Scalable Self-Tuning Implementation of Smith-Waterman Algorithm for Multicore CPUs

Faisal Sikder and Dilip Sarkar

***Keywords*-CPU Oblivious, Scalable Algorithm, Multicore CPU, OpenMP, Parallel algorithm, Shared memory architecture, Smith-Waterman algorithm.**

***Abstract*—Improved version of the Smith-Waterman algorithm (SWA) is most widely used for local alignment of a pattern (or query) sequence with a Database (DB) sequence. This dynamic-programming algorithm is computation intensive. To reduce time for computing alignment score matrix, parallel versions have been implemented on GPUs and multicore CPUs. These parallel versions have shown significant speedup when compared with their corresponding sequential versions.**

Our initial evaluation of an OpenMP parallelization of SWA has shown linear speedup on multicore CPUs, but a closer look at performance data from both sequential and parallel versions have revealed two undesired effects: (i) As the length of the DB sequence increases, the number of elements of the alignment score matrix H computed in per unit time initially increases, then reaches a maximum, and finally decreases continuously; (ii) the length of the DB sequence where decline starts is different for different CPUs. To overcome the computation rate decline we have proposed a run-time self-tuning algorithm. It determines the length, l, of a DB sequence that maximize computation rate during execution time. Then, divide computation of H into computation of a set of submatrices, such that the number of columns in each submatrix is about l.

Our study also found that the number of per-core-threads that delivers the highest rate of computation differs from CPU to CPU. Our proposed algorithm determines optimal number of threads during execution time and creates optimal number of threads for highest possible computation rate. Our extensive evaluations of the proposed self-tuning algorithm on three different multicore multi-CPU shared memory machines have shown significant performance improvement.

I. Introduction

Now most, if not all, computers are powered with one or more multicore CPUs. Since each of these cores can execute independent task, they collectively may increase computing power of a CPU. However, increase in computing power depends not only on the number of cores available, but it also depends on organization of the cores and the cache in the CPU, shared-memory access mechanism, and performance of translation lookahead buffer (TLB) used for virtual address translation.

While utilization of these cores for multiple *independent tasks* is straight forward, *efficient utilization* of these cores to solve *one large problem* requires a parallel algorithm that creates multiple tasks for concurrent execution on these cores. Moreover, for computation-intensive algorithms as the problem size grows memory access time may increase and instruction execution rate may reduce and performance may fail to scale up. Thus, implementation of parallel algorithms on a multicore CPU brings a new challenge for computation-intensive problems. Especially, for algorithms that use arrays of dimension two or higher as data structure, such as, matrix operations and dynamic programming.

Furthermore, as more cores are added to a single CPU, the designers alter cache organization, cache size, and interfaces between cache and cores. Also, memory access methodology changes. Thus, an efficient parallel algorithm implemented to solve a problem on one multicore CPU may not be efficient on another CPU, or future multicore CPUs. Therefore, *it is important to develop and implement parallel algorithms that are efficient and easily portable from one multicore CPU to other multicore CPUs while they maintain high efficiency, and will continue to be portable and efficient for evolving future generations of multicore CPUs.*

Another performance issue is instruction execution rate, which is affected by memory access rate and could be problem size dependent. This is specially true for algorithms which use arrays of dimension two or higher (see [12]). To overcome these issues cache-oblivious and resource-oblivious algorithms have been proposed (see [5], [6], [7], [12], and [23], and references therein). These algorithms recursively divide problem to be solved until the subproblems are small enough for holding in the cache. They are asymptotically optimal for sequential algorithms, but they require parallelization or need support from special software systems — both of which are nontrivial tasks. To the best of our knowledge no scalable self-tuning parallel algorithm exists for Smith-Waterman algorithm [21].

In this paper we propose, implement, and evaluate a parallel version of Smith-Waterman algorithm (SWA) [21] that is scalable as well as self-tuning. The algorithm does a two-step self-tuning: first it executes a sequential algorithm to find an optimal size for the problem, and then it finds an optimal number of threads for each core for the highest speedup. Finally, it divides the problem to be solved into smaller subproblems and *execute each of them in parallel* using optimal number of threads.

The rest of the paper is organized as follows. We briefly review work related to our work in Section II. In Section III, we present sequential Smith-Waterman algorithm and a naive parallel implementation of it using OpenMP. In Section IV, we present our self-tuning scalable parallel algorithm for Smith-Waterman algorithm. Some typical results from our evaluation of the proposed algorithm are presented in Section V. Section VI discusses our findings and potential future work.

Faisal Sikder and Dilip Sarkar are with the Department of Computer Science, University of Miami. E-Mails: f.sikder@umiami.edu and sarkar@miami.edu; May 28, 2014

II. Related Work

For space limitation we include only a few references. A more comprehensive list of references can be found in our report [20].

Sequence alignment is one of the most fundamental task in bioinformatics. Because of the size of the databases, sequence alignment is very time consuming. To reduce alignment time heuristic-based pairwise sequence alignment algorithms and software tools have been developed. These tools can be grouped into two general categories: *hash-table* based algorithms and *suffix-tree* based algorithms.

Hash table based software tools include BLAST [3] and FASTA [18]. As the number of data-sets in databases grows everyday, so does the computation time for finding all alignments of a query sequence. To curb the growth of computation time, mpiBLAST [8] and RPAlign [4] have exported BLAST and FASTA to distributed computing clusters. Because of higher memory and time complexity, suffix-tree based tools are fewer than hash-table based software tools. A prominent suffix-tree based software is MUMmer [10].

These heuristic-based tools work very well for finding locations of exact or near-exact alignments of a query sequence. However, if some symbols of a query sequence differs in few locations because of sequencing error, or some symbols were added or deleted because of sequencing errors they fail to identify locations of optimal alignments as can be done with the Smith-Waterman algorithm [21].

The Smith-Waterman [21] algorithm and its improved version [13] (which is also known as Smith-Waterman algorithm) have much better sensitivity and specificity, but require quadratic computation time and space. Thus, sequential version of the algorithm is impractical for long query and/or database sequences. To extend the scope of usefulness, parallel versions of Smith-Waterman algorithm have been implemented on various parallel computing platforms, including multicore CPUs [1], [22] and SIMD instruction set [11], [19], GPUs [9], [15], CPU and GPU based hybrid systems [16], CPU and FPGA based hybrid systems [17].

One of the parallel implementation of Smith-Waterman algorithm using GPU is CUDASW++ [15]. This CUDA based parallel implementation is claimed to have demonstrated significant speedup over CPU implementation. Khajeh-Saeed et al. implemented Smith-Waterman algorithm for single and multiple GPU systems [14]. They reported 45 times speedup over CPU version. Also, for 4-GPU systems they reported a speedup of about four over single GPU system.

Rongers [19] implemented SIMD based version of Smith-Waterman algorithm and claimed six times speedup. Farrar also used SIMD based implementations and received 2-8 fold speedup over others [11].

High Performance Genomics project recently implemented a parallel version of the Smith-Waterman algorithm using OpenMP, which is know as HPG-SW [1]. The algorithm has neither been published yet nor any documentation is available. However, the source code is available from the Internet [1]. In this algorithm, the number of threads to be used during run-time is hard-coded and requires to be changed manually.

Any GPU implementation is highly platform dependent and has restricted memory access mechanism. On the other hand, even low cost CPUs from different manufacturers have 4 to 8 cores. Thus, an efficient, scalable, and self-tuning parallel implementation of Smith-Waterman algorithm for all multicore CPU-types is very important.

III. Sequential and a Naive Parallel Implemntation of Smith-Waterman Algorithm

For the ease of presentation and completeness, first a sequential version of Smith-Waterman algorithm and then an OpenMP-based parallelization of it are presented. A brief evaluation of these two versions are also presented in this section to motivate our readers.

A. Sequential Version of Smith-Waterman Algorithm

Let $DB =< d_1, d_2, ..., d_m >$ be a database sequence of length m and $PT =< p_1, p_2, ..., p_n >$ be a pattern sequence of length n. For local alignment of a PT with a DB the Smith-Waterman algorithm [13], [21] is the most widely used algorithm. This computation intensive dynamic-programming algorithm runs in two phases: in *phase 1* it creates an alignment score matrix $H_{m+1 \times n+1}$, and in *phase 2* it obtains the optimal alignment.

Phase 1*-Compute the alignment score matrix:* The elements of first row and first column of H are initialized to 0, that is, for $i = 0$ and $0 \leq j \leq m$, $H_{0,j} = 0$, and for $1 \leq i \leq n$ and $j = 0$, $H_{i,0} = 0$. Other alignment scores for elements of H are computed using the equation 1; a reward is added to, or a penalty value is subtracted from $H_{i-1.j-1}$ for obtaining $H_{i,j}$. If $p_i = d_j$, a match occurs and the reward value is w_r. If $p_i \neq q_j$, a mismatch occurs and the penalty value is w_{pm}. If a gap is inserted in PT, the penalty value is w_{pg}. If an element is deleted from DB, the penalty value is w_{pd}.

$$H_{i,j} = \max \begin{cases} 0 & \\ H_{i-1,j-1} + w_r & \text{for a match} \\ H_{i-1,j-1} - w_{pm} & \text{for a mismatch} \\ H_{i-1,j} - w_{pd} & \text{for a deletion} \\ H_{i,j-1} - w_{pd} & \text{for an insertion} \end{cases} \qquad (1)$$

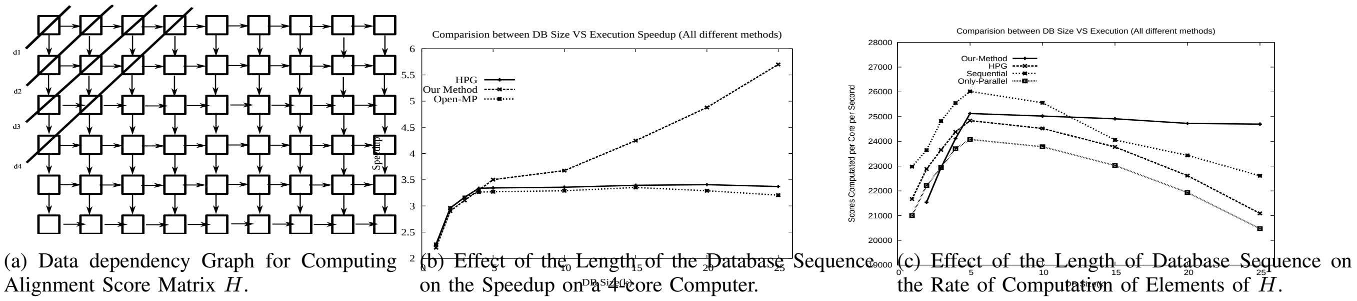
Phase 2*-Obtain the Optimal Alignment:* In this phase, starting from a highest score cell in H, an optimally aligned sequence is obtained. A move to the left cell inserts a gap in PT. An upward move represents a deletion of a symbol from DB. Finally a diagonally upward move indicates a matched or a mismatched symbol between PT and DB.

An outline for an initialization procedure is shown in **Algorithm 1**. The input to the initialization procedure are: m – the length of database sequence, n – the length of pattern sequence. The output from the procedure is initialized alignment score matrix H. A procedure for phase 1 of the Smith-Waterman algorithm is shown in **Algorithm 2**.

An OpenMP version is presented next.

B. Parallel Version of the Smith-Waterman Algorithm

After discussing data-dependency for computing H, a naive parallel version of the algorithm is presented in this section.



(a) Data dependency Graph for Computing Alignment Score Matrix H.

(b) Effect of the Length of the Database Sequence on the Speedup on a 4-core Computer.

(c) Effect of the Length of Database Sequence on the Rate of Computation of Elements of H.

Fig. 1. Data Dependency for computing alignment score matrix, Effect of the DB length on Speedup, and Effect of Number Threads for each Core.

Algorithm 1 Initialization of Sequential Version of the SWA

```
1: procedure SWInitialization($m$, $n$, $H$)
2:     // Set elements of the 0th row of $H$ to zero
3:     for ($i = 0; i < n; i$++) do $H(i, 0) = 0$;
4:     // Set elements of the 0th column of $H$ to zero
5:     for ($j = 0; j < m; j$++) do $H(0, j) = 0$;
6: end procedure
```

Algorithm 2 Sequential Version of the SWA

```
1: procedure SWSequential($m, n, H$)
2:     SWInitialization($m, n, H$)
3:     // Using Equation 1 compute elements of $H$
4:     for ($i = 1; i < n; i$++) do
5:         for ($j = 1; j < m; j$++) do Compute $H(i, j)$;
6: end procedure
```

Data Dependency for Computing H: Data dependency graph for computing alignment score matrix H is shown in Fig. 1(a). We can see from the data-dependency graph that alignment score matrix H can be computed in three different orders: (i) Row-wise from the top-most row to the bottom-most row in that order, and in each row elements are computed from the left-most column to the right-most column. (ii) Column-wise from the left-most column to the right-most column, and in each column elements are computed from the top-most row to the bottom-most row. (iii) Diagonally, from the top-left corner to the bottom-right corner as indicated by diagonal lines.

While row-wise and column-wise computation impose their restrict sequential orders, diagonal-wise computation eliminates this strict sequential order. Computation of all the elements in a diagonal can be done in any order or in parallel. However, computation of the elements of a diagonal requires that computation of all elements in the diagonal just above it have been completed.

Parallelization of Computation of H: Without loss of generality, let us assume that a multicore CPU has C cores, and for a nonzero integer k, $T(= k.C)$ rows of H can be computed in parallel by maintaining a diagonal order on these T rows. Let us assume that computation of r rows of H has been completed, and rows (r+1) to $(r + T)$ to be computed.

An outline of our naive OpenMP version is shown in **Algorithm 3**. This procedure sets the number of threads to

Algorithm 3 A Simple Parallelization of SWA Using OpenMP

```
1: procedure SWOpenMP($m$, $n$, $H$, $T$)
2:     SWInitialization($m, n, H$);
3:     omp_set_num_threads($T$);
4:     $TPos[T] = 0$;
5:     // $TPos[t]$ stores the col # thread $t$ has computed last
6:     $omp\_set\_num\_threads(T)$
7:     // Create a parallel section of $T$ threads
8:     #pragma omp parallel
9:         for ($k = 0; k < \lceil n/T \rceil; k$++) do
10:            // The thread $t, 0 \leq t < T$, computes
11:            //    rows $1, t + 1, 2t + 1, \cdots$ of $H$
12:            $i = k * T + t + 1$;
13:            for ($j = 1; j < m, j$++) do
14:                while ($TPos[t - 1] < j$) $wait()$;
15:                $Compute\ H(i, j)$;
16:                $TPos[t] = j$;
17:    end parallel section
18: end procedure
```

T, an integer multiple of C. Then it creates a `parallel section`, where all T threads compute the values of H in parallel. Since there are n rows, each thread computes approximately $\lceil n/T \rceil$ rows. If identification number of a thread is t, it computes rows $(t + 1)$, $2 * T + t + 1$ and so on. Because of an strict order for computation of elements of H, the element $H(i - 1, j)$ must be computed before the element $H(i, j)$ can be computed. To ensure this restriction, each thread waits in a while loop until the element just above it has been computed. An array $TPos$ is used for lock-free implementation of the restriction.

C. Perfomance Evaluation of the Naive Version

We implemented the procedure shown in **Algorithm 3** using OpenMP 3.0. To evaluate performance of our parallel version, we obtained the best known parallel implementation [1] (in the rest of the paper this implementation is refereed to as HPG-SW) and measured speedup on three computer systems (see Section V-A). We observed that for a PT sequence of fixed length as the length of the DB sequence is increased, the speedups steadily increase to a maximum (see Fig. 1(b)).

Effect of DB Length on Computation Rate: The most interesting observation is illustrated in Fig. 1(c). As the length of the DB sequence increases all implementations (except implementation proposed later) share a similar trend: the computation rate for each implementation i) initially increases, ii) then reaches a maximum, and iii) finally starts to decline. **Our first hypothesis**: *For a given length of PT sequence, there is an optimal length for the DB sequence.*

Our proposed method for implementation of self-tuning parallel algorithms is presented next.

IV. Proposed Scalable Self-Tuning Parallel Implementation of SWA for Multicore CPUs

In this section, we first describe a procedure to identify an optimal length for the DB sequence. Then we propose a procedure to determine optimal number of threads for each core. Finally, we use this procedures in our implementation of a self-tuning parallel Smith-Waterman Algorithm.

A. Procedure for Identification of an Optimal Size DB

Identification of an optimal length of the DB sequence starts with a given PT of relatively small size, say 200 symbols. The initial length of the DB sequence is same as that of the PT sequence, and the length of the DB sequence is increased by a constant $IncC$ until an optimal length is found. We used $IncC = 25$ for our experiments. For each DB sequence average time to compute one element of H is computed and compared with previously computed minimum average time. If most recently computed time is smaller than the currently known minimum, the minimum time is updated and corresponding length of the DB sequence is recoded. For determination of termination point, one can use a rule of $DblPc\%$: If most recently computed average time is $DblPc\%$ higher than the currently known minimum, the algorithm stops. We used 15% rule. An outline of the algorithm is shown in **Algorithm 4**.

Algorithm 4 Identification of Optimal Length DB Sequence

```
1:  procedure FindOptimalLengthDB($OpS$)
2:      $TBound = -100; n = 200; m = 200;$
3:      $BestTime = 200$; //A very large value
4:      while $TBound < DblPc$ do // use $DblPc\%$ rule
5:          $start = time();$
6:          SWSequential($m, n, H$);
7:          $end = time();;$
8:          $CTime = (n * m/(end - start));$
9:          $TBound = \frac{CTime - BestTime}{BestTime} \times 100;$
10:         if ( $BestTime > CTime$) then
11:             $BestTime = CTime; OpS = m;$
12:         $m = m + IncC$; // $IncC$ is increment constant
13: end procedure
```

Results obtained from running this algorithm on three shared-memory computers are illustrated in Fig. 2(a) (see Section V-A). An average computation time for one element of H was obtained by dividing total time for computing H by the number of elements in H. From the plots we observe that as the length of the DB size grows, the computation time for each computer goes through three distinct phases:(i) initially average computation time remains almost constant, (ii) then it decreases to a minimum, (iii) finally it starts to increase.

Therefore, empirical evidence support our hypothesis that there is an optimal length for the DB sequence. Thus, for achieving optimal computation rate the DB sequence must be partitioned and computation for each partition has to be performed at a time.

Next we focus on identification of optimal number of threads for each core.

B. Procedure for Idetification of Optimal Number of Threads

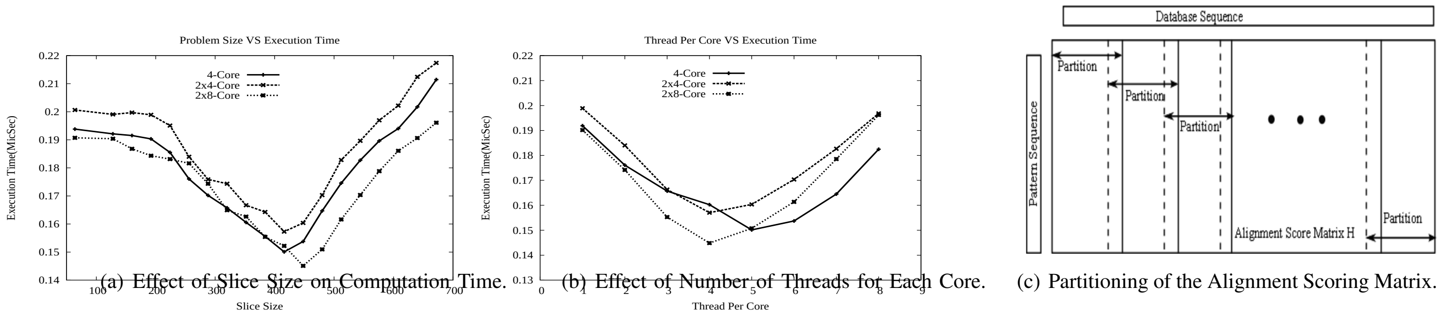
It is obvious that the number of threads should be no less than the number of cores. But can more threads per core help? If so, the next question is how many threads for each core? Also, is the thread to core ratio CPU/machine independent?

To answer these questions, we executed our OpenMP implementation of **Algorithm 3** on three different shared-shared memory computers (see Section V-A). Some results from this experiment are illustrated in Fig. 2(b). Three interesting observations can be made from the plots: (i) more than one thread per core is beneficial, (ii) there is an optimal number of threads for each core, and (iii) the optimal number varies from system to system. For the single CPU system the optimal ratio is five, while for the 2-CPU systems the optimal ratio is four.
Our second hypothesis: *For a given multicore computer, there is an optimal number of threads for each core.*

Algorithm 5 Identification of Optimal Number of Threads

```
1:  procedure OptimalThreads($OnTs$)
2:      $n = 200; m = 200$; $TBound = -100$;
3:      // $NtPc$ is # of threads for each core, $C$ is # of cores
4:      $NtPc = 1$; // initially 1 thread per core
5:      $BestTime = 200$; //a large initial value
6:      //number of cores: C
7:      while ($TBound < ThnPc$) do // use $ThnPc\%$ rule
8:          $start = time();$
9:          SWOpenMP($m$, $n$, $H$, $C * NtPc$);
10:         $end = time();$
11:         $CTime = (C * tr * n * m/(end - start));$
12:         $TBound = \frac{CTime - BestTime}{BestTime} \times 100;$
13:         if  ($CTime < BestTime$) do
14:             $BestTime = CTime$; $OnTs = C * NtPc$;
15:         $NtPc$++
16: end procedure
```

Identification of Optimal Number of Threads: The proposed steps for determining optimal number of threads for each core are similar to that for determining optimal length for DB sequence. We start with one thread per core, and increase it by one at every iteration, until we find the number that minimize computation time using our rule of $ThnPc\%$. The procedure is shown in **Algorithm 5**. Inside the while loop of this algorithm we call **Algorithm 3** with PT and DB whose lengths have been determined by calling procedure in **Algorithm 4**, or something similar. Unless the lengths are too small, it does not matter, since we are determining optimal number of threads.



(a) Effect of Slice Size on Computation Time. (b) Effect of Number of Threads for Each Core. (c) Partitioning of the Alignment Scoring Matrix.

Fig. 2. Evidence of Optimal DB Length, Optimal Number of Threads for each Core, and Optimal Partitioning of the Alignment Scoring Matrix.

C. *Procedures for Initialization of Submatrices*

Let OpS be the length of a DB sequence for which computation rate is highest. The proposed algorithm first executes procedure presented in **Algorithm 4** to determine OpS. Then computation of alignment score matrix H is divided into computation of $\lceil m/OpS \rceil$ smaller alignment score submatrices obtaied by dividindg H vertically, as shown in Fig. 2(c). Notice that the first row of each partition, except for first partition, is the last row of the partition just before it. The computation of each submatrix is done in parallel. *For keeping all elements of a partition in contiguous locations in the memory, we use a 3-D array. The first dimension of the array represents the partition number of H.*

Initailaiztion of Partitions of H: The initialization procedure assigns zeros to all elements of the 0th row all partitions. It also assigns zeros to all elements of the 0th column of the first partition. An outline of the procedure is shown in **Algorithm 6**. The elements of 0th column of other partitions are initialized by copying last column of the previous partition at the beginning of computation of the partition.

Algorithm 6 Initialization of the Partitions of H

```
1: procedure INTIPARTITIONEDH($m$, $n$, $H$, $OpS$)
2:     $partitoins = \lceil m/OpS \rceil$
3:     for ($k = 0; k < partitions; k$++) do
4:         // Set elements of the 0th row of $H$ to zero
5:         for ($j = 0; j < OpS; j$++) do $H(k, 0, j) = 0$;
6:     // Set elements of the 0th column of $H$ to zero
7:     for ($i = 0; i < n; i$++) do $H(0, i, 0) = 0$;
8: end procedure
```

Next we present a procedure that is executed by each thread for computing its share of the alignment score matrix H.

D. *Procedure for Each Thread*

The procedure shown in **Algorithm 7** is executed by all threads created in **Algorithm 8**. Each thread first computes $numRows$, the number of rows the thread has to compute. Next it enters into a loop to compute these rows of H. It repeats the process for all the partitions. Recall that there is an strict order for computation of elements of H. The element $H(k, i-1, j)$ must be computed before the element $H(k, i, j)$ can be computed. To ensure this restriction we use an array $TPos$ as shown in the procedure.

Algorithm 7 Task for Each Thread

```
1: procedure THREADTASK($n$,$H$,$T$,$partitions$,$TPos$)
2:     for ($k = 0; k < partitions; k$++) do
3:         $numRows = \lceil n/T \rceil$; // rows per thread
4:         // $s$ keeps the count of rows computed by the thread
5:         for ($s = 0; s < numRows; s$++) do
6:             // compute next row $i$ for thread $t$
7:             $i = s * T + t + 1$;
8:             //set the starting position of the DB slice
9:             for ($j = 1; j < OpS; j$++) do
10:                while ($TPos[t-1] < j$) do $wait()$;
11:                $Compute\ H(k, i, j)$; $TPos[t] = j$;
12: end procedure
```

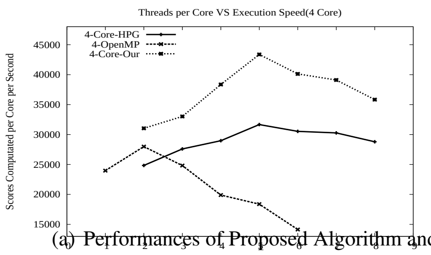
Algorithm 8 Self-Tuning Parallel SWA for Multicore CPUs

```
1: procedure SELFTUNINGOPTIMALTHREADS($m$, $n$, $H$;)
2:     FINDOPTIMALDB($OpS$)
3:     INTIPARTITIONEDH($m$, $n$, $H$, $OpS$);
4:     $partitions = \lceil m/OpS \rceil$;
5:     OPTIMALTHREADS($OnTs$);
6:     //$TPos$ Shared array to hold current position
7:     $TPos[OnTs] = 0$
8:     //$t$ set number of threads for OpenMP derivatives
9:     $omp\_set\_num\_threads(OnTs)$
10:    #pragma omp parallel
11:        THREADTASK($n$,$H$, $OnTs$, $partitions$, $TPos$)
12:    end parallel section
13: end procedure
```

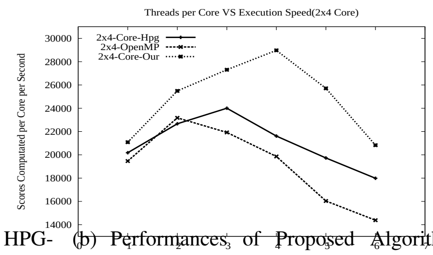
E. *Scalable Self-Tuning Implementation of SWA for Multicores*

Now we have all necessary procedures for a modular description of the *scalable self-tuning* parallel algorithm for computing alignment score matrix H. An outline of the algorithm that uses four procedures described earlier is shown in **Algorithm 8**.

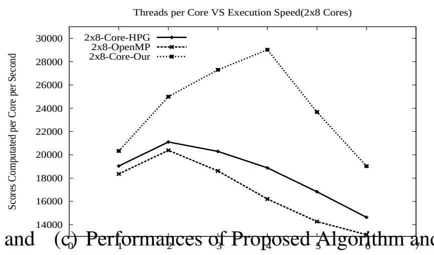
The algorithm first calls the procedure shown in **Algorithm 4** for computing an optimal length DB sequence. Next it calls the procedure shown in **Algorithm 6** for creating and initializing a 3-D array for alignment score matrix H. Then it calls the procedure shown in **Algorithm 5** for computing optimal number of threads, $OnTs$, and it is used for creating $OnTs$ threads for concurrent execution. Finally, from the



(a) Performances of Proposed Algorithm and HPG-SW on a 4-Core Single CPU Computer.



(b) Performances of Proposed Algorithm and HPG-SW on a 4-Core 2-CPU Computer.



(c) Performances of Proposed Algorithm and HPG-SW on 8-Core 2-CPU Computer.

Fig. 3. Comparison of Performances of Proposed Algorithm with Other Algorithms on Three Different Multicore Shared-Memory Computer Systems.

parallel section of the algorithm each thread invokes procedure shown in **Algorithm 7**.

In the next section we present some typical results obtained during our extensive evaluation of the algorithm [20].

V. Empirical Evaluation

Before presenting the results we briefly describe three shared memory computers that have been used for our evaluation.

A. Experimental Systems

We used both single CPU and double CPU systems. One computer has a single CPU and other two systems have double CPUs. Brief description of these machines, including operating systems and compilers used, are described next.

a) Single CPU System: This computer has a single AMD 4-Core 3.6GHz CPU with 8MB shared cache and 16GB RAM shared by all cores and is running 64-bit Ubuntu version 12.04. We used GCC compiler vesion 4.6.5 with OpenMP 3.0.

b) Double CPU Systems I: This computer system has two 4-Core AMD 2.2GHz CPUs. Each core has dedicated 512 KB cache. Two CPUs share 16GB RAM and is running 64-bit Red Hat Linux version 4.1.2. We used Intel C++ compiler version 11.1 with OpenMP 3.0.

c) Double CPU Syatem II: It has two 8-Core Intel Xeon 2.6GHz CPU with 20 MB of shared cache and 128GB RAM shared by both CPUs and is running 64 bit Red Hat Linux version 4.4.5. We used Intel C++ compiler version 11.1 with OpenMP 3.0.

B. Dataset for Evaluation

For evaluation of all algorithms we used genome sequence of *Pseudomonas aeruginosa* bacteria that was downloaded from NCBI genome database [2]. For testing purpose we used only a part of the sequence. The length of database sequence was varied from 1K to 25K and that of the query pattern was varied from 512 to 5K.

C. Performance Evaluation

We will present three sets of results in this section. The first set shows how computation rate is affected as the length of the database sequence is increased. We also show how speedup varies with the length of the database sequence. The second set demonstrates effect of length of partition of the database sequence on the per core computation rate. And the final three Figures show variation of computation rate as the number of threads per core is varied.

1) Effect of Length of the Database Sequence: The plots in Fig. 1(c) show that as the length of database sequence increases, rate of computation initially increases to a maximum and then decreases for all algorithms, except implementation proposed here. For ease of comparison computation rates have been normalized to per core. The proposed optimal algorithm reaches to a maximum and then retains the rate as the length of the database sequence increases.

The plots in Fig. 1(b) show speedup for our simple OpenMP parallel version, the HPG-SW implementation, and our optimal implementation. The simple OpenMP implementation has slightly lower speedup than the HPG-SW implementation. *Our optimal implementation not only has higher speedup, but it also has super-linear speedup when the length of the database sequence is greater than 10K.* The super-linear speedup is attributed to the fact that its computation rate does not decrease as the sequential version does (see Fig. 1(c)). Thus, mere speedup measurement may not reveal performance of parallel algorithm. We need to look more closely for increase and/or decrease of computation rate as the problem size increases.

2) Effect of the Length of Partition of H: We already know from our discussion in Section III-C that there is an optimal partition size for the H Matrix. But the plots in Fig. 2(b) reveal that the optimal size is not unique — computer system dependent. For both the 4-core single CPU computer and the 8-core double CPU computer the best performance is obtained when the partition length of H is about 375. On the other hand, for the 4-core double CPU computer the best performance is obtained when the length of the partition of H is about 350.

3) Effect of Number of Threads per Core: Recall that we have seen in Fig. 2(c) that there is an optimal number of threads for highest computation rate. We discovered several other interesting facts from our evaluation of the effect of the number of threads per core, some of which are presented here. We will divide our discussion into three paragraphs: 4-core single CPU computer, 4-core double CPU system, and 8-core double CPU system.

a) Single 4-core CPU Computer: Figure 3(a) shows that our optimal implementation attains highest computation rate when the number of threads per core is five, but for simple OpenMP implementation the highest computation rate is obtained when the number of threads per core is two. For the HPG-SW implementation the highest performance is obtained when number of threads per core is five, same as our optimal implementation.

b) Double 4-Core CPU Computer: Figure 3(b) shows that our optimal implementation attains highest computation rate when the number of threads per core is four, but for simple OpenMP implementation the highest computation rate is obtained when the number of threads per core is two. For the HPG-SW implementation the highest performance is obtained when number of threads per core is three.

c) Double 8-Core CPU Computer: Figure 3(c) shows that our optimal implementation attains highest computation rate when number of cores is four, but for simple OpenMP implementation and the HPG-SW implementation the highest performance is obtained when number of threads per core is two.

A general observation is that our optimal implementation significantly outperforms both simple OpenMP and HPG-SW implementations.

VI. Conclusions and Future Work

The parallel algorithm implementation technique presented in this paper for parallelization of Smith-Waterman algorithm for alignment of a query or pattern sequence, PT, with a database sequence, DB, is a new approach for implementing scalable self-tuning parallel algorithms for shared memory systems. The parallel algorithm we developed using the proposed approach scales well as the problem size grows. Moreover, the parallel algorithm implemented using the proposed technique deploys optimal number of threads for highest possible speedup. To the best of our knowledge, no other attempt has been made to optimize number of threads for achieving highest speedup.

A naive implementation using OpenMP failed to scale-up as the DB size increased. Similarly, the best known parallel implementation for multicore system, HPG-SW, from High Performance Genomics project [1] failed to scale-up. As discussed in Section V, our implementation maintained highest computation rate for all DB sizes, except when the DB size is too small. As a matter of fact, if DB size is too small, parallel algorithm should be avoided altogether.

Our implementation can be improved by parallelizing initialization part of the program. Also, the proposed technique can be used for implementation of other computation intensive problems, including most dynamic programming algorithms and matrix operations. *Efficient implementation of any algorithm has an added benefit of reduced energy consumption. Reduction of computation time for computation-intensive problems magnify the benefit of energy consumption.*

It would be interesting to develop analytical model for computing optimal number of threads for each core.

References

[1] HPG-SW. http://docs.bioinfo.cipf.es/projects/hpg-sw/files, downloaded in April 2013.

[2] NCBI resources. http://www.ncbi.nlm.nih.gov/, 2013.

[3] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[4] Sanghamitra Bandyopadhyay and Ramakrishna Mitra. A parallel pairwise local sequence alignment algorithm. *NanoBioscience, IEEE Transactions on*, 8(2):139–146, 2009.

[5] R.A. Chowdhury, Hai-Son Le, and V. Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 7(3):495–510, 2010.

[6] Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 207–216. ACM, 2008.

[7] Richard Cole and V. Ramachandran. Efficient resource oblivious algorithms for multicores with false sharing. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 201–214, 2012.

[8] Aaron Darling, Lucas Carey, and Wu-chun Feng. The design, implementation, and evaluation of mpiBLAST. *Proceedings of ClusterWorld*, 2003, 2003.

[9] Edans Flavius de O.Sandes and Alba Cristina M.A. de Melo. Retrieving Smith-Waterman alignments with optimizations for megabase biological sequences using GPU. *Parallel and Distributed Systems, IEEE Transactions on*, 24(5):1009–1021, 2013.

[10] Arthur L Delcher, Simon Kasif, Robert D Fleischmann, Jeremy Peterson, Owen White, and Steven L Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[11] Michael Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, 2007.

[12] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297, 1999.

[13] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, 1982.

[14] Ali Khajeh-Saeed, Stephen Poole, and J Blair Perot. Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors. *Journal of Computational Physics*, 229(11):4247–4258, 2010.

[15] Y. Liu and B. Schmidt. CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing. *Design Test, IEEE*, PP(99):1–1, 2013.

[16] Fernando Machado Mendonca and Alba Cristina Magalhaes Alves de Melo. Biological sequence comparison on hybrid platforms with dynamic workload adjustment. In *International Symposium on Parallel & Distributed Processing Workshops and PhD Forum*, 2013.

[17] N. Neves, N. Sebastiao, A. Patricio, D. Matos, P. Tomas, P. Flores, and N. Roma. BioBlaze: Multi-core SIMD ASIP for DNA sequence alignment. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 241–244, 2013.

[18] William R Pearson and David J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

[19] Torbjrn Rognes and Erling Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, 2000.

[20] Faisal Sikder and Dilip Sarkar. CPU oblivious and scalable parallel algorithm for Smith-Waterman algorithm for multicore and shared-memory compters. Technical report, University of Miami, 2013.

[21] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[22] Guangming Tan, Shengzhong Feng, and Ninghui Sun. Locality and parallelism optimization for dynamic programming algorithm in bioinformatics. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[23] Yuan Tang, Rezaul Chowdhury, Chi-Keung Luk, and Charles E Leiserson. Coding stencil computations using the pochoir stencil-specification language. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar'11)*, 2011.