Effect of Tripling the Memory Bandwidth on the CRAY MTA-2

Wendell Anderson (Naval Research Laboratory) and
Marco Lanzagorta (Scientific and Engineering Solutions)

ABSTRACT: *In July 2002, the Naval Research Laboratory installed and began operating a 40-processor Cray Multi-Threaded Architecture (MTA-2) computer. In September 2003, the basic clock rate was increased from 200 MHz to 220 MHz and a top plane was added to the MTA-2 increasing the available memory bandwidth from 11 Gigabytes per second to almost 32 Gigabytes per second. In order to evaluate the actual performance for scientific simulations obtained on the upgraded MTA-2, two of the acceptance codes (Flux and Alla) and two production codes (Lanczos and Causal) were run on a range of processors and an analysis of the running times was performed. No changes were needed to any of these codes to see improvements in the performance of the computational intensive portion of the codes. The codes ran at least 1.1 times faster with some codes running 2.5 times faster.*

1. Introduction

In July 2002, the Naval Research Laboratory installed and began operation of a 40-processor 200 MHz Cray Multi-Threaded Architecture (MTA-2) computer [1]. The MTA-2 has a flat, shared memory with uniform access times from any processor to any memory location. Each processor has 128 RISC-like hardware threads with each thread containing its own instruction counter, status words, and registers and can initiate up to three floating point instructions per cycle. On each clock cycle, the processor can activate a different hardware stream. Each thread can have up to eight concurrent memory references. A processor along with an I/O processor and memory units with 4 Gbytes of SDRAM is contained on a system board. Each board is connected to the system interconnection via four routing nodes, allowing data transfers to and from the board at full processor bandwidth. Since the MTA-2 supports 64-bit data, addresses, and instructions, each processor can ideally move an 8-byte word on each cycle; hence, the MTA-2 had an aggregate potential of requesting 64 Gigabytes of data each second. However the MTA-2's internal network was only capable of supporting 35-45% of this data rate.

In order to better match the aggregate rate that the processors could request data and the bandwidth of the network, in September 2003 Cray installed a top plane for each of the processors to increase the network bandwidth to handle the data requests from the processors. At the same time Cray increased the clock rate of the machine 10% to 220 MHz, bringing the potential data rate to 70.4 Gigabytes per second.

The MTA-2 was designed to overcome the challenges presented by the increasing ratio of processor clock rate to memory access speed. Processor speed has been following Moore's law (a doubling in speed every 18 months) while memory speeds have been increasing at closer to 10%. The time to access a single word from memory has grown to as much as tens of cycles The standard solution to this growth differential has been the development of high speed caches (often at several levels) where data may be temporarily stored. This "solution" relies on using data retrieved from memory to cache many times without having to access it again from main memory and updating data several times to cache before storing it to main memory.

Such solutions do not work well when there is little reusability of the data as in sparse problems or where the data does not fit in cache. The designers of the MTA-2 envisioned a different solution to this mismatch. By building hardware that supported multi-thread execution and the context switching from one thread to another within a single cycle, the memory access delay could be hidden by running enough threads simultaneously on each processor (typically 30-60 for memory intensive codes). In this way, at least one thread would be ready to execute on any given cycle.

2. Methodology

In order to evaluate the effect of the increased clock speed and the addition of the top plane, four codes (Flux and Alla from the original benchmark suite; Lanczos and Causal from the set of production codes running on the MTA-2) were analyzed using three different methods: measurements of wall clock time, measurements of system performance while the codes were actually running, and examinations of the parallelization and loop blocks generated by the compiler.

For each of the codes, a typical set of parameters was chosen and the wall clock time of the program was measured, using the UNIX date command run from the command line, for the program executing on a range from one processor to all forty. In addition computationally intensive portions of the program were timed using the Fortran 90 system_clock intrinsic.

While each of the codes was running on all forty of the MTA-2 processors, mtatop (Cray's version of the UNIX top command) and dashboard (a graphical version of mtatop displaying more information) were used to monitor system activity. From the mtatop output, the short-term average of CPU utilization, memory references per seconds, and floating point operations per second were recorded. The dashboard graphical display was also used to monitor the same three facets of program execution as well as the thread utilization percentage, traps per second, and the average ready streams per CPU.

Finally for the two production codes (Flux and Causal) where the vast majority of time is spent in a short section of the code the MTA-2 code analyzer canal was used to examine the parallelization of the code and the number of instructions, floating point operations, and memory accesses of the code.

3. Applications

3.1 Flux

Flux calculates the temperature dependent electronic excitation spectra and super conducting densities [2] of materials by iteratively solving over a four dimensional grid the non-linear equations of propagator functional theory, Dyson's equation and a self-iteration equation. Fast Fourier Transforms (FFT's) are used to transform between the position-imaginary time and crystal momentum-imaginary frequency spaces. This requires the addition of a frequency conditioning method to remove unphysical artifacts of the FFT procedure and to ensure that the solution is self-consistent.

Most of the time spent in solving the material problem is spent in updating the equations at each grid points and performing FFT's over each of the axis of the grid. Since the values at each grid point are a function of the grid point for the previous iteration and none of the current grid points, the updates of the grid point values is accomplished by an embarrassingly parallel set of quadruple nested door loops. The update of the values at the grid points required about 75% of the calculation time with most of the rest of the time spent in the FFT routines.

Table 1 contains the timings of a flux calculation over a 128x128x1x512 grid for both the old (200 MHz) and new (220 MHz) MTA-2.

Processors	Old Time (secs)	New Time (secs)	Speedup
1	6103	6478	0.94
2	3167	3298	0.96
4	1708	1724	0.99
8	915	896	1.02
16	577	465	1.06
24	372	294	1.29
32	311	229	1.36
40	286	191	1.50

Table 1: Flux Iteration times for MTA-2

Comparing the two time columns, the addition of the top plane provided a significant reduction in the time required to calculate the solution of the flux equations when the program was run on multiple processors with the total running time 1.5 times faster for 40 processors. While the code only scaled by a factor of 21 without the top plane, the scaling was 34 with the top plane. Scaling of the code is limited by a significant amount of formatted I/O after the convergence of the flux algorithm.

3.2 Alla

Alla is a fluid dynamics code for reactive Navier-Stokes and Euler simulations that utilizes a high-quality Godunov-type, finite-volume, explicit integration algorithm that is second-order accurate in space and time. The code uses an NRL-developed Fully Threaded Tree (FTT)-based local mesh refinement for obtaining highly resolved solutions [3]. FTT arranges cells in layers of refined meshes. Logically, the cells in FTT are organized as a standard oct-tree with pointers going to children, neighbors and parents. The thread pointers in FTT are inverted and directed from children to parent’s neighbors allowing creation and destruction of children without affecting neighbors. All FTT operations can be performed in parallel. Most of the time in the calculation portion of the code is spent in transversing the FTT tree and updating the simulated values at each grid point.

After every n steps, Alla writes a binary restart file and formatted ASCII data files that will later be used for post processing visualization. Since the formatting of the ASCII data and corresponding I/O is a significant portion of the running time on the MTA-2, calculations and I/O are overlapped requiring extra copies of the data to be saved until the data has been stored on disk. Once enough processors have been applied to the problem so that the calculations for n steps can be performed faster than data can be written to disk, the gain from extra resources is minimal and may even degrade as old data is accumulated.

Times for running Alla on a benchmark case on the old and new MTA-2 are given in Table 2.

Processors	Old Time (secs)	New Time (secs)	Speed Up
1	12910	10368	1.24
2	6342	5257	1.22
4	3126	2561	1.33
8	1711	1478	1.33
12	1346	1062	1.27
16	954	820	1.20
20	764	706	1.21
24	633	621	1.21
32	521	519	1.25
40	415	478	1.24

Table 2: Alla Iteration Times for MTA-2

Measurements from mtatop indicated that the program achieved a rate of 3.4G memory references per second on the upgraded machine or about 25% more bandwidth than the old MTA-2. This is consistent with the 20-33% improvement in the time to calculate 1000 iterations seen in Alla is it was run on a range from one to forty processors.

3.3 Lanczos

The Lanczos code grew out of a study of the low temperature thermodynamic properties on many body quantum systems [4]. Such studies involve three steps: the generation of the non-zero entries of very large sparse symmetric matrices (millions of rows and columns), the generation using a Lanczos method [5] of the non-zero entries of a symmetric tri-diagonal matrix of a much smaller order (thousands of rows and columns) whose eigenvalues approximate the largest and smallest eigenvalues of the sparse matrix, and the determination of the eigenvalues of the tridiagonal matrix. The first and last of these steps may be easily done on a PC, but the middle set of calculations require a high performance computer to perform the calculations in a reasonable length of time. Previously these calculations were performed using a C++/MPI program. For the MTA-2 a simple shared memory Fortran 90 program was written.

Table 3 shows the wall clock times using the C++/MPI program (top part of table) and the new F90 code on shared memory machines (bottom part). The X1 processor count is based on SSP's and the entry for 32P is N/A as the shared memory code runs only on 1 node (16 SSP's) of the X1. The matrix used in the calculation was 10 million by 10 million with an average of 33 elements per row and 1000 lanczos coefficient pairs were generated.

Platform	Speed MHz	16P mins	32P mins	Speed Up
SGI Origin	400	158	161	0.99
SGI Altix	1300	113	117	0.96
Cray X1	800	212	112	1.90
IBM P3	375	143	112	1.28
Intel Pentium III	933	103	74	1.39
COMPAQ ES45	1000	90	64	1.42
Intel Xeon	3060	69	49	1.40
SGI Altix	1300	110	148	0.75
SGI Origin	500	90	128	0.70
Cray X1	833	28	N/A	
Cray MTA-2 (old)	200	28	17	1.73
Cray MTA-2 (new)	220	23	11	2.06

Table 3: Wall Clock Times for 1000 Lanczos coefficients

Even thought the MTA-2 had the lowest clock rate, the coefficients were generated 4.5 times faster than the next best system and an order of magnitude faster than most of the systems. For the C++/MPI programs, while the system with the fastest clock produced the fastest result, little correlation was found between processor speed and wall clock time.

The code for the loop to calculate a diagonal (aval) and an off-diagonal (bval) entry of the tri-diagonal matrix from the sparse matrix R stored in a compressed row format (CRS) is:

```
Do index = 1, Imax
 Y(index) = 0
 Do index2 = I(index)+1, I(index+1)
  Y(index) = Y(index)+R(index2)*Vvec(J(index2))
 end do
end do
aval(iteration) = DOT_PRODUCT(Y,Vvec)
Uvec=Y+aval(iteration)*Vvec+Uvec
bval(iteration)=sqrt(DOT_PRODUCT(Uvec,Uvec))
Do index = 1, Imax
 Tmp = Vvec(index)
 Vvec(index)=Uvec(index)/bval(iteration)
 Uvec(index)=-Tmp*bval(iteration)
end do
```

Approximately 90% of the calculation time of this code is spent in the first six lines where the product of the sparse matrix and a dense vector v is performed. The code is the "same" for both real and complex matrices R with the only difference being whether the vectors and matrices are declared real or complex. In order to gain a feeling for the execution time of the first do loops on the MTA-2, canal was used to analyze the double do loop in the first six lines of the above code with most of the calculations performed in line four.

For the real case the execution of the one line of code in the inner loop requires three instructions, three memory references (one to retrieve the vector index, one to retrieve the sparse matrix element, and one to retrieve the vector element) and two floating point operations. For the MTA-2 this 3:3:2 ratio of instructions to memory operations to floating point operations indicates that the limiting factor on the program execution will be the ability of the MTA-2 to retrieve data from memory. Thus the reduced bandwidth of the MTA-2 without the top plane was expected to have a significant impact on the execution of the Lanczos program. Since the code has two floating point operations on every three cycles, and the hardware will support nine floating point instructions every three cycles, the sustained flops for this code is only 22% of peak or 5.8 Gigaflops.

A similar canal analysis for the case for a complex matrix leads to seven instructions, five memory references (one to retrieve the vector index, two to retrieve the complex matrix entry, and two to retrieve the complex vector entry) and eight floating point operations (six for the complex multiply and two for the complex add). Now the ratio of instructions to memory references to floating point operations is 7:5:8. Without the top plane the code is limited by the aggregate memory bandwidth of the MTA-2 internal network; with the top plane the code is limited by the ability of the MTA-2 to execute instructions. In seven cycles the MTA will be able to execute only eight floating point operations versus the twenty-one that the hardware will support. Thus the peak gigaflop ratio for this code will be less than 38% of peak or 10.0 gigaflops.

Measurements were also made for both the old and new MTA-2 of the program running on a real matrix and a complex matrix over a range of runs using one to forty processors. The matrices had 10 million rows and columns and on the average; each row had thirty-three non-zero elements. Table 4 contains the times of the loop for a real matrix and Table 5 for a complex matrix.

Processors	Old Time (secs)	New Time (secs)	Speedup
1	9.76	8.12	1.20
2	4.86	4.03	1.20
4	2.43	2.01	1.21
8	1.29	1.01	1.28
12	0.94	0.67	1.40
16	0.76	0.51	1.49
20	0.65	0.40	1.63
24	0.61	0.34	1.79
28	0.57	0.29	1.97
32	0.55	0.26	2.12
36	0.53	0.23	2.30
40	0.53	0.21	2.52

Table 4: Time per Iteration for Real Matrix

Processors	Old Time (secs)	New Time (secs)	Speedup
1	24.97	20.60	1.21
2	13.00	10.54	1.23
4	6.49	5.16	1.26
8	3.22	2.60	1.24
12	2.17	1.75	1.24
16	1.65	1.36	1.21
20	1.35	1.05	1.29
24	1.20	0.88	1.36
28	1.06	0.75	1.41
32	0.98	0.66	1.48
36	0.94	0.59	1.61
40	0.91	0.54	1.69

Table 5: Time per Iteration for Complex Matrix

From Tables 4 and 5, as more processors are applied to the problem, a nearly ideal scaling is seen for the real and complex case for the MTA-2 with the top plane. Previously the MTA-2 exhibited this type of scaling only to about 20 processors as the insufficient aggregate bandwidth of the network limited the total execution time of the code significantly. With the addition of the top plane and 10% clock speed increase running time of the code improved by a factor of 2.5 for the real case and 1.65 for the complex case. The better improvement for the real case was due to the real case being able to take full advantage of the increased bandwidth, while the limiting factor for the complex case was the rate at which instructions could be executed.

3.4 Causal

Causal models the propagation of an acoustic wave in water by using a fourth order in time and space finite difference time domain representation of the linear wave equation that has been modified by the addition of the derivative of the convolution between an operator and the acoustic pressure to take into account a dispersive medium [6-8]. This discretization of the wave equation and the addition of the derivative leads to a set of nested do loops of the form:.

```
do j=1,nz-1
 do i=1,nx-1
  fsq=cv(i,j)*dt*dt
  cw=sqrt(cv(i,j))
  grad=fsq*(-60.0*u(i,j,i1)+16.0*(u(i+1,j,i1)+u(i,j+1,i1)
        +u(i-1,j,i1)+u(i,j-1,i1))-(u(i+2,j,i1)+u(i,j+2,i1)
        +u(i-2,j,i1)+u(i,j-2,i1)))/(12.0*dx*dz)
        +fsq*fsq*(20.0*u(i,j,i1)-8.0*(u(i+1,j,i1)+u(i-1,j,i1)
        +u(i,j+1,i1)+u(i,j-1,i1))+2.0*(u(i+1,j+1,i1)
        +u(i-1,j+1,i1)+u(i+1,j-1,i1)+u(i-1,j-1,i1))
        +u(i+2,j,i1)+u(i-2,j,i1)+u(i,j+2,i1)+u(i,j-2,i1))
        /(12.0*dx*dx*dz*dz)
   u(i,j,i0)=2.0*u(i,j,i1)-u(i,j,i2)+grad
   sump1=0.00
   do ijk=1,ntau
```

```
    sump1=sump1+cpo0(ijk,i,j)*ucon(indx+ijk-1,i,j)
   end do
   cp(i,j,ic1)=sump1*dt
   deriv=2.0*fsq*(25.0*cp(i,j,ic1)+48.0*cp(i,j,ic2)
        +36.0*cpi,j,ic3)-16.0*cp(i,j,ic4)+
        +3.0*cp(i,j,ic5))/(12.0*cw*dt)
   u(i,j,i0)=u(i,j,i0)+deriv
   ucon(nlocm1,i,j)=u(i,j,i0)
  enddo
enddo
```

Canal was used to analyze the body of the nested loops. Most of the execution time was contained in the three lines that are required to update u(i,j,i0) by grad, the loop to calculate sump1, and the three lines to update u(i,j,i0) by deriv. The results are contained in Table 6.

	Instructions	Memory ops	Floating ops
grad	44	33	40
sump1	2	2	2
deriv	20	10	24

Table 6: Causal Operation Counts

A typical set of parameters for causal would be nx=2400, ny=2400, and ntau =1024. The updates of u(i,j,i0) require only 64 instructions while the do loop to calculate the value of cp requires 2408. Thus the execution time of the program is dominated by the ijk loop with its two instructions, two memory operations, and two floating point operations. While the code was not run on the old MTA-2, this loop would have been hampered by the limited aggregate bandwidth of the old MTA-2. Measurements using mtatop of the memory operations on the new MTA-2 were 7.5 Gig memory operations per second.

Table 7 presents the measurements of causal running on a range of one to forty processors. The code scales well across the range of processors.

Processors	Time (secs)	Scaling
1	6812	
2	3502	1.94
4	1747	3.89
8	893	7.62
12	611	11.14
16	469	14.52
20	382	17.83
24	325	20.96
32	255	26.71
40	217	31.39

Table 7: Causal Wall Clock Times

A second version of causal (referred to as causalrd2) was developed for the case where most of the medium is non-dispersive for the source of interest. In this case, for most of the points in the grid, of the variables in table 6 only the code for the calculation of grad will be executed. The correction for dispersion (the calculation of sump1 and deriv) will be performed on a only a relatively small percentage of the grid points. A typical problem for this case would be nx=6000 and nz=6000 with the dispersion calculations of the derivative over only 180,000 points. For these parameters each pass through the nested do loop will require 1600 million instructions compared to the 360 million for the code accounting for dispersion. Thus the running time of the code will be dominated by the calculation of grads.

This shift in where the bulk of the calculations are performed considerably complicates the behavior of the program. First, a load balancing problem arises as the amount of calculations for a particular i,j pair is no longer the same, but varies . The ratio of instructions to memory operations is closer to 1.3 than to 1. Also the number of memory references in calculating grad is nearly twice as many as one would expect from counting the number using the source code as some of the variables are reloaded from memory a second time. Thus, the code that is generated by the compiler is less efficient than the lanczos and causal codes. Future work will be devoted to improving the causalrd2 code. Timings for 5000 time steps with nx=6000 and nz=6000 are given in Table 8.

Processors	Time (secs)	Scaling
1	60787	
4	15264	3.78
8	7704	7.89
12	5162	11.78
16	3926	15.48
20	3234	18.79
24	2788	21.80
32	2179	27.89
40	2030	29.94

Table 8: Causalrd2 Wall Clock Times

4. Conclusions

In order to obtain an overall idea of the performance of the codes after the addition of the top plane and the 10% increase in clock rate, mtatop and dashboard were used to record the cpu utilization, memory references per second, floating point operations per second, and average number of ready streams per processor for each of the codes while running on all forty processors. Table 9 summarizes the results of the codes running on all forty processors.

Without the top plane, the 200 MHz MTA-2 was only able to achieve a maximum memory reference rate of 2.7 Gigawords per second. Increasing the clock rate by 10% to 220 MHZ would have increased this to only 3.0 Gigawords. All of the programs ran with peak memory reference rates exceeding this. Overall performance of these codes improved anywhere from a factor of 1.13 to 2.5 with the addition of the top plane.

	% CPU Utilization	GMem	Gigaflops	Ready Streams
Max	100	8.8	26.4	100
Flux	82	4.5	5.5	20
Alla	80	3.4	4.6	16
Lanczos (real)	94	7.5	4.5	26
Lanczos (complex)	80	5.1	6.8	5
Causal	96	7.5	6.9	28
Causalrd2	85	3.6	3.0	11

Table 9 Mtatop Measurements on Running Codes

These results provide insights as to how the MTA-2 compares with other HPC resources. Programs that perform well on the MTA-2 as compared to other HPC machines are those that require a high bandwidth to memory – i. e. those with little data reusability. When a code has such behavior the MTA-2 will not only show a high memory op count but also a high CPU utilization (as it requires an instruction to read from or write to memory) and the average number of ready streams count per processor will be greater than one (when there is no instruction ready to execute no useful work can be done). Codes that show high utilization and low memory operations are limited by instruction execution times and are better suited to machines that have high clock rates. Programs that have a small (< 1) number of ready streams do not have sufficient parallelism to run efficiently on the MTA-2, (on forty processors, a degree of parallelism of at least 1200 is generally required) and would probably be better suited on HPC machines with tens of processors.

Acknowledgments

The authors are deeply indebted to the Cray staff in Seattle, Washington especially Preston Briggs, and Jace Mogill, for their support and willingness to answer questions. Flux was developed by Dr. Daryl Hess, Alla by Dr. Alexei Kholkhov, Lanczos by Dr Steve Hellberg, and Causal by Dr. Guy Norton, all while they were at the Naval Research Laboratory. The MPI code version of Lanczos was developed under the High Performance Computer Modernization Program Office's (HPCMPO) Common High Performance Computing Software Support Initiative (CHSSI) and was run at the ARL and ASC MSRC centers. of the HPCMPO. Computations were performed on the MTA-2 provided under the auspices of the DOD HPCMPO and located at the Naval Research Laboratory in Washington, D. C.

About the Authors

Wendell Anderson is a mathematician and the head of the Research Computers Section of the Center of Computational Science at the Naval Research Laboratory. He can be reached at Naval Research Laboratory, Code 5593, 4555 Overlook Avenue Washington, D. C. 20375 or E-mail Wendell.Anderson@nrl.navy.mil. Marco Lanzagorta is a physicist for Scientific & Engineering Solutions. His interests include High Performance Computing, Quantum Computing, and Data Visualization. He can be reached at Naval Research Laboratory, Code 5593, 4555 Overlook Avenue Washington, D.C. 20375 or E-mail: Marco.Lanzagorta@nrl.navy.mil

References

1) Anderson, Osburn and Rosenberg, "NRL Report on its Cray MTA-2 (Multi-Threaded Architecture) Computer", *White paper to HPCMPO*, April 2003.

2) Diesz, Hess, and Serene, "Phase Diagram for the attractive Hubbard Model in two dimensions in a conserving approximation", *Physical Review B* 66 (2002).

3) Khokhlov "Fully Threaded Tree Algorithms for Adaptive Mesh Fluid Dynamics Simulations", *J. Computer. Phys.*, **143**, 519 (1998).

4) Anderson, Lanzagorta, and Hellberg, "Analyzing Quantum Systems Using the Cray MTA-2", *Proceedings of the Cray Users Group*, Columbus Ohio, May, 2003.

5) Cullum and Willoughby, Lanczos Algorithms for Large Symmetric Eigenvalue Computations, Birkhauser, Boston, 1985.

6) Blackstock, "Generalized Burgers equation for plane waves", *J. Acoustic. Soc. Am.* **77**, 2050, 1985

7) Szabo, "Time Domain wave equations for lossy media obeying a frequency power law", *J. Acoust. Soc. Am.* ***96**, 491, 1994*

8) Norton and Novarini. "Including dispersion and attenuation directly in the time domain for wave propagation in isotropic media", *J. Acoust. Soc. Am.* **113**, 3024 (2003)