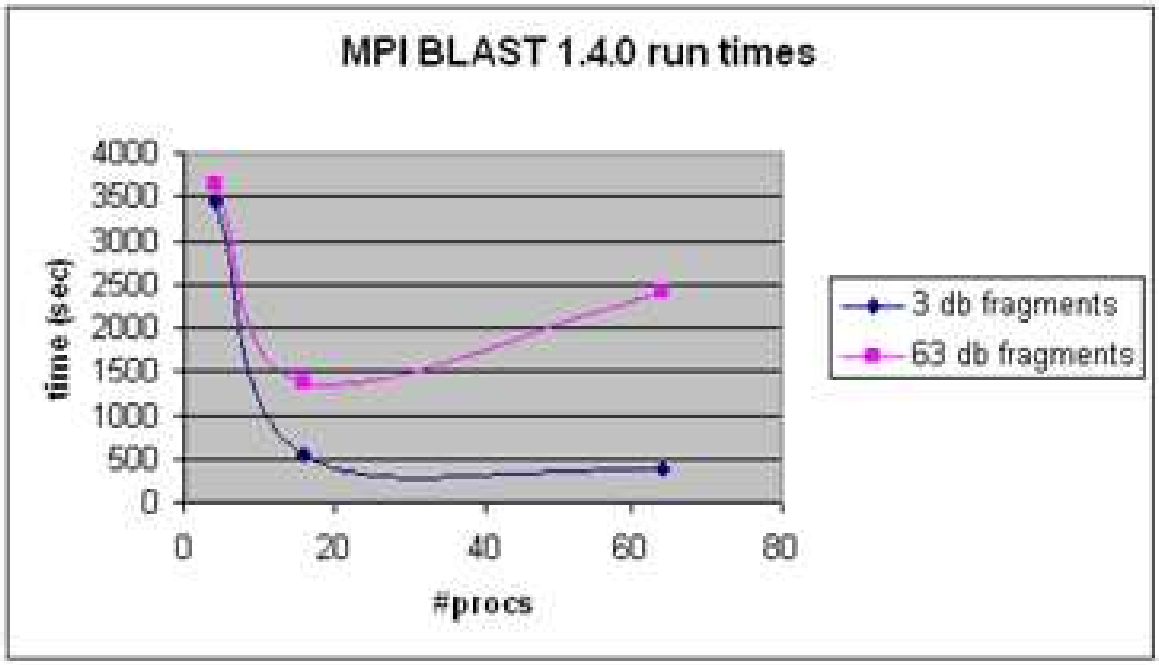

Figure 37. The results of profiling of MPI BLAST 1.4.0 on 99 queries against the nr database.

Small Parallel Benchmark Profiling: MPI BLAST In addition to the problems with scaling of MPI BLAST with respect to number of database fragments used, we also identified a key limitation which is described in detail below. MPI BLAST failed to produce any output for either of the OMB benchmark runs on both Jacquard and MPP2 in large part because of its inability to handle a very large input file. We show in following sections that MPI BLAST also would likely have not produced any output for much longer run-times because it exhausted memory on the systems we tried, forcing them into a swapping situation. On both Jacquard and MPP2, adverse effects on the system resulted from this problem, prompting system administrators to ask us to not run MPI BLAST anymore for problems of this size. After verifying that the problem was related to memory usage (detailed below), we ran a series of small benchmarks to study the effect of varying either number of processors (np), or number of queries (nq) in the input file. Results of this study are illustrated in Table [26]. These runs were performed with the 'hit table' output option selected.

The key finding here is that in the short 20 minute test runs we performed, once the input file contained 100000 queries, MPI BLAST failed to produce any output. We widened this constraint for the large OMB runs and allowed it to run for several hours in some cases, but in all large runs MPI BLAST failed to produce output. In table 1, one can see that increasing the number of processors while holding the size of the input file constant resulted in a degradation in performance. Additionally, using larger input files while holding the process count at 256 also resulted in a slight

procs	Time (m)	nq in	nq out	nq/(proc*min)	db frags
64	24.7	1000	1000	2.317194679	3
128	25.5	1000	826	0.925726847	3
256	25.35	1000	796	0.448691767	3
256	25.45	10000	699	0.392466313	3
256	25.43	100000	0	0	3

Table 26: MPI BLAST performance on MPP2 on varying input file sizes and process counts

degradation in performance at an input size of 10000 queries and no output at all in 20 minutes on 100000 queries. The negative impact to both testing sites for these runs which consume half the machine for hours or more without producing output is substantial both in terms of negative user impact (i.e. blocking other users during simple I/O tasks) and wasted CPU cycles. The I/O strategy of ScalaBLAST is aimed at taking full advantage of globally mounted filesystems to eliminate this serial I/O bottleneck, and the memory problems associated with trafficking this entire list over MPI.

0.17 Q4 Information

The problems and the machines remained the same. The algorithm was modified. A description now follows of the changes. A new science capability was delivered as a result.

0.17.1 Algorithm

Computational Approach of ScalaBLAST As outlined in the Introduction, the two key capability limitations to performing very large BLAST searches that were addressed by ScalaBLAST are 1) inability of query scheduling applications to handle databases too large to fit in-core without swapping and 2) poor scaling of database parsing methods. The following discussion covers the algorithm design aspects of the ScalaBLAST extension to BLAST which have been shown to overcome these two capability limitaitons of the conventional state-of-the-art sequence alignment implementations [[83]].

Database sharing at run-time Unlike the currently accepted parallel BLAST implementation MPI-BLAST, which requires the user to pre-format the target database by dividing it into pieces, ScalaBLAST opens the target database at run-time and

evenly divides it among memory segments in the processors available at run-time. As long as the aggregate memory of the processor collection is enough to hold the target database and associated data structures for bookkeeping, ScalaBLAST operates sensibly on any processor count we have tried it on. Database sharing in ScalaBLAST is achieved by instantiating a global array object across all the processors large enough to hold the character strings associated with each sequence. At run-time, the database reading function retrieves sequences from the global array object, rather than from the native memory-mapped files. This makes it possible to keep single database objects in-core for the duration of a run, eliminating the need for constant file access to database volumes which is the current approach for databases too large to fit in-core or too large to address with 4 byte integers.

Query scheduling and load balancing Rather than force all processors to work on a single query together, as is generally done by database parsing methods, ScalaBLAST divides the processors into process groups (of size 2 for now). At run-time, each process group is assigned a disjoint subset of the query list, eliminating the need for communication between process groups during the run. Within a process group, the work of aligning each query against all the database sequences is shared between the processors-each processor is responsible for obtaining partial results from exactly of the database global array object. Within a process group, the processors can work in relative asynchrony because they post the partial results to a small global array object shared by only the process group. So later when it is time to produce output, either processor can merge the partial results into a single output object and sent it to the NCBI print routines. To ensure that all process groups have a balanced fraction of the total query load, a crude approximation of the total work is estimated based on the number of characters in each query and the number of queries. The work estimate is calculated from the entire query list and divided such that each processor performs roughly the same volume of queries where volume is determined by number of total characters in the collection of sequences and a penalty for the number of sequences in the list. The specific parameters used for this static load balance scheme are outlined in the Parameter Selection section below.

Reporting results to files asynchronously To capitalize on the concurrency in print and query tasks, a two-tiered approach to output has been adopted for ScalaBLAST. As discussed above, the query list is distributed among the process groups participating in the alignment calculations. Since the query list fragments being processed are disjoint, each process group is free to print to its own files completely independently of all other process groups. This can be done using local scratch disk space, or over a network to a globally mounted filesystem. The second tier of concurrency occurs within process groups. The normal sequence of a BLAST query has interleaved printing and searching tasks. But ScalaBLAST has been designed to cleanly separate the query phase from the output phase. First a collection of input queries are run through the BLAST engine and the partial results posted indepently by the two processors in the process group. Since these partial results are shared in a

global array object, a single processor can proceed with the next BLAST calculation before its companion processor has finished the task. So each processor proceeds with its own partial queries posting the results after each assignment to the process group global array object. When a predetermined number of these queries has been completed, a processor switches from query mode to print mode. In print mode, each processor is assigned the duty of printing half of the results to its own output file. All that is required during this phase is for both halves of the query to have been completed at some point prior to printing. This is the only possible block in this sequence of operations. Iteration over several queries has been used as a way to eliminate the majority of blocking that would occur at this point, since most of the time one processor is only behind the other by one or two queries. Each processor prints its assigned output to a separate file (again, either in local scratch space or over a globally mounted filesystem), taking full advantage of the independence of the queries to drive I/O at its maximum rate.

Sequence block prefetching On distributed memory systems, the main performance consideration for ScalaBLAST arises from interconnect latency. Blocking 'get' operations supported by Global Array toolkit are used in the most straightforward application of ScalaBLAST to retrieve sequences from the global array sequence shared object to individual processors during a run. But each sequence has an average length shorter than 300 characters, and there are currently over 3 million sequences in the nr database. So for 1 million queries, each of 3 million short sequences would be individually sent via global array operations to each process group. For low latency interconnects, this represents a performance penalty of roughly 50interconnects, it is possible this bottleneck would dominate run time degrading computational efficiency to an unacceptable level and interfering with scalability. So ScalaBLAST was outfitted with a block prefetching option in which large sequence blocks are retrieved using nonblocking global array 'get' operations. As one block is being processed, the next one is being prefetched. When the end of the current block is reached in local memory, a 'wait' is issued until the prefetch operation has completed. Using this prefetch method, virtually all the overhead associated with sequence retrieval from the global array sequence object was hidden by overlapping fewer large retrieve operations with the query processing itself [[83]].

Exploiting concurrency between query and I/O tasks Concurrency in the nature of large BLAST searches has been exposed and exploited by ScalaBLAST in two separate design features. Since BLAST compares a query sequence against each sequence in the target database in a predictable order, (for the first pass), prefetching algorithm discussed above is a natural way to overlap communication cost of retrieving sequences with the computational work of processing a query. The second way in which ScalaBLAST exploits concurrency in large BLAST searches is by dividing the BLAST driver into query and print phases, rather than allowing them to stay interleaved as they are in the standard NCBI toolkit distribution. This makes it possible to iterate over several queries, accumulating partial results before moving

to the print phase. But the real benefit is more subtle. ScalaBLAST was designed to maximize I/O by allowing each processor to print results to its own file. If two processors perform a query task, then a single processor must collate and print those results, the second processor can only get as far as performing the next partial query before it must stop and wait. It cannot print until the first processor has caught up, posting its own partial results for the second query. This bottleneck would occur on each query and the resulting inefficiency would compromise scalability. There is often great variability in the duration of the query phase for a given sequence between the processor assignments for that query. For instance, one processor may find no significant hits in its portion of the database search while the other processor may find thousands of significant hits in its portion of the database for the same query. Likewise between queries, one may see a query with many hits and long alignments followed by a query with few hits and straightforward alignments. When this sort of variability in run time for tasks occurrs, it usually pays off to average over many tasks. The more tasks one can do before enforcing a 'barrier' (or 'sync' in the case of global arrays), the more likely the run times for each task are to even each other out. ScalaBLAST uses this to its advantage by scheduling multiple query tasks to be performed by each processor in the group (which can be done asynchronously) followed by a collection of print tasks. The only dependency in the tasks is that for a given query to be printed, both of its partial results must have been completed and posted. But since we can iterate over many queries before moving to the print phase, most of the variability in the task times averages out, increasing efficiency and throughput for very large jobs.

ScalaBLAST Code Overview

Layout of the ScalaBLAST driver The driver function for ScalaBLAST is derived from the NCBI toolkit file 'blastall.c'. When the user runs the ScalaBLAST executable on the command line of a submission script, the main driver ('ncbimain.c') calls a function called Nlm_Main, which is defined in the file 'sblastall.c'. This driver processes the input arguments and sets up the run using the same procedure used by 'blastall.c'. However, the normal BLAST executable immediately opens the user-specified output file and prints header information and begins printing tick marks to indicate the passage of time. ScalaBLAST has been modified so that it loops over 10 queries (or as many as remain in the process group's assignment) only performing query-related tasks. After all 10 partial queries are done and the results posted by a processor, that processor moves to the print phase. Even numbered processors are responsible for dumping the even numbered query's output to its file. Likewise odd-numbered processors handle odd-numbered queries. When a processor is printing, it first checks to make sure both partial results have been posted to the results global array object. If so, then it localizes the two half-sequence alignment objects from the global array and manually merges them into a single list. Then this linked list is sent to the normal NCBI print routine. When all printing is done, the processor waits for the other processor in the group to finish, and then both processors re-enter the

query phase if any queries remain unprocessed.

Creating this driver from the distributed NCBI 'blastall.c' file is done by first pushing all the argument setup code above the loop. Then a loop is created in which all the query-related functions occur in a subloop in the top, followed by a printing subloop at the bottom. Initialization of the query has to happen twice-once during the query phase and again during the printing phase so file pointers must be maintained so that they can be rewound appropriately once the query phase is finished. During the print phase of the driver, only_mask_loc (discussed below) is set to TRUE, forcing BLAST to initialize the data structures needed for printing output, but not actually perform the search.

Prefetching, as described above, is performed for each query during the first pass over the database. When the last database block is prefetched, a flag is set which allows future database operations to directly retrieve a single sequence using blocking 'get' calls. As a result, the driver must notify the database-reading function whenever a new sequence is begun. This is done using global variables set by the driver indicating it is the first time for this sequence through the database reading step. This variable is detected in the database-reading function, as described in the next section in greater detail.

The driver uses two primary hooks into the ScalaBLAST functions, contained in the file 'sb_funcs.c'. These functions are primarily for creating, initializing, and managing the global array objects. First the driver creates a the collection of global array objects, then assigns processor 1 the task of reading the user-specified binary sequence file, byte-swapping its binary entries, and posting the sequence data to the sequence global array. Processor 0 on each process group is responsible for opening the sequence index file and filling the global array objects local to each process group with the values of starting locations of each of the sequences. This way, each processor has access to the starting and ending index of each sequence in the global array sequence object.

The driver's next task is to assign one half of the database as the search target for the processors. In each group, proc 0 is responsible for searching the second half of the database and proc 1 is responsible for searching the first half of the database. This is done by overwriting the options-¿first_db_seq and options-¿final_db_seq values where the final sequence index for process 0 and first sequence index for process 1 is determined as the sequence containing the n/2 th character where n is the number of characters in the whole database.

The last special role of the driver occurs during the output phase. After re-initializing the query with only_mask_loc set to TRUE, the driver calls the ScalaBLAST function GA_Reconstitute_SeqAnnot, which localizes both partial results for a given query and merges the results into a local linked list structure which is then dispatched to the NCBI print routines.

Modifications to the NCBI toolkit source code for ScalaBLAST Rather than change the NCBI BLAST API, ScalaBLAST passes some variable to BLAST through global variables. These include a flag indicating whether prefetching should be used, information about the number of processors and the process identity, process group information, and control flags for managing passes through the database. These globals are referenced in the tools/readdb.c file.

ScalaBLAST intercepts native NCBI calls to database reading functions and a new block of code is executed instead. This block reads the ID number of the sequence (the sequence index, which is normally passed to the database reading function) and gets the sequence string from the global array object containing the sequence. If prefetching is used, this is read from a local buffer into which a block of sequences has been already placed. If prefetching is not used, a blocking GA_Get is issued to directly retrieve the sequence into local memory.

Currently, the block size of the prefetch unit is predetermined (by a $\#define$ directive) in both the NCBI standard file readdb.c, and in the ScalaBLAST driver file scalablastall.c. These values can be manually set to tune the prefetch block size for different systems, but they must agree with one another. To prevent the second initialization phase from doing actual alignment calculations, a new element (Boolean only_mask_loc) has been added to the BLAST_OptionsBlk struct defined in 'blastdef.h'. If only_mask_loc is set to TRUE, BLAST is re-routed to only create the data structures needed for printing the sequence alignment calculation and not attempt to perform an alignment. The modification required to intercept this new option is located in the file 'blastutl.c'. It tests if only_mask_loc is set to TRUE, and if so skips over the call to BioseqBlastEngineCore, which calls the core BLAST algorithm.

Porting ScalaBLAST to the Opteron According to the NCBI website and the NCBI code documentation, there is no native support for BLAST on 64 bit AMD Opteron. However, a patch is available from Scalable Informatics (http://downloads.scalableinformatics.com To use this patch, the newest version of the NCBI toolkit was downloaded to Jacquard and the accompanying ScalaBLAST had to be changed accordingly. The ScalaBLAST driver had to be completely reworked to function correctly with the newest version of the NCBI tookit and the Scalable Informatics patch was modified to build the NCBI toolkit on Jacquard to be used with ScalaBLAST. Because ScalaBLAST uses MPI and GA libraries in the readdb functions and in the Nlm_Main wrapper, the standard NCBI build must be modified so that the proper libraries and include files are available at compile and run-time. Similar changes were made to the NCBI build on MPP2, a 1,960 processor Itanium II cluster housed in the William R. Wiley Environmental Molecular Science Laboratory's (EMSL) Molecular Science Computing Facility (MSCF) so that ScalaBLAST could be made to run with GA and MPI libraries there as well.

DRAFT

ScalaBLAST build details Intel 8 compiler was used to build ScalaBLAST on MPP2, and pathcc was used to build both applications on Jacquard. On both platforms, the associated file in the ncbi/platform directory was modified to point to the correct compiler, etc... On MPP2, the following is the value of the NCBI_FLAGS1 variable: -c -O2 -tpp2 -ftz -ip -I/¡path to GA include¿ -I/¡path to ScalaBLAST home directory¿ -I/¡path to MPI include¿. Following is the value of the NCBO_OTHERLIBS variable: -L/¡path to GA libs¿ -lglobal -lma -larmci -llapack -lmpi -ltcgmsg-mpi -lpthread -lm -L/lib -L/¡path to mpi libs¿ -lguide. The same modifications had to be made to the corresponding values in the ncbi/make files makenet.unx makedemo.unx makeindx.unx and makeall.unx as well. To prevent multiple definitions of the pointer and boolean types from GA and from NCBI, the following changes had to be made to the GA include file matypes.h: the line $\#ifndef_CSO_NCBI_$ and a corresponding $\#endif$ flank the declaration of the Boolean and Pointer types. _CSO_NCBI_ is defined in the standard ScalaBLAST header file.

ScalaBLAST run-time environment On MPP2, the run-time and compile-time environments are set by the pnnl_env/1.11 module. Among other things, this sets LD_LIBRARY_PATH to point to the correct system, Quadrics and MPI libraries as well as scheduler (LSF) and resource manager (RMS)-specific libraries. For the normal NCBI toolkit to run, the /.ncbirc file must exist. On MPP2, this file contains directives pointing to default data and database locations. Data directory is for scoring matrices, etc.

```
[NCBI]
Data=/home/oehmen/GAMPI\_PREFETCH/PAR\_ncbi/data/

[BLAST]
BLASTDB=/home/oehmen/DB\_NEW/
BLASTMAT=/home/oehmen/GAMPI\_PREFETCH/PAR\_ncbi/data/
```

For this benchmark, databases, input and output files were located on the /home filesystem. The ScalaBLAST executable is copied to /scratch at run-time.

Internal ScalaBLAST run-time parameters Prefetching chunk size (CHUNK_SIZE) is set by a pair of $\#define$ directives in the standard ScalaBLAST header file, and in the /tools/readdb.h file to a value of 400000 on both MPP2 and Jacquard builds. In the current ScalaBLAST build, the size of the array holding partial query results is fixed at 40,000,000 int's on both MPP2 and Jacquard. The new sequence penalty (QUERY_PENALTY) is set in the ScalaBLAST driver to a value of 225 on both MPP2 and Jacquard. ScalaBLAST is currently hard-coded to expect exactly two processors per process group (ENCLOSURE_SIZE). This is optimal for both MPP2 and Jacquard in which dual processor nodes are present.

0.17.2 Data

Q4 Collected Data	Problem(0.16.1)	Problem(0.16.1)
source codes	yes	yes
build information	yes	yes
runtime information	yes	yes
total walltime	yes	yes

Table 27: Information collected during Q4 for both platforms and both problems w/ ScalaBLAST.

arch	procs	nq/p/min/mil	Time(m)	nq	problem
jacquard	4	3.751874872	21.45	88	ENV
jacquard	4	2.530568947	21.68	60	ENV
jacquard	16	2.817554891	8.033	99	99
jacquard	64	2.334546035	24.48	1000	1000
jacquard	320	1.900295609	361.82	60146	ENV
jacquard	320	1.872471727	361.9	59279	STP
MPP2	4	3.187685463	25.53	89	ENV
MPP2	500	2.522386515	108.4	37373	ENV
MPP2	1000	6.404846053	510.42	893677	STP
MPP2	1500	3.587774192	1097.43	1614513	ENV

Table 28: Q4 performace of ScalaBLAST on varying problem sizes, process counts

DRAFT

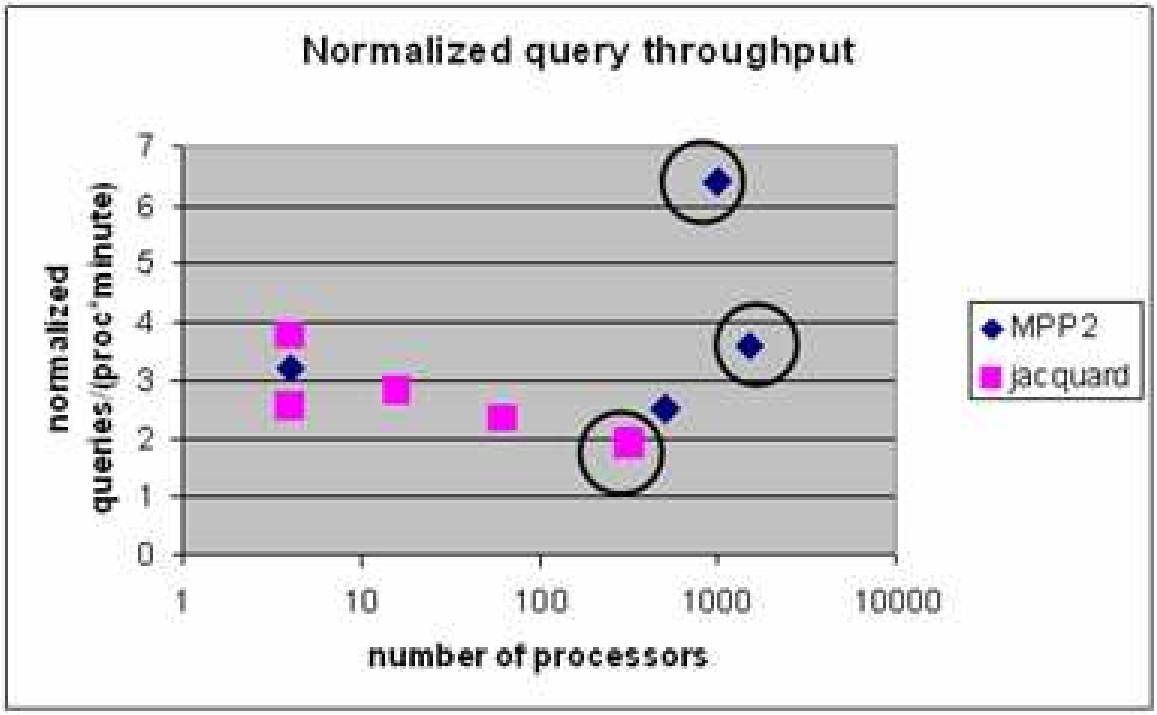

Figure 38. Normalized throughput of ScalaBLAST for a range of problem sizes and number of processors for both MPP2 and Jacquard. Both baseline problems were executed on both systems.

Note the results in table [28] show that ScalaBLAST throughput was consistently near or above 2 queries per processor per minute (normalized to 1 million proteins in the database for comparison to future benchmark numbers on larger databases). On jobs containing from 99 queries to 1.6 million proteins and from 4 processors to 1500 processors this level of throughput was achieved. The data from this table is illustrated in Fig[38], which shows that on Jacquard, ScalaBLAST demonstrated excellent scaling by maintaining per-processor throughput at a nearly constant level up to 64 processors. By 320 processors, the throughput is still good, but is beginning to decline over the 4 processor throughput without regard for output size (STP vs. ENV are identical). By contrast, ScalaBLAST running on MPP2 shows a virtually ideal (flat) per-processor throughput regardless of number of processors for the ENV benchmark, with a substantial gain in throughput when using recording only one top hit per query (STP benchmark). Large variability in run-time was observed on jacquard in conjunction with running with the database on the /u0 filesystem (better performance at 4 procs) vs. on the /projects filesystem (almost 1/3 drop in performance). Quotas enforced on /u0 prevented similar tests being performed with regard to where output was routed (in all cases, it was sent to /projects).

17. FY06 Improvement in Communication and I/O Overhead (ScalaBLAST) Improvements made in the area of high-throughput BLAST calculations in the course of this OMB software effectiveness study stem from effective partitioning of large input sets taking advantage of globally mounted filesystems, and effective memory

utilization. ScalaBLAST performance characteristics are illustrated in the following figures taken from NWPerf output for the 1000 processor MPP2 benchmark for the STP problem. Similar performance was observed for ScalaBLAST on the ENV benchmark. Figure[39] illustrates the low idle time which would accompany resource contention in the software-implemented shared memory model from Global Arrays on which ScalaBLAST is based. The correspondingly high user time illustrated in Figure[40] is indicative of the performance we observe for ScalaBLAST for a wide range of problems. The main reason for such high effectiveness is hiding latency from remote 'get' operations using prefetching available through Global Array nonblocking calls. Figure[41] shows that context switching quickly levels off during the calculation as processors contend less and less for off-processor sequence blocks even though only 1/1000 of the whole database is local to any given processor. This contention decreases through the run because of the variability in BLAST execution time for given queries. Since process groups are independent except for their sharing of the target database, randomness in run times spreads out the communication pattern intrinsically, increasing efficiency and scalability. This spread is also evident from Figure[42] which shows the pattern of processors writing to the Lustre file system. Even though each process group performs the same number of queries between file write operations, the time to execute each query block differs spreading out the overall I/O pattern, minimizing contention for I/O resources.

DRAFT

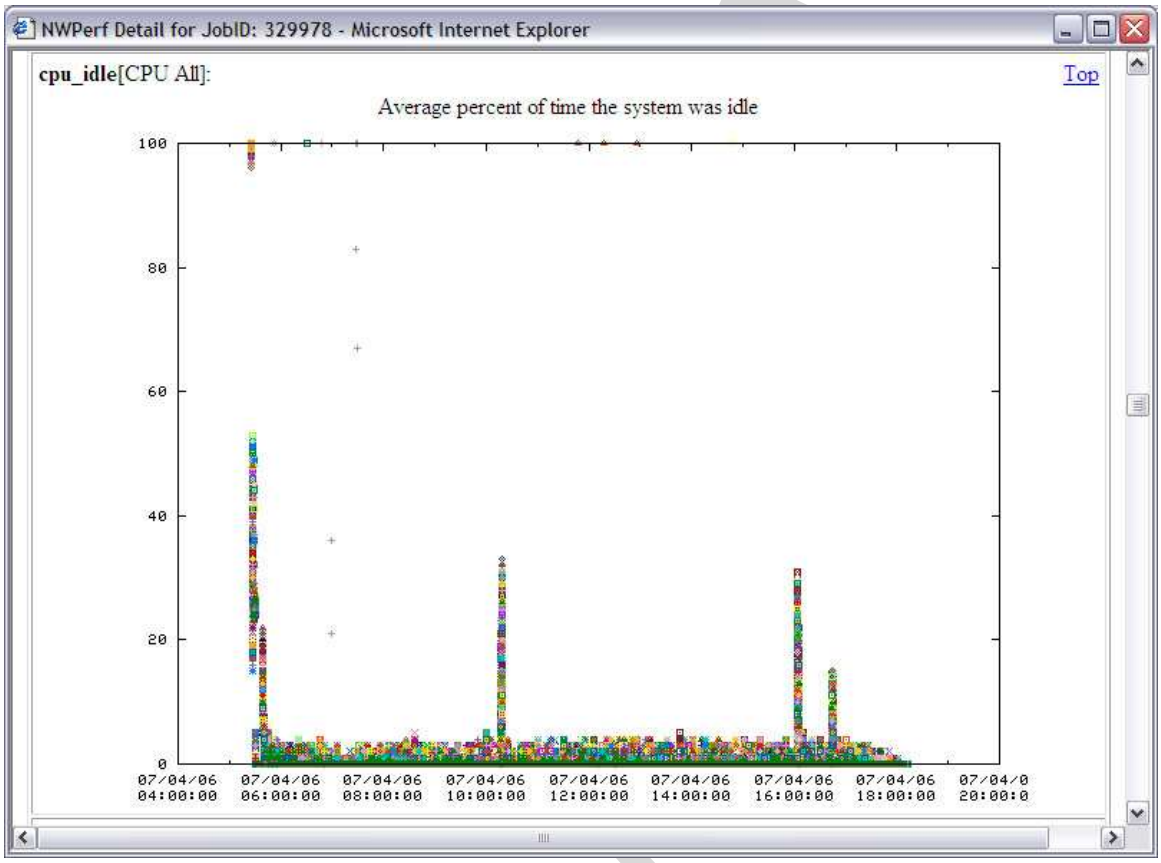

Figure 39. Idle time reported by NWPerf for STP benchmark for ScalaBLAST Almost no time is spent by any processor in the idle state.

DRAFT

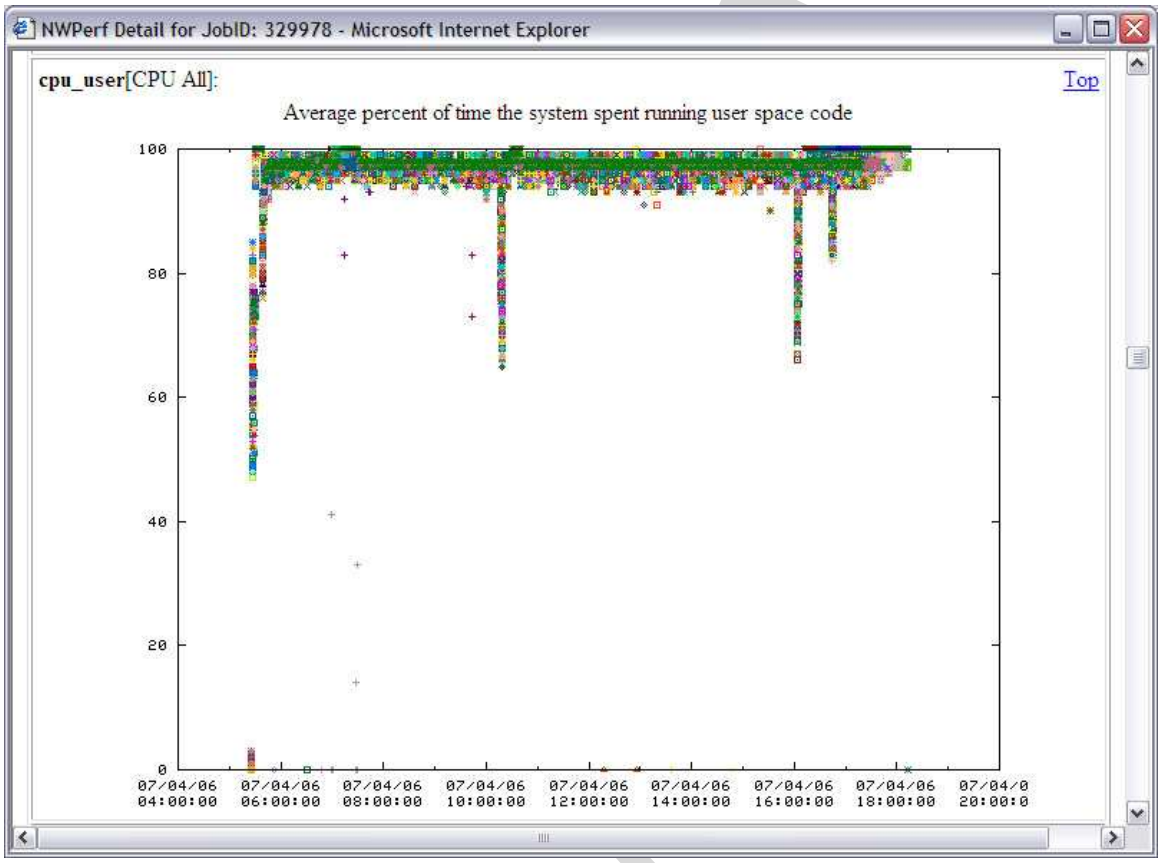

Figure 40. NWPerf results for STP benchmark ScalaBLAST run on 1000 processors showing virtually all the time spent by all processors in user space.

DRAFT

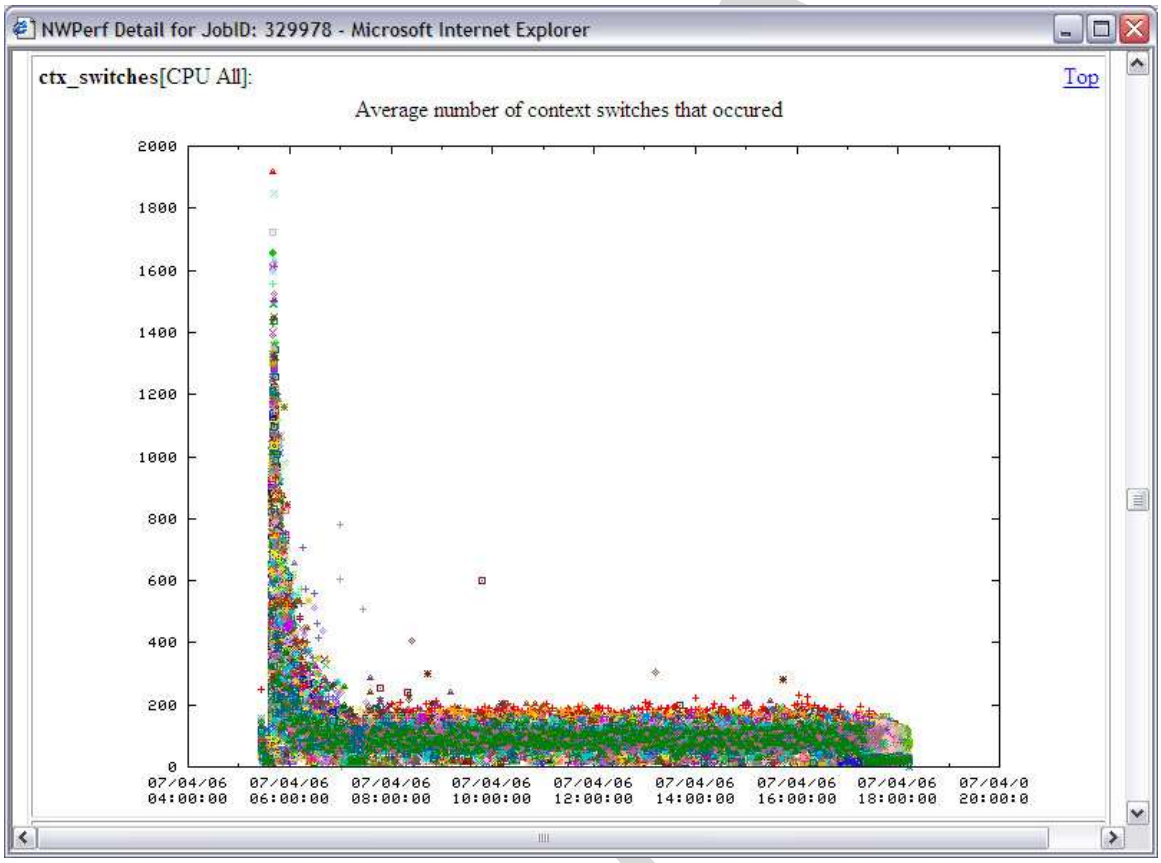

Figure 41. Context switching for the STP run of ScalaBLAST on 1000 processors dies out as the BLAST tasks become more out of sync on different processors. This eases resource contention for I/O and memory access.

DRAFT

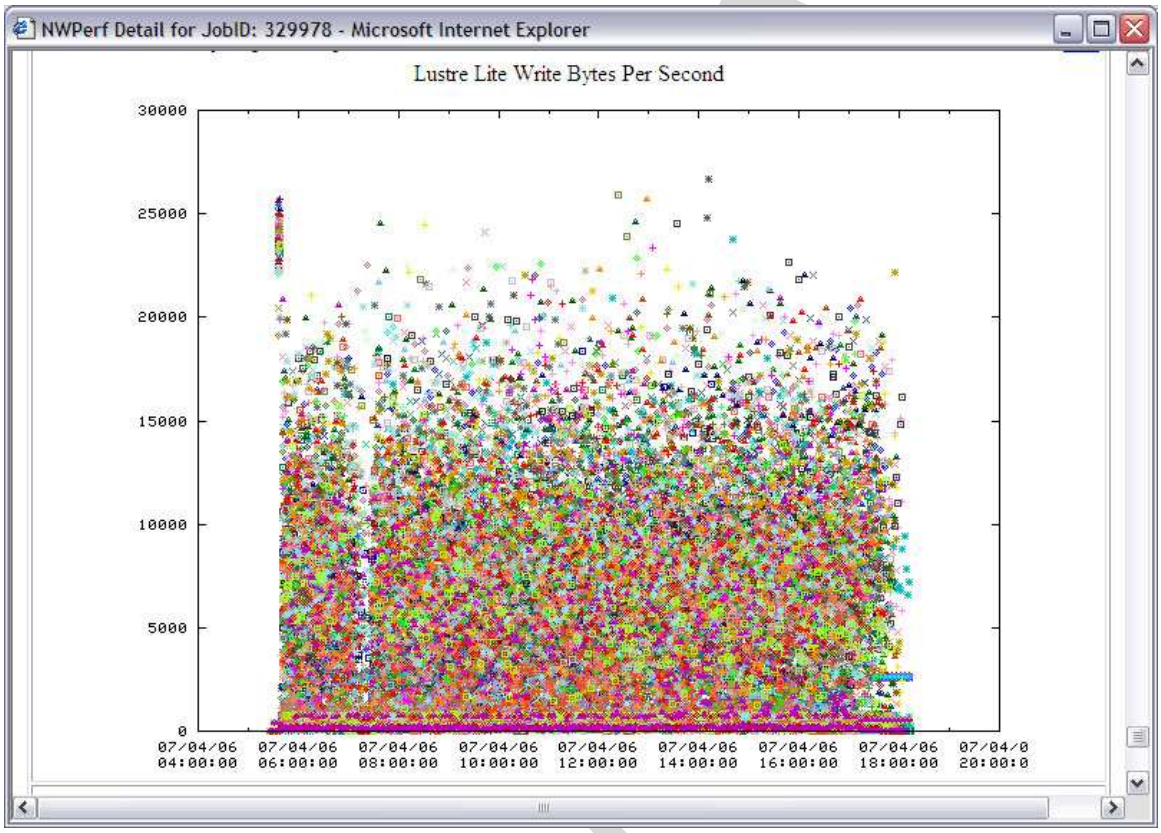

Figure 42. NWPerf results on MPP2 for ScalaBLAST STP benchmark on 1000 processors. After startup, write tasks to the global filesystem are evenly distributed across the processors.

A second tier of concurrency is exploited by iterating over separate halves of a database with the processors within a process group. This is done many times followed by storage of partial results in a process-group wide global array object before any of the results are merged and printed to the output file. Since each processor can proceed on its own with these half-database calculations, synchronization is only enforced after all iterations have been performed by both processors in the group. The consequence of this is that there is little blocking within a process group for synchrony or resource contention. The processors in the process group are proceeding with independent calculations requiring only sparse synchronization. Fig[43] illustrates the low likelihood of multiple processes blocked on system calls. The vast majority of datapoints lie on the value 0-the mean for this collection of data is 0.025 +/- 0.2.

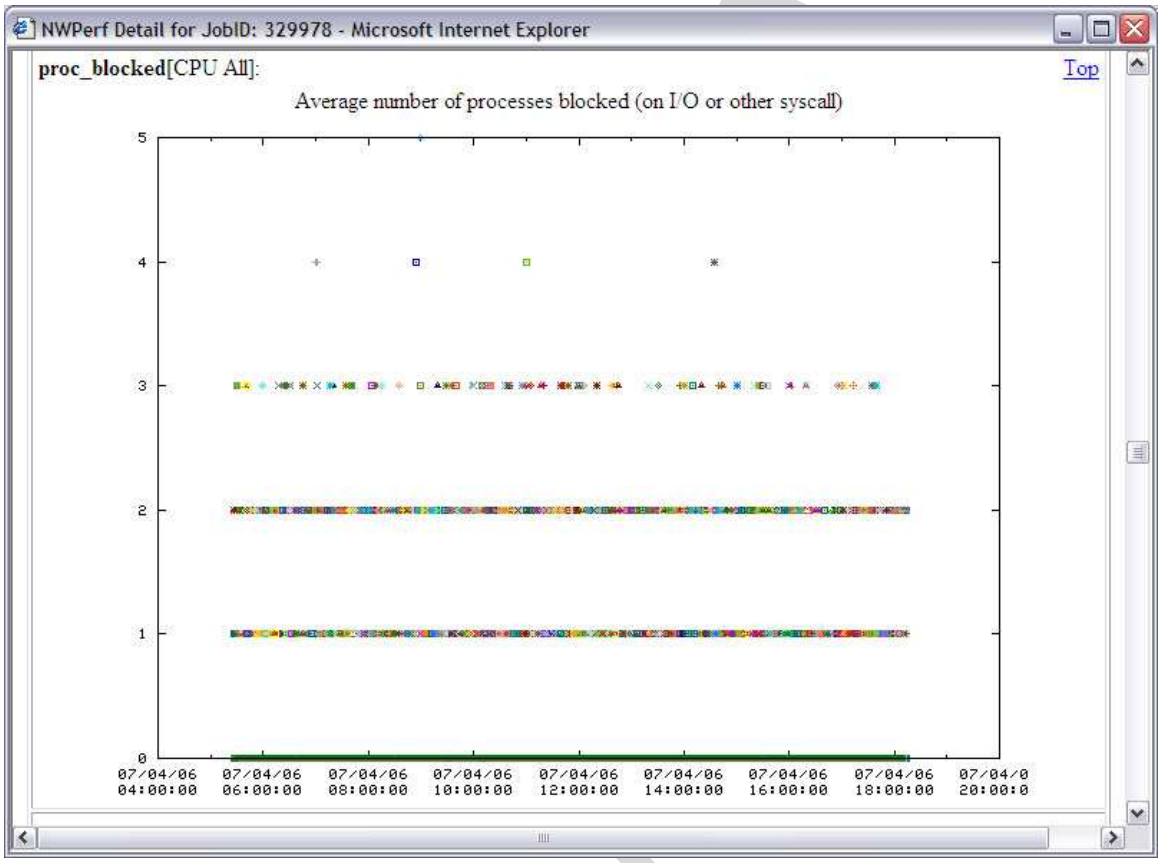

Figure 43. Blocking on I/O or other system calls are rare in the ScalaBLAST STP benchmark on MPP2 using 1000 processors. NWPerf output shows most of the data at a value of 0.

FY06 Improvement in Memory Utilization (ScalaBLAST) ScalaBLAST, based on the NCBI BLAST core algorithm, is not floating-point intensive. Figure[44] illustrates the flop rate of the high-throughput operation of ScalaBLAST at around 0.1The primary limitation of ScalaBLAST performance is memory-bandwith availability. Figure[45] illustrates the stall rate of the code with two key features: 1) all processors are equally likely to be stalled (rather than the high stall rate of the master node observed in the MPI BLAST benchmarks), and 2) load imbalance results in a higher stall rate at the end of the calculation as some process groups finish their tasks before others. Item 2 could be addressed using dynamic scheduling of the queries, which is an item for future development in ScalaBLAST. However, item 1 will remain a challenge since register-register stalls (of which the majority shown in Figure[45] are) may relate to dependency in computations being done by ScalaBLAST or the BLAST core itself. Given the weakness of many systems in terms of balance between peak flop rate and memory bandwidth and memory latency, ScalaBLAST was designed to utilize memory as efficiently as possible. Figure[46] illustrates the changing footprint of ScalaBLAST during the STP benchmark run. This figure clearly shows that all processors are operating well within the limits of the memory available on

MPP2 (8 GB per node on fatnodes and 6 GB per node on thin nodes).

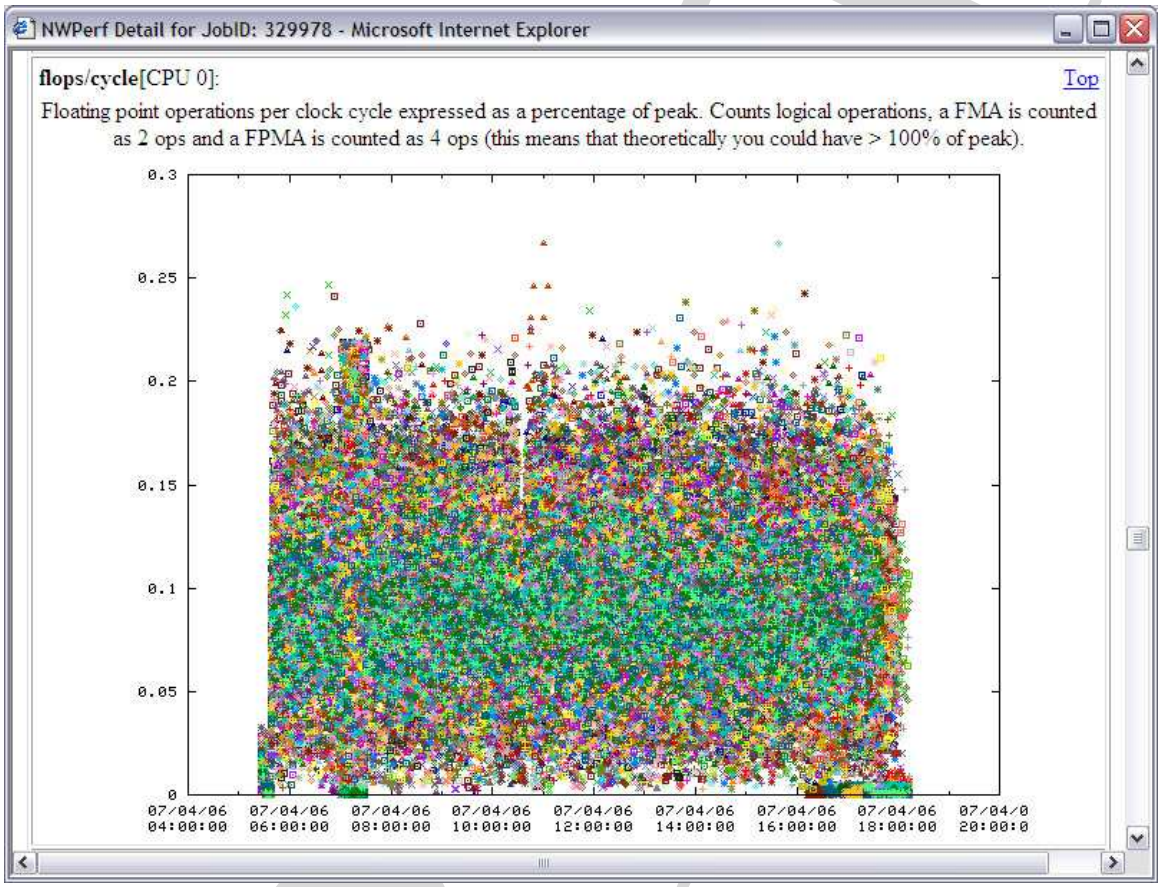

Figure 44. ScalaBLAST is not floating-point operation intensive with a consistent flop rate of below 0.25% of peak on MPP2, as revealed by NWPerf results for the STP benchmark on 1000 processors.

DRAFT

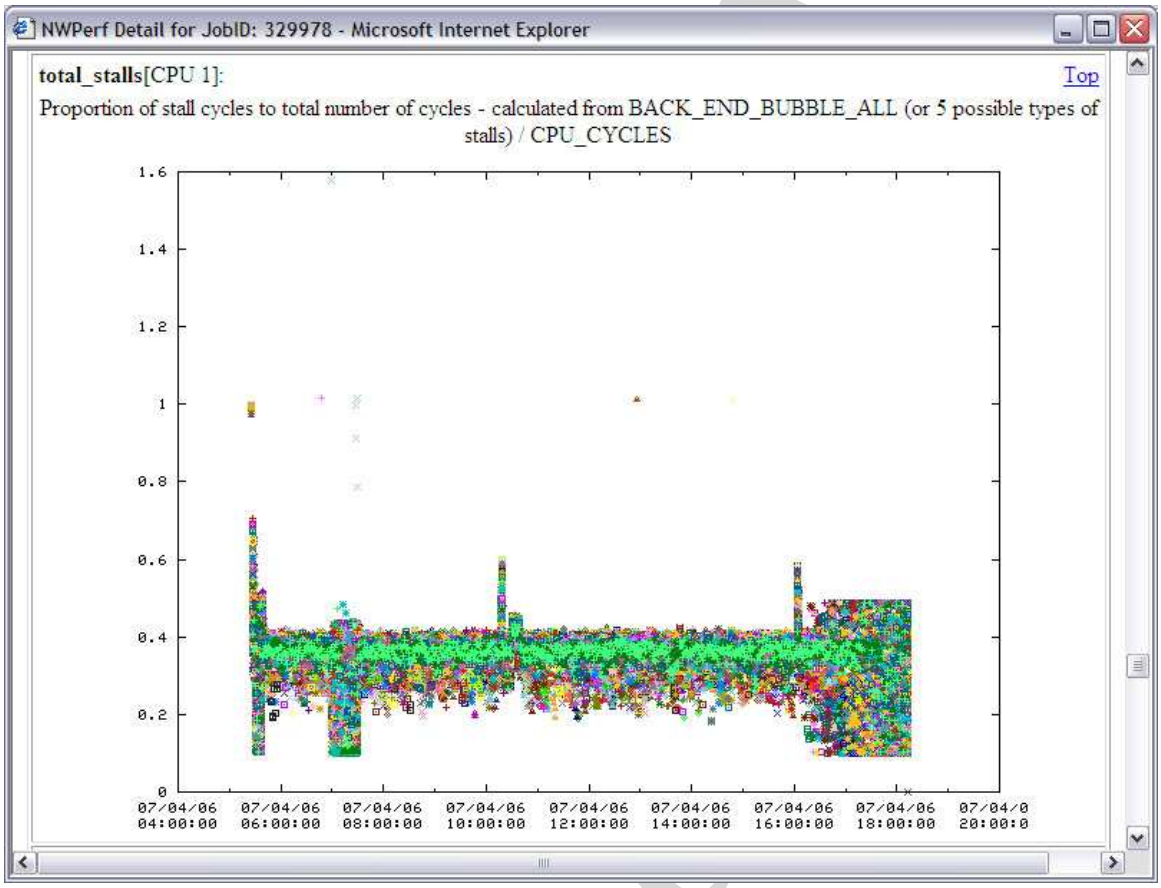

Figure 45. Stalls in the memory bus demonstrate that ScalaBLAST is memory-bandwidth limited even while producing high-throughput BLAST output. Rise in stalls near the end of the calculation marks the load-balancing limitation of static partitioning of the query list.

DRAFT

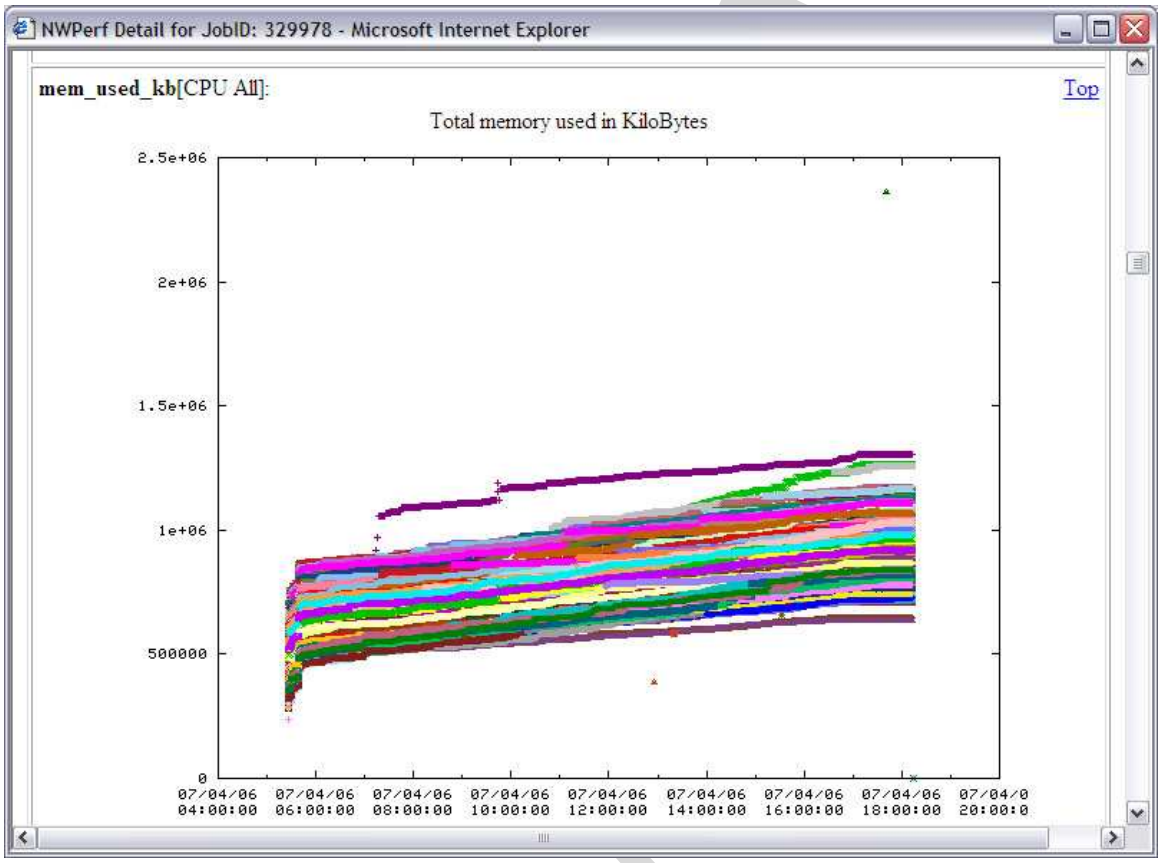

Figure 46. Global arrays help make memory management effective after startup. Jumps in memory usage generally coincide with partial results buffers being expanded in ScalaBLAST, as revealed by NWPerf output from MPP2.

Figure[47] demonstrates that no processors participate in swapping during the run, ensuring that limitaitons associated with swapping do not hinder ScalaBLAST operation or scalability.

DRAFT

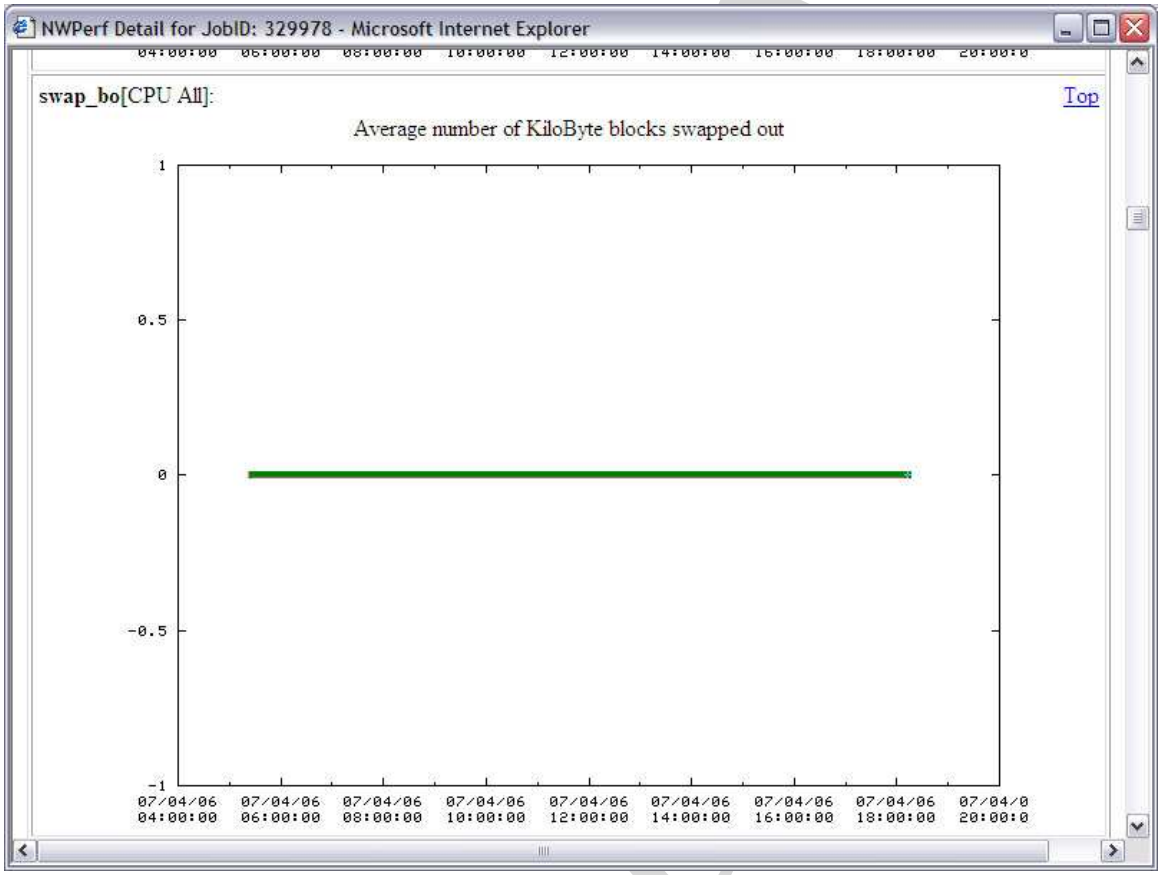

Figure 47. NWPerf results reveal that ScalaBLAST completely avoids swapping during the entire run on all processors. This output contains data from all 1000 processors for the entire STP benchmark run on MPP2.

0.18 Results

To overcome the limitations of MPI BLAST with respect to memory usage and scalability, ScalaBLAST was designed with an alternative approach, meant to maximize parallel I/O, memory bandwidth, and scalability as demonstrated in the FY06 improvement results, shown in the following sections.

New Result Compared with the failure of MPI BLAST to produce any output for either of the benchmark runs on both Jacquard and MPP2, ScalaBLAST succeeded in delivering a high-throughput of BLAST results per-processor on both platforms and for both benchmarks essentially without regard for number of processors involved in the calculation-the same as ideal scaling for both problems, as illustrated in Table [28].

Problem	System	Enhancement
(sec0.16.1)	MPP2, *mpp2.emsl.pnl.gov*	**new result**, see(**??**)

Table 29: FY06 Joule performance enhancement for the ScalaBLAST code. The science driver for this application is reduced to...

DRAFT

Bibliography

[1] M. H. Hettler, A. N. Tahvildar-Zadeh, M. Jarrell, T. Pruschke, and H. R. Krishnamurthy. Nonlocal dynamical correlations of strongly interacting electron systems. *Phys. Rev. B*, 58:R7475–R7479, 1998.

[2] T. A. Maier, M. Jarrell, and D. J. Scalapino, The Pairing Interaction in the 2D Hubbard Model. *Phys. Rev. B*, **74**, 2006.

[3] M. H. Hettler, M. Mukherjee, M. Jarrell, and H. R. Krishnamurthy. Dynamical cluster approximation: Nonlocal dynamics of correlated electron systems. *Phys. Rev. B*, 61:12739–12756, 2000.

[4] M. Jarrell, Th. Maier, M. H. Hettler, and A. N. Tahvildarzadeh. Phase diagram of the hubbard model: Beyond the dynamical mean field. *Europhys. Lett.*, 56:563, 2001.

[5] Th. Maier, M. Jarrell, Th. Pruschke, and J. Keller. A non-crossing approximation for the study of intersite correlations. *Eur. Phys. J B*, 13:613, 2000.

[6] Th. Maier, M. Jarrell, Th. Pruschke, and J. Keller. d-wave superconductivity in the hubbard model. *Phys. Rev. Lett.*, 85:1524, 2000.

[7] Th. Maier, M. Jarrell, Th. Pruschke, and M.H. Hettler. Quantum cluster theories. *preprint cond-mat/0404055*, 2004.

[8] M. Jarrell, Th. Maier, C. Huscroft, and S. Moukouri. A quantum monte carlo algorithm for non-local corrections to the dynamical mean-field approximation. *Phys. Rev. B*, 64:195130, 2001.

[9] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16:1–17, 1990.

[10] N.V. Prokof'ev and B.V. Svistunov. Polaron problem by diagrammatic quantum monte carlo. *Phys. Rev. Lett.*, 81:2514, 1998.

[11] A.L. Fetter and J.D. Walecka. *Quantum Theory of Many-particle Systems.* McGraw-Hill, New York, 1971.

[12] A.N. Rubtsov, V.V. Savkin, and A.I. Lichtenstein. Continuous time quantum monte carlo method for fermions. *preprint cond-mat/0411344*, 2004.

DRAFT

[13] E.A. Bourovski, N.V. Prokof'ev, and B.V. Svistunov. Truncated-determinant diagrammatic monte carlo for fermions with contact interaction. *Phys. Rev. B*, 70:193101, 2004.

[14] D.J. Scalapino and R.L. Sugar. Method for performing monte carlo calculations for systems with fermions. *Phys. Rev. Lett.*, 24:519–521, 1981.

[15] R. Blankenbecler, D.J. Scalapino, and R.L. Sugar. Monte carlo calculations of coupled boson-fermion systems. *Phys. Rev. D*, 24:2278–2286, 1981.

[16] J.E. Hirsch. Discrete hubbard-stratonovich transformation for fermion lattice models. *Phys. Rev. B*, 28(4059), 1983.

[17] N. Prokof'ev and B. Svistunov. Worm algorithms for classical statistical models. *Phys. Rev. Lett.*, 87(160601), 2001.

[18] A. Bulgac, J. E. Drut, and P. Magierski. Spin 1/2 fermions on a 3d-lattice in the unitary regime at finite temperatures. *preprint cond-mat/0505374*, 2005.

[19] J. Carlson, S.-Y. Chang, V. R. Pandharipande, and K. E. Schmidt. Superfluid fermi gases with large scattering length. *Phys. Rev. Lett.*, 91:050401, 2003.

[20] T.A. Maier, M. Jarrell, T.C. Schulthess, P.R.C. Kent, and J.B. White. A systematic study of superconductivity in the 2d hubbard model. *preprint cond-mat/0504529*, 2005.

[21] D.J. Scalapino. Superconductivity and spin fluctuations. *J. Low Temp. Phys.*, 117(3-4):179–188, 1999.

[22] N.E. Bickers, D.J. Scalapino, and S.R. White. Conserving approximations for strongly correlated electron systems: Bethe-salpeter equation and dynamics for the two-dimensional hubbard model. *Phys. Rev. Lett.*, 62:961, 1989.

[23] News. So much more to know ... *Science*, 309(5731):78–102, 2005.

[24] E. Dagotto. Correlated electrons in high-temperature superconductors. *Rev. Mod. Phys*, 66:763, 1994.

[25] S.R. White, D.J. Scalapino, R.L. Sugar, E.Y. Loh, J.E. Gubernatis, and R.T. Scalettar. Numerical study of the two-dimensional hubbard model. *Phys. Rev. B*, 64:195130, 1989.

[26] J.R. Anglin and W. Ketterle. Bose-einstein condensation of atomic gases. *Nature*, 416:211, 2002.

[27] D. Jaksch et al. Cold bosonic atoms in optical lattices. *Phys. Rev. Lett.*, 81:3108, 1998.

[28] T. Müller-Seydlitz et al. Atoms in the lowest motional band of a three-dimensional optical lattice. *Phys. Rev. Lett.*, 78:1038, 1997.

DRAFT

[29] Friebel et al. Co_2-laser optical lattice with cold rubidium atoms. *Phys. Rev. A*, 57:R20, 1998.

[30] G. Raithel et al. Cooling and localization dynamics in optical lattices. *Phys. Rev. Lett.*, 78:630, 1997.

[31] W. Hofstetter et al. High-temperature superfluidity of fermionic atoms in optical lattices. *Phys. Rev. Lett.*, 89:220407, 2002.

[32] S. Jochim et al. Bose-einstein condensation of molecules. *Science*, 302:2101, 2003.

[33] M. Greiner, C. Regal, and D. S. Jin. Emergence of a molecular bose-einstein condensate from a fermi gas. *Nature*, 426:537, 2003.

[34] M. Zwierlein et al. Observation of bose-einstein condensation of molecules. *Phys. Rev. Lett.*, 91:250401, 2003.

[35] C.A. Regal et al. Observation of resonance condensation of fermionic atom pairs. *Phys. Rev. Lett.*, 92:040403, 2004.

[36] Y. Ohashi and A. Griffin. Bcs-bec crossover in a gas of fermi atoms with a feshbach resonance. *Phys. Rev. Lett.*, 89:130402, 2002.

[37] M. Drechsler and W. Zwerger. Crossover from bcs-superconductivity to bose-condensation. *Ann. Phys.*, 1:15, 1992.

[38] P. Noziéres and S. Schmitt-Rink. Bose condensation in an attractive fermion gas: From weak to strong coupling superconductivity. *J. Low Temp. Phys.*, 59:195, 1985.

[39] A.J. Leggett. Cooper pairing in spin-polarized fermi systems. *J. Physique. C*, 41:7, 1980.

[40] M. Greiner et al. Quantum phase transition from a superfluid to a mott insulator in a gas of ultracold atoms. *Nature*, 415:39, 1998.

[41] L. Pezzé et al. Insulating behavior of a trapped ideal fermi gas. *preprint cond-mat/0401643*, 2004.

[42] H. Ott. Collisionally induced transport in periodic potentials. *Phys. Rev. Lett.*, 92:160601, 2004.

[43] G. Modugno et al. Production of a fermi gas of atoms in an optical lattice. *Phys. Rev. A*, 68:011601, 2003.

[44] A. Sewer, X. Zotos, and H. Beck. Quantum monte carlo study of the three-dimensional attractive hubbard model. *Phys. Rev. B*, 66:140504(R), 2002.

[45] M.W. Zwierlein, J.R. Abo-Shaeer, A. Schirotzek, C.H. Schunck, and W. Ketterle. Vortices and superfluidity in a strongly interacting fermi gas. *Nature*, 435:1047, 2005.

DRAFT

[46] A. Bulgac, J.E. Drut, and P. Magierski. Spin 1/2 fermions on a 3d-lattice in the unitary regime at finite temperatures. *preprint cond-mat/0505374*, 2005.

[47] S.M.A. Rombouts, K. Heide, and N. Jachowicz. Quantum monte carlo method for fermions, free of discretization errors. *Phys. Rev. Lett.*, 82:4155–4159, 1999.

[48] R. Staudt, M. Dzierzawa, and A. Muramatsu. Phase diagram of the three-dimensional hubbard model at half filling. *Eur. Phys. J. B*, 17:411–415, 2000.

[49] P. Anninos, Y. Zhang, T. Abel & M. Norman, "Cosmological Hydrodynamics with Multi-species Chemistry and Nonequilibrium Ionization and Cooling", New Astron. **2**, pp. 209-224, (1997).

[50] M.J. Berger and P. Colella, "Local Adaptive Mesh Refinement for Shock Hydrodynamics," J. Comput. Phys., **82**, pp. 64-84, (1989).

[51] G.L. Bryan, M.L. Norman, J.M. Stone, R. Cen, and J.P. Ostriker, "A Piecewise Parabolic Method for Cosmological Hydrodynamics", Computer Physics Communication, **89**, pp. 149-168, (1995).

[52] G.L. Bryan and M.L. Norman, "A Hybrid AMR Application for Cosmology and Astrophysics," in IMA Volume 117 on *Structured Adaptive Mesh Refinement (SAMR) Grid Methods*, eds. S.B. Baden, N.P. Chrisochoides, D. Gannon, and M.L. Norman, (Springer: New York), pp. 165-170 (2000).

[53] G. L. Bryan and M. L. Norman, "Statistical Properties of X-Ray Clusters: Analytic and Numerical Comparisons", Astrophys. J., **495**, pp. 80-90, (1998).

[54] G.L. Bryan "Fluids in the Universe: Adaptive Mesh Refinement in Cosmology", *Computing in Science and Engineering*, **1:2**, 46 (1999).

[55] G. Bryan, T. Abel & M. Norman, "Achieving Extreme Resolution in Numerical Cosmology Using Adaptive Mesh Refinement: Resolving Primordial Star Formation", submitted to *Proceedings of Supercomputing 2001*, `www.supercomp.org`

[56] R. Ferrell & E. Bertschinger, "Particle-Mesh Methods on the Connection Machine", Int. J. Mod. Phys. C **5**, pp. 933-956 (1994).

[57] W. D. Hillis & G. L. Steele, Jr., "Data Parallel Algorithms", Commun. ACM **29**, 12 (1986).

[58] R. W. Hockney & J. W. Eastwood, *Computer Simulation using Particles*, (McGraw-Hill, New York, 1988).

[59] T. Jena, M. L. Norman, D. Tytler, D. Kirkman, N. Suzuki, A. Chapman, C. Melis, P. Paschos, B. O'Shea, G. So, D. Lubin, W.-C. Lin, D. Reimers, E. Janknecht, and C. Fechner. A concordance model of the Lyman α forest at z= 1.95. *Monthly Notices of the Royal Astronomical Society*, 361:70–96, July 2005.

DRAFT

[60] N. Katz, D. Weinberg, & L. Hernquist, "Cosmological Simulations with TreeSPH", *Astrophysical J. Supp.*, 105, 19 (1996).

[61] Z. Lan, V. Taylor & G. Bryan, "Characterization of a Parallel Adaptive Mesh Refinement Application", in *Proceedings of Supercomputing2000*, `www.supercomp.org`.

[62] Z. Lan, V. Taylor & G. Bryan, "Dynamic Load Balancing for Adaptive Mesh Refinement Applications", in *Proceedings of the International Parallel and Distributed Processing Symposium 2001.*

[63] M.L. Norman and G.L. Bryan, "Cosmological Adaptive Mesh Refinement," in *Numerical Astrophysics*, eds. S. Miyama & K. Tomisaka, (Kluwer: Dordrecht, 1999), pp. 19-28.

[64] A. Darling and L. Carey and W.-C. Feng, "The design, implementation, and evaluation of mpiBLAST," in *Proceedings of ClusterWorld*, San Jose, CA. (2003).

[65] S. Needleman and C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins.," J. Mol. Biol., vol. 48, pp. 443-453 (1970).

[66] T. Smith and M. Waterman, "Overlapping genes and information theory," J. Theor. Biol., vol. 91, pp. 379-380 (1981).

[67] T. Smith, M. Waterman, and W. Fitch, "Comparative biosequence metrics," J. mol. Evol., vol. 18, pp. 38-46 (1981).

[68] W. Pearson and D. Lipman, "Improved tools for biological sequence comparison," Proc Natl Acad Sci U.S.A., vol. 85, pp. 2444-2448 (1988).

[69] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," J. Mol. Biol., vol. 215, pp. 403-410 (1990).

[70] S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs," Nucleic Acids Res., vol. 25, pp. 3389-3402 (1997).

[71] S. Brenner, C. Chothia, and T. J. P. Hubbard, "Assessing sequence comparison methods with reliable structurally identified distant evolutionary relationships," Proc. Natl. Acad. Sci. USA, vol. 95, pp. 6073-6078 (1998).

[72] B. Webb, J. Liu, and C. Lawrence, "BALSA: Bayesian algorithm for local sequence alignment," Nucleic Acids Res., vol. 2002, pp. 1268-1277 (2002).

[73] X. Meng and V. Chaudhary, "Bio-sequence analysis with cradle's 3SoC software scalable system on chip," presented at ACM symposium on applied computing, Nicosia, Cyprus (2004).

DRAFT

[74] K. Muriki, K. Underwood, and R. Sass, "RC-BLAST: Towards a portable, cost-effective open source hardware implementation," presented at HICOMB 2005, 4th IEEE international workshop on high performance computational biology, Denver, CO (2005).

[75] R. Bjornson, A. Sherman, S. Weston, N. Willard, and J. Wing, "TurboBLAST(r): A parallel implementation of BLAST built on the TurboHub.," presented at 16th International Parallel and Distributed Processing Symposium (IPDPS), Fort Lauderdale, FL (2002).

[76] T. Braun, T. Scheetz, G. Webster, A. Clark, E. Stone, V. Sheffield, and T. Casavant, "Identifying candidate disease genes with high-performance computing," J. Supercomputing, vol. 26, pp. 7-24 (2003).

[77] J. Wang and Q. Mu, "Soap-HT-BLAST: high-throughput BLAST based on web services," Bioinformatics, vol. 19, pp. 1863-1864 (2003).

[78] N. Camp, H. Cofer, and R. Gomperts, "High-throughput BLAST," (1998).

[79] H. Nicholas, G. Giras, V. Hartonas-Garmhausen, M. Kopko, C. Maher, and A. Ropelewski, "Distributing the comparison of dna and protein sequences across heterogeneous supercomputers," presented at ACM/IEEE conference on supercomputing, Albuquerque, NM (1991).

[80] T. Rognes, "ParAlign: a parallel sequence alignment algorithm for rapid and sensitive database searches," Nucleic Acids Res, vol. 29, pp. 1647-1652 (2001).

[81] X. Cao, S. C. Li, B. C. Ooi, and A. K. H. Tung, "Piers: An efficient model for similarity search in DNA sequence databases," SIGMOD record, vol. 33, pp. 39-44 (2004).

[82] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova, "Efficient data access for parallel BLAST," presented at 19th International Parallel and Distributed Processing Symposium (IPDPS), Denver, CO (2005).

[83] C. Oehmen and J. Nieplocha, "ScalaBLAST: A scalable implementation of BLAST for High Performance Data-Intensive Bioinformatics Analysis," IEEE Trans. Parallel. Dist. Sys., Special issue on high performance computational biology, vol. in press (2006).

[84] J. Nieplocha, R. Harrison, and R. Littlefield, "Global Arrays: A nonuniform memory access programming model for high-performance computers," J. Supercomputing, vol. 10, pp. 197-220 (1996).

[85] J. Nieplocha, J. Ju, and T. Straatsma, "A multiprotocol communication support for the global address space programming model on the IBM SP," presented at Euro-Par (2000).

[86] J. Nieplocha, M. Krishnan, B. Palmer, V. Tipparaju, and Y. Zhang, "Exploiting processor groups to extend scalability of the GA shared memory programming model," presented at ACM SIGMicro Computing Frontiers (2005).

DRAFT

[87] Bateman A, Coin L, Durbin R, Finn RD, Hollich V, Griffiths-Jones S, Khanna A, Marshall M, Moxon S, Sonnhammer EL, Studholme DJ, Yeats C, and E. SR., "The Pfam protein families database," Nucleic Acids Res, vol. 32, pp. D138-D141 (2004).

[88] J. Venter, K. Remington, J. Heidelberg, A. Halpern, D. Rusch, J. Eisen, D. Wu, I. Paulsen, K. Nelson, W. Nelson, D. Fouts, S. Levy, A. Knap, M. Lomas, K. Nealson, O. White, J. Peterson, J. Hoffman, R. Parsons, H. Baden-Tillson, C. Pfannkoch, Y.-H. Rogers, and H. Smith, "Environmental Genome Shotgun Sequencing of the Sargasso Sea," Science, vol. 304, pp. 66-74 (2004).

[89] M. Salisbury, "Parallel Blast: Chopping the database," in Genome Technology, pp. 21-22 (2005).

DRAFT