Early Experience with Scientific Programs on the Cray MTA-2

Wendell Anderson
Naval Research Laboratory

Preston Briggs
Cray, Inc.

C. Stephen Hellberg
Naval Research Laboratory

Daryl W. Hess
National Science Foundation

Alexei Khokhlov
University of Chicago

Marco Lanzagorta
Scientific & Engineering Solutions

Robert Rosenberg
Naval Research Laboratory

Abstract

We describe our experiences porting and tuning three scientific programs to the Cray MTA-2, paying particular attention to the problems posed by I/O. We have measured the performance of each of the programs over many different machine configurations and we report on the scalability of each program. In addition, we compare the performance of the MTA with that of an SGI Origin running all three programs.

1 Introduction

The Naval Research Laboratory has recently installed a 40-processor Cray MTA-2, the largest MTA assembled to date. The MTA, with its large, uniformly accessible memory, lightweight synchronization, and strong compilers, should provide a useful platform for large, scientific programs. We describe our experiences porting and tuning three such programs.

2 The Programs

When the MTA was installed at NRL, a few programs were selected for initial evaluation. Programs were chosen because they looked as though they might run well on the MTA *and* they did not run well on other machines.

These reasons may appear facetious; they are not. Based on our understanding of the architecture and the published experience with the Tera MTA (*e.g.*, [6, 10, 13]), we looked for programs that seemed to have plenty of inherent parallelism combined with plenty of relatively irregular data access.

SC'03, November 15–21, 2003, Phoenix, Arizona, USA

The second reason is perhaps more important. The MTA is a new architecture, created to solve problems not well addressed by other machines. If a program is running well on another machine, why go to the effort of porting it? Our intent was to attack real problems that had no other adequate solution, versus simply studying yet another computer architecture.

2.1 Exploring High Temperature Superconductors

The first application ported to the MTA was a research program called Flux [7], used to study high temperature superconductors.

The electronic excitation spectrum and the properties of the superconducting state are calculated for a simplified model of a high temperature superconductor. Experiments indicate the existence of an unusual metallic state in the high temperature superconductors that gives way to the superconducting state with decreasing temperature. A hallmark of this state is that, like the superconducting state, the metallic state exhibits a strong suppression of low-energy electronic excitations.

A possible physical explanation is that the metallic state resembles the superconducting state, but it is not a true superconducting state – superconductivity is "scrambled" by thermal and quantum mechanical fluctuations enhanced by the two-dimensional nature of the copper-oxide planes. Unlike ordinary superconductors where electrons bind together to form pairs at the phase transition to superconductivity, electrons form pairs at temperatures well above the transition to superconductivity. This physics is not contained in the standard theory of superconductivity, so a more sophisticated theoretical approach, a propagator functional theory, is used to calculate temperature dependent electronic excitation spectra and superconducting densities.

The nonlinear equations of the propagator functional theory, Dyson's equation and an equation for the self-

energy, are solved using an iteration algorithm suitable for parallel computers. FFTs are used to transform between position-imaginary time space and crystal momentum-imaginary frequency space. A frequency conditioning method is used to remove unphysical artifacts of the FFT procedure and to ensure that the solution has self-consistent properties expected from the formal analytical theory upon which the propagator functional theory is based.

The program, Flux, has been ported and rewritten several times during its career. It was originally written in CM Fortran for the CM-200 and structured to take advantage of the Connection Machine's FFT libraries. Later, it was ported to the CM-5, the Cray T3E, and the SGI Origin. When we began to port it to the MTA, it was written in a mixture of Fortran 77 (the 1D FFT) and HPF (everything else). Interestingly, the HPF code was automatically generated from the earlier version written in CM Fortran. Array syntax was used extensively, typically manipulating `complex` matrices with four dimensions. It required about 7000 lines of HPF, with minimal comments.

2.2 Modeling Strong Point Explosions

One early application ported to the MTA-2 was Alla, a hydrodynamical code used to model single point explosions. Early work with Alla was for the modeling of the explosions of supernovas. The code uses a high quality Godunov-type, finite-volume, explicit integration algorithm that is second-order accurate in space and in time, and uses a Fully Threaded Tree-based local mesh refinement for obtaining highly resolved solutions. Local mesh refinement is automatic and performed on the level of individual cells. The mesh is refined around shocks, fronts and regions of large gradients. It was written in a combination of Fortran 77 (for all of the computation), C (for I/O), and Fortran 90 (for its module facility).

The application, Alla, is written on top of a special library, FTT (Fully Threaded Tree), designed to support numerical applications requiring adaptive refinement of regular meshes [9]. The library supports data-parallel operations across its cells and the application achieves all of its parallelism through these operations. Using this approach, porting the entire application to a new shared-memory machine is largely a matter of rewriting one routine in the library. On the other hand, parallelizing FTT using MPI required two years of effort and expanded the size of the FTT library by a factor of nine.

2.3 Analyzing Low Temperature Quantum Systems

The third application, Lanczos, is used to support a study of the behavior of the charged ordered phase of manganites [8]. The manganites are a class of material with a general chemical formula of $(\mathrm{R,A})_{n+1}\mathrm{Mn}_n\mathrm{O}_{3n}$, the so-called Ruddelsden-Popper series [12]. The manganites exhibit a large variety of phases including ferro-magnetism, anti-ferro-magnetism, and charged-ordered (CO) phases. Where it exists, the CO phase is generally the lowest temperature phase; but, in some manganites, this phase has been seen to melt as the temperature is decreased further. Theoretically, this behavior has been obtained by using an extended Hubbard model to describe the material. The analysis of the properties of these materials involves the determination of the lowest eigenvalues of hundreds of square sparse matrices that are either real symmetric or complex Hermitian, some with dimensions greater than ten million [3].

Techniques developed at NRL to solve this problem use a Lanczos algorithm to find the lowest (and highest) eigenvalues of each matrix. The algorithm starts with a normalized random vector and generates a series of orthogonal vectors by multiplying the current vector by the Hamiltonian M and othogonalizing:

$$v_{n+1} = Mv_n - a_n v_n - b_n v_{n-1}$$

where

$$a_n = \frac{v_n \cdot M \cdot v_n}{v_n \cdot v_n}$$

and

$$b_n^2 = \frac{v_n \cdot v_n}{v_{n-1} \cdot v_{n-1}}$$

The a and b values are the diagonal and off-diagonal elements of a tridiagonal matrix whose lowest (and highest) eigenvalues approximate those of the original matrix. Four thousand (a, b) pairs are sufficient to estimate the eigenvalues of interest of the original NxN matrix. A standard LAPACK routine may be used to quickly find the eigenvalues of the tridiagonal matrix.

Because most of the calculations of the Lanczos coefficients occur in the matrix-vector multiplication, the algorithm is particularly efficient with large sparse matrices, such as Hubbard Hamiltonians. In theory, if the above algorithm was repeated to produce an NxN tridiagonal matrix and all of the calculations were exact, the set of eigenvalues for the two matrices would be the same. In practice, however, round-off errors occur and spurious copies of some of the eigenvalues are generated. These are removed during the post-processing of the tridiagonal matrix.

Lanczos was originally implemented in C++ using MPI. The program was organized to generate the elements of one of the irreducible matrices, with each processor responsible for generating the coefficients for a set of rows of the matrix. The program then calculates the Lanczos coefficients a_i and b_i by having each processor perform the part of the sparse matrix-vector product for the rows of the matrix stored in that processor. This requires that

each processor have the entire dense vector and that all processors be updated at each step with the new values of the v_{i+1} vectors. When all of the a_i and b_i coefficients have been calculated for the matrix, they are saved to a disk file and later transferred to a PC where the eigenvalues of the tridiagonal matrix are determined. The code has been ported to the IBM SP2 and SP3, the SGI Origin 2000 and 3000, and the Compaq ES45 and GS320.

This implementation has two aspects that limit scalability. First, each processor holds the entire vector, so no matter how many processors are employed, the total memory required on each processor increases with the dimension of the matrix. Second, at each Lanczos step, the portion of the vector updated by a processor must be communicated to every other processor. Therefore, the amount of data that must be sent to other processors is proportional to the size dimension of the matrix and is not reduced as additional processors are used.

3 The Computer

The Cray MTA-2 is the second generation of the Tera MTA [2]. It shares the same architecture and programming model [1], but benefits from improved mechanical design, packaging, connector technology, *etc.* In particular, the processors are much more highly integrated, requiring a single CMOS part versus 21 GaAs parts, with concomitant savings in power, cooling, footprint, and overall expense. Since the new machine implements the same architecture and instruction set, the same software tools are available, albeit expanded and more mature. Key features of the MTA include:

- Proprietary, multi-threaded processors,
- Shared, uniformly accessible memory, and
- Abundant, lightweight synchronization via full-empty bits.

Users interact with the machine through I/O nodes which host the software tools (compilers, debuggers, *etc.*) and support transparent execution of MTA executables.

3.1 The Processors

The MTA's processors are *multithreaded*. Each processor has 128 register sets, including 128 program counters, supporting up to 128 streams in hardware. A processor switches between ready streams at every clock tick, in an approximately round-robin fashion. This multithreading provides *latency tolerance*, allowing the processor to maintain 100% utilization in spite of distant memory references.

At every clock cycle, a processor issues one instruction from any one of many ready threads. Thus, when one thread requests a value from memory and is thereby delayed, instructions from other threads may execute, utilizing the processor during the interim. Moreover, each memory operation has an associated lookahead number that tells the processor how many more instructions it may execute from the same steam prior to the memory operation completing, thus providing even greater latency tolerance. Besides tolerating latency to memory, the multithreaded processors are able to tolerate branch latency, instruction-cache misses, synchronization delays, *etc.*

To the programmer (or compiler), each stream appears to be a fairly ordinary RISC processor. A stream has 32 general-purpose registers, each 64 bits long. Three operations are packed into each 64-bit instruction, providing a modicum of instruction-level parallelism. Additionally, each memory reference has a 3-bit *lookahead* field, as mentioned above, providing even more more ILP. The hardware directly supports 64-bit IEEE floating-point arithmetic, including NaNs and denorms, and 64-bit integer arithmetic.

3.2 The Memory

Since the processors tolerate memory latency, there are no data caches. Instead, there is a flat, uniformly accessible, shared memory physically distributed around the machine's internal network. The high-bandwidth network precludes unexpected memory-traffic bottlenecks. Every processor can issue a memory reference at every clock cycle without risk of network or memory congestion. The memory is byte addressed, with 8-byte words. Addresses are *hashed* in the processors to spread accesses throughout the system. The flat, uniform-access shared memory frees the programmer (and compiler) from all considerations of data placement, stride, *etc.*

3.3 Synchronization

The MTA provides four *state bits* associated with each memory word: a forwarding bit, a full-empty bit, and two data-trap bits. The full-empty bits are used by the compiler (and occasionally by the programmer) for synchronization, via memory references.

- Normal loads and stores ignore the full-empty bit.
- *Sync loads* wait for the full-empty bit to be set, then read the location and clear the bit, atomically.
- *Sync stores* wait for the full-empty bit to be clear, then write the location and set the bit, atomically.

Normal and sync operations all require 1 issue. Importantly, the processor does not stall when a stream must wait; synchronization delays are tolerated like all other forms of latency.

The architecture also provides an atomic *fetch-and-add* operation. The fetch-and-add and the sync memory operations are used frequently by the compiler and are available to the programmer through data declarations and intrinsics.

3.4 The NRL Machine

The NRL machine has 40 processors running at 200 MHz and 160 Gbytes of RAM. It was installed in early 2002 and accepted in September after extensive tests. As a part of the acceptance tests, Cray was required to demonstrate a price/performance advantage on several applications, including Flux and Alla, compared to the SGI Origin 3800 already installed at NRL.

The NRL machine is not (yet) a perfect implementation of the MTA architecture. It suffers from an inadequate internal network, limiting the bandwidth to approximately 50% of the intended peak. This limitation is reflected in poor scaling of memory-intensive programs when using more than about 24 processors. Despite this flaw, the network has enough bandwidth to enable the machine to meet its performance criteria. Cray is currently working to correct the problem.

4 Porting

All of the programs required a fair amount of work to port to the MTA. Flux was the most straightforward, requiring minimal interaction with the user. The other two required more interaction and some redesign [4].

4.1 Flux

Since Flux was written in HPF, some work was required before we could even attempt to compile the code. HPF is basically Fortran 95, extended with data distribution directives. The directives could simply be ignored, but we had to deal somehow with the `forall` statements, which are a part of Fortran 95 not yet supported by the MTA's compiler. Since there were too many to rewrite by hand (approximately 200), we wrote a special-purpose preprocessor to expand them into nested `DO` loops. Figure 1 shows a typical expansion.

These rewrites allowed us to compile and run the code. We note that the `forall` statements actually occurring in the code were very simple (as in the example) and did not require any deep analysis. Given that all the `forall` statements were so stylized, we speculate that they were automatically generated as part of the translation from CM Fortran.

After the rewrite, the Fortran 90 compiler was able to find plenty of parallelism, unsurprising given the number of `forall` statements in the original code along with the large amount of array syntax. The MTA's uniformly accessible memory meant that there was no stride sensitivity and no cache effects, so performance and scaling in the computational part was quite good. The I/O required more work (see Section 4.4).

A trouble spot on other machines centered around the 4-D FFTs. The original (HPF version) code used a sequence of triply nested `DO` loops, each containing a call to a simple 1-D FFT, apparently from *Numerical Recipes* [11]. All of the `DO` were flagged as parallel using the HPF `INDEPENDENT` directive, which we simply replaced with appropriate MTA directive. This was enough to achieve good performance on the MTA, but closer examination showed a significant performance problem on cache-dependent machines: In some situations, the 1-D FFT was being invoked with vectors of long stride (sometimes over 8 Mbytes). Copying the data and padding the arrays helped mitigate the problem on the SGI Origin.

4.2 Alla

Alla posed several problems. It was written in a particularly disciplined style, with all parallelism expressed explicitly via calls to a single routine, `ftt_doall`.

Consider the code excerpts in the left column of Figure 2. At the top, we see a call to `ftt_doall`, passing a pointer to the subroutine `TimeStepH_Work` (the other arguments specify a section of the adaptive mesh). The routine `ftt_doall` contains a single `do` loop that iterates over the blocks of the mesh, in parallel, passing each block to the procedure parameter, `Work`. The intent of the program's author was that porting would require little more than parallelizing this loop, using whatever scheme the target machine supported. For the MTA, we simply asserted the loop parallel and let the compiler parallelize it using a more scalable algorithm.

Code written in this style, running on the MTA, will not require any sort of dynamic load balancing as the mesh is refined. Instead of a fixed distribution of work among the processors that persists across the lifetime of the program, the running program will "go parallel" at the beginning of each parallel loop, distributing the iterations of that loop among threads running on all the available processors. At the end of the parallel loop, all the parallelism is collapsed to a single serial thread. This process repeats at each parallel region (which may include several loops); therefore, refinements to the mesh will automatically be reflected in new work distributions at each loop.

The application code (not the FTT library) contained a number of sections requiring synchronized updates, generally used to support reductions. Our initial approach was to use "update directives" to force the compiler to protect the locations during update using the MTA's full-empty bits. The code for `TimeStepH_Work` (on the left-hand side of Figure 2) illustrates the necessary directive.

At the top of the routine, a directive prevents the compiler from attempting to parallelize the routine. We use

```
      forall (l = 0:m1:1, k1 = 0:nx1:1, k2 = 0:ny1:1, k3 = 0:nz1:1)
        sigma(l, k1, k2, k3) = sigma(l, k1, k2, k3) -
                               2.0d0*tz*dcos(dble(k3)*delta_kz)*d_t3_r1z*qr_epsilon(l)


      do l = 0, m1
        do k1 = 0, nx1
          do k2 = 0, ny1
            do k3 = 0, nz1
              sigma(l, k1, k2, k3) = sigma(l, k1, k2, k3) -
                                     2.0d0*tz*cos(dble(k3)*delta_kz)*d_t3_r1z*qr_epsilon(l)
            end do
          end do
        end do
      end do
```

Figure 1: Rewriting simple `forall` loops

```
      ...
      call ftt_doall(0, 1, TimeStepH_Work)
      ...


      subroutine ftt_doall(L, K, Work)
      ib = lSCOff(L, K)
      ie = lSCOff(L, K) - 1
c$mta assert parallel
      do i = ib, ie, mind
        i1 = min(i + mind - 1, ie)
        n1 = i1 - i + 1
        call Work(n1, lSC(i))
      end do
      end


c$mta serial
      subroutine TimeStepH_Work(n, index)
      vas1 = 0.0
      do i = 1, n
        ip = index(i)
        rhor = 1.0/var(1, ip)
        as = sqrt(var(6, ip)*var(7, ip)*rhor)
        vel = max(abs(var(3, ip),
     .            abs(var(4, ip),
     .            abs(var(5, ip))*rhor
        vas1 = max(vas1, as + vel)
      end do
c$mta update
      vas = max(vas, vas1)
      end
```

```
      ...
      call mta_doall(0, 1, TimeStepH_Work)
      ...


      subroutine mta_doall(L, K, Work)
      ib = lSCOff(L, K)
      ie = lSCOff(L, K) - 1
      n1 = ie - ib + 1
      call Work(n1, lSC(ib))
      end


      subroutine TimeStepH_Work(n, index)
      vas1 = 0.0
      do i = 1, n
        ip = index(i)
        rhor = 1.0/var(1, ip)
        as = sqrt(var(6, ip)*var(7, ip)*rhor)
        vel = max(abs(var(3, ip),
     .            abs(var(4, ip),
     .            abs(var(5, ip))*rhor
        vas1 = max(vas1, as + vel)
      end do
      vas = vas1
      end
```

Figure 2: Specifying Parallelism in Alla

this since we know that this routine is always called in a parallel context and any additional parallelization would imply unnecessary overhead. The loop computes a reduction into `vas1`, a local variable, and the last line updates the global variable `vas`. The update directive causes the compiler to issue a *sync* load to access `vas` and leave the location *empty*. After computing the `max`, it uses a *sync* store to set the location *full*. The effect is to achieve an atomic update, but without OS or runtime overhead.

This approach yielded safe, parallel code in keeping with the original programmer's intent. Later, when scaling to larger configurations, even these sorts of lightweight synchronization became problematic and portions of the code were rewritten to expose reductions to the compiler. The code in the right column of Figure 2 illustrates the required changes. Here, we have modified the code (versus simply adding directives), creating a new routine, `mta_doall`. This new routine has the same behavior as `ftt_doall`, but relies on a single invocation of `Work`. The worker routine, `TimeStepH_Work`, is simplified slightly and relies on the compiler to recognize the reduction expressed in the `do` loop. The compiler's approach to solving reduction scales without hot spots and is therefore preferred.

Another interesting performance problem was exposed during early testing. The FTT library managed its free storage (a vector of cells) as a stack, linking together free cells. To ensure safety, accesses to the head of the list were synchronized using the associated full-empty bit. However, this location became a hot spot as all the threads (thousands of them) attempted to allocate and free large numbers of cells. To cure the problem, an MTA-specific high-bandwidth queue was implemented to manage the free cells.

Finally, a fair amount of work was required to handle the various I/O problems (see Section 4.4).

4.3 Lanczos

Two factors drove the decisions on how to use the MTA to solve the problem. The first was the desire to use the flat, shared memory and multithreading paradigm to reduce the time required to compute the tridiagonal matrix. The second was to minimize the amount of recoding. Analysis of the problem indicated that the bulk of the time (over 95%) would be spent in the calculation of the Lanczos coefficients, an operation that was just a small part of the original C++ code.

Therefore, we decided to use the current code to generate the matrices and write a new, MTA-specific program to calculate the Lanczos coefficients. The matrices are stored in a modified compressed row storage (CRS) format using three vectors, `R`, `I`, `J`. The code for the calculation is quite simple, taking only 14 lines of Fortran 90, with the first 6 lines devoted to the sparse matrix-vector multiply.

```
do indx = 1, Imax
  Y(indx) = 0
  do indx2 = I(indx) + 1, I(indx + 1)
    Y(indx) = Y(indx) + R(indx2)*Vvec(J(indx2))
  end do
end do
Aval(iteration) = dot_product(Y, Vvec)
Uvec = Y + Aval(iteration)*Vvec + Uvec
Bval(iteration) = sqrt(dot_product(Uvec, Uvec))
do indx = 1, Imax
  tmp = Vvec(indx)
  Vvec(indx) = Uvec(indx)/Bval(iteration)
  Uvec(indx) = -tmp*Bval(iteration)
end do
```

The code to determine the Lanczos coefficients is the same whether the original matrix is real symmetric or complex Hermitian. The only difference in the programs for the two cases is whether the variables `Y`, `R`, `Uvec`, `Vvec`, and `tmp` are declared as real or complex. Note that `Aval` and `Bval` are always real. Also, notice that there are no MTA-specific routines or directives in the code. Therefore, this code can run on any machine that has a Fortran 90 compiler. In fact, this same kernel is also running on the NRL's Origin 3800 (the only change required was in the code to read the input matrix from a disk file).

In this code, all the `DO` loops are parallelized, along with the dot products and the array assignment to `Uvec`. Most of the time in the kernel is spent computing the matrix-vector product (the doubly-nested loop). Analysis with the MTA's `canal` utility indicates that the inner loop requires 3 instructions per iteration when the matrix is real, including 3 memory references and 1 floating-point multiply-add.[1] When the matrix is complex, there are 7 instructions per iteration, with 5 memory references and 4 floating-point multiply-adds.

4.4 I/O

I/O performance is often a problem when porting applications to the MTA. The bottleneck is usually not the time required to get data on and off the disk (striping works well here); instead, it is the time to format the data, converting between the internal binary format and the required external format. The conversion is almost always written as a serial process. This problem also plagues other parallel machines, but since their serial performance is better than that of an MTA, it can often be ignored. For code ported to the MTA, it is often key.

For output, we use the general approach of allocating adequate memory to hold the entire data file, then formatting (in parallel) all the data into the buffer, and fi-

[1] Recall that the MTA architecture packs 3 operations into each instruction.

nally writing the entire buffer in a single system call.[2] Input is similar, in that we allocate enough memory to hold the entire data file, then read the entire file with a single system call, and finally convert all the data (in parallel) into binary. In cases where the data files are particularly large, we block the data into reasonably sized chunks, perhaps tens of gigabytes. Straightforward application of this technique sufficed for Flux; Alla was more challenging.

4.4.1 Flux

The main I/O problem for Flux was reading and writing the checkpoint files, one sixteenth of a four-dimensional array of `complex` values. The original output code looked like this:

```
open(unit=1, file=io_filename, status='unknown')
do j = 0, m/2
 do k1 = 0, nx/2
  do k2 = 0, ny/2
   do k3 = 0, nz/2
    write(unit=1, fmt=*) cmarray(j, k1, k2, k3)
   end do
  end do
 end do
end do
close(unit = 1)
```

The input code was similar, with an extra series of tricky (but parallel) array assignments to unfold the values along each axis.

This sort of loop nest will not be parallelized by the compiler, since doing so would surely change the order of values written to the file. Therefore, we manually rewrote the output code, first copying the required data in a temporary array of the appropriate size.

```
do j = 0, m/2
 do k1 = 0, nx/2
  do k2 = 0, ny/2
   do k3 = 0, nz/2
    temp(k3, k2, k1, j) = cmarray(j, k1, k2, k3)
   end do
  end do
 end do
end do
```

Note that we have rearranged the indices of `temp` (compared with `cmarray`), so that it may be treated as a simple vector of `complex` values, arranged in the desired order for output. The compiler parallelizes all four loops, with the innermost loop requiring 4 instructions per iteration (2 loads and 2 stores for each `complex` value). We pass `temp` to a C routine which allocates a buffer large enough to hold all the formatted values and formats the entire contents of `temp` in a single parallel loop, like this:

```
#pragma mta assert parallel
  for (i = 0; i < n; i++) {
    sprintf(buf + RECORD_LENGTH*i,
            " (%23.16E,%23.16E)",
            v[2*i], v[2*i + 1]);
    buf[RECORD_LENGTH*(i + 1) - 1] = '\n';
  }
```

Since C has no `complex` values, we treat the vector as a vector of `double`. The `sprintf` call formats the real and imaginary parts into the appropriate place in the buffer, terminating the string with a `NULL`. The assignment statement overwrites each `NULL` with an `EOL`.

Having formatted the data, we write the buffer to disk with a single call to Unix `write`, free the temporary storage, and return. Input is structured in a similar fashion, with the slight additional difficulty that we must search for record boundaries (`EOL` characters).

While this all seems somewhat baroque, it works quite well. Unfortunately, we have had to do this sort of thing over and over again, in one form or another. On the other hand, having gone to the effort to parallelize the formatting code, it is now *very* fast and the performance scales with the number of processors.

Of course, it would have been easier and faster to use an unformatted checkpoint file. In this case, we could have simply copied the data into `temp`, as above, and then written the entire `temp` array to disk in a single call to `write`. However, such an approach yields checkpoint files that cannot necessarily be restarted on another machine.

4.4.2 Alla

Alla writes both checkpoint files and a large number of data files (used to drive a visualization tool). The checkpoint files were relatively easy, being unformatted, but the sheer number of them proved an impediment to performance. The default was to write a checkpoint every 50 time steps, which was fine with a few processors. As we added more processors, the computational time shrank to the point that the time required to create and write the checkpoint files became significant.

The usual response would be to write fewer checkpoint files; after all, a checkpoint is intended to help avoid significant loss of computation in the event of a crash. A reasonable approach might be to write one per hour, say, so that no more than an hour of work is ever lost in a crash. In this case, we were constrained by the acceptance criteria to produce a defined output, including checkpoint files.

The visualization files were even harder: They required formatting and there were a lot of them. The formatting seemed exceptionally difficult to parallelize, since the various `fprintf` calls to any particular file were spread over several routines.

[2]Since the NRL machine has 160 Gbytes of RAM, this sort of approach is feasible.

We attacked both problems the same way, by making copies of all required data and spinning off threads (using `future` statements [5]) to perform the necessary formatting and output in an asynchronous fashion. To ensure that the checkpoints were useful, care was taken so that each checkpoint was completed before the next was begun. Of course, this approach can consume a fair amount of memory, but what is memory for if not to let our programs run faster? In this fashion, very little code had to be changed and the bulk of the I/O was moved off the critical path.

4.4.3 Lanczos

Unlike Flux and Alla, the Lanczos code was not a port of existing code from another machine, but rather an entirely new program. Thus we had the freedom to choose the formats of the input and output files. In the case of Lanczos, the input data is a very large file (several gigabytes) containing the matrix of interest stored in a modified CSR format, while the output is a file containing the a and b coefficients (several thousand) that must be readable on a PC. By choosing to save and read the input matrix as a binary file, we were able to avoid the formatting steps. For the output file, the choice was formatted ASCII. While implemented in a serial fashion, the small size of the file allowed us to write the data in a negligible amount of time.

5 Performance

Two aspect of performance seem interesting: comparisons with other machines of similar capability and scaling studies, comparing the performance of different configurations of the MTA. We have been able to perform fairly complete scaling studies along with some limited comparisons to other machines.

5.1 Flux

The performance of the Flux code was measured on three different platforms, over a range of machine configurations. In each case, we report elapsed wall clock times, in seconds, for a $128 \times 128 \times 512$ grid (a large problem). The times are reported for complete runs, including all I/O.

In our initial tests, we compared the MTA running our tuned code to an SGI Origin 2000 running the HPF code.

procs	Origin 2000	MTA
1	13,118	6,091
4	8,998	1,708
8	4,540	912
16	3,646	495
32	6,046	305

When we first saw these results, we were all extremely happy (and somewhat surprised). After porting the Fortran 90 code back to the SGI (thinking that the problem might be due to the HPF compiler), we eventually recognized the difficulty posed by the giant strides required of the FFT routine. Reworking the SGI code to avoid cache difficulties and substituting the native FTT implementation gave more reasonable results.

Table 1 gives timing and scalability comparisons for the MTA and the SGI Origin 3800, each running their best code. The Origin is a shared-memory multiprocessor based on MIPS processors. It is a NUMA (non-uniform memory access) machine that depends on a hierarchy of data caches to help avoid long latency penalties for distant accesses. The 3800 has a 500 MHz clock, versus 200 MHz for the MTA; nevertheless, performance is quite close, at least for smaller configurations. For the larger cases (*i.e.*, more processors), the MTA exhibits better scaling. Given a fully functional internal network, one might hope for even better results from the MTA. Figure 3 presents the scaling results, graphically.

5.2 Alla

Table 2 shows the performance of Alla for one benchmark case on the MTA and an SGI Origin 3800. The benchmark case simulates a 3-D strong point explosion in a cubic box, which results in a rapidly evolving flow with multiple shocks, contact discontinuities, and vortices. The simulation runs for 1000 time steps, just long enough for the spherical shock front to hit the x-z and y-z walls and flatten out a bit. The simulation begins with 40K cells and completes with almost 90K cells.

Figure 4 illustrates scaling performance for each machine. While the performance on a single processor is nearly identical, despite the difference in clock rate, the MTA's superior scaling gives it a significant advantage in larger configurations.

5.3 Lanczos

Given the large size of the problem, it was not possible to time the entire problem on each of our machines. Since the number of operations required to generate a specific (a, b) coefficient pair is the same, it is sufficient to calculate the average time to generate such a pair. To compare the MTA with other DOD high performance computers, the largest complex matrix (12 million by 12 million, with an average of 33 elements per row) for the problem described in [3] was chosen and run on 4 separate computer systems. The average times to generate an (a, b) pair are given here.

platform	*clock*	16p	32p	*speedup*
IBM P3	375	8.57	6.70	1.28
Compaq ES45	1000	5.41	3.81	1.42
Origin 3800	400	9.51	9.62	0.99
MTA	200	1.69	0.99	1.71

The first column shows the machine, the second column gives the clock rate in MHz, the third column given the time (in seconds) required using 16 processors, the fourth column shows the time required using 32 processors, and the fifth column shows the apparent scaling as we move from 16 to 32 processors.

Another way to look at the results above is in terms of efficiency. If the program was running at 100% efficiency, then when run on 16 processors, we would expect 16 memory accesses per cycle. In fact, for the 16 processor case, we achieve 0.6, 0.4, 0.5, and 5.9 memory accesses per cycle for the IBM, Compaq, Origin, and MTA, respectively.

We had two other issues we wanted to examine: the performance of the original code (in C++ and MPI) and the scalability of the code. Since the ratio of memory references to floating-point operations differs between the `real` and the `complex` cases, we ended up running 6 sets of experiments (`real` and `complex` on the MTA, the Origin using Fortran 90, and the Origin using C++ with MPI). Table 3 gives the detailed results.

Several things should be noted. First, the results on the SGI, especially for the MPI implementation, are relatively noisy; the timings we give are actually the minimum of several runs. On the SGI, the Fortran 90 code is several times faster than the MPI code. In both cases, the performance only scales to about 10 processors. Second, the MTA code continues to scale well beyond 14 processors, although the scalability eventually degrades. This is a result of the inadequate internal network bandwidth mentioned earlier. As should be expected, the degradation is worse for the `real` case than for the `complex` case (see Figure 5) since the ratio of memory accesses to instructions is 1:1 versus 5:7.

An interesting variation of this point can be observed in the results for the MPI implementation on the SGI. Here, the `real` version scales better than the `complex` versions, presumably because more `real` values can be transmitted per second with a given amount of bandwidth.

6 Summary

We have described our experiences porting and tuning three scientific programs for the Cray MTA-2. The case of Flux was interesting, since it showcased the abilities of the MTA's architecture and compiler to exploit parallelism expressed via Fortran 90's array syntax. Alla posed the most difficult tuning task, since it required the detection and repair of several hot spots that only appeared when running with a large number of processors. Lanczos was perhaps the most rewarding, since it was a relatively straightforward kernel-like program that achieved tremendous speedups over other available machines. Furthermore, these speedups were achieved by NRL scientists working independently, without reliance on Cray specialists.

In all cases, the problems posed by formatted I/O had to be recognized and solved. While the MTA has plentiful and scalable I/O capabilities, the bottleneck posed by formatting requirements (conversions between ASCII and binary) can be challenging. We showed three different approaches to the problem:

- Parallelize the formatting code,
- Do the formatting in the background, or
- Define the file format to avoid the problem.

Two of the programs (Alla and Lanczos) were actually parts of larger efforts where data was processed by different programs on different machines, each contributing its particular strength to the overall result. This approach is perhaps a defining characteristic of large, scientific efforts. The thrust of the effort is always to obtain useful results as quickly as possible.

While the exact performance comparisons should be taken with a few grains of salt, several conclusions can be safely drawn. The performance of each of our programs on the MTA continues to improve as processors are added; furthermore, the rolloff in scaling will surely change for the better when the internal network is brought up to specification. Moreover, the importance of system-wide bandwidth, versus mere clock rate, seems well established. We believe the MTA's superior performance and scaling for these programs, especially Alla and Lanczos, can be attributed to the MTA's architectural characteristics:

- Cheap and flexible parallelism,
- Flat, shared memory, and
- Abundant, lightweight synchronization.

Finally, our experience also reminds us once again of the importance of competitive comparisons. We attacked the Flux code because it seemed to run poorly on the SGI compared with the CM-5. When the MTA's performance seemed too good to be true, code comparisons led to significant performance improvements on the SGI. Without the ability to compare across different machines, we are often unsure about a program's potential performance, especially with complex programs.

	Origin 3800		MTA	
procs	*secs*	*scaling*	*secs*	*scaling*
1	7,300		6,091	
2	4,532	1.61	3,167	1.92
4	3,261	2.24	1,708	3.57
8	1,231	5.93	912	6.68
12	884	8.26	545	11.18
16	577	12.65	495	12.31
20	473	15.43	386	15.78
24	487	14.99	364	16.73
28	416	17.55	322	18.92
32	385	18.96	305	19.97
36	407	17.94	293	20.79
40	430	16.98	284	21.45

Table 1: Flux Performance Comparisons

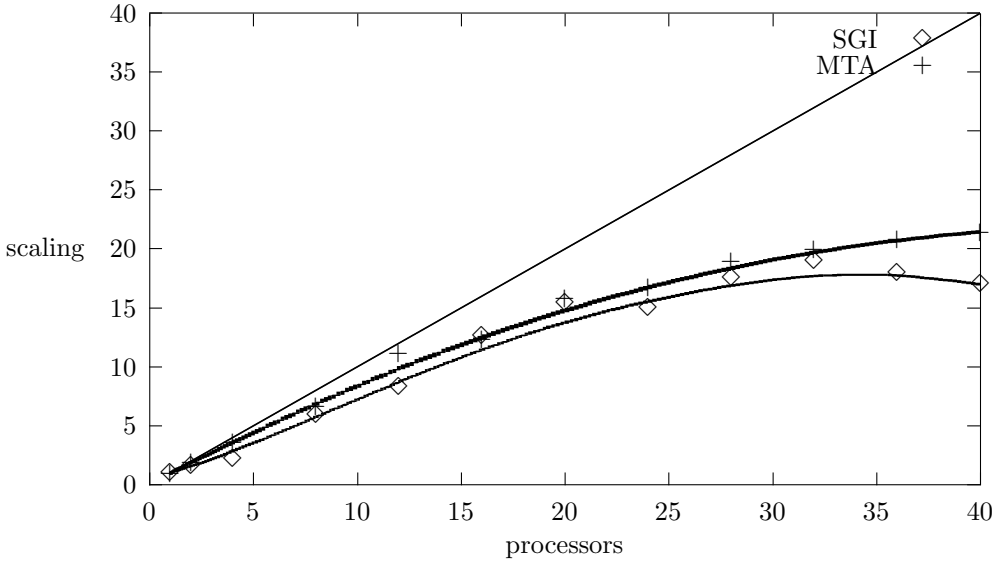

Figure 3: Flux Scaling Comparison

	Origin 3800		MTA	
procs	*secs*	*scaling*	*secs*	*scaling*
1	10,258		12,910	
2	6,244	1.64	6,342	2.04
4	4,682	2.19	3,126	4.13
8	3,466	2.96	1,711	7.55
12	2,616	3.92	1,346	9.59
16	2,394	4.28	954	13.53
20	2,965	3.46	764	16.90
24	2,513	4.08	633	20.39
28	2,466	4.16	537	24.04
32	2,652	3.87	521	24.78
36			551	23.43
40			415	31.11

Table 2: Alla Performance Comparisons

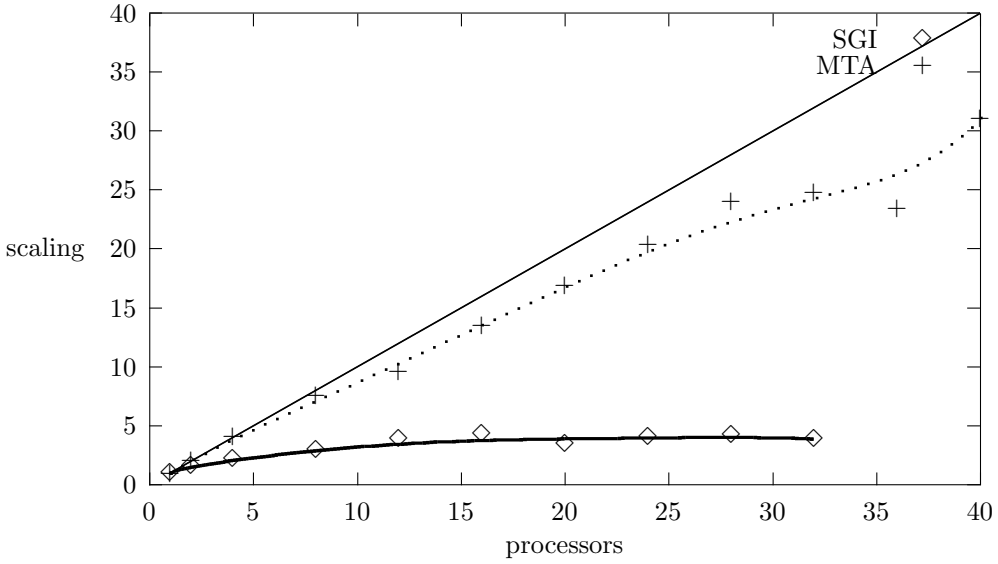

Figure 4: Alla Scaling Comparison

	Origin 3800 MPI				Origin 3800 Fortran				MTA Fortran			
	`complex`		`real`		`complex`		`real`		`complex`		`real`	
procs	*secs*	*scaling*	*secs*	*scaling*	*secs*	*scaling*	*secs*	*scaling*	*secs*	*scaling*	*secs*	*scaling*
1	47.62		32.10		46.62		19.40		24.97		9.76	
2	26.87	1.77	15.95	2.01	20.14	2.31	9.84	1.97	13.00	1.92	4.86	2.01
4	16.57	2.87	11.62	2.76	12.16	3.83	6.20	3.13	6.49	3.85	2.43	4.02
6	18.96	2.51	8.58	3.74	9.52	4.90	5.28	3.67	4.29	5.82	1.65	5.92
8	20.34	2.34	8.04	3.99	7.70	6.05	3.68	5.27	3.22	7.75	1.29	7.57
10	16.85	2.83	10.27	3.13	5.62	8.30	3.76	5.16	2.59	9.64	1.07	9.12
12	14.05	3.39	7.98	4.02	5.94	7.85	2.72	7.13	2.17	11.51	0.94	10.38
14	16.16	2.95	8.88	3.61	5.54	8.42	3.56	5.45	1.87	13.35	0.84	11.62
16	17.79	2.68	7.82	4.10	5.38	8.67	3.80	5.11	1.65	15.13	0.76	12.84
18	16.79	2.84	7.05	4.55	6.80	6.86	2.74	7.08	1.48	16.87	0.70	13.94
20	15.14	3.15	7.01	4.58	6.62	7.04	3.76	5.16	1.35	18.50	0.65	15.02
22	15.34	3.10	8.20	3.91	8.38	5.56	3.10	6.26	1.25	19.98	0.63	15.49
24	15.18	3.14	7.25	4.43	6.56	7.11	3.70	5.24	1.20	20.81	0.61	16.00
26	16.14	2.95	7.32	4.39	7.16	6.51	4.88	3.98	1.11	22.50	0.59	16.54
28	16.08	2.96	7.90	4.06	8.06	5.78	3.80	5.11	1.06	23.56	0.57	17.12
30	19.41	2.45	7.94	4.04	6.46	7.22	4.30	4.51	1.02	24.48	0.56	17.43
32	20.54	2.32	5.98	5.37	7.68	6.07	4.28	4.53	0.98	25.48	0.55	17.75
34	25.59	1.86	9.79	3.28	4.68	9.96	3.70	5.24	0.95	26.28	0.54	18.07
36	27.35	1.74	10.96	2.93	10.42	4.47	5.80	3.34	0.94	26.56	0.53	18.42
38	32.50	1.47	9.46	3.39	7.84	5.95	5.90	3.29	0.92	27.14	0.53	18.42
40	22.41	2.12	8.48	3.79	9.02	5.17	3.62	5.36	0.91	27.44	0.53	18.42

Table 3: Lanczos Performance Comparisons

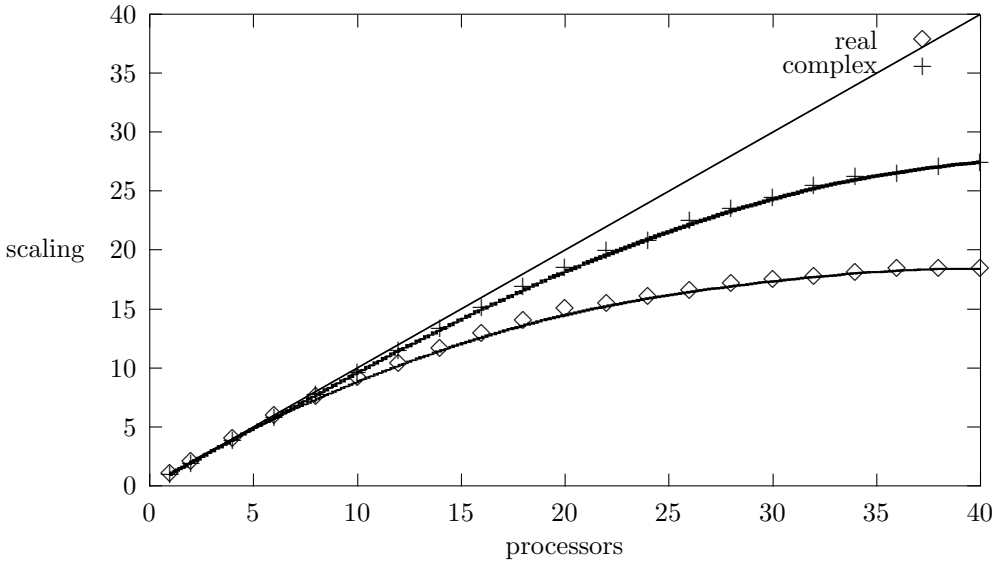

Figure 5: Lanczos Scaling Comparison

Acknowledgements

Our colleagues at NRL and Cray, including David Callahan, David Harper, Robert Henry, Jace Mogill, Jeanie Osburn, Rick Roberts, Wendy Thrash, and Greg Woodman, have all provided invaluable encouragement and support. In addition, comments from the program committee members enabled us to significantly improve our presentation.

Computations were performed on resources provided by the DOD High Performance Computer Modernization Program Office at the Aeronautical Systems Center, US Army Engineer Research and Development Center, and Naval Research Laboratory shared resource centers.

References

[1] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Proceedings of the 6th ACM International Conference on Supercomputing*, July 1992.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, pages 1–6, June 1990.

[3] Wendell Anderson, Marco Lanzagorta, and C. Stephen Hellberg. Analyzing quantum systems using the Cray MTA-2. In *Cray User Group Conference*, Columbus, Ohio, May 2003.

[4] Wendell Anderson, Marco Lanzagorta, and Robert Rosenberg. Experiences using the Cray Multi-Threaded Architecture (MTA-2). In *Department of Defense High Performance Computing Modernization Program Users Group Conference*, Bellevue, Washington, June 2003.

[5] David Callahan and Burton Smith. A future-based parallel language for a general-purpose highly-parallel computer. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 95–113. The MIT Press, Cambridge, Massachusetts, 1990.

[6] Larry Carter, John Feo, and Allan Snavely. Performance and programming experience on the Tera MTA. In *SIAM Conference on Parallel Processing*, San Antonio, March 1999.

[7] John J. Deisz, Daryl W. Hess, and Joseph W. Serenne. Phase diagram for the attractive Hubbard model in two dimensions in a conserving approximation. *Physical Review B*, 66(014539), 2002.

[8] C. Stephen Hellberg. Theory of the reentrant charge-order transition in the manganites. *Journal of Applied Physics*, 89:6627–6629, 2001.

[9] Alexei M. Khokhlov. Fully threaded tree algorithms for adaptive mesh fluid dynamics simulations. *Journal of Computational Physics*, 143(2):519–543, 1998.

[10] Leonid Oliker and Rupak Biswas. Parallelization of a dynamic unstructured application using three leading paradigms. In *Supercomputing*, Portland, November 1999.

[11] William H. Press, Saul A. Teuklosky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in Fortran 77: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.

[12] Myron B. Salamon and Marcelo Jaime. The physics of manganites: Structure and transport. *Reviews of Modern Physics*, 73:583–628, 2001.

[13] Allan Snavely, Larry Carter, Jay Boisseau, Amit Majumdar, Kang Su Gatlin, Nick Mitchell, John Feo, and Brian Koblenz. Multi-processor performance on the Tera MTA. In *Supercomputing*, Orlando, November 1998.