

More SCALABILITY, Less PAIN

This is the story of a simple programming model, its implementation for extreme computing, and a breakthrough in nuclear physics. A critical issue for the future of high-performance computing is the programming model to use on next-generation architectures. Described here is a promising approach: program very large machines by combining a simplified programming model with a scalable library implementation. The presentation takes the form of a case study in nuclear physics. The chosen application addresses fundamental issues in the origins of our Universe, while the library developed to enable this application on the largest computers may have applications beyond this one.

Today, the largest computers in the world have more than 100,000 individual processors. Scientists the world over anticipate new designs with more than one million processors in the near future, and applications will be ready to make use of such processing power. But these same scientists share the concern that the dominant approach for specifying parallel algorithms, the message-passing model and its standard instantiation in the Message Passing Interface (MPI), will become inadequate even as such processing power arrives. They fear that entirely new programming models and new parallel programming languages will be necessary, stalling application development to upgrade existing application code.

No doubt many applications will require significant change as we enter the exascale age (one quintillion, 10^{18} , floating point operations per second, flop/s), and some will be rewritten entirely. But it is hardly inevitable that the only way forward is through significant upheaval. The key to avoiding such complications is to adopt a simple programming model that may be less general than message passing while still being useful for most applications. If this programming model is then implemented in a scalable way, the application may be able to reuse most of its highly tuned code, simplify the way it handles parallelism, and still scale to hundreds of thousands of processors.

Here, then, is a case study from nuclear physics of just such an approach. Physicists from Argonne National Laboratory (ANL) developed Green's Function Monte Carlo (GFMC) method years ago but recently needed to scale it for execution on more processors than before, for nuclear theory compu-

tations for carbon-12 (^{12}C), carbon's most important isotope. The carbon-12 isotope was a specific target of the SciDAC Universal Nuclear Energy Density Functional (UNEDF) project (see Further Reading, p37), which funded the work described here in both the Physics Division and the Mathematics and Computer Science Division at Argonne. Computer time for the calculations was provided by the DOE Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program, and the large computations were carried out on the IBM Blue Gene/P (BG/P) at the Argonne Leadership Computing Facility (ALCF).

Nuclear Physics

A major goal in nuclear physics is to understand how nuclear binding, stability, and structure arise from the underlying interactions between individual nucleons. This approach will also allow the accurate calculation of nuclear matrix elements needed for some tests (such as double beta decay) of the standard model and of nuclei and processes not currently accessible in the laboratory. Such a calculation can be useful for astrophysical studies and for comparisons to future radioactive beam experiments.

In the past decade there has been much progress in several approaches to the nuclear many-body problem for light nuclei. The quantum Monte Carlo (QMC) method used here consists of two stages: variational Monte Carlo (VMC) to prepare an approximate starting wave function, and GFMC, which improves it.

The first application of Monte Carlo methods to nuclei was a VMC calculation by Pandhari-

Scientists the world over anticipate new designs with more than one million processors in the near future, and applications will be ready to make use of such processing power.

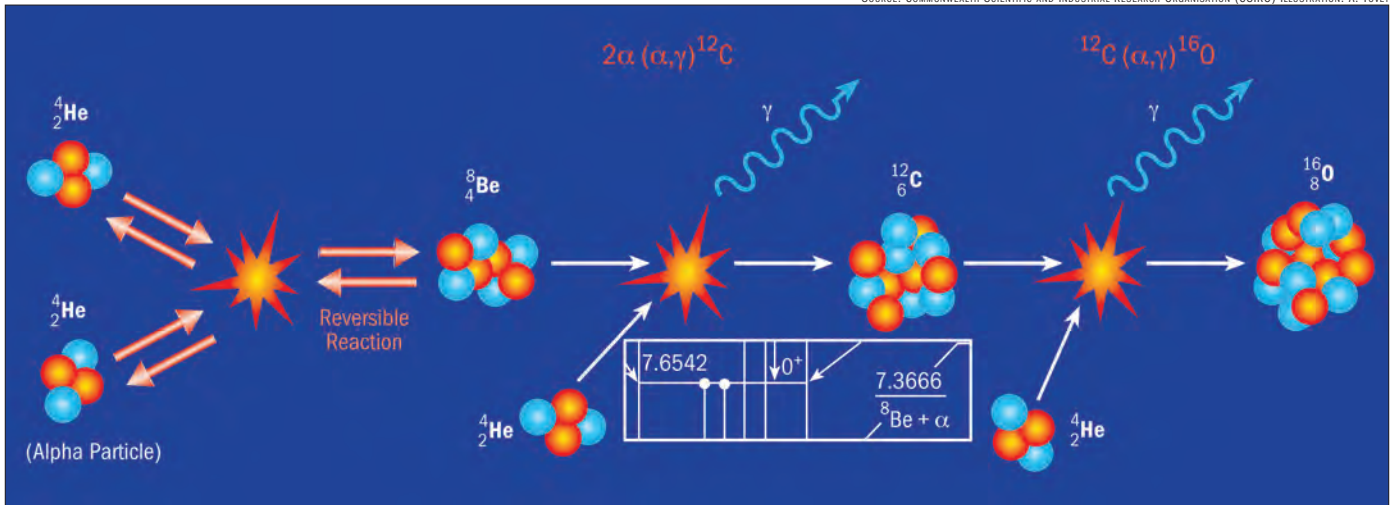


Figure 1. A schematic view of the carbon-12 (${}^{12}\text{C}$) and oxygen-16 (${}^{16}\text{O}$) production by alpha burning. The ${}^8\text{Be} + \alpha$ reaction proceeds dominantly through the 7.65 MeV triple-alpha resonance in ${}^{12}\text{C}$ (the Hoyle state). Both sub- and above-threshold ${}^{16}\text{O}$ resonances play a role in the ${}^{12}\text{C}(\alpha, \gamma) {}^{16}\text{O}$ capture reaction.

pande and collaborators, who computed upper bounds to the binding energies of hydrogen and helium in 1981. Six years later, Carlson improved on the VMC results by using GFMC and obtained essentially exact results (within Monte Carlo statistical errors of 1%). Reliable calculations of heavier nuclei arrived in the mid-1990s; the current frontier for GFMC calculations is carbon-12. The increasing size of the nucleus computable is due to both the increasing size of available computers and the significant improvements in the algorithms used. The importance of this work is highlighted by the award of the 2010 American Physical Society Tom W. Bonner Prize in Physics to Drs. Steven Pieper and Robert Wiringa. We will concentrate here on the developments that enable the IBM Blue Gene/P to be used for carbon-12 calculations.

According to the report, “Scientific Grand Challenges: Forefront Questions in Nuclear Science and the Role of High Performance Computing” (Further Reading, p37), reliable carbon-12 calculations will be the first step toward precise calculations of reaction rates for stellar burning (figure 1); these reactions are critical building blocks to life. The thermonuclear reaction rates of alpha-capture during stellar helium burning determine the carbon-to-oxygen ratio, with broad consequences for the production of all elements made in subsequent burning stages of carbon, neon, oxygen, and silicon. These rates also determine the sizes of the iron cores formed in Type II supernovae and thus the ultimate fate of the collapsed remnant into either a neutron star or a black hole. Therefore, the ability to accurately model stellar evolution and nucleosynthesis is highly dependent on a detailed knowledge of these two reactions. Yet, all current, realistic theoretical models

fail to describe the alpha-cluster states, and no fundamental theory of these reactions exists. The first focus is the Hoyle state in carbon-12, the target of GFMC calculations on BG/P.

The first step, VMC, finds an upper bound, E_T , to an eigenvalue, which involves the computation of a 36-dimensional integral over the positions of the 12 nucleons. The integral is computed by standard Monte Carlo methods. Over the years, sophisticated approximate eigenvectors for light nuclei have been developed. The total numbers of components in the vectors are 16, 160, 1,792, 21,504, and 267,168 for helium-4, lithium-6, beryllium-8, boron-10, and carbon-12, respectively. It is this exponential growth in components that constrains the GFMC methods to light nuclei.

The second step, GFMC, estimates the lowest-energy eigenstate from the VMC result. The VMC approximate vector can be thought of as the desired exact eigenvector plus contamination from the excited state vectors. These higher-energy contaminants are damped out by the GFMC propagation. During this process, the computed energy decreases until the remaining contamination is smaller than the Monte Carlo statistical errors, and the calculation is then stopped. The propagation is done in a series of small steps. Each step requires a new 36-dimensional integral. Thus the whole GFMC calculation is a huge integral, typically more than 10,000 dimensions. Various tests indicate that the GFMC calculations of p-shell binding energies have errors of 1–2%. As Monte Carlo samples are propagated, they can wander into regions of low importance and be discarded. Or they can find areas of large importance and will then be multiplied. Hence, the number of computed samples fluctuates during the calculation.

According to the report, “Scientific Grand Challenges: Forefront Questions in Nuclear Science and the Role of High Performance Computing,” reliable carbon-12 calculations will be the first step toward precise calculations of reaction rates for stellar burning.

What Is a Parallel Programming Model?

A programming model shows the way programmers think about the computer they are programming. Most familiar is the von Neumann model, in which a single processor fetches both instructions and data from a single memory, executes the instructions on the data, and stores the results back into the memory. Execution of a computer program consists of looping through this process over and over. In reality, today's sophisticated hardware may perform multiple instructions in parallel, but in the von Neumann programming model, each instruction is executed singularly.

In parallel programming models, the programmer explicitly manages the

simultaneous execution of multiple instructions. Two major classes of such models are shared-memory models, in which multiple processes fetch instructions and data from the same memory, and distributed-memory models, in which each processor accesses its own memory space. Distributed memory models are sometimes referred to as message-passing models, because they are like collections of von Neumann models with the additional property that data can be moved from the memory of one processor to that of another through the sending and receiving of messages.

Programming models come alive via computer languages (libraries), which provide

the actual syntax for writing parallel programs to be executed on parallel machines. In the case of the message-passing model, the dominant library is MPI, which has been implemented on all parallel machines. A number of parallel languages exist for writing programs for the shared-memory model. One such is OpenMP, which has both C and Fortran versions.

We say that a programming model is general if we can express any parallel algorithm with it. The programming models discussed so far are completely general; there may be advantages to using a programming model that is not completely general.

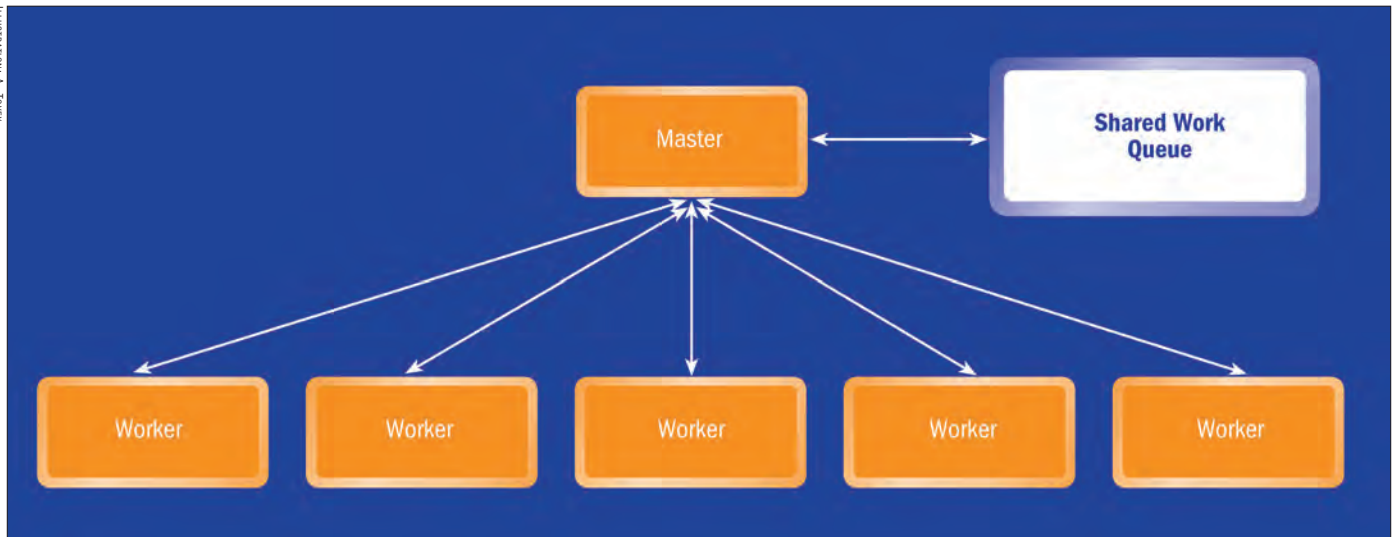


Figure 2. The classical Master/Worker Model. A single process coordinates load balancing.

The problem's large size demands the use of the largest available computers. The GFMC program was written in the mid-1990s using MPI, and until 2008, was largely unchanged. The method depended on the complete evaluation of several Monte Carlo samples on each node. Because of the discarding or multiplying of samples as they wandered, the work had to be periodically redistributed over the nodes. For carbon-12, only about 15,000 samples are needed, so using the many 10,000s of nodes on Argonne's BG/P requires a finer-grained parallelization in which one sample is computed on many nodes. The ADLB library was developed to expedite this approach.

The next nucleus of interest after carbon-12 will be oxygen. Estimates indicate this problem will be between 1,200 and 1,500 times harder than car-

bon-12 and will require exascale computing, which should become available within the decade.

A Simple Programming Model

Here we focus on a partially specialized, high-level programming model (see sidebar "What Is a Parallel Programming Model?"). One of the earliest parallel programming models is the master/worker model, shown in figure 2. In master/worker algorithms one process coordinates the work of many others. The programmer divides the overall work to be done into work units, each of which will be done by a single worker process. Workers ask the master for a work unit, complete it, and send the results back to the master, implicitly asking for another work unit to work on. There is no communication among the workers. While this model is unsuitable

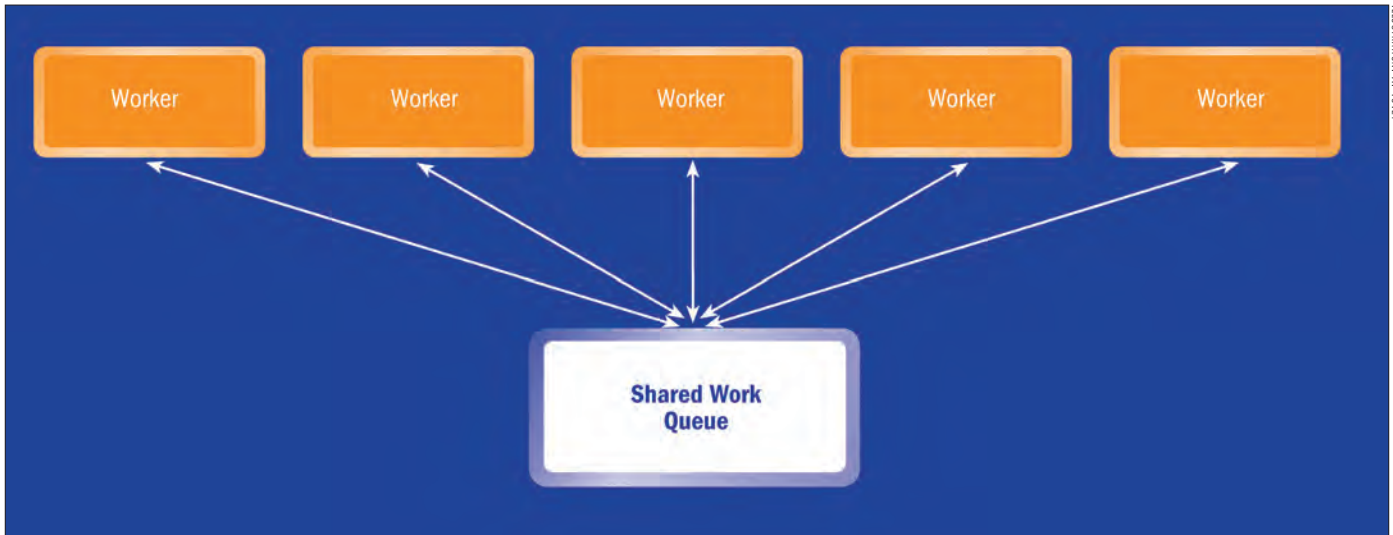


Figure 3. Eliminating the master process with ADLB.

for many important parallel algorithms, it has certain attractive features, such as automatic load balancing in the face of widely varying sizes of work unit. One worker can work on one “shared” unit of work while others work on many units that require shorter execution time; the master will ensure that all processes are busy as long as there is work to do. In some cases workers create new work units, which they communicate to the master so that it can keep them in the work pool for distribution to other workers as appropriate. Complications, such as sequencing certain subsets of work units, managing multiple types of work, or prioritizing work, can all be handled by the master process.

This model’s chief drawback is its lack of scalability. In other words, the master can become a communication bottleneck if there are many workers. Also, there may not be enough memory associated with the master process to store all the work units.

At the beginning of the SciDAC UNDEF project, Argonne’s GFMC code was well-behaved in the master/worker style, using Fortran-90 and MPI to implement the model. The simple master/worker model was improved by storing the shared work on the various workers. The master told workers how to exchange the work to achieve load balancing. Yet, for the carbon-12 problem, the number of processes needed to increase by at least an order of magnitude. To get there, the work units would have to be divided into much smaller units so that what was previously done on just one worker could be distributed to many. We wished to retain the basic master/worker model but modify and implement it in a way that would meet requirements for scalability (because we were targeting Argonne’s 163,840-core BG/P, among other machines) and management of complex, large, interrelated work units.

The key was to further abstract (and simplify) by eliminating the master as the process that controls

```

Basic calls:
ADLB_Init(num_servers, am_server, app_comm)
ADLB_Server()
ADLB_Put(type, priority, len, buf, answer_dest)
ADLB_Reserve(req_types, handle, len, type, prio, answer_dest)
ADLB_Ireserve(...)

ADLB_Get_reserved(handle, buffer)
ADLB_Set_problem_done()
ADLB_Finalize()

Other calls:
ADLB_{Begin,End}_batch_put()
Getting performance statistics with ADLB_Get_info(key)

```

Figure 4. The ADLB application programming interface.

work sharing. In this new model, shown in figure 3, “worker” processes access a shared work queue directly, without communicating with an explicit master process. They make calls to a library to “put” work units into this work pool and “get” them out to work on them. The library built to implement this model is called the Asynchronous Dynamic Load Balancing (ADLB) library. ADLB hides communication and memory management from the application, providing a simple programming interface (sidebar “A Parallel Sudoku Solver with ADLB” p 34).

The ADLB Library

The ADLB application programmer interface (API) is simple. The complete API is shown in figure 4, but only three of the function calls truly provide the essence of its functionality. The most important functions are shown in orange.

To put a unit of work into the work pool, an application uses `ADLB_Put`, specifying the work unit by an address in memory and a length in bytes, and assigning a work type and a priority. To retrieve a unit of work, an application goes through a two-step process. It first uses `ADLB_Reserve`,

A Parallel Sudoku Solver with ADLB

To illustrate how ADLB can be used to express large amounts of parallelism in a compact and simple way, we describe a parallel program to solve the popular game of Sudoku. A typical Sudoku puzzle is illustrated on the left side of figure 5. The challenge is to fill in the small boxes so that each of the digits 1–9 appears exactly once in each row, column, and 3 x 3 box. The ADLB approach to solving the puzzle on a parallel computer is shown on the right side of figure 5. The

work units to be managed by ADLB are partially completed puzzles, or Sudoku “boards.” We start by having one process put the original board in the work pool. Then all processes proceed to take a board out of the pool and find the first empty square. For each apparently valid value (which they determine by a quick look at the other values in the row, column, and box of the empty square) they create a new board by filling in the empty square with the value and put the resulting board

back in the pool. Then they get a new board out of the pool and repeat the process until the solution is found. The process is illustrated in figure 6.

At the beginning of this process the size of the work pool grows very fast as one board is taken out and is replaced with several. But as time goes by, more and more boards are taken out and not replaced at all since there are no valid values to put in the first empty square. Eventually a solution will be found or ADLB will detect that the work pool is empty, in which case there is no solution.

Progress toward the solution can be accelerated by using ADLB’s ability to assign priorities to work units. If we assign a higher priority to boards with a greater number of completed squares, then effort will be concentrated on boards that are closer to solution and the other boards may never need to be further processed.

A human Sudoku puzzle solver uses a more intelligent algorithm than this one, relying on deduction rather than guessing. We can add deduction to our ADLB program by inserting an arbitrary amount of reasoning-based completion of squares just before the “find first blank square” step in figure 5, where “ooh” stands for “optional optimization here.”

Ten thousand-by-ten thousand Sudoku, anyone?

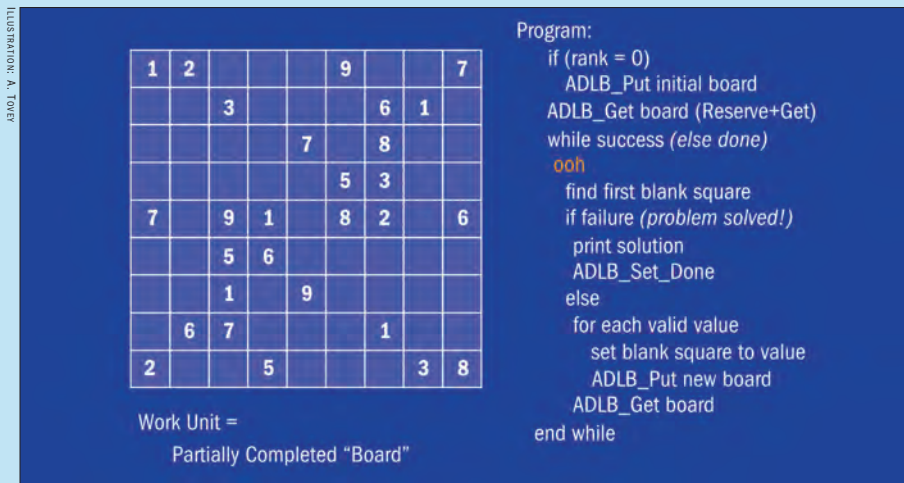


Figure 5. A Sudoku puzzle and a program to solve it with ADLB.

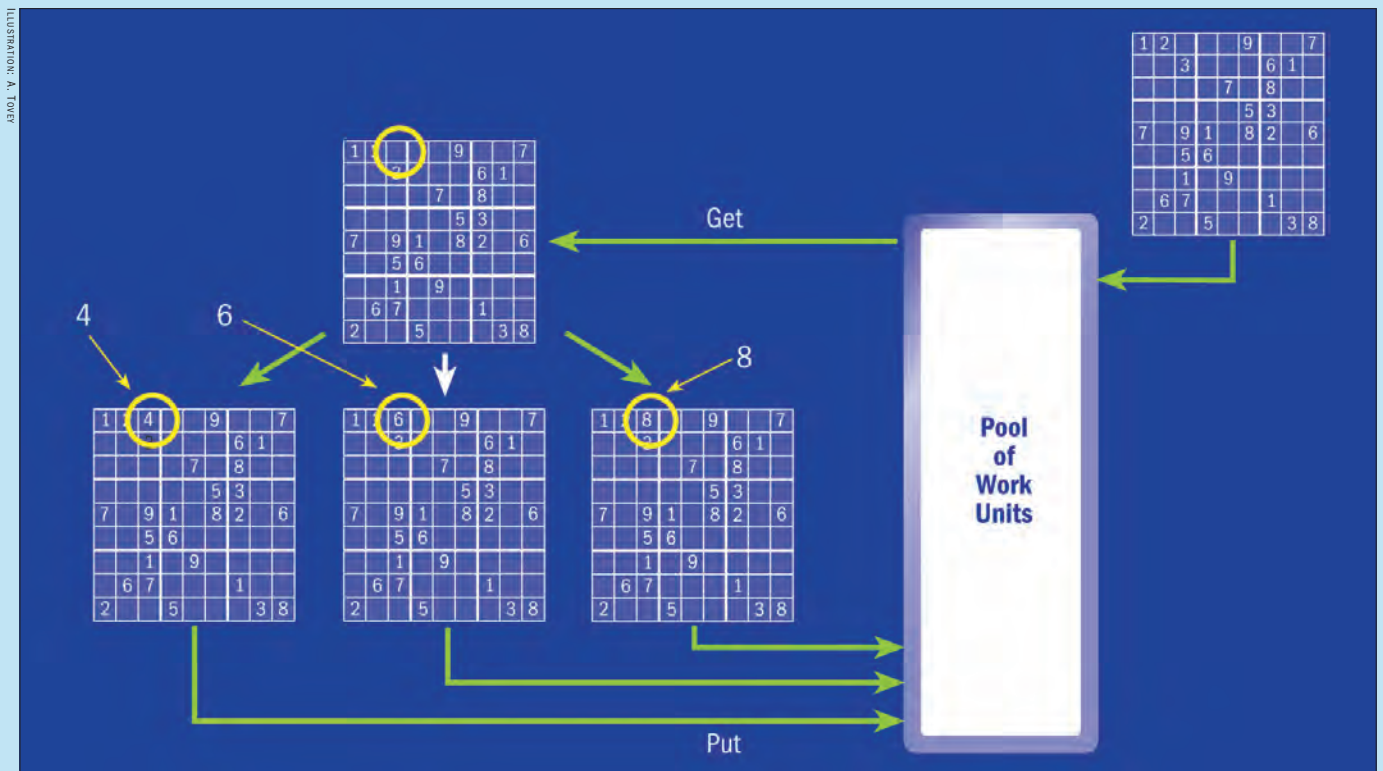
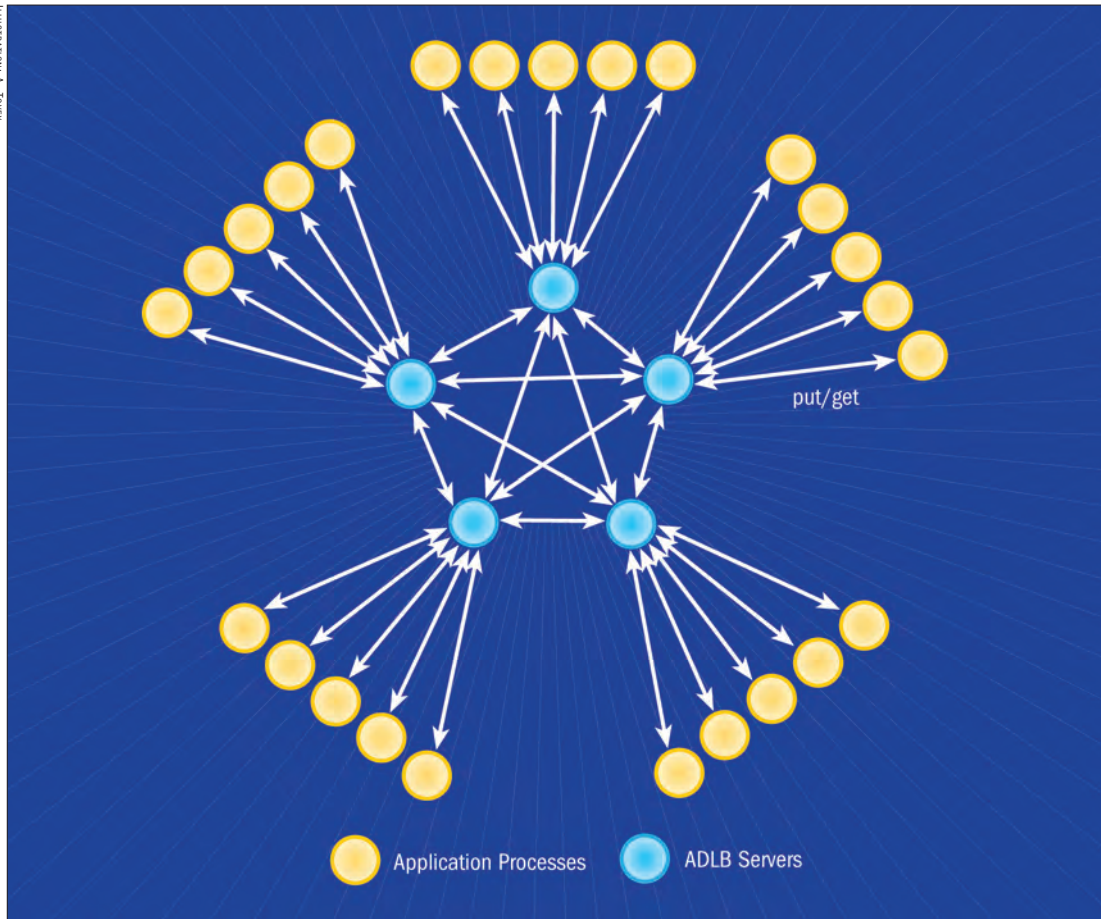


Figure 6. The Sudoku solution algorithm.



ADLB refers to the application processes as client processes, and each ADLB server communicates with a fixed number of clients.

Figure 7. *The architecture of ADLB.*

specifying a list of types it is looking for. ADLB will find such a work unit if it exists and will send back a “handle,” a global identifier, along with the size of the reserved work unit. (Not all work units, even of the same type, need be of the same size.) This gives the application the opportunity to allocate memory to hold the work unit. Then the application uses `ADLB_Get_reserved`, passing the handle of the work unit it has reserved and the address where ADLB will store it.

When work is created by `ADLB_Put`, a destination rank can be given, specifying the only application process that will be able to reserve this work unit. This provides a way to route results to specific processes that may be collecting partial results as part of the application’s algorithm. Usually such a process is the one that originated the work package.

Figure 7 shows the current implementation of ADLB. When an application starts, after calling `MPI_Init`, it then calls `ADLB_Init`, specifying the number of processes in the parallel job to be dedicated as ADLB server processes. ADLB designates these for itself and returns the rest in an MPI communicator for the application to use independently of ADLB. ADLB refers to the application processes as client processes, and each ADLB server

communicates with a fixed number of clients. When a client calls `ADLB_Put`, `ADLB_Reserve`, or `ADLB_Get_reserved`, it triggers communication with its server, which then may communicate with other servers to satisfy the request before communicating back to the client.

The number of servers devoted to ADLB can be varied. The aggregate memory on the servers must be large enough to hold the peak amount of work packages during the job. But it should not be much larger than the minimum necessary, because each server node is a node that is not performing application computations. For the GFMC application doing carbon-12, a ratio of about one server for every 30 application processes was best.

When a process requests work of a given type, ADLB can find it in the system either locally (that is, the server receiving the request has some work of that type) or remotely, on another server. Every server knows the status of work available on other servers, because a status vector circulates constantly around the ring of servers. As the status vector passes through each server, it receives information about the types of work the server is holding and updates the server with information about work on other servers. The price we pay for

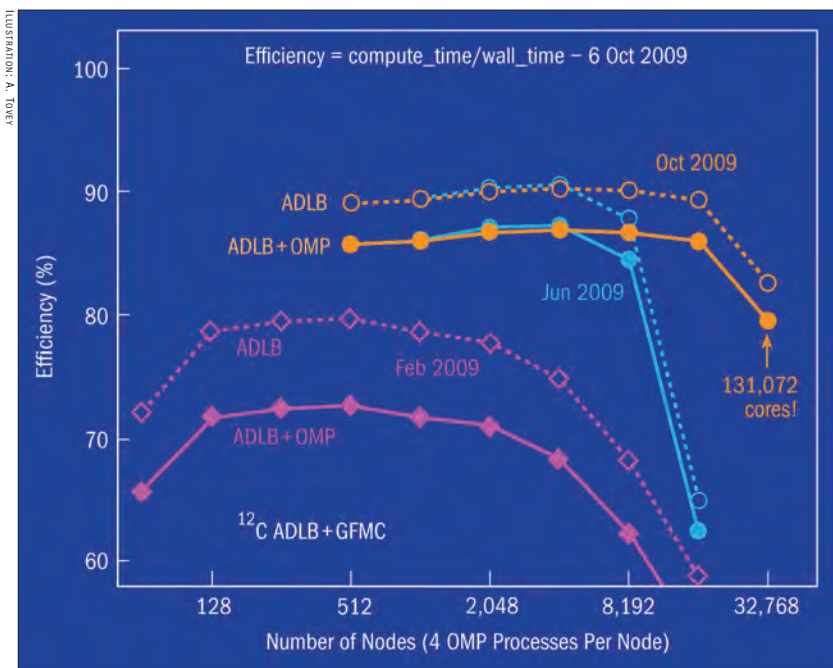


Figure 8. Scalability evolving over time.

the scalability and complexity of ADLB is that this information can always be slightly out of date.

As the clients of a server create work and deposit it on the server, that server may run out of memory. As memory becomes scarce, the server, using the information in the status vector, sends work to other servers that have more memory available. It can also tell a client to send work directly to a different server. This process is known as “pushing work.”

A significant amount of message passing, both between clients and servers and among servers, occurs without involvement from the application. The ADLB implementation uses a wide range of MPI functionality in order to obtain performance and scalability.

Scaling Up: The Story

As the project started, GFMC was running on 2,048 processes. The first step in scaling up was the conversion of some of the application code ADLB. This change was small, because the abstract master/worker programming model remained. A more substantial change was inserting ADLB calls to parcel work (that had all been done on one worker) to multiple workers and then collecting the results. This was achieved typically by replacing a loop that computed multiple results with two loops: the first loop does the work of ADLB_Put, and the second gathers the results. To avoid deadlocks, the second loop also accepts work packages of the same type, so at least one worker will always be available for processing of packages. After these changes, the application interface (figure 4, p33) remained stable while ADLB was being developed.

The ADLB implementation had to evolve to handle more and more client processes and to deal with multiple types of load balancing.

As GFMC was applied to carbon-12, the work units became too big for the memory of a single ADLB/MPI process, which was limited to one-fourth of the 2 Gb memory on a 4-core BG/P node. The solution was to adopt a hybrid programming model in which a single multithreaded ADLB/MPI process was run on each node of the machine, making all of each node’s memory available. This was accomplished by using the Fortran version of OpenMP for shared-memory parallelism within a single address space, which needed to be done only in the computationally intensive subroutines of the GFMC program. In some cases this could not be done because different iterations of the DO loop accumulate into the same array element. In such cases an extra dimension was added to the result array so that each thread could accumulated into its own copy of the array. At the end of the loop, these four copies were added together. OpenMP on BG/P proved to be very efficient; using four threads (cores) increases speed by a factor of 3.8 compared to using just one thread (95% efficiency). These changes were independent of using ADLB: even in a pure MPI implementation they would have been necessary.

Meanwhile, the ADLB implementation had to evolve to handle more and more client processes and to deal with multiple types of load balancing. At first the focus was on balancing the CPU load, the traditional target of master/worker algorithms. Next, it was necessary to balance the memory load, first by pushing work from one server to another when the first server approached its memory limit, and then by keeping memory usage balanced across the servers incrementally as work was created.

The last hurdle was balancing the message traffic. Beyond about 10,000 processors, the unevenness in the pattern of message traffic caused some servers to accumulate huge queues of unprocessed messages, making parts of the system unresponsive for large periods of time, reducing overall scalability. This problem was solved by balancing multiple requests between two servers for the same type of work.

Excellent Results with Full Carbon-12 Calculations

Prior to this work, a few calculations of the carbon-12 ground state had been made using the old GFMC program and merely hundreds of processors. These were necessarily simplified calculations with significant approximations that produced questionable results. The combination of BG/P and the ADLB version of the program has completely changed this situation. We first made one very large calculation of the carbon-12 ground state using our best interaction model and none of the previously necessary approximations. This calculation was made with an extra large number of Monte Carlo samples to allow

various tests of convergence of several aspects of the GFMC method. These tests showed that our standard GFMC calculation parameters for lighter nuclei were also good for carbon-12.

The results of the calculation were also very encouraging. The computed binding energy of carbon-12 is 93.2 MeV with a Monte Carlo statistical error of 0.6 MeV. Based on the convergence tests and our experience with GFMC, we estimate the systematic errors at approximately 1 MeV. Thus, this result is in excellent agreement with the experimental binding energy of 92.16 MeV. The computed density profile is shown in figure 9. The red dots show the GFMC calculation while the yellow curve is the experimental result, demonstrating good agreement between the results.

Full carbon-12 calculations are now possible on 8,192 BG/P nodes in two 10-hour runs. Such speed allows us to consider how to compute other interesting carbon-12 states and properties. An example is the first 2^+ excited state and its decay to the ground state, a decay that has proven to be a problem for other methods to compute.

Addressing the Next Challenges

The research challenge for the area of programming models for high-performance computing is to find a programming model that

- is sufficiently general that any parallel algorithm can be expressed
- provides access to the performance and scalability of HPC hardware
- is convenient to use in application code

It has proven difficult to achieve all three. This project has explored what can be accomplished by weakening the first requirement. ADLB is principally for algorithms in which the parallelism comes from having mostly independent processes accessing a shared pool of heterogeneous work units rather than explicitly communicating with one another. But within this limitation, it provides high performance in a portable way on large-scale machines via a much simpler method than having the application code implement the parallelism directly in MPI calls. Other applications of ADLB are currently being explored.

Although current performance and scalability are now satisfactory for production runs of states of carbon-12, challenges remain as one looks toward the next generation of machines and more complex nuclei such as oxygen-16 (^{16}O). New approaches to ADLB implementation may be needed. But because of the fixed application program interface for ADLB, the GFMC application itself may not have to undergo major modification. Two approaches are being

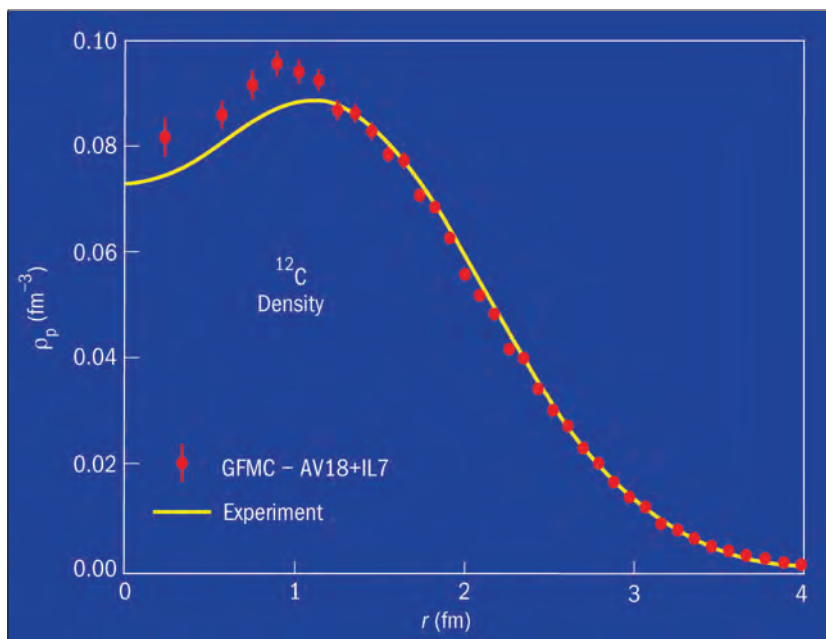


Figure 9. The computed (red dots) and experimental (yellow curve) density of the carbon-12 nucleus.

explored. The first is the use of the MPI one-sided operations, which will enable a significant reduction in the number of servers as the client processes take on the job of storing work units instead of putting them on the servers. The second approach, motivated by the potential need for more memory per work unit than is available on a single node, is to find a way to create shared address spaces across multiple nodes. A likely candidate for this approach is one of the PGAS languages, such as UPC (Unified Parallel C, see p22) or Co-Array Fortran. •

Contributors Dr. Ewing Lusk, Mathematics and Computer Science Division, ANL; Dr. Steven Pieper, Physics Division, ANL; Dr. Ralph Butler, Computer Science Department, Middle Tennessee University

Acknowledgements The work here relies heavily both on the U.S. Department of Energy's SciDAC program, for funding this collaboration between physicists and computer scientists, and on DOE's INCITE program for providing the computer time to obtain the results.

Further Reading

S. C. Pieper and R. B. Wiringa. 2001. Quantum Monte Carlo Calculations of Light Nuclei. *Annual Review of Nuclear and Particle Science* **51**: 53–90.

Scientific Grand Challenges: Forefront Questions in Nuclear Science and the Role of High-Performance Computing (a report from the workshop held January 26–28, 2009)

ADLB

<http://www.cs.mtsu.edu/~rbutler/adlb>

UNEDF

<http://www.unedf.org>