Optimizing Nuclear Physics Codes on the XT5

Rebecca J. Hartman–Baker and Hai Ah Nam*

Oak Ridge Leadership Computing Facility
Oak Ridge National Laboratory
P.O. Box 2008, MS–6008
Oak Ridge, Tennessee 37831-6008

ABSTRACT: Scientists studying the structure and behavior of the atomic nucleus require immense high-performance computing resources to gain scientific insights. Several nuclear physics codes are capable of scaling to more than 100,000 cores on Oak Ridge National Laboratory's petaflop Cray XT5 system, Jaguar. In this paper, we present our work on optimizing codes in the nuclear physics domain.
KEYWORDS: code optimization, nuclear physics, XT5

1 Introduction

Nuclear physicists have been consistent users of leadership high-performance computing resources over the last decade, with several nuclear physics codes capable of scaling to more than 100,000 cores on Oak Ridge National Laboratory's petaflop Cray XT5 system, Jaguar. Under the Universal Nuclear Energy Density Functional (UNEDF) SciDAC collaboration [2], nuclear physicists, working with applied mathematicians, computer scientists, and computational scientists, have developed and scaled their computational tools on leadership-class systems. Through the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program, [1] nuclear physics research has received sizable allocations on leadership class computing resources and earned early access to Jaguar with a Petascale Early Science project.

At the frontier of discovery, nuclear physics theory aims to explain the nature of nuclei, an understanding of which is fundamental to energy, medical, and biological research, and national security. The constituent nuclear particles, protons and neutrons, interact through the very complex strong nuclear force, which is governed by quantum chromodynamics (QCD), the theory describing quarks, gluons, and the strong force. Theorists are working toward a fundamental, unified, predictive description of nuclei based on the underlying theory of QCD.

To accomplish these objectives entails considering nuclei that span the entire chart of nuclides, and requires the use of a variety of theoretical and computational methods, such as those shown in Figure 1. Central to these theoretical approaches is solving the nuclear quantum many-body problem, Schrödinger's equation, for many interacting particles. Schrödinger's equation for an A-body system is given by

$$H\Psi(\mathbf{r}_1, \mathbf{r}_2, ..., \mathbf{r}_A) = \lambda\Psi(\mathbf{r}_1, \mathbf{r}_2, ..., \mathbf{r}_A) \qquad (1)$$

where H is the many-body Hamiltonian constructed from a given interaction and kinetic energy of the system, and Ψ is the many-body wavefunction with corresponding eigenvalue λ. The complexity of solving Schrödinger's equation increases exponentially with increasing numbers of particles and states that the system can access. There are only a few analytically solvable problems, and virtually exact numerical solutions are available for systems with at most three or four particles. To address systems with more particles requires the development of stable algorithms and reliable numerical many-body approximation methods, and high-performance computing resources upon which to perform the computations.

Each of the computational approaches is best applied to a specific mass region, as shown in Figure 1. *Ab initio* methods, including Green's Function Monte Carlo (GFMC), No-Core Shell Model (NCSM) and coupled-cluster (CC) methods, are

[1] `http://www.doeleadershipcomputing.org/incite-program/`

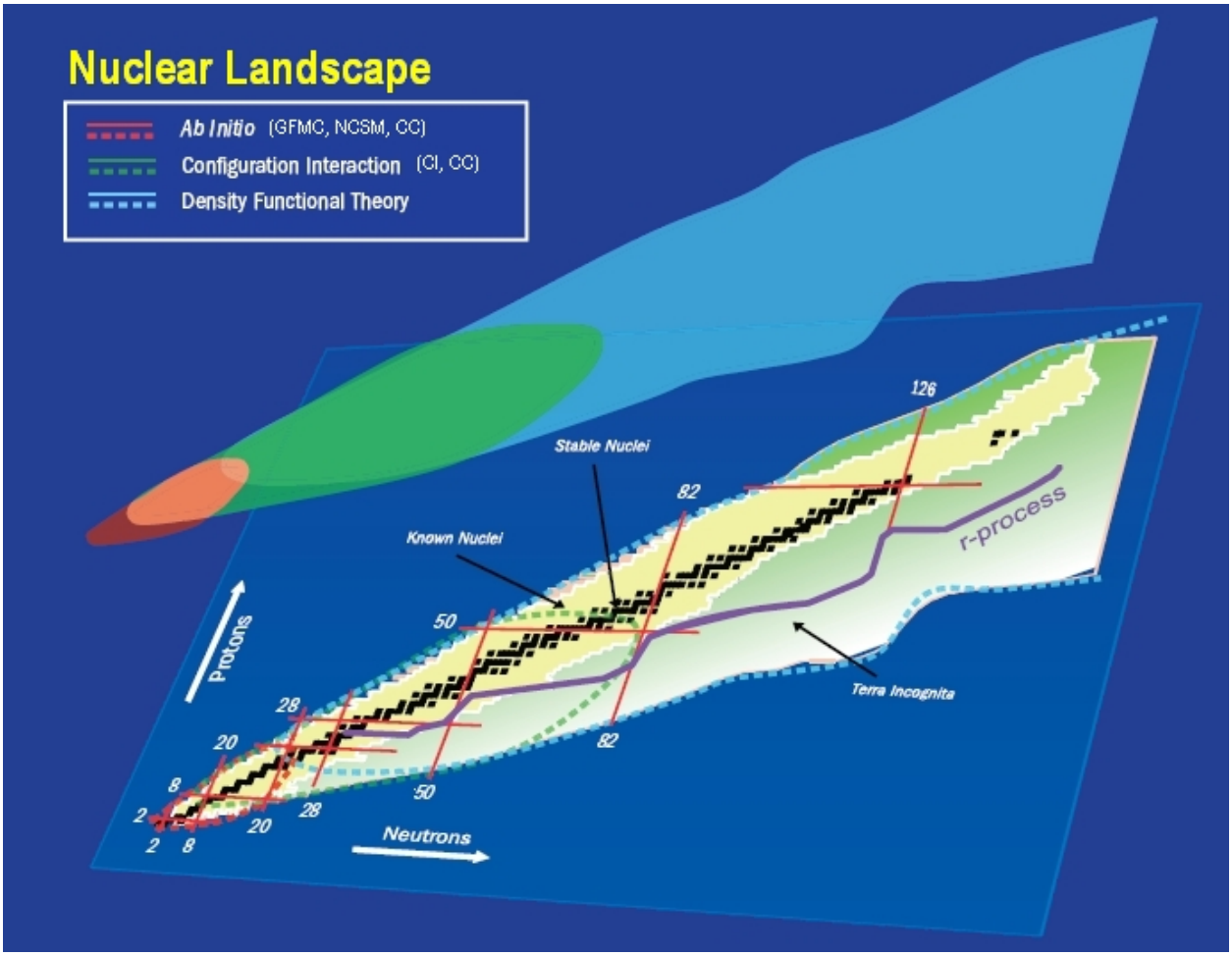


Figure 1: Theoretical methods and computational techniques to solve the nuclear many-body problem. The red vertical and horizontal lines show the magic numbers, reflecting regions where nuclei are expected to be more tightly bound and have longer half-lives. The anticipated path of the astrophysical r-process responsible for nucleosynthesis of heavy elements is also shown (purple line). The thick dotted lines indicate domains of major theoretical approaches to the nuclear many-body problem. For the lightest nuclei, *ab initio* calculations (GFMC, NCSM and the CC method), based on the bare nucleon-nucleon and three-nucleon interactions, are possible (red). Medium-mass nuclei can be treated by configuration interaction (CI) and coupled-cluster (CC) techniques (green). For heavy nuclei, the density functional theory, based on self-consistent/mean-field theory (blue), is the tool of choice. By investigating the intersections and overlaps of these regions we aim to establish a robust theory with high-quality predictive power and to solve the aforementioned physics problems. Adapted from Ref. [5].

Nucleus	Four major shells	Seven major shells
^4He	4×10^4	9×10^6
^8Be	4×10^8	4×10^{13}
^{12}C	6×10^{11}	4×10^{19}
^{16}O	3×10^{14}	4×10^{24}

Table 1: Examples of shell model dimensions for selected nuclei and given number of major shells. Taken from [6].

practical for light nuclei. Medium-mass nuclei are studied using configuration interaction (CI) or CC, and heavy nuclei using density functional theory (DFT). The algorithms underlying these methods employ a broad range of mathematical methods, from linear algebra to Monte Carlo.

Solving the nuclear many-body problem requires immense high-performance computing resources because of the enormity of the problem size. Model-space dimensions for configuration interaction, commonly known as shell model, calculations are given in Table 1. Shells are defined by harmonic oscillator single-particle orbits and shell model dimensions are given by all allowed single-particle product states for a given number of shells and a given nucleus. Current shell model applications can reach dimensions of 10^{10}.

To efficiently use allocated and limited computing hours and further scale applications to address larger problem sizes, domain scientists must augment with their daily research activities with code optimization. This paper describes profiling studies and optimizations to two representative codes running on Jaguar XT5, NUCCOR and Bigstick. The Nuclear Coupled-Cluster Oak Ridge (NUCCOR) application solves the nuclear many-body problem using the coupled-cluster approximation which scales only polynomially with the number of particles and single-particle states. NUCCOR was developed by David Dean and collaborators at ORNL where the single and double excitations of the protons and neutrons are computed, along with a third-order correction, to produce the energy of the system. Bigstick is a shell-model code developed by Calvin Johnson at San Diego State University and Erich Ormand at Lawrence Livermore National Laboratory, which performs an on-the-fly recalculation of the large, sparse Hamiltonian matrix and utilizes Lanczos diagonalization to solve for the lowest eigenvalues and eigenvectors to describe the ground- and excited-states of the system.

This paper is organized as follows. In section 2, we present performance profiling results using CrayPAT and VampirTrace on NUCCOR and Bigstick. Section 3 presents continued improvements to NUCCOR using compiler optimizations and Cray compiler feedback. Section 4 describes identifying high-impact regions of the code and the necessity and outcome of hand-tuning the application. Conclusions and future work are discussed in section 5.

2 Performance Profiling

Performance profiling can be performed with a variety of tools that do not require modifying the application and take varying levels of overhead depending on the depth of profiling desired. This highly useful first step provides application performance data that can be used to identify bottlenecks and plan for actionable steps to improve the code. Profilers commonly used on Jaguar include CrayPAT and VampirTrace.

2.1 CrayPAT

We profiled NUCCOR with CrayPAT[1]. CrayPAT is a package for instrumenting and tracing codes on Cray systems. It is part of the larger Cray performance tools package, which includes Apprentice 2 for visualizing the results of CrayPAT profiling. First the user code is profiled to get an overview of its behavior. Then CrayPAT can be used to further refine the profiling of the code, to trace the most important subroutines and ignore the minor ones, and further statistics can be gathered with another run of the instrumented code.

Information that can be gathered with CrayPAT includes time spent in subroutines (including the line numbers at which significant time is spent), performance imbalance, and hardware counter information. With Apprentice 2, this information can be visualized, and in addition a call tree can be viewed.

2.1.1 Analysis of NUCCOR

NUCCOR is written in Fortran and consists of approximately 16,000 lines of code across 14 files. All but one file are written in Fortran 90, and use advanced language features such as modules and user-defined types. Parallelization is implemented with

MPI. Communication within NUCCOR is exclusively collective.

We performed a CrayPAT analysis on the j-coupled version of NUCCOR, a code that takes advantage of symmetries in certain nuclear configurations to perform energy calculations on larger nuclei. From this analysis we observed that on the small test problem, the code spent more than half its time in a subroutine called `sort`. Examining this code, we found that it was a sorting algorithm reminiscent of bubble sort, with a computational complexity proportional to the square of the number of items to be sorted. We implemented a heap sort instead, and reduced sorting to about 3% of the total runtime. Upon consultation with the authors of this code, we learned that no sorting was necessary, and removed the sorting subroutine altogether, resulting in a 30% speedup on the full problem.

We also profiled the standard version of NUCCOR with CrayPAT, and found that it spends nearly 70% of its time in the subroutine `t2_eqn_store_p_or_n`. In this subroutine, partial sums in the coupled-cluster approximation are summed. It is a very large subroutine, consisting of more than 2000 lines of code, full of multiply nested loops and memory allocations and deallocations. This subroutine is the primary focus of our optimization efforts in NUCCOR.

2.2 VampirTrace

We profiled Bigstick with the VampirTrace [4] and Vampir [3] tools. VampirTrace instruments a code and produces trace files when run. The trace files are then loaded into Vampir, which is used to visualize the trace.

VampirTrace outputs a timeline trace of the workings of an application. Along the x-axis is the timeline of the code, and along the y-axis are the processor numbers. From this visualization of the trace, we can easily pick out the interesting and anomalous behaviors of the program. Subroutines or functions of particular interest can be color-coded to make them stand out. See Table 2 for the color coding used in the output reproduced here.

The instrumented code was run on 100 processors with three different inputs: 1. The standard `li4_7hw` input that comes with the code; 2. The previous input modified so that `Nexcite=6`; and 3. The second input modified to allow 500 Lanczos iterations. From these runs, we were able to make some interesting observations and provide some suggestions for improvements.

Subroutine	*Color*
Bigstick subroutines:	
`reorthogonalize`	Orange
Other Bigstick	Green
MPI subroutines:	
`MPI_Barrier`	Yellow
`MPI_Allreduce`	Blue
`MPI_File_Read_at`	Magenta
`MPI_File_Write_at`	Cyan
Other MPI	Red
VampirTrace overhead	White

Table 2: Colors of functions in Vampir visualization of Bigstick.

2.2.1 Analysis of Bigstick

The top-level Vampir visualization of Bigstick is illustrated in Figure 2. It resembles the patterns seen in algorithms that are applied sequentially across processes. This triangular processor pattern is seen in the orthogonalization of the Lanczos vectors at each iteration. One processor, p, writes to the Lanczos vector file, and then all processors 0 to $p-1$ read from the file and orthogonalize (See Figure 3).

We observe an unusually large number of `MPI_Allreduce`s, seen in blue, on Figure 3. Vampir allows the developer to drill-down by looking at smaller time intervals. Zooming in, we find that these MPI calls originate in the Bigstick subroutine `block_reduce` (the green portions between the blue parts in Figure 4). Bigstick spends more time in `MPI_Allreduce` than in any other MPI function except `MPI_Barrier`.

Bigstick spends a disproportionate amount of time in `MPI_Barrier`s. This indicates work imbalance. Interestingly, most of the time spent in barriers occur within the `clocker` subroutine, used for producing timings to measure the performance of the code. Putting `MPI_Barrier`s in `clocker` results in all processors reporting (nearly) the same timings for every subroutine, hiding evidence of load imbalance.

Using Vampir, we see anomalous behavior at the start of the application run, reproduced in Figure 5. All processors appear to be performing `write_wfn_header`, but all processors are writing

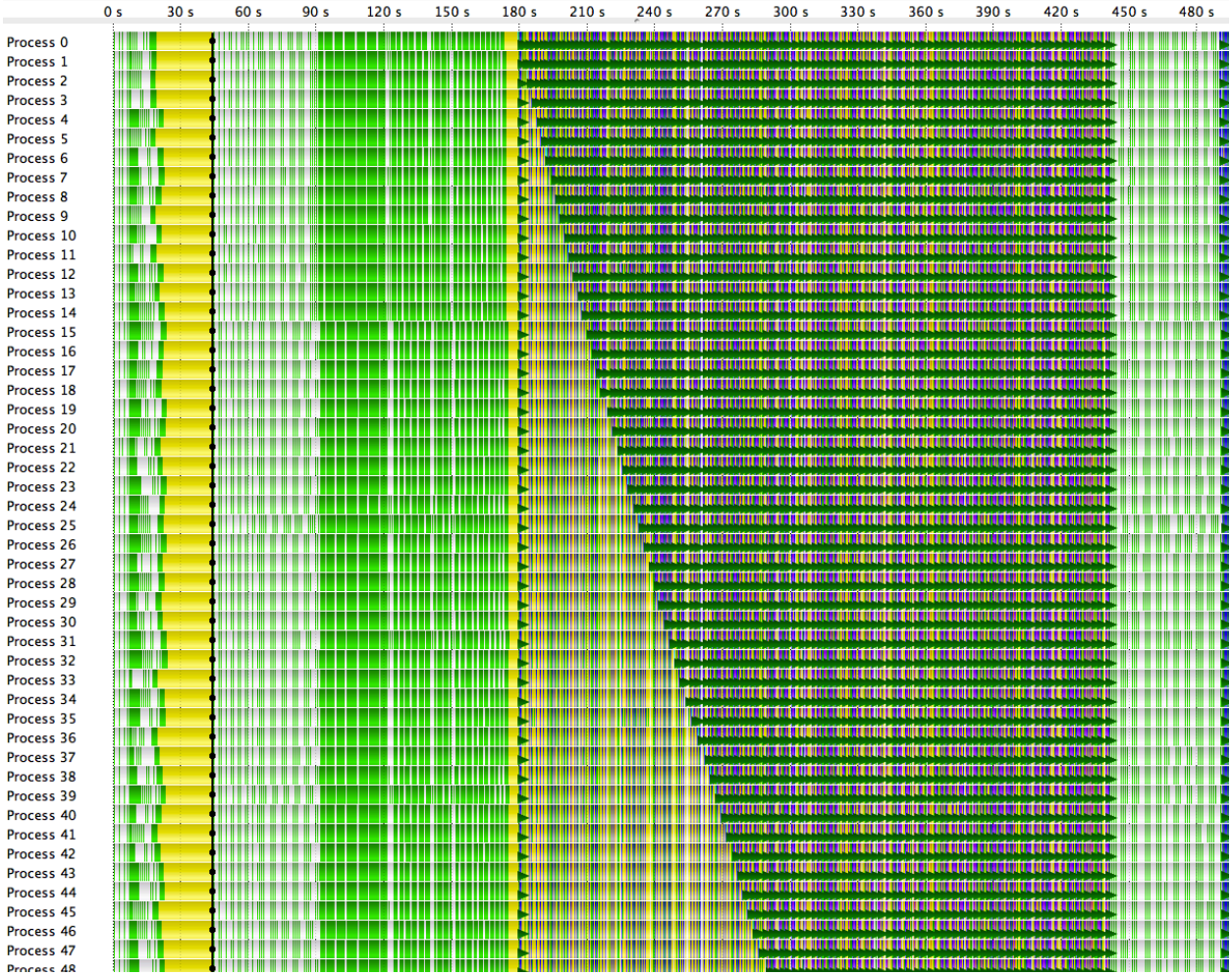
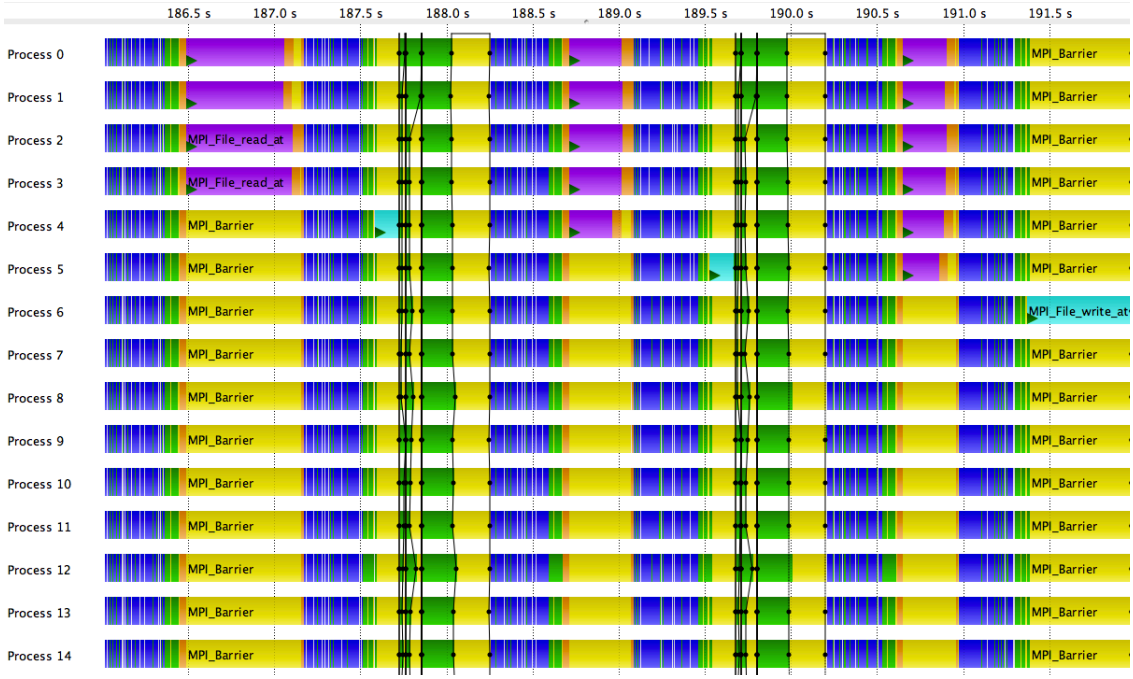
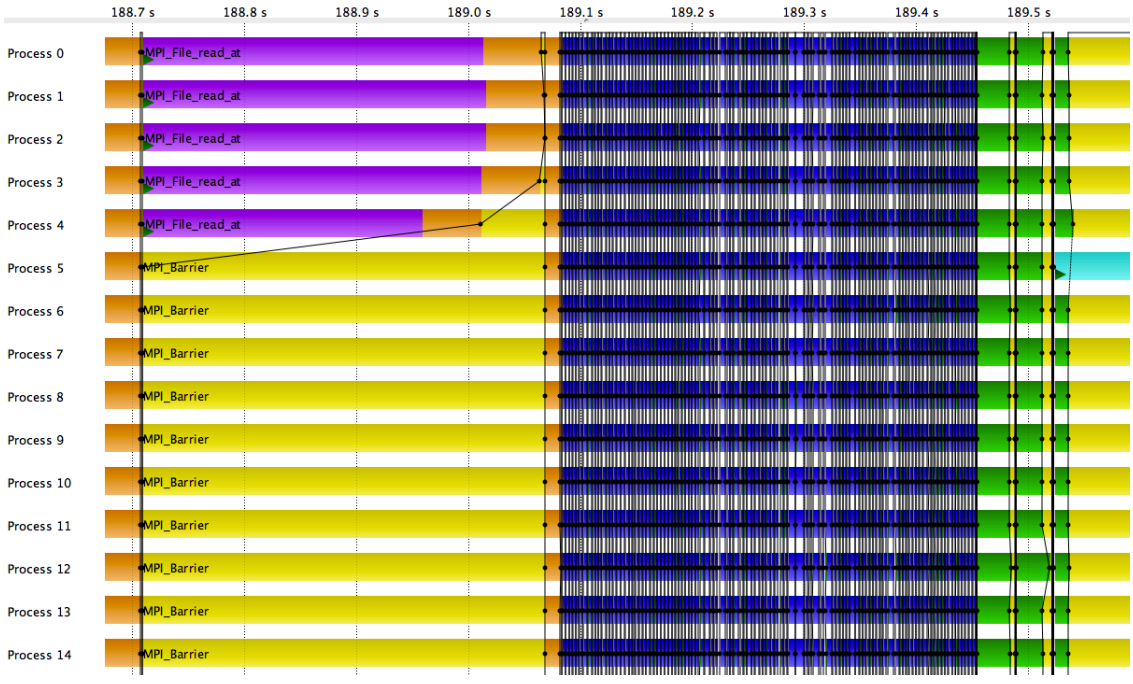


Figure 2: Overview of Bigstick with Vampir. Note the triangle that begins at the 180-second mark.



(a)



(b)

Figure 3: (a) Closeup of three steps within the triangle. Observe the global reorthogonalization followed by the reading by processors 0–3, followed by another reorthogonalization performed locally, then a series of `MPI_Allreduces`, and P4 writing to the file. In the next step, P0–4 read and P5 writes, etc. until all processors read and write. (b) One iteration within the triangle.

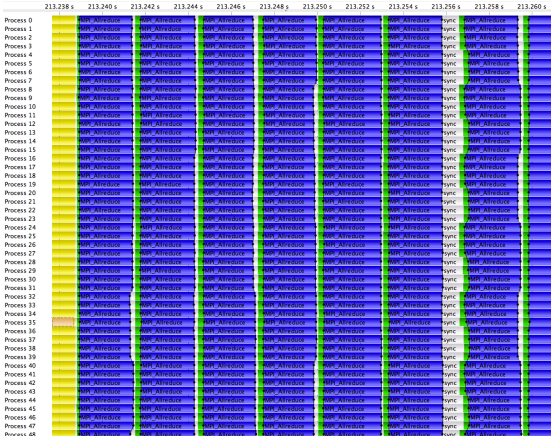


Figure 4: Closeup of `block_reduce` phase. Most of the time is spent in `MPI_Allreduce`. White portion (including portions labeled sync) represents VampirTrace overhead.

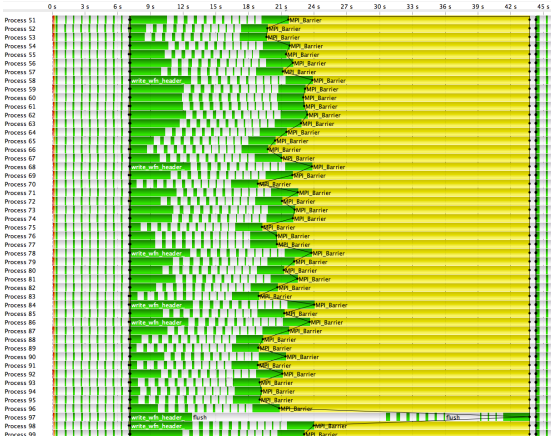


Figure 5: Time spent at beginning in `write_wfn_header` and other subroutines. Timeline is zoomed in to view the culprit processor that is causing all the others to delay. White represents VampirTrace overhead, which is primarily responsible for delay (probably slow writing to trace file due to file system issue), but note the drastically different times across processors spent in `write_wfn_header`.

the exact same data into the file, and simply overwriting one another. Fixing this could drastically reduce the time spent in the initialization phase, especially as the processor count grows and the contention for the file also grows.

2.2.2 Suggested Improvements

A lot of time is spent in `MPI_Barrier` because of the way the Lanczos method is implemented. The barriers are found primarily in the `clocker` subroutine, so removing them from `clocker` would simply cause processes to spend more time within the actual subroutines that `clocker` is bracketing. The net effect would be that processors would still spend a lot of time waiting, just not in a barrier. But this could allow `clocker` to be used as a first-order measure of the load imbalance, which could prove useful.

The bottleneck in the iterative stage is the way that the Lanczos vectors are written and read. This creates a sequentiality that results in many processes being idle while they are waiting for a few others to complete their work. Even if we run more iterations and every process is reading (as they were in the last 400 iterations of Run 3), there is still a delay as all processes wait for a single process to write.

Reducing the amount of orthogonalization performed would speed up the code. It is not usually necessary to orthogonalize at every step — in a worst-case scenario, this will result in some duplicate eigenvalues, but the extreme eigenvalues and eigenvectors should still be fairly accurate. Depending on the conditioning of the operator `applyh`, performing less frequent orthogonalization might not result in any significant loss of accuracy.

Another way to speed up the code is to combine collectives. In the `block_reduce` phase, if it were possible to combine `MPI_Allreduce` calls, it could result in some improvement in performance. The overhead in initializing collective operations could be partially eliminated. For small messages such as these, the time spent setting up the data transfer overshadows the time spent actually transferring data. Bundling these calls together could save some overhead.

If all the processors are writing different output to the file at the beginning of the code, then we suggest looking into ways to stage that writing differently. An algorithm in which processors all write to a single file simultaneously is not scalable as the process count rises. Scalability issues begin to show up by the 10,000-process mark. Reducing the number of processors accessing the file (e.g., group output so that one processor writes on behalf of several others), or using an I/O library such as ADIOS [7] that will do all the hard work of optimizing I/O would

both be good strategies to try.

3 Compiler Optimization

In the course of performance profiling, we observed anomalous behavior in another production version of NUCCOR using the Intel compiler on Jaguar. Additional investigation revealed this behavior to be due to a compiler bug. Since the nuclear physics community relies heavily upon the Intel compiler for high precision, this bug was particularly disconcerting to them. We were intrigued by this problem, however, and set out to see how NUCCOR would perform when compiled with each of the five compilers available on Jaguar, and expanded the experiments to include multiple optimization levels with each compiler as well.

This work led us to examine the performance of the code — in particular, one subroutine that occupied more than half the code's runtime — and test improvements to the algorithms in the code.

The default programming environment is the PGI programming environment, but other programming environments can be loaded. We were curious to see how the PGI compiler would compare to the open-source GNU compiler, the Intel compiler (a particularly interesting case because Jaguar's chips are manufactured by AMD), the Pathscale compiler, and Cray's own compiler, developed in-house for the XT5.

To test the performance of the compilers, we sought a problem that could be run on a moderate number of processors for under an hour. We chose a test problem that computed the energies of the ^{16}O (oxygen-16) nucleus. It runs on exactly 441 processors for roughly forty minutes.

3.1 Procedure

We compiled the code with all five compilers available on the machine, and at multiple optimization levels: `-O0` (no optimizations), `-O1` (some optimizations), `-O2` (moderate optimizations), `-O3` (high optimizations), without any flags (default optimization level), and at the highest optimization level (subjectively chosen based on recommendations from colleagues and compiler experts). The compilers, their versions, and the flags for highest optimization are shown in Table 3. We then ran the code on the test problem and compared timings.

Compiler	*Version*	*Highest Optimization Flags*
Cray	7.1.5	`-O3`
GNU	4.2.2	`-O2 -ffast-math -fomit-frame-pointer -mfpmath=sse`
Intel	11.1.046	`-O3`
Pathscale	3.2	`-Ofast`
PGI	9.0.4	`-fast`

Table 3: Compiler version and optimization information.

3.2 Results

We performed three runs of each compiler and optimization level combination, and averaged the results. In all cases the program produced identical output (with the exception of timings). In Table 4, the winners and runners-up for each optimization level are reproduced. Figure 6 summarizes the results at each optimization level in graphical form.

The Cray compiler is the overall champion, winning or placing at all optimization levels. At the `-O0` and `-O1` optimization levels (where performance is not all that important anyhow) it is only 6% slower at the worst. At the `-O2` and `-O3` optimization levels, it was about 25% faster than its nearest competitor. The surprising result was the performance of the Pathscale compiler with optimal flags — at all the other optimization levels, the Pathscale compiler's performance consistently ranked low, yet it was more than 20% faster than the Cray compiler at the highest level of optimization.

Profiling NUCCOR revealed that more than half the total runtime of the code was spent in one subroutine: `t2_eqn_store_p_or_n`. This function is full of deeply nested loops. While there are many individual loops taking up small portions of the runtime, iterative updates dominate the runtime for NUCCOR. So, if the efficiency of these loops could be improved, the overall runtime could be drastically reduced — a 50% reduction in time spent in `t2_eqn_store_p_or_n` would translate into a 25% reduction in total walltime. The Pathscale compiler's superior performance at the `-Ofast` optimization level was because it optimized these loops better than any other compiler.

Discussing the results of the above experiments with compiler developers at Cray, we found that

Optimization Level	*Best Performer*	*Second Best Performer*	*% difference in walltime*
`-O0`	PGI	Cray	3.54
`-O1`	Intel	Cray	5.88
`-O2`	Cray	PGI	24.9
`-O3`	Cray	PGI	26.2
No flags	Cray	Intel	26.4
Optimal	Pathscale	Cray	20.5

Table 4: Best performance at each optimization level. The final column indicates the percent difference in walltime of the NUCCOR runs of the best and second-best performing compilers.

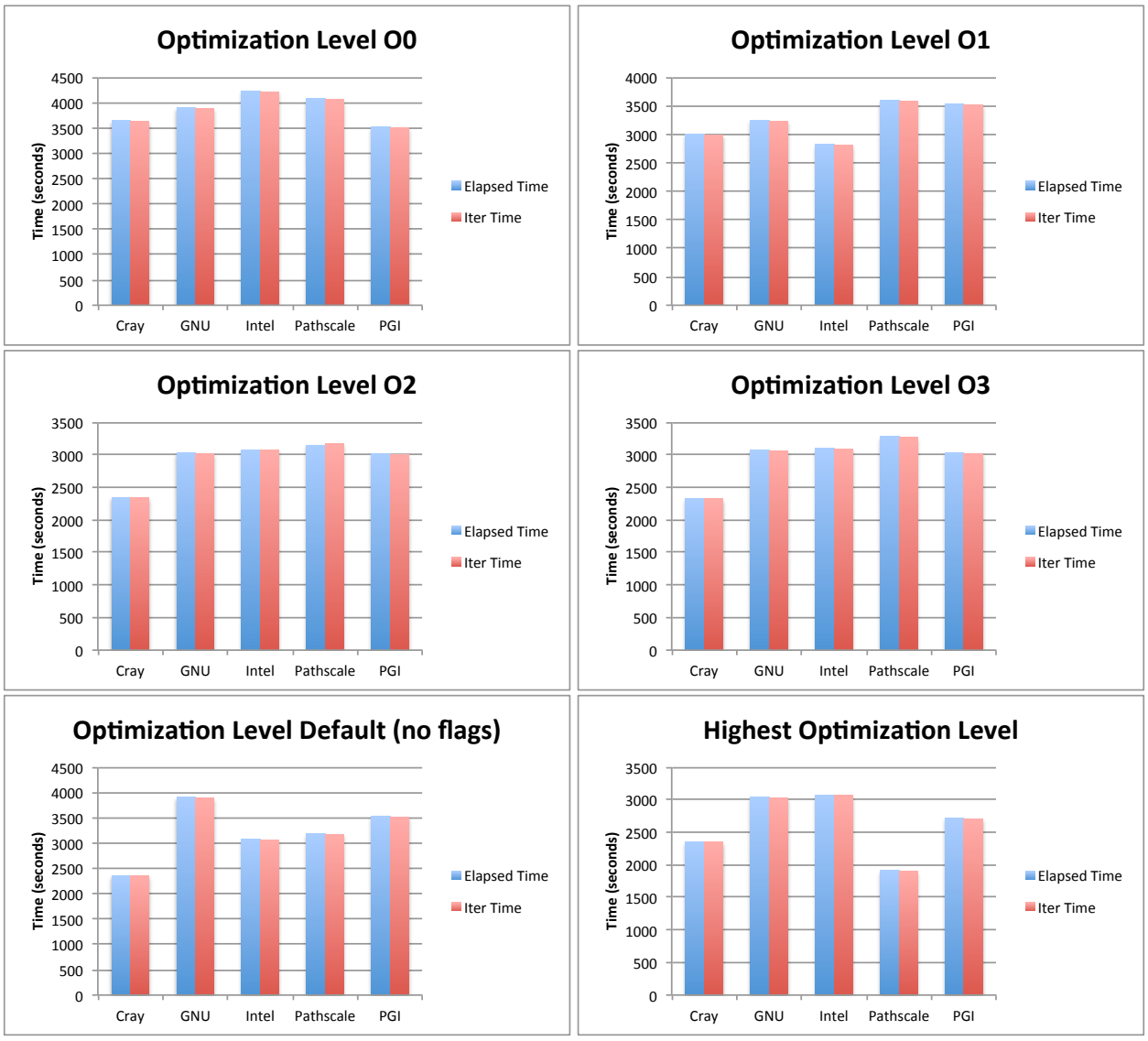


Figure 6: Performance across compilers (Cray, GNU, Intel, Pathscale, and PGI) at various optimization levels.

```
ii=0
do b=below_ef+1,tot_orbs
do j=1,below_ef
  ii=ii+1
  jj=0
  do a=below_ef+1,tot_orbs
  do i=1,below_ef
    jj=jj+1
    t2_ccm_eqn%f5d(a,b,i,j)= t2_ccm_eqn%f5d(a,b,i,j)
        + tmat7(ii,jj)
    t2_ccm_eqn%f5d(b,a,i,j)= t2_ccm_eqn%f5d(b,a,i,j)
        - tmat7(ii,jj)
    t2_ccm_eqn%f5d(a,b,j,i)= t2_ccm_eqn%f5d(a,b,j,i)
        - tmat7(ii,jj)
    t2_ccm_eqn%f5d(b,a,j,i)= t2_ccm_eqn%f5d(b,a,j,i)
        + tmat7(ii,jj)
    ops_cnt=ops_cnt+4
  end do
  end do
end do
end do
```

Figure 7: Sample loop in NUCCOR code exhibiting poor stride.

some loops did not have good performance characteristics. The authors of NUCCOR were thinking about physics, not compilers, when they wrote this code, and as a result have incorporated symmetry into their loops for ease of readability. Figure 7 illustrates this symmetric coding: we access `t2_ccm_eqn%f5d(a,b,i,j)` and its symmetric neighbor `t2_ccm_eqn%f5d(b,a,i,j)` within the same loop. Unfortunately, this makes it challenging for the compiler to optimize the code. As the code stood, the strides through memory were causing cache thrashing and increased bandwidth use. So we decided to test the impact of loop reordering and loop fission on this loop.

4 Compiler Feedback

NUCCOR spends the majority of its time performing the loop updates in the subroutine `t2_eqn_store_p_or_n`. This section of the code is characterized by deeply nested loops performing updates to arrays. There are many of these loops, but they follow a similar pattern, so any optimizations that we could apply to one loop could be applied to most of the others. In this section, we describe the types of loop optimizations that are relevant to our work with NUCCOR.

To figure out how to improve the performance of the loops in the code, we began by examining the annotated output of the compiler. The Cray compiler can produce an annotated listing of the code that details how it was able to optimize the code by invoking the compiler with the flag `-rm`. This annotated output file, marked with the `.lst` suffix, contains annotated line-by-line output followed by more detailed explanations of the compiler's optimization decisions. In Figure 8, we can see an example of this annotated output. The `r8` means that the compiler unrolled the `i` loop eight times.

4.1 Types of Loop Optimization

There are many possible optimizations that can be made to loops. The optimizations relevant to this discussion are loop unrolling, vectorization, reordering, and fission.

Loop unrolling is a means of improving the speed of executing a loop by reducing or eliminating loop

```
371.                ii=0
372. 1--------<     do b=below_ef+1,tot_orbs
373. 1 2------<     do j=1,below_ef
374. 1 2              ii=ii+1
375. 1 2              jj=0
376. 1 2 3----<       do a=below_ef+1,tot_orbs
377. 1 2 3 r8-<       do i=1,below_ef
378. 1 2 3 r8           jj=jj+1
379. 1 2 3 r8           t2_ccm_eqn%f5d(a,b,i,j)=... +tmat7(ii,jj)
380. 1 2 3 r8           t2_ccm_eqn%f5d(b,a,i,j)=... -tmat7(ii,jj)
381. 1 2 3 r8           t2_ccm_eqn%f5d(a,b,j,i)=... -tmat7(ii,jj)
382. 1 2 3 r8           t2_ccm_eqn%f5d(b,a,j,i)=... +tmat7(ii,jj)
383. 1 2 3 r8           ops_cnt=ops_cnt+4
384. 1 2 3 r8->       end do
385. 1 2 3---->       end do
386. 1 2------>     end do
387. 1-------->     end do
```

Figure 8: Annotated output of compiler for sample loop in Figure 7.

```
! original loop
  do i = 1,N
    x(i) = i
  end do
```

```
! unrolled loop
  do i = 1,N,2
    x(i) = i
    x(i+1) = i+1
  end do
```

Figure 9: Simple loop unrolling. Loop on right increments loop index only $N/2$ times, at the expense of larger executable size.

```
! original loop
  do i = 1,N
    do j = 1,M
      x(i,j) = i*j
    end do
  end do
```

```
! reordered loop
  do j = 1,M
    do i = 1,N
      x(i,j) = i*j
    end do
  end do
```

Figure 10: Simple loop reordering. Loop on right has better stride in column-oriented Fortran.

control structures. Executing control structures (e.g., incrementing the loop index) takes up valuable execution time that could instead be spent executing the contents of the loop. The tradeoff is that the size of the executable is increased, because instructions are explicitly written into the binary. For example, the loop in Figure 9, when unrolled, increments the index half the number of times as the original loop, but requires additional instructions be explicitly written into the binary.

Vectorization is the process of changing the loop to process one operation on multiple pairs of operands at once, rather than on only a single pair of operands. Vectorization can improve performance by several orders of magnitude, if it is done properly and the data sets are sufficiently large. Vectorization is possible only for loops without data dependence within the loop (e.g., `x(i+1)` must be independent of `x(i)`).

Reordering is the process of rearranging the indexing of a loop or multiply-nested loop to improve performance. Some looping arrangements access memory in an inefficient manner, requiring the loading and reloading of sections of an array into cache more frequently than necessary. Loop reordering, e.g., interchanging the inner and outer loop indices, can fix this problem. Figure 10 has an example of simple loop reordering, and Figure 11 shows the effect of incrementing column-wise or row-wise on the memory stride.

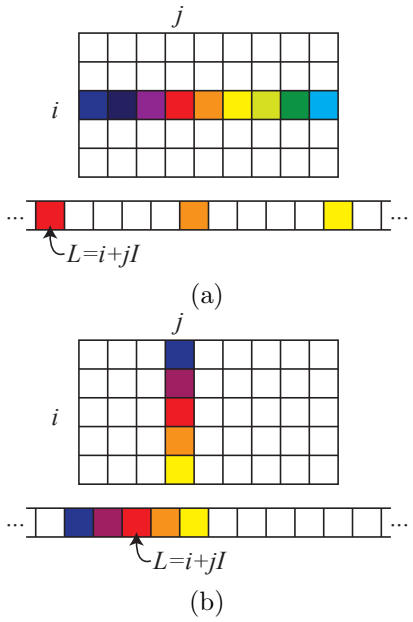


Figure 11: Memory stride when accessing an array (a) row-wise versus (b) column-wise. L represents address in memory space of element (i, j) in array.

Fission is the process of splitting one loop into multiple loops over the same index range, with the goal of improving the locality of reference. Reordering the loop can often fix problems with locality of reference, but if there are two arrays being incremented within the loop, one of which has good stride, it might make more sense to split the loop into two separate loops, each of which can be ordered to take advantage of locality of reference.

5 Hand Tuning

While the compiler can often perform some of the loop optimizations described above, for maximum performance, human intervention is required. We identified some loops from the NUCCOR subroutine `t2_eqn_store_p_or_n` that were good candidates for hand tuning. To test the efficacy of hand tuning, we created a test code comprised of a driver plus the code fragments in Figures 12, 13, and 14, representing the original loop, a reordering to improve memory stride, and loop fission, respectively, and compiled it under all five compilers at the highest optimization level. In addition, to gauge the impact of the Pathscale `-Ofast` flag, we compiled with both `-O3` and `-Ofast` for Pathscale.

Compiler	*Wall time for two iterations (s)*		
	Original	*Reordered*	*Fissioned*
Cray	16.982	9.522	2.527
GNU	16.744	9.351	4.208
Intel	17.046	9.558	4.523
Pathscale			
`-O3`	13.391	11.659	4.076
`-Ofast`	1.713	1.414	4.094
PGI	24.869	8.879	4.363

Table 5: Performance of reconfigured loops. Test program was compiled using optimal flags from Figure 3, except Pathscale as noted in table.

5.1 Results

With the exception of Pathscale compiled with `-Ofast`, the reordered loop took less time than the original loop, and the fissioned loop took less time than the reordered loop. For the Cray compiler, the reordering reduced the runtime of the loop by nearly a factor of 2, and the fissioning reduced the runtime by a factor of 8. In the case of Pathscale compiled with `-Ofast`, the fissioned loop had the worst performance, and the reordered loop improved performance slightly. The secret to the Pathscale compiler's success is not clear; the fact that it performed worse on the fissioned loop suggests that fission is not the technique used by the compiler.

6 Future Work

We have demonstrated the need for optimization in nuclear physics codes, and the potential for large performance gains. We are in the process of implementing what we have learned from these profiles, compiler feedback, and the testing of loop restructuring in NUCCOR. Preliminary results look promising.

For nuclear physicists to remain competitive users of leadership high-performance computing resources, their codes must continue to evolve. As we demonstrated with NUCCOR and Bigstick, there are a number of relatively simple changes that can be applied to improve performance.

Further performance gains can be made through more in-depth measures. Many nuclear physics

```
ii = 0
do b = abmin, abmax
   do j = ijmin, ijmax
      ii = ii+1
      jj = 0
      do a = abmin, abmax
         do i = ijmin, ijmax
            jj = jj+1
            f5d(a,b,i,j) = f5d(a,b,i,j) + tmat7(ii,jj)
            f5d(b,a,i,j) = f5d(b,a,i,j) - tmat7(ii,jj)
            f5d(a,b,j,i) = f5d(a,b,j,i) - tmat7(ii,jj)
            f5d(b,a,j,i) = f5d(b,a,j,i) + tmat7(ii,jj)
         end do
      end do
   end do
end do
```

Figure 12: Sample NUCCOR loop from Figure 7 as translated into test code.

```
do i = ijmin, ijmax
   jj = 0
   do a = abmin, abmax
      do j=ijmin, ijmax
         jj = jj+1
         ii = 0
         do b = abmin, abmax
            ii = ii+1
            f5d(a,b,i,j) = f5d(a,b,i,j) + tmat7(ii,jj)
            f5d(b,a,i,j) = f5d(b,a,i,j) - tmat7(ii,jj)
            f5d(a,b,j,i) = f5d(a,b,j,i) - tmat7(ii,jj)
            f5d(b,a,j,i) = f5d(b,a,j,i) + tmat7(ii,jj)
         end do
      end do
   end do
end do
```

Figure 13: Loop from Figure 12 reordered for better stride in `tmat7` variable.

```
ii = 0
do j = ijmin, ijmax
   do b = abmin, abmax
      ii = ii+1
      jj = 0
      do i = ijmin, ijmax
         do a = abmin, abmax
            jj = jj+1
            f5d(a,b,i,j) =
              f5d(a,b,i,j) +
              tmat7(ii,jj)
            f5d(a,b,j,i) =
              f5d(a,b,j,i) -
              tmat7(ii,jj)
         end do
      end do
   end do
end do
```

```
jj = 0
do i = ijmin, ijmax
   do a = abmin, abmax
      jj = jj+1
      ii = 0
      do j = ijmin, ijmax
         do b = abmin, abmax
            ii = ii+1
            f5d(b,a,i,j) =
              f5d(b,a,i,j) -
              tmat7(ii,jj)
            f5d(b,a,j,i) =
              f5d(b,a,i,j) +
              tmat7(ii,jj)
         end do
      end do
   end do
end do
```

Figure 14: Loop from Figure 12 fissioned for better performance.

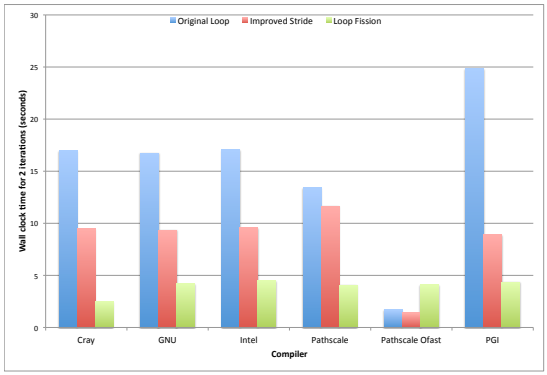


Figure 15: Performance of reconfigured loops compiled with Cray, GNU, Intel, Pathscale `-O3`, Pathscale `-Ofast`, and PGI compilers, respectively, on original (blue), reordered (red), and fissioned (green) loop.

codes use a manager-worker paradigm, with all processes marching in lock-step and coordinated by a central manager process, which becomes less scalable as we approach hundreds of thousands of processes. Rewriting the algorithms to be less centralized and more asynchronous would improve scalability. In many cases this would require a complete rewrite of the code, which will require a significant investment of person-hours that might instead be spent making scientific discoveries with existing codes. But in order for nuclear physicists to be able to solve large problems that are currently out of reach, this investment will have to be made. Restructuring nuclear physics codes to expose more parallelism would be beneficial, especially as we move toward hybrid systems.

7 Acknowledgments

This work was supported by the U.S. Department of Energy under Contract Nos. DE-AC05-00OR22725 with UT-Battelle, LLC (Oak Ridge National Laboratory (ORNL)) and under DE-FC02-07ER41457 (UNEDF SciDAC Collaboration). This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the

U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

8 About the Authors

Rebecca J. Hartman–Baker is a computational scientist at the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory. Her research interests include optimization, ill-posed problems, and load balancing at the petascale and beyond. She can be reached at *hartmanbakrj@ornl.gov*.

Hai Ah Nam is a computational scientist at the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory. Her research interests include low-energy nuclear structure research at large-scale. She can be reached at *namha@ornl.gov*.

References

[1] Craypat `http://www.olcf.ornl.gov/kb_articles/software-jaguar-craypat/`

[2] UNEDF `http://www.unedf.org`

[3] Vampir `http://www.nccs.gov/computing-resources/jaguar/software/?software=vampir`

[4] VampirTrace `http://www.nccs.gov/computing-resources/jaguar/software/?software=vampirtrace`

[5] G. F. Bertsch, D. J. Dean and W. Nazarewicz SciDAC Rev. 6, 42 (2007).

[6] D. J. Dean and G. Hagen and M. Hjorth-Jensen and T. Papenbrock, "Computational aspects of nuclear coupled-cluster theory," Comput. Sci. Disc. 1, 015008 (2008).

[7] J. Lofstead, Zheng Fang, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *Parallel & Distributed Processing*, 2009. IPDPS 2009.

[8] Nuclear Science Advisory Committee, "The Frontiers of Nuclear Science: A Long Range Plan," `http://www.sc.doe.gov/np/nsac/docs/Nuclear-Science.Low-Res.pdf`, 2007.