PARALLEL COMPUTING FOR RECONSTRUCTING PHYSICAL MAPS OF CHROMOSOMES

by

JINLING HUANG

(Under the Direction of SUCHENDRA BHANDARKAR)

ABSTRACT

This study designs and implements the parallel algorithms with several optimization approaches including simulated annealing, large step Markov chains (LSMC), evolutionary programming, and genetic algorithms, for a physical mapping problem based on the maximum likelihood estimator model. The parallel algorithms are implemented using a combination of inter-process communication via message passing and shared memory multithreaded programming and have provided good performance. Genetic algorithms using a heuristic crossover operator yields better results in terms of both solution accuracy and performance compared to the simulated annealing, LSMC and evolutionary programming approaches.

INDEX WORDS: Physical Mapping, Parallel Computing, Maximum Likelihood Estimator, Simulated Annealing, Large Step Markov Chains, Evolutionary Programming, Genetic Algorithm, MPI, Pthreads

PARALLEL COMPUTING FOR RECONSTRUCTING PHYSICAL MAPS OF CHROMOSOMES

by

JINLING HUANG

B.Agr., Henan Agricultural University, China, 1984

M.S., Kunming Institute of Botany, The Chinese Academy of Sciences, China, 1989

Ph.D., The University of Georgia, 2000

A Thesis Submitted to the Graduate Faculty of The University of Georgia

in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2002

PARALLEL COMPUTING FOR RECONSTRUCTING PHYSICAL MAPS OF CHROMOSOMES

by

JINLING HUANG

ACKNOWLEDGEMENTS

I am grateful to Dr. Suchi Bhandarkar for his support and guidance throughout my study, without which this project would have never been finished. I am also thankful to Drs. Hamid Arabnia and Thiab Taha for serving on my graduate advisory committee. Mr. Raghu Kota and Mr. Nan Li also provided valuable assistance and discussions for this project.

Finally, I would like to thank the UGA Computer Science department and all my friends for their support and helps during my study.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION TO PHYSICAL MAPPING AND PARALLEL COMPUTING

Mapping genetic markers on a chromosome is a central issue in the understanding of the genetic structure, functions and evolution of an organism. The importance of chromosome mapping is reflected in the major international cooperative efforts to study the whole genomes of several organisms, including *Homo sapiens*, *Mus musculus*, *Arabidopsis thaliana*, *Escherichia coli* etc. Chromosomal maps can be broadly divided into two major types, i.e., genetic maps and physical maps. Genetic maps represent genetic markers in their relative order along the chromosome, where the distance between two markers is a measure of their recombination frequency and denoted by centimorgans. Two genetic markers are one centimorgan apart if the recombination rate between them is 1%. Although genetic maps can be used to estimate the physical distance between genetic markers, the result of the estimation is often not very reliable since recombination frequencies vary in different regions of a chromosome (Watson et al. 1992). Physical mapping, on the other hand, determines physical locations of genetic markers on a chromosome. The distance between two markers in a physical map is measured by the number of intervening nucleotide base pairs. As a result, a physical map is of higher resolution and becomes a powerful tool to isolate genes and to study the organization and evolution of genomes.

This chapter gives a brief introduction to the procedure of physical mapping of chromosomes and parallel computing. In chapters 2, 3, and 4, three different computational models, i.e., simulated annealing (SA), large step Markov chain (LSMC) and the evolutionary algorithm, and their parallel implementation for constructing physical maps are discussed. Chapter 5 summarizes the results of this project and outlines directions for future research.

1.1 Physical Mapping of Chromosomes

The procedure of physical mapping can be divided roughly into two steps. First, large pieces of DNA called contigs derived from a library of cloned DNA fragments are ordered according to their positions in the genome. This can be done using techniques such as nonunique probes mapping (Alizadeh et al. 1995), unique probes mapping (Alizadeh et al. 1994; Greenberg and Istrail 1995; Jain and Myers 1997), unique endprobes mapping (Christof et al. 1997), restriction fragments mapping (Fasulo et al. 1997; Jiang et al. 1997), radiation-hybrid mapping (Ben-Dor and Chor 1997; Slonim et al. 1997), and optical mapping (Muthukrishnan and Parida 1997; Karp and Shamir 1998; Lee et al. 1998). Second, the cloned fragments are cut by restriction enzymes, and smaller DNA fragments are obtained and sequenced (shotgun-sequencing), and the detailed sequence is obtained via a sequence assembly procedure.

The physical mapping technique used in this project is based on *sampling without replacement* protocol (Prade et al. 1997). This protocol has been used successfully for several fungal organisms, including *Aspergillus nidulans*, *Aspergillus flavus*, *Schizosaccharomyces pombe*, *Pneumocystis carinii* etc. Under this protocol, chromosomes are first isolated using pulsed field gel electrophoresis and then radiolabeled. Each chromosome is then used to probe the genomic library *L*. As a result, clones in the genomic library can be sorted into three subsets *S*, *R*, and *O*. Clones in subset *S* are specific to a single chromosome of the organism under study whereas those in subset *R* can hybridize to several but not all the chromosomes. Clones in subset *O* can hybridize to all the chromosomes. Clones in subset *S* are radiolabeled and used to probe the genomic library *L* to derive a chromosome-specific probe set *P* and a clone set *C*. Specifically the probe set *P* and the clone set *C* can be obtained in the following procedure (Kececioglu et al. 2000; Bhandarkar et al. 2001):

(a) Initially, *P* is empty and *L* contains all the clones.

(b) During the *i*th iteration of the hybridization process, a new probe P_i is selected from the library *L* and used to hybridize against all the remaining clones in *L*. Remove the clone from the library *L* if it has a positive reaction with the probe. The clone/probe hybridization reactions are recorded in a binary hybridization matrix *H*. If the *i*th clone has a positive reaction with the *j*th probe, H_{ij} is coded as 1. Otherwise, H_{ij} is coded as 0 (Figure 1.1).

(c) Repeat step *b* until the library *L* becomes empty.

The result of this selection process will be a probe set *P* consisting of all the probes, a clone set *C* consisting of all the remaining clones in the library *L*, and a hybridization matrix *H* recording all the probe/clone hybridization reactions. The probe set *P* is essentially a maximal set of non-overlapping equal-length clones. If the probes in the probe set *P* can be ordered with respect to their positions along the chromosome, then by examining the 0 and 1 pattern of the probe/clone hybridization matrix, we can also infer the linking clones and construct the minimum overlapping probes and clones, which is also called the minimum tiling, along the chromosome (Figure 1.2). The minimum tiling in conjunction with the sequencing of each individual clone/probe in the tiling could then be used to reconstruct the DNA sequence of the entire chromosome. In practice, many difficulties may be encountered during the analysis. The most common error arises due to a false signature. This occurs when a clone and a probe have a false positive hybridization reaction and H_{ij} is coded as 1 when it in fact should be 0, or conversely a clone and a probe have a false negative hybridization reaction and H_{ij} is coded as 0 when it should be 1 (Bhandarkar et al. 2001).

In this project, we have used a maximum likelihood estimator (MLE) model for constructing physical maps. This model will find the optimal probe ordering and inter-probe spacings with the maximum probability of resulting in the observed data. Essentially it derives an objective function that takes two parameters, probes order and inter-probe spacings, and also takes into account the false hybridization errors. The mostly likely solution will have the lowest objective function value. To optimize the objective function value, we have used simulated annealing and its variant, the LSMC algorithm, and evolutionary algorithms such as genetic algorithm and evolutionary programming. The result can be used to order the entire clone set based on the available hybridization patterns.

Clones	Probes				
	P1	P2	P3	P4	P5
C1	1	0	0	0	0
C2	1	1	0	0	0
C3	1	1	0	0	0
C4	0	1	0	0	0
C5	0	1	1	0	0
C6	0	0	1	0	0
C7	0	0	1	1	0
C8	0	0	0	1	0
C9	0	0	0	1	1
C10	0	0	0	0	1
C11	0	0	0	0	1

Figure 1.1 An example of clone-probe ordering across the chromosome

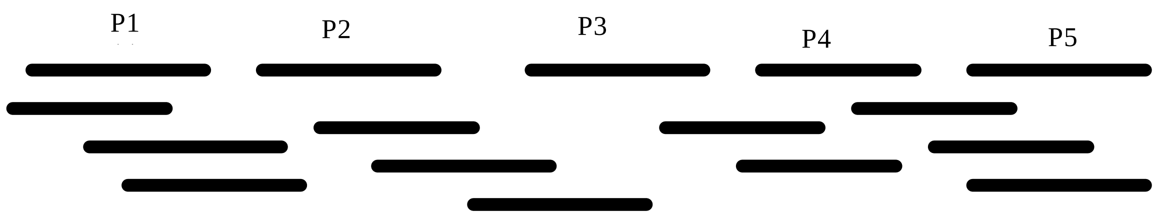

Figure 1.2 An example of clone-probe ordering along a chromosome

1.2 Parallel Computing

By definition, a parallel computer is a set of processing elements that are able to communicate and cooperate to solve a large problem faster than a single-processor computer could. Therefore a parallel architecture can be viewed as an extension of conventional computer architecture with emphasis on the communication and cooperation among the processing elements (Culler et al. 1999). Based on this definition, the term parallel computers essentially include parallel supercomputers, networks of workstations, and shared-memory multiprocessor computers.

According to Flynn (1966), the architecture of a computer can be broadly classified into four categories on the basis of their data streams and instruction streams. They are single instruction single data (SISD), multiple instruction single data (MISD), single instruction multiple data (SIMD), and multiple instruction multiple data (MIMD). Among these possible architectures, the SIMD and MIMD models constitute the most commonly encountered parallel computer systems. The SIMD system is the basic model for data parallelism and usually consists of a parallel array of processors that are usually mesh-connected and execute the same instructions simultaneously. The MIMD system is often used for task parallelism and consists of an assembly of processors that execute any task either independently or in concert with others (Fountain 1994).

Parallel computing involves identifying workloads that can be tackled in parallel, distributing them among processing elements, and managing the necessary data access, communication, and synchronization (Golub and Ortega 1993; Culler et al. 1999). A parallel program consists of several cooperating processes to solve a problem, and therefore is extremely useful for computation-intensive problems.

To parallelize a computational problem and accomplish the goal of a sequential program, several major steps (Culler et al. 1999) are taken:

(a) Identifying available concurrency in the underlying problem and the amount of concurrency exposed, and decomposing a larger computational problem into smaller subtasks.

(b) Assigning tasks to processes. The workload assigned to each process should be balanced, and the communication among processes and the run-time overhead to manage the assignment should be minimized.

(c) Managing the necessary data access, communication, and synchronization among processes. The cost of communication and synchronization should be minimized.

(d) Mapping or binding processes to processors. This can be done either by the program or by the operating system. The program can bind or map a process to a processor and specify which process is supposed to run in certain phases.

The performance of a parallel program is often measured by the speedup, which is defined as $speedup = \frac{Time \quad (1 \quad processor)}{Time \quad (n \quad processors)}$. The speedup of a program is limited by the portion of the computation that can be parallelized. Therefore, it is critical to study the problem and identify the available concurrency before an attempt is made to parallelize it. Another measurement of performance is efficiency, which is defined as $efficiency = \frac{speedup}{number \quad of \quad processors \quad used}$. Since speedup is typically less than the number of the processors used for execution, efficiency should be less than or equal to 1 or 100%.

The most widely used parallel programming models are Shared Memory Programming and Message Passing.

(1) Shared Memory Programming: Communication occurs implicitly as a result of access to a shared memory by multiple processing elements using conventional memory access instructions (Culler et al. 1999). Communication can be between independent processes running on different processors and sharing a common memory segment or between threads of the same process.

Recently, with the increasing popularity of shared-memory, symmetric multiprocessor (SMP) machines, multithreaded programming has become more widely used. A thread is essentially a flow of execution within a process with its own register set and stack. Compared to a process, a thread takes less time to create and terminate. Unlike processes, which require operating system intervention for inter-process communication, communication among threads is more efficient since all threads within a task share a common memory and file system and can communicate without invoking the operating system kernel (Tanenbaum 2001). The implementation of threads can be broadly classified into two major types, i.e., POSIX style threads and WIN32/OS/2 style threads (Prasad 1997). POSIX is a standard document produced by IEEE aiming to provide a standard interface for programming on different operating systems. The POSIX thread (Pthread) standard is supported by most of the UNIX vendors. The Pthread library provides a series of functions for thread creation, termination, management, and synchronization.

(2) Message Passing: Message passing is fundamentally processor-to-processor

communication with explicit I/O operations. Communication in the message passing model is integrated at the I/O level rather than into the memory system (Culler et al. 1999). Each process maintains a local memory, and communication among processes is realized via *send* and *receive* operations. In the simplest form, *send* specifies a local data buffer to be transmitted and a receiving process whereas *receive* specifies a sending process and a local data buffer into which the transmitted data is to be placed (Culler et al. 1999; Gropp et al. 1999). Parallel Virtual Machine (PVM) (Sunderam 1988) and Message Passing Interface (MPI) (Message Passing Interface Forum 1994) are two of the common software packages that allow programmers to program based on this programming model.

In this project, we have used a combination of message passing and shared memory programming models. The choice of the programming model is based on the nature of the problem to be tackled and the ease of use for that model. We first create multiple processes on a cluster of SMPs to parallelize the simulated annealing or the evolutionary algorithm using MPI. Within each process, multiple threads are spawned using Pthreads to parallelize the conjugate gradient descent search and the local exhaustive search procedures. For our specific implementation, we have used the Master/Slaves model. The master is mainly a thread/process responsible for coordinating and synchronizing the computation of slave threads/processes. Slave threads/processes are responsible for the actual computation.

CHAPTER 2

PARALLEL COMPUTING OF MAXIMUM LIKELIHOOD ESTIMATOR MODEL FOR CHROMOSOME PHYSICAL MAPPING

The Maximum Likelihood Estimator (MLE) model for constructing physical maps of chromosomes is based on the work of Shete, Kececioglu, and Arnold (1998). This model determines the ordering of probes Π in the probe set P and the inter-probe spacings Υ under a probabilistic model of hybridization errors due to false positives and false negatives. Under this model, the probes are first ordered as opposed to clones and the inter-probe spacings are calculated. Once the optimal probe orders are determined, the ordering of clones can be obtained by examining the resulting probe/clone hybridization.

2.1 Problem Formulation

The MLE function involves two independent vector parameters, i.e., probe ordering Π and inter-probe spacings Υ. The negative log-likelihood derived from the MLE function is computed as

$$f(\Pi,\Upsilon) = C - \sum_{i=1}^{k} \ln\left\{ R_i - \sum_{j=1}^{n+1} (a_{i,p_j} - 1)(a_{i,p_{j-1}} - 1)\min(\Upsilon_j, \mathrm{M}) \right\} \qquad (2.1)$$

where C is a constant given by

$$C = k\ln(N-M) - P\ln\frac{r}{(1-r)} - nk\ln(1-r) \qquad (2.2)$$

and

$$R_i = \sum_{j=1}^{n+1} (a_{i,pj} - 1)(a_{i,pj-1} - 1) \tag{2.3}$$

$$a_{i,j} = \begin{cases} \frac{h}{(1-r)} & if \quad h_{i,j} = 0 \quad and \quad j = 1,...,n \\ \frac{(h)}{r} & if \quad h_{i,j} = 1 \quad and \quad j = 1,...,n \\ 0 & otehwise \end{cases} \tag{2.4}$$

where

N is the length of the chromosome;

M is the length of a clone/probe;

n is the number of probes selected from the library;

κ is the number of clones in the library;

ρ is the probability of false positive;

η is the probability of false negative;

$H = ((h_{i,j}))_{1 \le i \le k, 1 \le j \le n}$ is the clone/probe hybridization matrix;

H_{ij} is the coded value for *i*th row and *j*th column in the hybridization matrix;

$\Pi = (\pi_1, \ldots, \pi_n)$ is a permutation of {1, …, n};

$Y = (Y_1, Y_2, ..., Y_n)$ is the inter-probe spacing vector and Y_i is the spacing between P_{p_i} and $P_{p_{-1i}}$, Y_1 is the spacing before the first probe and Y_{n+1} is the spacing after the last probe.

Optimization of the MLE function entails minimization of the negative log-likelihood function $f(\Upsilon, \Pi)$, which can be achieved in two levels. At the higher level, simulated annealing is used to optimize the objective function with respect to the discrete parameter Π. At the lower level, a conjugate gradient descent procedure is used to optimize the objective function with respect to the continuous parameter Υ (Bhandarkar et al. 2001).

2.2 Simulated Annealing Algorithm

Simulated annealing is an iterative optimization approach analogous to the gradual cooling process of a physical system. A typical simulated annealing algorithm starts with a given temperature and an initial state. The temperature is then decreased according to an annealing function. Each temperature value corresponds to an annealing step which consists of three phases as described in the following:

a. *Perturb*: An operator is applied to the current state and a new state is generated in this phase. In our case, the perturbation is achieved by reversing the order of a probe block that has been chosen randomly. Therefore the current solution x_i is perturbed to yield a new candidate solution x_j.

b. *Evaluate*: The new candidate solution x_j is evaluated using some criteria. In our case, the objective function value $f(\Pi, \Upsilon)$ of the new probe ordering x_j is calculated. This is achieved by searching for the optimal inter-probe spacing Y for the new order x_j using a conjugate gradient descent search and then calculating the negative log-likelihood function value under the new order x_j and its optimal inter-probe spacings.

c. *Decide*: If $f(x_j) < f(x_i)$, then x_j is accepted as the new solution; otherwise x_j is accepted as the new solution with the probability *p* computed using the Metropolis function $p = \exp\left(-\frac{f(x_j) - f(x_i)}{T_i}\right)$ at the temperature T_i whereas x_i is retained with probability (1-*p*). In our implementation, a random number is generated using a pseudorandom generator with a uniform distribution in the range [0, 1]. If this random number happens to be less than *p*, then x_j is accepted as the current solution. Otherwise x_i is retained.

For a given temperature T_i or an annealing step, a sufficient number of *Perturb – Evaluate - Decide* cycles should always lead to an equilibrium resulting in a stationary Boltzmann distribution of solution states (Mahfoud and Goldberg 1995). Therefore, a sufficient number of iterations should be run to approach the equilibrium. The series of solution states generated at a given temperature or an annealing step constitute a Markov chain since the *i*th solution is constructed strictly from the *i-1*th one. At higher temperatures, since almost any change to the current solution state can be accepted, the uphill movement is more likely and simulated annealing resembles a completely random search. This would prevent the searching from being trapped into a local optimal solution. At lower temperatures, the annealing step is more like a deterministic local search. The temperature continues decreasing until certain predefined criteria are met (Figure 2.1).

```
T = T_max;
Finished = false;
While (not Finished)
```

```
{
  for ($count$ = 1; $count$ <= $COUNT\_LIMIT$;  $count$ = $count$+1)
  {
     1  Phase one – Perturb
           Randomly perturb the current solution to generate a new candidate solution;
     2  Phase two – Evaluate
        (a) Compute the objective function value for the new candidate solution;
        (b) Compute $f\_delta$, the change in the value of the objective function;
     3  Phase three  - Decide
          If ($f\_delta < 0$)
             Accept the new solution.
          Else
             Accept the new solution with the probability $p$ computed using the
             Metropolis function.
  }
  Update the temperature using the annealing function $T = A(T)$;
}
```

Figure 2.1 Outline of a typical simulated annealing algorithm

2.3 Parallel Computing of Simulated Annealing Using MPI

Two slightly different approaches are used to parallelize the simulated annealing algorithm on a network of SMPs in this project.

a. Non-Interacting Local Markov Chain (NILM): This approach is based on multiple independent searches. The total number of iterations in a given annealing step is divided evenly among the SMP machines in the network. Each machine is assigned a unique random seed number to generate its own Markov chain of solutions. The tasks run asynchronously on each machine. At the end of the annealing process, the locally optimal solutions obtained from all the SMPs are compared and the one with the lowest value is chosen as the final solution (Figures 2.2, 2.3). In the case of our MPI implementation, the local optimal objective values from slaves are sent to the master via function calls *MPI_Send* and *MPI_Recv*. The master compares and identifies the global optimal value and then broadcasts its identifier to all the processes via the function call *MPI_Bcast*. After receiving the identifier, the process with the global optimal value will send its local optimal solution to the master.

```
Finished = false;
While (not Finished)
{
   for (count = 1;  count <= COUNT_LIMIT;  count = count +1)
   {
      1   Phase One – Perturb
          Randomly perturb the existing solution to generate a new solution;
      2   Phase Two – Evaluate
          (a) Compute the objective function value for the new candidate solution;
```

```
        (b) Compute *f_delta*, the change in the value of the objective function;
    3   Phase Three – Decide
        If (*f_delta* < 0)
            Accept the new solution.
        Else
            Accept the new solution with the probability *p* computed using
            the Metropolis function.
    }
    Update the temperature using annealing function $T = A(T)$;
}
(a) Receive local optimal function values from slaves;
(b) Compare local optimal objective function values from slaves and identify the global
    minimal value and its process identifier (*minId*);
(c) Broadcast *minId* to all processes;
(d) Receive probe ordering and inter-probe spacings from the process *minId*.
```

Figure 2.2 Master process for the NILM algorithm

```
Finished = false;
While (not Finished)
{
   for (count = 1;  count <= COUNT_LIMIT;  count = count +1)
```

```
  {
    1  Phase One – *Perturb*
       Randomly perturb the existing solution to generate a new solution;
    2  Phase Two – *Evaluate*
       (a) Compute the objective function value for the new candidate solution;
       (b) Compute *f_delta*, the change in the value of the objective function;
    3  Phase Three – *Decide*
       If (*f_delta* < 0)
           Accept the new solution.
       Else
           Accept the new solution with the probability *p* computed using
            the Metropolis function.
  }
  Update the temperature using the annealing function $T = A(T)$;
}
(a) Send local optimal function value to the master;
(b) Receive from the master the identifier *MinId* of the process which yields the global
    minimal objective function value;
(c) If (*MinId* = *myId*)
        Send optimal probe ordering to the master;
        Send optimal inter-probe spacings to the master.
```

Figure 2.3 Slave process for the NILM algorithm

b. Periodically Interacting Local Markov Chain (PILM): This approach differs from NILM in that, instead of evaluating local objective solutions at the end of the simulated annealing function, the local solutions from slaves are compared once an annealing step is completed. In the case of our MPI implementation, slaves send their local optimal solutions to the master at the end of each annealing step. The master compares and identifies the one with the lowest local objective function value as the optimal solution at that annealing step and broadcasts this solution to all processes as the current solution for the next annealing step (Figures 2.4, 2.5).

```
T = T_max;
Finished = false;
While (not Finished)
{
   for (count = 1;  count <= COUNT_LIMIT;  count = count+1)
   {
      1   Phase one – Perturb
          Randomly perturb the current solution to generate a new candidate solution;
      2   Phase two – Evaluate
          (a) Compute the objective function value for the new candidate solution;
          (b) Compute f_delta, the change in the objective function value;
      3   Phase three  - Decide
            If (f_delta < 0)
            Accept the new solution.
        Else
            Accept the new solution with the probability p computed using the
            Metropolis function.
    }
    (a) Receive local minimal objective function values and process Ids from slaves;
    (b) Compare the local minimal objective function values obtained from slaves and
        identify the global minimal objective function value and its associated process Id
        (minId);
    (c) Broadcast minId to all the processes;
    (d) Receive probe ordering and inter-probe spacings from process minId;
    (e) Update the temperature using the annealing function $T = A(T)$;
}
```

Figure 2.4 Master process of the PILM algorithm

```
T = T_max;
Finished = false;
While (not Finished)
{
    for (count = 1; count <= COUNT_LIMIT; count = count+1)
    {
        1   Phase one – Perturb
```

```
            Randomly perturb the current solution to generate a new candidate solution;
        2   Phase two – *Evaluate*
            (a) Compute the objective function value for the new candidate solution;
            (b) Compute f_delta, the change in the value of the objective function;
        3   Phase three - *Decide*
              If ($f\_delta < 0$)
                  Accept the new solution;
              Else
                  Accept the new solution with the probability $p$ computed using the
                  Metropolis function;
    }
            Send local minimal objective function value and process Id to the master;
            Receive minId from the master;
            If ($myId = minId$)
                  Broadcast probe ordering and inter-probe spacings to all processes;
            Update the temperature using the annealing function $T = A(T)$;
}
```

Figure 2.5 Slave process for the PILM algorithm

2.4 Conjugate Gradient Descent Procedure

The MLE objective function is a convex function with respect to inter-probe spacings Y for a given probe ordering ? . Thus it possesses a unique local minimum that is also its global minimum (Kota 2000; Bhandarkar et al. 2001). This global minimum can be reached using local search techniques such as the steepest descent and conjugate gradient descent procedures. The steepest descent technique is an iterative procedure consisting of three steps: (1) Start with an initial feasible value of Y, (ii) Compute the downhill gradient at Y and (iii) Minimize the objective function in the direction of this downhill gradient and update the value of Y (Kota 2000; Bhandarkar 2001).

The initial value of Y = (Y_1, …, Y_n) is initialized by evenly dividing the sum of inter-probe spacings among each of the Y_i's. Once the initial Y has been determined, we can estimate the local downhill gradient of the MLE objective function by

$$-\nabla f(\Pi, \hat{Y}) = -\left(\frac{\partial f(\Pi, Y)}{\partial Y_1}, \ldots, \frac{\partial f(\Pi, Y)}{\partial Y_n}\right)\Big|_{Y=\hat{Y}} = (U_1, \ldots, U_n)\Big|_{Y=\hat{Y}} \tag{2.5}$$

where

$$U_l = \sum_{i=1}^{k} \frac{-(a_{i,p_l}-1)(a_{i,p_{l-1}}-1)\mathrm{I}(Y_l) - ((a_{i,p_n}-1)\mathrm{I}(Y_{n+1}))}{R_i - \sum_{j=1}^{n+1}(a_{i,p_j}-1)(a_{i,p_{j-1}}-1)\min(Y_j, M)} \tag{2.6}$$

and

$$\mathrm{I}(x) = \begin{cases} 1 & if \quad x \le M \\ 0 & otherwise \end{cases} \tag{2.7}$$

The current value of $Y = Y_{old}$ is updated to yield a new value such that $Y_{new} = Y_{old} + sU$, where s is a scalar, by moving along the local downhill gradient direction $U = -\nabla f(\Pi, \hat{Y})|_{\hat{Y} = Y_{old}}$. To minimize the MLE objective function f along the local downhill gradient, an optimal value of $s = s^*$ needs to be determined such that

$$f(\Pi, \hat{Y} + s^* U) = \min_s f(\Pi, \hat{Y} + sU) \tag{2.8}$$

where

$$f(\Pi, \hat{\mathrm{Y}} + sU) = C - \sum_{i=1}^{k} \ln\left\{ R_i - \sum_{j=1}^{n+1} (a_{i,p_j} - 1)(a_{i,p_{j-1}} - 1) \min(\hat{\mathrm{Y}}_j + sU_j, \mathrm{M}) \right\} \quad (2.9)$$

and

$$\mathrm{Y}_{n+1} = N - n\mathrm{M} - \sum_{i=1}^{n} \mathrm{Y}_i \,. \quad (2.10)$$

To find s^*, we consider

$$\frac{\partial f(\Pi, \hat{\mathrm{Y}} + sU)}{\partial s} = \sum_{i=1}^{k} \frac{\sum_{j=1}^{n+1} (a_{i,p_j} - 1)(a_{i,p_{j-1}} - 1) U_j I(\hat{\mathrm{Y}}_j + sU_j)}{R_i - \sum_{j=1}^{n+1} (a_{i,p_j} - 1)(a_{i,p_{j-1}} - 1) \min(\hat{\mathrm{Y}}_j + sU_j, \mathrm{M})} \quad (2.11)$$

where $U_{n+1} = -\sum_{i=1}^{n} U_i$. In addition, since the MLE objective function $f_\Pi(\mathrm{Y})$ is convex with respect to Y, the local optimum for *s* is also a global optimum. Also, the boundary conditions on the inter-probe spacings result in the following constraints:

(i) $Y_j + sU \geq 0$, for *j = 1, …, n*. i.e., no inter-probe spacing should be nonnegative.

(ii) $\sum_{j=1}^{n} (\mathrm{Y}_j + sU_j) \leq N - n\mathrm{M}$, i.e., the sum of the inter-probe spacings should not exceed the sum of available gaps, which is $N - nM$.

With these constraints, we can further bracket the value of *s* as follows:

$$0 \leq s \leq \min\left\{ \min_{j \in (1,\ldots,n+1): U_j < 0} \left\{ \frac{-\hat{\mathrm{Y}}_j}{U_j} \right\}, \min_{j \in (1,\ldots,n+1): U_j > 0} \left\{ \frac{\mathrm{M} - \hat{\mathrm{Y}}_j}{U_j} \right\} \right\} \quad (2.12)$$

Once the upper and lower limits of *s* have been determined, *s** can be calculated using the bisection method (Press et al. 1988). New inter-probe spacings Y can be calculated as $\mathrm{Y}_{new} = \mathrm{Y}_{old} + s^* U$.

Under the inter-probe spacing constraints, the value of Y_i's, where $1 \le i \le n+1$, lies in the range between 0 and M. If the Y vector is on the boundary defined by the constraints and the local downhill gradient vector U points outside the feasible region at that point, we will have to reset the value of *s* to zero and stop the iterative procedure even though the gradient has not vanished (Bhandarkar et al. 2001). This is handled by the *Project* routine in this project.

The gradient computation and the solution update steps of the steepest descent or conjugate gradient descent procedures are carried out until the gradient vector attains a magnitude less than a predefined threshold. Ideally, the gradient vector should become 0. But this may not happen because of the numerical errors associated with the computation of *s* and the gradient vector on the computers (Machaka 1998; Kota 2000; Bhandarkar et al. 2001).

The serial algorithm for conjugate gradient descent search is outlined in Figure 2.6.

```
1   Start with an initial guess of $Y = Y_i$;
    Calculate gradient $G = \nabla f(\Pi, Y_i)$;
    $G = G_1 = G_2 = -G$;
2   while(1)
    {
        Project $G$;
        If ($G$ vanished)
            Break;
            Bracket the minimum along the direction $G$ ;
            Minimize along the direction $G$ : Find the optimal $s*$ such that
                $f(\Pi, \Upsilon_i + s*G) = \min_s f(\Pi, \Upsilon_i + s*G)$;
            $\Upsilon = \Upsilon + s*G$;
            $\Delta f = f(\Pi, \Upsilon_i) - f(\Pi, \Upsilon_{i+1})$;
            $\Upsilon_i = \Upsilon_{i+1}$;
            if $(\Delta f < 10^{-5})$
                Break;
            Calculate gradient $G = \nabla f(\Pi, \Upsilon_i)$;
            $g_1 = (G + G_2)G$;
            $g_2 = G_2 G_2$;
            $g_3 = g_1 / g_2$;
            $G_2 = -G$;
            $G = G_1 = G_2 + g_3 G_1$;
        }
3   $\Upsilon = \Upsilon_i$;
```

Figure 2.6 Serial algorithm for conjugate gradient descent search

2.5 Parallelization of the Conjugate Gradient Descent Procedure

Parallelization of the conjugate gradient descent procedure follows the data parallelism model, where several processing elements perform an action on separate subsets of the data set simultaneously. In our project, the gradient vector G and inter-probe spacings Y are divided into subsets to be processed concurrently. Implementation of the parallel algorithm follows the Master/Slave model, where both the master and slaves are implemented using POSIX threads. Information on shared variables is updated globally. The contention scope of the threads is set to *PTHREAD_SCOPE_SYSTEM* using function *pthread_att_setscope*. This would ensure that all threads within a process are scheduled globally in the system. Slaves are responsible for most of the computation. Coordination and synchronization among slaves are carried out by the master. Synchronization is realized using data types *mutex* and *semaphore* from the Pthread library and the *barrier* variable implemented by us. *Mutex* is used to ensure that a critical section is executed atomically. *Semaphore* is used to coordinate the order of execution between the master and slaves. The *barrier* variable is employed so that no thread can proceed further until all threads reach the same phase. This would prevent certain threads from updating the global variables when some other threads are still using them. Two *barrier* variables have been used in our implementation. One of these *barrier* variables is used for coordinating the execution of slaves. This is useful when the computation is conducted by slaves and does not need coordination by the master. The other *barrier* variable is used when coordination and synchronization by the master are necessary. Each slave thread is bound onto a processor using the UNIX function call *processor_bind* so that the time spent in switching among processors is reduced to a minimum. The master is not bound to any processor since the time used by the master for coordination and synchronization is insignificant compared to that used by the slaves.

The algorithms for the master and slave threads are shown in Figures 2.7 and 2.8 respectively.

```
1   Start with initial guess of $\mathrm{Y} = \mathrm{Y}_i$;
    Divide the $\mathrm{Y}$ and $G$ by marking the bounds of the beginning and end of each
    subvector to be acted up by slaves;
    Spawn the slave threads and pass them the bounds of their subvectors as the
    arguments;
2   while (1)
    {
        (a) Project with the master thread to coordinate the slave actions;
            *Master-Slave barrier* to allow all threads finishing *Project;*
        (b) Read *exit-bool[index]* for all the slaves.
            If all *exit-bool[index]* are 1
                *EXIT* = 1
                Break;
            Else
                *EXIT* = 0;
        (c) Bracket with the master thread to coordinate slave threads;
        (d) Minimization with the master to coordinate slave actions.  Minimizing along
            the direction $G$ to find the optimal $s*$ such that
            $f(\Pi, \mathrm{Y}_i + s^* G) = \min_s f(\Pi, \mathrm{Y}_i + sG)$; Update $\mathrm{Y}$ using $\mathrm{Y}_{i+1} = \mathrm{Y}_i + s^* G$;
        (e) $\Delta f = f(\Pi, \Upsilon_i) - f(\Pi, \Upsilon_{i+1})$;
            If $(\Delta f < 10^{-5})$
                EXIT = 1;
                Break;
        (f) Gradient with the master to coordinate slave actions.
        (g) Calculate global variables $G_1$ and $G_2$ and coordinate generating new
            gradient for next iteration;
    }
```

Figure 2.7 Master conjugate gradient descent procedure using Pthreads

```
1   Read the bounds for $\Upsilon_{ci}$ and initialization data;
    Calculate gradient $G_c = \nabla f(\Pi, \Upsilon_{ci})$;
    $G_c = G_{c1} = G_{c2} = -G_c$;
2   while (1)
    {
        (a) Project $G_c$;
        (b) If ($G_c$ vanished)
                exit-bool[index] = 1;        //index here is the process Id
        (c) Read EXIT;
            If ( EXIT = 1)
                Break;
    (d) Bracket the minimum along the direction $G_c$ by updating the values of
         SLOW and SHIGH variables;
    (e) Minimize along the direction G to find the $s*$ such that
        $f(\Pi, \Upsilon_i + s^* G) = \min_s f(\Pi, \Upsilon_i + sG)$;
    (f) Update $\Upsilon$ using $\Upsilon_{i+1} = \Upsilon_i + s^* G$;
    (g) Read EXIT;
        If (EXIT = 1)
            Break;
    (h) Calculate gradient $G_c = \nabla f(\Pi, \Upsilon_{ci})$;
        $g_1 = (G_c + G_{c2})G_c$;
        $g_2 = G_{c2} G_{c2}$;
    (i) Do following operations using mutex-lock:
        $G_1 += g_1$;
        $G_2 += g_2$;
        $g_3 = g_1 / g_2$;
        $G_{c2} = -G_c$;
        $G_c = G_{c1} = G_2 + g_3 G_{c1}$;
}
```

Figure 2.8 Slave conjugate gradient descent procedure using pthreads

2.6 Parallel Computing for MLE Using the Combination of MPI and POSIX Threads

Parallel programs of the MLE were run on the SMP cluster in the Computer Science Department at the University of Georgia, which consists of 1 front-end server (babbage) and 8 identical compute nodes. Each compute node is a SunOS SMP machine with 4 processors.

To exploit the computing power of the cluster, we have parallelized the physical mapping algorithms at two levels using a combination of message passing and shared memory programming. At the higher level, the simulated annealing function is parallelized using MPI. Specifically, we created multiple processes on the SMP machines and partitioned the workload of a single annealing step among the processes. Each process starts with an independent seed and generates its own Markov chain of solutions at a given annealing step. Processes communicate using MPI. Under the PILM model, slave processes send their locally optimal solutions to the master when a given annealing step is finished. The master identifies the best solution amongst the slave processes and broadcasts that solution to all the slaves for the next annealing step. When the NILM model is used, inter-process communication via MPI is carried out only at the end of the entire simulated annealing function.

At the lower level, the conjugate gradient descent procedure is further parallelized using multithreaded programming techniques within each process as mentioned above. Outlines of these parallel algorithms are shown in Figures 2.9-2.12.

```
T = T_max;
Finished = false;
```

```
While (not *Finished*)
{
   for (*count* = 1;  *count* <= *COUNT_LIMIT*; *count* = *count*+1)
   {
      1   Phase one – *Perturb*
          Randomly perturb the current solution to generate a new candidate solution;
      2   Phase two – *Evaluate*
          (a) Master conjugate gradient descent procedure. This would spawn slave threads
          for the procedure to compute the optimal inter-probe spacings for the new
          candidate solution;
          (b) Compute the objective function value at the new probe order;
          (c) Compute *f_delta*, the change in the value of the objective function;
      3   Phase three  - *Decide*
            If (*f_delta* < *0*)
                Accept the new solution.
            Else
                Accept the new solution with the probability *p* computed using the
                  Metropolis function.
   }
   Update the temperature using the annealing function *T* = *A(T)*;
}
(a)  Receive local optimal objective function values from slaves;
```

```
(b) Compare local optimal objective function values and identify the global minimal
    values and its process identifier (minId);
(c) Broadcast minId to all the processes;
(d) Receive probe ordering and inter-probe spacings from the process minId.
```

Figure 2.9 Master algorithm of NILM_MLE using MPI and Pthreads

```
T = T_max;
Finished = false;
While (not Finished)
{
   for (count = 1;  count <= COUNT_LIMIT;  count = count+1)
   {
     1   Phase one – Perturb
         Randomly perturb the current solution to generate a new candidate solution;
     2   Phase two – Evaluate
         (a) Master conjugate gradient descent procedure. This would spawn slave
             threads for the procedure and compute the optimal inter-probe spacings for
             the new candidate solution;
         (b) Compute the local minimum objective function value for the new probe
             order;
         (c) Compute f_delta, the change in the value of the objective function;
```

```
    3  Phase three  - Decide
         If (f_delta < 0)
              Accept the new solution.
         Else
              Accept the new solution with the probability p computed using the
              Metropolis function.
    }
    Update the temperature using the annealing function T = A(T);
}
(a)  Send local optimal function value to the master;
(b)  Receive from master the identifier MinId of the process which yields the global
     minimal objective function value;
(c)  If (MinId = myId)
          Send optimal probe ordering to the master;
          Send optimal inter-probe spacings to the master.
```

Figure 2.10 Slave algorithm of NILM_MLE using MPI and Pthreads

```
T = T_max;
Finished = false;
While (not Finished)
{
```

```
for (count = 1;  count <= COUNT_LIMIT;  count = count+1)
{
  1    Phase one – Perturb
       Randomly perturb the current solution to generate a new candidate solution;
  2    Phase two – Evaluate
       (a) Master conjugate gradient descent procedure. This would spawn slave
           threads for the procedure and compute the optimal inter-probe spacings for
           the new candidate solution;
       (b) Compute the objective function value for the new probe order;
       (c) Compute f_delta, the change in the value of the objective function;
  3    Phase three  - Decide
       If (f_delta < 0)
           Accept the new solution.
       Else
           Accept the new solution with the probability p computed using
           the Metropolis function.
 }
  (a) Receive local optimal objective function values and process Ids from slaves;
  (b) Compare local optimal objective function values from slaves and identify the
      global optimal objective function value and its associated process Id (minId);
  (c) Broadcast minId to all the processes;
  (d) Receive probe ordering and inter-probe spacings from process minId;
  (e) Update the temperature using the annealing function T = A(T);
```

```
}
```

Figure 2.11 Master algorithm of PILM_MLE using MPI and Pthreads

```
T = T_max;
Finished = false;
While (not Finished)
{
   for (count = 1;  count <= COUNT_LIMIT;  count = count+1)
   {
      1   Phase one – Perturb
          Randomly perturb the current solution to generate a new candidate solution;
      2   Phase two – Evaluate
          (a) Master conjugate gradient descent procedure. This would spawn slave
              threads for the procedure and compute the optimal inter-probe spacings for
              the new candidate solution;
          (b) Compute the objective function value for the new probe order;
          (c) Compute f_delta, the change in the value of the objective function;
      3   Phase three  - Decide
          If (f_delta < 0)
               Accept the new solution.
          Else
               Accept the new solution with the probability p computed using
```

```
                the Metropolis function.
    }
  (a) Send local optimal objective function value and process Id to the master;
  (b) Receive *minId* from the master;
      If (*myId* = *minId*)
          Broadcast local optimal probe ordering and inter-probe spacings to all the
          processes;
  (c) Update the temperature using the annealing function $T = A(T)$;
}
```

Figure 2.12 Slave algorithm for PILM_MLE using MPI and Pthreads

2.7 Experimental Results

All the algorithms in this project are implemented using the C programming language. The performance of the algorithms is measured using speedup and efficiency as the metrics. Speedups are calculated only for PILM algorithms where the final objective function value for the fastest execution is used as the point *P* for measurement. The speedup is measured for all executions (i.e., executions with number of processors = 4, 8, 16, and 32) using the approximate time to reach *P* against the serial MLE execution time. Due to the nature of parallelization of simulated annealing, different executions often yield different values at a given annealing step. Therefore, their respective time to reach *P* is estimated based on linear interpolation of the time scales of the immediately previous and next annealing steps. Speedups for NILM algorithms are not given since the time to reach *P* cannot be correctly estimated. It should also be noted that since the serial simulated annealing algorithm constructs a single Markov chain of solution states at a given annealing step whereas the parallel simulated annealing algorithms consist of multiple independent ones, this parallelization technique may sometimes lead to speedups that exceed the number of processors used.

Algorithms were tested using artificially generated (i.e., synthetic) datasets (Shete 1998) with the number of probes ranging from 50 – 500 and the real datasets derived from chromosomes II to VII of the fungus *Aspergillus nidulans*. The real datasets were made available by Dr. Jonathan Arnold from the Genetics Department at the University of Georgia. Testing of parallel algorithms of MLE-based physical mapping using a combination of MPI and Pthreads was carried out using 1, 2, 4, 8 SMP machines, each with 4 processors. Therefore, a total of 1, 4, 8, 16, and 32 processors were used respectively. For simulated annealing, the initial temperature was set as 1 and 100 iterations were used for each annealing step. These parameters were empirically determined.

2.7.1 Performance of Conjugate Gradient Descent Procedure

To evaluate the performance of the parallel algorithm for the conjugate gradient descent procedure, the procedure was first tested with a fixed number of iterations for the serial algorithm, and then the same workload was decomposed among the 2-4 processors on the same SMP machine. Both synthetic datasets for probe number *nprobe*=50, 200, 500 and the real dataset *cosmid2* were used for performance comparison and the results are shown in Figures 2.13 and 2.14 and Tables 2.1 and 2.2.

Table 2.1 Speedups for parallel conjugate gradient descent procedure

Dataset	Nproc=1	Nproc=2	Nproc=3	Nproc=4
Nprobe=50	1	1.69	2.32	2.13
Nprobe=200	1	1.81	2.82	2.48
Nprobe=500	1	1.89	2.80	2.99
Cosmid2	1	1.87	2.78	2.79

Table 2.2 Efficiencies (%) for parallel conjugate gradient descent procedure

Dataset	Nproc=1	Nproc=2	Nproc=3	Nproc=4
Nprobe=50	100	85	77	53
Nprobe=200	100	91	94	63
Nprobe=500	100	95	93	75
Cosmid2	100	94	93	70

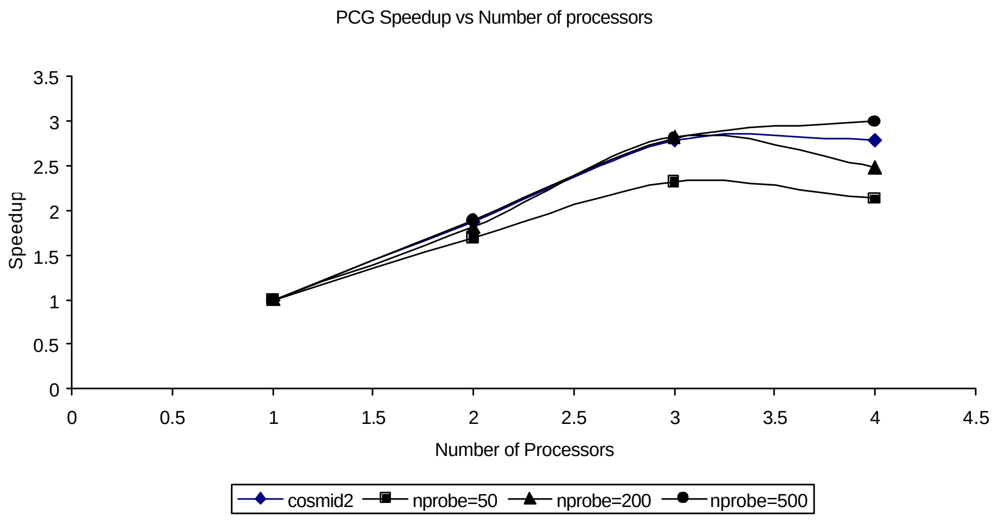

Figure 2.13 Speedup versus number of processors used for the parallel conjugate gradient descent procedure

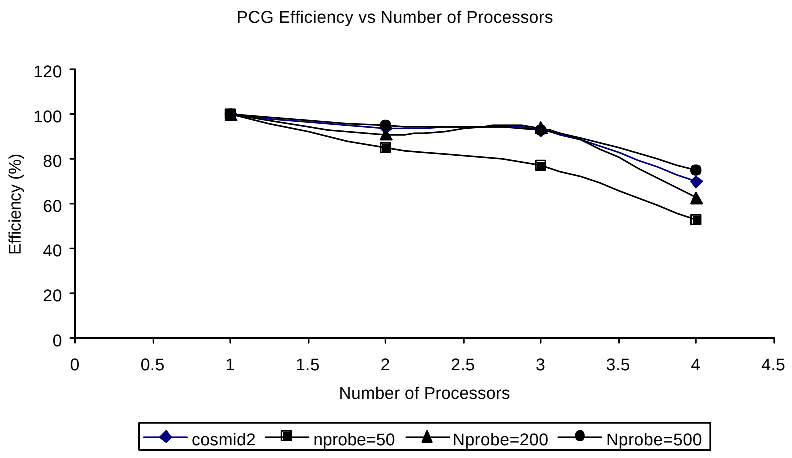

Figure 2.14 Efficiency versus number of processors used for parallel conjugate gradient descent procedure

The parallel algorithm for conjugate gradient descent procedure yields a good performance with efficiencies ranging from 53%-95. When only 2 or 3 processors are used, the efficiencies are in the range 77%-95% (Table 2.2, Figure 2.14). In all datasets, execution using 4 processors shows lower efficiency. The lowest efficiency comes from the synthetic dataset *nprobe* = 50. This low efficiency is due to the small dataset and its relatively high synchronization overhead. As expected, the general trend is that the speedup increases with respect to the number of processors used. For synthetic datasets *nprobe* = 50 and 200, the speedups begin to degrade when the number of processors increases to 3. Therefore for these datasets, the parallel conjugate gradient descent procedure with 3 processors runs the fastest whereas the serial algorithm is the slowest. The synthetic dataset *nprobe* = 500 shows continuous increase in the speedup when 4 processors are used. The speedups for the real dataset *cosmid2* derived from chromosome II of *Aspergillus nidulans*, which has 109 probes and 2046 clones, are approximately equal when 3 and 4 processors are used (2.79 for 4 processors versus 2.78 for 3 processors). As anticipated, given a fixed number of processors, the larger the dataset, the higher the speedup and the more efficient of the utilization of computing power (Figures 2.1, 2.2). Therefore, the synthetic dataset *nprobe* = 50 shows the lowest speedup for a given number of processors whereas *nprobe* = 500 almost always shows the highest speedup.

2.7.2 Performance for Parallel MLE Algorithm

Figure 2.15 shows the execution time of the parallel MLE-based physical mapping algorithm versus the number of processors used for a given dataset (*cosmid2* in this case).

No two of these executions have the same final objective function value. The execution time is significantly reduced when more processors are used.

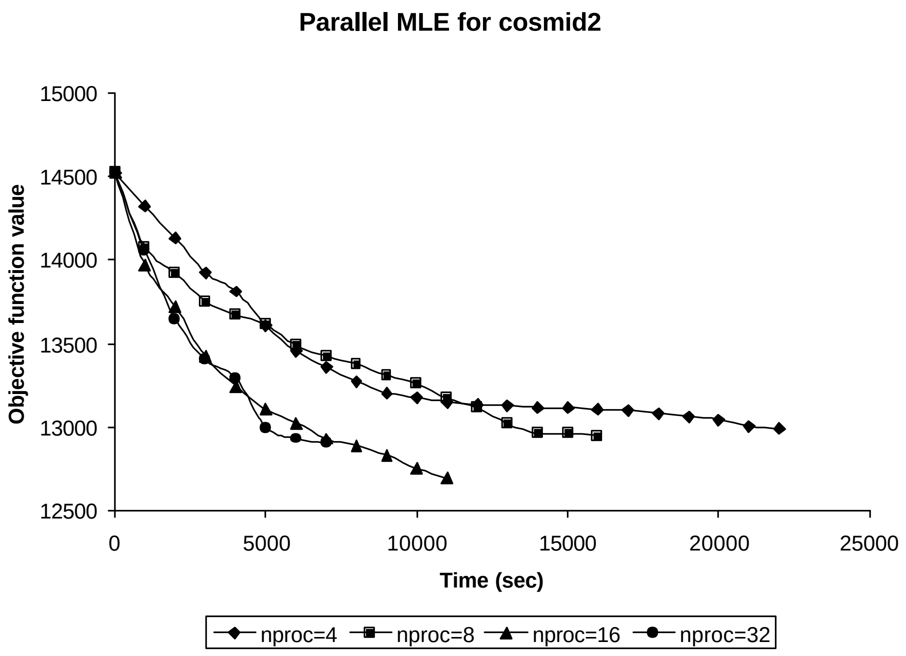

Figure 2.15 Execution of the parallel MLE-based physical mapping for *cosmid2*

Figures 2.16 and 2.17 respectively show the speedup and efficiency versus the number of processors used during the execution for different datasets. As for the parallel conjugate gradient descent procedure, the more the processors used, the higher the speedup (Figure 2.16). However, the increase in speedup with respect to the number of processors used is a slow and gradual process. This is illustrated by the efficiency values as shown in Figure 2.17. In both the synthetic dataset *nprobe* = 50 and the real dataset *cosmid2*, execution using 4 processors (one SMP machine) yields the highest efficiency, from 73% (*nprobe* = 50) to 110% (*cosmid2*). The efficiency for *cosmid2* is more than 100% due to the stochastic nature of parallel simulated annealing processes as mentioned above. The efficiency fluctuates when 8 and 16 processors are used. When 32 processors are used, the efficiency turns out to be the lowest.

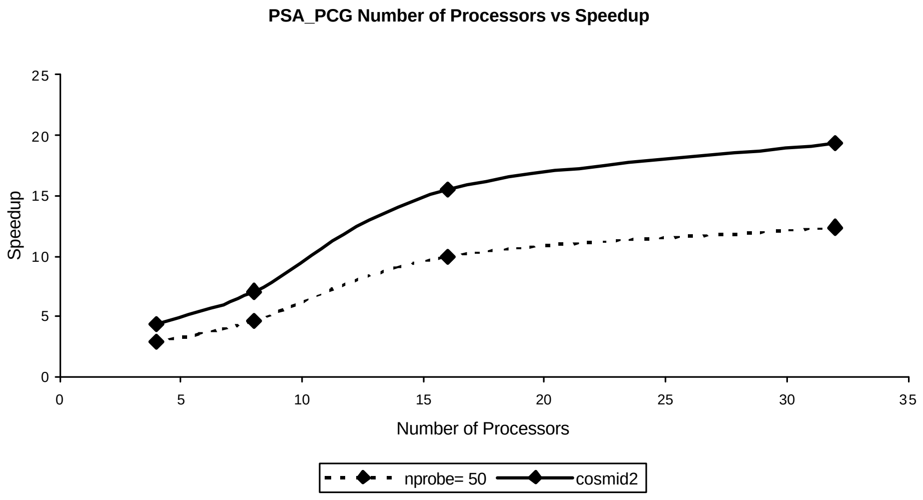

Figure 2.16 Speedup versus processor number for parallel MLE-based physical mapping algorithm

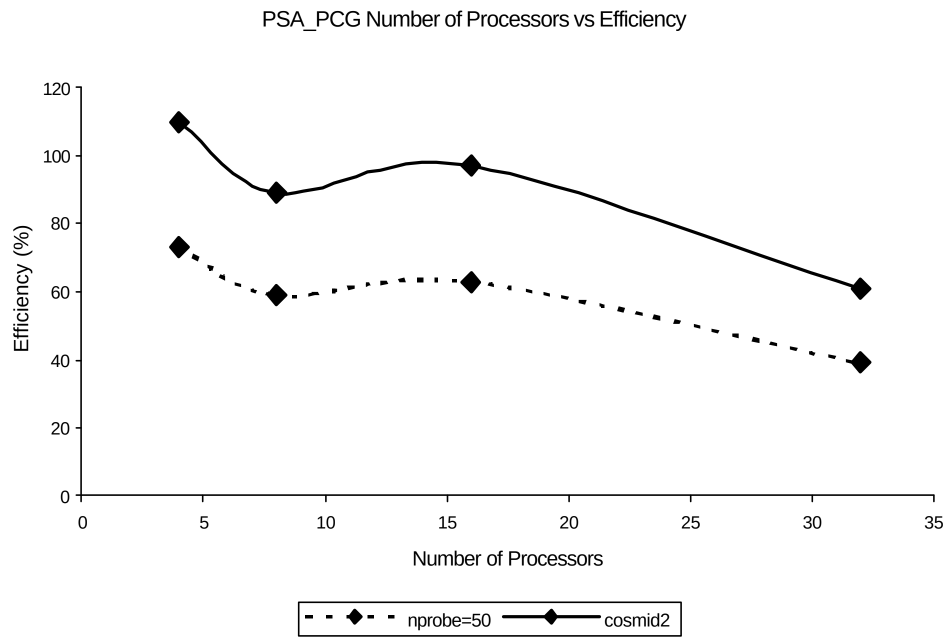

Figure 2.17 Efficiency versus processor number for parallel MLE-based physical mapping algorithm

Higher efficiency at lower number of processors is partially due to the communication protocol used in this project. It should be noted again that our parallel algorithms employ two levels of parallelization with different communication protocols. At the higher level, inter-process communication is achieved via message passing and the task of simulated annealing is evenly distributed among multiple slave processes. At the lower level, we have used shared memory multithreaded programming (usually the thread number equals the number of processors in the SMP machines, and in our case, 4 threads are usually used) to parallelize the conjugate gradient descent procedure. When only four processors are used, four threads in one process cooperate to compute the conjugate gradient descent procedure. In this case, simulated annealing is essentially not parallelized. Since threads share the same address space, communication among threads is more efficient than among processes. Therefore the performance of parallel MLE using only shared memory, here the 4-processor scenario, would theoretically be higher. With further increase in the number of processors used, the proportion of inter-process communication via message passing also becomes higher, and the efficiency would correspondingly gradually degrade.

To illustrate the superior performance of shared memory multithreaded programming to the inter-process communication via message passing, we have used 4 and 8 SMPs to spawn an equal number of processes for synthetic datasets *nprobe* = 50 and 100. Within each process, only one thread is used. Essentially this approach would parallelize the simulated annealing function but keep the conjugate gradient descent procedure serial. By doing so, we would eliminate the use of shared memory and ensure that all the overheads are due to inter-process communication. The result of test, as shown in Table 2.3, indicates that the speedups achieved using only inter-process communication are consistently lower compared to the shared memory multithreaded programming or a combination of shared memory multithreaded programming and inter-process communication via message passing. The degradation of the speedup due to inter-process communication via message passing is especially apparent for the dataset *nprobe* = 100, where the speedup using shared memory programming is 3.7 but only 1.7 when message passing is used, suggesting a much higher overhead of inter-process communication. This overhead caused by inter-process communication may also contribute to the lower efficiencies associated with the higher number of processors in our timing test for the parallel MLE-PILM algorithm (see Figure 2.17), where more inter-process communication would occur.

Table 2.3 Parallelization of MLE using shared memory versus inter-process communication via message passing

Dataset	Shared Memory 4 processors	Shared Memory + Message Passing 8 processors	Message Passing 4 processors	Message Passing 8 processors
nprobe=50	2.9	4.7	2.3	3.7
nprobe=100	3.7	4.2	1.7	3.2

The size of the dataset greatly affects the performance of the parallel MLE. It is anticipated that data parallelism for smaller datasets will incur more overheads due to more frequent inter-process communication. This is also evidenced by Figures 2.15 and 2.16 to certain degree. The real dataset *cosmid2* has 109 probes and shows consistently higher efficiency values than the synthetic dataset *nprobe* = 50. Execution using synthetic datasets *nprobe* = 100 and 200 show similar scenarios but their efficiencies are occasionally more than 100% due to the nature of the parallelization approach used in this project.

CHAPTER 3

PARALLEL COMPUTING FOR LARGE STEP MARKOV CHAINS ALGORITHM

3.1 Large Step Markov Chains Algorithm

The Large Step Markov Chains (LSMC) algorithm was originally proposed by Martin et al. (1991) for the Travelling Salesman Problem (TSP). This algorithm combines deterministic local search with stochastic optimization. The LSMC is an iterative procedure. During each iteration, a search for a new locally optimal solution is conducted. The new locally optimal solution is compared with the current solution that is also a locally optimal solution and is accepted based on probability *p* calculated using the Metropolis function. When this is done, a new intermediate solution is created and a new local exhaustive search is performed for the next new locally optimal solution. This process is repeated until a predefined criterion is satisfied.

Specifically, the LSMC algorithm comprises of the following steps:

1. *Kick*: Let $f(x) = f(x_1, x_2, ..., x_n)$ be an *n*-variable function that is to be minimized. Let x_i be the current solution which is also a locally optimal solution. In this step, a large *Kick* or non-local perturbation is applied to the current solution to yield a new intermediate solution x_j.
2. *Local exhaustive search*: Solution x_j is improved using a local exhaustive search technique to yield a locally optimal solution x_k.

3. Decide: If $f(x_k) < f(x_i)$, then x_k is accepted as the new solution. If $f(x_k) \geq f(x_i)$, then x_k is accepted as the new solution with probability *p*, which is computed using the Metropolis function $p = \exp\left(-\frac{f(x_k) - f(x_i)}{T_i}\right)$ at a given temperature T_i whereas x_i is retained with probability $(1 - p)$.

The LSMC algorithm was first introduced by Kota (2000) in the context of MLE-based physical map reconstruction. In this case, each probe resembles a city in the TSP and the ordering of the probes corresponds to the salesman's tour.

The *Kick* step is problem dependent and represents a non-local perturbation. Care should be taken that the new intermediate solution should not lead one back to a locally optimal solution that has been encountered before. In our project, a *double-bridge* change (Martin et al. 1991) is used as the *Kick* step. The *double-bridge* change is a non-local perturbation that involves a *4*-change. A perturbation is termed as a *k*-change if it removes *k* different links from a sequence (or tour) and reconnects them into a new legal sequence. According to Lin and Kernighan (1973), both *2*-opt and *3*-opt are sequential changes that keep the tour connected during the intermediate steps. The *4*-opt that consists of two *2*-changes is the first non-sequential change, where the first *2*-change disconnects the tour and the second *2*-change reconnects it, and thereby leads to a non-local intermediate step.

The exhaustive search for a locally optimal solution in this project uses the *2-opt* heuristic, which tests all possible *2*-changes to identify the best local solution. A heuristic is *n*-opt if it tests all possible *n*-changes during the local exhaustive search step. It is also possible to use a *3*-opt or a *4*-opt heuristic for local exhaustive search, but the execution time would be much longer since the *k*-opt heuristic is computationally intensive for *k>2*. In fact, the straightforward LSMC algorithm based on the *k*-opt local exhaustive search technique is even slower than the simulated annealing algorithm since the MLE objective function value needs to be computed and evaluated at each local search step. To overcome this problem, Kota (2000) modified the local exhaustive search procedure by evaluating the Hamming distance instead of the objective function value for each search step. The Hamming distance $d(P_i, P_j)$ between two probes P_i and P_j is defined as the measure of the dissimilarity, i.e., the number of unmatched digital signature bits, between the two probes. The optimal probe sequence resulting from the local exhaustive search should have the following property:

$$\min\left(D = \sum_{i=1}^{n-1} d\left(P_i, P_{i+1}\right) \right)$$

where n is the total number of probes, and D is referred to as the total linking distance of the probe ordering. This approach is reasonable since two probes that hybridize with the same set of clones should have similar hybridization signatures and their dissimilarity should be smaller, i.e., their physical separation on the physical maps should be minimal. Thus, the total inter-probe dissimilarities should also be minimal for an optimal solution. As a result of this modification, the local exhaustive search time is greatly reduced without compromising the accuracy of the solution. This leads to a variant of the straightforward LSMC that uses a combination of local exhaustive search based on the Hamming distance and stochastic search based on the MLE objective function as shown in Figure 3.1.

```
T = T_max;
Finished = false;
While (not Finished)
{
    for (count = 1;  count <= COUNT_LIMIT;  count = count + 1)
    {
      1   Phase one – Kick
          Make a double-bridge change to the current locally optimal ordering $P_i$
               to yield a new ordering $P_j$;
      2   Phase two – Local exhaustive search
          (a) Calculate the Hamming distance for $P_j$ called $D_j$;
               local_best_value = $D_j$;
               local_best_order = $P_j$;
          (b) Make a 2-change to $P_j$ to yield $P_k$;
              Evaluate the Hamming distance for $P_k$, called $D_k$;
              If ($D_k$ < local_best_value)
                   local_best_value = $D_k$;
                   local_best_order = $P_k$;
          (c) If all possible 2_change have been tested
                   GOTO Phase Three;
               Else
                    GOTO step 2(b);
```

```
        3   Phase Three – *Decide*
            If *local_best_order* causes the value of MLE function to decrease
                Accept *local_best_order*.
            Else
                Accept *local_best_order* with probability $p$ computed
                 using Metropolis function.
        }
        Update the temperature using annealing function $T = A(T)$;
}
```

Figure 3.1 LSMC algorithm using a combination of Hamming distance and MLE objective function

Compared to straightforward simulated annealing, LSMC takes the advantage of the deterministic local exhaustive search and, in most cases, leads to a better solution. Its superiority can be shown in Figure 3.2 for the real dataset *cosmid2*, where the number of iterations is 100 for a given annealing step. LSMC yields a better solution but takes only about 33% of the time taken by the simulated annealing algorithm. It should also be noted here that even with its significant computational advantage, the execution of LSMC still takes about 10 hours. If the problem size and the number of iterations are increased, the execution time would be even longer. Therefore, execution time is still an important issue for application of the LSMC to the MLE-based physical mapping algorithm.

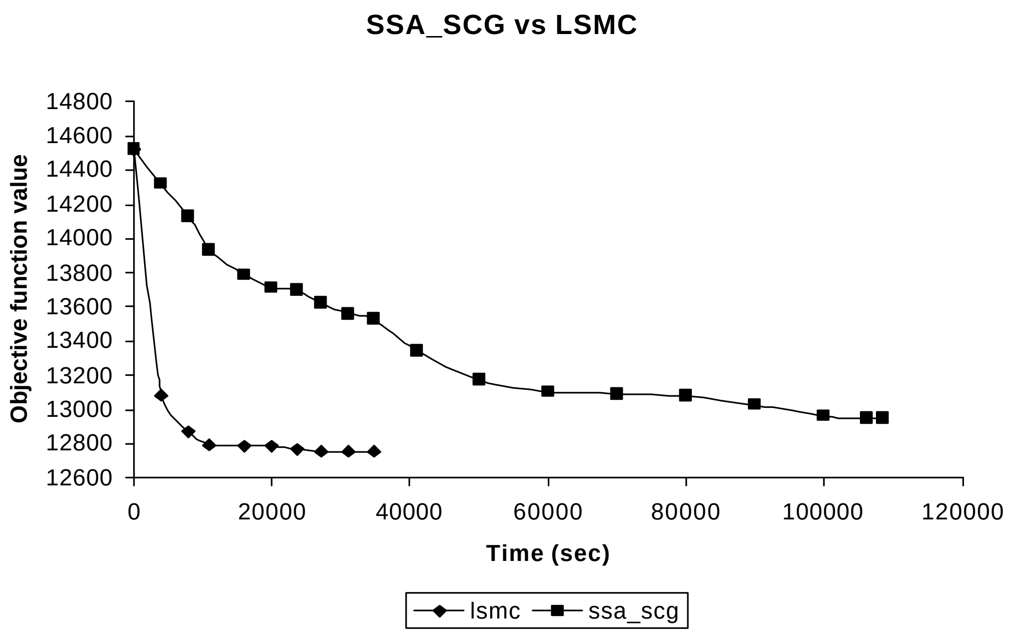

Figure 3.2 Timing of simple simulated annealing and LSMC for *cosmid2*

3.2 Parallelization of LSMC

Parallelization of the LSMC algorithm is realized through parallelizing three phases of the LSMC algorithm. They are

(1) Parallelization of the local exhaustive search procedure;

(2) Parallelization of the simulated annealing procedure;

(3) Parallelization of the conjugate gradient descent procedure.

The local exhaustive search procedure in LSMC attempts all possible *2*-changes to reach the locally optimal solution. This is realized by using two loops. The outer loop shifts a probe position at each iteration so that the last probe in the sequence becomes the first, and the first becomes the second, and so on. The inner loop conducts all possible *2*-

changes and the candidate solution with the current minimum Hamming distance (*best_distance*) is recorded as the *best_order*. After the outer loop finishes execution, the *best_order* and *best_distance* are returned, and comprises the locally optimal solution after a large *Kick* has been applied (Figure 3.3).

```
temp_order = current_order;
best_order = current_order;
best_distance = current_distance;
for (i = 1;  i <= nprobe;  i++)
{
    if (i = 1)
    {
        temp = current_order[n];
        for (i = nprobe;  i >= 2;  i--)
            current_order[i] = current_order[i-1];
            current_order[1] = temp;
     }
     for (n1 = 2;  n1< nprobe; n1++)
          for (n1 = 2;  n2 < nprobe; n2++)
          {
              (a) reverse probe sequences between n1 and n2 ;
              (b) copy the reversed sequence into temp_order;
              (c) calculate the Hamming distance temp_distance for temp_order;
              (d) if temp_distance < best_distance
                      best_distance = temp_distance;
                      best_order = temp_order;
          }
}
```

Figure 3.3 Algorithm for local exhaustive search of LSMC

To parallelize the local exhaustive search procedure, we have used the Master/Slave model. The outer loop with *nprobe* iterations is evenly divided among *N* slaves. Therefore each slave carries out $\frac{nprobe}{N}$ iterations, and calculates its *local_best_distance* and *local_best_order*. When slaves finish the execution, they report their *local-best_distances* and *local_best_orders* to the master which would identifies the *best_distance* and its corresponding probe order as the optimal solution after a *Kick* operation has been applied.

The parallel local exhaustive search algorithm with the master and slave processes is shown in Figures 3.4 and 3.5.

```
Spawn N slave_threads for local exhaustive search;
threadAlive = N;    //threadAlive is a global variable
if (threadAlive > 0)
```

```
    wait;
for (i = 1;  i <= $N$;  i++)
{
    if ($global\_best\_distance$ > $local\_best\_distance[i]$)
        $global\_best\_distance$ = $local\_best\_distance[i]$;
        $minIndex$ = $i$;
}
$global\_best\_order$ = $local\_best\_order[minIndex]$.
```

Figure 3.4 Master process for parallel local exhaustive search algorithm

```
$index$ = $ThreadIndex$;
$range$ = $nprobe/N$;
$base$ = $index*range$ + 1;
Shift $current\_order$ for $range*index$ positions;
$temp\_order$ = $current\_order$;
$best\_order$ = $current\_order$;
$best\_distance$ = $current\_distance$;
for (i = $base$;  i <= $range$;  i++)
{
    if (i = 1)
    {
            temp = current_order[nprobe];
            for (j = nprobe;  j >= 2;  j--)
                current_order[j] = current_order[j-1];
            current_order[1] = temp;
        }
        for (n1 = 2;  n1< nprobe;  n1++)
            for(n1 = 2;  n2 < nprobe;  n2++)
            {
                (a) reverse probe sequences between n1 and n2;
                (b) copy the reversed sequence into temp_order;
                (c) calculate the Hamming distance temp_distance for temp_order;
                (d) if temp_distance < best_distance
                        best_distance = temp_distance;
                        best_order = temp_order;
            }
}
local_best_distance[index] = best_distance;
local_best_order[index] = best_order;
threadAlive --;
Exit and destroy thread.
```

Figure 3.5 Slave process for parallel local exhaustive search algorithm

The master and slave processes for the parallel LSMC algorithms are shown in Figures 3.6 to 3.9.

```
$T$ = $T\_max$;
$Finished$ = $false$;
While (not $Finished$)
{
    for ($count$ = 1;  $count$ <= $COUNT\_LIMIT$; $count$ = $count$+1)
    {
        1    Phase one – Kick
             Make a double-bridge change to the current locally optimal ordering $P_i$
             to yield an intermediate solution $P_j$;
        2    Phase two – Local Exhaustive Search
             Invoke the master local exhaustive algorithm, which would spawn slaves to carry
             out the search for a locally optimal solution $P_k$;
        3    Phase three - Evaluate
             (a) Invoke the master conjugate gradient descent procedure. This would spawn
                 slave threads for the procedure and compute the optimal inter-probe spacings
                 for the locally optimal solution $P_k$;
             (b) Compute the objective function value for $P_k$;
             (c) Compute $f\_delta$, the change in the value of the objective function;
```

```
  4  Phase three - *Decide*
       If (*f_delta* < *0*)
          Accept the new locally optimal solution $P_k$.
       Else
          Accept the new locally optimal solution $P_{k;}$ with the probability $p$ computed
           using the Metropolis function.
  }
  Update the temperature using the annealing function $T = A(T)$;
}
(a) Receive locally optimal objective function values from slaves;
(b) Compare locally optimal objective function values and identify the global minimal
    values and its process identifier (*mindId*);
(c) Broadcast *minId* to all the processes;
(d) Receive probe ordering and inter-probe spacings from the process *minId*.
```

Figure 3.6 Master process of the NILM_LSMC algorithm using MPI and Pthreads

```
T = T_max;
Finished = false;
While (not Finished)
{
    for (count = 1;  count <= COUNT_LIMIT;  count = count+1)
    {
        1   Phase one – Kick
              Make a double-bridge change to the current locally optimal ordering $P_i$
              to yield an intermediate solution $P_j$;
        2   Phase two – Local Exhaustive Search
            Invoke the master local exhaustive algorithm, which would spawn the slaves to
            carry out the search for a locally optimal solution $P_{k;}$
        3   Phase two – Evaluate
            (a) Invoke the master conjugate gradient descent procedure. This would spawn
                  slave threads for the procedure and compute the optimal inter-probe
                  spacings for the new locally optimal solution $P_k$;
            (b) Compute the objective function value for $P_{k;}$
            (c) Compute f_delta, the change in the value of the objective function;
        4   Phase three  - Decide
              If (f_delta < 0)
                  Accept the new locally optimal solution $P_{k.}$
              Else
                  Accept the new locally optimal solution $P_{k;}$ with the probability p computed
                  using the Metropolis function.
      }
      Update the temperature using the annealing function $T = A(T);$
}
```

```
(a) Send locally optimal function value to the master;
(b) Receive from master the identifier MinId ;
(c) If (MinId = myId)
        Send locally optimal probe ordering to the master;
        Send optimal inter-probe spacings for the locally optimal probe ordering to the
          master.
```

Figure 3.7 Slave process of the NILM_MLE algorithm using MPI and Pthreads

```
T = T_max;
Finished = false;
While (not Finished)
{
   for (count = 1;  count <= COUNT_LIMIT;  count = count+1)
   {
     1   Phase one – Kick
           Make a double-bridge change to the current locally optimal ordering $P_i$
           to yield an intermediate solution $P_j$;
     2   Phase two – Local Exhaustive Search
           Invoke master local exhaustive search algorithm, which would spawn the slaves
             to carry out the search for a locally optimal solution $P_{k;}$
     3   Phase three - Evaluate
```

```
            (a) Invoke master conjugate gradient descent procedure. This would spawn slave
                threads for the procedure and compute the optimal inter-probe spacings for
                the new locally optimal solution;
            (b) Compute the objective function value for the new locally optimal solution;
            (c) Compute f_delta, the change in the value of the objective function;
        4   Phase three - Decide
            If (f_delta < 0)
                Accept the new locally optimal solution $P_k$.
            Else
                Accept the new locally optimal solution $P_k$ with the probability p computed
                using the Metropolis function.
    }
    (a) Receive locally optimal objective function values and process Ids from slaves;
    (b) Compare locally optimal objective function values from slaves and identify the
        global optimal objective function value and its associated process Id (minId);
    (c) Broadcast minId to all the processes;
    (d) Receive locally optimal probe ordering and inter-probe spacings from process
        minId;
    (e) Update the temperature using the annealing function $T = A(T)$;
}
```

Figure 3.8 Master process of the PILM_LSMC algorithm using MPI and Pthreads

```
T = T_max;
Finished = false;
While (not Finished)
{
   for (count = 1;  count <= COUNT_LIMIT;  count = count+1)
   {
      1   Phase one – Perturb
              Make a double-bridge change to the current locally optimal ordering $P_i$
              to yield an intermediate solution $P_j$;
      2   Phase two – Local Exhaustive Search
              Invoke master local exhaustive search algorithm, which would spawn the slaves
              to carry out the search for a locally optimal solution $P_{k;}$
      3   Phase three - Evaluate
              (a) Invoke master conjugate gradient descent procedure. This would spawn slave
                  threads for the procedure and compute the optimal inter-probe spacings for
                  the new locally optimal solution $P_{k;}$;
              (b) Compute the objective function value for the new probe order;
              (c) Compute f_delta, the change in the value of the objective function;
      4   Phase three  - Decide
              If (f_delta < 0)
                  Accept the new locally optimal solution.
              Else
                  Accept the new locally optimal solution with the probability p computed
```

```
                using the Metropolis function.
        }
    (a) Send locally optimal objective function value and process Id to the master;
    (b) Receive *minId* from the master;
        If (*myId* = *minId*)
            Broadcast locally optimal probe ordering and inter-probe spacings to all
             the processes;
    (c) Update the temperature using the annealing function $T = A(T)$;
}
```

Figure 3.9 Slave process for the PILM_MLE algorithm using MPI and Pthreads

In this project, we have exploited the computing power of a network of shared-memory multiprocessor computers for implementation of the LSMC algorithm. The master and slaves for local exhaustive search procedure are implemented using POSIX threads. The implementation of parallel simulated annealing and parallel conjugate gradient descent search is the same as in Chapter 2.

3.3 Experimental Results

3.3.1 Parallel Local Exhaustive Search

The parallel algorithm for the local exhaustive search using shared memory has yielded good performance as shown in Figures 3.10 and 3.11. In larger synthetic datasets with *nprobe* = 100 and 200, the speedup increases linearly with respect to the number of processors used. The synthetic dataset *nprobe* = 50 shows lower speedups. The good performance of the parallel local exhaustive search algorithm can also be demonstrated by the efficiency values shown in Figure 3.11 where efficiencies for *nprobe* = 100 and 200 are all above 85%. The lowest efficiency comes from the synthetic dataset *nprobe* = 50, where the efficiency is only about 53% when 4 processors are used. This is expected since *nprobe* = 50 is the smallest dataset for the test and its associated communication overhead will be higher compared to the larger datasets. It can be seen that for larger datasets with *nprobe* = 100 and 200, parallelization using 2 processors yields the best efficiencies (93% and 96% respectively) for the local exhaustive search procedure. For the smaller dataset with *nprobe* = 50, efficiency is the highest when 3 processors are used.

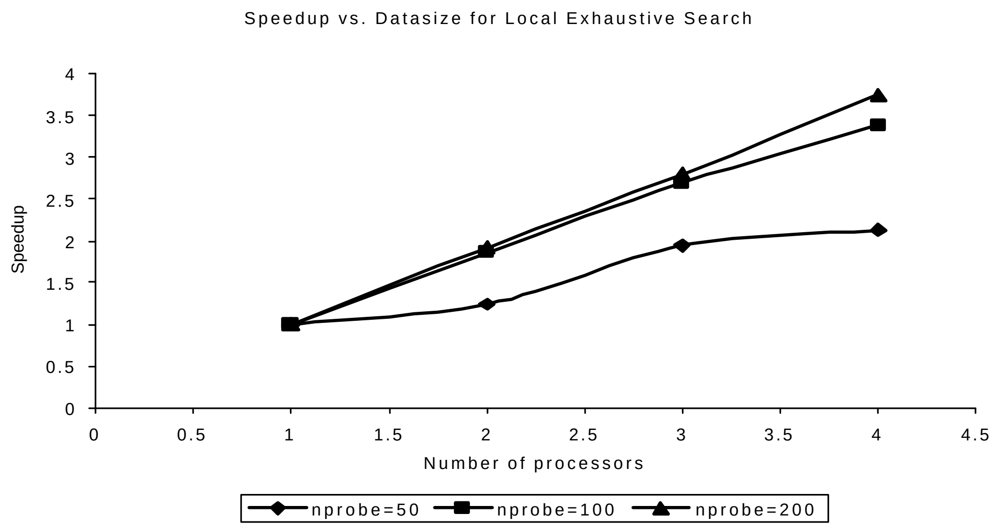

Figure 3.10 Speedup versus number of processors for the parallel local exhaustive search algorithm

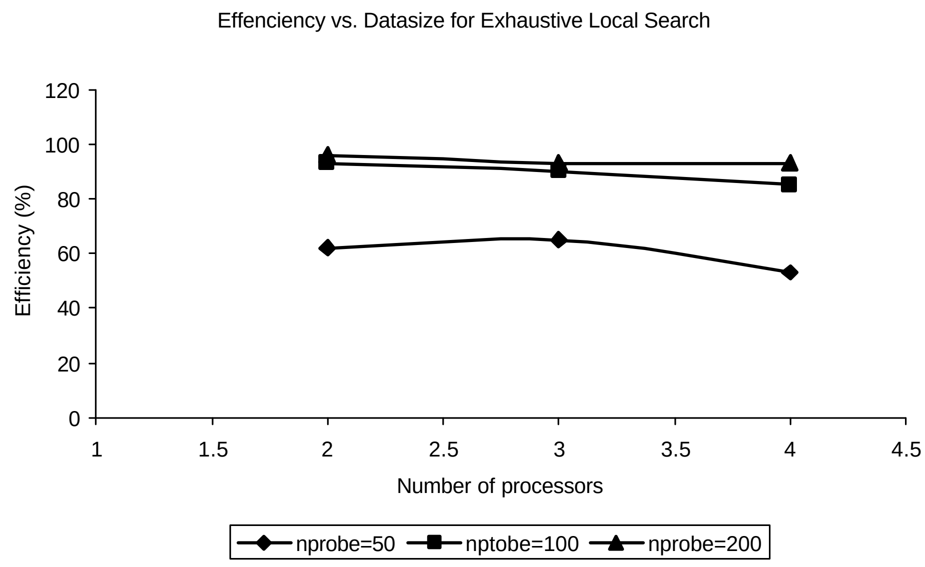

Figure 3.11 Efficiency versus number of processors for the parallel local exhaustive search algorithm

3.3.2 Performance of the Parallel LSMC Algorithm

Timing tests using both synthetic datasets and real datasets show the increase in the speedup with respect to the number of processors used. A higher number of processors often leads to higher speedups for a given dataset. Figure 3.12 shows the speedups of different datasets. With no exception, speedup is the highest when 32 processors are used and the lowest for 4 processors. For example, for the synthetic dataset of *nprobe* = 100, the speedup changes from 3.22 to 5.33, 13.65 and 21.57 respectively with the processor number increasing from 4 to 8, 16, and 32. It is foreseeable that with the further increase in the number of processors, the overhead of inter-process communication will also become greater and the speedup will eventually reach its maximum and thereafter decrease. In our timing tests, since the cluster consists of only eight SMP machines (with

4 processors per SMP), we were unable to detect this phenomenon. As expected, the speedup tends to increase for large datasets (Figure 3.11). In all cases, the synthetic dataset *nprobe* = 200 yielded the highest speedup for a given number of processors. The real dataset *cosmid2*, although with a probe number of 109, has always yielded the lowest speedups.

Increase in speedup is not always proportional to the increase in the number of processors. For most of the synthetic datasets, there seemed to be a steady and rapid increase in the speedup with respect to the number of processors. In the case of real dataset *cosmid2*, the increase in the speedup is insignificant compared to the increase in the number of processors. The synthetic dataset *nprobe* = 50 yields slightly better results than the real dataset *cosmid2*, but is significantly lower than *nprobe* = 100 and *nprobe* = 200 in terms of speedup.

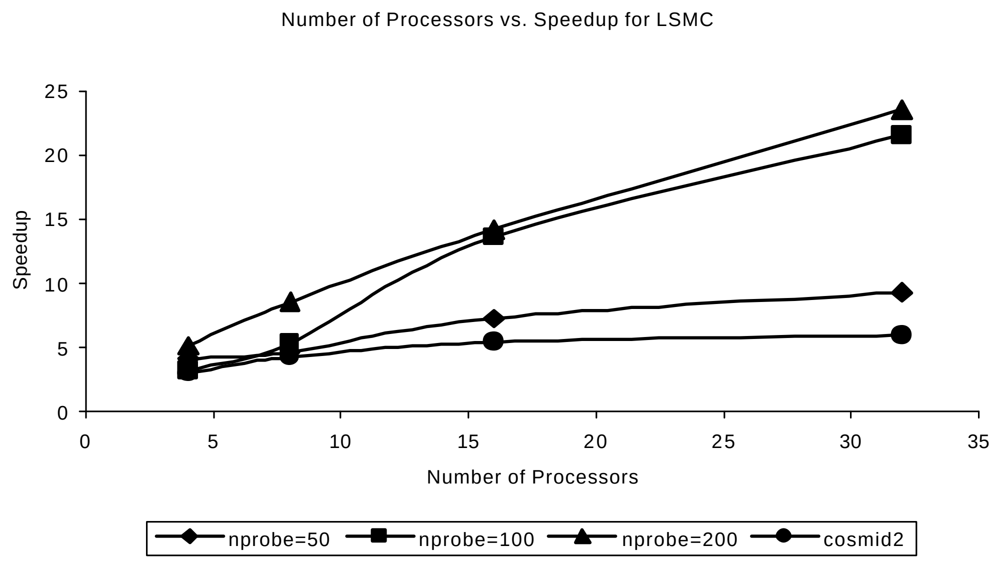

Figure 3.11 Performance of parallel LSMC algorithms

CHAPTER 4

EVOLUTIONARY ALGORITHMS FOR PHYSICAL MAPPING PROBLEMS

The evolutionary algorithm is an adaptive problem solving procedure modeled on the natural evolutionary process. According to Darwin (1859), an organism has the capability of increasing its numbers geometrically. Since the natural resources needed for the organism are limited, there is a struggle for existence and the fittest individual will have the best chance to survive and reproduce. The underlying basis for natural selection is genetic variations arising from mutation and recombination (i.e., crossover). The basic evolutionary unit is the population that consists of a number of individuals.

Evolutionary algorithms use heuristic search/optimization techniques to obtain the best possible solution in a vast solution space using a population or subset of potential solutions. Given certain information about the problem domain, evolutionary algorithms focus on a subset of potential solutions that would eventually converge to a globally optimal solution (Bhandarkar et al. 1994). Evolutionary algorithms are robust in that they are relatively unaffected by the presence of spurious locally optimal solutions in their pursuit of a globally optimal solution.

A typical evolutionary algorithm consists of the following components (Michalewicz 1992; Bhandarkar et al. 1994):

a. A population of candidate solutions, which are biologically referred to as chromosomes. The size of the population should be sufficiently large to maintain enough genetic variations, which would essentially determine the effectiveness of evolution.

b. Genetic operators, which typically consist of crossover and mutation. Genetic operators are mechanisms to introduce genetic variations. Crossover creates new chromosomes through the recombination of two parents during reproduction. Mutation is a random change in a single chromosome designed to introduce a novel offspring.

c. Objective function and associated fitness measurement. The objective function measures the performance or fitness of a given chromosome. The fitness value of a chromosome determines its probability of survival into and reproduction for the next generation.

d. Selection. This is the process that determines which chromosomes will survive into the next generation. The higher the fitness of a chromosome, the more likely it is to survive and reproduce. Selection leads to a new population consisting of better offspring but usually also reduces the genetic variations in the population.

An evolutionary algorithm is implemented as an iterative procedure. To search for the optimal solution of a given problem, the evolutionary algorithm starts from a subset of initial solutions (chromosomes) that evolve into different but better ones over successive generations (iterations). This process is repeated until a certain stopping criterion is met. The stopping criterion is a heuristic since, strictly speaking, the evolutionary process is endless barring complete extinction.

A typical evolutionary algorithm is described in Figure 4.1.

```
procedure EA;
{
    $t = 0$;
    Initialize population $P(t)$;
    Evaluate the fitness of individuals of $P(t)$;
    while (1)
    {
        $t = t+1$;
        Select parents for reproduction;
        Apply genetic operator(s) to create offspring;
        Evaluate and select individuals for survival into the next generation;
        Create the new generation;
        Replace current population by the new generation;
    }
    Select the best individual from the population as the optimal solution to the problem.
}
```

Figure 4.1 A typical evolutionary algorithm

Varieties of the evolutionary algorithm include evolutionary programming, evolutionary strategies and genetic algorithms. In this project, we have used an evolutionary programming/LSMC hybrid and two slightly different genetic algorithms to obtain a solution to the MLE-based physical mapping problem.

4.1 Evolutionary Programming/LSMC Hybrid for Reconstructing Physical Maps of Chromosomes

The evolutionary programming technique was first proposed by Fogel et al. in 1966 and has been used in a variety of areas. A typical evolutionary programming uses mutation as the sole genetic operator to generate offspring. Unlike the simulated annealing algorithm which starts with a single candidate solution, the evolutionary programming technique comprises of a population of initial candidate solutions, each going through a number of generations before reaching a locally optimal solution. The final optimal solution is chosen from an ensemble of locally optimal solutions. Since a larger solution space is sampled, the final solution is often better than the one obtained from the simulated annealing algorithm. However, since a population of solutions has to be searched, the execution time is also often longer.

A major advantage of using evolutionary programming is its ease of parallelization. Since simulated annealing starts with one solution, parallelization of the algorithm is often difficult. Efforts to parallelize the simulated annealing algorithm often decompose the Metropolis function or the annealing step, but the results often vary depending on the underlying problem. On the other hand, since evolutionary programming iterates through a population of solutions, it is inherently parallel. A workload comprising of a population of solutions can be easily partitioned amongst processors.

To combine the strengths of simulated annealing and evolutionary programming approaches, we have proposed a hybrid of evolutionary programming and LSMC as depicted in Figure 4.2 for the physical mapping problem. In this algorithm, a population of chromosomes is first created by *double-bridge* operations followed by local exhaustive searches. Since *double-bridge* is a non-local perturbation technique, its application will yield a set of distinct solution states. The local exhaustive search, on the other hand, will ensure that each chromosome in the initial population represents a locally optimal solution to the problem. Each chromosome in the population then starts at the initial temperature and an independent seed and goes through the LSMC process. Essentially, the evolutionary/LSMC hybrid algorithm comprises an ensemble of independent MLE processes where the final solution to the problem is chosen after the execution of the algorithm.

```
Create population $P$ of $N$ solutions using double-bridge and local exhaustive search;
$T = T\_max$;
while (1)
{
   for (i=1; i<=$N$; i++)
   {
      for (j=1; j<=$M$; j++)
      {
            (a)  Perform the double-bridge perturbation on P[i] to create an offspring;
            (b)  Perform local exhaustive search starting from the offspring
                 and identify the local optimal solution P[i]*;
            (c)  Perform conjugate gradient descent and compute the local minimum
                 objective function value at P[i]*;
            (d)  Compute f_delta, the change in the value of the objective function between
                 P[i] and P[i]*;
            (e)  If (f_delta < 0)
                      Accept the new solution P[i]*.
                 Else
                      Accept the new solution P[i]* with the probability p computed using
                      the Metropolis function.
        }
    }
    Update the temperature using annealing function T = A(T);
}
Choose the best chromosome from the population.
```

Figure 4.2 A hybrid evolutionary programming/LSMC algorithm for MLE-based physical mapping

4.2 Genetic Algorithms for Physical Mapping of Chromosomes

Genetic algorithms differ from evolutionary programming by introducing the recombination (crossover) operator. During the recombination, two parental chromosomes are selected. Child chromosomes are generated by exchanges of parental chromosomal segments and, as a result, bring more genetic variations into the population. However, since the crossover operator creates genetic variations only by shuffling the parental chromosomes, these genetic variations will gradually be depleted after a certain number of generations. To introduce novel variations into the population, mutations should be conducted occasionally. Therefore, chromosomal recombination should be accompanied by occasional mutations. Since most mutations have deleterious effects, mutation rates should usually be kept relatively low. As in evolutionary programming, selection is strictly based on the fitness of individual chromosomes. Newly created individuals with higher fitness are more likely to replace the less fit parents and survive into the next generation.

Many crossover operators have been proposed. These include partially matched crossover (PMX) (Goldberg 1989; Michalewicz 1994; Falkenauer 1998), cycle crossover (CX) (Goldberg 1989; Michalewicz 1994), order crossover (OX) (Goldberg 1989; Michalewicz 1994), matrix crossover (MX) (Michalewicz 1994) etc. However, since offspring created by these crossover operators are mostly the random assortments of the parental chromosomes, they are often less fit than their parents. As a result, most of the newly created offspring will be deleted from the population even though a significant amount of time has been spent creating and evaluating them. To circumvent this problem, a heuristic crossover operator was proposed by Jog et al. (1989) for the TSP. Specifically, the heuristic crossover for the TSP does the following:

(a) Randomly select two chromosomes.

(b) Choose a start node from one of the selected chromosomes for crossover.

(c) Compare two edges leaving from the start node between two parents and choose the shorter edge; if the shorter edge leads to an illegal sequence, choose the other edge; if both edges introduce illegal sequences, choose an edge from the remaining nodes that has the shortest distance from the start node.

(d) Choose the new node as the start node and repeat step *c* until a complete sequence is generated.

In our project, two parents are chosen randomly or using the roulette wheel selection procedure (Goldberg 1989) followed by application of the heuristic crossover operator with a slight modification. During the selection of a start node, if that node is close to the end of the sequence, in many cases, recombination will not be effective if two parents are similar. For example, if the start node is selected at a point after which sequences of both parental chromosomes are the same, no actual exchange of parental chromosomal segments will occur even though the heuristic crossover operator has been applied. To avoid this scenario, we start from the beginning of the sequence whenever a node close to the sequence end is selected as the start point. This often leads to a much improved child in a single heuristic crossover operation. Mutation in this project is implemented using the non-local *double-bridge* operation or the non-local *double-bridge* operation followed by a local exhaustive search. The mutation rate is set dynamically based on the genetic variations retained in the population. During the early stages of the annealing process, since genetic variations in the population are relatively high, application of the heuristic crossover usually creates better offspring. Therefore the mutation rate is kept relatively low. During the later stages of the annealing process, when the genetic variations in the population are gradually reduced due to selection, newly created offspring will tend to have the same or even lower fitness than their parents. As a result, the mutation rate will be allowed to increase correspondingly so that more variations could be introduced into the population.

Two slightly different genetic algorithms have been used in this project as shown in Figures 4.3 and 4.4. In the first genetic algorithm (Figure 4.3), parents are chosen randomly to create offspring and selection is strictly based on a deterministic search. During each iteration, either mutation or heuristic crossover will be applied. If the heuristic crossover is applied, the parent with lower fitness will be replaced by the newly created child. A local exhaustive search is conducted on each individual in the population and the local optimal solution yielded by the search will be accepted if it has a lower objective function value. The second genetic algorithm uses the fitness-based roulette wheel procedure to choose parents for reproduction and incorporates the simulated annealing process. During each iteration, heuristic crossover operation will be conducted to create the offspring which is subject to a local exhaustive search. Mutation is conducted on the offspring based on certain probability. If the offspring x_j has a lower objective function value than the less fit parent, it will replace that parent. Otherwise, the offspring is accepted with the Boltzmann logistic probability, which is computed by $p = \dfrac{1}{1 + e^{(E_i - E_j)/T}}$.

```
Create the initial population P of N chromosomes using double-bridge followed by local
exhaustive search;
while (1)
{
   for (i=1; i<=N; i++)
   {
          (a) If ( drand()<prob )
                    Perform mutation using the double-bridge perturbation on P[i];
              Else
                   Select two parents randomly and apply the heuristic crossover operator to
                   create the offspring which will replace the less fit parent;
          (b) Perform local exhaustive search from P[i] and identify the locally optimal
              solution P[i]*;
          (c) Perform the conjugate gradient descent procedure and compute the objective
              function value for P[i]*;
          (d) Compute f_delta, the change in the value of the objective function between
              P[i] and P[i]*;
          (e) If (f_delta < 0)
                  Accept the new solution P[i]*.
   }
   Update prob;
}
```

Figure 4.3 Genetic algorithm for reconstructing physical maps using deterministic search

```
Create the initial population $P$ of $N$ chromosomes using double-bridge followed by local
exhaustive search;
$T = T\_max$;
while (1)
{
  for (i=1; i<=$N$; i++)
  {
      (a) Select two parents using the roulette wheel procedure;
      (b) Apply heuristic crossover operator on the selected parents to create offspring
          $S$;
      (c) Perform local exhaustive search from $S$ and identify the locally optimal
          solution $S*$ ;
      (d) Perform conjugate gradient descent procedure and compute the objective
          function value at $S*$;
      (e) If (drand()<prob)
              Apply mutation operation on $S*$;
      (f) Compute f_delta, the change in the value of the objective function between
          the less fit parent and $S*$;
      (g) Retain the less fit parent with the probability $p$ given by the Boltzmann
          function.
  }
  Update prob;
```

```
    Update the temperature $T = A(T)$;
}
```

Figure 4.4 Genetic algorithm for reconstructing physical maps using simulated annealing

4.3 Parallelization of the Evolutionary Programming/LSMC Hybrid Algorithm

Parallelization of the evolutionary programming/LSMC hybrid takes advantage of the inherent ease of parallelization of an evolutionary programming algorithm. In our project, two levels of parallelization are again used. At the higher level, the population of solutions is partitioned evenly into subpopulations among the SMP machines. Each process iterates through the candidate solutions in the subpopulation and identifies the locally optimal solution on a single SMP. The globally optimal solution is chosen from the locally optimal solutions. At the lower level, conjugate gradient descent and local exhaustive search procedures are performed on a single SMP using shared-memory multithreaded programming as in the case of the parallel simulated annealing and LSMC algorithms (see Chapters 2 and 3).

The parallel evolutionary programming/LSMC hybrid algorithm is shown in Figure 4.5.

```
Partition Population $P$ of $N$ chromosomes among $x$ processes (one process per SMP) such
that each process deals with a subpopulation of $S = N/x$ individuals;
$T = T\_max$;
while (1)
```

```
{
    for (i=1; i<=S; i++)
    {
        for (j=1; j<=M; j++)
        {
            (a) Perform a double-bridge perturbation on P[i] to create an offspring;
            (b) Invoke master local exhaustive search procedure on the offspring.
                This would spawn slave local exhaustive search threads in the process and
                identify the locally optimal solution P[i]*;
            (c) Invoke master conjugate gradient descent procedure. This would spawn slave
                conjugate gradient descent threads in the process and also compute the
                objective function value given by P[i]*;
            (d) Compute f_delta, the change in the value of the objective function between
                P[i] and P[i]*;
            (e) If (f_delta < 0)
                    Accept P[i]* and replace P[i].
                Else
                    Accept the P[i]* with the probability p computed
                    using the Metropolis function.
        }
    }
    Update the temperature using annealing function T = A(T);
}
```

For the master:

(a) Receive *process_Id* and *local_best* from all the slaves;

(b) Identify the minimum among the *local_best* and its associated *process_Id*, say *minId*;

(c) Broadcast *minId* to all the slaves;

(d) Receive the solution from the *minId*.

For slaves:

(a) Identify the individual with the lowest objective function value *local_best* in the subpopulation and send that value and *process_Id* to the master;

(b) Receive *minId* from the master;

(c) If (*minId* = *myId*)

    Send the individual with the lowest objective function value to the master.

Figure 4.5 Parallelization of the evolutionary programming/LSMC hybrid algorithm for physical mapping of chromosomes

4.4 Parallelization of the Genetic Algorithms

Parallelization of the genetic algorithms in our projects also partitions the population into subpopulations among SMPs. Each subpopulation searches for a locally optimal solution on a SMP. Under the PILM model, slaves will send their subpopulations at the end of each annealing step to the master which will reshuffle these subpopulations and redistribute them to slaves again for the next annealing step. This process is similar to the gene flow among natural populations such that the variations among populations will be reduced but those within the population will increase. In our case, more genetic variations will be introduced into the subpopulation at each annealing step. When slaves finish the annealing process, they send their locally optimal solutions to the master which would identify the globally optimal solution to the problem. Under the NILM model, no gene flow process is required at each annealing step. Slaves only need to send their locally optimal solutions to the master at the end of the whole annealing process.

4.5 Experimental Results

The evolutionary programming/LSMC hybrid was tested on the same cluster of SMP machines in the Department of Computer Science at the University of Georgia using the synthetic dataset *nprobe* = 50. Tests for larger datasets were not conducted due to the significant amount of computing time required. The size of the population was limited to 10 individuals. The genetic algorithms were tested in the same computational environment using the synthetic datasets with *nprobe* = 50, 100, 200, and the real datasets *cosmid2* and *cosmid3* as the parallel simulated annealing and LSMC algorithms.

4.5.1 Evolutionary Programming/LSMC Hybrid

The execution of the serial evolutionary programming/LSMC hybrid algorithm for the synthetic dataset *nprobe* = 50 takes 76863 seconds and yields an objective function value of 1601.973232. Compared to the serial LSMC algorithm that runs about 10746 seconds using the same dataset, the convergence time for this hybrid algorithm is about 7 times longer. Considering the fact that the population consists of 10 candidate solutions, the convergence time for the hybrid algorithm is reasonable. However, the final objective function value obtained from the hybrid algorithm is lower than that from the LSMC (1601.973232 versus 1624.094166). This would suggest that sampling from an ensemble or a population of solutions has an inherent advantage in obtaining a better solution to the problem.

Parallel evolutionary programming/LSMC hybrid algorithm has been tested for the synthetic dataset *nprobe* = 50 using 4 SMPs (16 processors). When 10 chromosomes are limited in the population, the execution takes about 9501 seconds and yields an objective function value 1621.156043, which is slightly better than the value of serial LSMC algorithm for the same dataset and in a shorter execution time. When the population size is increased to *N* = 20, the final objective function value is 1595.202606, which is even better than that from the serial evolutionary programming/LSMC hybrid algorithm, but the execution time is also correspondingly longer (26677 seconds). These tests indicate that the parallel evolutionary programming/LSMC hybrid algorithm has the intrinsic merit to yield better results in a shorter time scale if enough care is taken to select a proper population size for the execution.

4.5.2 Genetic Algorithms

Both genetic algorithms have yielded good results in terms of the final objective function value and the execution time. The execution time and final objective values using different datasets for the deterministic search-based genetic algorithm are shown in Table 4.1. The results of the simulated annealing-based genetic algorithm are similar and will not be shown here.

Table 4.1 Timing comparison between the serial deterministic search-based genetic algorithm and the serial LSMC algorithm

Data	GA Time	GA Value	LSMC Time	LSMC Value
nprobe=50	3714	1549.81	10746	1624.09
nprobe=100	6544	4274.29	7459	4297.50
nprobe=200	25035	11200.49	105893	11515.13
cosmid2	25469	12789.89	34704	12757.55
cosmid3	21542	12476.97	30138	12501.88

For the same dataset, the genetic algorithm almost always has a better performance than the LSMC (Table 4.1). The only exception is the real dataset *cosmid2*, where the genetic algorithm yields a higher objective function value (12789.89 versus 12757.55) but in a shorter execution time (25469 seconds versus 34704 seconds). The superiority of the genetic algorithm is especially apparent for the synthetic dataset *nprobe* = 50, where the genetic algorithm takes a little more than 1/3 of the execution time of LSMC (3714 seconds versus 10746 seconds), but yields a much improved solution to the problem (1549.81 versus 1624.09). This result is impressive considering the fact that the genetic algorithm starts with a population of 10 chromosomes and that it is also very hard to decrease the objective function value further when it reaches a certain point. Compared to the simple simulated annealing approach (Chapter 2 and Figure 4.6), the genetic algorithm takes about 25% of the execution time (3714 seconds versus 15068 seconds) for the synthetic dataset *nprobe* = 50 to yield a significantly lower objective function value (1549.81 versus 1665.37). Therefore, genetic algorithms using the heuristic crossover by far provide the best solution to our physical mapping problem.

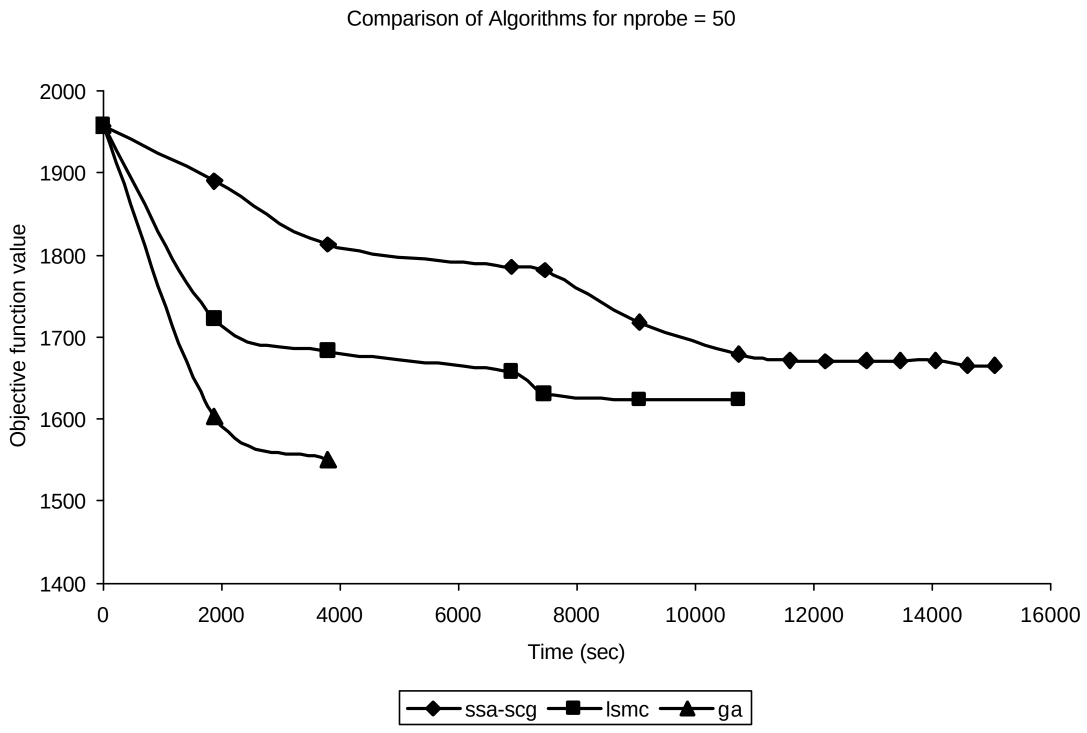

Figure 4.6 Timing comparison among SA, GA, and LSMC for *nprobe* = 50. Each algorithm used the serial conjugate gradient descent procedure

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

Reconstruction of physical maps of chromosomes in this project is based on a maximum likelihood estimator model that derives an objective function of two parameters, i.e., the probe ordering and the inter-probe spacings. The most likely solution to the problem will have the lowest objective function value. In this project, we have used several optimization approaches, including simulated annealing, LSMC, and evolutionary algorithm (evolutionary programming and genetic algorithm), for the physical mapping problem. Parallelization of the algorithms is carried out at two levels. At the higher level, we have partitioned the workloads of the simulated annealing and evolutionary algorithms among SMPs using the inter-process communication via message passing. At the lower level, a conjugate gradient descent search and a local exhaustive search are parallelized using shared-memory multithreaded programming. These parallel algorithms are implemented on a cluster of SMP machines, and both synthetic and real datasets have been used to test the effectiveness and performance of the algorithms.

Simulated annealing is our first attempt for the physical mapping problem. In this algorithm, each annealing step iterates through a number of *perturb-evaluate-decide* cycles. The probe order is first randomly perturbed to generate a new candidate solution that is subject to a conjugate gradient descent search to yield the optimal inter-probe spacings for the new order. In our parallel simulated annealing algorithm, the workload of each annealing step is evenly divided among multiple SMP machines. Within each SMP, we spawn multiple POSIX threads and bind them onto processors to conduct the local exhaustive search. The parallel simulated annealing algorithm has provided some good performance, and the measured efficiencies for many of the test cases are above 80%.

The LSMC is a variant of the simulated annealing algorithm with the addition of a deterministic local exhaustive search after a large *Kick* has been applied to generate a non-local intermediate solution. In our project, the Hamming distance rather than the MLE objective function value has been used in the local exhaustive search procedure, which leads to an improved solution in a shorter time. Both the local exhaustive search and conjugate gradient descent search procedures are parallelized at the lower level. Parallelization of the local exhaustive search yields an efficiency of more than 90% for majority of the tested datasets. Both the speedups and efficiencies of the parallel LSMC algorithms are encouraging.

In addition to the simulated annealing and the LSMC algorithms, we have also used the evolutionary algorithms, including an evolutionary programming/LSMC hybrid and two versions of the genetic algorithm, for our physical mapping problem. The evolutionary programming/LSMC hybrid algorithm starts with a population of locally optimal solutions, each going through an independent LSMC process.   Our serial evolutionary programming/LSMC hybrid algorithm takes a longer time than the LSMC, but yields a better solution. Parallelization of the evolutionary programming/LSMC hybrid has yielded a much improved solution in a shorter time compared to the serial LSMC algorithm.

In our two versions of the genetic algorithm for the physical mapping problem, we have used the heuristic crossover operator to create offspring followed by occasional mutations. The serial versions of the genetic algorithm are consistently better than the simulated annealing and LSMC in both effectiveness and performance. For the synthetic datasets *nprobe* = 50, the genetic algorithms take only 25% and 33% percent of the execution time of simulated annealing and LSMC respectively but yield much improved solutions.

By far, the genetic algorithms using the heuristic crossover have provided the best solutions to the physical mapping problem in our project. The implementation of parallel genetic algorithms in this project can be easily carried out using the same combination of inter-process communication and shared-memory programming. Improvement of the efficiency of the genetic algorithms may also be possible by incorporating load-balancing techniques during the implementation.

BIBLIOGRAPHY

Alizadeh, F., R. M. Karp, D. K. Weisser, and G. Zweig. 1994. Physical mapping of chromosomes using unique probes. Pp. 489-500 in Proceedings of the ACM-SIAM Conference on Discrete Algorithms. ACM Press: New York.

Alizadeh, F., R. M. Karp, L. A. Newberg, and D. K. Weisser. 1995. Physical mapping of chromosomes: a combinatorial problem in molecular biology. Algorithmica 13(1/2): 52-76.

Ben-Dor, A. and B. Chor. 1997. On constructing radiation hybrid maps. Pp. 17-26 in Proceedings of the ACM Conference on Computational Molecular Biology. ACM Press: New York.

Bhandarkar, S. M., Y. Zhang, and W. D. Potter. 1994. An edge detection technique using genetic algorithm-based optimization. Pattern Recognition 27: 1159-1180.

Bhandarkar, S. M., Machaka, S. A., Shete, S. S., and R. N. Kota. 2001. Parallel computation of a Maximum Likelihood estimator of a physical map. Genetics 157: 1021-1043.

Christof, T., M. Junger, J. D. Kececioglu, P. Mutzel, and G. Reinelet. 1997. A branch-and-cut to physical mapping of chromosomes by unique end probes. J. Comput. Biol. 4(4): 433-447.

Culler, D. E., Singh, J. P., and A. Gupta. 1999. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers: San Francisco.

Darwin, C. 1859. On the Origin of Species. 1st edition. London

Flynn, M. J. 1966. Very high speed computing systems. Proc. IEEE 54: 1901-1909.

Falkenauer, E. 1998. Genetic Algorithms and Grouping Problems. John Wiley & Sons: New York.

Fasulo, D. P., T. Jiang, R. M. Karp, R. Settergren and E. C. Thayer. 1997. An algorithmic approach to multiple complete digest mapping. Pp. 118-127 in Proceedings of the ACM Conference on Computational Molecular Biology. ACM Press: New York.

Fogel, L. J., A. J. Owens, and M. J. Walsh. 1966. Artificial Intelligence through Simulated Evolution. Wiley: New York.

Fogel, D. B. 1988. An evolutionary approach to the traveling salesman problem. Biol. Cybern. 60: 139-144.

Fountain, T. J. 1994. Parallel Computing: Principles and Practice. Cambridge University Press: New York, NY.

Goldberg, D. E. 1989. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley: New York.

Golub, G., and J. M. Ortega. 1993. Scientific Computing: An Introduction with Parallel Computing. Academic Press: San Diego.

Greenberg, D. S., and S. Istrail. 1995. Physical mapping by STS hybridization: algorithmic strategies and the challenge of software evaluation. J. Comput. Biol. 2(2): 219-273.

Gropp, W., Lusk, E., and A. Skjellum. 1999. Using MPI, 2nd ed. MIT Press: Cambridge, MA.

Jain, M., and E. W. Myers. 1997. Algorithms for computing and integrating physical maps using unique probes. J. Comput. Biol. 4(4): 449-466.

Jiang, T. and R. M. Karp. 1997. Mapping clones with a given ordering or interleaving. Pp. 400-409 in Proceedings of the ACM-SIAM Conference on Discrete Algorithms. ACM Press: New York.

Karp, R. M., and S. Shamir. 1998. Algorithms for optical mapping. Pp. 117-124 in Proceedings of the ACM-SIAM Conference on Discrete Algorithms. ACM Press: New York.

Kececioglu, J. D., Shete, S. S., and J. Arnold. 2000. Reconstructing distances in physical maps of chromosomes with nonoverlapping probes. Pp. 183-192 in Proceedings of the ACM Conference on Computational Molecular Biology. Tokyo, Japan.

Kota, R. N. 2000. Parallel Algorithms for the Maximum-Likelihood Model for Chromosome Reconstruction. MS thesis, University of Georgia.

Lee, J. K., V. Dancik, and M. S. Waterman. 1998. Estimation for restriction sites observed by optical mapping using reversible-jump Markov chain Monte Carlo. Pp. 147-152 in Proceedings of the ACM-SIAM Conference on Discrete Algorithms. ACM Press: New York.

Lin, S., and B. Kernighan. 1973. An effective heuristic for the traveling salesman problem. Operations Research 21: 498-516.

Machaka, S. A. 1998. Parallel Algorithms for Chromosome Physical Mapping Using a Cluster of Workstations. MS thesis, University of Georgia.

Mahfoud, S. W. and D. E. Goldberg. 1995. Parallel recombinative simulated annealing: A genetic algorithm. Parallel Computing 21: 1-28.

Martin, O., S. W. Otto, E. W. Felten. 1991. Large-Step Markov Chains for the Traveling Salesman Problem. Complex Systems 5: 299.

Michalewicz, Z. 1994. Genetic Algorithms + Data Structures = Evolution Programs, 2nd ed. Springer-Verlag: Berlin.

Message Passing Interface Forum. 1994 MPI: A Message-Passing Interface standard. Intl. Journ. Supercomputer Appl. and High Perf. Computing 8: 165-416 .

Muthukrishnan, S., and L. Parida. 1997. Towards constructing physical maps by optical mapping: an effective, simple, combinatorial approach. Pp. 209-219 in Proceedings of the ACM Conference on Computational Molecular Biology. ACM Press: New York.

Prade, R. A. et al. 1997. *In vitro* reconstruction of the *Aspergillus* (= *Emericella*) *nidulans* genome. Proc. Natl. Acad. Sci. USA. 94: 14564-14569.

Prasad, S. 1997. Multithreading Programming Techniques. McGraw-Hill: New York.

Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. 1988. Numerical Recipe in C. Cambridge University Press: New York.

Shete, S. S. 1998. Estimation Problems in Physical Mapping of a Chromosome and Branching Processes with Immigration. Ph.D. dissertation, University of Georgia.

Shete, S., Kececioglu, J., and J. Arnold. 1998. Estimation problems in physical mapping of a chromosome and in a branching process with immigration. Unpublished manuscript, Department of Statistics, University of Georgia.

Slonim, D., L. Kruglyar, L. Stein and E. Lander. 1997. Building human genome maps with radiation hybrids. Pp. 277-286 in Proceedings of the ACM Conference on Computational Molecular Biology. ACM Press: New York.

Sunderam, V. S. 1990. *PVM: A framework for parallel distributed computing*. Concurency: Practice and Experience 2(4): 315—349.

Tanenbaum, A. S. 2001. Modern Operating Systems. 2nd ed. Prentice Hall: Upper Saddle River, New Jersey.

Watson, J. D., Gilman, M., Witkowski, J., and M. Zoller. 1992. Recombination DNA. 2nd. Scientific American Books: New York.