Whole Genome Comparison using Commodity Workstations

Arpith C. Jacob[1], Sugata Sanyal[2]

[1]Department of Computer Science and Engineering
Vellore Institute of Technology
Vellore, TN 632014 INDIA
arpith@arpith.com

[2]School of Technology and Computer Science
Tata Institute of Fundamental Research
Mumbai, 400005 INDIA

***Abstract*—Whole genome comparison consists of comparing or aligning two genome sequences in the hope that analogous functional or physical characteristics may be observed. Sequence comparison is done via a number of slow rigorous algorithms, or faster heuristic approaches. However, due to the large size of genomic sequences, the capacity of current software is limited.**

In this work, we design a parallel-distributed system for the Smith-Waterman dynamic programming sequence comparison algorithm. We use subword parallelism to speedup sequence to sequence comparison using Streaming SIMD Extensions (SSE) on Intel Pentium processors. We compare two approaches, one requiring explicit data dependency handling and the other built to automatically handle dependencies. We achieve a speedup of 10 - 30 and establish the optimum conditions for each approach. We then implement a scalable and fault-tolerant distributed version of the genome comparison process on a network of workstations based on a static work allocation algorithm. We achieve speeds upwards of 8000 MCUPS on 64 workstations, one of the fastest implementations of the Smith-Waterman algorithm.

***Index Terms*—Sequence comparison, Dynamic programming, Subword parallelism, MMX.**

I. Introduction

ADENINE, Guanine, Cytosine, and Thymine make up the genetic code of every living organism. The genetic make-up of a large number of organisms is being sequenced, exponentially increasing the amount of available data. Computers play an important role in managing and analyzing this data, including database searching, sequence alignment and structure prediction.

Sequence similarity searches is one of the most performed tasks in Computational Biology. Similarity searches identify closely related sequences from a database, on the assumption that a high degree of similarity often implies similar function or structure. When looking for similar sequences in a database, an alignment score is calculated for every sequence in the database, which represents its similarity with the query sequence. Rigorous algorithms to calculate the optimal alignment score based on the dynamic programming technique are sensitive, but extremely slow on workstations. Hence, faster heuristic alternatives are widely used but have the disadvantage of not being able to detect distantly related sequences. The Smith-Waterman dynamic programming method is used to calculate the optimum local alignment score between a pair of sequences. Due to its compute-intensive nature, they are rarely used for large scale database searches.

Whole genome comparison is the next step toward understanding organisms. By definition, it is the comparison or alignment of two genomic sequences. It is extremely useful for tasks such as deducing the history of evolution of organisms, or determining coding and functional noncoding regions. However, current software is unable to perform this extraordinary task because of the immense computational and space requirements.

In this paper we describe a parallel-distributed implementation of the dynamic programming technique, exploiting the inherent fine-grained and coarse-grained parallelism. We use Intel's MMX/SSE2 a form of subword parallelism, to speedup the algorithm on a single processor, and design a distributed system to distribute the search process on a network of workstations.

II. Related Work

Parallelization of the Smith-Waterman database search algorithm proceeds on two fronts: *fine-grained* and *coarse-grained* parallelism. In the fine-grained approach the pairwise comparison algorithm is parallelized, where each processing element performs a part of the matrix calculation to help determine the optimal score. This is most widely used in single instruction stream, multiple data stream computers where each processor executes the same instruction simultaneously with little communication overhead between the processors. For example each processing element may be assigned a single character of the query sequence, while the database is shifted linearly through the processing elements, to perform the comparison in O(M + N) time. In multiple instruction stream, multiple data stream computers, the coarse-

grained approach is used where subsets of the database is independently searched on the processing elements. Each processing element is assigned an equal portion of the database and uses the query sequence to evaluate the optimal scores. The success of the coarse-grained approach depends on the load balancing strategy used.

Five architectures used for sequence comparison are described in Hughey [1]: special purpose VLSI, reconfigurable hardware, programmable co-processors, supercomputers and workstations.

Special purpose VLSI provides the best performance but is limited to a single algorithm. The Biological Information Signal Processing system (BISP) was one of the first systolic array processors for high speed sequence comparison. Others include SAMBA and Fast Data Finder (FDF).

Reconfigurable hardware are based on field programmable gate arrays (FPGAs) or similar designs. They are more versatile than special purpose VLSI and can be adapted for different algorithms, however such reconfiguration is a laborious task. Examples include Splash, DeCypher and Bioccelerator based on FPGAs, and MGAP based on its own architecture.

Programmable co-processors strive to balance the flexibility of reconfigurable hardware with the speed and high processing element density of special purpose VLSI. PIM is a programmable co-processor with 64 1-bit PEs in a chip. Kestrel is a 512 element array of 8-bit PEs capable of performing a wide variety of tasks including sequence alignment using the Smith-Waterman algorithm.

Supercomputers can be effectively used for sequence analysis. The MasPar MP-2 is a single instruction stream, multiple data stream computer that executes the same instruction simultaneously on all its processing elements. BLAZE, an implementation of the Smith-Waterman algorithm, was written for the MasPar MP1104 supercomputer containing 4096 processors. Alpern, Carter & Gatlin [2] suggested the use of microparallelism on the Intel Paragon i860 to pack four numbers in a single 64-bit Z-buffer register.

Workstations are inexpensive and widely available, but can be very slow. Uniprocessor integer implementations on Sun UltraSparc and DEC Alpha workstations give modest speeds. Wozniak [3] presented an implementation that used the SIMD visual instruction set of Sun UltraSparc microprocessors to simultaneously calculate four rows of the dynamic programming matrix. Rognes and Seeberg [4] used SIMD multimedia extension instructions on Intel Pentium microprocessors to produce one of the fastest implementations on workstations. Networks of workstations have also been used effectively to distribute the database search process on a number of workstations. Strumpen [5] used a massively parallel approach to distribute the database search process in a heterogeneous environment on more than 800 workstations in the internet. Martins & Cuvillo [6] presented an event-driven multithreaded implementation of the sequence alignment algorithm for the EARTH architecture, on a Beowulf cluster of 128 Pentium Pro microprocessors.

With the advent of cheap and powerful workstations with parallel processing capabilities, it has become possible to realize substantial speed improvements for the sequence comparison algorithm. In this work, two fine-grained approaches using the Intel Pentium 4 microprocessor's MMX/SSE2 technology are reviewed. A coarse-grained distributed system integrating the fine-grained technique is built which achieves speeds comparable with the best implementations of the Smith-Waterman algorithm on special purpose hardware.

Sequence B

Sequence A		T	C	G	A	C	A	T	A
	0	0	0	0	0	0	0	0	0
A	0	0	0	0	5	0	5	0	5
C	0	0	5	0	0	10	3	1	0
T	0	5	0	1	0	3	6	8	1
A	0	0	1	0	6	0	8	2	13
G	0	0	0	6	0	2	1	4	6
G	0	0	0	5	2	0	0	0	0
C	0	0	5	0	1	7	0	0	0
A	0	0	0	1	5	0	12	5	5

Fig. 1. Comparison Matrix: Optimal score: 13, Match: 5, Mismatch: –4, Penalty: 0 + 7k.
Optimal Alignment: A C A T A
A C _ T A

III. The Smith-Waterman Algorithm

Needleman & Wunsch [7] and Sellers [8] introduced the global alignment algorithm based on the dynamic programming approach, which was the first step in sequence comparison algorithms. Smith & Waterman [9] introduced an $O(M^2N)$ algorithm to identify common molecular subsequences, which was able to take into account evolutionary insertions and deletions. Gotoh [10] modified this algorithm to run in O(MN) time by considering affine gap penalties. Each of these algorithms depended on saving the entire M * N matrix in order to recover the alignment. The large space requirement problem was solved by Myers & Miller [11] who presented a quadratic time and linear space algorithm, based on the divide and conquer approach. Aho, Hirschberg and Ullman [12] proved that such comparison algorithms, which compare symbols to see if they are equal or unequal, have to take time proportional to the product of their string lengths.

Two sequences A and B of length M and N respectively are compared using a substitution matrix ∂, and an affine gap weight model. The gap penalty is given by: $W_i + kW_e$ where

$W_i > 0$ and $W_e > 0$. W_i is the penalty for initiating a gap and W_e is the penalty for extension of the gap, which varies linearly on the length of the gap. The substitution matrix ∂ lists the probabilities of change from one nucleotide or amino acid into another in the sequence. There are two widely used families of matrices: the Percent Accepted Mutation matrices (PAM) and the Block Substitution Matrices (BLOSUM).

A maximization relation is used in order to calculate the optimum local alignment score. The optimal local alignment score is calculated using the following recurrence relations:

$$E(i, j) = F(i, j) = H(i, j) = 0 \qquad \text{if } i = 0 \text{ or } j = 0$$

$$E(i, j) = \max \left\{ \begin{array}{l} E(i - 1, j) - W_e \\ H(i - 1, j) - W_i - W_e \end{array} \right\}$$

$$F(i, j) = \max \left\{ \begin{array}{l} F(i, j - 1) - W_e \\ H(i, j - 1) - W_i - W_e \end{array} \right\}$$

$$H(i, j) = \max \left\{ \begin{array}{l} 0 \\ E(i, j) \\ F(i, j) \\ H(i - 1, j - 1) + \partial(A_i, B_j) \end{array} \right\}$$

The highest value in the H matrix gives the optimal score. One of a possible many optimal alignments can be retrieved by retracing the steps taken during the H matrix computation, from the optimal score back to a zero term. The recurrence can be understood as follows: the E matrix holds the score of an alignment that ends with a gap in sequence A. When calculating the $E(i, j)^{th}$ value, both extending an existing gap by one space, or initiating a completely new gap is considered. The F matrix calculations can be explained in a similar fashion. The $H(i, j)^{th}$ cell value holds the best score of a local alignment that ends at position A_i, B_j. Hence, alignments with gaps in either sequence, or the possibility of increasing the alignment with a matched or mismatched pair are considered. A zero term is added to the recurrence relation in order to discard negatively scoring alignments and restart the local alignment.

To quantify the performance of dynamic programming algorithms, the performance measure *millions of dynamic programming cell updates per second* or *MCUPS* is defined. It represents the number of cells in the H matrix that can be computed per second, and includes all memory operations and corresponding E/F matrix cell evaluations. It is calculated as $(M * N) / t_{dp} / 10^6$ where t_{dp} is the time taken in seconds, to evaluate the entire H matrix and return the optimal score.

IV. Fine-Grained Approach

General-purpose microprocessors have been evolving to include media processing in their domain of workloads. Media processing programs have ever increasing demands to handle real-time animation, voice and video. Multimedia extensions have been added to the Instruction Set Architectures (ISAs) of most microprocessors to support such workloads [13]. These extensions exploit the concept of subword parallelism where the basic unit of computation is split into subwords, with each unit capable of performing the same operation simultaneously. This is a form of single instruction stream, multiple data stream (SIMD) parallel processing ideally suited for applications that exhibit a high level of data parallelism. They are usually implemented in hardware and cost very little in terms of extra hardware requirements.

General-purpose microprocessors represent a low cost, flexible solution compared to pure hardware solutions that are significantly costlier, hard to program and are restricted to a single algorithm. Perhaps more importantly, general-purpose microprocessors are already widely available which makes it an even more attractive option to exploit. Implementations of the Smith-Waterman algorithm using multimedia extensions typically take the fine-grained approach in parallelizing the pairwise comparison of two sequences. Wozniak [3] used the visual instruction set (VIS) of Sun UltraSparc microprocessors to simultaneously calculate four rows of the dynamic programming matrix. The implementation achieved speeds of 18 MCUPS on a single 167Mhz UltraSparc microprocessor, a two-fold speedup over the algorithm implemented using integer instructions on the same machine. They also implemented a coarse-grained version, LASSAP (a LArge Scale Sequence compArison Package), on a SUN Enterprise 6000 server with 12 processors achieving speeds of 200 MCUPS. Rognes and Seeberg [4] used a different approach to the parallelization of the Smith-Waterman algorithm, using multimedia extensions (MMX) and achieved speeds of 150 MCUPS on a single Pentium III 500Mhz microprocessor. This represented a six-fold speedup relative to the sequential algorithm using integer instructions on the same hardware.

A. Multimedia Extensions on Intel Microprocessors

Intel introduced the Pentium MMX microprocessor [14] in 1997 making cheap SIMD processing available for workstation applications. MMX (MultiMedia eXtensions) technology aliased the eight 64-bit MMX registers with the floating point registers of the x87 FPU. MMX technology allows up to eight byte operations to be performed in parallel. SIMD processing was enhanced with the addition of SSE2 (Streaming SIMD Extensions) on the Pentium 4 microprocessor. SSE2 is capable of handling sixteen simultaneous byte operations in its 128-bit XMM registers.

The basic integer instructions are extended into SIMD versions and include arithmetic, shift, comparison, data transfer, and conversion instructions. The speedup is achieved by allowing the same operation to be performed on multiple data elements. Because of the smaller number of bits available for each data type, overflows or underflows occur more

frequently. They are handled by hardware in the Intel architecture by two methods: *wraparound arithmetic* and *saturation arithmetic*.

Wraparound arithmetic like integer operations in C, simply truncate the most significant bit when an out of range result occurs. In effect, the result value is always taken modulo 2^n where each subword is of n bits. A more attractive solution for most applications is saturation arithmetic, where the result saturates at an upper or lower bound. When the result of an operation overflows the width of any subword, the result is limited to the largest representable value. Similarly, if an underflow occurs, the result is limited to the smallest representable value. Hence, for unsigned integer data types of n bits, underflows are clamped to 0, and overflows to $2^n - 1$. Saturation arithmetic is advantageous because it offers a simple way to eliminate unneeded negative values or perform arithmetic operations automatically limiting results without causing errors, and is used in our implementation.

Compiler support for SIMD instructions is still rudimentary. Some compilers such as the Intel C/C++ compiler provide intrinsics that support SIMD instructions. The Intel C/C++ compiler can also detect and optimize simple loops that exhibit parallelism and automatically generate vectorized code that use SIMD instructions. Specialized libraries for common functions utilizing SIMD operations have also been released by various vendors. Most applications however require careful parallelization of algorithms, and hand coding using SIMD instructions utilizing assembly language. Such code is not portable among different microprocessors and has to be rewritten for different processor platforms.

B. Challenges in Parallelizing the Smith-Waterman Algorithm

Parallelizing the dynamic programming algorithm is done by calculating multiple rows of the H matrix simultaneously. Data dependencies that exist within this parallel computation must be handled appropriately. Figure 2 shows the data dependencies of each cell in the H matrix. The value in the (i, j)th cell depends on (i – 1, j – 1)th, (i – 1, j)th and (i, j – 1)th cell values. Hence, before a cell in the matrix can be computed, the cells immediately above, to the left and diagonally across must be available.

The alignment matrix H, can be evaluated in parallel rows, columns or anti-diagonals. Elements in an anti-diagonal depend solely on the previous anti-diagonal in the matrix, hence the problem of data dependencies is automatically handled. Computation can proceed on parallel wave fronts diagonally across the alignment matrix. Proceeding in horizontal rows or vertical columns presents a challenge for parallel computation since cells in a row or column depend on other cells in the same row or column. This method however, does have its advantages once the data dependencies are appropriately handled.

C. Diagonal Approach

When computation proceeds diagonally across the alignment matrix the interdependencies are automatically handled, and thus is an ideal solution for parallel computation.

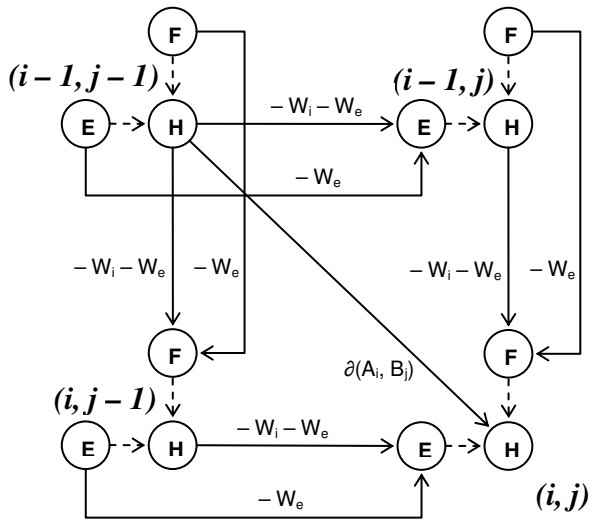


Fig. 2. Data dependencies in the similarity matrix.

The main disadvantage lies in the fact that the loading of the substitution scores for a diagonal cannot be parallelized. The substitution scores cannot be accessed linearly from memory, but have to be independently loaded for each cell in the diagonal. The symbols from the two sequences that correspond to a particular cell in the alignment matrix have to be read, and a look-up into the substitution table be made in order to calculate the corresponding match or mismatch score. This procedure has to be repeated for every element in the diagonal before parallel computation can proceed. This non-linear memory access pattern adversely affects the performance of the algorithm.

The second disadvantage is that the size of the diagonal varies at the beginning and the end of the matrix sweep. If for example computation occurs in parallel for four cells of a diagonal of the alignment matrix at a time, the calculation of the first and last three diagonals presents a problem. The first three diagonals of the alignment matrix with one, two and three cells respectively do not present enough cells to load the 4-way SIMD word. An elegant way to overcome the non-uniformity of computation is to add three dummy symbols both at the beginning and the end of the query sequence. Parallel computation of the alignment matrix can then proceed uniformly through the length of the query sequence, without the need to take care of special cases. Appropriate entries must be added in the similarity table between the dummy symbol and each symbol in the alphabet, the dummy symbol included, with a score of zero so that the optimal score remains unchanged.

Computation proceeds along successive diagonals of length equal to the SIMD word through the entire length of the query sequence. The next four rows of the matrix are then computed in the same manner. This proceeds along the entire length of the database sequence in steps equal to the SIMD word length. If the database sequence length is not a multiple of the SIMD word length, the sequence must be concatenated at the end with an appropriate number of dummy symbols. The speedup

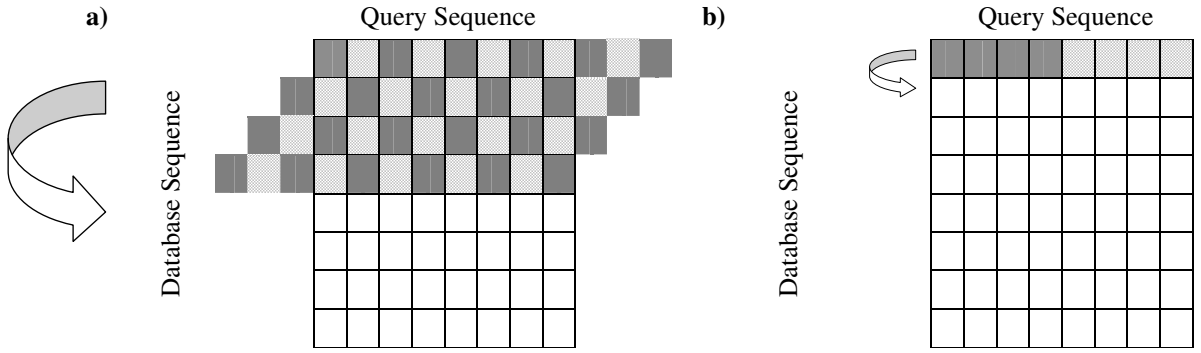


Fig. 3. Fine-grained parallelization of the Smith-Waterman algorithm using 4-way subword processing. a. Diagonal approach. b. Horizontal approach.

achieved over the sequential version is due to the simultaneous computation of four rows. The number of loop iterations is reduced by a factor of four and along with it, the number of memory references and other loop overhead.

Figure 4.a shows the algorithm for the comparison of a query sequence with a database sequence using SIMD processing by the diagonal method. Because alignments are not computed, the whole matrix need not be stored in memory. To calculate the i^{th} row of the H matrix, only the $(i - 1)^{th}$ row of the E and H matrix along with the scalars VF (F[i][j - 1]), VHC (H[i][j – 1]) and VHD (H[i – 1][j – 1]) are required. Each scalar is a SIMD word register capable of holding SIMD_WIDTH subwords. Since saturated arithmetic is used elements in the H matrix are automatically clamped to zero, hence the zero term is not required in the recurrence relation.

SIMD operations can be performed with signed or unsigned integers. Except for certain elements in the substitution matrix, all results of computation are strictly greater than or equal to zero. Representation of negative numbers requires an extra bit and thus reduces the maximum representable integer value by half. To overcome this problem, all elements in the similarity matrix are biased by a positive value so that no element is below zero. The bias can then be subtracted during computation without affecting the optimum score. The maximum score that can be computed by the algorithm using a SIMD word holding integers of n bits is: $2^n - 1 - BIAS$. Any score equal to this value must be recomputed using a higher precision algorithm.

In order to obtain optimum performance, a number of techniques have been used to speedup the code. Performance reduces drastically if the code and data that is referenced in the inner loop of the algorithm is not contained in the first level cache of the microprocessor. The two E and H arrays are interleaved together as an array of structures as it is faster to access memory in sequential order. SIMD words in memory are aligned at appropriate boundaries to maximize performance. 64-bit memory access with MMX registers requires the target address to be aligned at 8 byte boundaries and 128-bit memory access with XMM registers requires alignment at 16 byte boundaries. Non-aligned memory access reduces performance drastically, and in some cases may raise exceptions. Calculation of the substitution score vector provides a major performance hurdle due to its non-linear memory access pattern. Each subword requires one addition, one multiplication and two memory reference instructions in the outer loop, along with three memory read instructions in the inner loop. These instructions are suitably rearranged to reduce the effects of memory latency.

D. Horizontal Approach

When computation proceeds horizontally along the rows of the alignment matrix the interdependencies are not resolved. To calculate the value of the $(i, j)^{th}$ cell in the H matrix, the values of the $(i, j - 1)^{th}$ cell in the F and H matrices are required. This makes it impossible to calculate horizontal cells of the H matrix in parallel.

An interesting empirical observation on the working of the Smith-Waterman algorithm for biological sequences (*SWAT optimization*) was made by Phil Green and later implemented in the SWAT program. In most cells of the E, F and H matrices, the values are less than zero (clamped to zero when using saturated arithmetic) and thus do not contribute to H. As can be seen from the recurrence below, the $(i, j)^{th}$ cell value in the F matrix will remain zero if the $(i, j - 1)^{th}$ cell value is already zero, so long as $H(i, j - 1) <= W_i + W_e$.

$$F(i, j) = \max \left\{ \begin{array}{l} F(i, j-1) - W_e \\ H(i, j-1) - W_i - W_e \end{array} \right\}$$

If the H value is below this threshold, F will remain zero along the row and need not be taken into account for the rest of the iteration. The effectiveness of this speed optimization depends heavily on the threshold value. If the gap open and gap extend penalties are very small, most H values will be above the threshold and there will be no speedup in the algorithm.

The SWAT optimization offers a solution to the horizontal data dependency problem when calculating the H values in parallel. For example, when using a 4-way SIMD word, the F values can be ignored from the iteration if the four H values in its relation are below the threshold $W_i + W_e$. If one or more of the H values exceeds this threshold, the F values must be recalculated using a sequential process.

a)

```
int sw_simd_diag( char *query, int m, char *dbseq,
                  int n, int **subst_matrix,
                  int gap_init, int gap_ext,
                  int bias )
{
  int i, j;
  int E[m + 2 * SIMD_WIDTH - 2];
  int H[m + 2 * SIMD_WIDTH - 2];
  vector_int VE, VHC, VHD, VF, VH,
             Vsubst_matrix, Vgap_init,
             Vgap_ext, Vbias, Vscore;

  Vgap_init = { gap_init, ..., gap_init };
  Vgap_ext = { gap_ext, ..., gap_ext };
  Vbias = { bias, ..., bias };

  for ( i = 0; i < m + 2 * SIMD_WIDTH - 2; i++ )
  {
    E[i] = 0;
    H[i] = 0;
  }

  Vscore = { 0, ..., 0 };
  for ( i = 0; i < n / SIMD_WIDTH; i++ )
  {
    VF = VHC = VHD = { 0, ..., 0 };
    VE = { 0, ..., E[SIMD_WIDTH - 1] };
    VH = { 0, ..., H[SIMD_WIDTH - 1] };

    for ( j = 0; j < m + SIMD_WIDTH - 1; j++ )
    {
      VF = MAX( VF, VHC - Vgap_init ) - Vgap_ext;
      VE = MAX( VE, VH - Vgap_init ) - Vgap_ext;

      Vsubst_matrix =
      { subst_matrix[query[j + SIMD_WIDTH - 1]]
                    [dbseq[i * SIMD_WIDTH]],
        ...,
        subst_matrix[query[j]]
                    [dbseq[i * SIMD_WIDTH +
                     SIMD_WIDTH - 1]]
      };

      VHC = MAX( VE, VF,
                 VHD + Vsubst_matrix - Vbias );
      VHD = VH;
      Vscore = MAX( Vscore, VHC );

      E[j] = VE >> ( SIMD_WIDTH - 1 );
      VE = ( VE << 1 ) | E[j + SIMD_WIDTH];
      H[j] = VHC >> ( SIMD_WIDTH - 1 );
      VH = ( VHC << 1 ) | H[j + SIMD_WIDTH];
    }
  }

  return MAX( Vscore[0], ...,
              Vscore[SIMD_WIDTH - 1] );
}
```

b)

```
int sw_simd_horiz( int **qprof_matrix, int m,
                   char *dbseq, int n,
                   int gap_init, int gap_ext,
                   int bias )
{
  int i, j;
  int E[m], H[m];
  vector_int X, T, VE, VF, VH,
             Vgap_init, Vgap_ext, Vbias, Vscore;

  Vgap_init = { gap_init, ..., gap_init };
  Vgap_ext = { gap_ext, ..., gap_ext };
  Vbias = { bias, ..., bias };

  for ( i = 0; i < m; i++ )
  {
    E[i] = 0;
    H[i] = 0;
  }

  Vscore = { 0, ..., 0 };
  for ( i = 0; i < n; i++ )
  {
    X = VF = { 0, ..., 0 };

    for ( j = 0; j < m / SIMD_WIDTH; j++ )
    {
      VH = H[j];
      VE = E[j];
      VE = MAX( VE, VH - Vgap_init ) - Vgap_ext;

      T = VH >> ( SIMD_WIDTH - 1 );
      VH = X | ( VH << 1 );
      X = T;

      VH = VH +
           qprof_matrix[dbseq[i]]
                       [j * SIMD_WIDTH] - Vbias;
      VH = MAX( VH, VE );

      VF = ( VH << 1 ) |
           ( VF >> ( SIMD_WIDTH - 1 ) );
      VF = VF - Vgap_init - Vgap_ext;

      if ( VF > 0 )
      {
          T = VF;
          while ( T > 0 )
          {
            T = ( T << 1 ) - Vgap_ext;
            VF = MAX( VF, T );
          }
          VF = MAX( VH, VF );
          VF = MAX( VH, VF + Vgap_init );
       }
       else
       {
          VF = VH;
       }
       H[j] = VH;
       E[j] = VE;

       Vscore = MAX( Vscore, VH );
    }
  }

  return MAX( Vscore[0], ...,
              Vscore[SIMD_WIDTH - 1] );
}
```

Fig. 4. Parallel Smith-Waterman algorithm. a. Diagonal approach. b. Horizontal approach.

An advantage with the horizontal method is that the loading of the substitution scores is greatly simplified. The substitution table is no longer referenced for loading the substitution scores for each subword of the SIMD word. Instead, the scores can be loaded with a single memory read operation using a query sequence profile table. The query sequence profile table contains the substitution scores of the query sequence placed horizontally across the matrix, versus an imaginary sequence made up of all symbols in the alphabet. The query sequence profile is created once for the query sequence and is used for pairwise comparison with all sequences in the database. When loading the horizontal substitution scores for the recurrence in the inner loop, the query sequence profile table is referenced, indexed by the database symbol of the current row and the query sequence position of the inner loop iteration.

Figure 4.b shows the algorithm for the comparison of a query sequence with a database sequence using SIMD processing by the horizontal method. Much of the optimization techniques used in the diagonal method are relevant here. The query sequence profile table is computed once before the database comparison procedure and is usually small enough to fit in the first level cache of the microprocessor. The conditional loop presents a problem because it is cumbersome to implement using Intel's media processing ISA. Further, it increases the runtime because of the possibility of misprediction of the branch target address. Thus, the SIMD conditional loop is unrolled.

E. Results

The two algorithms were implemented using MMX and SSE2 technology and tests were carried out on a Pentium III 500Mhz with 128MB RAM running Windows 2000, and a Pentium 4 1.4Ghz machine with 128MB RAM running Windows NT. MMX technology provides 64-bit SIMD processing, which were used to pack eight 8-bit numbers and four 16-bit numbers. SSE2 technology uses 128-bit SIMD processing which provides for the possibility of packing sixteen 8-bit numbers or eight 16-bit numbers in a single SIMD word. The user interface, file handling and memory allocation code was written in C and compiled using the Visual C/C++ 6.0 compiler. The Smith-Waterman algorithm was written in assembly language and compiled using the Netwide ASeMbler 0.98.08 (NASM).

In the tests, the local alignment score between two DNA sequences was calculated without recovering the alignment. The pam47 substitution matrix was used which assigns a value of +5 for a match and -4 for a mismatch between two nucleotides. A bias value of +4 was used to eliminate negative elements from the substitution matrix. An affine function 0 + 7k was used for the gap open and gap extension penalties.

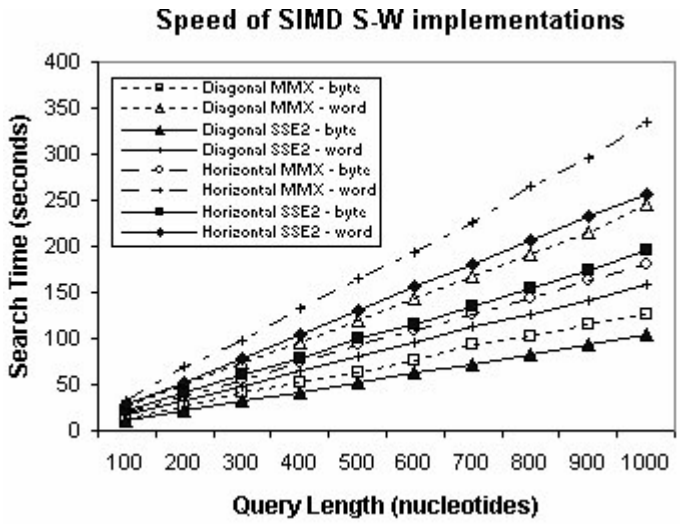
The tests were performed using query sequences ranging in length from 100 – 1000 nucleotides varying in steps of 100. The annotated Drosophilia genome release 3.0 [15], containing 17,878 sequences with a total of 28,249,452 nucleotides was used for performing the database search. Timings were measured by reading the microprocessor timestamp counter before and after completion of the target function. The timestamp counter is incremented at every clock cycle and is read using the assembly mnemonic RDTSC. The difference of the two counter values divided by the microprocessor clock speed in hertz, gives the time in seconds taken by the function to execute. For each test, the total program runtime, total I/O overhead, total time spent in the Smith-Waterman function, and their average times were noted. The speed in MCUPS for comparing a query sequence of length M and a database sequence of length N using the Smith-Waterman algorithm, is calculated as: $(M * N) / t_{dp} / 10^6$ where t_{dp} is the time taken in seconds to execute the algorithm.

Plots of search times versus query lengths for different SIMD implementations on the Pentium 4 machine are shown in figure 5.a. The bulk of the program time (96-97%) is spent in the Smith-Waterman sequence comparison function. Only a small percent of the time is spent as overhead for reading the sequences from disk. For a gap penalty of 0 + 7k, the diagonal method was found to be 1.30 to 1.87 times faster than the horizontal method. Using the 128-bit XMM registers on processors with SSE2 technology doubles the size of the SIMD word as compared to the 64-bit MMX registers of the older MMX technology. Theoretically, this should offer a two fold speed increase as compared to MMX. Practically, the speedups ranged from 1.17 to 1.40 as with an increase in the SIMD word length there is a corresponding increase in clock cycles. Surprisingly, the horizontal approach using byte precision on SSE2 technology was slower than its MMX implementation by 14%.

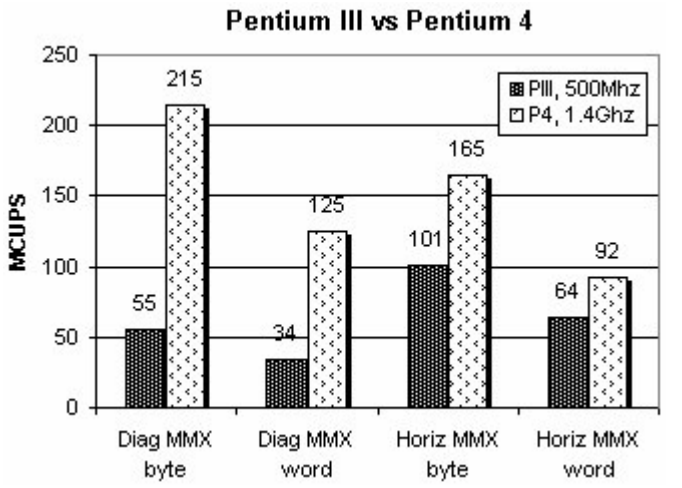
As expected, searches using 8-bit subwords in the SIMD word as compared to searches using subwords of 16-bits were found to be faster by a factor of 1.31 to 1.79. Most comparison scores in sequence searches are well below the maximum value representable in 8 bits. Any score close to 255 represents an interesting match which is investigated by other means, irrespective of its actual score. Hence in most cases, byte precision is sufficient for database searching.

Another interesting observation made is the scalability of the SIMD implementation between processors in the same family. Figure 5.b shows the speedup of the MMX implementations of the two methods when run on a Pentium 4 microprocessor as compared to a Pentium III microprocessor. The diagonal approach using MMX technology experiences a performance boost of 3.68 and 3.91 for the byte and word precisions respectively. The horizontal approach achieves more modest speedups of 1.44 and 1.63 for the byte and word precisions. SIMD processing is an integral part of the modern microprocessor, and will get faster with newer implementations. Because the two approaches are CPU bound, faster SIMD implementations in newer microprocessors will result in a corresponding performance increase for the application.

a)



b)



c)

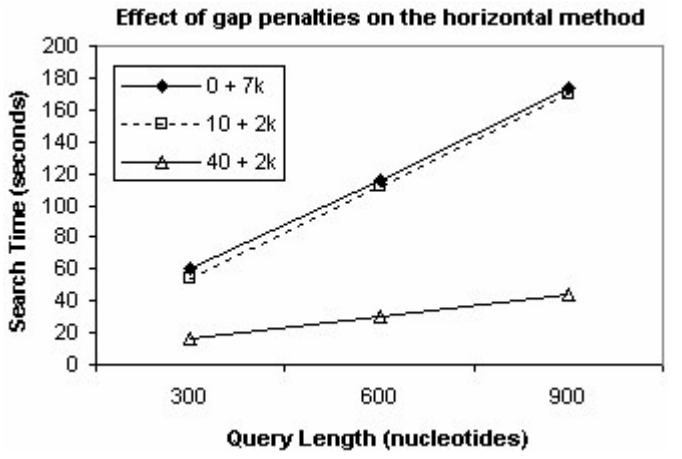


Fig. 5. Performance of parallel implementation. a. Search times versus query lengths for different implementations. b. Scalability of subword processing on Intel microprocessors. c. Effect of SWAT optimization.

Figure 5.c shows the effect of gap penalties on the horizontal method. As mentioned before, a higher gap penalty increases the probability of the F value in a row remaining at zero, since the threshold $W_i + W_e$ is increased. As more of the F matrix becomes zero, the horizontal method benefits since it has to do less computation and there is a drastic speed increase. Searching the database using a query of length 900 with a gap penalty of 0 + 7k takes 174 seconds, while a search with a penalty of 40 + 2k takes only 43.9 seconds (however, with the change in the gap penalty the optimal scores are no longer equal). The speed of the horizontal method varies from 142 MCUPS with a gap penalty of 0 + 7k, to its saturation point at approximately 580 MCUPS with a gap penalty of 40 + 2k. The diagonal method on the other hand does not incorporate the SWAT optimization and experiences constant speeds for different gap penalties. Hence, database searching with a high gap penalty favors the horizontal method over the diagonal method.

Finally, the various implementations of the Smith Waterman algorithm are compared in table 1. The linear version using integer registers offers a speed of only 9 MCUPS which is impractical for searching large databases. Parallel SIMD implementations offer excellent speedups ranging from 10 to 62 times the sequential code depending upon the parallelization method, SIMD technology, precision and gap penalty. Searching a database containing mostly unrelated sequences favors using byte precision, as this provides the maximum speed. For sequences that generate comparison scores that saturate with byte precision, the slightly slower word precision implementation may be used. Finally, the diagonal implementation is favored over the horizontal implementation for low gap penalties as the horizontal implementation degenerates due to the low threshold.

V. Coarse-Grained Approach

The speedup experienced for database searching on general-purpose microprocessors though significant, is insufficient for searching large databases. Whole genome comparison – the comparison of two sets of sequences from two organisms – is severely limited, as much higher speeds are required. A flexible, high speed and easily available solution at a reasonable cost is desired.

An obvious answer is to harness the power of a network of workstations. Networks of workstations equipped with SIMD processing capabilities, are becoming increasingly available in universities, companies and research institutions. These workstations are typically left idle for a number of hours in a day, when they can be used more productively as a sort of virtual supercomputer. The coarse-grained approach is taken in speeding up the search process, where the target database is divided equally among the available workstations in the network. Here, each workstation calculates the optimal comparison scores between the query sequence set and its assigned portion of the database. Since the coarse-grained approach permits workstations to search the database independently, there is less interprocessor communication overhead and hence better scalability.

TABLE 1
SPEEDUP OF VARIOUS SMITH-WATERMAN IMPLEMENTATIONS

Algorithm	Technology	Precision	Penalty	MCUPS	Speedup [a]
Linear	Integer	DWord	0 + 7k	9	1
Diagonal	MMX	Byte	0 + 7k	215	24
Diagonal	MMX	Word	0 + 7k	125	14
Diagonal	SSE2	Byte	0 + 7k	266	30
Diagonal	SSE2	Word	0 + 7k	175	19
Horizontal	MMX	Byte	0 + 7k	165	18
Horizontal	MMX	Word	0 + 7k	92	10
Horizontal	SSE2	Byte	0 + 7k	142	16
Horizontal	SSE2	Word	0 + 7k	108	12
Horizontal	SSE2	Byte	10 + 2k	153	17
Horizontal	SSE2	Byte	40 + 2k	559	62

[a]Tests carried out on a Pentium 4, 1.4Ghz processor

In this work, we are interested in parallelizing the database search process using the dynamic programming technique efficiently on a LAN, containing dedicated workstations capable of SIMD processing. An architecture with the following characteristics is desired: good performance (speedup and scalability), evenly distributed load, robustness and tolerance to faulty workstations.

A. The Distributed Smith-Waterman Database Comparison Architecture

The distributed architecture (figure 6) supporting coarse-grained database searching consists of a master, an ftp server and a number of workers, connected by an interconnecting network. The master is the point of control of the system. Before the start of computation, the master divides the database sequences into a number of buckets according to the number of workers expected. The subdivided buckets are then uploaded by the master onto the ftp server. The ftp server is the data repository and facilitates data exchange between the master and the workers. The master assigns a work packet to each worker, which includes details such as the location of the sequence database on the ftp server, the substitution matrix, gap penalties and the location on the ftp server where the comparison scores are to be uploaded once the search process is completed.

1) The Master Process: The master process is started by passing the two genome (termed arbitrarily as the query and database sequence sets) to be compared, the substitution scores and gap penalty to be used, the url and path to the ftp server, and the number of workers expected for client side computation. The master divides the database into equal sized buckets using the bucket workload balancing technique, and uploads them to the specified path on the ftp server. The query sequence set is uploaded without splitting.

The master process contains three types of threads: the control, reception and communication threads. The control thread is the main interface between the user and the distributed database search system. It responds to user commands, updates the display and keeps track of the amount of work left on the individual workers. The reception thread acts as the rendezvous point between the master and the workers. After connection establishment and initial handshake, a new worker is assigned a particular bucket to search. The communication thread is the channel for communication between the master and a worker. Status information is constantly streamed at regular intervals (set by the user) from the worker to the master. The master uses this information to update a running display of work completed, and the speed of the distributed system. Upon completion of the database search, the control thread downloads the comparison scores from the ftp server and collates the data into a single result file. The collation process is a time consuming task as the scores from the different workers have to be assembled in the original order of the database.

2) The Data Exchange Server: The ftp server is used to handle the data exchange between the master and the workers. Once equal portions of the database have been created, they are stored along with the query sequences on the ftp server for retrieval by the workers. Similarly, the comparison scores generated on each worker is written to a file and uploaded to the ftp server. Since the master is always lightly loaded no matter how many workers are connected to it, and the ftp server is loaded only at the start and completion of the search process, both can be run on the same workstation.

3) The Worker Process: The workers are processes, each running on an independent workstation. The worker process contains two threads: the control and computation threads. The control thread takes care of communication with the master. It gets the search parameters, downloads its bucket file and query sequence set to the local hard disk, constantly updates the master on the progress of the worker and uploads comparison scores once the database search is complete. The control thread also updates the master on the progress of computation at regular time intervals. The computation thread performs the Smith-Waterman sequence comparison procedure between the sequences in the downloaded bucket file and the query sequence set. Functions for sequence comparison that use SIMD processing are chosen from one of the various implementations presented in the previous section. Maximum time must be spent by the worker process in the computation thread rather than in the control thread to increase efficiency.

B. Load Balancing

The success of the coarse-grained approach is highly dependent on the load balancing strategy employed, which must be able to assign approximately equal portions of the database to each workstation. Further, a static allocation algorithm is desired because of the high communication overhead between workstations associated with dynamic allocation algorithms. A slightly modified version of the "bucket" method suggested by Yap, Frieder and Martino [16] which is a combination of the static allocation "portion" method and dynamic allocation "master-worker" methods. Yap defines the percentage of load imbalance (PLIB), to compare the effectiveness of a workload balancing technique as:

$$\text{PLIB} = \frac{\text{Largest load - Smallest load}}{\text{Largest load}} \times 100$$

PLIB is the time difference between the fastest finishing and slowest finishing workstations. A good workload balancing technique must have a PLIB close to zero. PLIB signifies the degree of parallelism achieved. A lower PLIB achieves better parallelism and thus a better speedup.

In the portion method, the database is partitioned into a number of portions proportional to the number of workstations. Division of the sequences in the database into equal portions presents a problem since the sequences are of varying lengths. Depending on the number of nucleotides in the database and the number of workstations present, an ideal portion size is calculated. Sequences from the database are added to a portion until the total length exceeds the ideal size by X percent, after which allocation moves to the next portion. This procedure is continued until all sequences are added to one of the portions. This method has low communication overhead since the allocation is done statically before the computation begins. However, it doesn't achieve a PLIB close to zero.

In the master-worker method, a master process is assigned the task of dynamically distributing sequences and collecting comparison scores from workstations. The master sends the next largest available sequence in the database to a workstation when a request is made. Sequences are distributed to free workstations until the comparison scores for the entire database are calculated. This method has a low PLIB but a very high communication overhead. The master may also become a source of system bottleneck if there are too many requests from a large number of workstations, resulting in idle workstation times, i.e. it has poor scalability.

The bucket method is a static equivalent of the above dynamic allocation scheme, where the database is partitioned into buckets similar to the portion method. The database decomposition procedure differs as follows:

The sequences in the database are sorted in descending length order. Starting from the longest sequence, each one is placed into a bucket that has the smallest value for the function: $t_b = (\sum n_b) / S_{MCUPS}$ where $\sum n_b$ is the current sum of sequence lengths in the bucket, and S_{MCUPS} is the speed of the workstation to which the bucket is assigned. The function takes into account the heterogeneity of workstations in a network by adding the speed of sequence comparison into the function. This brings PLIB close to zero and the algorithm has a very low communication overhead. In case of a homogeneous network, the speed term can be omitted from the equation, simplifying it to the Yap algorithm that aims to minimize the difference in the total sequence lengths among the buckets. Because the work is divided before computation begins, the number of workstations and their speeds of comparison must be known in advance. Another potential problem is the degradation of this scheme due to faulty workstations. If even a single workstation process terminates during computation the time for database comparison doubles, because the corresponding bucket can only be allocated to another workstation once it completes its existing workload (i.e. PLIB becomes 50%).

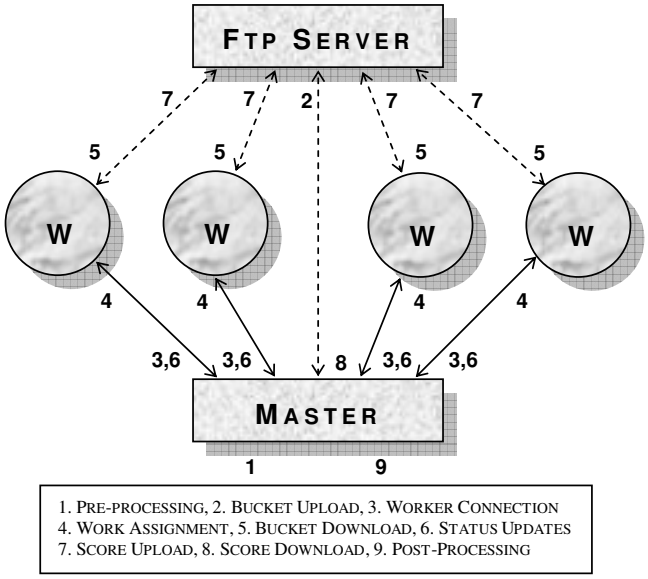


Fig. 6. The distributed Smith-Waterman architecture.

C. Implementation Details

The Mithral client-server software development kit [17] was used to build the distributed system. The development kit provides platform independent network functions (TCP/IP connections), thread control and file system functions. The libcurl [18] library compiled as a DLL was used to provide file transfer support through the ftp protocol. The code was written in C and assembly, and compiled in Visual C/C++ 6.0 and the Netwide ASeMbler 0.98.08 (NASM) on Windows NT. The decision was taken to produce windows executables due to the largely Windows NT workstations used in the laboratory.

The system was used for whole genome comparison, where one set of annotated sequences of an organism was compared against another. One of the genomes (termed the database set) is split into equal sized buckets, with sequences in each bucket being compared against the other genome (termed the query set). A metadata file is created for the database which contains the bucket number in which each sequence of that genome is placed. It is used to collate the scores generated by the workers into a single file, in the original order of the genome.

The SIMD Smith-Waterman implementation used is crucial in determining the speed and precision of the distributed system. For whole genome comparison good precision of the comparison scores is desired. The byte and word precision SSE2 implementations of the diagonal method are used. The sequence comparison score with the byte implementation is first calculated. If saturation occurs, the score is reevaluated with the word implementation. Hence, the range for the possible values of the comparison score is between 0 and (65,535 – BIAS). The number of high scoring (greater than 255 – BIAS) pairs of sequences affects the speed of the distributed system since they will have to be reevaluated using

TABLE 2
PERFORMANCE OF "BUCKET" LOAD BALANCING ALGORITHM: EXPERIMENT 1

Workstations	Largest Load (nucleotides)	Smallest Load (nucleotides)	PLIB[a]
2	1825501	1825497	0.000219
4	912759	912744	0.001643
8	456402	456348	0.011832
16	228231	228132	0.043377
32	114120	114003	0.102524

[a]Genome: Bacillus Subtilis 168, Sequences: 4100, Nucleotides: 3650998

TABLE 3
PERFORMANCE OF "BUCKET" LOAD BALANCING ALGORITHM: EXPERIMENT 2

Workstations	Largest Load (nucleotides)	Smallest Load (nucleotides)	PLIB[a]
16	100691	100593	0.097327
32	50388	50286	0.202429
48	33630	33516	0.338983
64	25233	25116	0.463679

[a]Genome: Haemophilus Influenzae, Sequences: 1739, Nucleotides: 1610500

the word precision implementation. Since this particular distributed system was run on a network of homogeneous workstations, the pure homogeneous metric was used for the load balancing algorithm.

D. Results

The experiments were carried out on 64 Windows NT workstations in the GA, MIS and ME computer labs of the T.S. Santhanam Computing Centre. A separate Windows workstation was used to run the master process and a workstation running Redhat Linux was used to run the ftp server. Each node is a single Pentium 4 microprocessor running at 1.4Ghz, with 128MB RAM. The interconnection network is switched 100Mbps ethernet.

The first experiment carried out was the whole genome comparison of the annotated Bacillus Subtilis 168 genome (4100 DNA sequences; 3,650,998 nucleotides) with the Mycoplasma Genitalium G-37 genome (484 DNA sequences; 528,750 nucleotides). A total of 1,370,339 (69%) pairwise comparisons produced scores below the byte saturation level, while 614,061 (31%) pairwise comparisons had to be reevaluated with the word implementation. The second experiment carried out was the whole genome comparison of the annotated Escherichia coli k-12 genome (4405 DNA sequences; 4,130,746 nucleotides) with the Haemophilus influenzae genome (1739 DNA sequences; 1,610,500 nucleotides). A total of 1,529,612 (77%) pairwise comparisons produced scores below the byte saturation level, while 454,788 (23%) pairwise comparisons had to be reevaluated with the word implementation.

1) Evaluation of the "bucket" Load Balancing Technique: Tables 2 and 3 quantify the performance of the "bucket" load balancing technique for the two experiments in terms of PLIB. The tables display the size of the largest and the smallest buckets in nucleotides and the corresponding PLIB value.

Excellent parallelism is achieved by the bucket technique with PLIB remaining very close to zero for all experiments. The difference between the largest and the smallest load, which is at the most around a hundred nucleotides, determines the extra time that will be taken to complete the task. The extra time in seconds that will be taken by the slowest workstation is calculated as: (Largest load – Smallest load) * Size of query genome / S_{MCUPS}. Even for the worst performing case this is not in excess of a few seconds. PLIB is dependent on the genome that is split into buckets. In whole genome comparison, PLIB must be calculated for both sets of sequences, and the best performing set selected for creating the buckets. The performance of the load balancing technique in practice is also dependent on the composition of the sequences in the genome. The number of high scoring sequences in a bucket affects the number of reevaluations performed on a workstation with the word implementation of the comparison algorithm. It is difficult to determine in advance the sequences that will have a comparison score above the saturation level of the byte implementation, but in general longer sequences have a higher probability of generating larger scores. The bucket method takes this into consideration by first sorting the database sequences in descending order before allocation.

2) Fault Tolerance: The ability to detect and overcome intermittent or permanent faults with the workers is crucial in the successful operation of the distributed system. The distributed system described is equipped with basic facilities to overcome these faults. Intermittent faults caused by network connectivity problems or heavy loading on a worker are detected upon its failure to report status information to the master. Permanent faults caused by the termination of a worker process on a workstation, results in the loss of the work done by that worker. The distributed system reassigns the bucket to a new worker when one becomes available. However, because the work is assigned statically by the bucket load balancing technique, the performance of the distributed system suffers drastically if even a single worker process terminates.

3) Speedup: Figure 7 shows the absolute speedup of the distributed system on a number of workstations. The absolute speedup compares the runtime of the parallel program to the best sequential program, so that all overhead for parallelization is taken into consideration. Excellent speedup of linear nature is observed due to the independence of the workers and the excellent load balancing technique used.

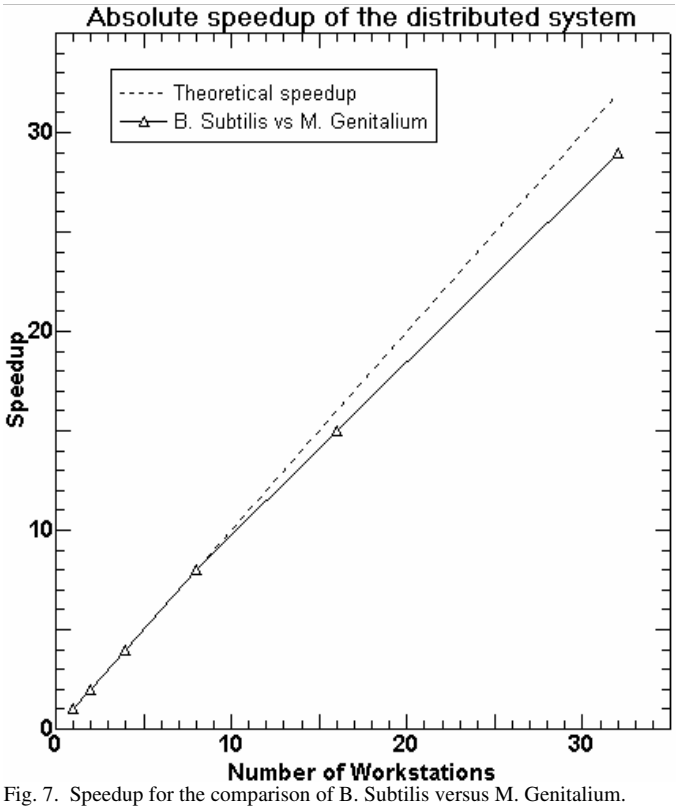


Fig. 7. Speedup for the comparison of B. Subtilis versus M. Genitalium.

TABLE 4
EXPERIMENT 1: RUNTIME COMPARISON OF B. SUBTILIS VS M. GENITALIUM

Workstations	Runtime (seconds)	Maximum MCUPS	Average MCUPS	MCUPS / Workstation[a]
1	11337	170	170	170
2	5569	350	347	173
4	2842	708	679	170
8	1444	1400	1337	167
16	745	2730	2591	162
32	396	5343	4875	152

[a]Genome 1: Bacillus Subtilis 168, Sequences: 4100, Nucleotides: 3650998
Genome 2: Mycoplasma Genitalium G-37, Sequences: 484, Nucleotides: 528750

TABLE 5
EXPERIMENT 2: RUNTIME COMPARISON OF E. COLI VS H. INFLUENZAE

Workstations	Runtime (seconds)	Maximum MCUPS	Average MCUPS	MCUPS / Workstation[a]
16	2472	2886	2691	168
32	1311	5639	5074	159
48	1358	7090	4899	102
64	1233	8076	5395	84

[a]Genome 1: Haemophilus influenzae, Sequences: 1739, Nucleotides: 1610500
Genome 2: Escherichia coli k-12, Sequences: 4405, Nucleotides: 4130746

Tables 4 and 5 show the runtimes of the two experiments. Maximum MCUPS is the maximum instantaneous speed achieved during the distributed computation, while Average MCUPS is the average speed of the distributed system taking into account the time overhead spent for creating the buckets and collating the output generated by the workers. MCUPS / WORKSTATION measures the processor efficiency for the distributed system. Experiment 1 produces excellent results for tests performed on up to 32 workstations. Speeds obtained are linear, and excellent processor efficiency is observed. The actual speeds obtained however, are highly dependent on the composition of sequences in the genome.

Experiment 2 compares genomes that are much larger in size and is a more time consuming task. It produces good speeds on up to 32 machines after which there is a significant drop in performance. There are many reasons for this performance drop. Firstly, a large number of workstations connecting simultaneously to the ftp server causes network congestion and overloads the ftp server. A simple workaround is to start groups of workers at different intervals of time. Secondly, there is a small overhead for splitting the genome into buckets at the beginning of computation. This is typically in the order of 60 seconds and increases with the number of workstations used. However, the significant reason for this performance drop is due to the time taken in collating the output produced by the different workers into a single result file. This is in the order of up to 300 seconds for 64 workers, and significantly decreases the performance of the distributed system if there is not a lot of work to be done. However, during collation of the output, the distributed system has completed the computation and the workers are no longer in use.

The power of the distributed system is illustrated by the maximum speeds achieved on various workstations (figure 8). A maximum speed of 8076 MCUPS is achieved using 64 workstations, a speedup of approximately 2.8 when compared to 16 workstations. Even higher speeds are possible for genomes of different compositions. Such speeds have until now been achievable only with costly, special purpose hardware.

Table 6 compares the performance of the dynamic programming problem on various machines. The distributed system described in this paper, stands second only to the special purpose hardware solution, BioSCAN. The price of the distributed system is a misnomer since in most institutions a large number of workstations are already available and can be used with little to no additional cost. The advantage of this system is the high speeds achieved with widely available hardware.

E. Limitations

It is worthwhile documenting the limitations and problems encountered when using the workstation cluster for distributed processing. As mentioned before, the ftp server and the interconnecting network must be capable of servicing a large number of clients simultaneously. The ftp server is a source of bottleneck during the beginning and end of the computation. This problem can however be alleviated by starting the clients in different batches, so that the load on the ftp server is reduced. A potential problem is in the availability of a large number of machines. Since the workstations in a general laboratory are used by numerous people at varying times it

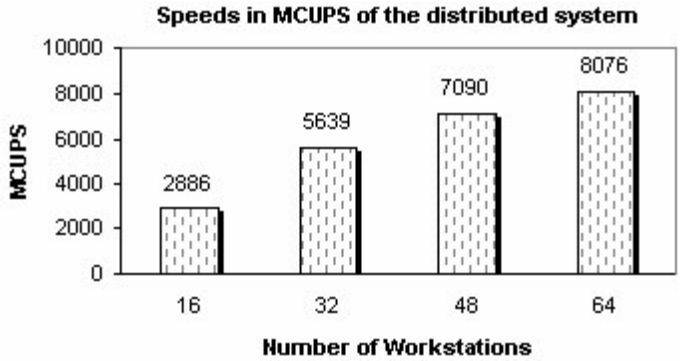


Fig. 8. Speeds achieved in the comparison of E. Coli versus H. Influenzae.

may be impossible to reserve a large number of workstations for a specific interval of time. Strategic time intervals when there is less demand must be targeted. Another practical problem encountered was the failure of the power supply source. Because sequence comparison is a CPU intensive task, distributed computing on a workstation cluster increases the power consumption of the CPUs and can heavily load the power supply.

VI. Conclusion

The aim of this work was to test the feasibility of using parallel features of commonly available workstations for comparing whole genome sequences. A number of algorithms and implementations were designed using multimedia extensions for SIMD processing to achieve significant speedups on Pentium workstations.

A distributed system using a cluster of Pentium 4 workstations was built integrating the SIMD implementations. Excellent performance characteristics were observed with speeds comparable to the fastest implementations on special purpose hardware. General-purpose microprocessors are constantly being updated with the latest technology which can offer significant performance boosts. The newly introduced Simultaneous Multi-threading (hyper-threading) technology available on Pentium 4 microprocessors offers further potential speed increases.

Acknowledgment

A. C. Jacob thanks the manager of Centre for Technical Support at T.S. Santhanam Computing Centre, Muthu G., and the laboratory personnel Mr. Ravikumar V. (GA), Mr. Elson Jeeva T. (MIS), Mr. Srinivasan V. and Ms Dharani (ME) for their gracious help in providing the extensive equipment required to conduct the various experiments in this work.

TABLE 6
Comparison of various Sequence Comparison Architectures [1]

System	Type [a]	PEs	MCUPS	Cost (k$)
BioSCAN	SP	12992	25000	20
64 Pentium 4s[b]	WC	64	8076	68
BISP	SP	256	3200	20
Kestrel	PC	1024	1600	30
DeCypher II-15	RH	1920	1400	173
SAMBA	SP	128	730	60
Mercury - 2	SP	64	640	-
Maspar MP - 2	GS	16684	500	1000
Biocelerator - 1	RH	16	250	66
FDF – 3	SP	3360	230	50
Paragon	GS	32	25	500
5 Alpha AXP300s	WC	5	17	50

[a] GS - Supercomputer, PC - Programmable co-processors, RH - Reconfigurable Hardware, SP - Special-purpose VLSI, WC - Workstation Cluster, WS - Workstation.
[b] Our distributed implementation

References

[1] Richard Hughey, "Parallel hardware for sequence comparison and alignment," Computer Applications in the Biosciences, 12(6):473—479, 1996.

[2] Bowen Alpern, Larry Carter, and Kang Su Gatlin, "Microparallellism and high-performance protein matching," In Proceedings of Supercomputing '95, San Diego, California, December 3–8, 1995. ACM SIGARCH and IEEE Computer Society.

[3] Andrzej Wozniak, "Using video-oriented instructions to speed up sequence comparison," Computer Applications in the Biosciences, 13(2):145—150, 1997.

[4] Torbjorn Rognes and Erling Seeberg, "Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors," Bioinformatics, 16(8):669—706, 2000.

[5] Volker Strumpen, "Parallel molecular sequence analysis on workstations in the internet," Technical report, Department of computer science, University of Zurich, 1993.

[6] Wellington S. Martins, Juan B. del Cuvillo, Francisco J. Useche, Kevin B. Theobald, and Guang R. Gao, "A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison," In Proceedings of the Pacific Symposium on Biocomputing, 311—322, 2001.

[7] Saul B. Needleman and Christian D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two sequences," Journal of Molecular Biology, 48(3):443—453, 1970.

[8] Pattern H. Sellers, "On the theory and computation of evolutionary distances," SIAM Journal of Applied Mathematics, 26:787—793, 1974.

[9] Temple F. Smith and Michael S. Waterman, "Identification of common molecular subsequences," Journal of Molecular Biology, 147(1):195—197, 1981.

[10] Osamu Gotoh, "An improved algorithm for matching biological sequences," Journal of Molecular Biology, 162(3):705—708, 1982.

[11] Eugene W. Myers and Webb Miller, "Optimal alignments in linear space," Computer Applications in the Biosciences, 4(1):11—17, 1988.

[12] Aho, A.V., D.S. Hirschberg and J. D. Ullman, "Bounds on the complexity of the longest common subsequence problem," Journals of the ACM, 23(1):1—12, 1976.

[13] Ruby B. Lee, "Multimedia extensions for general-purpose processors," Proceedings IEEE Workshop on Signal Processing Systems, 9—23, 1997.

[14] Alex Peleg, Sam Wilkie and Uri Weiser, "Intel MMX for multimedia PCs," Communications of the ACM, 40(1):25—38, 1997.

[15] Berkeley Drosophila Genome Project, private communication, 2003. Available: http://www.fruitfly.org/sequence/sequence_db/na_whole-genome_CDS_dmel_RELEASE3.FASTA.gz

[16] Tieng K. Yap, Ophir Frieder, Robert L. Martino, "Parallel computation in biological sequence analysis," IEEE Transactions on Parallel and Distributed Systems, 9(3):283—293, 1998.

[17] Mithral – cs-sdk, private communication, 2003. Available: http://www.mithral.com/projects/cosm/

[18] Libcurl, private communication, 2003. Available: http://curl.sf.net/