A Scientific-Computer User's Assessment of the Cray MTA

by

Keith Taylor
Senior Applications Specialist
CSAR, Manchester Computing

Email: keith.taylor@mcc.ac.uk

9th October, 2000

1. Introduction

The Cray Multi-Threaded Architecture, or MTA for short, provides a radical alternative to the clustered SMP (Symmetric Multi-Processor) approach advocated by the vast majority of manufacturers for providing supercomputing facilities aimed at solving the most challenging scientific and engineering problems. Whereas the latter forces the programmer to undertake a substantial rewrite of his code in order to get it running efficiently in parallel, in particular, by ensuring his data are efficaciously placed so as to minimize remote memory accesses, the MTA processor removes this onerous task by hiding the time taken to load and store, behind the ability to run any one of a large number of threads of execution, as we shall explain in Section 2.

I think it is fair to say that the MTA approach offers the first practical automatic parallelizing system which will allow the user to take his serial program and almost immediately obtain the benefits of parallel execution. In the first keynote speech of the recent "Sixth European SGI/Cray MPP Workshop" (see **http://www.man.ac.uk/mrccs/mpp-workshop6/proc/pdf/reinhardt.pdf**), Steve Reinhardt of SGI emphasized the difficulty of exploiting clustered SMPs, a fact that has deterred Independent Software Vendors from porting their products onto these machines, as well as putting off many potential scientific users who could benefit from the undoubted performance promised by these architectures. His thesis was that there is no alternative for the foreseeable future. I believe the existence of the MTA disproves this.

In this report, I will give a scientific user's point of view of the present state of the MTA. I aim to be as fair as possible, not only extolling its virtues, but also pointing out its current deficiencies. I firmly believe that, given sufficient effort to remedy its undoubted problems, the MTA points the way to a much brighter future for High Performance Computing.

2. The Multi-Threaded Architecture in Summary

At present, the main sources of information about the MTA are to be found at **http://www.cray.com/product/systems/craymta** and **http://www.npaci.edu/MTA/links.html**, the latter being more scientific in content. I will attempt to summarize its most important hardware features here.

The key to the MTA's processor design is fast context switching, which takes place every cycle, between up to 128 complete hardware register sets called 'stream's, each supporting a single thread of execution. This, together with an extremely 'rich' interconnect, guarantees, to a high probability, that any potential delays caused by references to data in memory, which is all remote, are completely hidden. To increase this likelihood, there is provision for up to 8 outstanding memory references per thread.

The whole architecture has been built on the assumption that, for a large multi-user computer system, there will always be something useful that each processor can be doing, that is, threads that can be run whilst others are waiting for memory references to be satisfied. Memory is completely flat, making the careful data layout required for the T3E, for example, completely unnecessary, and indeed irrelevant. In fact, to reduce the potential for conflict, the user's data are thrown out completely at random, even 'contiguous' elements of an array. So the user has no control over data placement, even if he wanted it!

To provide an ever-increasing bandwidth, the network linearly scales as the architecture grows. The design rule is that there are $p^{3/2}$ communication nodes for p processors. Each node has 4 connections, with a link to which a resource - processor, memory module, or I/O processor - may be attached. There is one I/O processor for every processor, and typically one 0.5 GByte memory module for each of the other type of resource. Thus, for example, if p = 16, 16 processors, 16 I/O processors and 32 memory modules fully populate 64 nodes. For 64 processors, there would be 256 nodes, of which 128 would be unoccupied, but would provide extra bandwidth. The aim is to supply one 8-byte word to each processor, every cycle (currently 4 nanoseconds).

The MTA has been designed from the outset to serve the needs of the program developer. In principle, he need know nothing of the underlying hardware to obtain good parallel performance from a serial code. By contrast with a typical conventional MPP (Massively Parallel Processor), such as the T3E:

- there is no message passing or similar paradigm required to achieve parallelism;
- the careful data placement and cache management normally necessary to achieve reasonable performance is no longer an issue;
- strided array access is no longer a concern.

3. What the MTA Lacks

Before further discussing its advantages, I feel it is only fair to point out what I think the machine presently lacks in order for it to be accepted by the scientific-computing fraternity at large.

3.1 A Fortran 90 Compiler

Apart from C and C++, the MTA only has a FORTRAN 77 compiler available, although most Cray specific intrinsics are provided, including ‘MALLOC’ and ‘FREE’, for managing dynamically sized arrays. The inability to size arrays dynamically was a major reason why many Fortran programmers resorted to C before the arrival of Fortran 90. There are many other features which make the latter language attractive and thus, after a comparatively slow initial take-up, there is now a substantial body of Fortran 90 code around. Hence, it is important that Cray remedies this omission as soon as possible.

I have been told that a Fortran 90 compiler should be available this autumn.

3.2 An MPI (Message Passing Interface) ‘Library’

Given that the MTA is inherently parallel, this may need only to be a set of almost content-less wrappers for the most part. Many MPP users will have probably ‘mislaid’ their original serial codes, which were subsequently ‘MPIized’, by now, and thus, although basically cosmetic, an MPI ‘library’ would go a long way to making the MTA immediately available to this community. I doubt whether the average scientific user would be willing to maintain a separate MTA-specific version of their codes.

3.3 A Decent Debugger

The MTA has its own version of ‘gdb’ called, not surprisingly, ‘tdb’. This is driven by command lines and, frankly, is painful to use after being exposed to window-based debuggers such as ‘totalview’. If the latter can cope with interactive debugging on many processors of the T3E, I guess it shouldn’t be too difficult to port it to the MTA. I would hope that Tera’s recent acquisition, with subsequent name change, should smooth the path.

3.4 A Decent Profiler

‘traceview’, which provides a time history of your run, is all very well, but it’s not easy, if possible, to extract the vital information you need to start optimizing, like the sub-program units which are run, how often they are invoked, and what proportion of the runtime they individually occupy. ‘traceview’ is also very slow, unless you’re extremely selective about what it records. But, how would you know what to choose, until you’ve profiled the code? Something like ‘apprentice’ or ‘pat’ would make a welcome addition to the portfolio of tools available on the MTA.

3.5 Directory Cross-Mounting

Currently, all tools run on a Sun front-end. All executables and associated large input files have to be 'ftp'ed to the MTA at the SDSC (San Diego Supercomputing Center); and lengthy results may also have to be returned by the same mechanism. This is inconvenient. Cross-mounted directories, like we have for the Cray T3E, SGI Origin2000, and Fujitsu VPP300, at CSAR, would make life much easier. Apparently, it is just the SDSC policy not to cross-mount workstations and production machines, which precludes this in their installation. So, there should be no difficulty, in principle.

I have been informed that NSF will be along soon.

3.6 Machine Availability

Since the start of the New Year, about the world's only MTA, at SDSC, has been very unreliable. The problems stem from an enhancement in which the infrastructure has been expanded to accommodate an increased number of processors from 8 to 16, although all the extra processors aren't in place yet. With only one machine available, Cray is having to prove as it builds. The design ideas may be mature, around 25 years old, but the technology is less so.

In mitigation, I would say that the SDSC's MTA was far more stable before Christmas, when it was a purely 8-processor architecture, and the results obtained then bear ample testimony to its capability for being made into a serious production machine. A vast improvement should occur when CMOS starts to come on-stream (later this year?). See http://www.tera.com/products/systems/craymta/cmos.html.

Additionally, to its credit, Cray does not try to sweep the problem under the carpet. SDSC provides a web page for users, http://tera.sdsc.edu/MTA/graph-cgi/graphs.html, which shows recent usage and downtime, as well as giving information about current developments. I should also say that the SDSC MTA appears to have been a lot more reliable over the last few weeks.

3.7 A Rudimentary Editor

Although there is no point in editing source code on the MTA where there is no compiler, nevertheless, it would be nice to be able to create simple shell scripts to control the running of executables, manage output files et cetera. Currently, I find it most convenient to cut and paste commands from a 'palette' file on another machine which ***does*** have an editor. Small amounts of output, timings for example, are transferred back and recorded in similar fashion.

4. <u>What the MTA, Cray and SDSC Do Currently Provide</u>

4.1 Machine Access

This is via a Sun workstation to which one 'ssh's. All the tools, editor, compiler, linker et cetera, run there. This is sensible, since you obviously want the MTA to concentrate on running your computationally intensive applications. However, it means you have to 'ftp' executables and large input/output files backwards and forwards. See **3.5** above.

4.2 The FORTRAN 77 Compiler, 't77'

This is excellent compared to many I have encountered. For instance, when I compiled John Brooke's code (see Section **5.2** below), the compiler kindly warned me that a particular scalar was used before initialization, something you would normally only find out on the Cray, for example, at runtime, if you had ensured that the memory is initialized to 'NaN's. Most people, of course, don't bother, relying on the default setting to zero, thus running incorrect code for years and years and years and . . .

The man page for 't77' lists around 70 options. But of these, only 8 are concerned with parallelization and optimization. Contrast this with the literally dozens of separate optimization options offered by 'f90', not to mention the myriad of possible combinations, with which the T3E programmer may have to grapple if he is to attain reasonable serial performance for his code.

Two other 't77' options are worthy of note:

- '-case_sensitive' which, not surprisingly, makes symbolic names case-sensitive. I see this as particularly useful for avoiding typographical errors, if one is going to write source in mixed case, in conjunction with the IMPLICIT NONE statement. Programmers from a C background, and even some from a Fortran background, can often forget that Fortran is not case sensitive.

- '-no_dp' which treats DOUBLE PRECISION as the same length as REAL, that is IEEE 64-bit. The MTA does have 128-bit real arithmetic, ***not*** in IEEE format, but it is a lot slower, even though Tera has adopted a more efficient form due to William Kahan (**<u>http://www.cs.berkeley.edu/~wkahan</u>**).

There are a number of MTA specific compiler directives available for more detailed tuning of code. And, as reported from the audience at my recent talk, OpenMP directives are recognized, which should at least make the porting of code parallelized using this shared-memory paradigm easier.

4.3 The Compiler Analysis Tool, 'canal'

If the compiler is excellent, then this is brilliant. 'canal' not only tells you about loops which it couldn't parallelize, with reasons, but gives you a detailed breakdown of each it could, counting instructions, floating point operations, memory references, register spillages, ***and*** the compiler's chosen number of threads which can easily be increased via a directive to obtain possibly even better performance.

It should be emphasized, however, that the compiler does such a good job, that the average user will not require to use 'canal'. However, nice to have, particularly within a group such as CSAR.

4.4 The Assembler Facilities

…'tcc', the C compiler, offers an assembler option, thereby creating '.s' files. There is also 'tera-dis', a disassembler. I haven't investigated either of these facilities yet, principally because I originally thought they were fatuous. However, having talked to Cray's David Tanqueray, I now realize I may have been a bit hasty. There appears to be lots of scope for optimizing the machine code, especially avoiding unnecessary loads and stores, that is memory references, and making best use of the registers associated with each 'stream'.

4.5 Help and advice

I have been extremely impressed by all the Cray people associated with the MTA, with whom I've had dealings. They are extremely open, and any queries I've had, have been answered rapidly and informatively. I would especially like to pick out, without intending offense to others, Cray's representative at the SDSC, John Feo, who has had a particularly trying time recently. (See Section **3.6**.) Also Preston Briggs, who gave an extremely good course here at CSAR, in the middle of March.

4.6 Support for Multi-User Job Management

Because of the way the machine was designed from the outset, the primary aim of its operating system is to keep all the processors busy, all of the time. It makes no real distinction between different users' jobs. All it's really interested in is threads, ready to run, from anywhere. Thus, effectively, it already is a batch-processing engine with its main emphasis on throughput. I am sure this will strike a chord with any organization wishing to acquire and manage an MTA for general use.

5. Assessing the MTA

There is ample evidence, already, that the MTA can easily outperform an equivalent Cray T3E, and particularly, SGI Origin2000, in the right circumstances. Two especially convincing papers exist which clearly come down in favour of the MTA, in realistic scientific computing settings:

http://www.nersc.gov/~oliker/papers/pdcs00.pdf

and

http://www.nersc.gov/~oliker/papers/sc99.pdf.

The most striking feature of these is how little the serial code had to be rewritten in order for the MTA to outshine its two competitors. The latter paper, for example, which was concerned with the parallelization of the re-meshing operations required for a Computational Fluid Dynamics application running on a dynamic 'unstructured' grid, showed that the code increased by 100% when an MPI version was created for the T3E and Origin, as opposed to a 2% increase for the MTA. The best runtime achieved on the T3E was 3 seconds on 160 processors, whereas the MTA achieved 0.35 seconds on 8!

I have involved myself in the following set of assessment activities.

5.1 Investigating the Performance of a Dot Product

The primary purpose of this exercise was to develop a reliable and consistent timing procedure for measuring the MTA's performance. The example chosen was deliberately simple so that the results from 'canal' and execution times were easy to interpret and compare with those from a single processor of the Cray T3E-1200E. The outcome is described in Section 6.1.

5.2 Porting and Optimizing 'alnpar'

This is a serial code, supplied by John Brooke of CSAR, which has been used for assessing other machines, including Silicon Graphics workstations. Being essentially a collection of operations on sequentially accessed arrays, it seems eminently suitable for the Cray's processor and memory system. I compiled and ran it successfully, after a little bug-fixing via 'totalview', on a single Application node of CSAR's Cray T3E-1200E, for comparison, and then ported it to the MTA. It is indeed the experience gained whilst carrying out this activity, as well as my extensive reading around the subject, that has formed the basis for most of this report.

The results of the comparison are presented in Section 6.2.

5.3 Gaussian

Gaussian is a popular chemistry package which runs poorly on CSAR's T3E, 'turing', being typically scalable to at most about 32 processors in favourable circumstances. This is because its parallel version is built on top of 'Linda', a venerable virtual shared-memory API (Application Programming Interface), much heralded in the mid to late 80's. The programming model, employing 'tuples'; is comparatively simple, but can sustain big performance hits compared to more explicit parallelization employing message passing.

Another source of inefficiency is the comparatively poor provision of I/O support on the T3E. This means that the option is taken to recalculate integrals every time they are needed, rather than calculate them once, store them to disk, and retrieve as required.

The MTA addresses both problems. Firstly, for the MTA, explicit parallelism is no longer relevant, and so only the serial version of Gaussian need be ported. Secondly, since the MTA provides an I/O processor for every processor, I believe that at least part of the integrals can be saved to and retrieved from disk, with advantage. Currently, there is no parallel I/O implemented in Fortran for the MTA, but there are low-level routines in existence which would enable the development of a suitable library to provide this facility.

Gaussian has been ported to the MTA and is currently undergoing optimization. I believe that when Gaussian for the MTA is released, it will do much to enhance the reputation of the machine.

6. Some Preliminary Results

6.1 Dot Product on the MTA and T3E-1200E

I have already explained the background behind this example in Section 5.1. Here are the essentials of the `PROGRAM`:

```
************************************************************************
* KT_test_system_clock.f: created by KT, 06/07/00, to do the obvious.  *
************************************************************************
      PROGRAM TEST_SYSTEM_CLOCK
        IMPLICIT NONE
        INTEGER NUMBER_OF_REPEATS,SIZE_OF_A
************************************************************************
* Choose the number of repeats, and size of A and B, with the next two *
* statements.                                                          *
************************************************************************
        PARAMETER(NUMBER_OF_REPEATS=1000)
        PARAMETER(SIZE_OF_A=1000000)
        REAL A(SIZE_OF_A),B(SIZE_OF_A),TOTAL_TIME,AVERAGE_TIME,
     $       MAXIMUM_TIME,MINIMUM_TIME,STANDARD_DEVIATION,
     $       MAXIMUM_MFLOPS_PER_SEC,A_DOT_B,KT_DOT_PRODUCT,
     $       TIME_ARRAY(NUMBER_OF_REPEATS)
        INTEGER DECIMAL_DIGIT,REPEAT_NUMBER,START_COUNT,COUNT_RATE,
     $          COUNT_MAX,FINISH_COUNT
        WRITE(*,*) 'KT_test_system_clock: started.'
************************************************************************
* Fill A and B with pseudo-random numbers >= 0 and < 1.                *
************************************************************************
        CALL KT_RANDOM_NUMBER(SIZE_OF_A,A)
        CALL KT_RANDOM_NUMBER(SIZE_OF_A,B)
************************************************************************
* Calculate and write out the statistics of A.                         *
************************************************************************
        CALL KT_CALCULATE_TIME_STATISTICS(SIZE_OF_A,A,0.0,'?',
     $               TOTAL_TIME,AVERAGE_TIME,MAXIMUM_TIME,MINIMUM_TIME,
     $                        STANDARD_DEVIATION,MAXIMUM_MFLOPS_PER_SEC)
        . . .
************************************************************************
* Calculate and write out the statistics of B.                         *
************************************************************************
        . . .
        DO REPEAT_NUMBER=1,NUMBER_OF_REPEATS
          WRITE(*,*) 'KT_test_system_clock: REPEAT_NUMBER = ',
     $               REPEAT_NUMBER,'.'
          CALL SYSTEM_CLOCK(START_COUNT,COUNT_RATE,COUNT_MAX)
C$TERA FENCE
          A_DOT_B=KT_DOT_PRODUCT(SIZE_OF_A,A,B)
C$TERA FENCE
          CALL SYSTEM_CLOCK(FINISH_COUNT,COUNT_RATE,COUNT_MAX)
          TIME_ARRAY(REPEAT_NUMBER)=REAL(FINISH_COUNT-START_COUNT)/
     $                              REAL(COUNT_RATE)
        ENDDO
************************************************************************
* Check the dot product result and show the processor clock's          *
* count-rate.                                                          *
************************************************************************
        . . .
************************************************************************
* Calculate and write out the timing statistics.                       *
************************************************************************
        CALL KT_CALCULATE_TIME_STATISTICS(NUMBER_OF_REPEATS,TIME_ARRAY,
     $          2*SIZE_OF_A*1.0E-06,'KT_calculate_time_statistics.out',
     $               TOTAL_TIME,AVERAGE_TIME,MAXIMUM_TIME,MINIMUM_TIME,
     $                        STANDARD_DEVIATION,MAXIMUM_MFLOPS_PER_SEC)
        . . .
        WRITE(*,*) 'KT_test_system_clock: terminated.'
      END
```

We first generate two `REAL` arrays of a million elements, containing pseudo-random numbers. Secondly, we check their statistical properties, for reasons I'll explain in the next but one paragraph.

Next, we time the dot product function for a thousand repeats, recording the result for each pass separately. The MTA version of `SYSTEM_CLOCK` is based on two implementation-specific C functions which return the user time, making allowance for periods when your program has been de-scheduled, and the processor's clock frequency. On the T3E, we simply resort to the Fortran 90 intrinsic procedure of the same name. The `C$TERA FENCE` directives ensure that the compiler is not allowed to mix up the instructions generated for `SYSTEM_CLOCK` and the dot product, which it might otherwise do in its quest for efficiency.

Then, we check the result of the dot product; and finally we calculate and record the statistics of the timings we have collected. `KT_CALCULATE_TIME_STATISTICS` has been written in such a way that, although it was intended to analyze timing results, it can also handle any series of statistical data. `KT_RANDOM_NUMBER` should produce an array of pseudo-random numbers between `0` and `1`. This it does on the T3E with average 1/2 and standard deviation $\sqrt{(1/12)}$. The dot product also correctly produces 1/4. My MTA version, however, was based on the supposedly T3E-compatible `RANF`, which erroneously generates pseudo-random numbers between 1/2 and 1, with the associated average of 3/4, and standard deviation $\sqrt{(1/48)}$. The dot product comes out at 9/16, which is at least consistent.

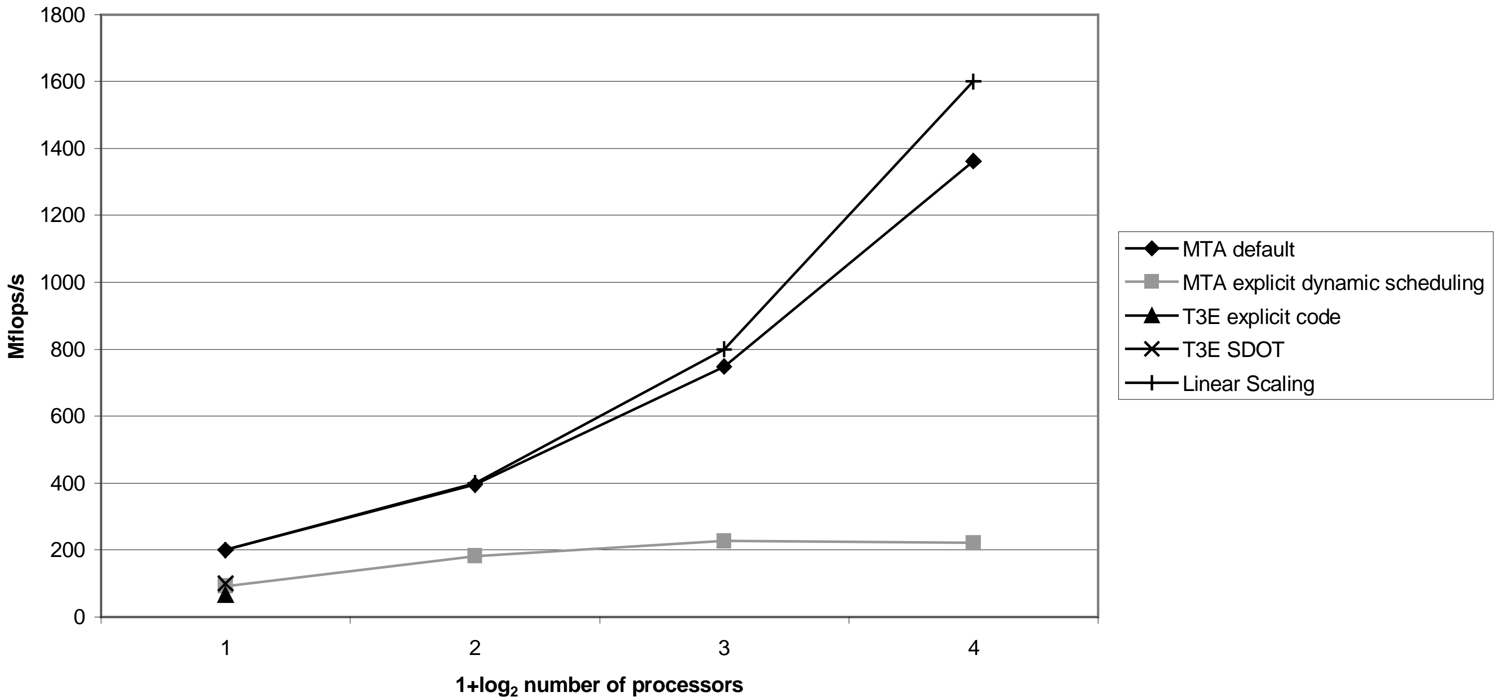
The code for the dot product is shown below in the form returned by 'canal'.

```
         | **********************************************************************
         | * KT_dot_product.f: created by KT, 07/07/00, to provide a FORTRAN 77   *
         | *                   FUNCTION based on the definition of the Fortran 90 *
         | *                   intrinsic procedure DOT_PRODUCT given on page 169  *
         | *                   of "Fortran 90 Explained" by Michael Metcalf and   *
         | *                   John Reid, Oxford University Press, 1990.          *
         | *                                                                      *
         | *                   SIZE_OF_VECTORS is a scalar default integer        *
         | *                   holding the size of the vectors;                   *
         | *                                                                      *
         | *                   VECTOR_A is a rank-1 default real array holding    *
         | *                   the first vector.                                  *
         | *                                                                      *
         | *                   VECTOR_B is a rank-1 default real array holding    *
         | *                   the second vector.                                 *
         | *                                                                      *
         | *                   DOT_PRODUCT returns the dot product of the two     *
         | *                   vectors as a default real.                         *
         | **********************************************************************
         | C$TERA NO INLINE
         |       REAL FUNCTION KT_DOT_PRODUCT(SIZE_OF_VECTORS,VECTOR_A,VECTOR_B)
         |         IMPLICIT NONE
         |         INTEGER SIZE_OF_VECTORS
         |         REAL VECTOR_A(SIZE_OF_VECTORS),VECTOR_B(SIZE_OF_VECTORS)
         |         INTEGER INDEX
         |         KT_DOT_PRODUCT=0.0
         |         DO INDEX=1,SIZE_OF_VECTORS
    3 P$ |           KT_DOT_PRODUCT=KT_DOT_PRODUCT+VECTOR_A(INDEX)*VECTOR_B(INDEX)
** reduction moved out of 1 loop
         |         ENDDO
         |       END


       Additional Loop Information


Parallel region   1 in kt_dot_product_
       Multiple processor implementation
       Requesting at least 30 streams

Loop   2 in kt_dot_product_ in region    1
       In parallel phase 1
       Dynamically scheduled, variable chunks, min size = 13

Loop   3 in kt_dot_product_ at line 27 in loop    2
       Loop summary: 2 memory operations, 2 floating point operations
              2 instructions, needs 30 streams for full utilization
              pipelined
```

The `C$TERA NO INLINE` directive is a belt to the `C$TERA FENCE` braces. We also notice that the compiler has rather intriguingly chosen to schedule the loop 'dynamically'. 'Block' and 'interleaved' are also available, of which, see more presently.

Clearly, each pass of the do loop requires two new array elements to be loaded. Hence, the minimum of two instructions required for Loop 3 as reported by 'canal'. (I should say, here, that each MTA instruction consists of three parts: the first contains a memory reference; the second, an arithmetic operation; and the third, a branch or one of a limited set of further arithmetic operations. Thus, in principle, it is possible to achieve 3 flops per instruction, memory accesses permitting. There is a floating point ***and*** integer multiply-add available for an instruction's second part.) The two floating-point operations easily fit into these. So, one should expect 250 Mflops/s, running on one processor at the current clock rate of 250 MHz.

Now let's look at the actual results.



(The rather odd looking format arises from the use of an Excel spreadsheet.) The best that a single processor of the T3E could manage was 99.14 Mflops/s, using the BLAS routine `SDOT`. The original code and the Fortran 90 intrinsic procedure `DOT_PRODUCT`, surprisingly, both came out the same at 65.72 Mflops/s. Basically, there is little that the T3E hardware can do to mask the latency of the memory accesses except set up a couple of look-ahead buffers (confusingly termed 'streams') to the arrays.

The MTA, on the other hand, can do a much better job, achieving 200 Mflops/s on one processor, 80% of our expectation. And, it is extremely easy to increase the performance by running on more than one: just add the compiler option '-par' and execute the program with the command 'trun'. ***Please note: the code remains exactly the same; there are no extra statements associated with parallelization, anywhere.*** Thus, we get, by default, near perfect linear speed up on 2 processors. After that, things begin to tail off, but 8 processors still achieve 68% parallel efficiency.

One other point of interest. I tried the three different available modes of loop scheduling by inserting the appropriate compiler directive. The only one that made a significant difference was `C$TERA DYNAMIC SCHEDULE` which, as you can see, made the performance considerably worse!

6.2 'alnpar'

The major lesson I learnt from this example was that it's very difficult to beat the compiler with just options and directives. Perhaps, I will have to resort to assembler, after all.

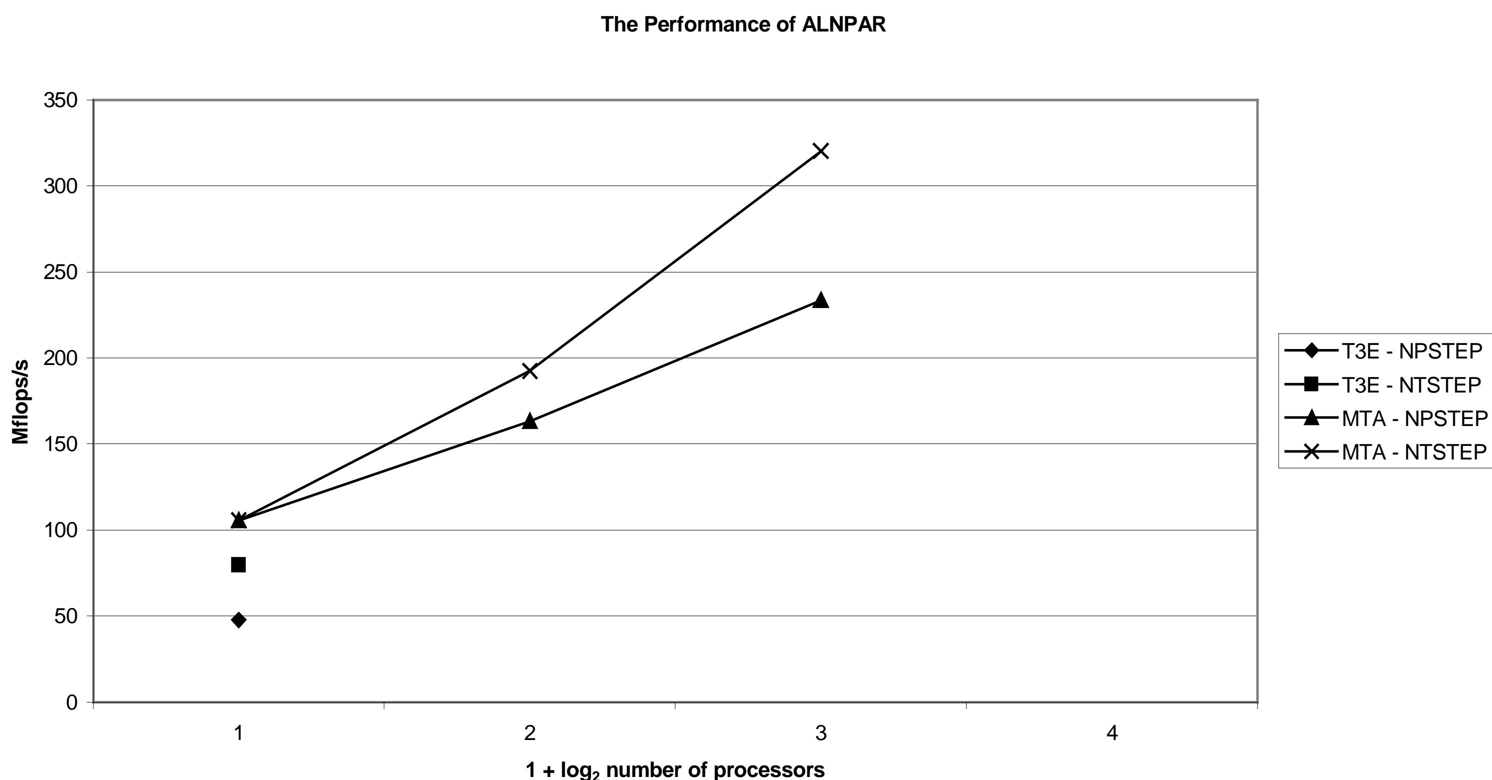
'alnpar' is a two-dimensional finite difference code, based on a 5-point stencil, which evolves by calculating field values at the next time step from the previous two. Using 'apprentice' on the T3E, I found that the bulk of the work is carried out inside two `SUBROUTINE`s, `NPSTEP` and `NTSTEP`. They are very similar in structure. The 'canal' output for the latter is shown below.

```
|       INCLUDE 'KT_ntstep_header.inc'
         | C$TERA NO INLINE
         |       SUBROUTINE NTSTEP(TCOEF1,TCOEF2,TCOEF3,TCOEF4,TCOEF5,TPCOF1,
         |      $                  TPCOF2,TPCOF3,TPCOF4,TPCOF5,TMINUS,TZERO,TPLUS,
         |      $                  PZERO,DX,DTH,DTIME,KT_NI,KT_NJ,XSTART,INLIN,
         |      $                  DIFFA,UNDER,AL0,ALPHA,HSN,HCS,SN,CS)
         |       IMPLICIT DOUBLE PRECISION (A-H,O-Z)
         |       INCLUDE 'KT_NI_NJ_SIZES.inc'
         |       DIMENSION TMINUS(NJ,NI),TZERO(0:NJ+1,NI),TPLUS(0:NJ+1,NI),
         |      $          TCOEF1(NJ,NI),TCOEF2(NJ,NI),TCOEF3(NJ,NI),TCOEF4(NJ,NI),
         |      $          TCOEF5(NJ,NI),TPCOF1(NJ,NI),TPCOF2(NJ,NI),TPCOF3(NJ,NI),
         |      $          TPCOF4(NJ,NI),TPCOF5(NJ,NI),PZERO(0:NJ+1,NI),
         |      $          DIFFA(NJ,NI),UNDER(NJ,NI),ALPHA(0:NJ+1,NI),AL0(NJ,NI),
         |      $          HCS(NI),HSN(NI),SN(NJ),CS(NJ)
         |       COMMON /PAR/PI,BD,CALPHA,COMEGA,A
         |       DOUBLE PRECISION KT_1OVER2DTH,KT_1OVER2DX,KT_2TIMES_DTIME,
         |      $                 KT_1OVER_HSN_I
         |       KT_1OVER2DTH=0.5/DTH
         |       KT_1OVER2DX=0.5/DX
         |       KT_2TIMES_DTIME=2.0D0*DTIME
         | C$TERA BLOCK SCHEDULE
         |       DO 29 I=2,NI-1
    3 P  |         X=(I-1)*DX
    3 P  |         KT_1OVER_HSN_I=1.0/HSN(I)
         |         DO 39 J=1,NJ
         |           TH=(J-1)*DTH
    3 PP |           C=HCS(I)-CS(J)
    3 PP |           BN=-PZERO(J,I)*SN(J)+C*(PZERO(J+1,I)-PZERO(J-1,I))*
         |      $        KT_1OVER2DTH
    3 PP |           BTH=-PZERO(J,I)*(1-HCS(I)*CS(J))*KT_1OVER_HSN_I+X*C*
         |      $         (PZERO(J,I+1)-PZERO(J,I-1))*KT_1OVER2DX
         |           BSQ=BN*BN+BTH*BTH+TZERO(J,I)**2
    3 PP |           TPLUS(J,I)=TCOEF1(J,I)*TZERO(J,I-1)+TCOEF2(J,I)*TZERO(J-1,I)+
         |      $              TCOEF3(J,I)*TZERO(J+1,I)+TCOEF4(J,I)*TZERO(J,I+1)+
         |      $              TCOEF5(J,I)*TMINUS(J,I)+TPCOF1(J,I)*PZERO(J,I-1)+
         |      $              TPCOF2(J,I)*PZERO(J-1,I)+TPCOF3(J,I)*PZERO(J+1,I)+
         |      $              TPCOF4(J,I)*PZERO(J,I+1)-ALPHA(J,I)*DIFFA(J,I)-C*
         |      $              (BTH*X*(ALPHA(J,I+1)-ALPHA(J,I-1))*KT_1OVER2DX+
         |      $                                BN*(ALPHA(J+1,I)-ALPHA(J-1,I))*
         |      $              KT_1OVER2DTH)*UNDER(J,I)*KT_2TIMES_DTIME
         |    39   CONTINUE
         |    29 CONTINUE
         |       AV=0.0D0
         |       DO 10 J=1,NJ
    4 P  |         TPLUS(J,NI)=0
    4 P$ |         AV=AV+TPLUS(J,2)
** reduction moved out of 1 loop
         |    10 CONTINUE
         |       AV=AV/DFLOAT(NJ)
         |       DO 11 J=1,NJ
    5 -  |         TPLUS(J,1)=AV
         |    11 CONTINUE
         |       END
```

```
       Additional Loop Information


Parallel region   1 in ntstep_
       Multiple processor implementation
       Requesting at least 60 streams

Loop   2 in ntstep_ in region    1
       Parallel section of loop from level 1

Loop   3 in ntstep_ at line 22 in loop    2
       In parallel phase 1
       Block scheduled
       Not loop scheduled: too many registers
       Expecting 6240 iterations
       Parallel section of loop from level 2
       Loop summary: 53 instructions, 46 floating point operations
              28 loads, 1 stores, 16 reloads, 1 spills, 1 branches, 0 calls

Loop   4 in ntstep_ at line 44 in region    1
       In parallel phase 2
       Interleave scheduled
       Expecting 160 iterations
       Loop summary: 2 memory operations, 1 floating point operations
              3 instructions, needs 60 streams for full utilization
              pipelined

Loop   5 in ntstep_ at line 49
       Expecting 160 iterations
       Loop summary: 1 memory operations, 0 floating point operations
              1 instructions, needs 30 streams for full utilization
              pipelined
```

The core of the code is a double do loop, denoted ‘3’ by ‘canal’. To assist the compiler, I replaced the passing of the dimensions NI and NJ by formal argument, with an INCLUDE file which contains their constant values expressed through PARAMETER statements, as in the main PROGRAM. ‘t77’ was then able to generate a better estimate of the optimum number of streams, for example. I had previously been rather disappointed, however, to find that just inlining the SUBROUTINEs was insufficient to enable the compiler to recognize that the do-loop lengths are in fact constant. (It is claimed that ‘t77’ can do a lot of whole program optimization if just one ‘.pl’ – program library-file is used.) But, it ***was*** able to work out that their internal structure did not prevent parallelization of the external time-stepping loop.

Directing the compiler to block schedule the outer loop seems to produce, by a small margin, the best performance figures, which we show below.



Despite launching many runs, I have so far been unable to obtain results for 8 processors on the MTA. I should also emphasize that no optimization has been carried out on the T3E, and so those performance figures, merely there for comparison, are perhaps a little unfair. I believe that marked improvement on that machine would require optimization of cache reuse and T3E stream number, with attendant significant code changes.

I will admit that I have tried to improve the speed on the MTA by similarly extensive code amendments. Worried by the number of reloads in Loop 3, I explicitly set up temporary scalars for the `J-1` and `J`th elements of the various arrays which are required to calculate `TPLUS`. These could then be saved in registers for use in subsequent loops. But, to no avail. (I actually achieved a reduction of 39% in `NTSTEP`'s speed!) The value for `NJ` of 160 produces blocks which are just too small to overcome the effect of the extra statements involved. However, it is heartening, once again, to observe from the graph that greater speed can be achieved without any explicit code parallelization, by running on more than one processor.

7. Conclusion

Today, there are essentially three different types of HPC architecture:

- clustered SMPs built out of workstation-type commodity processors;
- machines constructed from powerful proprietary vector processors;
- the MTA.

The first two types demand that, for efficient exploitation, the user program in a way which suits the architecture, if he is to extract reasonable performance. For the first especially, this means that the application should ideally be amenable to the domain-decomposition approach in which essentially the same program is run on a different part of the global data on each processor. This is often described as SPMD (Single Program, Multiple Data). Moreover, the data should be laid out statically in such a way that load balance between processors is achieved, and that most data is loaded from and stored to local memory to minimize the delays inherent in remote memory accesses. Additionally, for good single processor performance, accesses to arrays should be made from contiguous elements in positive unit-strided fashion.

Clearly, many applications do not fall into this convenient pattern. For example, linear algebra involving sparse matrices. Also, SPMD programs which act on dynamically changing amounts of data as demonstrated in the second reference at the beginning of Section 5. Periodically, there are substantial delays as the refined data is redistributed to reestablish load balancing and minimize remote memory accesses. Finally, there remain the many applications that are simply not SPMD, and require to be run substantially different parts of the whole program on different processors, such as in a functional decomposition approach. Many applications combine both domain and functional decomposition. Designing and implementing an efficient split of the executable and data across processors can be a nightmare!

The MTA avoids all the above. You retain the serial form of your application. Data placement is not an issue. What more can I say? It is a machine designed to make the programmer’s life easy as easy as possible and avail him of the advantages of parallel processing without any of the normal pain. Although it presently has many shortcomings, not least its recent lack of reliability, the MTA is useable now. Despite its lack of maturity in both hardware and software, it provides ample evidence that the multi-threaded approach is viable. I believe that Cfs should acquire one as soon as possible.