PVM Algorithms for Some Problems in Bioinformatics

Hongmei He, Xuan Liu, Matthew Newton, Ondrej Sýkora

Department of Computer Science,Loughborough University,Loughborough, Leicestershire LE11 3TU, The United Kingdom
{H.He, X.Liu, M.C.Newton, O.Sykora@lboro.ac.uk}

Abstract. We design and analyze implementation aspects of a PVM version of the well known Smith-Waterman algorithm, and then we consider other problems important for bioinformatics, such as finding longest common substring, finding repeated substrings and finding palindromes.

Key words : Smith-Waterman algorithm, PVM algorithms ,the longest common string,the longest repeated string,the longest Palindrome

1 Introduction

String database searching is one of the most important and challenging tasks in bioinformatics. It is necessary to find the best match between two given DNA or protein strings. In the match, we have a penalty for opening gaps or extending gaps for each of the strings. The best match is the one with the minimum sum of such penalties. Pairwise comparison provides computer tools to directly compare two strings. They are the starting points for all kinds of string analysis. These tools can be very useful in string analysis, cloning projects, PCR analysis, and many more.

The developers of hardware and software database searching and handling are facing a great challenge from the genetic-string information that rises quickly to large amounts. Although the computing resources have increased exponentially for decades, the genetic string information maybe has extended beyond the growth speed of the computing power. If the above facts keep on going, it will be necessary to use more expensive supercomputers to search existing databases.

Cluster computing is a relatively new field of research in parallel computing. A cluster computer typically exists as a set of PCs or workstations interconnected by a switch or a fast ethernet network. In a certain sense a cluster is just a parallel computer with a, possibly, slower interconnection network. Clusters offer incredible computing power at a fraction of the cost of parallel supercomputers. In comparison, their communication power is modest, and no dedicated software is provided. For communication one mostly relies on the concepts of PVM [12] or the MPI library of communication routines [10], which provides a reasonably efficient set of primitives.

2 Smith-Waterman Algorithm

When looking for strings in a database similar to a given query string, the search programs compute an alignment score for every string in the database. This score represents the degree of similarity between the query and database string. The score is calculated from the alignment of the two strings, and is based on a substitution score matrix and a gap-penalty function. A dynamic programming algorithm computing the optimal local-alignment score was first described by Waterman and Smith [17]. Later Gotoh [4] reduced the complexity of the algorithm. Both versions have been implemented many times.

Database searches using the algorithm are unfortunately quite slow on ordinary computers, so many heuristics have been developed, such as FASTA and BLAST. These methods have greatly reduced the running time , however, at the expense of sensitivity. As a result, a distantly related string may not be found in a search by using these heuristic algorithms.
To get faster, but optimal solutions, one should use parallel computing. For example the authors of [18] used a 64 processor system to align 324 protein strings in 13 hours instead of single processor machine running 29 days to execute the same work.

One of the first parallel implementations of Smith-Waterman algorithm or Gotoh's version of it due to Sitting et al. [15] and by [6]. In [7] a cluster implementation of Gotoh's version [4] of the Smith-Waterman algorithm was designed and run on a cluster of workstations using the PVM paradigm. They claimed to achieve similar performance to a massively parallel computer Intel iPSC/860 hypercube.

Special parallel hardware to implement the Smith-Waterman algorithm was developed by more researchers e.g. [3, 11, 8, 13]. Systolic algorithms to implement the Smith-Waterman algorithm were also created [14].

Our PVM implementation of the Smith-Waterman Algorithm has been run on a cluster of 20 Sun ULTRAsparc 5 workstations running Debian GNU/Linux. They are connected with 100Mbit Ethernet using Cisco 2950 switches. We tested it for different sizes of strings and for different relative sizes and for different numbers of workers.

We used PVM, because it is standard and it frees the algorithm designer from load balancing, resource control, fault tolerance and other problems with parallel software and it is still quite popular as well.

2.1 PVM Smith-Waterman algorithm

{y is the pattern array, x is the text array, $y[u, v]$ is the substring of y from index u to v, and p is the number of workers in the cluster, g and t are the pattern and text lengths respectively, worker(0) means master} (see Algorithm 1)

Algorithm 1 Parallel Smith-Waterman Algorithm

MASTER :

```
1: Send (y, all); k= 0;
2: while (k < t) do
3:    Send (x[k, k + (g/p) –1],all);
4:    k = k + g/p
5: end while
```

Worker(r) :

```
1: while (k < t) do
2:    if data from worker(r-1) was received then
3:       WS(y[r(g/p),(r+1)(g/p)–1],x[k, k+(g/p)–1]}
4:       Send (Return data of WS(y[r(g/p),(r+1)(g/p)–1],x[k, k+(g/p)–1]},
5:       worker(r + 1));
6:    end if
7:    if r = p then
8:       worker(r) returns Best
9:    end if
10: end while
```

Smith-Waterman Algorithm :

```
1: WS(string1,string2)
2: {Define f[i, j] as maximal similarity score, d as the cost of deletion and sim[i, j]
   as the similarity of the i-th character of the pattern and the j−th character of
   the text}
3: Best = 0; f[0, 0] = 0;
4: for (0 < i ≤ string1_length) do
5:    for (0 < j ≤ string2_length) do
6:       {f[i, j]=max{f[i−1, j]−d, f[i, j−1]−d,f[i-1, j-1]+sim[i, j]};
7:       Best = max ( f[i, j],Best)
8:    end for
9: end for
```

2.2 Test Results

In the first experiment, we used pattern and text of six different lengths, from 0.5K to 16K. We measured the running time for each pair of a text and a pattern. We also tested different numbers of workers (see Fig. 1). In another experiment, we used the same texts and patterns and worker numbers, but ran only 8 machines. In Fig. 2 both cases are compared.

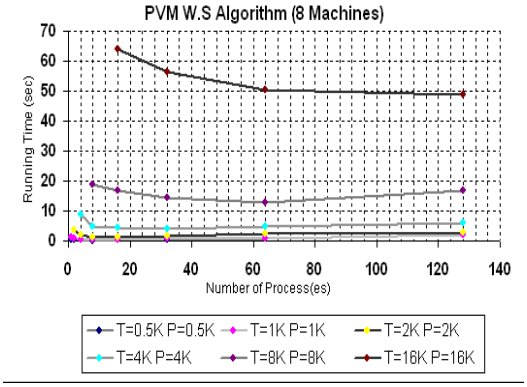

Fig. 1

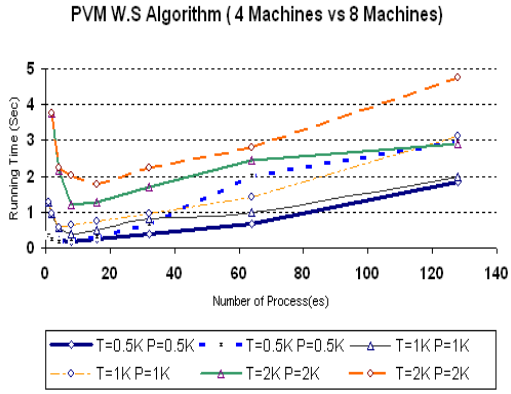

Fig. 2

2.3 Analysis

The Running time of the Gotoh's version of the Smith-Waterman algorithm is $O(pattern\_length \times text\_length)$. From our experiments, the PVM algorithm running time is: $O((text\_length)^2/p)$, where p is the number of processors. Test results show that the number of processors plays a role, if the total length of input becomes larger. The results showed that the relative length of the pattern and text does not matter too much. If the total length of input exceeds a limit, it matters if we use more virtual processors than the real number of processors.

3 PVM algorithm for finding repeated strings

An important and very usual activity in bioinformatics is detection of repeated patterns in strings.

A repeated substring is one having at least two matching occurrences at distinct positions within the string, with the possibility that such occurrences may overlap. A repeated substring may be said to be maximal if the match of a pair of its instances cannot be extended further in either direction : Given string y, with length $n > 0$, identify and locate the longest substring $|x|$ occurring at one or more distinct string overlapping positions in y.

There are many sequential methods for finding repeated substrings [16]. Our PVM algorithms are efficient, especially in the case of finding all repeated substrings.

First we discuss a general problem of finding a non-empty substring x of a string y which repeats in a non-overlapping position in the string y, i.e. the substring x of length m repeats in y, so that there are two positions $k < l$ in the string y such that $x[i] = y(k + i - 1) = y(l + i - 1)$ for $1 \leq i \leq m$ and $k + m < l$. A special subproblem of this problem is to find the longest tandem substring in a string. When looking for the longest tandem substrings we want to find concatenated two longest repeated substrings. It means the first element of the second longest substring is next to the the final element of the first longest substring.

What follows is the PVM algorithm for both non-overlapping and overlapping cases.(see Algorithm 2)

3.1 Test Results

The following Fig. 3 shows testing results for searching longest repeated substring.

3.2 Analysis

Work (we count the number of comparisons executed) of this PVM algorithm is m^2 and the running time is: $O(m^2/p)$, where p is the number of workers, m is

Algorithm 2 Parallel Longest Repeated Substring Algorithm

MASTER :

{x is the text array, p is the number of workers in the cluster, g is the text lengths.}
Initialisation
Send the x to all workers.
Receive the Longest Repeated Substring,maxRepeatedStr from all workers.
Compare each result from workers and output the Longest Repeated Substring.

Worker(r) :

define maxRepeatedStr with three elements, the length of string,$maxRepeatedStr.len$,the first start position in x, $maxRepeatedStr.x1$, and the second start position in x, $maxRepeatedStr.x2$
Initialize maxRepeatedStr with the element, len=0;
step=$g/(2 \times p) + 1$;
frontPart=$WorkerID \times step$;
rearPart=$(2(p-1) - WorkerID) \times step$;
for $(k = frontpart; k < frontPart + step; k++)$ **do**
  { substring=searchSubstr(x, g, k) }
  if(substring.len>maxRepeatedStr.len) maxRepeatedStr=substring;
end for
if $(frontPart != rearPart)$ **then**
  for (k=rear; k<rearPart+step & k<g; k++) **do**
    substring=searchSubstr(x, g, k)
    if(substring.len>maxRepeatedStr.len) maxRepeatedStr=substring;
  end for
end if
send the maxRepeatedStr to Master.

Search Longest Repeated Substring :

SearchSubstr(X,g, k)
{
define maxZeroString.
Initialise $maxZeroString$ with the element $len = 0$;
$i = 0$
while $(i < g - k)$ **do**
  $val[i] = X[i] - X[i + k]$
  i++;
end while
set variable, len to record the length of current 0 string
while $(i < g - k)$ **do**
  i=the start position of the next 0 string in array,val
  $counter = thelengthofthe0string$
  i=i+counter
  if ($k <$ $counter$) $len = k$; (overlap)
  else $len =$ $counter$; ($k \geq$ $counter$, no overlap)
  if (len>maxZeroString.len) {
  maxZeroString.len=len;
  maxZeroString.x1 = $i-$ counter;
  maxZeroString.x2 = $i + k-$ counter;
  }
end while
Return maxZeroString;
}

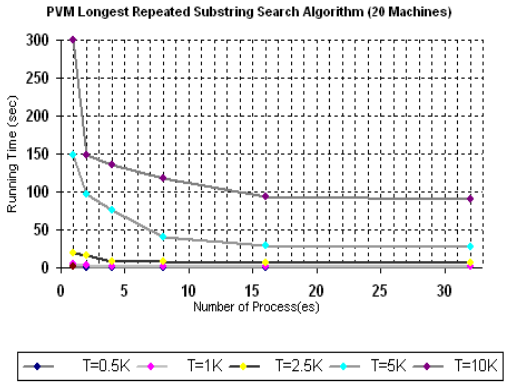

Fig. 3

the string size. The parallel algorithm decreases the running time for searching longest repeated substring, but, if we used too many workers to find the longest repeat string in small string, the running time would increase.

3.3 Discussion of the overlapping case

The Fig. 4 shows the procedure of searching longest substring.

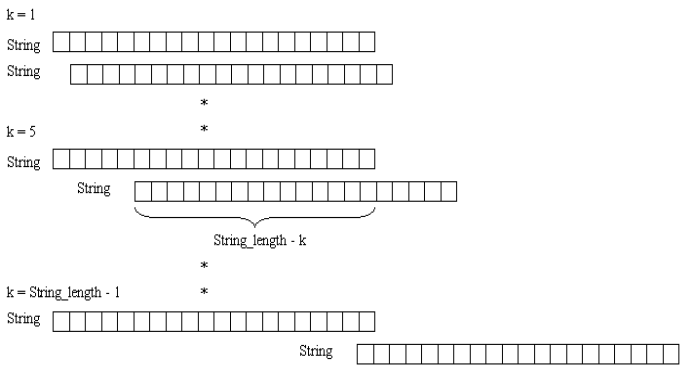

Fig. 4

We use k for the number of how many times the string is shifted. We calculate the difference of the common part of the two strings, count the size of the longest matching substring, and get the corresponding start positions of the longest repeating substrings. We use a variable, *counter*, to record the length of the matching substring, and we consider overlapping of repeated substrings too.

In the non-overlapping case, for example, using a string, "AAAAAAA", we can say the longest repeating substring is "AAA", whose first start position is 0, and second start position is 3. In another example, a string: "AGTCAGTCA", the longest repeating substring is "AGTC" and rather than "AGTCA". However, in the PVM algorithm, if there are overlapping substrings in the string, the value of *counter* could be more than the real value of length. Fig. 6 describes the procedure of finding an overlapping substring. For the example (Fig. 5a) of "AAAAAAA", we have $k = 1$, $counter = 6$, and obviously, it is not correct to record the value of *counter*. For the example (Fig. 5b) of "AGTCAGTCA", we have $k = 4, counter = 5$ and we also cannot record the current value of *counter*. Actually, k is the real length of repeating substrings instead of *counter*. So we just need to record the value of k as it is largest, when $k < counter$.

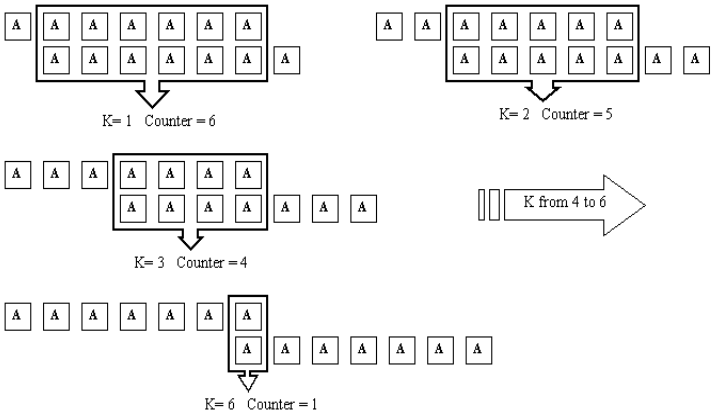

Fig. 5(a)

3.4 Load balancing - onion peeling principle

To keep the workload balanced equally over processes, we suggest using the so called onion peeling principle which ensures almost equal distribution of workload. In Fig. 6 we show an example with 4 workers and computation of 8 blocks of different size. The size of neighbouring blocks differs by 1. Computation of the 8 blocks will be distributed over these 4 workers as follows. Like peeling an onion skin, the first worker computes the outside layer, which are the first and the eighth blocks. The second worker will execute the computation of the next layer, which are the second and the seventh blocks, and so on.

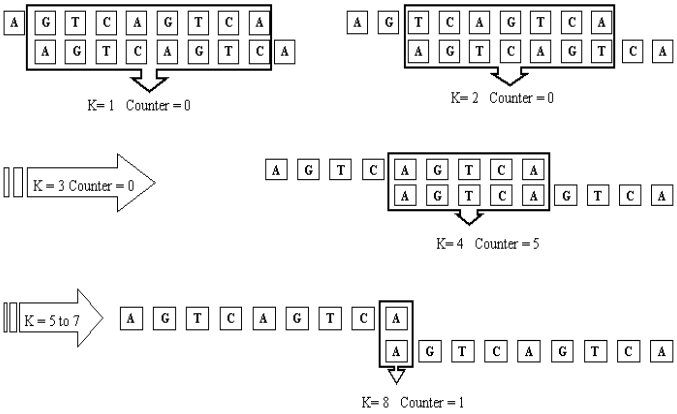
K= 1 Counter = 0
K= 2 Counter = 0
K = 3 Counter = 0
K= 4 Counter = 5
K = 5 to 7
K= 8 Counter = 1

Fig. 5(b)

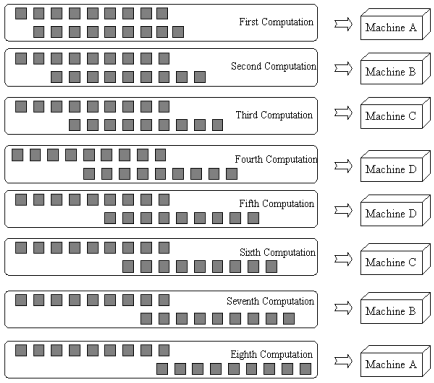
First Computation
Machine A
Second Computation
Machine B
Third Computation
Machine C
Fourth Computation
Machine D
Fifth Computation
Machine D
Sixth Computation
Machine C
Seventh Computation
Machine B
Eighth Computation
Machine A

Fig. 6

4 PVM algorithm for the longest common substring

The problem of finding the longest common substring is an important problem too. E.g. in the DNA sequencing shotguns[2] are used, one should find pairs of shotguns with the longest common prefix from one shotgun matching a suffix of the other shotgun .

4.1 Longest Common Substring Algorithm

A longest common substring of two strings is a substring common to both, having maximal length, i.e. it is at least as long as any other common substring of the strings. Given two strings x and y, with lengths $|x| = m, |y| = n$, where $0 < m <= n$, find $lcs(x, y)$, where $lcs(x, y)$ is a longest common substring of x and y. We could also be interested in f longest common substrings where f is a constant.

There are many sequential methods to solve this problem (see e.g. [16, 1]). We suggest a relatively simple parallel algorithm with a load balancing idea described in the following subsection.

In the following pseudocode the ID of a worker is denoted by "WorkerID".(see Algorithm 3)

4.2 Test result

We used 20 machines in the cluster to run the PVM algorithm. Different length text strings and pattern strings were applied to the PVM algorithm. We also tested different numbers of workers.(see Fig.7)

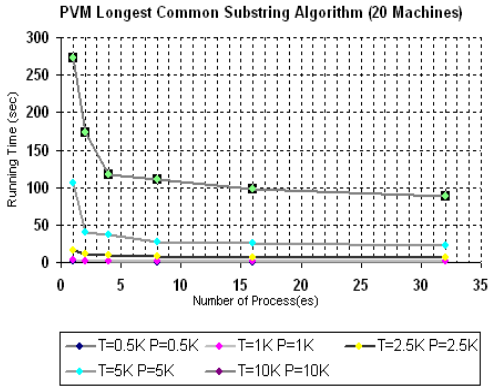

Fig. 7

Algorithm 3 Parallel Longest Common Substring Algorithm

MASTER :

{y is the pattern array, x is the text array,p is the number of workers in the cluster, g and t are the pattern and text lengths respectively}

Initialization :

Send the y to all workers.
Send the x to all workers.
Receive the Longest Common Substring from all workers.
Compare each result from workers and output the Longest Common Substring.

Worker(r) :

define maxCommonString with three elements, the length of string,$maxCommonString.len$, the start position in y, $maxCommonString.y$, and the start position in x, $maxCommonString.x$
Initialize maxCommonString with the element, len=0;
Step=$g + t/(2p) + 1$;
$FrontPart = WorkerID \times step$;
According to the distribution of machines, each worker in the cluster includes two parts, front part and rear part.
$RearPart = (p + WorkerID) \times step$;
for (frontpart < k< frontPart + step) **do**
  substring=LCS(y, x, g, t, k)
  if ($substring.len > maxCommonString.len$)
  $maxCommonString = substring$;
end for
for (k=rear; k<rearPart+Step & k<g+t; k++) **do**
  substring=LCS(y, x, g, t, k)
  if ($substring.len > maxCommonString.len$)
  $maxCommonString = substring$;
end for
send the maxCommonString to Master.

Search Longest Common Substring :

LCS(Y,X,g,t,k)
define maxZeroString.
Initialise $maxZeroString$ with the element $len = 0$;
if (g-k-1>0){Start1=g-k-1; Start2=0}
else { Start1=0; Start2=k-g+1;}
if (g-Start1)<(t-Start2)
$L = g - Start1$
else $L = t - Start2$
set m=0; maxLen=0;
set i=Start1;j=Start2;
while (m < L) **do**
  $val[m++] = Y[i++] - X[j++]$
end while
set $i = 0$; set maxlen=0;
while ($i < L$) **do**
  i=the start position of next 0 string in array, val;
  counter=the length of the 0 string
  i=i+counter;
  if ($counter > maxZeroString.len$)
  {$maxZeroString.len = counter$;
  $maxZeroString.y = Start1 + i - counter$;
  $maxZeroString.x = Start2 + i - counter$; }
end while
return maxZeroString;

4.3 Analysis

Work (we count the number of comparisons to be executed) of this PVM algorithm is $\min^2(|x|, |y|) = m^2$ and the running time is: $O(\min^2(|x|, |y|)/p) = O(m^2/p)$, where p is the number of processors.

4.4 Load balancing–Tandem Cascade

In parallel Longest Common Substring Algorithm, we use a kind of load balancing implementation method,which is similar to the onion peeling principle. As Fig. 8 shows, the computation between two strings contains two parts,which are the upper string sliding from the right end to the central of the lower string and going on to slide from the central to the left end. The longest computing time will happen when the upper string on the central position of the lower string.We arrange these computation to each worker in tandem, which means any PVM worker will be arranged some of easy computations(like 1st Computation and 2nd Computation)and some of hard computations(like 10th Computation and 11th Computation).Other PVM workers may be arranged the 3rd,4th,12th and 13th computation for keeping the load balance,and so on.

5 PVM algorithm for the Longest palindrome substring

A string x of length n such that: $x(i) = x(n - i + 1), 1 \leq i \leq n$, is called a palindrome. We design a PVM algorithm to find the longest palindrome and it can be easily and efficiently applied for finding f longest palindromes, or for finding all palindromes of at least some constant length.(see Algorithm 4)

5.1 Test results

In our experiment, we run our PVM algorithm on a 20 machine cluster. We tested different string sizes from 0.5kb to 10kb. We also used different numbers of workers.(see Fig.9)

5.2 Analysis

This PVM algorithm does m^2 comparisons and the running time is: $O(m^2/p)$, where p is the number of processors, m is the string size. From the Fig. 9, we can see that the best running time would be achieved (obviously) if we applied more workers. But, if the longest palindromes are "tiny", the quickest running time is achieved if "less" workers is applied. This is due to the fact that there is "larger" number of palindromes and reporting them to the master increases transmission time.

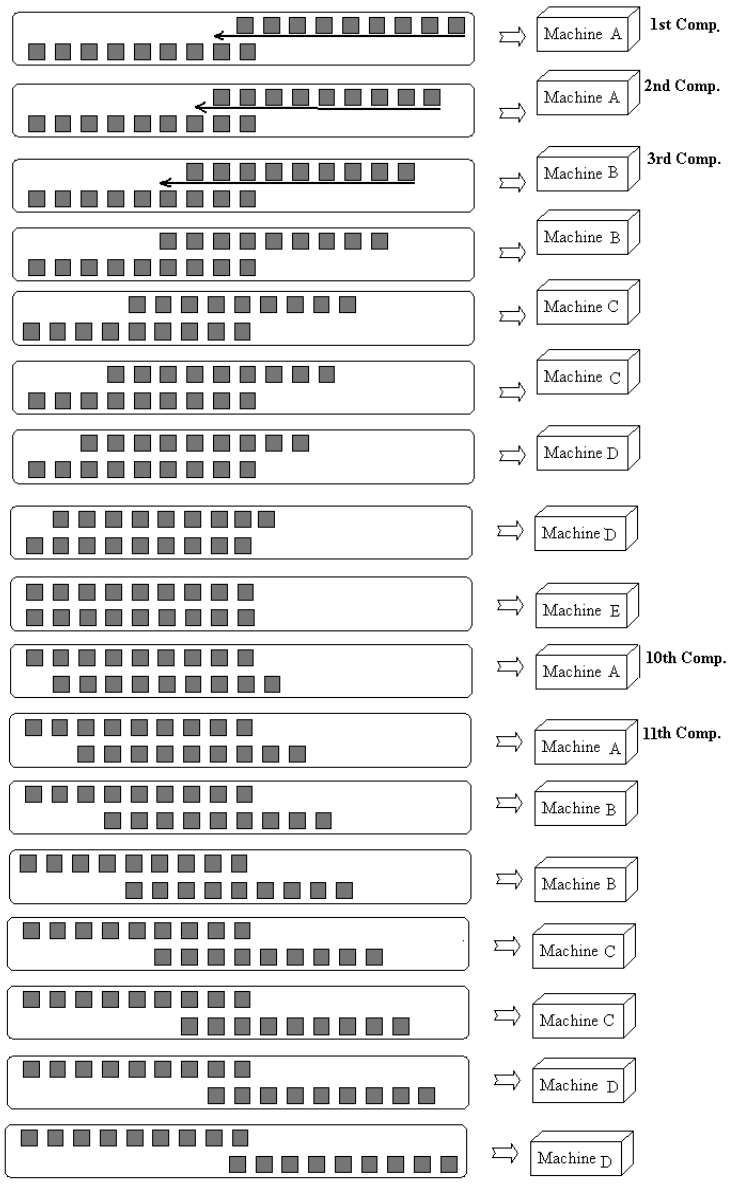
Machine A
1st Comp.
Machine A
2nd Comp.
Machine B
3rd Comp.
Machine B
Machine C
Machine C
Machine D
Machine D
Machine E
Machine A
10th Comp.
Machine A
11th Comp.
Machine B
Machine B
Machine C
Machine C
Machine D
Machine D
Number of Machine=5

Fig. 8

Algorithm 4 Parallel Longest Palindrome Substring Algorithm

MASTER :

1: {X is the text array,p is the number of workers in the cluster, g is the text lengths.}
2: ***Initialisation***
3: Send X to all workers.
4: Receive the Longest Palindrome Substring from all workers.
5: Compare each result from workers and output the Longest Palindrome Substring.

Worker(r) :

1: define maxPalindrome with two elements, the length of string,$maxPalindrome.len$,the start position in x, $maxPalindrome.x$
2: Initialize maxPalindrome with the element, len=0;
3: step=$g/p + 1$;
4: FrontPart=WorkerID $\times$ step;
5: RearPart=(p+WorkerID)$\times$step
6: **for** (frontpart<i<frontPart + step) **do**
7: substring=searchPLD(x, g, i)
8: if(substring.len>maxPalindrome.len) maxPalindrome=substring;
9: **end for**
10: **for** (i=rear; i<rearPart+step & i<2$\times g$; i++) **do**
11: substring=searchPLD(x, g, i)
12: if(substring.len>maxPalindrome.len) maxPalindrome=substring;
13: **end for**
14: send the maxPalindrome to Master.

Search Longest Palindrome :

1: SearchPLD(X,g, k)
2: {
3: define $maxZeroString$.
4: Initialise $maxZeroString$ with the element $len = 0$;
5: if (g-k-1)>0 {$start1 = g - k - 1$; $start2 = (g - 1)$}
6: else start1= 0 {$start1 = 0$;$start2 = 2 \times (g - k - 1)$}
7: if $(i > 0)$ $L = StringLength - i$
8: else $L = j + 1$
9: set $m = 0$; $i = start1$; $j = start2$;
10: **while** $(m < L)$ **do**
11: val$[m++] = X[i++] - X[j--]$
12: **end while**
13: **while** $(i < L)$ **do**
14: i=the start position of the next 0 string in array,val
15: $counter = thelengthofthe0string$
16: $i = i + counter$
17: if $(counter > maxZeroString.len)$ {
18: $maxZeroString.len = counter$;
19: $maxZeroString.x = start1 + i - counter$;
20: }
21: **end while**
22: Return $maxZeroString$;
23: }

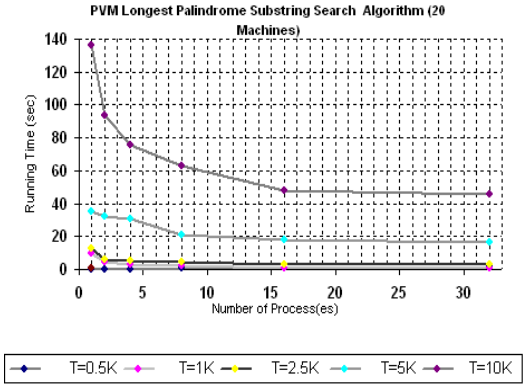

Fig.9

6 Conclusions

In this article we designed and tested PVM algorithms to compute some important problems in bio-informatics.

Among those we designed and analyzed implementation aspects of a PVM version of the well known Smith-Waterman algorithm and PVM algorithms for finding the longest common substring, finding repeated substrings and finding palindromes.

We used the so called onion peeling principle to evenly distribute the workload.

We observed some interesting facts: e.g. increase of transmission time causing slowdown if larger number of workers was used.

References

1. Crochemore, M., Rytter W., Text Algorithms, Oxford Univerity Press 1994.
2. Czabarka, E., Konjevod, G., Marathe, M.V., Percus, A.G., Torney, D.C., Algorithms for optimizing production DNA sequencing, in: *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00),* 2000, 399–408.
3. Dale, D., Grate, L., Rice, E., Hughey, R., The ucsc kestrel general purpose parallel processor, in: *Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications* (1999), 1243–1249.
4. Gotoh, O., An improved algorithm for matching biological sequences, *Journal of Molecular Biology* **162** (1982), 705–708.
5. Grice, J.A., Hughey, R., Speck, D., Reduced space sequence alignment, *Comput. Appl. Biosci.,* **13** (1997), 45–53.
6. Guan, X., Mural, R.J., Mann, R., Uberbacher, E.C., On parallel search of DNA sequence databases, in: *Proc. 5th SIAM Conference on Parallel Processing for Scientific Computing,* SIAM 1991, 332–337.

7. Guan, X., Mural, R.J., Uberbacher, E.C., Sequence comparison on a cluster of workstation using the PVM system, in: *Proc. 9th Intl. Parallel Processing Symposium,* IEEE Computer Society 1995, 190–195.
8. Hughey, R., Parallel hardware for sequence comparison and alignment, *Comput. Appl. Biosci.* **12** (1996), 473-479.
9. Martins, W.S., del Cuvillo, J.B., Useche, F.J., Theobald, K.B., Gao, G.R., A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison, in: *Pacific Symposium on Biocomputing,* 2001.
10. http://www.mcs.anl.gov/mpi/
11. http://www.paracel.com/gm/brochure.pdf
12. http://www.epm.ornl.gov/pvm/
13. Rogens, T., Seeberg, E., Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics* **16** (2000), 699–706.
14. Schmidt, B., Schroder, H., Schimmler, M., Massively parallel solutions for molecular sequence analysis, in: *Proceedings of IPDPS'2002.*
15. Sitting, D., Foulser, D.F., Carriero, N., McCorkle, G., Miller, P.L., A parallel computing approach to genetic sequence comparison: the master-worker paradigm with interworker communication, *Computers and Biomedical Research* **24** (1991), 152–169.
16. Stephen, G.A., String Searching Algorithms, Lectures Notes Series on Computing **3** World Scientific, 1994.
17. Waterman, M.S., Smith, T.F., RNA secondary structure: A complete mathematical analysis, *Mathematical Bioscience* **42** (1978), 257–266.
18. Yap, T.K., Frieder, O., Martino, R.L., Parallel computation in biological sequence analysis, *IEEE Transactions on Parallel and Distributed Systems* **9** (1998), 283–294.