Optimising Genomics Codes on the Cray MTA-2

Jon Gibson, *CSAR, University of Manchester*

ABSTRACT: *This paper explains the issues in optimising the genomics codes, Phrap and Cross-Match, for performance on the Cray MTA-2. Phrap assembles shotgun DNA sequence data and Cross-Match compares any two sets of DNA sequences. The performance of Cross-Match is assessed and compared to that of an SGI Origin 2000.*

1. Introduction

The Cray MTA-2

The Cray MTA-2 (Multi-Threaded Architecture) uses multiple lightweight threads on each processor as a novel way to bypass the increasing problem of memory latency on high performance architectures. It can have up to 128 of these threads on each of 16 to 256 processors (noting that the largest machine currently in existence is the 40 processor one at NRL). By switching between the active threads at each clock cycle, the processor is kept busy and the time taken to access memory is not wasted. This avoids the need for caches and the multi-level memory hierarchy normally associated with high performance architectures. In fact, the machine provides a scalable uniform access to a global shared memory, at a bandwidth of 2.4GB/s and with an impressive 4GB of memory per processor. Another desirable feature of the machine is that it is “easy to program”. The parallelism is mainly loop-based and is programmed using directives (in a similar way to OpenMP) or, where possible, implemented automatically by the compiler. Additionally, the MTA's uniform memory access simplifies the task of the programmer, since issues such as data locality and optimal cache usage are not relevant.

The Codes

Phrap, whose name is derived from the initial letters of **phr**agment **a**ssembly **p**rogram, assembles shotgun DNA sequence data, typically from a single input file containing the reads. The process of hierarchical shotgun sequencing is shown in Fig. 1. A library is constructed by fragmenting the target genome and cloning it into a large-fragment cloning vector; here, BAC vectors are shown. The genomic DNA fragments represented in the library are then organized into a physical map and individual BAC clones are selected and sequenced by the random shotgun strategy. Finally, the clone sequences are assembled to reconstruct the sequence of the genome. Note that DNA sequences are made up of four different *nucleotides*, represented by the letters A, C, G and T.

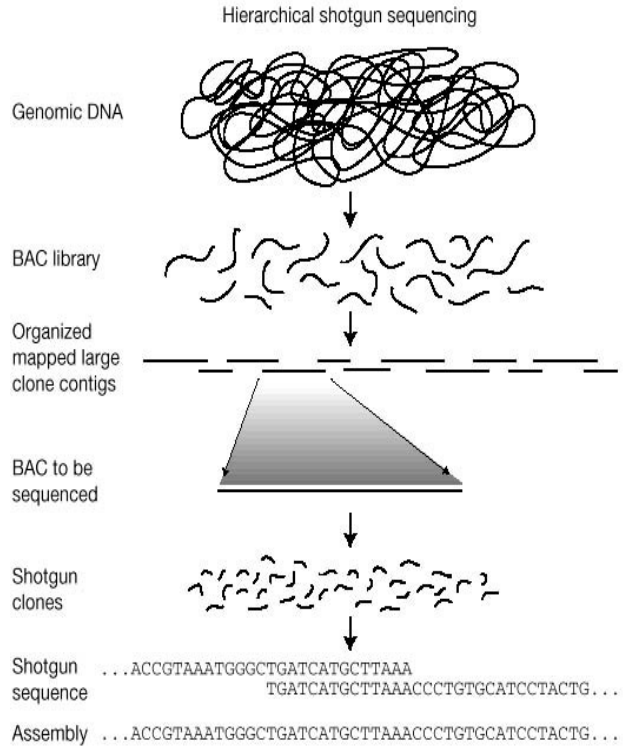
Hierarchical shotgun sequencing
Genomic DNA
BAC library
Organized mapped large clone contigs
BAC to be sequenced
Shotgun clones
Shotgun sequence
...ACCGTAAATGGGCTGATCATGCTTAAA
TGATCATGCTTAAACCCTGTGCATCCTACTG...
Assembly
...ACCGTAAATGGGCTGATCATGCTTAAACCCTGTGCATCCTACTG...

Figure 1. Hierarchical shotgun sequencing (© Nature)

Cross-Match, which is really just a subset of the Phrap code, compares any two sets of (long or short) DNA sequences, looking for possible matches. Typically, it inputs two files: the *query* sequences and the *subject* sequences. However, it can also compare the different sequences against one another in a single input file.

Both of these programs use a banded version of SWAT, an efficient implementation of the **S**mith-**Wat**erman algorithm. The Smith-Waterman algorithm finds the optimal local alignment of any two sequences. It is an iterative, matrix-based calculation, where all possible pairs of residues, one from each sequence, are represented in a 2D array of cells and all possible alignments by pathways through this array. The highest score is assigned to each cell out of those for all the possible paths/alignments leading to it. These scores are based on the previous score for the alignment in question, the similarity score for the pair of residues and any gap penalties, for gaps introduced into the sequence. The optimal local alignment is then determined by tracing the pathway back from the highest scoring cell.

2. Parallelisation of Cross-Match

Cross-Match first of all reads in the sequence and, if present, quality data from input file. It then finds pairs of reads having matching words of a given minimum length. Each such word is used to define a band in the Smith-Waterman matrix, centred on this match. The program eliminates any exact duplicate reads and then does swat comparisons of pairs of reads which have matching words, computing the (complexity-adjusted) swat score. It prints out the matches at the end.

Before we started this work, modifications had already been made to the code so that it would run in parallel on a number of different architectures (specifically: Compaq Alphas, Hewlett Packards, Pentium III PC's, SGI's and Suns). The code applicable to each given architecture was picked out using *#ifdef* statements. The following shows a code sample for running on an SGI machine. The *#pragma*

```
#ifdef SGI
#define    thread_num()  mp_my_threadnum()
#endif
..............
find_all_scores (db, nprocs)
  Database *db; int nprocs;
{
..............
#ifdef SGI
#pragma parallel
#pragma local (entry1,tempseq,i)
 {
#pragma pfor iterate (entry1=ies; ief-ies+1; 1)
#pragma schedtype (dynamic)
#endif
 for (entry1 = ies; entry1 <= ief; entry1++) {
#ifdef SGI
      i = thread_num();
#endif
..............{body of loop}..............
```

lines are SGI directives for parallelising sections of the code, in this case the for-loop over index *entry1*. The *mp_my_threadnum()* function gives the unique number assigned to the thread in question.

The approach to parallelisation is different on the MTA, since all tasks are dynamically scheduled and thread numbers are something that the programmer should not have to consider. However, the existing code allocates memory separately for each thread, which means that only one piece of work with a given "thread number" should be allowed to execute the loop at any one time. There is also the question of how to generate the thread numbers, since the MTA system does not provide anything of easy practical use. The ported code is shown below.

```
#ifdef MTA
  __sync int lock$[MAX_PROCS];  int temp;
#endif
..............
find_all_scores (db, nprocs)
  Database *db; int nprocs;
{
..............
#ifdef MTA
#pragma mta assert parallel
#endif
    for (entry1 = ies; entry1 <= ief; entry1++) {
#ifdef MTA
#pragma mta assert local tempseq, i, temp
      i = entry1%nprocs;
      lock$[i] = 1;
#endif
..............{body of loop}..............
#ifdef MTA
      temp = lock$[i];
#endif
```

The equivalent of the thread number in this code is given by *entry1%nprocs*, the value of the loop counter (each one representing a block of work to be allocated to a thread) modulo the number of threads which can be executed simultaneously (i.e. for which memory has been allocated). Only one of these thread numbers can be executed at any one time because of the lock, which is implemented using an array of *sync* variables, large enough to hold all the thread numbers. When a thread executes the *lock$[i] = 1;* line, it can be thought of as acquiring the lock, so that no other thread with the same *i* value can execute the loop until the lock is released, via the *temp = lock$[i];* statement.

An alternative approach that was looked at here was to replace the loop by a pair of nested loops, with the inner loop being run over *MTA_NUM_THREADS*, the number of available threads. However, this was found to be the inferior alternative since there was a run-down in the thread usage as each set of inner loops was completed.

In order to check what the compiler has done with the code, the programmer can use the **c**ompiler **anal**ysis tool, canal. For the section of code under consideration, it produces the following output.

```
|         |#ifdef MTA
|         |#pragma mta assert parallel
|         |#endif
|         |for (entry1 = ies; entry1 <= ief; entry1++) {
|         |#ifdef MTA
|         |#pragma mta assert local tempseq, i, temp
| 12 p |      i = entry1%nprocs;
| 12 D |      lock$[i] = 1;
|         |#endif
| 12 p |      tempseq = get_seq(entry1);
| 12 D |      find_scores (entry1, tempseq, i);

|Loop 12 in find_all_scores at line 569 in region 9
|  in parallel phase 2
|  interleave scheduled
|  dependences carried by: dot_time
|  dependences carried by: rep_time
|
|Parallel Region 9 in find_all_scores
|  multiple processor implementation
|  requesting at least 115 streams
```

The top half of this output shows the code, with 12 being the loop number. The letters p and D both indicate that the loop will be executed concurrently due to the *assert parallel* directive, with the D's being statements that would otherwise have prevented parallelisation due to possible dependencies. More information is provided in the lower half, including the type of scheduling and the number of streams being requested.

3. Performance of Cross-Match

Upon running the Cross-Match code, of which the *find_all_scores* function in our example forms the major part, we found that parallelism is only limited by the number of reads in the input file and the number of available threads on the machine. We clearly need a large data set to fully exploit the machine and we found the ideal candidate in the form of the Fugu fish, shown in Fig. 2.

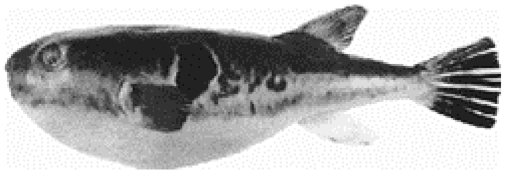
Figure 2. The Fugu Fish

Its DNA has recently been sequenced and the data is freely available. Suprisingly perhaps, it seems that the Fugu fish is genetically remarkably similar to humans, hence it has generated a lot of interest.

Running with a 10MB, 10,000 entry input file on one processor, the diagnostic tool *traceview* shows the following performance for the *find_all_scores* loop.

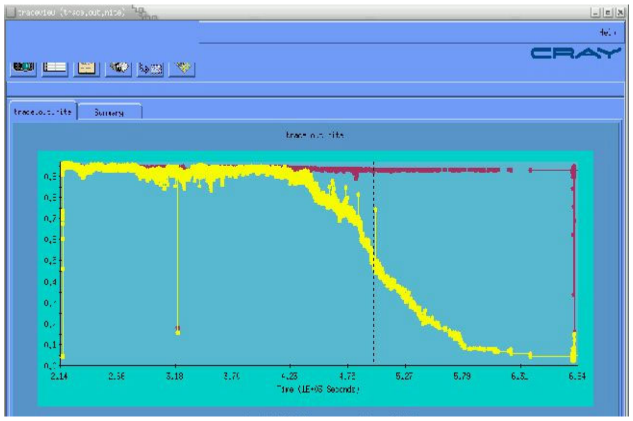

Figure 3. Traceview output.

The dark red line on the graph shows the number of available threads and the yellow line shows the number of threads which are running. The vertical scale is a little unclear but the last point on the scale is at 0.9: 90% CPU usage. The graph shows that the processor was being saturated at first but that there is also a significant run-down time as the available data gets used up and the number of threads actually running decreases. The possibilities for improving the load-balancing here are to be investigated but it becomes less of a problem as the size of the input file increases – 10MB is still relatively small.

Figure 4 shows how the number of threads scales with the number of processors the job is run on. Since all our

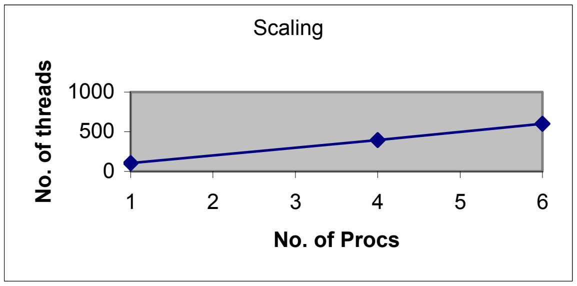

Figure 4. Linear Scaling of Threads

results were from an eight processor machine, we can only show scaling up to six processors, but as long as sufficient data is available, scaling should continue to be linear.

We compared the time that the code took to do 50% of the swatting (i.e. of the swat comparisons) on the MTA with the time taken on an SGI Origin 2000, for runs on one processor. We are comparing the first 50% so that we are only looking at the part of the calculation where the MTA is saturated, i.e. we are striving to compare performance with the effects of problem-size on the MTA being ignored. We found that the MTA did 50% of its swatting in 651 seconds, compared to 360 seconds on the Origin. The code does run well on the Origin and when you consider that it has twice the clock speed, 400MHz as opposed to the MTA's 200MHz, their relative performance seems unsurprising. Clearly, the MTA cannot compete with problems of this size. It is only going to do so if and when the problem size is such that efficient cache use on the Origin is no longer possible. Further investigation is currently being undertaken.

The enormous data sets associated with genomics and the inherently parallel nature of the processing could provide an excellent opportunity for the MTA to show off its potential. However, although it is starting to provide some promising results, the real test will be the comparison of its performance when those really big data sets are run.

Acknowledgements

The author would like to thank Jim Maltby of Cray Inc, Seattle for all his help on the project, as well as everyone else who has been involved, including Steve Jordan and Gail Alverson of Cray, and Stephen Pickles, Mike Pettipher and Keith Taylor of CSAR, University of Manchester.

About the Author

Jon Gibson is a high performance computing consultant at CSAR, Manchester Computing, University of Manchester, U.K. E-mail: jon.gibson@man.ac.uk