A Parallel Approximation Hitting Set Algorithm for Gene Expression Analysis

D. P. Ruchkys* and S. W. Song†
Universidade de São Paulo
Instituto de Matemática e Estatística
Departamento de Ciência da Computação
05508-090 São Paulo, SP, Brazil
dpr at ime. usp. br, song at ime. usp. br

Abstract

With the recent DNA-microarray technology, it is possible to measure the expression levels of thousands of genes simultaneously in the same experiment. A genetic network is a model that describes how the expression level of each gene is affected by the expression levels of other genes in the network. Given the results of an experiment with n genes and m measures over time ($m << n$), we consider the problem of finding a subset of genes (k genes, where $k << n$) that explain the expression level of a given target gene under study. We consider the coarse-grained multicomputer (CGM) model, with p processors. In this paper we first present a sequential approximation algorithm of $O(m^4n)$ time and $O(m^2n)$ space. The main result is a new parallel approximation algorithm that determines the k genes in $O(m^4n/p)$ local computing time plus $O(k)$ communication rounds, and with space requirement of $O(m^2n/p)$. The p factor in the parallel time and space complexities indicates a good parallelization. We also show preliminary promising experimental results on a Beowulf machine. To our knowledge there are no CGM algorithms for the problem considered in this paper.

1 Introduction

With the recent DNA-microarray technology, it is possible to measure the expression levels, in terms of mRNA abundance, of thousands of genes simultaneously in the same experiment. In such experiments, one way is to monitor changes in gene expression over time, with thousands of gene expression measures taken over a sequence of time intervals.

To infer a genetic network from gene expression measurements, we are urgently in need of computational methods to handle this large amount of data to extract meaningful information. A genetic network is a model that describes how the expression level of each gene is affected by the expression levels of other genes in the network. We consider the Boolean genetic network where gene expressions levels are either one or zero. A tutorial on gene expression data analysis and modeling can be found in [5].

In this paper we explore the use of parallel computers for this purpose and present a new parallel approximation algorithm for the hitting set problem that can be used to infer which genes affect the expression level of a given target gene.

Given the results of an experiment with n genes and m measures over time, we consider the problem of finding a subset of genes (say k genes) that explain the expression level of a given target gene under study. We consider a parallel computer with p processors. We first present a sequential approximation algorithm of $O(m^4n)$ time and $O(m^2n)$ space. The main result is a new parallel approximation algorithm that determines the k genes in $O(m^4n/p)$ local computing time plus $O(k)$ communication rounds, and that requires $O(m^2n/p)$ space. To our knowledge there are no CGM algorithms for the problem considered in this paper.

There are several previous works that infer genetic networks through gene expression analysis. Liang, Fuhrman and Somogyi [11] use a reverse engineering method based on information theory. Ideker, Thorsson and Karp [9] propose a very elegant algorithm called Predictor that is based on the hitting set problem [7]. Some other related material can be found in [1, 2]. Our paper is based on [9] and a greedy approximation algorithm to solve the hitting set problem [8].

*Partially supported by CNPq.
†Partially supported by FAPESP Proc. No. 99/07390-0, CNPq Proc. No. 52.3778/96-1, 46.1230/00-3, and 521097/01-0 and CNPq/NSF Collaborative Research Program Proc. No. 68.0037/99-3.

2 The Coarse-Grained Multicomputer CGM model

A simple computing model, called *Bulk Synchronous Parallel Model (BSP)* has been presented by Valiant [12]. It gives reasonable predictions on the performance of the algorithms implemented on existing, mainly distributed memory, parallel machines. It is also one of the earliest models to consider communication costs and to abstract the characteristics of parallel machines with a few parameters. A BSP algorithm consists of a sequence of supersteps separated by *synchronization barriers*. In a superstep, each processor executes a set of independent operations using local data available in each processor at the start of the superstep, as well as communication consisting of send and receive of messages. An *h-relation* in a superstep corresponds to sending or receiving at most *h* messages in each processor.

A similar model is the *Coarse Grained Multicomputers - CGM*, proposed by Dehne *et al* [3, 4]. In this model, p processors are connected through any interconnection network. The term *coarse granularity* comes from the fact that the problem size in each processor n/p is considerably larger than the number of processors p. A CGM algorithm consists of a sequence of rounds, alternating well defined local computing and global communication. Normally, during a computing round we use the best sequential algorithm for the processing of data available locally. A CGM algorithm is a special case of a BSP algorithm where all the communication operations of one superstep are done in the h-relation. The CGM algorithms implemented on currently available multiprocessors present speedups similar to the speedups predicted in theory [3]. The CGM algorithm design goal is to minimize the number of supersteps and the amount of local computation.

We are interested in obtaining parallel algorithms that can be implemented on available parallel machines and obtain compatible execution times as predicted in the CGM model, independent of the particular type of interconnection network used.

3 The Sequential Algorithm based on the Hitting Set Problem

This section is based on [9].

Assume an experiment where n genes of a set $E = \{a_0, a_1, \ldots, a_{n-1}\}$ have their expression levels monitored over time with m measurements. Typically we have $n >> m$. We have thus a total of nm values, which are considered to be either 1 or 0. Value 1 denotes expression while value 0 denotes non-expression.

The input to the algorithm is an $m \times n$ matrix M and let a_r be the target gene under study.

As an example, consider the following input matrix M, obtained by a series of experimental perturbations, where any number of genes can be forced to a low or a high level. A forced low value is indicated by "-" and a forced high value is indicated by "+".

$$M = \left|\begin{array}{cccc} a_0 & a_1 & a_2 & a_3 \\ 1 & 1 & 1 & 0 \\ - & 1 & 0 & 1 \\ 1 & - & 0 & 0 \\ 1 & 1 & - & 1 \\ 1 & 1 & 1 & + \end{array}\right| \begin{array}{l} \\ \text{row } 0 \\ \text{row } 1 \\ \text{row } 2 \\ \text{row } 3 \\ \text{row } 4 \end{array}$$

Figure 1. Example matrix M.

Consider all pairs of rows (i, j) of M where the expression levels of the target gene a_r differ. For each such pair (i, j), find the set S_{ij} of all other genes whose expression levels also differ at these two rows. One of these genes is at least responsible for the different expression levels of a_r.

In the example, let a_3 be the target gene we are interested in. Its values differ at rows (0,1), (0,3), (1,2), and (2,3). Row 4 is excluded because gene a_3 is itself perturbed. The pair (0,1) gives the set $S_{01} = \{a_0, a_2\}$, because at rows (0,1) the expression levels of a_0 and a_2 differ. Similarly we get $S_{03} = \{a_2\}$, $S_{12} = \{a_0, a_1\}$, and $S_{23} = \{a_1\}$.

Find the smallest set HS to explain the differences over all row pairs (i, j), so that at least one gene in HS is present in each set S_{ij}. Such a set is called a hitting set, as we define below.

Hitting Set Problem

Given a finite set E and a finite collection $\mathcal{S} = \{S_1, S_2, \ldots, S_w\}$ of subsets of E, find a subset called *hitting set* $HS \subseteq E$ of minimum cardinality such that $HS \cap S_i \neq \emptyset$ for all $1 \leq i \leq w$.

In our example, we have $\mathcal{S} = \{S_{01}, S_{03}, S_{12}, S_{23}\}$. This gives the hitting set $HS = \{a_1, a_2\}$. That is, the expression level of gene a_3 can be explained by those of genes a_1 and a_2.

4 Approximation Algorithms for the Hitting Set Problem

The hitting set problem is NP-hard [7] and there are two known approximation algorithms for its resolution.

One approximation algorithm, due to Bar-Yehuda and Even, is based on the primal-dual approximation method applied to the hitting problem [6, 8]. It is an α-approximation algorithm where α = $\max_i |S_i|$. α is called the approximation ratio and it means that the obtained approximate answer is at most α times worse than the correct answer. In our case, in terms of the input size, α = $\max_i |S_i| = O(n)$.

The other approximation algorithm is based on a greedy approach [8, 10]. It is a β-approximation algorithm where β = $\ln |\mathcal{S}| + 1$. Again, in terms of the input size, $|\mathcal{S}|$ can be $O(m^2)$. Thus $\beta = O(\log m)$.

We have chosen the second approach to base our parallel algorithm because $n >> m$ and we expect to obtain a better approximation ratio.

5 A Sequential Approximation Algorithm

Let the input be an $n \times m$ expression level matrix M. To facilitate the explanation, we consider the column of the expression levels of the target gene a_r also as a separate input, given as a $m \times 1$ vector v. In other words, the expression levels of the target gene a_r are given in the corresponding column of the matrix M as well as in the vector v.

We must first find the row pairs (i, j) of v where the expression levels of a_r (elements of v) differ. Each such row pair will give rise to a set S_{ij}. That is, for each row pair (i, j) encountered, the algorithm finds the other genes of M that have different expression levels in rows i and j. These genes are the elements of the set S_{ij}.

5.1 Data Structures

We use two data structures as shown in Figure 2 corresponding to the example of Figure 1.

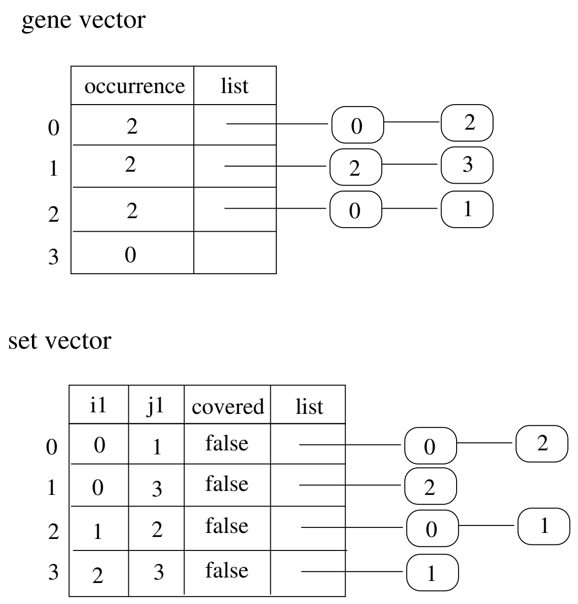

Figure 2. Data Structures Used.

One data structure is the *gene vector* where each entry corresponds to one of the n genes and a pointer to the list of sets S_{ij} it belongs to. The entries are numbered from 0 to $n-1$, corresponding to genes $a_0, \ldots, a_{n-1}$. In the greedy approximation algorithm, we repeatedly choose the element that covers the most uncovered sets, that is, we repeatedly choose the one that occurs most times in sets S_{ij}. Thus in the gene vector we have the *occurrence* field to keep track of the number of times each gene appears in sets S_{ij}.

The other data structure is a *set vector* where each entry corresponds to a set S_{ij}. Each entry contains the values of the indices $i1$ and $j1$ indicanting the row pair $(i1, j1)$, a field called *covered* to indicate the set has been covered (by a gene of the hitting set, to be explained later) and a pointer to the list of genes that belongs to the set. The entries are numbered from 0 to to the number of sets. It can be seen that the number of sets is at most $O(m^2)$.

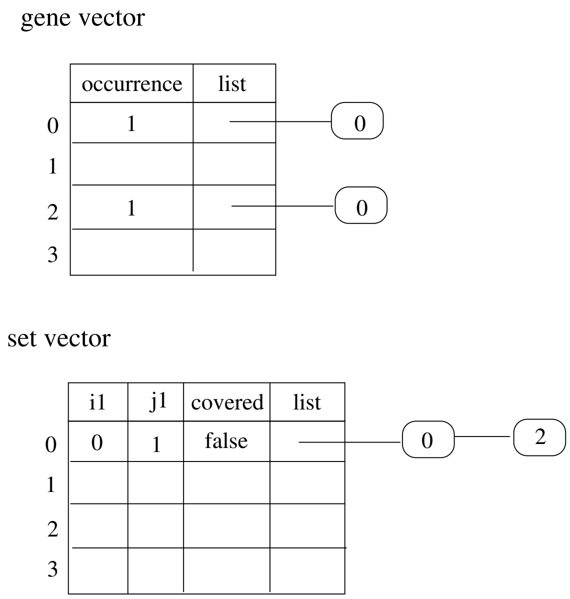

Figure 3. The gene vector and set vector after considering pair (0,1) of the example.

Let us show how the data structures of Figure 2 are constructed. Initially all the pointers in both gene and set vectors are nil. Consider the row pair (0,1) where the expression levels of the target gene a_3 differ. In this row pair, the expression levels of genes a_0 and a_2 also differ. So these two genes constitute the first set $S_{01} = S_0 = \{a_0, a_2\}$ and this is indicated in the list corresponding to the set 0, in the set vector. In the gene vector, genes 0 and 2 occur once each and their list pointers point to set 0. The two data structures after considering row pair (0,1) are shown in Figure 3.

Similarly this is done to all row pairs where the expression levels of the target gene differ and we have the final situation shown in Figure 2.

5.2 The Greedy Strategy to Find the Hitting Set

Having constructed the data structures from the given input, we are now ready to use the greedy strategy to find an approximate answer for the hitting set. In this strategy, we repeatedly choose the element that covers the most uncovered sets, that is, the one with the maximum value of *occurrence*.

We start by examining the gene vector and find the gene with the maximum value in *occurrence*. In the example we have three genes with the maximum occurrence 2. Suppose we select gene a_0 to be part of the hitting set. Since gene a_0 appears in sets 0 and 2, we mark these two sets as *covered* by setting the field *covered* to true. Also we update the *occurrence* field of all genes that belong to the covered sets. Thus the occurrence fields of genes 0, 1, and 2 will be 0, 1 and 1 respectively.

Now we restart all over again to pick the gene with the maximum occurrence, until all sets are covered.

5.3 Sequential Approximation Algorithm

We now present the sequential approximation algorithm based on the greedy strategy.

Sequential Approximation Algorithm:

Input: An $m \times n$ matrix M, an $m \times 1$ vector v, the index r indicating the target gene a_r.

Output: A vector HS containing the genes the target gene a_r depends on.

Observation: The algorithm builds the data structures and calls the GreedyHitSet algorithm to obtain the output. For our data structures, we call the gene vector E and the set vector S.

$size \leftarrow 0$
for ($i = 0$ to $m-1$) **do**
  for ($j = i+1$ to $m-1$) **do**
    if ($v[i]$ and $v[j]$ are different) **then**
      $S[size].i1 \leftarrow i$
      $S[size].j1 \leftarrow j$
      $S[size].covered \leftarrow false$
      for ($k = 0$ to $n-1$) **do**
        if ($k \neq r$) **then**
          if ($M[i][k]$ and $M[j][k]$ are different) **then**
            $E[k].occurrence \leftarrow E[k].occurrence + 1$
            Insert k in $S[size].list$
            Insert $size$ in $E[k].list$
          end if
        end if
      end for
      $size \leftarrow size + 1$
    end if
  end for
end for
$HS \leftarrow$ GreedyHitSet
return HS;

GreedyHitSet Algorithm:

Input: Gene vector E and its size n, set vector S and its size $size$.

Output: A vector HS containing the genes the target gene depends on.

Observation: The list elements have two fields: an info field and a link field pointing the the next element or to nil.

1: **while** ($size > 0$) **do**
2:   Select the gene x with the maximum value of $occurrence$ in E.
3:   Insert x in vector HS
4:   $Y \leftarrow E[x].list$
5:   **while** ($Y \neq nil$) **do**
6:     $z \leftarrow info(Y)$
7:     **if** ($S[z].covered = false$) **then**
8:       $S[z].covered \leftarrow true$
9:       $size \leftarrow size - 1$
10:       $K \leftarrow S[z].list$
11:       **while** ($K \neq nil$) **do**
12:         $s \leftarrow info(K)$
13:         $E[s].occurrence \leftarrow E[s].occurrence - 1$
14:         $K \leftarrow link(K)$
15:       **end while**
16:     **end if**
17:     $Y \leftarrow link(Y)$
18:   **end while**
19: **end while**
20: return HS

5.4 Space and Time Complexities of the Sequential Algorithm

The Sequential Approximation Algorithm constructs the data structures that takes $O(nm^2)$ time and then calls the GreedyHitSet algorithm. The GreedyHitSet algorithm has a main loop that is executed $size$ times, where $size$ is the size of the set vector S, that is $O(m^2)$. Inside this main loop, line 2 obtains the maximum in $O(n)$ time, the loop in line

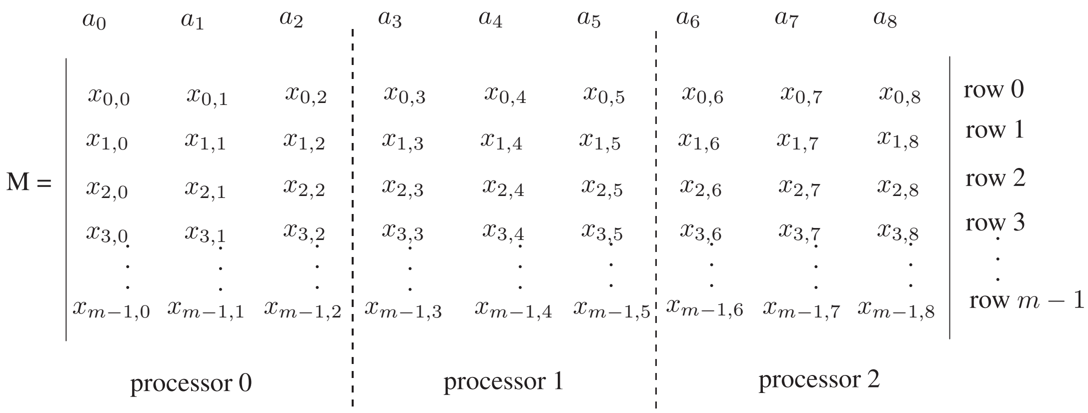

Figure 4. Partitioning the input matrix onto the processors.

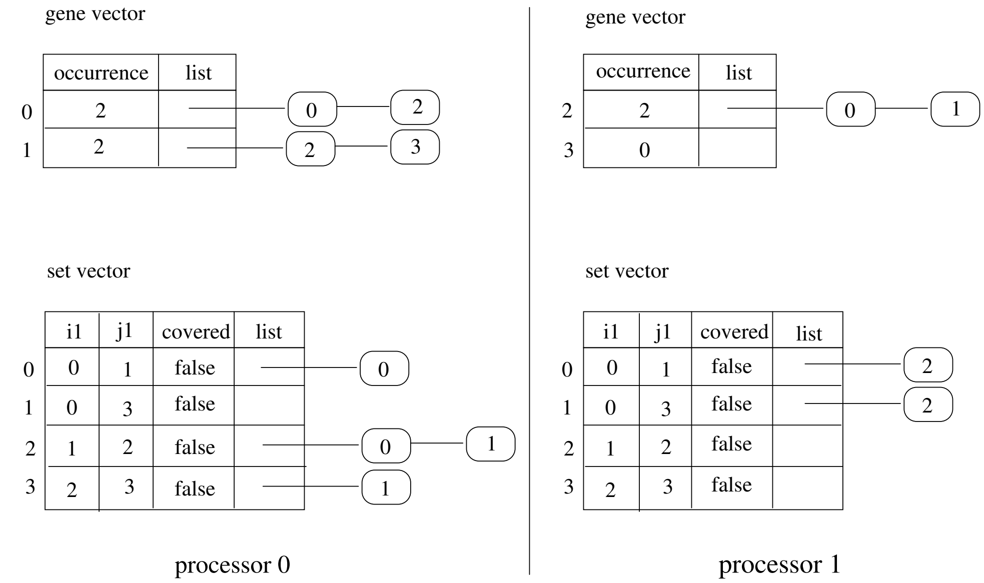

Figure 5. The data structures for the parallel algorithm.

5 is executed $O(m^2)$ times and the nested loop of line 11 takes $O(n)$ time thus the loop of line 5 takes $O(m^2 n)$ time. Therefore the main loop of line 1 takes $O(m^4 n)$ which is also the overall time complexity of the Sequential Approximation Algorithm.

In terms of space complexity, two $O(m^2 n)$ size data structures are used.

6 The Parallel Approximation Algorithm

The input $m \times n$ matrix M is partitioned vertically to be stored in each processor, as shown in Figure 4.

The corresponding parallel algorithm is obtained in a very elegant and efficient way, by splitting the responsibilities of each processor so that each deals only with those genes stored in it. Not only the algorithm is naturally parallelized, also the data structures can be split basically into p parts. Thus the computing time and space requirements are equally divided by the p processors.

Each processor has a gene vector and a set vector, however, these data structures take only $1/p$ of the original space of the sequential algorithm. This is shown in Figure 5. Notice that the gene vector of each processor contains only entries that refer to the genes stored in the processor. Regarding the set vector in each processor, the lists of genes in the data structure again only refer to the genes stored in the processor.

The processors need only to communicate to find the maximum of the *occurrence* field (to obtain a gene to be inserted into HS) and do the update of the *covered* and *oc-*

currence fields. Therefore the number of communication rounds is equal to the size of HS.

The resulting parallel approximation algorithm is shown in the following.

The Parallel Approximation Algorithm for processor i:

Input: A partition of the input matrix M, that is, M_i of size $m \times \frac{n}{p}$ of M corresponding to $\frac{n}{p}$ genes stored in processor i, a vector v_i of size m corresponding to the expression values of the target gene, and an index r of this target gene.

Output: A vector HS containing the genes the target gene depends on. Processor P_0 will store the vector HS.

Observations: We denote by E_i the gene vector and by S_i the set vector of processor i. We further assume, for simplicity, that the gene vector of processor i has indices that start at $(i\frac{n}{p})$. For the same reason, the matrix M_i of each processor will have its rows indexed by 0 to $m-1$ and its columns indexed by $(i\frac{n}{p})$ to $(((i+1)\frac{n}{p})-1)$. The list elements have two fields: an info field and a link field pointing to the next element or to nil.

```
size ← 0
for (x = 0 to m − 1) do
  for (y = x + 1 to m − 1) do
    if (v_i[x] and v_i[y] differ) then
      S_i[size].i1 ← x
      S_i[size].j1 ← y
      S_i[size].covered ← false
      for (z = (i n/p) to (((i + 1) n/p) − 1)) do
        if (z ≠ r) then
          if (M_i[x][z] and M_i[y][z] differ) then
            E_i[z].occurrence ← E_i[z].occurrence+
            1
            Insert z in S_i[size].list
            Insert size em E_i[z].list
          end if
        end if
      end for
      size ← size + 1
    end if
  end for
end for
while (size > 0) do
  x ← the element with maximum occurrence value in
  E_i
  if (i ≠ 0) then
    - send x, E_i[x].occurrence and i to P_0
    - receive from P_0 the id number of processor that
    contains the element chosen by P_0 as the maxi-
    mum between the maximum values received and
    call this id owner
  else
    - receive x, E_i[x].occurrence and sender from
    all processors
    - x ← element of maximum occurrence value among
    all maximum values received from other proces-
    sors.
    - owner = id number of the processor that con-
    tains element x
    - insert x in HS
    - inform all processors that owner has been cho-
    sen
  end if
  if (i = owner) then
    send the list E_owner[x] to all processors and put it
    in Y
  else
    receive the list E_owner[x] from the processor owner
    and put it in Y
  end if
  while (Y ≠ nil) do
    z ← info(Y)
    if (S_i[z].covered = false) then
      S_i[z].covered ← true
      size ← size − 1
      K ← S_i[z].list
      while (K ≠ nil) do
        s ← info(K)
        E_i[s].occurrence ← E_i[s].occurrence − 1
        K ← link(K)
      end while
    end if
    Y ← link(Y)
  end while
end while
```

6.1 Space and Time Complexities of the Parallel Algorithm

Note that the parallel algorithm partitions the entire sequential $O(m^4 n)$ processing in a very natural way equally among the p processors. The parallel algorithm thus takes $O(m^4 n/p)$ time.

In addition to the processing time, the parallel algorithm requires $O(k)$ communication rounds where k is the number of elements in the hitting set. We can express the hitting set size k in terms of the input parameter m. It is clear that the hitting set of a collection of sets $\mathcal{S}$ cannot be larger than the cardinality of $\mathcal{S}$, which is in turn limited by $O(m^2)$. Thus the parallel algorithm requires $O(m^2)$ communication rounds.

The sequential algorithm requires $O(m^2 n)$ space which is conveniently partitioned among the p processors giving the space requirement of $O(m^2n/p)$ in each processor.

7 Experimental Results

Input matrix	p=1	p=2	p=4	p=8
20 × 1024	0.010228	0.005328	0.003695	0.002597
20 × 2048	0.022747	0.011604	0.006052	0.003892
20 × 4096	0.045081	0.022191	0.011053	0.006871

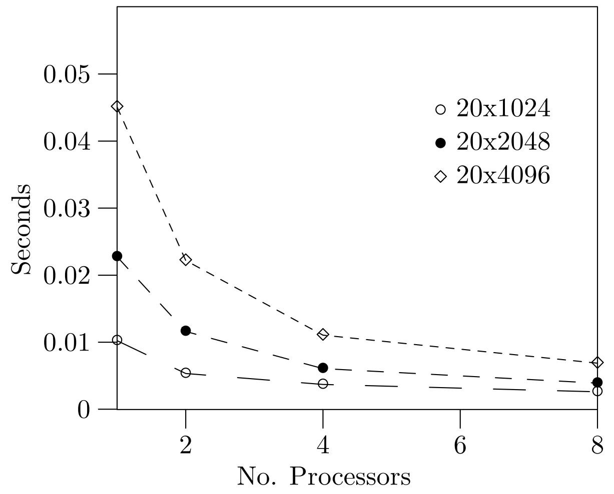

Figure 6. Curves of the observed times

We have implemented the proposed algorithm on a Beowulf with 8 nodes. Each node has a 1.2GHz AMD Athlon processor and 768 MB of RAM memory. The nodes are connected through a 100 MB interconection network.

The preliminary results shown above are very promissing and show good sppedup and scalability of the algorithm.

8 Conclusion

Given the results of an experiment with n genes and m measures over time we consider the problem of finding the subset of k genes that explain the expression level of a given target gene. We first present a sequential approximation algorithm of $O(m^4n)$ time and $O(m^2n)$ space. The main result is a new parallel approximation algorithm that determines the k genes in $O(m^4n/p)$ local computing time plus $O(k)$ communication rounds, and requires $O(m^2n/p)$ space. Thus the parallel algorithm has been obtained in a very elegant and efficient way, by splitting the responsibilities of each processor so that each deals only with those genes stored in it. Not only the algorithm is naturally parallelized, also the data structures can be split basically into p parts.