Sequence Alignment on the Cray MTA-2*

Shahid H. Bokhari
Department of Electrical Engineering
University of Engineering & Technology
Lahore 54890, Pakistan
shb@acm.org

Jon R. Sauer
Eagle Research & Development
11001 West 120th. Ave., Suite 400
Broomfield, CO 80201
sauer@EagleRD.com

Abstract

The standard algorithm for alignment of DNA sequences using dynamic programming has been implemented on the Cray MTA-2 (Multithreaded Architecture-2) at ENRI (Electronic Navigation Research Institute), Japan. Descriptions of several variants of this algorithm and their measured performance are provided. It is shown that the use of "Full/Empty" bits (a feature unique to the MTA) leads to implementations that provide almost perfect speedup for large problems on 1–8 processors. These results demonstrate the potential power of the MTA and emphasize its suitability for bioinformatic and dynamic programming applications.

1. Introduction

We describe the results of a series of experiments with sequence alignment algorithms on the Cray MTA-2 (Multithreaded Architecture-2) supercomputer [1, 2]. This research is an outgrowth of our prior work on simulating an ultrafast silicon based DNA sequencer [3, 8]. Our prior work addressed the problems of simulating the molecular dynamics of the ultrafast sequencer on the Cray MTA-2. We are now investigating the problems of analyzing the massive amounts of information that such a sequencer would generate. Clearly, very large amounts of computational power will be needed to analyse the resultant volumes of data.

As a first step, we have embarked on a study of the implementation of traditional DNA sequence alignment algorithms on the very non-traditional Cray MTA-2 supercomputer. The unusual architecture of this machine permits us to parallelize algorithms without having to concern ourselves with explicit details of parallel communications, mapping, load balancing, etc. The MTA-2 thus serves as an interesting alternative to more conventional supercomputers, clusters and 'piles' of PCs. It is potentially of great value to the bioinformatics community because it promises parallelization of existing serial code with minor modifications.

In the following, we first briefly discuss the architecture of the MTA-2. We start our presentation of sequence alignment by describing an implementation of the 'naive' algorithm for brute force exact matching. We show that, even for this simple algorithm, there are situations where careful algorithm design is needed for good performance.

The bulk of this paper is taken up by an exploration of the classic dynamic programming algorithm for approximate sequence alignment. We provide details of several implementations and describe the performance of the codes in detail. The MTA-2 is seen to permit easy implementation of the basic dynamic programming algorithm and demonstrates excellent speedup. It is a convenient machine for a researcher to develop and explore various alternative algorithms for a given problem.

2. Cray MTA Architectural Features

We first describe the general approach that the Cray MTA[1](Multithreaded Architecture) uses to obtain good parallel performance on arbitrary codes. We then discuss the hardware and basic performance specifications of this machine.

The MTA's approach to achieving high performance is to invest in additional hardware and software to support parallelism, possibly at the expense of additional compiler overhead. This approach does not permit the use of commodity microprocessors for parallel processing and requires a pro-

*This research was supported by the National Institutes of Health, grant R21HG02167-01. Access to the Cray MTA-2 was provided by Cray Inc., Cray Japan Inc., and the Electronic Navigation Research Institute (ENRI), Japan. Additional support was provided by the Optoelectronic Computing Systems Center, University of Colorado, Boulder.

[1]Formerly known as the Tera MTA. In April 2000, Tera Computer Company acquired the Cray Research business from Silicon Graphics, Inc. and subsequently changed its name to Cray Inc.

tracted cycle of development and production. However, the potential benefits are very attractive.

In the following brief overview of the MTA [2], we limit ourselves to features relevant to our code. Detailed information may be found at `www.cray.com/products/systems/mta/ ps-docs.html`.

Zero Overhead Thread Switching. An MTA processor has special purpose hardware (*streams*) that can hold the state of up to 128 threads. The *state* of each stream includes registers, condition codes, and a program counter. On each clock cycle, each processor switches to a different resident thread and issues one instruction from that thread. A blocked thread, e.g., one waiting for a word from memory or for a synchronization event, generally causes no overhead—the processor just executes the instructions of some other ready threads.

Pipelined Processors. Each processor in the MTA has 21 stages. As each processor issues an instruction from a different stream at each clock tick, at least 21 ready threads are required to keep it fully utilized. Since the state of up to 128 threads is kept in hardware, this target of 21 ready threads is easy to achieve.

Flat Shared Memory. The MTA has a byte addressable memory. Full/empty tag bits (described below) are associated with 64-bit words. Addresses are scrambled by hardware to scatter them across memory banks [2]. As a result, the memory has no locality, and there are no issues of partitioning data or mapping memory on the machine.

Extremely Fine-grained Synchronization. Each 64-bit word of memory has an associated *full/empty* bit. A memory location can be written into or read out of using ordinary loads and stores, as in conventional machines. Load and store operations can also be under the control of the full/empty bit. For example, a "read-when-full, then set-empty" (`y = readfe[x]`) operation atomically reads data from a location only after that location's full/empty bit is set *full*. The full/empty bit is set *empty* during the read operation atomically with reading the data. If the full/empty bit is not set, the thread executing the read operation suspends (in hardware) and is later retried. The thread resumes when the read operation has completed. This is a low overhead operation since the thread is simply removed from and later reinserted into the ready queue. This feature allows extremely fine-grained synchronization and is detailed in Section 2.1, below.

2.1. Parallel Implementation

Behavior of the Full/Empty (F/E) bits. The operations described below are atomic with respect to reading or writing and changing the state of the full/empty bit.

A synchronized write into a variable succeeds when it is

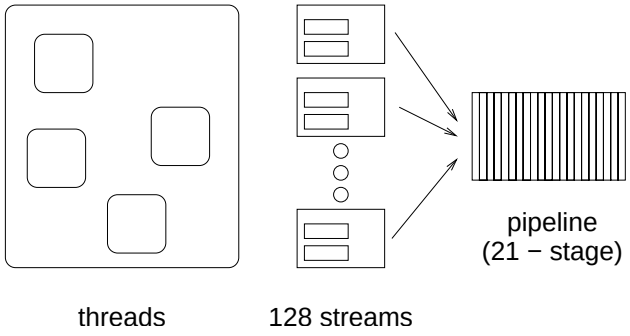

Figure 1. The MTA (1 processor).

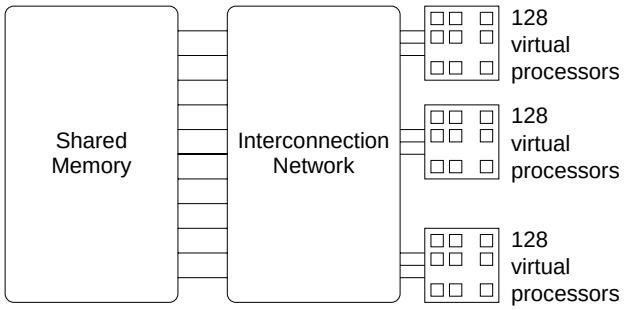

Figure 2. A view of the MTA (multiple processors). Each stream may be thought of as a virtual processor.

empty. If the variable is *full* then, the write blocks until it becomes *empty*. When the write completes, the location is set *full*. A thread attempting a synchronized write into a *full* location will be suspended (by *hardware*) and will resume only when that location becomes *empty*.

A synchronized read from a variable succeeds when it is *full*. If it is *empty*, then the read blocks until it becomes *full*. When the read completes, the location is set *empty*. A thread attempting a synchronized read from an *empty* location will be suspended (by *hardware*) and will resume only when that location becomes *full*.

There are several ways of using the full/empty bits, for example, in `pe[i] = pe[i] - pdiff;` the update to `pe[i]` can be done as follows:

1. Perform a synchronized read of `pe[i]`.
2. Perform the subtraction (in registers).
3. Store the result to `pe[i]` under a synchronized write.

The update to `pe[i]` is guaranteed to be atomic with respect to other loop instantiations wanting to update the same `pe[i]`.

Synchronized Variables can be declared thus:
`sync float pe[100];`.

In this case, writes and reads to/from `pe[]` will follow the full/empty rules given above.

Machine Generics are machine language instructions such as `writeef()` ("wait until a variable is empty, then write a value into it and set the full/empty bit to full") that can be invoked from within Fortran or C.

To ensure that `pe[i] = pe[i] - pdiff;` is handled properly when several threads are using the same value of `i`, we could use `writeef(pe[i],readfe(pe[i])-pdiff));`. Machine generics such as `writeef` and `readfe` become *individual* MTA machine instructions. This technique is the most flexible and gives full control to the programmer. We have used it to great advantage in our codes.

Compiler Directives. Directives can be used to make the compiler use full/empty bits to ensure correct updating. For example, the directive

```
#pragma mta update
pe[i] = pe[i] - pdiff;
```

instructs the compiler to insert, in the statement that immediately follows the directive, appropriate machine instructions to insure that the update to `pe[i]` is atomic.

Compiler Detection. The compiler can also detect program statements where use of full/empty bits would be required and insert the required machine instructions. This is the least intrusive solution but may not work in all cases.

3. Exact Matching

The naive $O(mn)$ exact matching algorithm for matching a pattern P of length m against a text T of length n is easily implemented in parallel on the MTA. The essential code is as follows

```
#pragma mta assert parallel
   for (i=0; i < n-m+1; i++) {
      int j;
      for(j=0; j < m ; j++){
         if (P[j]!=T[i+j])
            break;
      }
      if (j >= m){
         found++;
         F[int_fetch_add(&Ptr,1)]=i;
      }
   }
```

Here `found` is the number of matches; the locations of these matches are stored in array F. The `int_fetch_add` function is *essential* when multiple threads are storing into the same array. The compiler directive `#pragma mta assert parallel` is used to reassure the compiler that it is safe to parallelize the loop that follows.

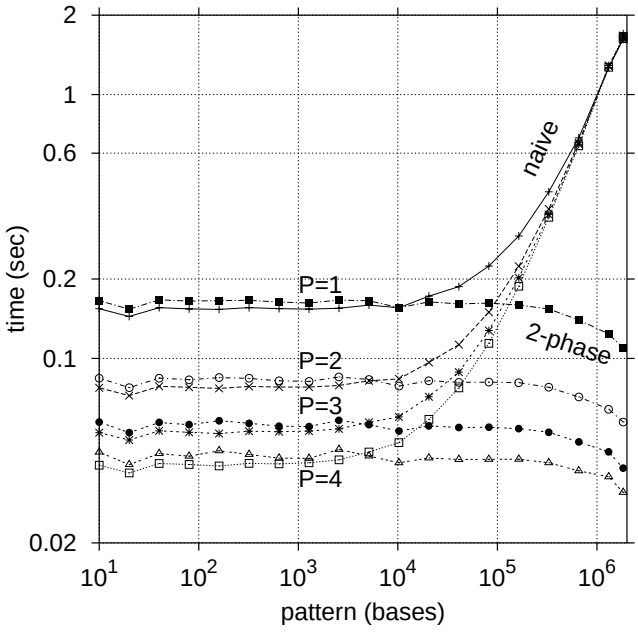

Figure 3. Comparison of the naive and 2-phase algorithms for exact matching on 1–4 processors. Text size is 1.8 million bases.

This implementation gives excellent speedup if $m \ll n$ (Fig. 3). However, when m is large, a serious problem arises. The one, potentially successful, substring is executed by one thread while the remaining threads quickly run out of work. Thus only a very tiny fraction of the power of the machine is applied to the one successful substring. This is a fundamental scheduling issue that arises, for example, in classical job-shop scheduling and also in the cutting stock problem.

This algorithm was tested on the DNA of *H. Influenzae* [7], which has 1.8 million bases. In Fig. 3, the y-axis is time (secs). The x-axis is pattern (substring) size m, varying from 10 to 1.8 million bases. The plots shows the time required to find a substring of *H. Inf*. The substring is chosen to lie at the very end of *H. Inf*, as that is the worst case problem. The figure illustrates that the performance of the naive algorithm shows good speedup for small substring sizes but is useless beyond about 10^4 bases.

The solution to this problem is to implement a two phase algorithm in which the first phase serves to identify a number of potential substring starting points. In the second phase the pattern P is equipartitioned into a number of blocks which are tested in parallel. If all blocks match, success is returned.

We can see in Fig. 3 that the two phase algorithm incurs a

small constant overhead, but otherwise gives excellent performance and speedup. As the substring size becomes large, the time to find a match *decreases*. This is due to two factors. Firstly, the amount of work required to match a pattern of length m against a text of length n is proportional to $(n - m + 1)n$, $m \leq n$. This expression varies from $2(n-1)$ for $m = 2$ to n, for $m = n$. Thus for $m = n$ (extreme right hand points in 2-phase plots), the intrinsic amount of work is halved. Secondly, as the pattern size, m, becomes large, the overhead of parallelization is reduced, as there are fewer potential starting points to consider. Thus there is a dramatic fall in the time required to find a match as the pattern size increases.

4. Approximate Matching using Dynamic Programming

Our implementation of the dynamic programming algorithm for sequence matching is based on the presentation in the seminal text of Gusfield [6]. We match a pattern P of size $|P| = m$ against a text T of size $|T| = n$. The strings P and T are taken for the alphabet `c, t, a, g`. An integer matrix D of size $m \times n$ is used for the actual dynamic programming. The standard recurrence relation is used:

$$
\begin{aligned}
D(i,j) \quad = \quad & \min(\\
& D(i-i,j)+1, \\
& D(i,j-1)+1, \\
& D(i-1,j-1)+ \\
& \qquad \text{if}(\mathrm{P(i)} \neq \mathrm{T(i)}) \text{ then } 1 \text{ else } 0 \\
&)
\end{aligned}
$$

Figure 4 shows the dependencies between the elements of the matrix. It is clear that the updating of matrix D can proceed along rows, columns or antidiagonals, as shown in Fig. 5. On serial computers the choice between row or column order would depend on the way the 2d matrix D was mapped onto 1d memory. On conventional parallel machines, the antidiagonal order would be preferred as it leads to the maximum amount of parallelism.

The Cray MTA-2 has a *uniformly accessible* shared memory and is thus insensitive to row or column order differences. Furthermore, as we shall demonstrate below, its very fine grained synchronization mechanism (based on Full/Empty bits) frees it from the necessity of using antidiagonal order. In fact, for some ranges of problem sizes, antidiagonal order has slightly poorer performance than row order.

5. Implementation

Several variants of the standard dynamic programming algorithm were implemented and evaluated on the 4 processor MTA-2 at ENRI [5] and an 8 processor machine at Cray Seattle. The strings tested were subsets of the genome of *H. Influenzae* [7], which has size $\approx 1.8 \times 10^6$ bases. For each experiment, a substring "text" T of size 2^9 to 2^{15} of this genome was first copied out and then a "pattern" P of equal size was generated by randomly replacing 50% of the bases of T by a random selection from `c, t, a, g`. Pattern P was then aligned against text T.

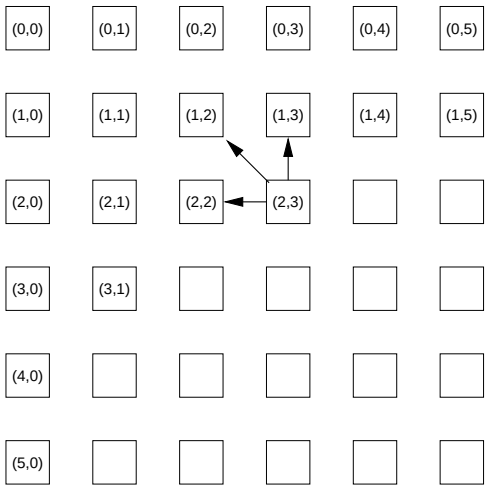

Figure 4. Dependencies in the matrix.

In all, 8 different implementations of the algorithm were

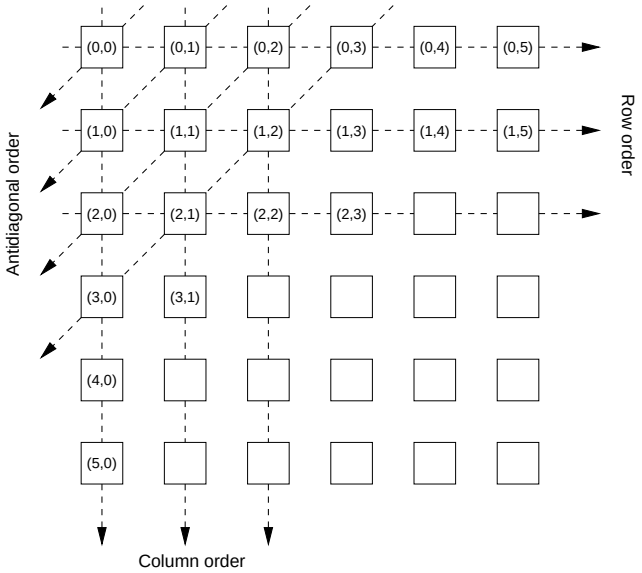

Figure 5. Three different orders for updating the matrix.

tested. The four algorithms that were best for some range of problem sizes are described below. For each of the algorithms we give the relevant portions of the CANAL (Compiler Analysis) output.

5.1 Antidiagonal

The antidiagonal algorithm is listed below. The algorithm iterates serially over the $m + n$ antidiagonals of the matrix. Each element of the antidiagonal is computed in parallel. Synchronization is handled by the compiler (which has to be reassured through the `assert parallel` pragma).

```
     |/* antidiagonal */
     |for(k=1; k<=m+n; k++){
     | int i, j;
 6 - |   if(k<=m){
     |     j = 1;
     |#pragma mta assert parallel
     |     for (i=k ; i >=1; i--) {
 8 SD|       D[i][j]=MIN(MIN
               (D[i-1][j]+1,D[i][j-1]+1),
                D[i-1][j-1]+(P[i]!=T[j]));
     |       j++;
     |     }
     |   }
     |   else{
     |     i = m;
     |#pragma mta assert parallel
     |     for (j=k-m+1 ; j <=n; j++){
10 SD|       D[i][j]=MIN(MIN
               (D[i-1][j]+1,D[i][j-1]+1),
                D[i-1][j-1]+(P[i]!=T[j]));
     |       i--;
     |     }
     |   }
     |}

Loop  6 in dyn at line 24 in region 1

Loop  7 in dyn in loop 6
      In parallel phase 3
      Dynamically scheduled,
         variable chunks, min size = 4

Loop  8 in dyn at line 28 in loop 7
      Loop summary: 6 memory operations,
         0 floating point operations
         9 instructions,
         needs 64 streams for full util.
         pipelined

Loop  9 in dyn in loop 6
      In parallel phase 3
      Dynamically scheduled,
         variable chunks, min size = 4

Loop  10 in dyn at line 36 in loop 9
      Loop summary: 5 memory operations,
         0 floating point operations
         8 instructions,
         needs 68 streams for full util.
         pipelined
```

5.2 Rowwise with F/E Synchronization

In this case, ordinary rowwise order is used. The updating of each element of a row is synchronized through the use of Full/Empty bits. The `readff` and `writeef` operators ensure that no matrix element is updated unless the elements it depends on are themselves ready. Should this not be the case, the special-purpose hardware of the MTA-2 switches over to some other update with zero overhead.

Two variants of this algorithm were implemented. The row-col FE version (shown below) has the text loop on the inside and the pattern loop on the outside. The col-row FE version (not shown) is *vice versa*. There are slight, but intriguing, differences in performance between the two.

```
     |/* rowwise with FE */
     |#pragma mta assert parallel
     |for (i=1; i <=m; i++) {
     |  int j, myPi;
7 p  |  myPi = P[i];
     |  for(j=1; j <= n ; j++){
     |    int v, h, d, m1, m2;
8 DS |    v= readff(&D[i-1][j])+1;
8 DS |    d= readff(&D[i-1][j-1])+
                 (myPi!=T[j]);
8 p- |    m1 = MIN( d, v);
8 DS |    h= readff(&D[i][j-1])+1;
     |    m2 = MIN( m1,h);
8 DS |    writeef(&D[i][j], m2);
     |  }
     |}

Loop   7 in dyn at line 28 in region 1
       In parallel phase 4
       Interleave scheduled

Loop   8 in dyn at line 31 in loop 7
       Loop summary: 5 memory operations,
         0 floating point operations
         12 instructions,
         needs 90 streams for full util.
         pipelined
       3 instructions added
         to satisfy dependences
```

5.3 Antidiagonal with F/E Synchronization

This algorithm combines the best features of preceding two. Iteration through the matrix is in antidiagonal order and Full/Empty bits are used for synchronization.

```
     |/* antidiagonal with FE */
     |#pragma mta assert parallel
     |for(k=1; k<=m; k++){
     |  int i, j=1;
     |  for (i=k ; i >=1; i--) {
     |    int  v, h, d, m1, m2;
10 DS|    v= readff(&D[i-1][j])+1;
10 DS|    d= readff(&D[i-1][j-1])+
                  (P[i]!=T[j]);
10 DS|    h= readff(&D[i][j-1])+1;
10 p-|    m1 = MIN( d, v);
     |    m2 = MIN( m1,h);
10 DS|    writeef(&D[i][j], m2);
     |    j++;
     |  }
     |}
     |#pragma mta assert parallel
     |for(k=2; k<=n; k++){
     |  int i=m, j;
     |  for (j=k ; j <=n; j++){
     |    int  v, h, d, m1, m2;
12 DS|    v= readff(&D[i-1][j])+1;
12 DS|    d= readff(&D[i-1][j-1])+
                  (P[i]!=T[j]);
12 DS|    h= readff(&D[i][j-1])+1;
12 p-|    m1 = MIN( d, v);
     |    m2 = MIN( m1,h);
12 DS|    writeef(&D[i][j], m2);
     |    i--;
     |  }
     |}

Loop  10 in dyn at line 30 in loop 9
       Loop summary: 6 memory operations,
         0 floating point operations
         12 instructions,
         needs 90 streams for full util.
         pipelined.  2 instructions added
           to satisfy dependences

Loop  11 in dyn in region 1
       In parallel phase 4
       Interleave scheduled

Loop  12 in dyn at line 44 in loop 11
       Loop summary: 6 memory operations,
         0 floating point operations
         12 instructions,
         needs 88 streams for full util.
         pipelined. 3 instructions added
           to satisfy dependences
```

6. Experimental Results

The plot in Figure 6 presents the results of our experiments. We vary processors from 1 to 8 and problem size from 512×512 to 32768×32768. We omit the range of problem sizes for which the antidiagonal algorithm is not competitive. In these initial experiments we align equal patterns and texts. In later experiments we explore the performance of our algorithms when the pattern is smaller than the text.

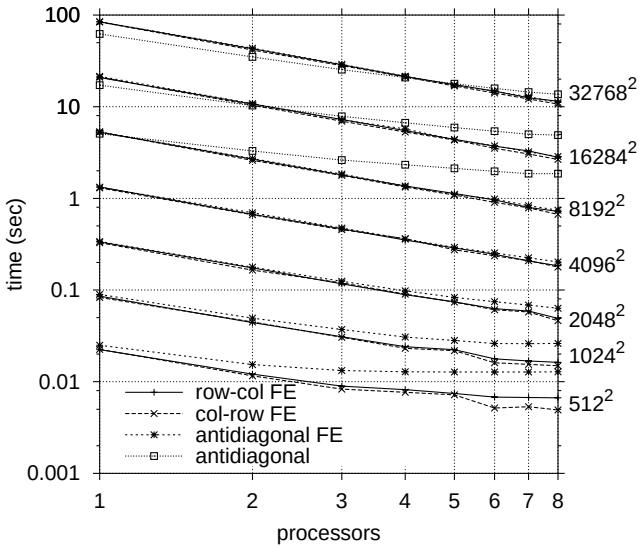

Figure 6. Aligning patterns and texts of equal sizes. Run times of only the dynamic programming phase.

Consider first the two algorithms that use Full/Empty bits, antidiagonal with FE and row-col/col-row with FE. Fig. 6 shows the run times of the dynamic programming phase only (omitting the time for the $O(m+n)$ traceback phase, which is difficult to parallelize). The run times of these algorithms are indistinguishable for problems of size 4096×4096 and larger. Furthermore, these algorithms exhibit almost perfect speedup for these problem sizes. For problems smaller than these, the antidiagonal algorithm (which would be expected to have better performance on a parallel processor) is slightly worse than the rowwise algorithm. This is because the antidiagonal approach "uncovers" parallel work at a rate proportional to the size of the antidiagonals (i.e. $1, 2, 3, \ldots, m, \ldots, 3, 2, 1$) while the rowwise algorithm does so proportionally to the size of the rows ($m, m, m, \ldots$). The Full/Empty mechanism of the MTA is able to attend to those elements of the matrix whose dependencies have been satisfies while holding others until later. A conventional parallel processor would not be able to react to unsatisfied dependencies with such alacrity.

Figure 7 shows the total run time ($O(mn)$ dynamic programming phase, parallelized, plus the $O(m+n)$ traceback phase, not parallelized). As would be expected, the $O(m+n)$ component makes a slight difference for small

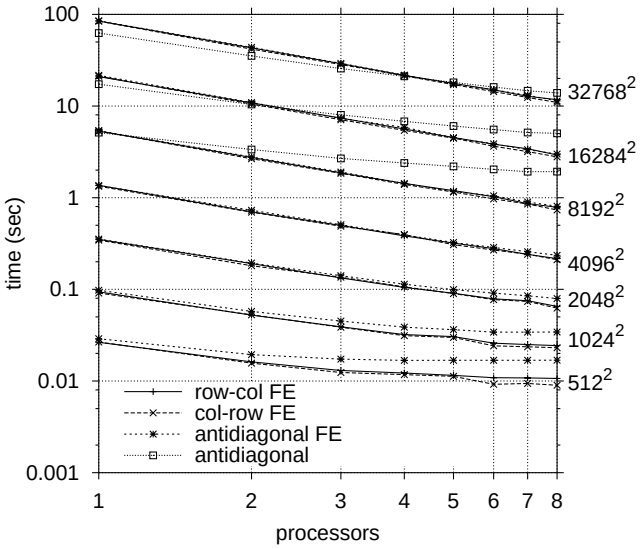

Figure 7. Aligning patterns and texts of equal sizes. Total run times (dynamic programming phase plus traceback).

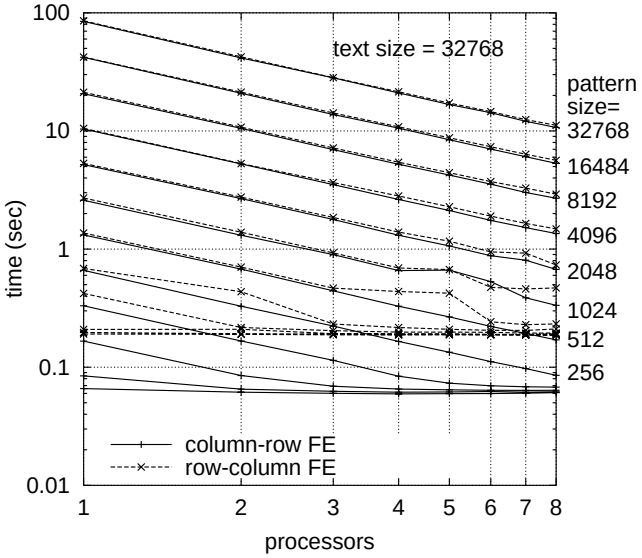

Figure 9. Investigating an interchange in the row/column order. Dynamic programming phase only.

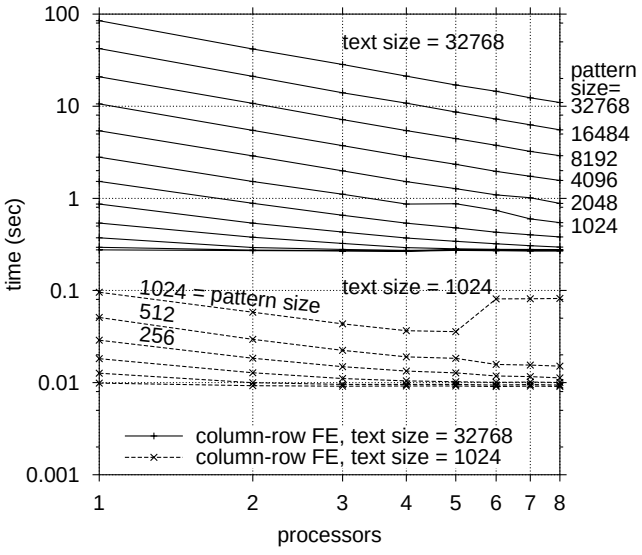

Figure 8. Aligning patterns of various sizes against texts of sizes 32768 and 1024 bases.

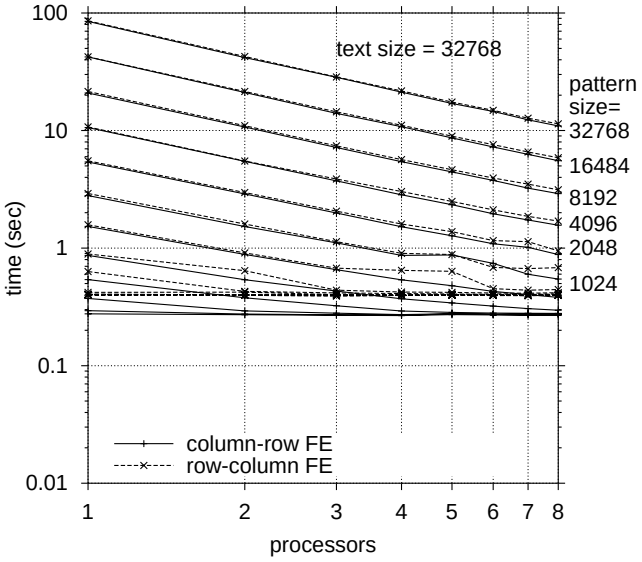

Figure 10. Investigating an interchange in the row/column order. Total time (dynamic programming phase plus traceback).

problems, but is negligible otherwise.

Turning to the "plain" (i.e. without Full/Empty) antidiagonal algorithm, it is seen that it outperforms the others for large problem sizes. This is probably because there is enough parallelism in large problems to keep the MTA busy without having to depend on (and suffer the overhead, however slight, of) the Full/Empty mechanism.

Figure 8 shows the timings obtained when the column-row FE algorithm is run with various pattern sizes aligned against texts of sizes 32768 and 1024 bases. As would have been expected, large problems show perfect speedup. As the pattern size becomes smaller, performance saturates because of lack of available work (i.e. small $m \times n$).

The above figures illustrate some intriguing results obtained while investigating the order in which the dynamic programming matrix is filled in. Fig. 9 shows only the dynamic programming time, while Fig. 10 gives the total time. In Section 6.2 we present the row-column FE version, wherein the iteration is:

```
for (i = 1; i <= m; i++) {
   m pattern elements
   for (j = 1; j <= n ; j++){
      n text elements
   }
}
```

The alternate column-row FE order is:

```
for (j = 1; j <= n ; j++){
   n text elements
   for (i = 1; i <= m; i++) {
      m pattern elements
   }
}
```

It can be seen that the column-row algorithm gives significantly better performance for small pattern sizes. This is presumably because the column-row order is more effective in uncovering parallelism for the hardware. A precise performance model of this phenomenon is an interesting challenge for future research. Figure 10 shows, however, that the advantage of the column-row order is minimized when the traceback time is added in.

7. Conclusions

We have described how the standard dynamic programming sequence alignment algorithm may be ported to the Cray MTA-2. Once the standard algorithm has been ported, many other variants are very easy to implement. For example, after completing the work described in this paper, we were able to port the classical Smith-Waterman algorithm [4] with no more than a few hours of work.

The unusual architecture of the Cray MTA-2 permits easy parallelization of a vast range of algorithms. We expect that future developments in the field of supercomputing will include the MTA and its variants and related architectures. Such machines will be easy to program and will provide attractive alternatives to the more traditional supercomputer architectures that are in vogue today.

The next steps in this research will be

1. Testing of these algorithms on larger MTA-2 configurations.
2. Implementation and testing of a linear (as opposed to quadratic) space alignment algorithm.
3. Exploration of other sequence alignment algorithms that might usefully be ported to the MTA-2.

8. Acknowledgments

We are grateful to the staffs at Cray Research (Seattle), Cray Japan and ENRI for their generous assistance. This work would not have been possible without the support and encouragement of Dick Russel, Bracy Elton and Simon Kahn. We are indebted to David Callahan for his insightful comments. The support provided by Kazuki Watanabe, Toshiomi Muto, Susumu Igawa, Toshiaki Ishimoto and Satoru Yoshioka is gratefully acknowledged.

References

[1] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Proc. Int. Conf. Supercomputing*, pages 188–187, 1992.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proc. Int. Conf. Supercomputing*, pages 1–6, 1990.

[3] S. H. Bokhari, M. A. Glaser, H. F. Jordan, Y. Lansac, J. R. Sauer, and B. V. Zeghbroeck. Parallelizing a DNA simulation code for the Cray MTA-2. In *Proceedings of the IEEE Computer Society Bioinformatics Conference, CSB2002*, pages 291–302, 14-16 August, 2002.

[4] P. Clote and R. Backofen. *Computational Molecular Biology, An Introduction*. John Wiley & Sons, 2000.

[5] ENRI. Electronic Navigation Research Institute. www.enri.go.jp.

[6] D. Gusfield. *Algorithms on Strings Trees and Sequences*. Cambridge University Press, 1997.

[7] H. influenzae. www.tigr.org/tigr-scripts/ CMR2/ GenomePage3.spl?database=ghi.

[8] J. Sauer and B. V. Zeghbroeck. Ultra-fast nucleic acid sequencing device and a method for making and using the same. US Patent No. 6,413,792, July 2, 2002.