The Journal of Supercomputing, 11, 61–86 (1997)

Chromosome Reconstruction from Physical Maps Using a Cluster of Workstations

SUCHENDRA M. BHANDARKAR suchi@cs.uga.edu
SALEM MACHAKA machaka@cs.uga.edu
Department of Computer Science, University of Georgia, 415 Boyd Graduate Studies Research Center, Athens, GA 30602-7404, USA

Editor: Richard Draper

Abstract. Ordering clones from a genomic library into physical maps of whole chromosomes presents a central computational problem in genetics. Chromosome reconstruction via clone ordering is shown to be isomorphic to the classical NP-complete *Optimal Linear Arrangement* problem. Parallel algorithms for simulated annealing and microcanonical annealing based on Markov chain decomposition are proposed and applied to the problem of chromosome reconstruction via clone ordering. These algorithms are implemented on a cluster of UNIX workstations using the Parallel Virtual Machine (PVM) system. PVM is a software system that permits a heterogeneous collection of networked computers to be viewed by a user's program as a single monolithic parallel computing resource. The parallel algorithms are implemented and tested on clonal data derived from Chromosome IV of the fungus *Aspergillus nidulans*. Perturbation methods and problem-specific annealing heuristics for the parallel simulated annealing and parallel microcanonical annealing algorithms are proposed and described. Convergence, speedup and scalability characteristics of the various parallel algorithms are analyzed and discussed.

Keywords: simulated annealing, microcanonical annealing, parallel processing, chromosome reconstruction, clone ordering

1. Introduction

A central problem in genetics, right from its very inception, is that of creating maps of entire chromosomes[31] which could then be used to reconstruct the chromosome's DNA sequence. These maps are central to the understanding of the structure of genes, their function and their evolution. The advent of molecular genetics has led to a wealth of DNA markers along a chromosome making feasible the mapping of entire genomes such as the human genome [5, 10]. The large number of DNA markers and the ease with which molecular markers are assayed has shifted the problem from one of collecting and mapping the data for genetic maps to the computational and statistical problem of assembling entire maps.

Chromosomal maps fall into two broad categories – *genetic maps* and *physical maps*. In their pioneering work, Lander and Green[24] provided for the first time, a computational tool for the assembly of genetic maps with many markers. This breakthrough was of both, conceptual and practical significance, and aided scientists in hunting down genes of biological, medical and statistical interest. While genetic maps narrow the search for genes to a particular chromosomal region, it is a physical map that ultimately allows the recovery and molecular manipulation of genes of interest.

A physical map is defined as a partial ordering of distinguishable DNA fragments or *clones* by their position along the entire chromosome. Examples of chromosomal physical maps include cytological maps, radiation–hybrid maps, ordered clonal libraries (i.e. *contig maps*) and ultimately a chromosome's entire DNA sequence. Physical maps were essential in the initial isolation of the homeotic genes in *Drosophila melanogaster*[6], the homeotic flowering genes of *Arabidopsis thaliana*[28] and the complete genetic regulatory circuit for the asexual development in *Aspergillus nidulans*[33]. The physical map has begun to provide fundamental insights into the molecular genetics, the identification of new structural features of chromosomes of unknown function, the role of meiotic recombination and the role of sex in the evolution of species. It is for these reasons that the creation of physical maps for each of the human chromosomes and those of several model systems has been identified as the central goal of the Human Genome Project[10].

1.1. The Physical Mapping Problem

Several techniques exist for generation of physical maps from contig libraries such as those based on chromosome walking[6, 30] and chromosome jumping[11]. These techniques however, are tedious, subject to interference by repeated DNA sequences and confounded when regions of the chromosome are absent from the contig library. Our approach, on the other hand, is one based on clone ordering[13]. A hybridization profile is generated for each clone in the contig library. The clones in the library are then ordered along a chromosome, with respect to their position inferred using the hybridization profile, in order to generate a chromosomal physical map. Our technique for clone ordering uses data generated by sequence tagged site (STS) content mapping in which individual clones in a chromosome-specific library[8] are selected as probes based on a random sampling scheme described in [34] and hybridized to all clones in the library. Each clone is scored for the presence or absence of hybridization to each probe, resulting in the assignment of a digital *signature* to each clone. A *1* in a specific position in the clonal signature indicates hybridization to a specific probe whereas a *0* in a specific position in the clonal signature indicates absence of hybridization to that specific probe. Clones are assigned *id* numbers which may indicate, for example, a grid location on a particular petri plate in a clonal library. It is to be noted that the *id* number does not provide any information about a clone's physical location on the chromosome. This technique for physical map assembly has proven adaptable to a wide variety of experimental protocols.

The entire clonal library is represented by a binary clone/probe hybridization matrix with each row representing a clone and each column representing a specific probe. A *1* in the ith row and jth column in the binary matrix indicates the hybridization of the ith clone to the jth probe whereas a *0* indicates absence of hybridization. Each row of the clone/probe hybridization matrix is the digital signature of that particular clone. When clones overlap, they tend to hybridize to the same probes, and their digital signatures tend to be similar. Probes link clones into contiguous blocks or contigs by their shared pattern of clonal hybridization. As a consequence, the similarity between signatures of clones can be used to reconstruct their true ordering along the chromosome in much the same way as books are organized by call number on shelves in a library. The clones when

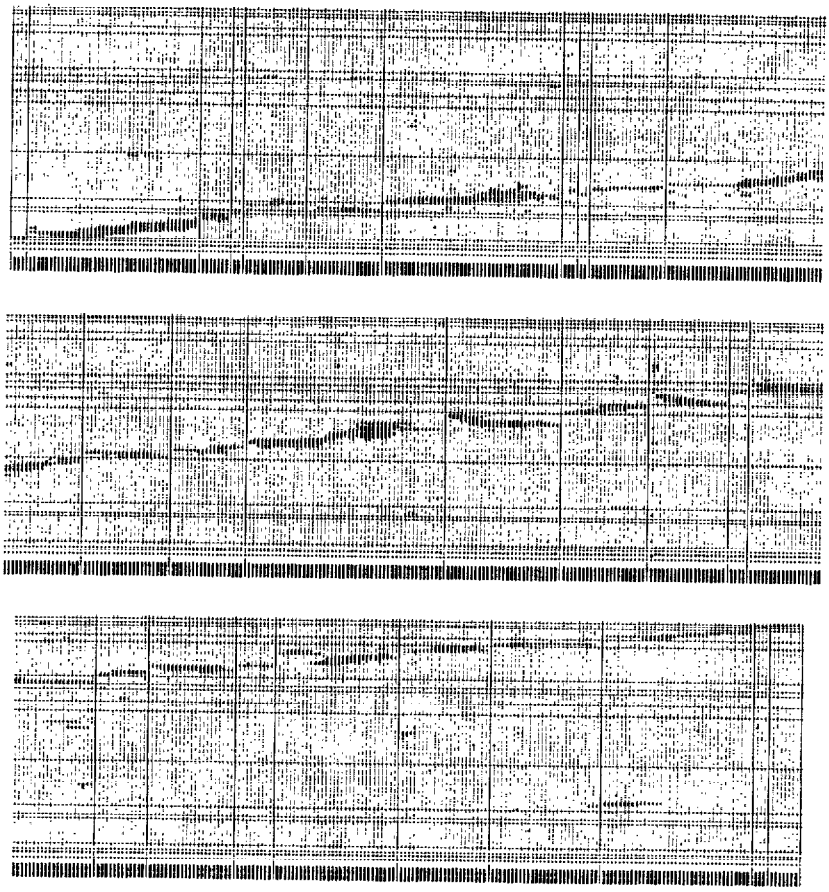

Figure 1. An Ordered Physical Map of *Aspergillus nidulans*, Chromosome IV

permuted into their inferred order along the chromosome yields the desired physical map which can be visualized as a two-dimensional layout[27] as shown in Figure 1. In the two-dimensional layout (Figure 1), contiguous blocks of clones or contigs are separated by horizontal lines along one dimension whereas *cells* (i.e., contiguous block of probes ordered by their position along the chromosome and linked together by the intervening overlapping clones) are separated by vertical lines along the other dimension.

1.2. Computational Complexity of Optimal Clone Ordering

Since the relative ordering of the clones within the contig library is unknown, we are faced with the computational problem of reconstructing the true clonal order along the chromosome. We formulate this problem as one of constructing a clone ordering that minimizes the sum of differences of signatures across a reconstructed chromosome as follows:

1. The *Hamming* distance $d(C_i, C_j)$ between two clonal signatures C_i and C_j is defined to be the measure of their dissimilarity.

2. The reconstruction error measure D for an ordering is defined as the sum of the Hamming distances between all pairs of successive clones in the ordering:

$$D = \sum_{i=1}^{n-1} d(C_i, C_{i+1})$$

   D is also referred to as the total linking distance of the clone ordering.

3. The desired ordering is deemed to be that which results in minimization of the ordering error D. Let D_m denote the minimum reconstruction error associated with the space of all possible clonal orderings and D_0 the reconstruction error associated with the *true* ordering. It can be shown that

$$\lim_{n \to \infty} prob(|D_m - D_0| > \epsilon) = 0$$

   That is to say, D_m converges in probability to D_0 as the size n of the clonal library grows[37].

The problem of coming up with an optimal ordering can be shown to be isomorphic to the classical NP–complete *Optimal Linear Arrangement* (OLA) problem for which no polynomial-time algorithm for determining the optimal solution is known[16]. Deriving an optimal solution via exhaustive search of all possible solutions to the OLA problem would result in an algorithm with exponential-time complexity. This has prompted the development of heuristic optimization algorithms that are capable of arriving at near-optimal solutions in polynomial-time on average. However, deterministic optimization algorithms have a tendency to get trapped in a *locally* optimal solution that may be far from the desired *globally* optimal solution. An attractive alternative is to use stochastic optimization techniques such as simulated annealing[18, 22] and microcanonical annealing[12] which are known to be robust in the presence of local optima in the solution space and also have general applicability.

1.3. An Overview of Simulated Annealing

Simulated Annealing (SA) is a general purpose combinatorial optimization technique[22, 23] which has proved useful in solving classical NP-complete and NP-hard problems such

as the Traveling Salesman Problem[7, 15] and complex combinatorial problems in a variety of application domains such as computer–aided design for VLSI[4, 9, 20, 29, 36], image processing[18], neural computing[2] and operations research[26]. SA emulates the physical process of annealing or gradual cooling that is used to create a crystalline solid occupying a low energy state from its high-energy molten state.

The SA algorithm (Figure 2) consists of three phases (i) Perturb or Move Phase (ii) Evaluate Phase and (iii) Decide Phase as described below:

Perturb Phase: Given an n-variable objective function of the form $f(\mathbf{x}) = f(x_1, x_2, \ldots, x_n)$ to be minimized and a current solution $\mathbf{x}_i$, the current solution $\mathbf{x}_i$ is systematically perturbed to yield another candidate solution $\mathbf{x}_j$. In our case, the clone ordering is systematically permuted by reversing the ordering within a block of clones. The block of clones is chosen by randomly selecting the two endpoints (i.e., clone positions) of the clone block to be reversed. The selection of the two endpoints is done using a pseudorandom number generator with a uniform distribution in the range $0 \ldots (n-1)$ where n is the number of clones.

Evaluate Phase: The new solution $\mathbf{x}_j$ is evaluated using the objective function i.e., $f(\mathbf{x}_j)$ is computed. In our case, the total linking distance associated with the new solution (i.e., new clone ordering) is computed.

Decide Phase: If $f(\mathbf{x}_j) < f(\mathbf{x}_i)$ then $\mathbf{x}_j$ is accepted as the new solution. If $f(\mathbf{x}_j) \geq f(\mathbf{x}_i)$ then $\mathbf{x}_j$ is accepted as the new solution with probability p the value of which is computed using the Metropolis function

$$p = \exp\left(-\frac{f(\mathbf{x}_j) - f(\mathbf{x}_i)}{T}\right) \tag{1}$$

at a given value of temperature T whereas $\mathbf{x}_i$ is retained with probability $(1-p)$.

As is evident from the Decide Phase in Figure 2, the SA algorithm is capable of *probabilistically* accepting new candidate solutions with higher values of f as compared to the current solution (i.e., new candidate solutions that are locally suboptimal). This gives SA the capability of climbing out of local minima which deterministic optimization techniques such as hill-climbing search or dynamic programming lack. The parameter T is referred to as the temperature of the system. The perturb-evaluate-decide cycle, which constitutes a single iteration of the SA algorithm, is carried out a fixed number of times for a given value of T which is then systematically reduced. A sequence of monotonically decreasing positive values for T denoted by $\{T_k\}$ is called an *annealing schedule* and the generating function for this sequence is called the *annealing function*. The annealing schedule represents the manner in which the SA algorithm is *annealed* or *cooled* towards a global minimum. As can be seen from the Decide Phase, at sufficiently high temperatures, SA resembles a random search whereas at lower temperatures it acquires the characteristics of a deterministic local search or hill-climbing search.

The SA algorithm generates an asymptotically ergodic (and hence stationary) Markov chain of solution states at a given temperature value. The annealing schedule $\{T_k\}$ is said to

```
1.  T = T_max;
2.  Finished = FALSE;
3.  while (not Finished)
4.  beginwhile
5.      for (count=1; count ≤ COUNT_LIMIT; count = count + 1)
6.      beginfor
7.          1. Phase One - Perturb
8.              (a) Randomly perturb the existing solution to
9.              generate a new candidate solution;
10.         2. Phase Two - Evaluate
11.             (a) Compute the cost associated with the new candidate
12.             solution;
13.             (b) Compute Δf, the change in cost function that
14.             would result if the new candidate solution were to
15.             replace the existing solution;
16.         3. Phase Three - Decide
17.             (a) Accept the new candidate solution if it causes the
18.             cost function to decrease;
19.             (b) accept the new candidate solution with probability
20.             P_T(Δf) computed using the Metropolis function
21.             if it causes the cost function to increase;
22.     endfor
23.     Update the temperature using the annealing function T = A(T);
24.     If the cost function has not changed for the past successive
25.     K temperature values then Finished = TRUE;
26. endwhile
```

Figure 2. Outline of a serial simulated annealing algorithm

be asymptotically good if the Markov chain of solution states converges to a global minimum with unit probability as $k \rightarrow \infty$. It has been shown that logarithmic annealing schedules of the form $T_k = R/\log k$ for some value of $R > 0$ are asymptotically good i.e., the SA algorithm converges to a global minimum with unit probability given a logarithmic annealing schedule[18]. In fact, previous efforts at using SA for generating physical maps from clonal libraries have proved very successful[13, 14]. However, the fact that asymptotically good annealing schedules are typically lengthy makes SA a computationally intensive procedure in spite of its avoidance of local minima in the objective function landscape. Parallelization of SA is therefore imperative if one desires high quality solutions to complex multivariate optimization problems, such as reconstruction of complex chromosomes via clone ordering, in a reasonable time frame.

1.4. An Overview of Microcanonical Annealing

Microcanonical Annealing (MCA)[12], like SA, is also a stochastic optimization algorithm that emulates the physical process of cooling a high-energy system to a low-energy state. The primary difference between SA and MCA is that in the case of SA, the system is assumed to be in thermal equilibrium with its environment (i.e., the system and its environment are assumed to be at the same temperature) whereas in the case of MCA, the system is assumed to be thermally insulated from its environment. As a result, in MCA, the total energy of the system, (i.e., sum of kinetic energy and potential energy), is always conserved. Such a system is referred to as a *microcanonical ensemble* in statistical thermodynamics. The potential energy of the system is the multivariate objective function f to be minimized whereas the kinetic energy $E_k > 0$ is represented by a demon or a collection of demons. In the latter case, the total kinetic energy is the sum of all the demon energies. In the context of our problem, a demon is assigned to each distinct pair of clones. A square symmetric matrix E_k of size $n \times n$, where n is the number of clones, is used to store the demon energy for each clone pair. Thus, an entry $E_k(i, j)$ refers to the kinetic energy of the demon associated with the ith and jth clone.

Analogous to the serial SA algorithm, the serial MCA algorithm (Figure 3) consists of three phases: (i) Perturb or Move Phase (ii) Evaluate Phase and (iii) Decide Phase. The Perturb and Evaluate phases in MCA are identical to their SA counterparts. The Decide Phase in MCA is as follows:

Decide Phase: If $f(\mathbf{x}_j) < f(\mathbf{x}_i)$ then $\mathbf{x}_j$ is accepted as the new current solution. If $f(\mathbf{x}_j) \geq f(\mathbf{x}_i)$ then $\mathbf{x}_j$ is accepted as the new current solution only if $E_k(p, q) \geq f(\mathbf{x}_j) - f(\mathbf{x}_i)$ where $E_k(p, q)$ is the kinetic energy of the demon associated with the clones at positions p and q which correspond to the endpoints of the clone block within which the ordering is being reversed. If $f(\mathbf{x}_j) \geq f(\mathbf{x}_i)$ and $E_k(p, q) < f(\mathbf{x}_j) - f(\mathbf{x}_i)$ then $\mathbf{x}_i$ is retained. In the event that $\mathbf{x}_j$ is accepted as the new current solution the kinetic energy demon is updated $E_k(p, q) = E_k(p, q) + f(\mathbf{x}_j) - f(\mathbf{x}_i)$.

In the MCA algorithm, the kinetic energy demons play the same role as the temperature parameter in the SA algorithm and are subject to the same annealing schedule. The MCA

```
1.  Initialize the demon energies:
2.      for (i = 0; i < n; i = i + 1)
3.          for (j = 0; j < n; j = j + 1) E_k[i][j] = E_max;
4.  Finished = FALSE;
5.  while (not Finished)
6.  beginwhile
7.      for (count = 1; count ≤ COUNT_LIMIT; count = count + 1)
8.      beginfor
9.          1. Phase One - Perturb
10.             (a) Randomly perturb the existing solution
11.             to generate a new solution;

12.         2. Phase Two - Evaluate
13.             (a) Compute the potential energy associated with
14.             the new solution;
15.             (b) Compute Δf, the change in potential energy
16.             that would result if the new solution were to replace
17.             the existing solution;

18.         3. Phase Three - Decide
19.             (a) Accept the new solution if Δf < 0
20.             (b) If Δf ≥ 0 accept the new solution
21.             if E_k[p][q] ≥ Δf ≥ 0;
22.             (c) If the new solution is accepted, update the
23.             kinetic energy demons:
24.                 E_k[p][q] = E_k[p][q] − Δf;
25.                 E_k[q][p] = E_k[q][p] − Δf;
26.     endfor

27.     Update the kinetic energy demon using the annealing function:
28.     for (i = 0; i < n; i = i + 1)
29.         for (j = 0; j < n; j = j + 1) E_k[i][j] = A(E_k[i][j]);
30.     If the potential energy has not changed for K consecutive
31.     iterations then Finished = TRUE;
32. endwhile
```

Figure 3. Outline of the serial microcanonical annealing algorithm

algorithm, like the SA algorithm, generates an asymptotically ergodic Markov chain of solution states that converges in the limit to a globally optimal solution. MCA has a computational advantage over SA since the computation of the exponential function in the Decide Phase of the latter (equation (1)) is replaced by simple addition in the case of the former.

1.5. The Parallel Virtual Machine (PVM)

The Parallel Virtual Machine (PVM) is a software environment created to exploit parallel/distributed computing across a variety of computer types[32]. A salient feature of PVM is that it makes a collection of heterogeneous computers appear as a single large virtual parallel machine that is based on a distributed–memory, message–passing paradigm. Although PVM allows several degrees of heterogeneity between its constituent computers in terms of the underlying architecture, data format, native operating system, memory capacity, computational power etc., in our case, the machines used are six identical high–performance SUN SPARC5 workstations running UNIX. It is important to point out that the performance of PVM depends on the network used to interconnect the various constituent computers. The latency of message transmission over the network becomes important when a task is sitting idle waiting for a message. It is even more important when the performance of the parallel algorithm is sensitive to the message arrival time. However, a principal advantage of PVM over a dedicated parallel machine is the general applicability of the former. As opposed to a dedicated parallel machine, PVM can take advantage of the computational power and the inherent strengths of different computers with different underlying architectures. The resulting speedup is often comparable to that of a dedicated parallel machine. The only cost introduced, however, is that resulting from the communication latency of the general purpose network used.

The PVM system is composed of two modules. The first module is a daemon, called pvmd, that resides on all the computers comprising the parallel virtual machine. The second module is a library of PVM interface routines that contain a set of primitives needed for message–passing, spawning processes, coordinating/synchronizing various tasks, broadcasting and collecting data, and modifying the virtual machine at run–time. The PVM system currently supports high–level programming languages such as C, C++ and FORTRAN. In this paper, all the algorithms discussed have been implemented in C using the corresponding PVM library.

1.5.1. PVM Primitives - A Brief Overview

In the following discussion we describe the key PVM primitives (in the C programming language) and the task/data decomposition techniques used to ensure proper flow of data and control in the context of our specific application. In the PVM system, an application is viewed as consisting of different tasks or components. A component, in this case, is considered to be a fairly independent (i.e., stand–alone) process, and therefore, is not thought of as a procedure or subroutine, but as a larger unit of an application. The PVM

software provides the appropriate environment in which to execute a set of cooperating tasks or components.

A process in PVM is an executing instance of a component. A PVM process executes on a physical computing platform termed as the host. Processes can be spawned either from a start-up program that is manually initiated, or from any other component. The primitive function pvm_spawn is used to spawn tasks. The arguments to this function are: the task_name, the input arguments passed, a flag which is used to specify the architecture on which to spawn the task or to start tasks under the debugger, the host_name, the number of such tasks spawned and an id_array that contains the task_id's of the spawned tasks (if more then one task is spawned). The primitive function pvm_kill, is used to kill a PVM task such as in the case of error. The only argument required, in this case, is the task_id of the task to be killed. The primitives pvm_parent and pvm_mytid return the parent_task_id and the task_id of the calling process respectively.

Interprocess communication via message passing, is one of the basic features supported by PVM. The message passing in PVM is buffered. The primitive function pvm_initsend is used to dynamically create a buffer and make it active for the purpose of holding a message. PVM contains a set of primitive functions for packing various types of data into an active buffer. For example, the primitive function pvm_pkbyte in this set is used to pack an array of bytes into the active buffer whereas the primitive pvm_pkint is used to pack an array of integers. The primitive functions from this set can be called successively and several times in order to pack various kinds of data into a single message. Other routines for buffer manipulation exist, but are tailored for a more low–level manipulation of buffers, especially when multiple buffers are used.

Once the data is packed in the form of a message, one can use the primitive function pvm_send to send it to another process. The required arguments for pvm_send are the task_id and the message_tag (which serves as a message identifier). Alternatively, the primitive function pvm_mcast can be used to broadcast the packed data to all the active processes running under the PVM system. Conversely, the primitive functions pvm_receive and pvm_upk_datatype are respectively used to receive a message from a certain task and unpack it. The interprocess communication primitives discussed thus far work in an asynchronous mode. To ensure the proper flow of data and control, synchronization needs to be performed explicitly. Whenever a process sends a message, it cannot resume execution until it receives an acknowledgement from the recipient process. Even though this tends to increase the interprocess(or) communication overhead in a typical application, it ensures correct execution of the application program. Other PVM primitives include those for dynamic grouping of processes and dynamic (re)configuration of the virtual machine. Deleting and adding hosts, for instance, can be performed dynamically using the pvm_addhosts and pvm_delhosts primitives. The above list of PVM functions is by no means exhaustive. We have just attempted to give a brief overview of the most commonly used functions. We refer the interested reader to [17] for a more detailed description of PVM and its functions.

The purpose of this paper is to present the design and analysis of parallel annealing algorithms using PVM that are suitable for the problem of chromosome reconstruction from overlapping clones or contig maps. The focus of the paper is on parallel algorithm design and performance analysis in terms of speedup, efficiency of parallelism and the scalability

of speedup and efficiency with respect to increasing number of processors and increasing problem size. The remainder of the paper is organized as follows: Section 2 discusses various parallel annealing models. Section 3 discusses the application of the parallel simulated annealing (PSA) and parallel microcanonical annealing (PMCA) models to the problem of chromosome reconstruction from contig maps. Section 4 contains some experimental results on a clone data set derived from Chromosome IV of the fungus *Aspergillus nidulans*. In Section 5 the paper is concluded and future research directions are outlined.

2. Parallel Annealing Algorithms

Parallelization of annealing algorithms has been attempted by several researchers especially in the area of computer–aided design, image processing and operations research[3, 19]. Parallelization strategies for the SA and MCA algorithms can be broadly classified as:

(A) Functional Parallelism within a Move: In this approach, each individual move or perturbation is evaluated faster by breaking up the task of evaluating each move into subtasks that can be performed in parallel by multiple processors. This form of parallelism, however, is application–specific since the process of decomposing the task of evaluating each move into subtasks is very much dependent on the objective function under consideration[36]. Also, since each individual subtask need not be identical, this approach could potentially cause a load imbalance among the processors leading to a situation where certain processors have to wait for other processors to finish.

(B) Control Parallelism: In this mode of parallelism each processor works on its independent version of the SA or MCA algorithm. Control parallelism comprises of the following subcategories:

- **(i) Multiple Active Iterations:** The basic move–evaluate–decide iteration cycle is so decomposed among multiple processors such that several iteration cycles are active at any time. Due to the dependency between successive iteration cycles, processors are engaged in *speculative* computation. However, speculative computation needs to be performed very judiciously to obtain appreciable speedup[35].
- **(ii) Parallel Markov Chain Generation:** The annealing mechanism in serial SA and serial MCA is that of generating an asymptotically ergodic first-order Markov chain of solution states at each value of temperature[18] or kinetic energy[12]. The process of Markov chain generation can be parallelized on a set of processors using an algorithm that is well suited for systolic architectures[1, 3, 19, 21].
- **(iii) Multiple Searches:** Several searches can be carried out concurrently on multiple processors. The searches could be interacting in which case the processors exchange data periodically or the searches could be non–interacting in which case the processors proceed independently of each other[3, 25].

(C) Data Parallelism: This form of parallelism involves distributing the state variables in a multivariate solution vector amongst the individual processors in the multiprocessor

architecture. Each processor can perturb its state variables independently of the other processors. However, the Evaluate Phase and the Decide Phase present certain problems in a distributed–memory environment since information about the global state is needed to evaluate correctly the cost function and decide whether or not to accept the current move. There are three possible sub-approaches to parallel evaluation of multiple moves or perturbations:

(i) Parallel Evaluation of Multiple Moves with Acceptance of a Single Move: In this case, although a number of moves are considered and evaluated in parallel, only one move is accepted. The other moves are rejected although many of the rejected moves may be good moves. The convergence characteristics of the parallel algorithm are similar to those of the sequential algorithm. However, since many good moves are wasted, the speedup is limited[9].

(ii) Parallel Evaluation of Multiple Moves with Acceptance of Multiple Non-interacting Moves: A set of non–interacting (i.e., independent) moves is determined either statically or dynamically. Parallelism is then applied to the moves within the set. The convergence characteristics of the parallel algorithm are similar to those of the serial algorithm. However, due to the overhead of determining a maximal set of non–interacting moves, the speedup is limited[20].

(iii) Parallel Evaluation and Acceptance of Multiple Moves: In this case multiple moves are evaluated for acceptance without regard to their independence based on possibly inaccurate information. Some error in the computation of the cost function by different processors is allowed. This approach allows for maximum parallelism but the convergence characteristics of the parallel algorithm may be drastically different from those of the serial algorithm thereby leading to unexpected and/or undesirable final results. This problem can be alleviated by tracking the error and synchronizing the processors from time to time[4].

3. Parallel SA and MCA using PVM

Of the various parallelization strategies for annealing algorithms, we deemed the ones based on multiple searches (i.e., B(iii) in Section 2) to be the most promising from the viewpoint of implementation using PVM. As already mentioned, a candidate solution in the serial simulated annealing or microcanonical annealing algorithm can be considered to be an element of an asymptotically ergodic first-order Markov chain of solution states. Consequently, we have formulated and implemented two models of parallel simulated annealing (PSA) and parallel microcanonical annealing (PMCA) algorithms based on the distribution of the Markov chain of solution states using PVM. These models incorporate the parallelization strategies discussed under B(iii) in Section 2 and are described below:

- The Non–Interacting Local Markov chain (NILM) PSA and PMCA algorithms.
- The Periodically Interacting Local Markov chain (PILM) PSA and PMCA algorithms.

In the NILM PSA algorithm and NILM PMCA algorithm, each processor within the PVM system runs an independent version of the serial SA or MCA algorithm respectively. In essence, there are as many Markov chains of solution states as there are physical processors within the PVM system. Each Markov chain is local to a given processor and at any instant of time each processor maintains a candidate solution which is an element of its local Markov chain of solution states. The serial SA and MCA algorithms (Figures 2 and 3) run asynchronously on each processor i.e., at each temperature value or kinetic energy value each processor iterates through the perturb–evaluate–accept cycle COUNT_LIMIT number of times concurrently (but asynchronously) with all the other processors.

The perturbation function uses a parallel random number generator to generate the Markov chains of solution states. By assigning a distinct seed to each processor at the start of execution, we ensure that each processor contains a Markov chain of solution states that is independent from those in most or all of the other processors. The evaluation function and the decision function (for PSA and PMCA) are executed concurrently on the solution state within each processor. On termination of the annealing processes on all the processors, the best solution is selected from among all the solutions available on the individual processors. The NILM model is essentially that of multiple independent searches described in B(iii) in Section 2.

In the PILM PSA algorithm and the PILM PMCA algorithm, at each temperature or kinetic energy value, the perturb–evaluate–accept cycle is executed concurrently in all the processors COUNT_LIMIT number of times just as in the case of the NILM PSA algorithm and the NILM PMCA algorithm respectively. However, just before the parameter T or E_k is updated using the annealing function, the best candidate solution from among those in all the processors is selected and duplicated on all the other processors. The goal of this synchronization procedure is to refocus the search in the more promising regions of the solution space. This suggests that the PILM PSA or PILM PMCA algorithm should be potentially superior to its NILM counterpart. It should be noted that in the case of all four algorithms, PILM PSA, PILM PMCA, NILM PSA and NILM PMCA, a single Markov chain of solution states is generated entirely within a single processor. The PILM model is essentially that of multiple periodically interacting searches described in B(iii) in Section 2.

In the case of all the four algorithms, NILM PSA, NILM PMCA, PILM PSA and PILM PMCA, a master process is used as the overall controlling process. The master process runs on one of the processors within the PVM system. The master process spawns child processes on each of the other processors within the PVM system, broadcasts the data subsets needed by each child process, collects the final results from each of the child processes and terminates the child processes. The master process, in addition to the above mentioned functions for task initiation, task coordination and task termination, also runs its own version of the SA or MCA algorithm just as any of its child processes.

The master process for the PILM PSA and PILM PMCA algorithms is depicted in Figure 4. In the case of the PILM PSA and the PILM PMCA algorithms, at the end of a predetermined number of perturb–evaluate–decide iterations at each temperature or kinetic energy value, the master process collects the results from each child process along with its own result, broadcasts the best result to all the child processes and also replaces its own result with the

best result. The master process updates its temperature or kinetic energy value using the annealing function and proceeds with its local version of the SA or MCA algorithm. On reaching the final temperature or kinetic energy value, the master process collects the final results from each of the child processes along with its own results, selects the best result as the final solution and terminates the child processes. The master process for the NILM PSA and NILM PMCA algorithms (Figure 6) is similar to that in the case of the PILM PSA and PILM PMCA algorithms discussed earlier. The major difference is that in the case of the NILM PSA and NILM PMCA algorithms there is no periodic interaction between the master process and any of the child processes during the annealing procedure.

The child process in the case of the PILM PSA and PILM PMCA algorithms (Figure 5) collects the clonal data and certain initialization parameters from the master process. Each of the child processes runs an independent version of the SA or MCA algorithm on the received clonal data. The child processes interact periodically with the master process. At the end of the predetermined number of iterations at each temperature or kinetic energy value, each child process sends its result to the master process and waits for the master process to transmit the best result thus far. On receipt of the best result, each child process updates its temperature or kinetic energy value using the annealing function and proceeds with its local version of the SA or MCA algorithm on the received result. When the final value of the temperature or kinetic energy is reached, each child process transmits its result to the master process. The child process in the case of the NILM PSA and NILM PMCA algorithms (Figure 7) is similar to the one in the case of the PILM PSA and PILM PMCA algorithms except that there is no periodic interaction between the child process and the master process.

4. Experimental Results

The PSA and PMCA algorithms were implemented on a PVM system comprising of a cluster of eight SUN SPARC5 UNIX workstations using C as the programming language. The various algorithms were run on a clone data set derived from Chromosome IV of the fungus *Aspergillus nidulans* which was made available to us by Professor Jonathan Arnold, Department of Genetics, University of Georgia, Athens, Georgia. The data set consisted of 592 clones with each clone having a 115 bit signature.

In order to obtain a fair comparison between the various PSA and PMCA algorithms, the product (denoted by λ) of the number of processors n and the maximum number of iterations performed by a single processor at a given temperature or kinetic energy value (i.e., COUNT_LIMIT) was kept constant. For example, if one of the algorithms is run with 4 processors with COUNT_LIMIT = 50,000 then with 2 processors, COUNT_LIMIT would be 100,000 and with 1 processor, COUNT_LIMIT would be 200,000.

Figures 8 and 9 show the linking distance as a function of execution time with a varying number of processors (i.e., annealing curves) for the NILM PSA algorithm for $\lambda = 200,000$ and $\lambda = 1,200,000$ respectively. Similarly, Figures 10 and 11 depict the annealing curves for the PILM PSA algorithm for $\lambda = 200,000$ and $\lambda = 1,200,000$ respectively. We can readily observe that the NILM PSA algorithm has a faster rate of convergence than the PILM PSA algorithm in terms of execution time. However, the PILM PSA algorithm was

```
1.  master()
2.  beginmaster
3.      Phase 1: Initial Setup
4.          (a) Spawn Child Processes;
5.          (b) Read Input: clonal data, clonal distance matrix and
6.          initial seeds for the parallel random number generator;
7.          (c) Broadcast the input to all spawned child processes;

8.      Phase 2: The Annealing Algorithm and Process Coordination
9.          In the case of PSA:
10.             T = T_max; Finish = (T ≤ T_min);
11.         In the case of PMCA:
12.             E_k = E_max; Finished = (E_k ≤ E_min);

13.         while (not Finished)
14.         beginwhile
15.             for (Count = 1; Count ≤ COUNT_LIMIT; Count = Count + 1)
16.             beginfor
17.                 (a) Perturb Phase: same as serial SA or MCA;
18.                 (b) Evaluate Phase: same as serial SA or MCA;
19.                 (c) Decide Phase:
20.                     if PSA: SA decision criterion;
21.                     if PMCA: MCA decision criterion;
22.             endfor
23.             Receive clone ordering from each child process;
24.             Select best clone ordering;
25.             Send best clone ordering to each child process;
26.             if PSA: Update temperature T = A(T);
27.             if PMCA: Update kinetic energy E_k = A(E_k);
28.         endwhile

29.     Phase 3: Output Result;
30. endmaster
```

Figure 4. The master process for the PILM PSA/PMCA algorithm

```
1.  child()
2.  beginchild
3.      Phase 1: Initial Setup
4.          Receive Input from Master Process: clonal data, clonal distance
5.          matrix and initial seeds for parallel random number generator;

6.      Phase 2: Annealing Algorithm and Coordination with Master Process;
7.          In the case of PSA:
8.              T = T_max; Finished = (T ≤ T_min);
9.          In the case of PMCA:
10.             E_k = E_max; Finished = (E_k ≤ E_min);
11.         while (not Finished)
12.         beginwhile
13.             for (Count = 1; Count ≤ COUNT_LIMIT; Count = Count +1)
14.             beginfor
15.                 (a) Perturb Phase: same as serial SA or MCA;
16.                 (b) Evaluate Phase: same as serial SA or MCA;
17.                 (c) Decide Phase:
18.                     if PSA: SA decision criterion;
19.                     if PMCA: MCA decision criterion;
20.             endfor
21.             Send clone ordering to Master process;
22.             Receive best clone ordering from Master process;
23.             if PSA: Update temperature T = A(T);
24.             if PMCA: Update kinetic energy E_k = A(E_k);
25.         endwhile

26.     Phase 3: Exit;
27. endchild
```

Figure 5. The child process for the PILM PSA/PMCA algorithm

```
1.  master()
2.  beginmaster
3.      Phase 1: Initial Setup
4.          (a) Spawn Child Processes;
5.          (b) Read Input: clonal data, clonal distance matrix and
6.          initial seeds for parallel random number generator;
7.          (c) Broadcast the input to all spawned child processes;

8.      Phase 2: Annealing Algorithm
9.          In the case of PSA:
10.             T = T_max; Finished = (T ≤ T_min);
11.         In the case of PMCA:
12.             E_k = E_max; Finished = (E_k ≤ E_min);
13.         while (not Finished)
14.         beginwhile
15.             for (Count = 1; Count ≤ COUNT_LIMIT; Count = Count +1)
16.             beginfor
17.                 (a) Perturb Phase: same as serial SA or MCA;
18.                 (b) Evaluate Phase: same as serial SA or MCA;
19.                 (c) Decide Phase:
20.                     if PSA: SA decision criterion;
21.                     if PMCA: MCA decision criterion;
22.             endfor
23.             if PSA: Update temperature T = A(T);
24.             if PMCA: Update kinetic energy E_k = A(E_k);
25.         endwhile

26.     Phase 3: Choosing the Best Result
27.         (a) Receive clone ordering from each child process;
28.         (b) Select best clone ordering;
29.         (c) Output Result;
30. endmaster
```

Figure 6. The Master Process for the NILM PSA/PMCA algorithm

```
1.  child()
2.  beginchild
3.      Phase 1: Initial Setup
4.          Receive Input from Master Process: clonal data, clonal distance
5.          matrix and initial seeds for parallel random number generator;

6.      Phase 2: Annealing Algorithm;
7.          In the case of PSA:
8.              T = T_max; Finished = (T ≤ T_min);
9.          In the case of PMCA:
10.             E_k = E_max; Finished = (E_k ≤ E_min);
11.         while (not Finished)
12.         beginwhile
13.             for (Count = 1; Count ≤ COUNT_LIMIT; Count = Count +1)
14.             beginfor
15.                 (a) Perturb Phase: same as serial SA or MCA;
16.                 (b) Evaluate Phase: same as serial SA or MCA;
17.                 (c) Decide Phase:
18.                     if PSA: SA decision criterion;
19.                     if PMCA: MCA decision criterion;
20.             endfor
21.             if PSA: Update temperature T = A(T);
22.             if PMCA: Update kinetic energy E_k = A(E_k);
23.         endwhile

24.     Phase 3: Send Final Result;
25.         (a) Send final clone ordering to master process;
26.         (b) Exit;
27. endchild
```

Figure 7. The Child Process for the NILM PSA/PMCA algorithm

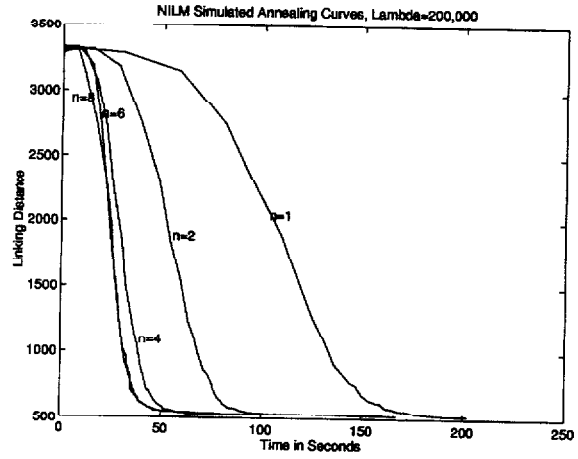

Figure 8. Annealing curves for the NILM PSA algorithm: Linking distance as a function of execution time for varying no. of processors, $\lambda = 200,000$

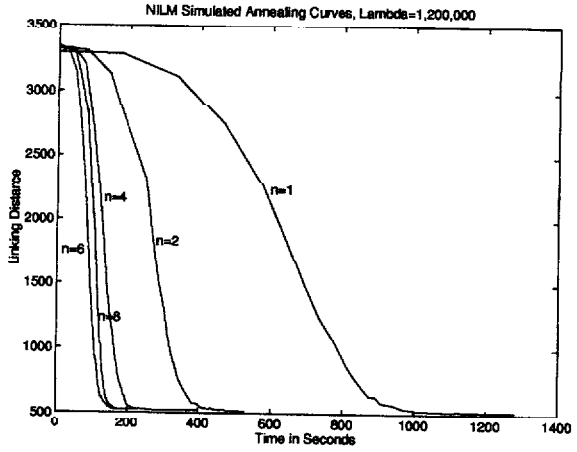

Figure 9. Annealing curves for the NILM PSA algorithm: Linking distance as a function of execution time for varying no. of processors, $\lambda = 1,200,000$

found to have a faster rate of convergence in terms of the number of iterations. This can be attributed to the fact that although the PILM PSA algorithm needs fewer iterations than the NILM PSA algorithm, the average time per iteration in the case of the former is greater due to the overhead of interprocessor communication and synchronization. The interprocessor communication and synchronization in the case of the PILM PSA algorithm arises due to

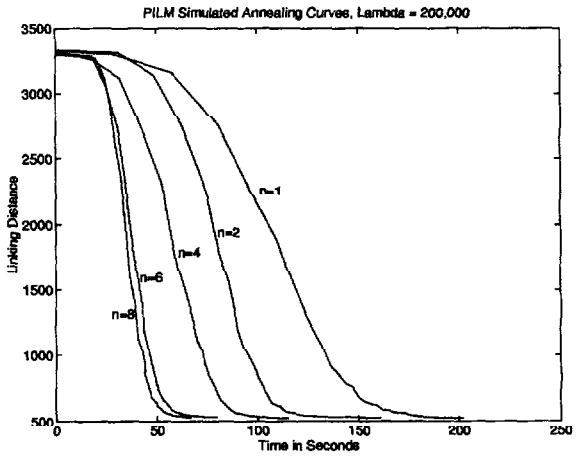

Figure 10. Annealing curves for the PILM PSA algorithm: Linking distance as a function of execution time for varying no. of processors, $\lambda = 200,000$

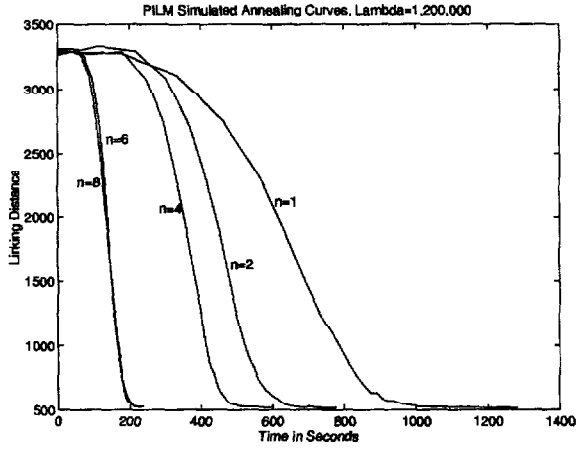

Figure 11. Annealing curves for the PILM PSA algorithm: Linking distance as a function of execution time for varying no. of processors, $\lambda = 1,200,000$

the fact that before each temperature update, each child process needs to convey its solution to the master process and wait for the master process to broadcast the best solution. On the other hand, in the case of the NILM PSA algorithm, the only interprocessor communication overhead involved is the initial broadcasting of the clonal data by the master process to

the child processes and collection of the final results by the master process from the child processes.

Table 1 shows the speedup results for all the four algorithms: NILM PSA, PILM PSA, NILM PMCA and PILM PMCA for a varying number of processors n and varying values of λ. For a given value of n, varying the value of λ directly affects the the interprocessor communication overhead ratio $\tau = T_{comm}/T_{cpu}$ where T_{comm} is the total time spent in interprocessor communication and T_{cpu} is the total time spent in computation. A higher value of λ for a given value of n implies a lower value for τ and vice versa. For a linking distance equal to the optimal value of 550, the NILM PSA algorithm is seen to have a speedup of 3.1 for $n = 8$ and $\lambda = 200,000$ and a speedup of 6.1 for $n = 8$ and $\lambda = 1,200,000$. This result is expected since the interprocessor communication overhead ratio τ is lower at higher values of λ resulting in a higher speedup. The PILM PSA algorithm is seen to have a speedup of 3.0 for $n = 8$ and $\lambda = 200,000$ and a speedup of 4.7 for $n = 8$ and $\lambda = 1,200,000$. As expected, the PILM PSA algorithm has a lower speedup than the NILM PSA algorithm for given values of n and λ due to the greater interprocessor communication overhead of the former. As also expected, for a given value of n the speedup of the PILM PSA algorithm shows an increasing trend with increasing λ.

Figures 12 and 13 depict the annealing curves for the NILM PMCA algorithm for $\lambda = 200,000$ and $\lambda = 1,200,000$ respectively. Similarly, Figures 14 and 15 depict the annealing curves for the PILM PMCA algorithm for $\lambda = 200,000$ and $\lambda = 1,200,000$ respectively. We observe that the parallelization of the MCA algorithm does not result in any speedup (Table 1). On the contrary, there is a degradation in performance as the number of processors (i.e., PVM hosts) is increased. This phenomena can be explained by the fact that the serial MCA algorithm is considerably faster than the SA algorithm for the same number of iterations. For given values of n and λ, the interprocessor communication overhead ratio τ is far greater in the case of the PMCA algorithm as compared to the PSA algorithm. The interprocessor communication overhead dominates the performance of the PMCA algorithm for smaller values of λ resulting in a degradation in performance as the number of processors is increased.

As expected, the NILM PMCA and PILM PMCA algorithms show an increasing trend in speedup (just as the NILM PSA and PILM PSA algorithms respectively) with increasing λ for a given value of n. The NILM PMCA algorithm shows a speedup of 0.4 for $n = 8$ and $\lambda = 200,000$ and a speedup of 1.3 for $n = 8$ and $\lambda = 1,200,000$ (Table 1). The PILM PMCA algorithm, on the other hand, shows a speedup of 0.6 for $n = 8$ and $\lambda = 200,000$ and a speedup of 2.1 for $n = 8$ and $\lambda = 1,200,000$ (Table 1).

5. Conclusions and Future Directions

In this paper, we designed and analyzed two models for a parallel simulated annealing (PSA) algorithm and two models for a parallel microcanonical annealing (PMCA) algorithm in the context of chromosome reconstruction via clone ordering. These models were based on the decomposition of the Markov chain of solution states across multiple processors and were termed as the Periodically Interacting Local Markov chain (PILM) model and the Non-Interacting Local Markov chain (NILM) model. The algorithms were implemented using

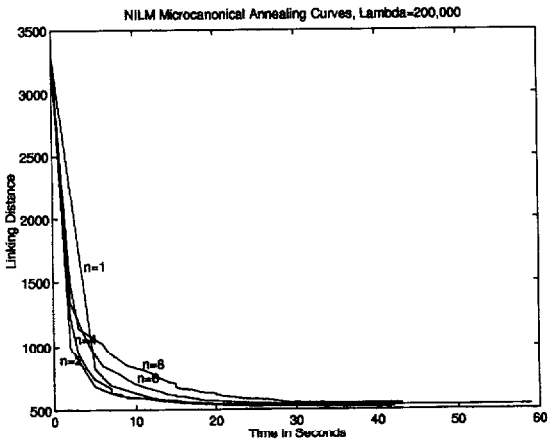

Figure 12. Annealing curves for the NILM PMCA algorithm: Linking distance as a function of execution time for varying no. of processors, $\lambda = 200,000$

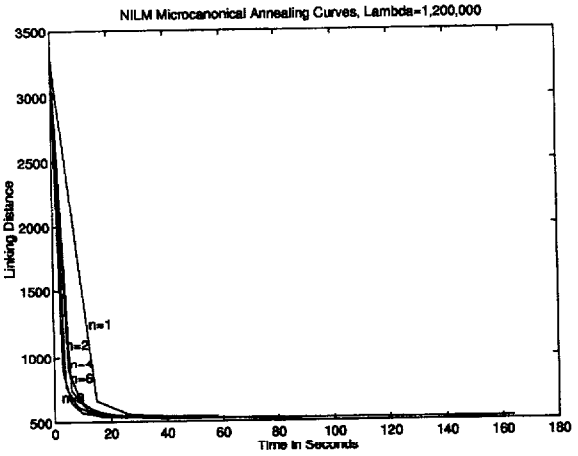

Figure 13. Annealing curves for the NILM PMCA algorithm: Linking distance as a function of execution time for varying no. of processors, $\lambda = 1,200,000$

the Parallel Virtual Machine (PVM) software environment which allows a heterogeneous collection of computers or processors to be programmed as a dedicated parallel computer. In our case the PVM system comprised of eight SUN SPARC5 workstations running UNIX.

Between the NILM PSA and PILM PSA models, the former showed faster convergence to a globally optimal solution and higher speedup as well. This could be attributed to

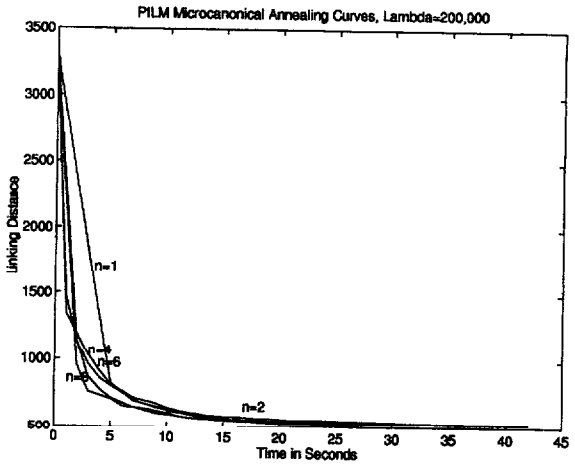

Figure 14. Annealing curves for the PILM PMCA algorithm: Linking distance as a function of execution time for varying no. of processors, $\lambda = 200,000$

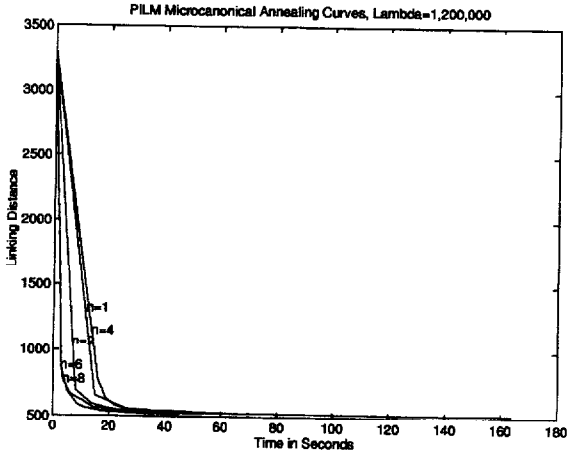

Figure 15. Annealing curves for the PILM PMCA algorithm: Linking distance as a function of execution time for varying no. of processors, $\lambda = 1,200,000$

the higher interprocessor communication overhead associated with the PILM PSA model. Both, the PILM and the NILM PMCA models showed very little improvement in terms of speed of convergence and speedup over the serial MCA algorithm. This can be attributed to the fact that the PMCA algorithm has a higher interprocessor communication overhead ratio τ, compared to the PSA algorithm. In other words, since the serial MCA algorithm is

Table 1. Speedup Results for the PSA and PMCA algorithms

Linking Distance = 550					
λ	n	NILM PSA	PILM PSA	NILM PMCA	PILM PMCA
200,000	1	1.0	1.0	1.0	1.0
	2	1.8	1.4	0.8	0.8
	4	3.0	1.8	0.8	1.0
	6	3.4	2.7	0.6	0.9
	8	3.1	3.0	0.4	0.6
400,000	1	1.0	1.0	1.0	1.0
	2	2.1	1.4	0.8	1.2
	4	3.9	1.8	0.9	1.4
	6	5.8	2.9	0.9	1.2
	8	5.5	3.5	0.6	1.1
800,000	1	1.0	1.0	1.0	1.0
	2	2.2	1.2	1.4	1.3
	4	3.7	1.9	2.0	1.7
	6	6.3	3.2	1.6	1.9
	8	5.4	3.5	1.2	1.5
1,000,000	1	1.0	1.0	1.0	1.0
	2	1.8	1.5	1.6	1.7
	4	4.5	2.0	1.9	1.7
	6	5.5	3.3	1.5	1.9
	8	5.5	4.7	1.6	2.4
1,200,000	1	1.0	1.0	1.0	1.0
	2	2.3	1.5	1.4	1.3
	4	4.7	1.9	1.8	1.1
	6	6.7	4.7	2.1	1.6
	8	6.1	4.7	1.3	2.1

inherently fast and computationally efficient, the performance of the PMCA algorithm was dominated by the interprocessor communication overhead leading to an overall performance degradation as the number of processors was increased. This was further corroborated by our observation that decreasing the value of τ (by increasing the number of iterations of the PMCA algorithm performed by each processor for a given kinetic energy value) improved the speedup characteristics of the NILM and PILM PMCA algorithms.

In summary, since the serial MCA algorithm is considerably faster than the serial SA algorithm, the NILM and PILM PMCA algorithms are better suited for a coarser-grained parallelization for a problem of given computational size as compared to the NILM and PILM PSA algorithms respectively. Alternatively, for a given granularity of parallelization the NILM and PILM PMCA algorithms are better suited for problems of larger computational size as compared to the NILM and PILM PSA algorithms respectively.

In the algorithms described here, the inter-clone distances were precomputed and stored in a distance matrix which was then accessed at run time. Although computationally efficient, straightforward distance matrix lookup techniques are not scalable in terms of memory since the memory requirements grow quadratically with respect to the clonal data size.

Data structures that are scalable in terms of memory and simultaneously allow time-efficient retrieval need to be explored. Also, the clonal data set is typically sparse; characterized by a large number of 0's and very few 1's in the clonal signature. Sparse matrix representation techniques in the context of representation of clonal data need to be explored to allow for more efficient storage and consequently more scalable algorithms.

Acknowledgments

The support of the National Science Foundation (NSF BIR 94–22896, CCR–8717033 and CDA–8820544) is gratefully acknowledged. The authors wish to thank Professor Jonathan Arnold, Department of Genetics, University of Georgia, Athens, Georgia for providing the clonal data set. The authors also wish to thank the anonymous reviewers for their insightful comments and suggestions which greatly improved the paper.

References

1. E.H.L. Aarts, F.M.J. de Bont, J.H.A. Habers and P.J.M. van Laarhoven, A Parallel Statistical Cooling Algorithm, *Lecture Notes in Computer Science: Proc. 3rd Annual Symp. Theoretical Aspects of Computer Science*, Vol. 210, pp. 87–97, Springer–Verlag Inc., Berlin, Germany, 1986.
2. E.H.L. Aarts and K. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*, Wiley, New York, 1989.
3. R. Azencott, (Ed.), *Simulated Annealing: Parallelization Techniques*, John Wiley Inc., New York, NY, 1992.
4. P. Banerjee, M. H. Jones and J. S. Sargent, Parallel Simulated Annealing Algorithms for Cell Placement on the Hypercube Multiprocessor, *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, pp. 91–106, Jan. 1990.
5. B. Bellane–Chantelot, B. LaCroix, P. Ougen, A. Billault, S. Beaufils, S. Bertrand, I. Georges, F. Gilbert, I. Gros, G. Lucotte, L. Susini, J.J. Codani, P. Gernouin, V.G. Pook, J. Lu-Kuo, T. Ried, D. Ward, I. Chumakov, D. Le Paslier, E. Barilott, and D. Cohen, Mapping the Whole Human Genome by Fingerprinting Yeast Artificial Chromosomes, *Cell*, Vol. 70, pp. 1059–1068, 1992.
6. W. Bender, M. Akam, F. Karch, P.A. Beachy, M. Peiffer, P. Spierer, E.B. Lewis, and D.S. Hogness, Molecular Genetics of the Bithorax Complex in Drosophila Melanogaster. *Science*, Vol. 221, pp. 23–29, 1983.
7. E. Bonomy and J.L. Lutton, N-City Traveling Salesman Problem and Metropolis Algorithm, *SIAM Rev.* Vol. 26 No. 84 pp. 551–568, 1984.
8. H. Brody, J. Griffith, A.J. Cuticchia, J. Arnold, and W.E. Timberlake, Chromosome-specific Recombinant Libraries from the Fungus *Aspergillus nidulans*, *Nucleic Acids Res.* Vol. 19, pp. 3105–3190, 1991.
9. A. Casotto, F. Romeo and A. Sangiovanni–Vincentelli, A Parallel Simulated Annealing Algorithm for the Placement of Macro Cells, *IEEE Trans. Computer-Aided Design*, pp. 838–847, Sept. 1987.
10. D. Cohen, I. Chumakov, and J. Weissenbach, A First-Generation Physical Map of the Human Genome, *Nature*, Vol. 366, 698–701, 1993.
11. F.S. Collins, M.L. Drumm, J.L. Cole, W.K. Lockwood, G.F. Vande Woude and M.C. Ianuzzi, Construction of a General Human Chromosome Jumping Library with Application to Cystic Fibrosis, *Science*, Vol. 235, pp. 1046–1049, 1987.
12. M. Creutz, Microcanonical Monte Carlo Simulation, *Physics Review Letters*, Vol. 50, No. 19, pp. 1411–1414, May 1983.
13. A. J. Cuticchia, J. Arnold and W. E. Timberlake, The Use of Simulated Annealing in Chromosome Reconstruction Experiments Based on Binary Scoring, *Genetics*, Vol. 132, pp. 591–601, Oct. 1992.
14. A.J. Cuticchia, J. Arnold, and W.E. Timberlake, ODS: Ordering DNA Sequences - A Physical Mapping Algorithm Based on Simulated Annealing, *CABIOS*, Vol. 9, No. 2, pp. 215–219, 1993.
15. E. Felten, S. Karlin and S.W. Otto, The Traveling Salesman Problem on a Hypercubic MIMD Computer, *Proc. IEEE International Conference Parallel Processing*, pp. 6–10, 1985.

16. M.S. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP–Completeness*, W.H. Freeman, New York, NY, 1979.
17. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam, *PVM Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA, 1994.
18. S. Geman and D. Geman, Stochastic Relaxation, Gibbs Distribution and the Bayesian Restoration of Images, *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 6, pp. 721–741, 1984.
19. D.R. Greening, Parallel Simulated Annealing Techniques, *Physica D*, Vol. 42, pp. 293–306, 1990.
20. R. Jayaraman and R. Rutenbar, Floor Planning by Annealing on a Hypercube Multiprocessor, *Proc. IEEE Intl. Conf. Computer Aided Design*, pp. 346–349, Nov. 1987.
21. Y. Kim and M. Kim, A Stepwise Overlapped Parallel Annealing Algorithm on a Message Passing Multiprocessor System, *Concurrency: Practice and Experience*, Vol. 2, No. 2, pp. 123–148, 1990.
22. S. Kirkpatrick, C. Gelatt Jr. and M. Vecchi, Optimization by Simulated Annealing, *Science*, Vol. 220, No. 4598, pp. 498–516, May 1983.
23. P.J.M. van Laarhoven and E.H.L. Aarts, *Simulated Annealing: Theory and Applications*, D. Reidel Pub. Co., Dordrecht, Holland, 1987.
24. E.S. Lander, and P. Green, Construction of multi-Locus Genetic Linkage Maps in Humans. *Proc. Natl. Acad. Sci.*, Vol. 84, pp. 2363–2367, 1987.
25. F.H. Lee, *Parallel Simulated Annealing on a Message-Passing Multicomputer*, Ph.D. dissertation, Dept. of Electrical Engineering, Utah State University, Logan, UT, 1995.
26. S. Lin and B. Kernighan, An Effective Heuristic for the Traveling Salesman Problem, *Operations Research*, Vol. 21, pp. 498–516, 1973.
27. E. Maier, J.D. Hoheisel, L. McCarthy, R. Mott, A.V. Grigoriev, A.P. Monaco, Z. Larin, and H. Lehrach, Yeast Artificial Chromosome Clones Completely Spanning the Genome of *Schizosaccharonyces Pombe*, *Nature Genetics*, Vol. 1, pp. 273–277, 1992.
28. E.M. Meyerowitz, *Arabidopsis*, a Useful Weed, *Cell*, Vol. 47, pp. 845–850, 1989.
29. S. Nahar, S. Sahni and E. Shragowitz, Simulated Annealing and Combinatorial Optimization, *International Journal of Computer Aided VLSI Design*, Vol. 1, pp. 1–23, 1989.
30. C.L. Smith and R.D. Kolodner, Mapping of *Escherichia coli* Chromosomal Tn5 and F Insertions by Pulsed Field Gel Electrophoresis, *Genetics*, Vol. 119, pp. 227–236, 1988.
31. A.H. Sturtevant, The linear arrangement of six sex-linked factors in Drosophila as shown by their mode of association, *Jour. Exp. Zool.*, Vol. 14, pp. 43–49, 1913.
32. V. Sunderam, PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*. Vol. 2, No. 2, pp. 315–339, 1990.
33. W. E. Timberlake, Molecular Genetics of *Aspergillus* Development, *Ann. Rev. Genetics*, Vol. 24, pp. 5–36, 1990.
34. Y. Wang, R.A. Prade, J. Griffith, W.E. Timberlake, and J. Arnold, A Fast Random Cost Algorithm for Physical Mapping. *Proc. Natl. Acad. Sci.* Vol. 91, pp. 11094–11098, 1994.
35. E.E. Witte, R.D. Chamberlain and M.A. Franklin, Parallel Simulated Annealing using Speculative Computation, *IEEE Trans. Parallel and Distributed Systems*, Vol. 2, No. 4, pp. 483–494, Oct. 1991.
36. C.P. Wong and R.D. Fiebrich, Simulated Annealing-based Circuit Placement on the Connection Machine System, *Proc. Intl. Conf. Computer Design*, pp. 78–82, Oct. 1987.
37. M. Xiong, H.J. Chen, R.A. Prade, Y. Wang, J. Griffith, W.E. Timberlake, and J. Arnold, On the Consistency of a Physical Mapping Method to Reconstruct a Chromosome *In Vitro*, *Genetics*, Vol. 142, No. 1, pp. 267–284, 1996.
38. P. Zhang, E.A. Schon, S.G. Fischer, E. Cayanis, J. Weiss, S. Kistler, and P.E. Bourne, An Algorithm Based on Graph Theory for the Assembly of Contigs in Physical Mapping of DNA, *CABIOS*, Vol. 10, pp. 309–317, 1994.