View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Lehigh University: Lehigh Preserve

Lehigh University
Lehigh Preserve

Theses and Dissertations

1999

Run-time analysis of parallel genetic algorithms for protein folding on the 2-D HP model

Lin Lu
Lehigh University

Follow this and additional works at: http://preserve.lehigh.edu/etd

Recommended Citation
Lu, Lin, "Run-time analysis of parallel genetic algorithms for protein folding on the 2-D HP model" (1999). *Theses and Dissertations*. Paper 622.

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

Lu, Lin

Run-Time Analysis of Parallel Genetic Algorithms for Protein Folding on the 2-D HP Model

January 2000

RUN-TIME ANALYSIS OF PARALLEL GENETIC ALGORITHMS FOR PROTEIN FOLDING ON THE 2-D HP MODEL

by

Lin Lu

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering and Computer Science

Lehigh University

December 1999

This thesis is accepted in partial fulfillment of the requirements for the degree of Master of Science.

Dec. 6th, 1999

(Date)

Prof. Eunice E. Santos
(Thesis Advisor)

Prof. Bruce D. Fritchman
(Chairman of EECS)

Acknowledgments

This thesis represents the culmination of 2 years and 4 months of study and research within the Department of Electrical Engineering & Computer Science at Lehigh University. I wish to express my warm thanks to those who have given me assistance and encouragement during my time here.

First, I would like to thank my advisor Dr. Eunice E. Santos. I am grateful for her guidance on my research and academics. Moreover, I was deeply impressed by our interesting discussions on diverse topics spanning culture, arts and history. I have been fortunate to have collaborated with Dr. Eugene Santos Jr., in Department of Computer Science & Engineering at the University of Connecticut. I wish to thank him for his very helpful support.

In addition, I am appreciative of my colleagues Mr. Murugiah Muraleetharan and Ms. Jirada Kuntraruk for creating the friendly atmosphere in our Parallel and Distributed Lab.

Finally, I would like to thank my family and, in particular, my father. From a young age, his help and encouragement has proved invaluable in the development

of my career.

The research work I have presented in this thesis has been with the collaboration of Dr. Eunice E. Santos and Dr. Eugene Santos Jr.. Moreover, I have been supported by a Lehigh Milestone Fellowship, an EECS Department Fellowship and the National Science Foundation.

Contents

Acknowledgments iii

List of Figures vii

Abstract 1

1 Introduction 2

1.1 Protein Folding on 2D-HP Model 4

1.2 Genetic Algorithm and Parallel Genetic Algorithm 7

1.3 LogP model 13

2 Sequential GA analysis 15

2.1 Analysis Notation 15

2.2 Sequential GA Analysis 16

3 Running-Time of Parallel Genetic Algorithms 17

3.1 Master-Slave Parallel Genetic Algorithms 17

3.1.1 Non-overlapped Single Master ms-PGAs 18

3.1.2 Overlapped Single Master ms-PGAs 20
3.1.3 Overlapped Multi-Master ms-PGAs 21
3.1.4 Evaluation 24
3.2 Fine-grained PGAs 24
3.2.1 Analysis 24
3.2.2 Evaluation 26
3.3 Coarse-grained PGAs 27
3.3.1 Classic Coarse-grained PGAs 29
3.3.2 Two New Coarse-grained PGAs 31
3.3.3 Evaluation 36

4 Implementation **38**
4.1 Performance Results 38
4.2 Evaluation 43

5 Conclusion and Future Work **44**
5.1 Conclusion 44
5.2 Future Work 47

Bibliography **49**

Biography **54**

List of Figures

1.1 An example of one conformation on 2-D HP. This protein molecule has an amino acid sequence of HPHPHHPHHHPHPPH. Here dot represents H and small circle represents P. 6

1.2 Main GA processing in a Genetic Algorithm 7

1.3 Crossover processing in a Genetic Algorithm (single-point crossover). 9

1.4 Mutation processing in a Genetic Algorithm 10

3.1 Token Ring Coarse-grained PGA 32

3.2 Loosely-coupled and Coarsely-coupled Coarse-grained PGA 34

4.1 Experimental Run-times when Protein Length=20 39

4.2 Experimental Run-times when Protein Length=36 40

4.3 Experimental Run-times when Protein Length=48 41

4.4 Experimental Run-times when Protein Length=64 42

Abstract

In this thesis, we study the problem of using parallel genetic algorithms (PGAs) for solving the protein-folding problem on the 2-D HP model. Theoretical analysis of running time is derived for several well-known types of PGAs including: master-slave, fine-grained, coarse-grained and their variants based on LogP, a portable parallel model. From performance data gathered, the theoretical analysis presented have been shown to successfully predict the running times. Two new variants of coarse-grained PGAs, which based on a very simple topology (ring) are designed based on the intention of rapidly spreading valuable solutions. The implementation results have shown that they achieve the similar results as the classic coarse-grained PGAs which are based on completed connected graph.

Chapter 1

Introduction

Currently, one of the most important open problem in biochemistry is the problem of protein native structure prediction. A protein is a linear polymer molecule formed by 20 different kinds of natural amino acids. Under certain circumstances, the sequence of amino acids can be rotated and folded into different conformations. The protein molecule will continue transforming until it reaches a stable conformation, which is called a *native structure*, then it will be capable of performing its biological functions [11] (Dill 1995). Hence, the ability to foresee the native structure of a protein molecule simply by knowing its amino acids sequence is quite rewarding in the biochemical and biomedical areas.

The folding process is very complicated and the details of the folding are still not completely known, but it is believed that the native structure of a protein molecule corresponds to its minimum free energy state (the thermodynamic hypothesis [1]

(Anfinsen 1973)). So one way to approach predicting native structure of a protein molecule is to compute and find the global minimum free energy conformation. However, the processing is complicated and it has been shown that finding the lowest free energy conformation of a protein is an NP-Hard Problem [28] (Unger 1993).

Therefore, several models have been presented for the protein folding problem. Perhaps the most successful, best-studied model [8] (Chan and Dill 1993), is the well-known two-dimensional hydrophobic hydrophilic model, or 2D-HP model [18] (Dill 1990). In the 2-D HP model, a protein molecule is assumed to consist of only two kinds of amino acids: H (hydrophobic) and P (hydrophilic). All the amino acids have the same size and each of them is represented as a "bead" placed upon a crossing point on a 2-D lattice. The connection bonds have identical bond lengths and are perpendicular to each other. Each connection within the molecule is represented as a line. Thus, the conformation of the amino acid chain can be represented as a self-avoiding walk in the 2-D lattice.

To find the stable native state of a protein, ideally, we should compute all the possible conformations of the protein molecule, calculate the sum of the free energies and determine the global lowest free energy conformation. However, it is not practicable to do such an exhaustive search [12] (Fraenkel 1993). Since the number of conformations grows exponentially with the increasing chain length, it has been proven that protein folding on the 2-D HP model is NP-complete [9] (Crescenzi 1998) [2] (Berger 1998). Thus, the question is "How should one find the global optimum

without a thorough global search on the protein's conformation energy landscape?" Several approximate algorithms were presented, such as the chain growth algorithm [4] (Bornberg-Bauer 1997), fast protein folding approximate algorithms [15] (Hart 1995). The common methods for the protein folding problem utilize Monte Carlo techniques [26] (Shakhnovich 1991) [13] (Ming-Hong Hao 1995), simulated annealing, and genetic algorithms [29] (Unger 1993).

Among these methods, genetic algorithms have been found to be more effective than the rest on a simple lattice model [23] (Patton 1995). A genetic algorithm (GA) is an efficient search technique based on natural selection and population genetics. During the evolution processing, a GA tries to accumulate good solutions and reject poor ones to achieve better approximate results in a limited time period. Due to the huge amount of computation a GA needs to perform, parallel genetic algorithms (PGAs) were implemented in order to achieve more efficient running times.

The focus of this thesis is the run-time analysis of PGAs for protein folding on 2-D HP. Preliminary results will be published in [24] (Santos 2000).

The following sections are intended to provide further motivation of the research work in this thesis while also providing some essential backgrounds.

1.1 Protein Folding on 2D-HP Model

The 2D HP lattice model represents the general properties of globular proteins. It captures essential biochemical characters of protein molecules while still maintaining

simplicity. By assuming that the hydrophobic interaction is the dominant force in protein folding, a protein in the HP modal is simply symbolized as a specific sequence of 2 kinds of amino acids: hydrophobic (H) and hydrophilic (P) monomers, instead of the 20 amino acids which exist in nature. A protein conformation is represented by a self-avoiding walk on the 2D lattice with the restriction that no two amino acids can occupy the same position on the lattice. The folding movement of the chain is represented as a sequence of moves where each is encoded relative to the prior movement. The computation of the conformational energy is also simplified. Each interaction between two H monomers which are adjacent in space but not adjacent in the sequence, called an H-H bond, will be counted as providing a contact energy of -1. All interactions between any other bands will be counted as 0. Thus the total free energy of a protein molecule is the sum of the contact energy between every H-H bond.

Therefore, conceptually, in a 2D-HP model [16] (Hart 1996),

(1) all the types of amino acids are represented by a set A={H,P},

(2) protein instances are represented by a binary sequence consisting of H and P.

(3) an energy formula specifying how the conformational energy is computed by

$$E = \sum(e(a,b))$$

Let $e(a,b)$ stand for the contact energy between amino acid a and b, and $label(a)$ stand for amino acids a's position in the sequence. Then, we have

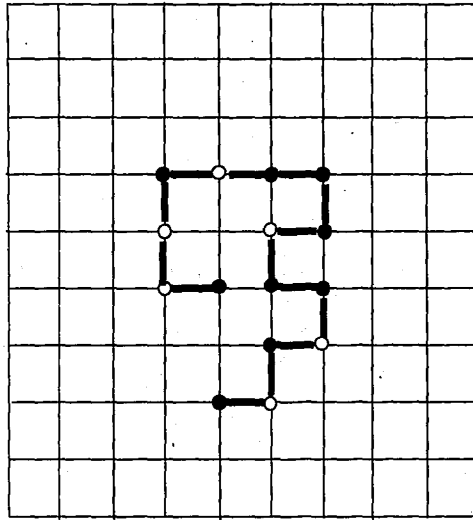


Figure 1.1: An example of one conformation on 2-D HP. This protein molecule has an amino acid sequence of HPHPHHPHHHPHPPH. Here dot represents H and small circle represents P.

$$e(a, b) = \begin{cases} -1 & \text{if } a = b = H, \text{ and } \mid label(a) - label(b) \mid \neq 1 \\ 0 & \text{otherwise} \end{cases}$$

(4) the conformation structure is presented as a self-avoiding walk on a 2D-lattice.

In this thesis, a movement walk is represented by a string consisting of E, S, W, N (standing for the four different directions on the 2-D lattice: east, south, west and north). Thus the example conformation shown in figure 1.1 on 2D-HP model can be represented by ENENWNENWWWSSE. Free energy of this conformation is 3.

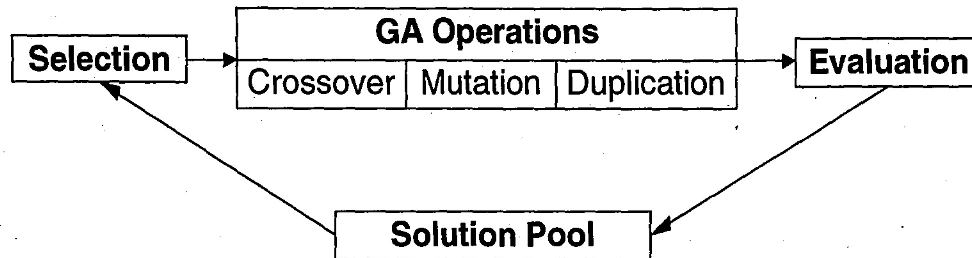


Figure 1.2: Main GA processing in a Genetic Algorithm

1.2 Genetic Algorithm and Parallel Genetic Algorithm

Genetic algorithms (GAs) utilize the same optimization procedures as natural genetic evolution. The whole process consists of a number of generations. In each generation, three phases are executed. These phases are called selection, genetic operation (duplication, crossover or mutation), and evaluation. There is no specific restriction on the number of generations. There have been several papers[17] (Hart 1995) which focus on evaluating the parameters in GAs in order to achieve more efficient overall run-times and faster convergence.

Genetic algorithms rely on appropriate encodings of potential solutions via string representations. At the beginning of the algorithm, an initial population which has a large amount of potential solutions has been created. Specific types of genetic operators are applied to this population. These genetic operators are typically the

common operators of crossover, mutation and replication. After those genetic operators are applied, new potential solutions will be produced. In order to set a criterion for choosing appropriate results, a fitness function will be computed. Solutions with the higher fitness function value will have a higher chance of reproducing. Apparently, the diversity of the population is important in order to maintain a large amount of individual solutions to ensure that many combined features may emerge and that solutions will not be trapped by local optima results. In order to keep a population's diversity, the genetic operators should be carefully chosen.

Since our goal is to utilize GAs to solve the protein folding problem, we will discuss standard GA operators as they are applied to protein folding on 2-D HP. Preliminary results have been presented in [24] (Santos 2000). For the protein folding problem, the potential solutions are the conformations of a protein molecule structure. In our implementation, a self-avoiding walk on the 2D-HP model is represented by a string consisting of E, W, N, S, (E-east, W-west, N-north, S-south), showing the direction for the each step of the walk on the 2-D lattice. The fitness function computes the free energy of the conformation. Hence, in a crossover operation, two parents' conformation strings are combined to make a child. Figure 1.3 provides an example in crossover. The digits in the string are not changed. Instead, the digits may be rearranged in different ways. The digit in the child sequence will keep the same position as the digit in the parent sequence. It is common knowledge that crossover is responsible for most of the diversity within the population.

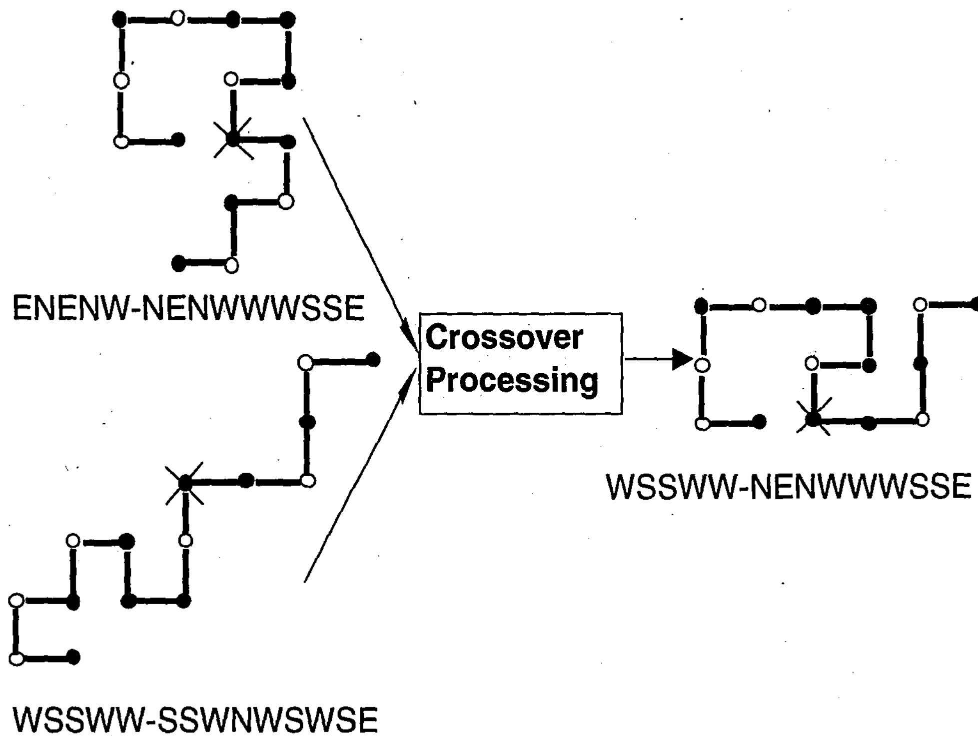


Figure 1.3: Crossover processing in a Genetic Algorithm (single-point crossover).

Mutation is another way to maintain the diversity of a population. It is a random change in a given digit in a string. Unlike crossover, it changes the digit of the string. Mutation is important for a genetic algorithm. It prevents the algorithm from getting trapped at a local optimal answer.

Utilizing the operations described above, we see that the basic steps of a genetic algorithm are comprised of the following [22, 21] (Muhlenbein 1991):

1. Generate an initial population of potential solutions.

2. Create new individuals by using crossover, and/or mutation genetic operators.

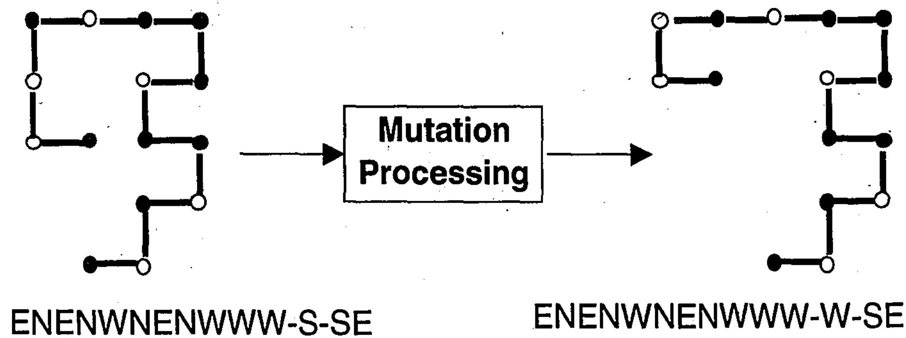


Figure 1.4: Mutation processing in a Genetic Algorithm

3. Determine the fitness value of each individual in the population.

4. If an acceptable solution is reached (based on its fitness value), report the solution and terminate. If better solutions are reached, update the solutions utilizing the new, better solutions in the population pool.

5. If the number of iterations exceeds a given value, report this and report the best solution that was reached. Otherwise return to step 2.

GAs have been successfully applied to solving several NP-complete problems, such as traveling salesman problem, scheduling in a job shop, mapping east Asia languages onto conventional keyboards etc. We note that genetic algorithms are not guaranteed to find an optimal solution. However, by choosing proper population size [22] (Giguere 1998) and utilizing more generations, it is clear that the good enough potential results may be achieved. However, more time will be needed. In order to get better results in a limited time, in the past few years, parallel genetic algorithms

have started to be utilized. GAs have been able to demonstrate an ability towards achieving better computational results with less overall computational time.

When we parallelize sequential GAs (sGAs) into Parallel Genetic Algorithms (PGAs), difficult problems can potentially be solved with less processing time [27] (Shonkwiler 1993) [22] (Muhlenbein 1991).

Due to different population storage methods, PGAs can be classified to multi-population PGA and global population PGA [6] (Cantu-Paz, Erick 1998). It also can be classified into three categories (based on different grain sizes): Master-slave, Fine-grained and Coarse-grained. In this thesis, we utilized the latter categorization.

1. In master-slave PGAs, only the process of computing fitness value is parallelized. Processors are divided such that there are a master and multiple slaves. The master processor assigns the fitness computation tasks to slaves and collects the results after the slaves finish computation. In this thesis, in order to solve the protein folding problem on 2-D HP, we will analyze and implement three different types of master-slave PGAs.

    - non-overlapping master-slave PGAs. (single-population storage and clearly division)
    - overlapping master-slave PGAs. (single-population storage, no clearly division)
    - overlapping multi master-slave PGAs. (multi-population storage, no clearly division)

2. In fine-grained PGAs, all three processes in sequential GA (selection, GA operation, evaluation) are parallelized. The population storage can be either shared (single population) or distributed (multiple population). In the distributed case, each processor can only perform GA operations either within its own subpopulation or from the sub-solution sent from its neighbors. Because of this restriction, strictly speaking, fine-grained PGA is not a typical GA. In this thesis, we analysis and implement the fine-grained PGA based on distributed memory on a 2D torus.

3. In coarse-grained PGAs, each processor runs sGA on its own subpopulation independently [19] (Shyh-Chang Lin 1994). In order to spread the results, data immigration among all the processors can occur every several generations.

In this thesis, we analysis and implement three types of coarse-grained PGA.

- Offset coarse-grained PGA: in immigration generation, one processor creates an offset randomly and broadcasts it to all the other processors. Each processor will communicate with the processor with the proper rank. (based on completed connected graph).

- Token coarse-grained PGA: the immigration data run through the whole distributed system like tokens. (based on a ring.)

- Loosely-coupled and tightly-coupled coarse-grained PGA: processors are binded to several tightly-coupled groups. Immigration happens frequently within the

tightly-coupled group, and less between the groups. There is a sender processor and a receiver processor in each group in charge of communicating with other groups (based on a ring).

There are also hybrid PGAs [20] (Merkle 1996), which combine the characteristics from those three PGAs.

In order to ensure that our analysis will be portable from one parallel/distributed machine/model/network to another, we must utilize a portable parallel machine model. In the next subsection, we discuss the parallel model we will use for our analysis.

1.3 LogP model

LogP [10] (Culler 1996) is a model of a distributed-memory multiprocessor in which processors communicate by point-to-point messages. The model specifies the performance characteristics of the interconnection network, without describing the structure of the network. The main parameters of the model are:

L: an upper bound on the latency, or delay, incurred in communicating a message containing a numerical value from its source module to its target module.

o: the overhead, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor can not perform arithmetic operations.

g: the gap, defined as the minimum time interval between consecutive message

transmissions or consecutive message receptions a message.

P: the number of processor/memory modules

Therefore, on the LogP model, sending a fixed sized message from one processor to another processor will require $2o + L$ time steps. All our protein folding PGAs will be analyzed on LogP.

Chapter 2

Sequential GA analysis

2.1 Analysis Notation

In order to analyze the various running times, we have assumed several notations provided below. Moreover, assuming a protein length of n, in our algorithm/implementation, we have been able to determine values dependent on n. And they are provided in square brackets, i.e. [ ].

- P_0 — population size;
- $g(p)$ — # of individuals selected per generation
- i —individual $i \in P$;
- $size(i)$—[n]— length of individual encoding;
- S_p—[$O(1)$]— running time of selecting one individual;

- F_o—$[O(n)]$— running time of a GA operation;
- C_f—$[O(n)]$— running time of calculating fitness function for one individual;
- G — the number of generations;
- $B(1, P_0)$—$[P_0]$—time of determining the best solution out of whole population.

We note that we assume $g(p) >> n$.

2.2 Sequential GA Analysis

The traditional sequential GA (denoted by sGA) operates on a fixed population. The main phases in an sGA include: selection, GA operations (crossover, mutation, duplication), and evaluation.

Hence, in a sequential GA the following steps are performed:

1) For each generation, the running time includes:

the time to select individual	S_p ;
the time to perform one GA operation	F_o ;
the time to call the fitness function	C_f .

2) After all the generations, the time of finding best solution so far should also be included $B(1, P_0)$

Therefore, the running time of a sequential GA is :

$T(sGA) = ((S_p + F_o + C_f)g(p))G + B(1, P_0) = O(ng(p)G + P_0).$

Chapter 3

Running-Time of Parallel Genetic Algorithms

In this chapter, we will design PGAs based on the three categories (master-slave, fined-grained, coarse-grained) discussed previously or on variants based on these categories. We will also provide analysis of parallel run-time under the LogP model. For ease of analyzing, we assume $L >> g$ and $g(p) >> L$ where L is the latency and g is the gap as described in the LogP model.

3.1 Master-Slave Parallel Genetic Algorithms

For master-slave PGAs (ms-PGAs), one processor works as the "master" while the remaining work as "slaves." The master selects individuals, and executes GA operations sequentially. Slaves evaluate fitness functions of the individuals in parallel.

The master is also in charge of distributing individuals to slaves and collecting results from the slaves.

For all analysis in this section, when a master assigns individuals to slaves, it delivers a message of size $size(i)$. However, when master collects fitness values from slaves, it delivers a message of size 1 (simply the fitness value).

3.1.1 Non-overlapped Single Master ms-PGAs

The first Master-slave PGA that we analyze is the non-overlapped, single-master ms-PGA. All the population is stored only in the master's memory. For each generation, the master finishes both selections, and GA operations on all the selected individuals. It then assigns them to the slaves. The master will be idle until all the slaves finish their fitness computations. Hence, slaves work concurrently, but master and slaves never do computation simultaneously. (There is some overlapped communication time between master and slaves).

We denote this GA by ms01PGA. Since there is explicit division between two generations, running time of each generation is identical. There are G iterations in the algorithm. In each generation, the master will select all the sampling individuals from the population pool and perform GA operations on them and assign them to all the slaves. Since the last slave ((P-1)th slave) is the last slave both for starting and ending computation, we will focus on the computation on the (P-1)th slave. Assumption: master will assign the individuals to the slaves equally proportionally.

Hence each slave will receive $\lceil \frac{g(p)}{(P-1)} \rceil$ individuals.

ALGORITHM:

G loops:

Master:

1) select individual $S_p \times g(p)$

2) do GA operation $F_o \times g(p)$

3) assign P-2 individuals to P-2 slaves $L + 2o + g \times ((P-2)size(i) - 1)$

Slave(P-1th):

$\lceil \frac{g(p)}{P-1} \rceil$ loops:

1) recieve individuals from master $L + 2o + g \times (size(i) - 1)$

2) compute the fitness values of the individuals

C_f

3) send fitness value to master $L + 2o$

Master:

1) find the best result to output $B(1, P_0)$

So, the running time is:

$$T(ms01PGA) = G \times [g(p) \times (S_p + f_o) + L + 2o + g \times ((P-2) \times size(i) - 1)$$
$$+\lceil \frac{g(p)}{P-1} \rceil (C_f + g \times (size(i) - 1) + 2L + 4o)] + B(1, P_0)$$
$$= O(g(p)Gn + \frac{g(p)LG}{P} + P_0)$$

3.1.2 Overlapped Single Master ms-PGAs

The second master-slave PGA that we will analyze is the single population overlapped ms-PGA. All the population is stored in the master processor. The computation phases will be overlapped with communication phases. Hence, there is no explicit generation here. When the master finishes a GA operation on a random selected individual, it will send the individual to one slave immediately then continue to select another individual to perform a GA operation. Thus, the master will not idle while waiting for results from slaves, (except the last (P-1)individuals) and will not wait for another generation to start. Therefore, master will not stop until it finishes performing GA operartions on all $G \times g(p)$ selected individuls.

We denote this GA by ms02PGA.

Assume

1) $(P-1) \times (g \times (size(i)-1) + o + S_p + f_o) <= C_f$,

2) master distributes individuals to slaves equally.

ALGORITHM:

Master (part 1):

$G \times g(p)$ loops:

1) select individual	S_p
2) do GA operation	F_o
3) distribute individual to slaves	$o + g \times (size(i) - 1)$

Master (after some steps):

1) collect the results from the slaves o

end $G \times g(p)$ loops:

Slave (the P-1th):

1) idle $L + (P-1) \times ((S_p + F_o) + o$

$+g \times (size(i) - 1))$

2) $\lceil \frac{Gg(p)}{(P-1)} \rceil$ loops:

1) receive $o + g \times (size(i) - 1)$

2) evaluate C_f

3) send o

Master(part 2):

1) gets the final result $L + o$

2) find the best result to output $B(1, P_0)$

In this algorithm, Master(part1) works simutaneously with (P-1)th slave, running time is:

$$T(ms02PGA) = (P-1) \times (S_p + f_o) + \lceil \tfrac{G \times g(p)}{P-1} \rceil \times C_f + (P-1) \times (o + g \times (size(i) - 1))$$
$$+ \lceil \tfrac{G \times g(p)}{P-1} \rceil \times (2o + g \times (size(i) - 1)) + 2L + o + B(1, P_0)$$
$$= O(\tfrac{g(p)Ggn}{P} + P_0)$$

3.1.3 Overlapped Multi-Master ms-PGAs

The last master-slave PGA that we will analyze is the overlapped, multi-master ms-PGA. In multi-master overlapped ms-PGA, the whole population is stored in

all the master processors distributedly. Each master processor can randomly select individuals from its local subpopulation and perform the GA operations, then it will make assignments to the slave processors. Since in our implementation, the slaves can be called by any of the masters, good load balancing among slaves should be considered. For each of the analysis, we assume distribution to slaves is again equally proportional such that task loading can be simply a one to one mapping. Since the total number of individuals is the same as single master mode, the slaves will have the same amount of individuals for fitness value computation. The only difference is that the masters' selection, GA operation, distribution and collection will run in parallel. After the masters finish all the operations, one best result of those masters should be chosen.

We assume P_m denotes the number of the masters. Therefore $P_m \times P = P_0$.

We denote this type of GA by ms03PGA.

Assume $(P - P_m) \times (g \times (size(i) - 1) + o + S_p + f_o) <= C_f$,

ALGORITHM:

Master (part 1):

$\frac{G \times g(p)}{P_m}$ loops:

1)select individual	S_p
2)do GA operation	F_o
3)distribute individual to slaves	$o + g \times (size(i) - 1)$

Master (after some steps):

1)collect the results from the slaves	o
end $G \times g(p)$ loops:	
Slave (the $P - P_m$th):	
1) idle	$L + (P - P_m) \times ((S_p + F_o) + o$ $+ g \times (size(i) - 1))$
2) $\lceil \frac{Gg(p)}{(P-P_m)} \rceil$ loops:	
1) receive	$o + g \times (size(i) - 1)$
2) evaluate	C_f
3) send	o
Master(part 2):	
1) gets the final result	$L + o$
2) find local best	$B(1, \frac{P_0}{P_m})$
3) find global best result:	$\lceil log(P_m) \rceil \times (1 + L + 2 \times o +$ $g \times (size(i) - 1))$

The total running time is:

$$T(ms03PGA) = (P - P_m) \times (S_p + f_o) + \lceil \frac{G \times g(p)}{P - P_m} \rceil \times C_f + (P - P_m) \times (o + g$$
$$\times (size(i) - 1)) + \lceil \frac{G \times g(p)}{P-1} \rceil \times (2o + g \times (size(i) - 1)) + 2L + o$$
$$+ B(1, \frac{P_0}{|P_m|}) + \lceil log(P_m) \rceil \times (1 + L + 2 \times o + g \times (size(i) - 1))$$
$$= O((\frac{g(p)Ggn}{P} + \frac{P_0}{|P_m|})$$

3.1.4 Evaluation

Analyzing our results, we see that all the varient msPGAs do not provide any significant benefit over sGA. In fact, since these are only asymptotic results, msPGAs may be even worse than sGA for protein folding using 2D-HP. For msP03GA, we see that there is the potential for fast running times especially when the gap g is small. In the master-slave category, the computation of fitness function is parallelized. However, in order to achieve overall parallelization, extra communication time may be necessary. Hence if the fitness function is not very difficult, or the interconnection network has heavy traffic, it is not efficient to use this master-slave mode.

3.2 Fine-grained PGAs

3.2.1 Analysis

In fine-grained GAs, a single population is distributed among the processors. Each processor selects individuals from itself or its neighbors, executes GA operations and evaluates the fitness function in parallel. We denote this type of GA by fPGA.

In general, assume every processor has m neighbors ($0 \leq m \leq P$), and select d individuals from each neighbor every t generations.

Using the same analysis notation defined in section 2.1, we can get that, for each processor p, p stores $\frac{P_0}{P}$ individuals in its local memory. Moreover, in each generation, $m \times d$ individuals are selected from remote processors, and $\frac{g(p)}{P} - m \times d$

individuals are selected locally.

For more accurate analysis, we have two more assumptions:

1. Each processor has a receiving buffer and a sending buffer.

2. $L + 2o + g \times (size(i) - 1) < (S_p + F_o + C_f)$

Since for every t generations, each processor will select individuals from its neighbors, we consider t generations as a "super-generation". Hence, there are $\frac{G}{t}$ super-generations in the algorithm. Within one super-generation, there are $t-1$ normal generations and one generation where communication will occur. Since the selection process is also parallelized, for each processor, only $\lceil \frac{g(p)}{P} \rceil$ individuals need to be selected from the local memory to perform the GA operations.

ALGORITHM:

$\frac{G}{t}$ loops:

1) t-1 loops:

$\lceil \frac{g(p)}{P} \rceil$ loops:

1) selection from local S_p

2) do GA operation f_o

3) do evaluate C_f

end $\lceil \frac{g(p)}{P} \rceil$ loops

end t-1 loops.

2) select d individual to sent $m \times d \times S_p$

send/receive d individual	$m \times d \times (o + g \times (size(i) - 1))$
3) select from local	$(\lceil \frac{g(p)}{P} \rceil - d) \times S_p$
4) do GA operation	$\lceil \frac{g(p)}{P} \rceil \times f_o$
5) evaluate	$\lceil \frac{g(p)}{P} \rceil \times C_f$
end $\frac{G}{t}$ loops.	
find its local best result :	$B(1, \frac{P_0}{P})$
find global best result:	$\lceil logP \rceil \times (1 + L + 2 \times o$ $+g(size(i) - 1))$

Analyzing running time, we see that:

$$T(fPGA) = G(\lceil \frac{g(p)}{P} \rceil (S_p + F_o + C_f) + mo\frac{d}{t} + \frac{d}{t}mg(size(i) - 1)) + B(1, \frac{P_0}{P})$$
$$+\lceil logP \rceil (1 + L + 2o + g(size(i) - 1))$$
$$= O(\frac{Gg(p)n}{P} + \frac{dGgnm}{t} + \frac{P_0}{P}).$$

3.2.2 Evaluation

If m is quite small, the result may not be very efficient, this is due to the fact that a good solution is not able to be spread quickly. It may more likely to be trapped in the local optimal traps. Because of the restriction of the individual selection (i.e. every processor can only pick the individual from itself or its neighbor), it is not a typical GA [6] (Erick 1998). For different topology interconnection network, different schedules of sending and receiving information are needed. If the topology is not a regular one or the processor has different numbers of neighbors, the schedule can

become more complex. So, the algorithms may not be portable, however it will work well on specific parallel machines, like CM-1, Maspar MP-1. In our implementation, fine-grained GA is based on a 2D torus, since many massively parallel computers utilize a 2D grid [6] (Erick 1998). It order to simplify the implementation, we utilize 2D torus to ensure every processsor will have same amount of neighbors. Thus for each of the processor, it can either select individual from its own local storage, or from its four neighbors' storage.

Analyzing our results against sequential GAs and multi-master-slave PGAs, we observe that the potential for much faster running times than those from sGAs is quite likely, especially if $\frac{d}{t} < \frac{Gg(p)n}{P}$. Moreover, further analysis shows that fPGA should run faster than msPGAs.

3.3 Coarse-grained PGAs

In coarse-grained GAs, every processor has its own sub-population and works as an sGA. Between any two sub-populations some solutions can be exchanged with each other [25] (Santos Jr. 1999).

Assume the number of immigration individuals is d, and that immigration will occur every t generations. Some parameters/factors for immigration have to be considered:

1. **The topology that connects the subpopulations**. In our implementation, the topologies are simple rings or completly connected graphs due to both

simplicity and portabity.

2. **The number of immigration individuals**. If immigration individuals are randomly chosen, it may lead to slow convergence. However, at the other extreme, if the immigration individuals are the best solutions of that sub-population, additional time is required to determine such best solutions. In our implementation, in order to have a tradeoff, we immigrate only one best solution with other randomly chosen solutions. Since we only chose one best solution, this means only one more comparison will be added for each evaluation. To simplify the analysis, we omit O(1) running times to pick up the best solution at this point.

3. **The frequency of immigration**. Typically, the first several generations require less immigration since it is not useful to spread solutions which have not be refined. However, as time passes, more immigration will be particularly useful in order to spread good solutions quickly. To simplify our theoretical analysis, the frequency of immigration is constant in our analysis and implementation.

For course-grained PGAs, there has been some research focus on providing accuracy of the results [14] (Hart 1996) [7] (Erick 1997). But, there has been little in the literature which provides in-depth discussion on how message-passing is implemented in general. This is an important criteria for determining parallel efficiency.

In this chapter, we provide three different methods for course-grained message-passing taking into account types of machines/models and/or networks.

3.3.1 Classic Coarse-grained PGAs

Our first algorithm is the classic coarse-grained PGA. Each processor has the possibility to communicate with any other processors. Its interconnection topology is a completely connected graph. During the immigration processing, one of the processors randomly creates an offset and broadcasts to all the other processors. Afterwards, each processor exchanges the information with the processor having the proper offset.

We denote this type of coarse-grained PGA by CPGA.

Similar to the case of fine-grained PGAs, we use the analysis notation in section 2.1, and assume that every t generations, d individuals are selected to immigrate. And also, assume each processor has a receiving buffer and a sending buffer.

Since every t generations, each processor will send/recieve d individuals from any other neighbor, we consider t generation as a super-generation. Hence, there are $\frac{G}{t}$ super-generations in the algorithms. Within one super-generation, there are $t-1$ normal generations and one generation where immigration occurs. Since the selection process is also parallelized, for each processor, only $\lceil \frac{g(p)}{P} \rceil$ individuals need to be selected to perform the GA operations.

ALGORITHM:

$\frac{G}{t}$ loops:

Step	Cost
In immigration generation:	
1) for processor 0:	
create random offset	1
broadcasts offset to P-1 processors	$o + (P-2) \times g$
for rest of processors:	
receive random offset	$L + o$
2) select d individual to sent	$d \times S_p$
send/receive d individual	$d \times (o + g \times (size(i) - 1))$
3) select from local (keep reserve part)	$\lceil \frac{g(p)}{P} \rceil - d$
4) do GA operation	$\lceil \frac{g(p)}{P} \rceil \times F_o$
5) evaluate	$\lceil \frac{g(p)}{P} \rceil \times C_f$
t-1 loops:	
1) select from local	$\lceil \frac{g(p)}{P} \rceil \times S_p$
2) do GA operation	$\lceil \frac{g(p)}{P} \rceil \times F_o$
3) evaluate	$\lceil \frac{g(p)}{P} \rceil \times C_f$
end $\frac{G}{t}$ loops.	
find its local best result :	$B(1, \frac{P_0}{P})$
find global best result:	$\lceil logP \rceil \times (1 + L + 2 \times o$ $+ g(size(i) - 1))$

The running time is:

$$T(CPGA) = G(\lceil \frac{g(p)}{P} \rceil (S_p + F_o + C_f) + o\frac{d}{t} + \frac{d}{t} g(size(i) - 1) + \frac{1+o+(P-2)\times g}{t})$$
$$+B(1, \frac{P_0}{P}) + \lceil \log P \rceil (1 + L + 2 \times o + g(size(i) - 1))$$
$$= O(\frac{Gg(p)n}{P} + \frac{dGg}{t} n + \frac{P_0}{P}).$$

3.3.2 Two New Coarse-grained PGAs

In this section, we introduce two new approaches for coarse-graned PGAs.

The purpose in designing these two algorithms is based on:

(1) spreading "good results" as soon as possible and as widely as possible, and

(2) efficiently communicating valuable information while still keeping the GA's random character.

In classic coarse-grained PGA, immigration individuals are ramdomly selected. It is quite possible that "good results" can not immigrate. Or even it immigrats once, it is hard to continue to immigrate to more processors. Thus, we design an algorithm, called token-ring coarse-grained PGAs, to ensure good result spreading to all the processors.

Token-ring Coarse-grained PGAs

Our second algorithm relies on "token-passing" along a directed ring of processors, instead of using point-to-point immigration method to transfer information. Token-passing is to allow good results to be continiually passed to processors in hopes for a faster result convergence. When a token reaches one processor, some part of the

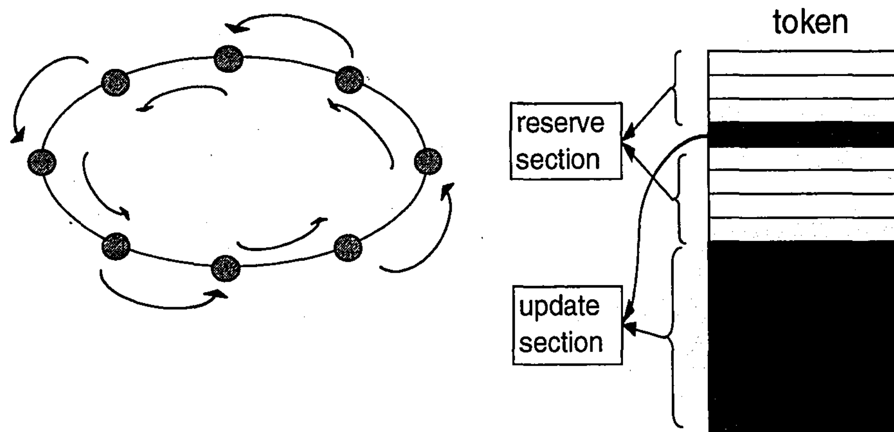


Figure 3.1: Token Ring Coarse-grained PGA

token will be updated, while the remaining portion will not be updated until it reaches the processor from which it originated.

We choose a ring since

(1) it is a simple topology which can be embedded into several other topologies, and

(2) it is straightforward to design a communication scheme among processors.

There are $\frac{P}{2}$ tokens being passed in the interconnection network at any time. Each token has two parts, one is "reserve part" (to be forwarded as is), the other is "update part" (to facilitate change). For every processor, when it receives the token, it will update the random part while keeping the reserve part to be delivered to the next processor. In addition, we divide the reserve part equally into P parts. When each processor receives the token, it will update its part in the reserve part.

This ensures that the reserve part will be refreshed every tP generations. Assume the reserve part is r in the token. When the processor receives the token, it needs to update $\frac{r}{P}+d-r$ of the individuals' information. We note that besides the operations on the token, the algorithm is similar to the fine-grained GA on distributed-memory with large t.

We refer to a token-passing course-grained PGA as TPGA.

The running time is:

$$\begin{aligned} T(TPGA) &= G(\lceil \tfrac{g(p)}{P} \rceil (S_p + F_o + C_f) + o\tfrac{d}{t} + \tfrac{d}{t} g(size(i) - 1) - \tfrac{\frac{(P-1)r}{P}}{t}) S_p) \\ &\quad + B(1, \tfrac{P_0}{P}) + \lceil \log P \rceil (1 + L + 2 \times o + g(size(i) - 1)) \\ &= O(\tfrac{Gg(p)n}{P} + \tfrac{dGg}{t}(n - \tfrac{Pr}{t}) + \tfrac{P_0}{P}). \end{aligned}$$

Loosely-coupled and Tightly-coupled Coarse-grained PGAs

Our third algorithm uses the concept of tightly-coupled and loosely-coupled. We form a group of processors whose latency between each other is much smaller than the average latency. We call this group of processors the tightly-coupled group. In each tightly-coupled group, there is one sending and one receiving processor which sends [receives] information to [from] other tightly-coupled groups. Communication among the tightly-coupled group is the same as in normal coarse-grained PGAs. We assume $\frac{P}{2}$ senders and $\frac{P}{2}$ receivers. Moreover, we denote L_0, o_0 and g_0 as the latency, overhead, and gap in the tightly-coupled group. Furthermore, t_1 denotes the rate of individuals exchanging between the tightly-coupled groups, and t_2 denotes the rate of individuals exchanging within a group.

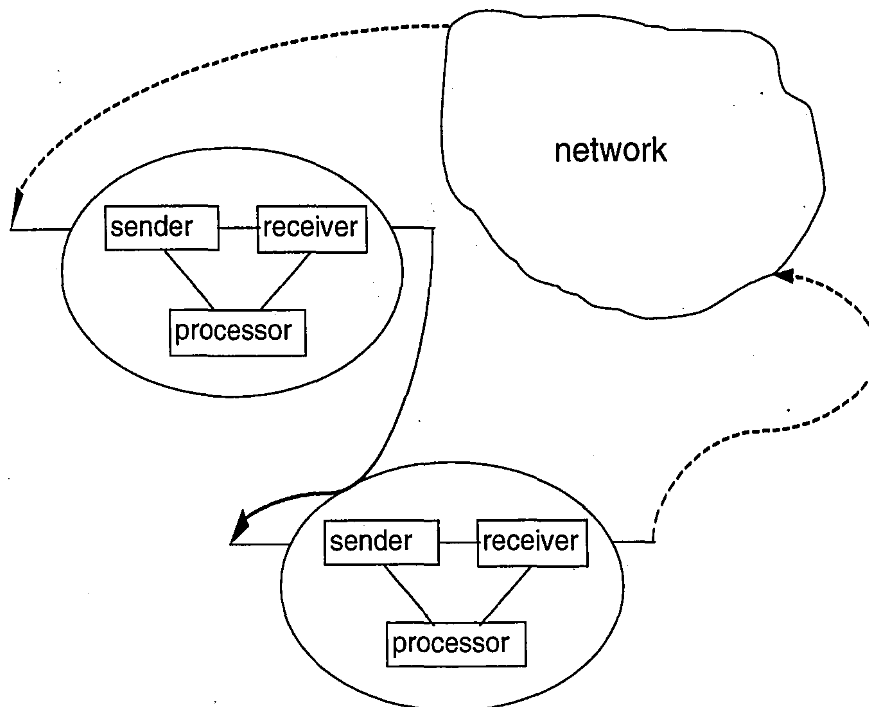


Figure 3.2: Loosely-coupled and Coarsely-coupled Coarse-grained PGA

Lastly, we denote this type of genetic algorithm by tPGA.

We combined $t_1 \times t_2$ generations as a phase, hence there are $\frac{G}{t_1 \times t_2}$ phases in the algorithm. In each one of the phases, there is one communication between tightly-coupled groups, and $t_2 - 1$ communication within the tightly-coupled group.

ALGORITHM:

in $\frac{G}{t_1 \times t_2}$ loops:

communication generation among tight-coupled groups:

1) select d individual to sent	$d \times S_p$
send/receive d individual	$d \times (o + g \times (size(i) - 1))$

2) select from local	$(\lceil \frac{g(p)}{P} \rceil - d) \times S_p$
3) do GA operation	$\lceil \frac{g(p)}{P} \rceil \times F_o$
4) evaluate	$\lceil \frac{g(p)}{P} \rceil \times C_f$

end communication generation among tight-coupled groups.

$t_2 - 1$ loops:

communication generation within tight-coupled groups:

1) select d individual to send	$d \times S_p$
send/receive to group member	$d \times (o_0 + g_0 \times (size(i) - 1))$
2) select from local	$(\lceil \frac{g(p)}{P} \rceil - d) \times S_p$
3) do GA operation	$\lceil \frac{g(p)}{P} \rceil \times F_o$
4) evaluate	$\lceil \frac{g(p)}{P} \rceil \times C_f$

end communication generation within tight-coupled groups:

$t_1 - 1$ loops:

1) select from local	$(\lceil \frac{g(p)}{P} \rceil) \times S_p$
2) do GA operation	$\lceil \frac{g(p)}{P} \rceil \times F_o$
3) evaluate	$\lceil \frac{g(p)}{P} \rceil \times C_f$

end $\frac{G}{t_1 \times t_2}$ loops

find local best result :	$B(1, \frac{P_0}{P})$
find global best result:	$\lceil logP \rceil \times (1 + L + 2 \times o$
	$+g(size(i) - 1))$

So in total, the running time is:

$$T(tPGA) = G(\lceil \frac{g(p)}{P} \rceil (S_p + f_o + C_f) + \frac{d}{t_1}(o + g(size(i) - 1)) + \frac{d}{t_2}(o_0 + g_0(size(i) - 1)))$$
$$+B(1, \frac{P_0}{P}) + \lceil logP \rceil (1 + L + 2o + g(size(i) - 1))$$
$$= O(\frac{Gg(p)n}{P} + \frac{dgnG}{t_1} + \frac{Gdg_0n}{t_2} + \frac{P_0}{P})$$

3.3.3 Evaluation

There is relatively little extra effort needed to convert a serial GA into a multiple population GA. The key point to design a good coarse-grained GA is to try to spread the good solution quickly and decrease the communication cost of the immigration. We have presented several ways to achieve this:

1. **Broadcast the best solutions.** Choose fewer immigration individuals and less immigration frequency. But all of them are the best solution of that sub-population.

2. **Choose some good solutions to be a "token" running in the whole system.** Every processor catches the token and replaces parts of the token with its good and randomly selected solutions.

3. **Create clusters including "receive" processors and "send" processors** which only deal with receiving or sending during the immigration communication to the other clusters.

Analyzing our results, we see that the token-passing course-grain PGA has the better asymptotic run-time. Therefore, the simple mechanics of passing information

along a ring seems to be efficient. When we compare the results against sGA, master-slave and fine-grained, we see that when t is large, the running time of fine-grained should match coarse-grained. Coarse-grained PGAs should run more efficiently compared to the remaining genetic algorithms analyzed.

Chapter 4

Implementation

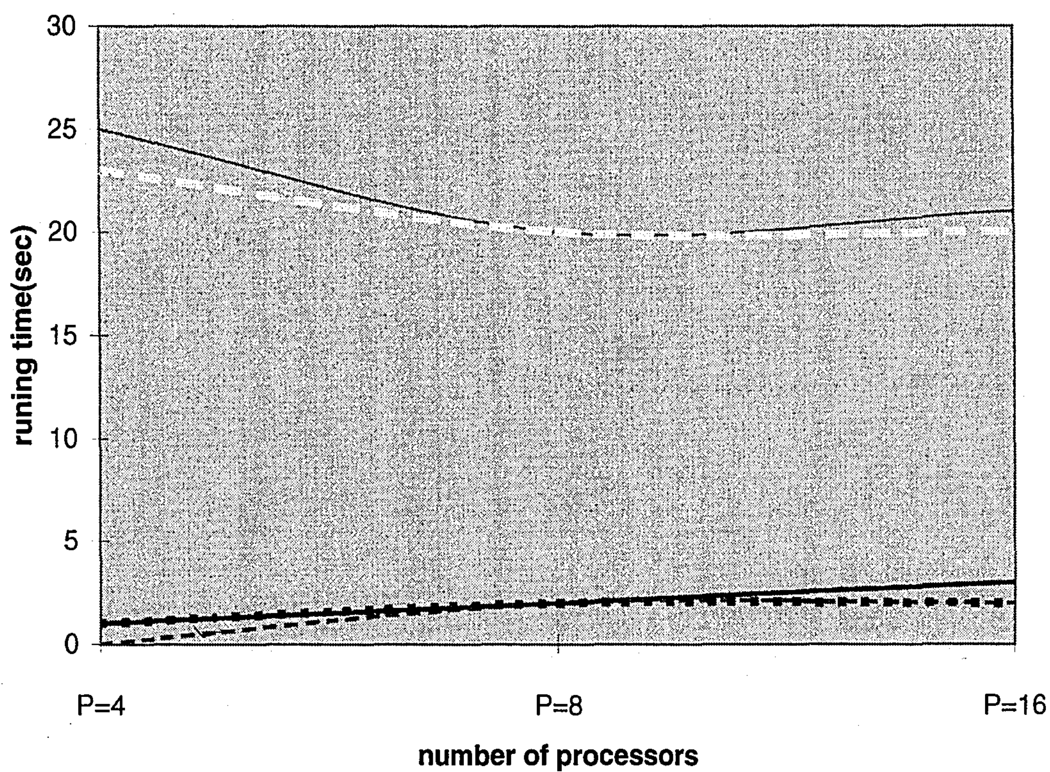
4.1 Performance Results

We implemented all the different PGAs we have described, using MPI on a cluster of Sparc Ultra1Workstations. [1] We ran simulations for proteins of length 20, 36, 48, and 64. The number of processors either physically or virtually ranged from 1 to 16. For each protein length, we provided two figures representing the amount of parallel execution time required for each PGA. The results appear in Figure4.1, 4.2, 4.3 and 4.4. We mapped a straight line in every graph to denote the running time of sGA. The test bed we used in our implementation comes from *http : //www.cs.sandia.gov/tech_reports/compbio/tortilla–hp–benchmarks.html.*

As we can see, our theoretical analysis predicted which genetic algorithm would be the most efficient. For small numbers of processors, there are shown that sGA

[1] The source code is available via email:lil3@eecs.lehigh.edu.

Protein Length=20



sGA
ms01GA
ms02GA
ms03GA
fGA
cGA
TGA
tGA

Figure 4.1: Experimental Run-times when Protein Length=20

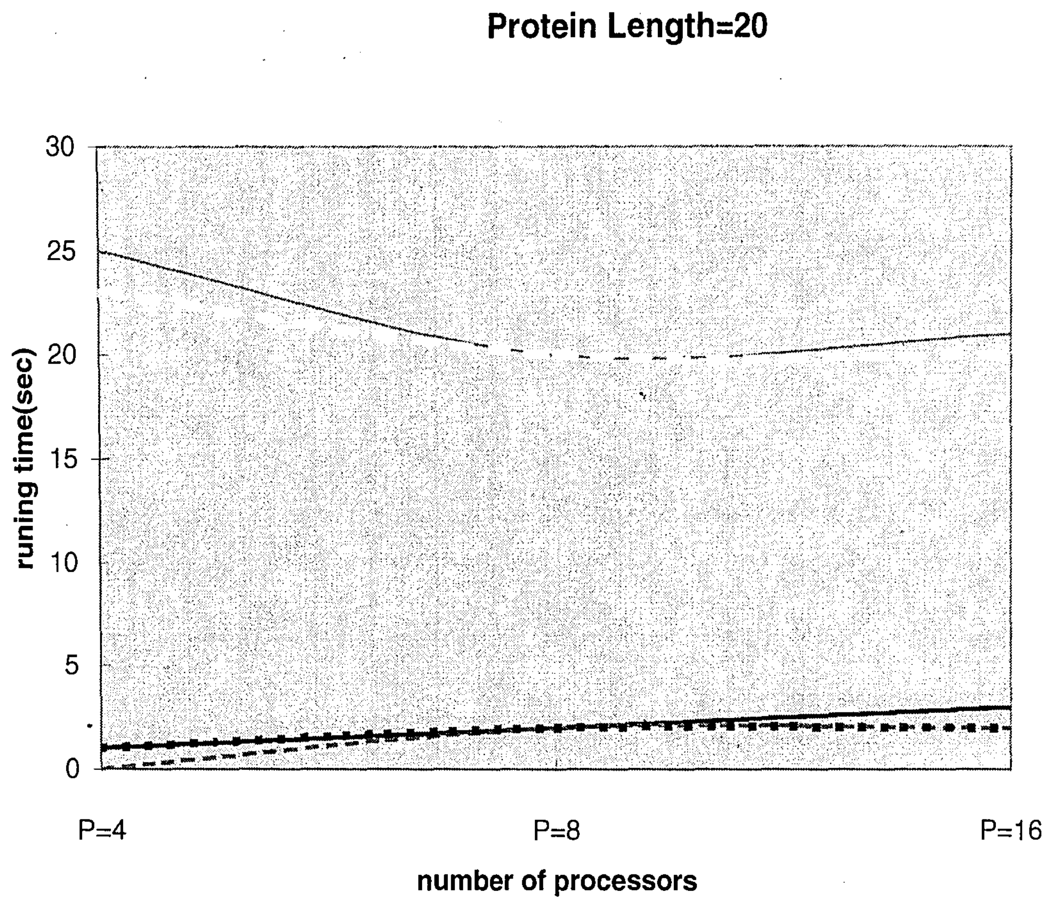


Figure 4.1: Experimental Run-times when Protein Length=20

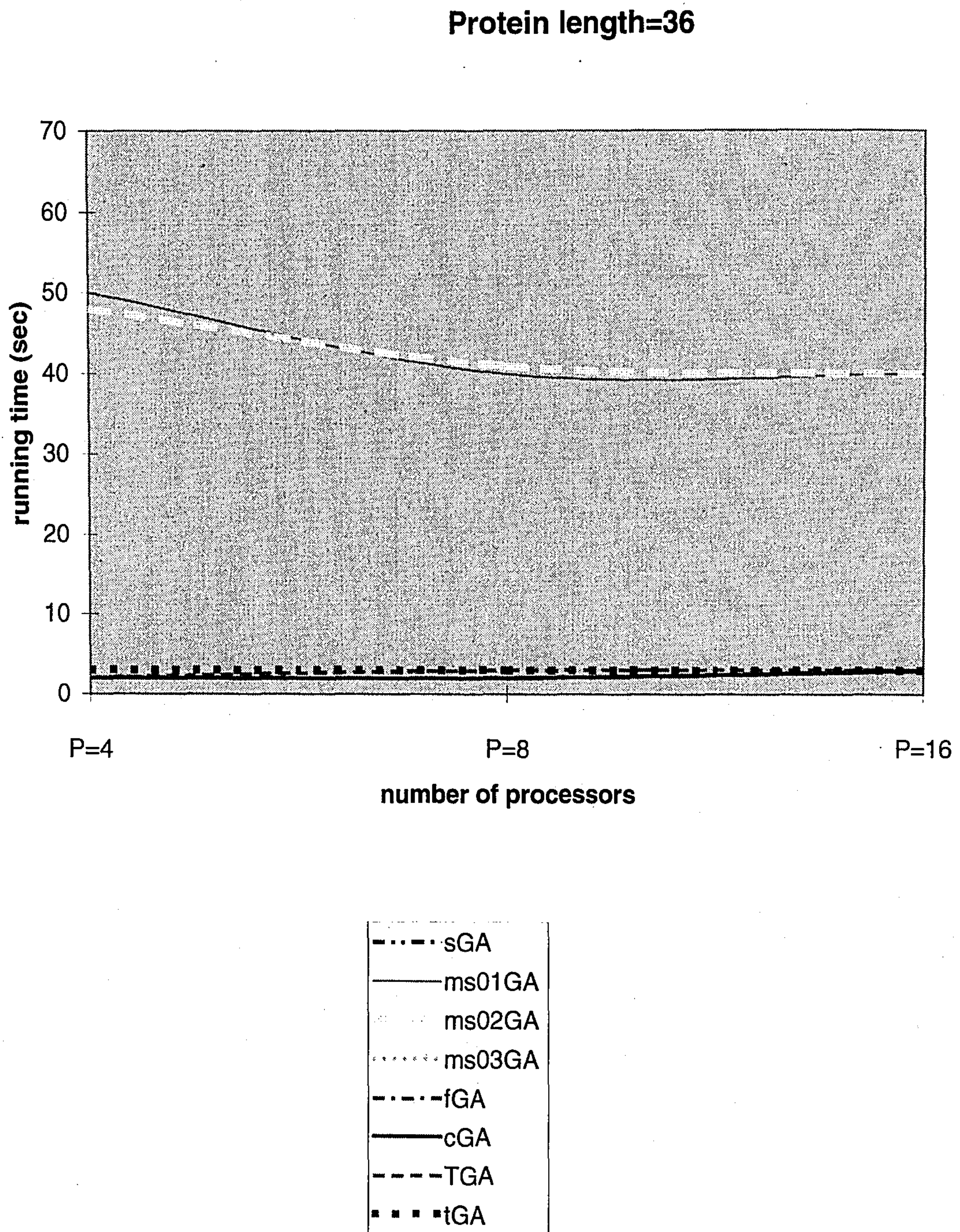


Figure 4.2: Experimental Run-times when Protein Length=36

Protein length=36

running time (sec)

70
60
50
40
30
20
10
0

P=4
P=8
P=16

number of processors

sGA
ms01GA
ms02GA
ms03GA
fGA
cGA
TGA
tGA

Figure 4.2: Experimental Run-times when Protein Length=36

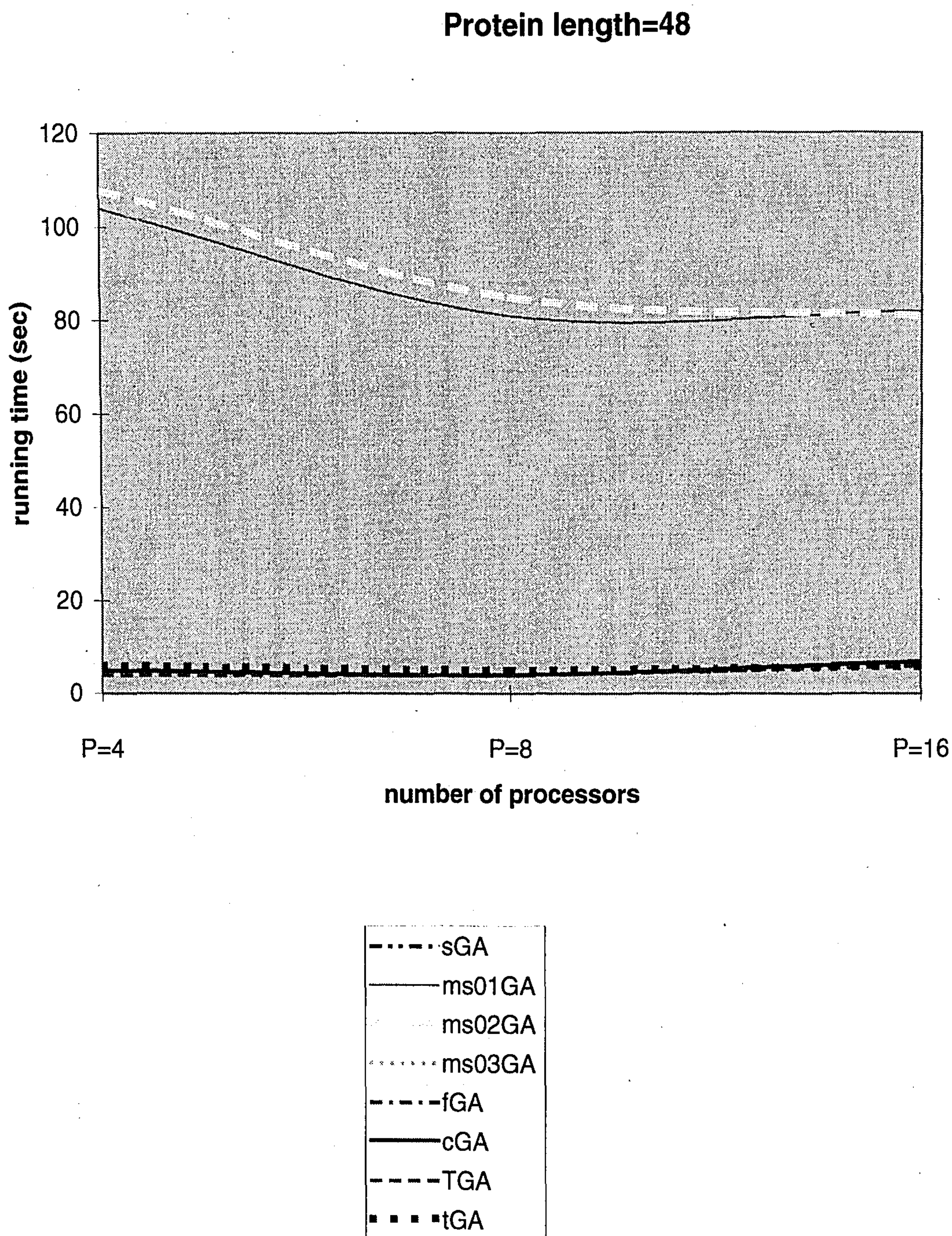
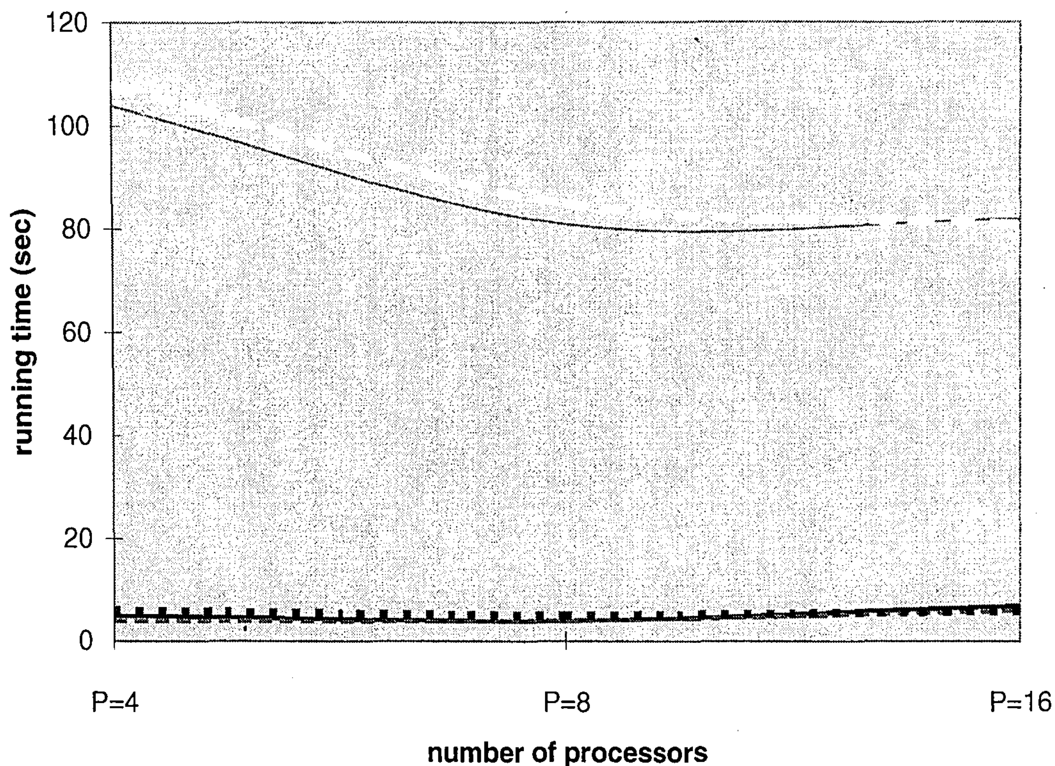


Figure 4.3: Experimental Run-times when Protein Length=48



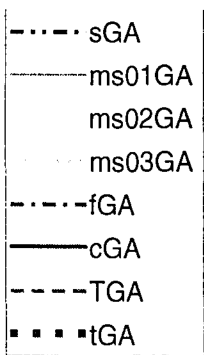


Figure 4.3: Experimental Run-times when Protein Length=48

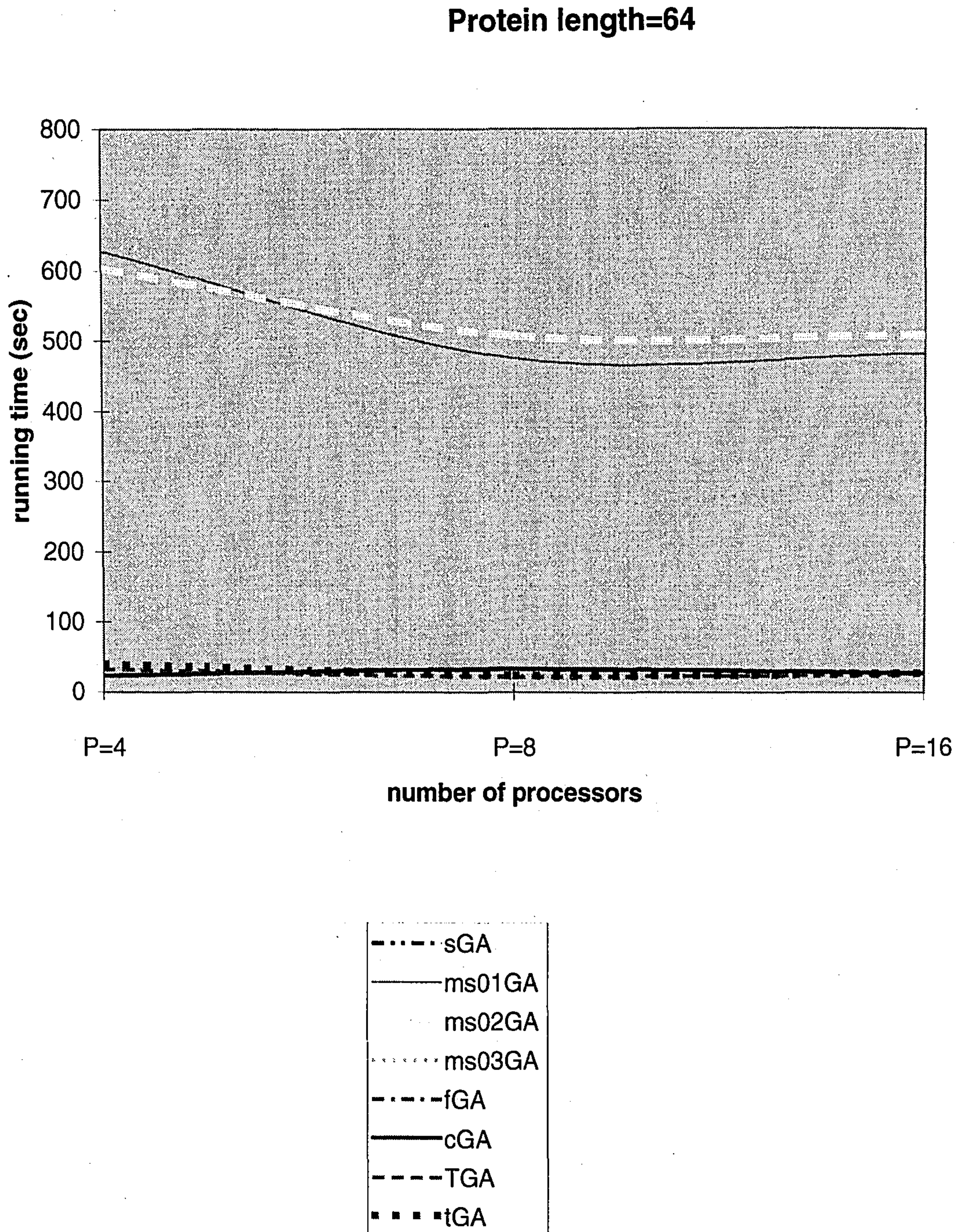
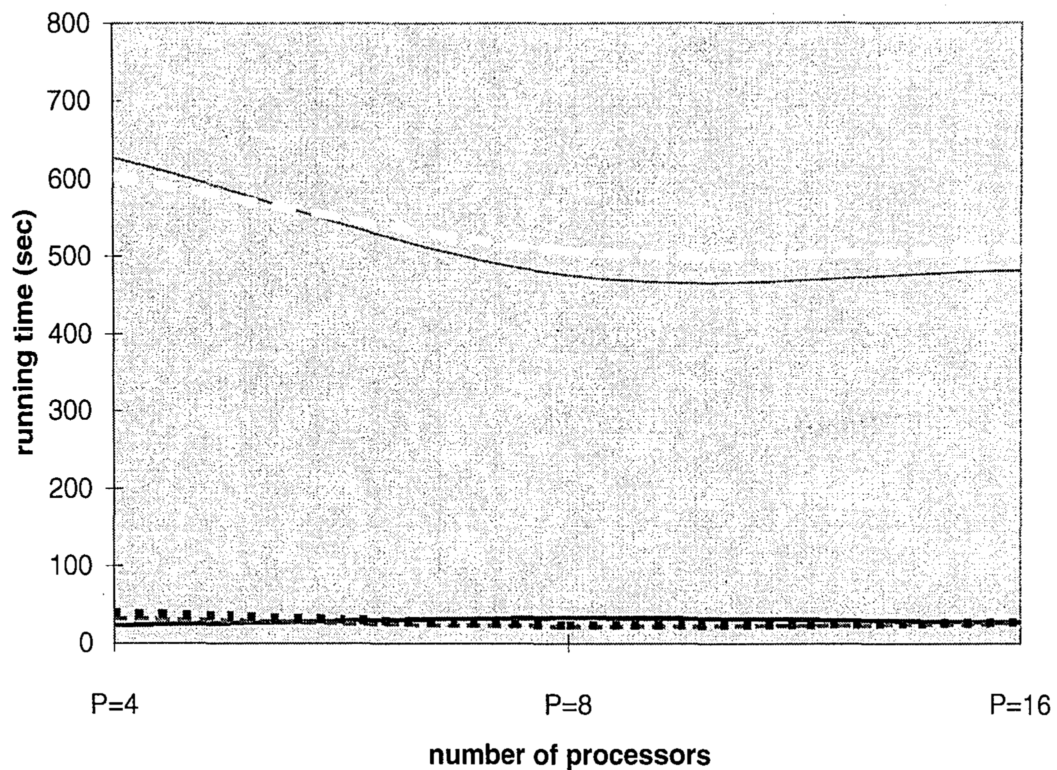


Figure 4.4: Experimental Run-times when Protein Length=64



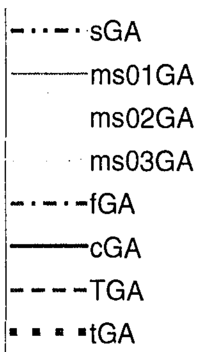


Figure 4.4: Experimental Run-times when Protein Length=64

is the most efficient. That is to be expected since the message time overheads for small numbers of processors have increased the overall running time, which makes a PGA potentially more inefficient than the sGA.

4.2 Evaluation

From these eight PGA's implementations, we find that:

1. Coarse-grained PGAs are much better than Master-slave PGAs and better than fine-gained PGAs when the length of the protein molecule is long.

2. Among the three types of Master-Slave PGAs, multi-master PGA works much better than single-master PGA. Actually, the multi-master PGA already has some similararities to coarse-grained PGA, except for the lack of communication between the masters.

3. When we used four/eight/sixteen processors to implement coarse-grained PGA, we found that when the length of protein molecule increased, its speedup was nearly optimal (in all four test cases: protein length equals 20, 36,48 and 64). This implied that good results that are spread quickly and widely truly aid PGAs in improvement of performance.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Protein folding is a vital problem in bio-chemistry. One of the most fundemental and well-known modls for protein folding is the 2-D HP model. In order to obtain solutions for protein folding on 2-D HP, several methods have been utilized. Genetic algorithms have been shown to be one of the most effective. Therefore, the focus of this problem has been the design and analysis of GAs, in particular parallel GAs, for protein folding for 2-D HP. Moreover, our results are extendible to other fundemental optimization problems.

Genetic algorithms have been implemented to solve various NP-hard problems. Moreover, they have proven to be quite efficient [3] (Bianchini 1995). Even so, in order to find good problem solutions, sGAs may require large amounts of time.

Much research has been done to consider how to parallizes sGAs into PGAs.

Clearly, sGAs are complicated algorithms due to the need to consider several parameters. Turning our attention to PGAs, we see that even more parameters must be considered based on the different design/implementation of each PGA. In fact, PGA designers or implementors must deal with determining the interaction of not only sGA parameters and parallel machine parameters, but also a new host of other parameters which include dealing with information exchange patterns across processors as well as acquiring further parameters for each specialized PGA (i.e. number of masters vs. number of slaves in multi-master msPGAs).

Since there is a multitude of parameters from which to consider, it is important to only choose those parameters which are deemed to be the most important for the PGAs at hand. This is due to the simple fact that while more parameters potentially create more realism, they will also create a reduction in the degree of usability. Furthermore, the GA community has focused their attention on a very important issue, mainly determining convergence rates for fast convergence of solutions. However, a key point that has been neglected is the design and theoretical analysis of PGA run-times and efficiencies. This fundamental point is the building block of the research presented in this thesis.

In this thesis, we theoretically designed and analyzed the running times of several fundamental parallel PGAs for the problem of protein folding on 2D-HP. The results we obtained from our implementation are consistant with our theorectical results. This shows that:

1. To solve protein folding on 2-D HP model, among all three PGA categories, coarse-grained PGAs are the most efficient (based on the same amount of processors, same test bed and same GA parameters: population size, the amount of individual sampling, crossover rate, mutation rate and number of generations).

2. our asymptotic growth rate results can give a direct gudiance on the parameter choosing for PGAs to solve protein folding on 2-D HP model.

3. researchers can predict the run-time performance of complicated parallel algorithms using realistic parallel models.

We also designed two new coarse-grained PGAs: token-ring and loosely-coupled and tightly-coupled. The motivation to design these two coarse-grained PGAs is based on:

1. spreading "good results" as soon as possible and as widly as possible to faster the result convergence on all the processor while still keepin the GA's random character.

2. maintaining efficient communication while also keeping simple implementation.

These two new coarsed-grained PGAs are base on a ring topology, which can be embedded into many other complicated toplogies. The implementation shows

that these two PGAs achieve as good results as the classic coarse-grained PGA which is based on a completely connected graph (based on same immigration rate, immigration size and same sampling technique).

5.2 Future Work

In order to obtain a more complete picture of coarse-grained PGAs, potential improvements include:

1. For token-ring coarse-grained PGA, when the number of the processor increases, ring become a too simple topology to spread the result because it takes long time to finish one round. We plan to expand token-ring into a 2D-torus, on each row and each column, there is a token-ring coarse-grained PGA running on it.

2. For loosely-couple and tightly-coupled PGA, since we only utilized processors which reside on the same LAN in implementation of the tightly-coupled PGAs, the results is very similar to the classic PGA. We plan to implement this algorithm on different LANs and potentially a mixed LAN/WAN configuration in the future.

Finally, since our theoretical analysis has successfully derived the run-time for PGAs to solve protein folding on a simple model, i.e. 2-D HP, we can now focus

attention towards more complicated and more realistic/accurate models, such as CHARMM (Chemistry at HARvard Macromolecular Mechanics) [5] (Brooks 1983).

Bibliography

[1] C. B. Anfinsen. Principles that given the folding of protein chains. *Science*, 181:223–230, 1973.

[2] B. Berger and T. Leighton. Protein folding in the hydrophobic-hydrophilic(hp) model is np-complete. *Journal of Computational Biology*, pages 27–40, 1998.

[3] R. Bianchini, C. M. Brown, M. Cierniak, and W. Meira. Combining distributed populations and periodic centralized selections in coarse-grain parallel genetic algorithms. In *Proceeding of the International Conference on Aritificial neural Networks and Genetic Algorithms*, April 1995.

[4] E. Bornberg-Bauer. Simple folding model for hp lattice proteins. In *Proceedings of Bioinformatics German Conference on Bioinformatics GCB '96*, pages 125–36. Springer-Verlag, 1997.

[5] B. R. Brooks, R.E. Bruccoleri, et al. Charmm: A program for macromolecular energy, minimization, and dynamics calulations. *Journal of Computational Chemistry*, 4(2):187–217, 1983.

[6] E. Cantu-Pax. A survey of parallel genetic algorithms. *Calculateurs Paralleles*, 10(2):141–171, 1998.

[7] E. Cantu-Paz and D. E. Goldberg. Predicting speedups of idealized bounding cases of parallel genetic algorithms. In *Proceedings of the Seventh International Conference on Genetic Algorithms*. Morgan Kaufmann, 1997.

[8] H. S. Chan and K. A. Dill. The protein folding problem. *Physics Today*, pages 24–32, February 1993.

[9] P. Crescenzi, D. Goldman, C. Papadimitriu, A. Piccolboni, and M. Yannakanis. On the complexity of protein folding. *Journal of Computational Biology*, 5(3):423–465, 1998.

[10] D. Culler, R. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. Logp: a practical model of parallel computation. *Communications of the Association for Computing Machinery*, 39(11):78–85, November 1996.

[11] K. A. Dill, S. Bromberg, K. Yue, K. M. Fiebig, D. P. Yee, P. D. Thomas, and H. S. Chan. Principles of protein folding-a persepctive from simple exact models. *Protein Science*, 4:561–602, 1995.

[12] A. S. Fraenkel. Complexity of protein folding. *Bulletin of Mathematical Biology*, 55(6):1199–1210, 1993.

[13] M. H. Hao and H. A. Scheraga. Computational approach to the statistical mechanics of protein folding. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, volume 1, pages 478–505, 1995.

[14] W. E. Hart, S. Baden, and S. Belew, R. K. Kohn. Analysis of the numerical effects of parallelisam on a parallel genetic algorithm. In *The 10th International Parallel Processing Symposium*, pages 606–612, 1996.

[15] W. E. Hart and S. Istrail. Fast protein folding in the hydrophobic-bydrophilic model within three-eighths of optimal. In *Proceedings of Twenty-seventh Annual ACM Symposium on Theory of Computing(STOC95)*, pages 157–68, 1995.

[16] W. E. Hart and S. Istrail. Invariant patterns in crystal lattices: Implications for protein folding algorithms. In *7th Combinaorial Pattern Matching Annual Symposium*, pages 288–303, 1996.

[17] W. E. Hart, T. E. Kammeyer, and R. K. Belew. The role of development in genetic algorithms. *Foundations of Genetic Algorithms*, 3:215–332, 1995.

[18] K. F. Lau and K. A. Dill. A lattice statistical mechanics model of the conformational and sequence spaces of proteins. *Macromolecules*, 22:3986–3997, 1989.

[19] S. C. Lin, W.F. Punch, and E. D. Goodman. Coarse-grain parallel genetic algorithms: categorization and new approach. In *Proceeding of Sixth IEEE Symposium on Parallel and Distributed Processing*, pages 28–37, 1994.

[20] L. D. Merkle, G. B. Lamont, G. H. Gates, and R. Pachter Jr. Hybrid genetic algorithms for minimization of a polypeptide specific energy model. In *Proceeding of 1996 IEEE International Conference on Evoltionary Comutation*, pages 396–400, 1996.

[21] H. Muhlenbein. Asynchronous parallel search by the parallel genetic algorithm. In *Proceeding of the third IEEE Symposium on Parallel and distributed processing*, pages 526–533, 1991.

[22] H. Muhlenbein. Evolution in time and space- then parallel genetic algorithm. *Foundations of Genetic Algorithms*, pages 316–337, 1991.

[23] A. L. Patton, W. F. Punch, and E.D. Goodman. A standard ga approach to native protein conforamtin prediction. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 574–581, July 1995.

[24] E. E. Santos, L. Lu, and E. Santos Jr. Efficiency of parallel genetic algorithms for protein folding on the 2-d hp model. In *The Third International Workshop on Frontiers in Evolutionary Algorithms*, 2000.

[25] E. Santos Jr., S. E. Shimony, and E. Williams. Solving hard computational problems through collections (portfolios) of cooperative heterogeneous algorithms. In *Proceedings of the 11th International FLAIRS Conference*, pages 356–360, 1999.

[26] E. Shakhnovich, G. Farztdinov, A. M. Gutin, and M. Karplus. Protein folding bottlenecks: A lattice monte carlo simulation. *Physical Review letters*, 67(12):1665–1668, September 1991.

[27] R. Shonkwiler. Parallel genetic algorithms. In *Proceeding of the Fifty International Conference on Genetic algorithms*, pages 199–205, 1993.

[28] R. Unger and J. Moult. Finding the lowest free energy conformation of a protein is an np-hard problem:proof and implications. *Bulletin of Mathematical Biology*, 55(6):1183–1198, 1993.

[29] R. Unger and J. Moult. Genetic algorithms for protein folding simulations. *Journal of Molecule Biology*, 231:75–81, 1993.

Biography

Lin Lu was born in Suzhou, China on July 7th, 1974, to Zhiyong Lu and Hangfan Zhu. She began to study in East China Normal University in 1992, and got her Bachelor Degree of Science in Computer Science in 1996. From 1996 to 1997, she worked as a system administrator in the Financial Department of Warner-Lambert Sino-US Suzhou Capsulgel Company. In 1997, she started her graduate program in Lehigh University. She expect to get her Master Degree of Science in Computer Science in January 2000.

END OF TITLE