VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

THE PARALLEL GENETIC ALGORITHM FOR MULTICORE SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. LUKÁŠ VRÁBEL

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PARALELNÍ GENETICKÝ ALGORITMUS PRO VÍCEJÁDROVÉ SYSTÉMY

THE PARALLEL GENETIC ALGORITHM FOR MULTICORE SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. LUKÁŠ VRÁBEL
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ JAROŠ
SUPERVISOR

BRNO 2010

Zadání práce

1. Seznamte se s genetickými algoritmy a jejich aplikacemi v oblasti numerické optimalizace.

2. Prostudujte paralelizaci algoritmů pro systémy se sdílenou pamětí.

3. Navrhněte novou techniku paralelizace genetického algoritmu pro systémy se sdílenou pamětí. Zaměřte se na maximální možné zrychlení daného algoritmu oproti sekvenční verzi.

4. Navrženou koncepci realizujte a experimentálně ověřte zrychlení oproti sekvenční verzi.

5. Zhodnoťte dosažené výsledky a diskutujte možné pokračování v projektu.

Abstrakt

Genetický algoritmus je optimalizačná metóda zameraná na efektívne hľadanie riešení rozličných problémov. Je založená na princípe evolúcie a prirodzeného výberu najschopnejších jedincov v prírode. Keďže je táto metóda výpočtovo náročná, bolo vymyslených veľa spôsobov na jej paralelizáciu. Avšak väčšina týchto metód je z historických dôvodov založená na superpočítačoch alebo rozsiahlych počítačových systémoch. Moderný vývoj v oblasti informačných technológií prináša na trh osobných počítačov stále lacnejšie a výkonnejšie viacjadrové systémy. Táto práca sa zaoberá návrhom nových metód paralelizácie genetického algoritmu, ktoré sa snažia naplno využiť možnosti práve týchto počítačových systémov. Tieto metódy sú následne naimplementované v programovacom jazyku C za využitia knižnice OpenMP určenej na paralelizáciu. Implementácia je následne použitá na experimentálne ohodnotenie rozličných charakteristík každej z prezentovaných metód (zrýchlenie oproti sekvenčnej verzii, závislosť konvergencie výsledných hodnôt od miery paralelizácie alebo od vyťaženia procesoru, . . .). V poslednej časti práce sú prezentované porovnania nameraných hodnôt a závery vyplývajúce z týchto meraní. Následne sú prediskutované možné vylepšenia daných metód vyplývajúce z týchto záverov, ako aj možnosti spracovania väčšieho množstva charakteristík na presnejšie ohodnotenie efektivity paralelizácie genetických algoritmov.

Abstract

Genetic algorithm is a powerful optimization and search method successfully used in practice to solve many different problems. Underlying concept is based on the evolutionary mechanics observed in nature. As the GAs are computationaly intense applications, it is natural that there are many efficient methods for parallelization of these algorithms. However, most of these methods deal with supercomputers or large computer clusters with specialized hardware, as these were the most common parallel architectures in the past. With modern-day computers the trend in personal computer design is also moving towards parallel architectures bringing small and cheap parallel multicore processors. That's why it is imperative to have efficient methods to exploit capabilities of this system. This document presents prototypes of new methods of parallel genetic algorithms designed especially for these multiprocessor computers with shared memory.

Klíčová slova

paralelní, genetický algoritmus, vícejádrové systémy, víceprocesorové systémy, OpenMP

Keywords

parallel, genetic algorithm, multicore, multiprocessor system, OpenMP

Citace

Lukáš Vrábel: The Parallel Genetic Algorithm for Multicore Systems, diplomová práce, Brno, FIT VUT v Brně, 2010

Citation

Lukáš Vrábel: The Parallel Genetic Algorithm for Multicore Systems, master's thesis, Brno, FIT VUT v Brně, 2010

The Parallel Genetic Algorithm for Multicore Systems

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jiřího Jaroše

. . . .
Lukáš Vrábel
May 26, 2010

Acknowledgment

I would like to thank Ing. Jiří Jaroš for information, ideas and help with the thesis.

Contents

Chapter 1

Introduction

As genetic algorithms are relatively computationaly demanding applications, it is essential to design the most optimal methods to reduce the time needed to find a good solution. One way to achieve this is to exploit new generation of cheap multicore and multiprocessor personal computers with shared memory.

The purpose of this document is to analyze advantages and bottlenecks of these computer systems, and also to explore various approaches to parallelizing GAs. Based on this analysis, a design of new methods of parallel genetic algorithm is presented. The goal of these methods is to achieve maximum possible speedup trying to utilize the strengths and avoid the weaknesses of shared memory multiprocessor systems, as well as to explore possible ways to paralellization of genetic algorithm.

These new methods and their variations are then implemented using low-level multithreading OpenMP library. This implementation is used to benchmark various aspects of these methods as speedup, fitness convergence, performance under heavy load, and effect of selection and migration variants on these aspects. Finally, the results of the experiments are discussed, and possible ways of future research based on the discussions are proposed.

1.1 Organization

This document is organized into three main parts:

First part describes history and basic principles and models of genetic algorithms in chapter two, introduces genetic operators of mutation, crossover and selection in chapter three and basic description and taxonomy of various approaches used to parallelize genetic algorithms in the fourth chapter.

Second part consisting of chapter five provides general information on parallel computing, hardware architectures realizing parallel computation (especially symmetric multiprocessing and multiprocessor systems with non-uniform memory access), its bottlenecks and introduction to theoretical background used to assess the speedup of parallel systems and algorithms.

The last part, consisting of chapters six to eight, presents design goals based on analysis of current multiprocessor systems. These goals are used to design and implement the new methods of parallel genetic algorithm. Methods are then analyzed and experimental results are gathered from multiple runs on multiprocessor systems of various configurations. Chapter seven presents results and conclusions based on these results. Discussions on conclusions and future research on the subject are proposed in chapter eight.

Chapter 2

Genetic Algorithms

2.1 Introduction

The genetic algorithm is optimization and adaptive heuristic search technique based on the evolutionary ideas of natural selection and genetics. GAs simulates processes in nature necessary for evolution, such as inheritance, mutation, selection, and crossover, that ensures the survival of the fittest. A GA allows a population composed of many individuals to evolve under specifed selection rules to a state that maximizes the *fitness* (i.e.,minimizes the cost function).[13]

Some of the advantages of a GA include that it

- Optimizes with continuous or discrete variables
- Doesn't require derivative information
- Simultaneously searches from a wide sampling of the cost surface
- Deals with a large number of variables
- Is well suited for parallel computers
- Optimizes variables with extremely complex cost surfaces (they can jump out of a local minimum)
- Provides a list of optimum solutions, not just one

These advantages are intriguing and produce stunning results when traditional optimization approaches fail miserably.

Of course, the GA is not the best way to solve every problem. Sometimes the traditional methods have been tuned to quickly find the solution of a well-behaved convex analytical function of only a few variables. For such cases the calculus-based methods outperform the GA, quickly finding the minimum while the GA is still analyzing the costs of the initial population. For these problems the optimizer should use the experience of the past and employ these quick methods. However, many realistic problems do not fall into this category. In addition, for problems that are not overly difficult, other methods may find the solution faster than the GA. The large population of solutions that gives the GA its power is also its disadvantage when it comes to speed on a serial computer - the cost function of each of those solutions must be evaluated. However, if a parallel computer is available, each processor can evaluate a separate function at the same time. Thus the GA is optimally suited for such parallel computations [13].

2.2 Methodology

Genetic algorithms are usualy implemented as a computer simulation in which a population of representations (called genotype or chromosomes) of possible solutions (called genomes) to an optimization problem evolves, thus moving toward better solutions. Usually, solutions are represented as binary strings (containig 0s and 1s), but other encodings are also possible. The evolution starts on a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. If the algorithm has terminated due to a maximum number of generations, a satisfactory solution may or may not have been reached [26].

A typical genetic algorithm requires two things to be defined:

- a genetic representation of the solution domain
- a fitness function to evaluate the solution domain

A standard representation of the solution is as an array of bits. Arrays of other types and structures can be used in essentially the same way. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, that facilitates simple crossover operation. Variable length representations may also be used, but crossover implementation is more complex in this case. Tree-like representations are explored in Genetic programming[26].

```
randomize current_generation
evaluate current_generation
while ( not ending_condition )
{
  select parents for reproduction from current_generation
  create offsprings using crossover and mutation
  evaluate offsprings
  create new_generation from offsprings and parents
  swap new_generation and current_generation
}
```

Figure 2.1: Pseudocode for general genetic algorithm

The fitness function is defined over the genetic representation and measures the quality of the represented solution. The fitness function is always problem dependent. For instance, in the knapsack problem we want to maximize the total value of objects that we can put in a knapsack of some fixed capacity. A representation of a solution might be an array of bits, where each bit represents a different object, and the value of the bit (0 or 1) represents whether or not the object is in the knapsack. Not every such representation is valid, as the size of objects may exceed the capacity of the knapsack. The fitness of the solution is the sum of values of all objects in the knapsack if the representation is valid, or 0 otherwise. In some problems, it is hard or even impossible to define the fitness expression[26].

Once we have defined the genetic representation and the fitness function, GA proceeds to randomly initialize a population of solutions, then improve the overall fitness through continuous application of crossover, mutation and selection operators. Diagram iluustrating this process is on figure 2.2 with corresponding pseudocode on figure 2.1.

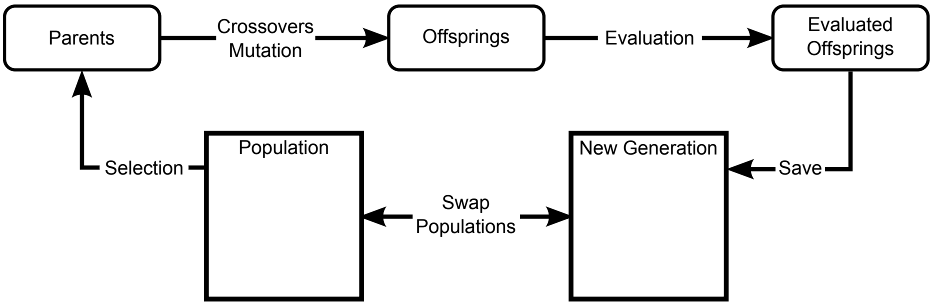

Figure 2.2: New generation created from offsprings of successful parents. After the new generation is filled, it becomes the current population.

2.3 Steady-State GA

The variant of genetic algorithm described in previous chapter is called *generational GA*, because the evolution proceeds in discrete steps called generations. On the other hand, the Steady-state genetic algorithm (SSGA) is model based on notion of continuous evolution.

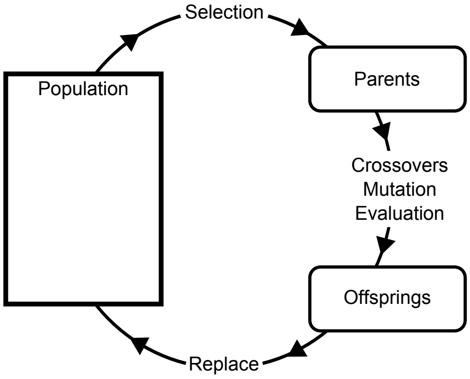

Figure 2.3: Evolution process of steady-state GA.

The difference from generational GA model is that in SSGA there are no generations, thus offsprings created by crossover and mutation get back to the population immediately. To keep the size of population constant, some of the genomes of current population must vacate a slot for these new offsprings. This process is called a *replacement strategy.* As current generation and new generation overlaps, offsprings compete with parents as soon as they enter the population [19]. Schematic diagram of SSGA evolution process is displayed on figure 2.3.

The replacement strategy is important factor of SSGA - it affects the population dynamics to large degree. Replacement of the worst individual is widely used strategy, and it has been shown that it induces high selection pressure (more on selections in section 3.3) even if the parents are selected randomly [19]. Analysis in [28] also shows that the decrease of population variance under steady-state GA is twice compared to the traditional GA working on generations. On the other hand, the evolution process requires half the computational steps of generational GA to achieve the same convergence [28]).

There are different replacement strategies that try to solve this problem by enforcing higher population diversity. Most of them belongs to the category of *crowding* methods, because they are based on idea that new genomes are more likely to replace similar individuals already present in the population. Thus the population will not accumulate a lot of genomes with similar characteristics, preserving the multiple local optima [19].

2.4 History

In the 1950s and the 1960s, there were several independent studies of evolutionary systems coducted by computer scientists with the idea that evolution could be used as an optimization tool for solving various engineering problems. The basic idea was to evolve a population containig candidate solutions to a given problem, using operators based on natural selection and genetic variation. [24]

Genetic algorithms (GAs) were invented in 1960s by John Holland and were developed in the 1970s by Holland and his colleadues and students the University of Michigan. In contrast with then established evolutionary programming and evolution strategies, Holland's original goal was to formally study the phenomenon of adaptation as it occurs in nature rather than design algorithms to solve specific problems. The main idea was to develop ways in which the mechanisms of natural adaptation might be imported into computer systems. Holland's 1975 book Adaptation in Natural and Artificial Systems presented the genetic algorithm as an abstraction of biological evolution and gave a theoretical framework for adaptation under the GA. Holland's GA is a method for moving from one population of *chromosomes* (for example strings of ones and zeros) to a new population by using a kind of *natural selection* together with the operators of crossover, mutation, and inversion, inspired by genetics. The selection operator chooses those chromosomes in the population that will be allowed to reproduce, and it ensures that on average, the fitter chromosomes produce more offspring than the less fit ones. Crossover combines subparts of two chromosomes, imitating biological recombination of genomes between two organisms. Mutation randomly changes the values of some locations in the chromosome, and inversion reverses the order of a contiguous section of the chromosome, thus rearranging the order in which genes are arrayed [24].

Holland's introduction of a algorithm based on population, with mutation, crossover and inversion was a major innovation. Moreover, Holland was the first who attempted to create a solid theoretical background for computational evolution. This theoretical foundation, based on the concept of *schemas*, was the basis for almost all of the theoretical research on GAs [24].

2.5 Genome Encoding and Problems

Although GAs typically represents chromosomes as a string of bits, they are not restricted to bitstrings. A number of early proponents of GAs developed GAs that use other representations such as real-valued parameters, permutations (also called ordered chromosome, used for example in *travelling salesman problem* or *scheduling problem*) and treelike hierarchies [1].

Even for binary strings, there is still a choice to be made as to which binary coding scheme to use for encoding a numerical parameters. Most of empirical studies have found that Gray code is produces superior results to the standard binary coding for the commonly used test problems[1]. One reason is that the standard encoding introduces Hamming cliffs - two numerically adjacent values may have bit representations that are many bits apart. This could be a problem if the fitness function in gradual to some degree i.e. small changes in the variables correspond to small changes in the function value. This is often the case for functions with numeric parameters (also referred to as *causality*).

As an example, consider a five-bit parameter, with a range from 0 to 31. If it is encoded using the standard binary coding, then 15 is encoded as 01111, whereas 16 is encoded as 10000. In order to move from 15 to 16, all five bits need to be changed. On the other hand, using Gray coding, 15 would be represented as 01000 and 16 as 11000, differing only in 1 bit.

When choosing an alternative representation, the choice of appropriate crossover operator is critical. For example: in case of real-valued parameters as chromosome representation, a possible crossover operator could take the parameter values of the two parents to define an interval from which a new parameter is chosen. As the GA makes progress, it will narrow the range over which it searches for new parameter values[1].

Chapter 3

Genetic Operators

This chapters briefly describes the genetic operators for most of the basic genome encodings. As mutation is often the most simple operation, it is described as first, followed by more complex crossover operator. Because crossover is usually the most important part of genetic algorithm, large part of this chapters deals with various means of recombining the parent genomes to create offsprings. The third and final section present some of the selection operators, their advantages and disadvantages.

3.1 Mutation

The main characteristic of mutation operators is that they operate on a single individual to produce a new individual. Most mutation operators with typical parameter settings are relatively likely to generate offspring close to the parent solution[1].

Mutation operators are normally understood to serve two primary functions. The first function is as an exploratory move operator, used to generate new points in the space to test. The second is the maintenance of variability in the *gene pool* - the set of genomes available to recombination in the population. This is important because most recombination operators generate new solutions using only genetic material available in the parent population. lf the range of gene values in the population becomes small the opportunity for recombination operators to perform useful search tends to diminish accordingly[1].

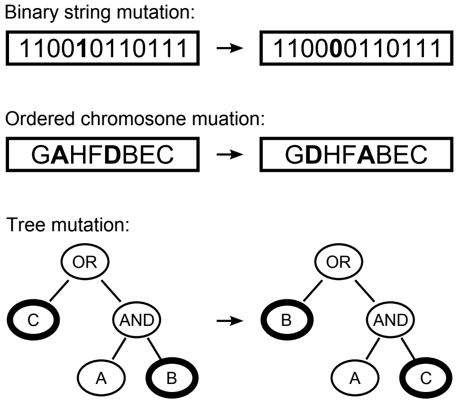

Figure 3.1: Example of mutation operation on genomes with different encodings

A common view in the GA community, dating back to Holland's book Adaptation in Natural and Artificial Systems, is that crossover is the major instrument of variation and innovation in GAs, with mutation insuring the population against permanent fixation at any particular locus and thus playing more of a background role[24]. Example of mutation operation on different genome encodings are on figure 3.1.

3.2 Crossover

The intuitive idea behind crossover is easy to state: given two individuals who are highly fit, but for different reasons, ideally what we would like to do is create a new individual that combines the best features from both. Of course, since we presumably do not know which features account for the good performance (if we did we would not need a search algorithm), the best we can do is to recombine features at random. This is how crossover operates. It treats these features as building block's scattered throughout the population and tries to recombine them into better individuals via crossover. Sometimes crossover will combine the worst features from the two parents in which case these children will not survive for long. But sometimes it will recombine the best features from two good individuals creating even better individuals provided these textures are collectible [1].

The success or failure of a particular crossover operator depends on the representation of the problem, its fitness function, encoding, and specific details of the GA. It is still a very important open problem to fully understand the interactions of these aspects and to what extent they affect the behaviour of GA. That is also the reason that there is no definitive guide to choosing the best type of crossover for a given problem. [24]

3.2.1 One-point crossover

One-point crossover is the simplest form: a single crossover position is chosen at random and the parts of two parents after the crossover position are exchanged to form two offspring (see figure 3.2). The basic idea is to recombine building blocks (schemas) of different genomess [24]. Single-point crossover has some shortcomings: it cannot combine all possible schemas, it often destroys schemas with longer defining lengths and the segments exchanged between the two parents always begin and end with the endpoints of the strings.

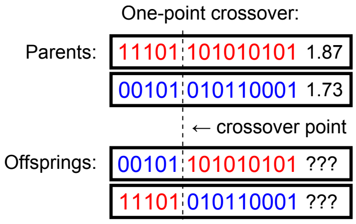

Figure 3.2: Example of One-point corrosover on binary string

3.2.2 Two-point crossover

In two-point crossover, two positions are chosen at random and the segments between them are exchanged (see figure 3.3). Two-point crossover is less likely to disrupt schemas with large defining lengths and can combine more schemas than single-point crossover. In addition, the segments that are exchanged do not necessarily contain the endpoints of the strings. Again, there are schemas that two-point crossover cannot combine [24].

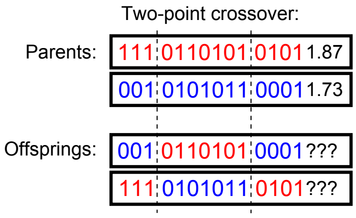

Figure 3.3: Example of Two-point corrosover on binary string

3.2.3 Uniform crossover

Some practitioners believe strongly in the superiority of parameterized uniform crossover, in which an exchange happens at each bit position with probability p (typically 0.5 or 0.7, see figure 3.4).

Parameterized uniform crossover has no positional bias. Any schemas contained at different positions in the parents can potentially be recombined in the offspring. However, this lack of positional bias can prevent coadapted alleles from ever forming in the population, since parameterized uniform crossover can be highly disruptive of any schema [24].

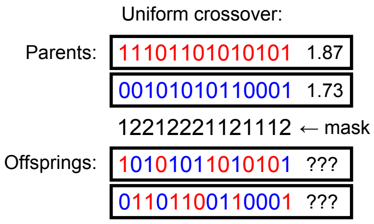

Figure 3.4: Example of Uniform corrosover on binary string using random mask for bit selection

3.2.4 Ordered chromosome crossover

When the chromosome is an ordered list, such as a list the cities to be travelled for the traveling salesman problem, a direct swap of genes may not be possible, because it would introduce duplicates and remove necessary candidates from the list. Instead, crossover point is selected on the parents. The chromosome up to the crossover point is retained for each parent, and the information after the crossover point is ordered as it is ordered in the other parent [26](for example see figure 3.5). *Note that there are more ways to crossover ordered chromosomes.*

3.2.5 Tree crossover

Usually used in genetic programming, where the genome is in fact a parse-tree of a computer program. The operation begins by independently selecting one random point in each parent

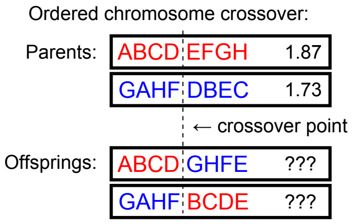

Figure 3.5: Example of Ordered chromosome corrosover with one crossover point

to be the crossover point for that parent. The crossover fragment for a particular parent is the subtree which has as its root the crossover point. The first offspring is produced by deleting the crossover fragment of the first parent from the first parent and then inserting the crossover fragment of the second parent at the crossover point of the first parent. The second offspring is produced in a symmetric manner [17] (see figure 3.6).

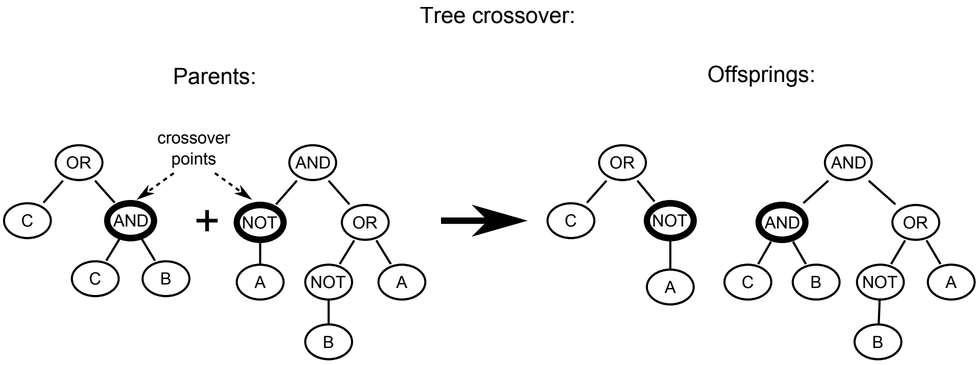

Figure 3.6: Example of Tree crrosover on logical expressions

3.3 Selection

Genetic algorithms use a selection mechanism to select individuals from the population to insert into a mating pool. Individuals from the mating pool are utilized as a parents to generate new offspring, with the resulting offspring forming the basis of the next generation. As the individuals in the mating pool are the ones whose genes are inherited by the next generation, it is desirable that the mating pool be comprised of "good" individuals. A selection mechanism in GAs is simply a process that favors the selection of better individuals in the population for the mating pool. The *selection pressure* is the degree to which the better individuals are favored: the higher the selection pressure, the more the better individuals are favored. This selection pressure drives the GA to improve the population fittness over succeeding generations. The convergence rate of a GA is largely determined by the selection pressure, with higher selection pressures resulting in higher convergence rates. Genetic algorithms are able to to identify optimal or near optimal solutions under a wide range of selection pressure. However, if the selection pressure is too low, the convergence rate will be slow, and the GA will unnecessarily take longer to find the optimal solution. If the selection pressure is too high, there is an increased chance of the GA prematurely converging to an incorrect (sub-optimal) solution. [22]

Elitism, first introduced by Kenneth De Jong (1975), is an addition to many selection methods that forces the GA to retain some number of the best individuals at each generation. Such individuals can be lost if they are not selected to reproduce or if they are destroyed by crossover or mutation. Many researchers have found that elitism significantly improves the GA's performance. [24]

There are many different techniques which a genetic algorithm can use to select the individuals to be copied over into the next generation. Some of the most popular methods are:

3.3.1 Roulette-wheel selection

A form of fitness-proportionate selection in which the chance of an individual's being selected is proportional to the amount by which its fitness is greater or less than its competitors' fitness. (Conceptually, this can be represented as a game of roulette - each individual gets a slice of the wheel, but more fit ones get larger slices than less fit ones. The wheel is then spun, and whichever individual "owns" the section on which it lands each time is chosen) [20].

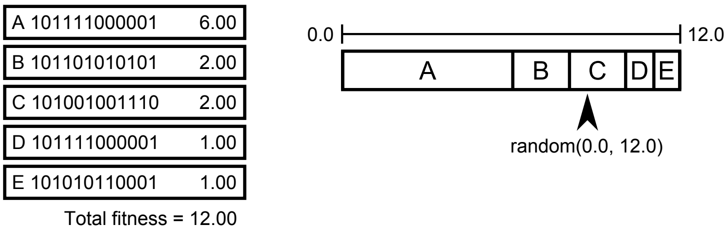

Figure 3.7: Roulette-wheel selection, where each genome has probability based on its fitness

Probabilty of being selected is $p_i = \frac{f_i}{\Sigma_{j=1}^{N} f_j}$, where N is the number of individuals in the population (Figure 3.7). Disadvantage of this approach is that the selection pressure heavily depends on the fitness variances in the population. Another disadvantage is that two passes are requied for evaluating probabilities - one to compute the mean fitness and one to compute the expected value of each individual.

3.3.2 Rank selection

Each individual in the population is assigned a numerical rank based on fitness, and selection is based on this ranking rather than absolute differences in fitness. The advantage of this method is that it can prevent very fit individuals from gaining dominance early at the expense of less fit ones, which would reduce the population's genetic diversity and might hinder attempts to find an acceptable solution. [20]

Ranking avoids giving the far largest share of offspring to a small group of highly fit individuals, and thus reduces the selection pressure when the fitness variance is high. It also keeps up selection pressure when the fitness variance is low: the ratio of expected values of individuals ranked i and $i+1$ will be the same whether their absolute fitness differences are high or low. [24] Disadvantage of this selection method is potentially time-consuming sorting procedure, which is required for rank assignment. Figure 3.8 shows probabilities of genomes in rank selection.

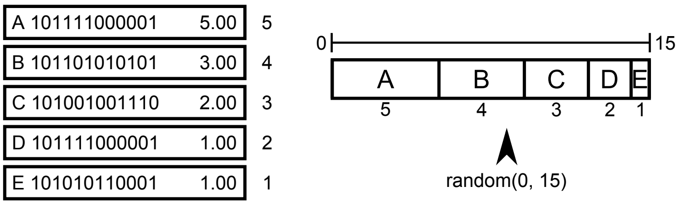

Figure 3.8: Rank selection, where each genome has probability based on its rank

3.3.3 Tournament selection

Tournament selection provides selection by holding a tournament among n competitors, with n being the tournament size. The winner of the tournament is the individual with the highest fitness of the n tournament competitors, and the winner is then inserted into the mating pool.

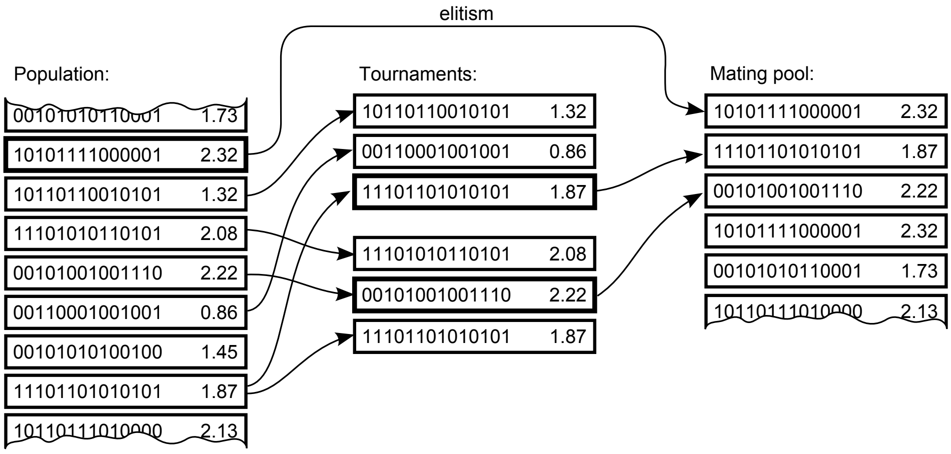

Figure 3.9: Tournament selection with elitism and tournament size of 3. Only the best genome from tournament gets selected.

Increased selection pressure can be provided by simply increasing the tournament size n, as the winner from a larger tournament will, on average, have a higher fitness than the winner of a smaller tournament. Tournament selection is increasingly being used as a GA selection scheme because it's simple to code and is efficient for both non-parallel and parallel architectures [22]. Process of selection is on figure 3.9.

Chapter 4

Parallel Genetic Algorithms

4.1 Introduction

As genetic algorithms usually require more computation time than other heuristic approaches, the basic motivation of GA parallelization is the reduction of the processing time needed to reach an acceptable solution [11].

Parallel GAs are complex non-linear algorithms that are controlled by many parameters that affect the quality of their search and their efficiency. In particular, the design of parallel GAs involves choices such as using one population or multiple populations. In both cases, the size of the population or populations must be determined carefully, and when multiple populations are used, one must decide how many to use. In addition, the populations may remain isolated or they may communicate by exchanging individuals. Communication involves extra costs and additional decisions on topologies, on how many individuals are exchanged, and on the frequency of communications [4].

4.2 Classification

The parallel GAs can be divided into three main classes[3]:

- Global single-population master-slave parallel GA
- Fine-grained parallel GA (also called *Massively parallel GA* [11] or *Neigborhood model* [1])
- Multi-deme parallel GA (also called *Multiple-population* [3], *Distributed GA* [11], *Coarse-grained GA* [4] or *Island model* [1])

4.2.1 Global Parallel GA

This class of Parallel GA is called global because all clients operate on one population. Probably the easiest way to parallelize GAs is to distribute the evaluation of fitness among several slave processors while one master executes the GA operations (selection, crossover, and mutation, see figure 4.1). Master-slave GAs are important for several reasons:

- They explore the search space in exactly the same manner as a serial GA, and therefore the existing design guidelines for serial GAs are directly applicable
- They are very easy to implement, which makes them popular with practitioners

- In many cases master-slave GAs result in significant improvements in performance [4]

The most common operation that is parallelized is the evaluation of the individuals, because the fitness of an individual is independent from the rest of the population, and there is no need to communicate during this phase. The evaluation of individuals is parallelized by assigning a fraction of the population to each of the processors available. Communication occurs only as each slave receives its subset of individuals to evaluate and when the slaves return the fitness values.

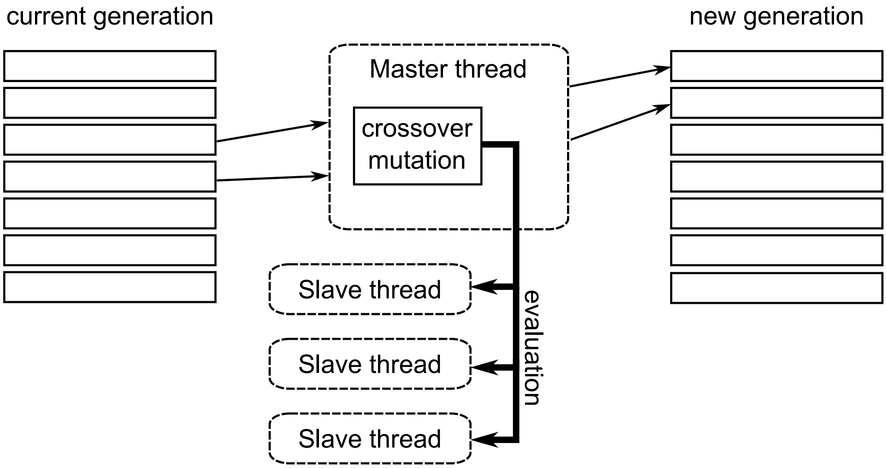

Figure 4.1: Master-Slave PGA, where master executes genetic operations and stores the population, while slaves evaluates new genomes.

If the algorithm stops and waits to receive the fitness values for all the population before proceeding into the next generation, then the algorithm is synchronous. A synchronous master-slave GA has exactly the same properties as a serial GA, with speed being the only difference. However, it is also possible to implement an asynchronous master-slave GA where the algorithm does not stop to wait for any slow processors, but it does not work exactly like a serial GA [3].

4.2.2 Fine-grained Parallel GA

Fine-grained parallel GAs are suited for massively parallel computers and consist of one spatially-structured population. Selection and mating are restricted to a small neighborhood, but neighborhoods overlap permitting some interaction among all the individuals (see figure 4.2). The ideal case is to have only one individual for every processing element available [3]. Usual parameters of fine-grained PGA are size and shape of neigborhood.

4.2.3 Multi-Deme Parallel GA

Multi-Deme Parallel genetic algorithms are the most popular parallel methods. Such algorithms assume that several subpopulations (demes) evolve in parallel and that is why this PGA is also called multiple-population or multiple-demes genetic algorithm.

The models include a concept of migration (movement of an individual string from one subpopulation to another). It uses multiple demes (populations) that occasionally exchange some individuals in a process called migration. A specification of an island GAs defines the

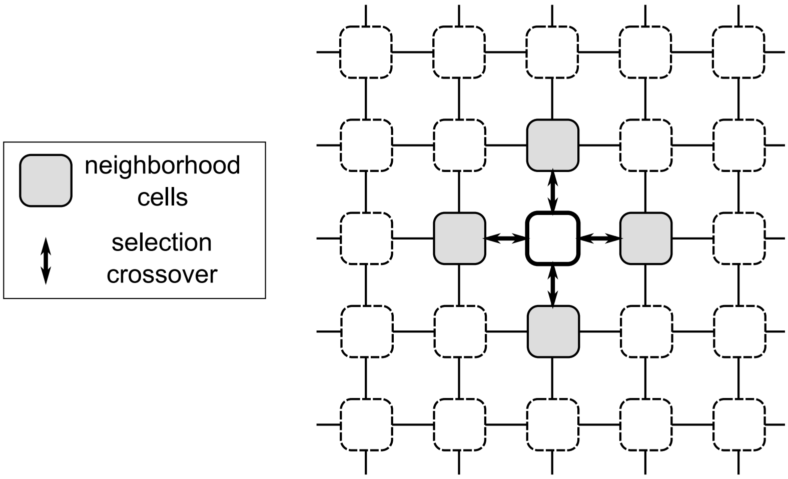

Figure 4.2: Fine-grained PGA with 2D spacial structure and 1-neighborhood

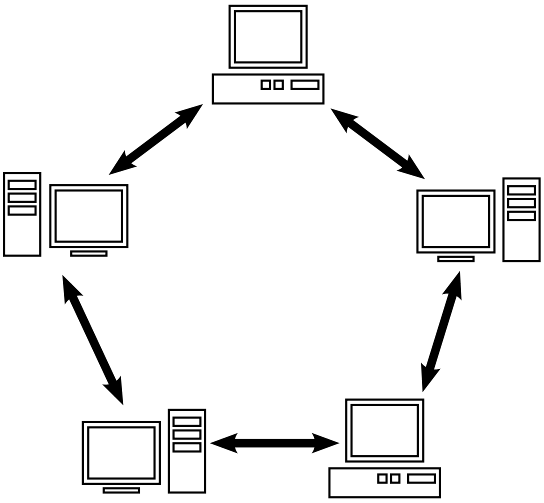

Figure 4.3: Multi-Deme PGA on home computer cluster with local neighborhood migration.

size and number of demes, the topology of the connections between them, the migration rate (the fraction of the population that migrates), the frequency of migrations and the policy to select emigrants and to replace existing individuals with incoming migrants. All these seven new parameters have a great influence on the quality of the search and on the efficiency of the algorithm. Because they are controlled by many parameters, the multiple population PGAs are the hardest to use [11]. Example of multi-deme configuration is on figure 4.3.

4.2.4 Hierarchical PGA

This class of algorithms are called hierarchical because at higher level they are multiple-deme algorithms with single-population parallel GAs (either master-slave or fine-grained) at the lower level. A hierarchical parallel GAs combines the benefits of its components, and it promises better performance than any of them alone [3].

Chapter 5

Multiprocessor Systems

5.1 Introduction

Multiple processors were used to be the exclusive domain of mainframes and high-end servers. Today, they estabilished a firm base in all kinds of systems, including high-end PCs and workstations [14].

The size of components used to build both high-end and desktop machines have continually decreased in the past few decades. Shortly before 1990, Intel announced that the company had put a million transistors onto a single chip (the i860). A few years later, the threshold of 10 million transistors was achieved by several companies. In the meantime, technological progress has made it possible to put billions of transistors on a single chip. The rate at which instructions were fetched could be increased as data paths became shorter. The main source of advances in processor performance was the raising of the clock speed. However, this approach had inherent limitations, particularly the heat emissions and power consumption was getting increasingly hard to deal with [6].

Therefore, computer architects have begun to explore other strategies for increasing hardware performance and making better use of the available on the chip available to them: multiple processors that share memory. At first the processors were configured in a single machine, and later, on a single chip. This approach is known as *multicore*. Simultaneous multithreading platforms, multicore machines, and shared-memory parallel computers all provide system support for the execution of multiple independent instruction streams known as *threads* [6].

5.2 SMP

SMP stands for symmetric multiprocessing which represents multiprocessor system with one shared memory. Processors and physical memory are usually connected by a bus or crossbar switch (Figure 5.1 and 5.2). All the processors in the SMP system share the physical memory uniformly - access time to any memory location is independent of processor making the request or memory chip containing the requested data (it is also called UMA - uniform memory access). Since every processor has its own private cache memory, it is essential to ensure cache coherency - that every copy of the shared data is the same in every cache memory. That's why SMP architecture is also called ccUMA (cache coherent Uniform Memory Acces) [10].

SMP is the most popular parallel architecture today[10]. SMP computers are usually

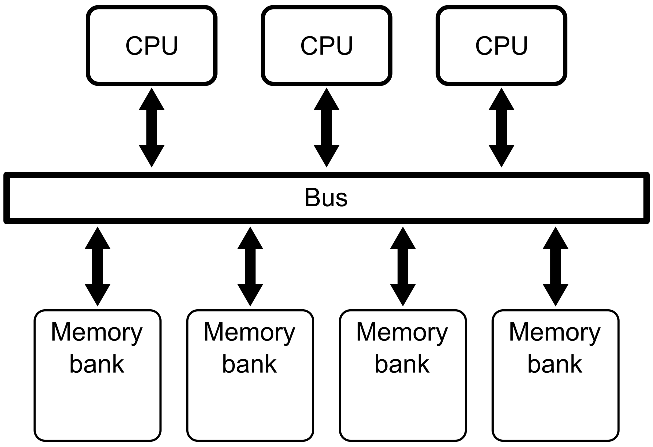

Figure 5.1: Connection between processors and memory by bus - only one processor can access the memory at once.

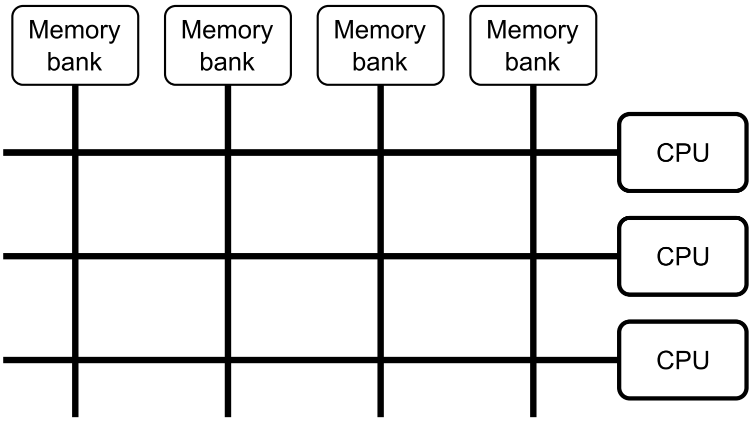

Figure 5.2: Connection between processors and memory by crossbar. This connection allows some degree of simultaneous memory access, thus achieving greater efficiency than bus solution.

used as servers or building blocks (nodes) for larger supercomputer clusters. Symmetric multiprocessing also dominates home computer market with multicore architecture containing two or more independent processing units called *cores* on one chip[10].

5.3 NUMA

The main disadvantage of SMP architectures is the memory bottleneck of the shared memory bus, which prevents effective scalability of the SMP system. This drawback can be eliminated by providing each processor with its own local memory. Processors are then connected by shared bus (as seen in diagram on figure 5.3) or high-throughput connections (diagram on figure 5.4) to get access to non-local memory. As latency of local and non-local memory is different (see 5.1), this approach is known as NUMA (or ccNUMA - cache coherent Non-Uniform Memory Access [15]).

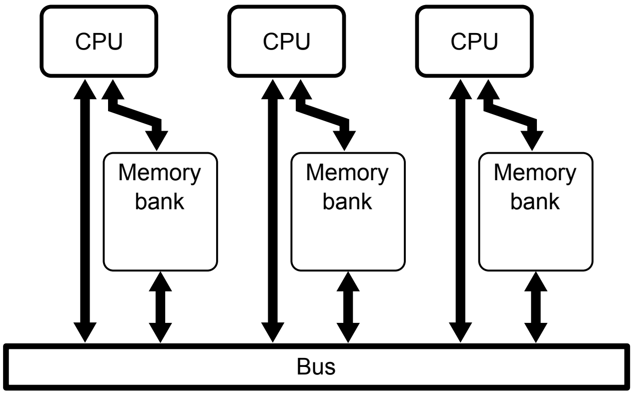

Figure 5.3: Each processor is directly connected to local memory. Shared bus or crossbar is used to access non-local memory.

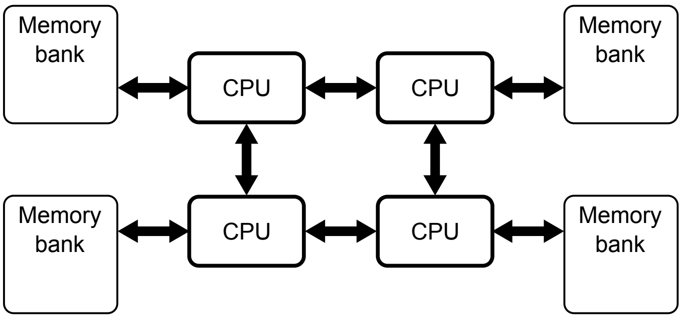

Figure 5.4: Each processor is connected to its neighbour by high-speed connection. To access some of the data, two additional processors must be enquired (two "hops" are required) in the worst case. Table in 5.1 describes various access times on AMD Opteron system, which uses this form of inter-CPU communication [16].

Data location	Page-hit latency (ns)	Page-miss latency (ns)
Local memory	65	95
Adjacent processor	100	120
Two "hops" processor	140	160

Table 5.1: AMD Opteron memory access latency (source: [16])

As each processor have immediate access to its local memory and this communication channel is not shared with other processors, it can access the memory anytime without need for synchronization mechanism. This ensures better scalability in comparison with UMA systems. On the other hand, rather complicated inter-processor communication systems and protocols must be employed between cache controllers to ensure cache coherence [26] (more on cache coherence is in section 5.5). This can be obviously problem and can cause poor performance in case that more processors tries to access the same memory resource.

5.3.1 Hybrid NUMA

Modern system often combine UMA and NUMA models to create hybrid architectures, thus exploiting advantages from both approaches. Processors are grouped together with memory to form nodes. Memory acces time within the node is the same for all the cores in the node - there is local unified memory access. These nodes are then connected by bus or high-speed interconnect to allow access to non-local memory banks at slightly higher latencies, thus creating non-uniform access between the nodes[15]. There is example of such system on figure 5.5.

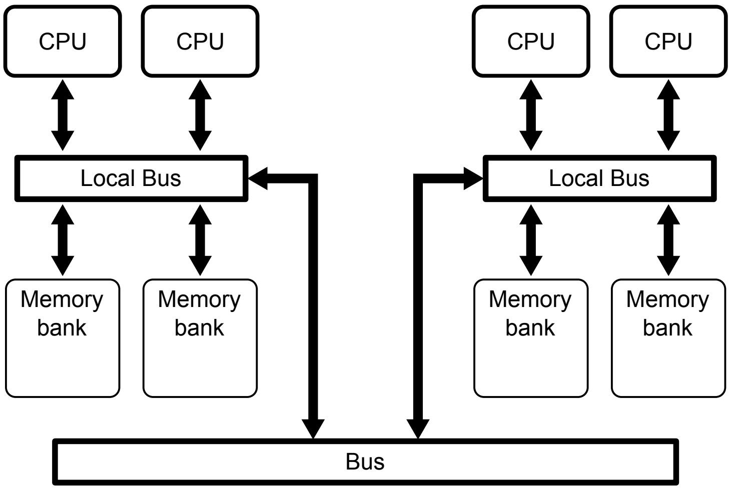

Figure 5.5: Hybrid architecture combinig UMA and NUMA approaches to gain advantages from both models. Server platforms like Intel Xeon are examples of such system [15].

5.3.2 OS Support

To ensure the best performance, operating system must be aware of the architure on wich it runs. On SMP and single processor systems, each memory page is as good as another. But this does not hold for NUMA architectures, where distance between the memory and processor matters. The distribution of processors and memory into the nodes must be taken into account when designing the scheduler and the kernel to find the most optimal way of memory and thread mapping. The kernel memory allocation system must minimize the distance between physical memory on which the thread's data are mapped and the processor which the thread is running on [9]. The scheduler should also avoid migration of process or thread to another node, as this would mean that all of the data had to be copied also. Another optimization problem is shared code of OS standard libraries, which is normally present only once in the memory. Optimally, the parts which are used by the multi-process application should be mirrored to each node of NUMA system [9].

5.4 Speedup

In parallel computing, speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm. Speedup is defined as:

$$S_N = \frac{T_1}{T_N} \tag{5.1}$$

Where N is the number of processors, T_1 is the execution time of the sequential algorithm and T_N is the execution time of the parallel algorithm on system with N processors[26]. To obtain estimated speedups of parallel programs, researchers have been using two different, but mathematicly equivalent formulas - *Amdahl's Law* and *Gustafson's Law* [29].

Amdahl's law is named after computer architect Gene Amdahl, and is used to find the maximum expected speedup to an overall system when only part of the system is improved. It states that serial program can be decomposed into two portions - s and p, where p is the proportion of a program that can be made parallel, s is the proportion that cannot be parallelized (remains serial), and $s + p = 1$. Then the maximum speedup that can be achieved by using N computational units is:

$$\frac{s+p}{(s) + \frac{p}{N}} \tag{5.2}$$

After using $s + p = 1$ and $s = 1 - p$, formula can be reduced to:

$$\frac{1}{(1-p) + \frac{p}{N}} \tag{5.3}$$

As N approaches infinity, the maximum speedup tends to $1/(1 - p)$. This means that performance falls rapidly as N is increased once there is even a small component of $(1 - p)$[26], as can be seen on figure 5.6.

Main prerequisite to applying the *Amdahl's Law* is that the serial and parallel programs must compute the same total number of steps for the same input, which is hard to satisfy for commonly used algorithms and produce confusion [29]. Often the parallel implementation is directly crafted from the corresponding serial implementation of the same algorithm, therefore an alternative formulation was proposed by John Gustafson known as *Gustafson's Law*. In Gustafson's formulation, a new serial percentage is defined in reference to the overall processing time using N processors and it is dependant on N. This N dependent serial percentage is easier to obtain than that in Amdahl's formulation via computational experiments.

The difference from Amdahl's formulation is that Gustafson takes already parallelized version of problem as basis for computing time needed to run that version on sequential computer. The execution of the program on a *parallel* computer is decomposed into $s+p = 1$, where s is the serial fraction of the program and p is the parallel fraction. On a sequential computer, the relative time would be $s+Np$, where N is the number of processors in parallel case [26]. Speedup is therefore:

$$\frac{s+Np}{s+p} \tag{5.4}$$

Because $s + p = 1$ and $p = 1 - s$, we can reduce this formula to:

$$s + N(1-s) \tag{5.5}$$

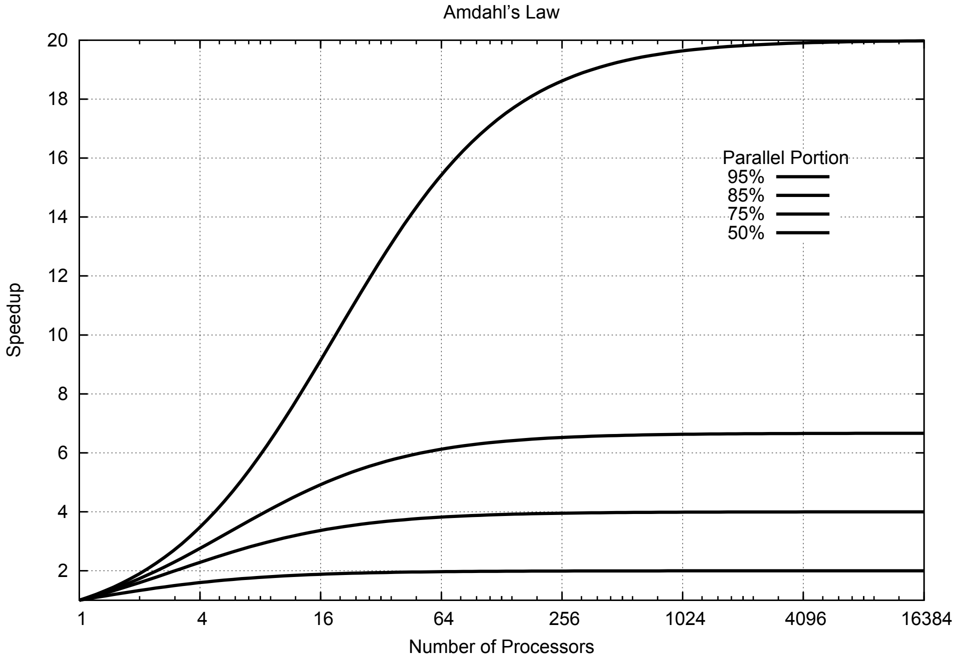

Figure 5.6: Graph of expected speedup for different levels of parallelization based on Amdahl's law

Gustafson discovered, that in the real-world applications, the parallel part p of a program tends to scale with the problem size, but the serial part s, consisting of program loading, serial bottlenecks and I/O, remains constant [12]. So as the problem size grows, s diminishes and the speedup approaches number of processors N.

It has been observed that theoretical speedups could be overly optimistic due to overhead incurred while parallelizing the code. This overhead often includes additional code requied to parallelize task as well as communication latency due to shared memory bandwidth and ensuring cache coherence [2].

5.5 Cache Coherence

Althogh theoretical speedup of algorithm can be computed independently of the architecture of system on which the algorithm is implemented, real speedup is limited by things like memory bandwitdth, communication speed and by *cache coherence mechanism*.

One of the major challenges facing computer architects today is the growing difference in processor and memory speed. Processors have been consistently getting faster. But more rapidly they can perform instructions, the quicker they need to receive the values of operands from memory. Unfortunately, the speed with which data can be read from and written to memory has not increased at the same rate [27]. This effect is illustrated on figure 5.7.

In response, the vendors have built computers with hierarchical memory systems, in which a small, expensive, and very fast memory called *cache memory*, supplies the processor with data and instructions at high rates. Each processor of an multiprocessor system needs its own private cache if it is to be fed quickly; hence, not all memory is shared[6]. Figure 5.8 shows an example of a generic, cache-based dual-core processor.

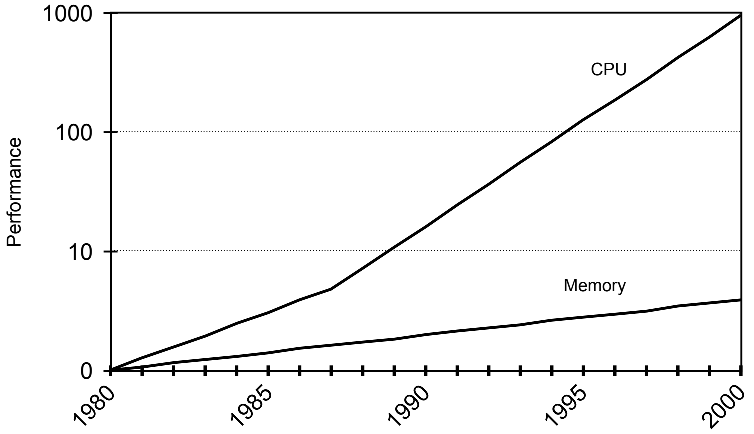

Figure 5.7: Difference between processor and memory speeds in years 1980 - 2000 (source: [27])

In a uniprocessor system, new values computed by the processor are written back to cache, where they remain until their space is required for other data. At that point any new values that have not already been copied back to main memory are stored back there. This strategy does not work for multiprocessor systems. When one processor of such system stores results of local computations in its private cache, the new values are accessible only to code executing on that processor. If no extra precautions are taken, they will not be available to instructions executing elsewhere on an machine until after the corresponding block of data is displaced from cache. But it may not be clear when this will happen. In fact, since the old values might still be in other private caches code executing on other processors might continue to use them even then[6].

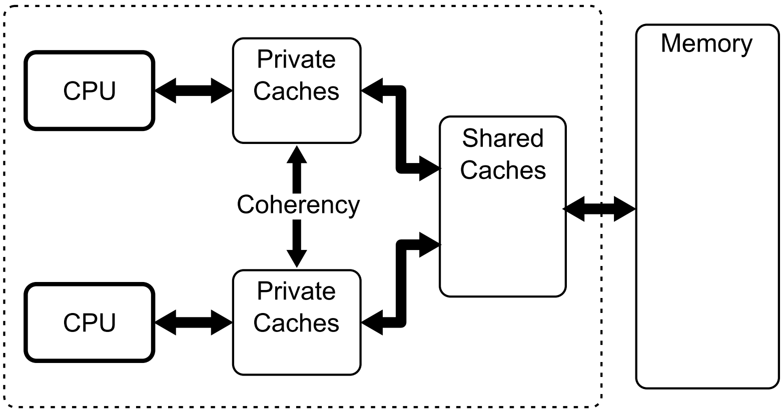

Figure 5.8: Diagram of generic dual-core system with cache memories

This is known as the *memory consistency problem.* A number of strategies have been developed to help overcome it. Their purpose is to ensure that updates to data that have taken place on one processor are made known to the program running on other processors, and to make the modified values available to them if needed. A system that provides this functionality transparently is said to be *cache coherent*[6].

Chapter 6

New Methods of PGA

6.1 Motivation

As multiprocessor architecture is dominating the computer systems market, beginning from personal computers, through high-end workstations used for demanding computations, and including even massively parallel supercomputer clusters, which are using multicore chips as building blocks [18]. These systems provides better scalability of computation performance and lower power consumption over the single-processor machines, and because of this, it has became a trend in computer design.

As genetic algorithms are computationally demanding applications, it is imperative to have efficient approach to exploit advantages of these systems. This chapter presents prototypes of two new methods specifically designed for multiprocessor architectures.

Description of the methods' inner workings, advantages and disadvanages is presented in the second and third section, with section two describing Symmetric Method, and section three describing Asymmetric Method.

As there is difference between SMP and NUMA systems, there are two variations for both methods. Each variation is adapted to its corresponding architecture to avoid performance loss. These variations are described in fourth section of this chapter.

Various implementation details as random generator and fitness function used for benchmarking are represented in section five.

6.2 Design Principles

The most efficient parallel algorithm would have following properties:

- No communication
- No shared resources
- No sequential code

However, only limited number of problems could be parallelized efficiently while conforming to this restriction. Unfortunately, as crossover operation is one of the core parts of GA[8], it is not possible to completly adhere to aforementioned principles (especially the restriction of the shared resources). Nevertheless, minimization of communication, shared resources and sequential parts are key principles in designing new methods of PGA.

These methods are described in following two sections. Only variants for SMP systems are described, and the modification of these methods for NUMA systems is then presented in section 6.5.

6.3 Symmetric Method

The first of new methods of parallel genetic algorithm presented in this section belongs to category of global parallel GA, because there is only one distributed population. It is similar to master-slave model, with some properties of multi-deme parallel GA.

In traditional master-slave model, slave threads are used only to evaluate genomes while master thread performs other evolution operators and stores the generation (figure 4.1). This model requires communication between master thread and slave thread. On the other hand, in Symmetric Method, all the threads are equal and each thread performs, in addition to evaluation of fitness, full evolution process with selection, crossover and mutation, thus eliminating master-slave hierarchy and therefore also the need for communication. Figure 6.1 shows this process on system with two processors.

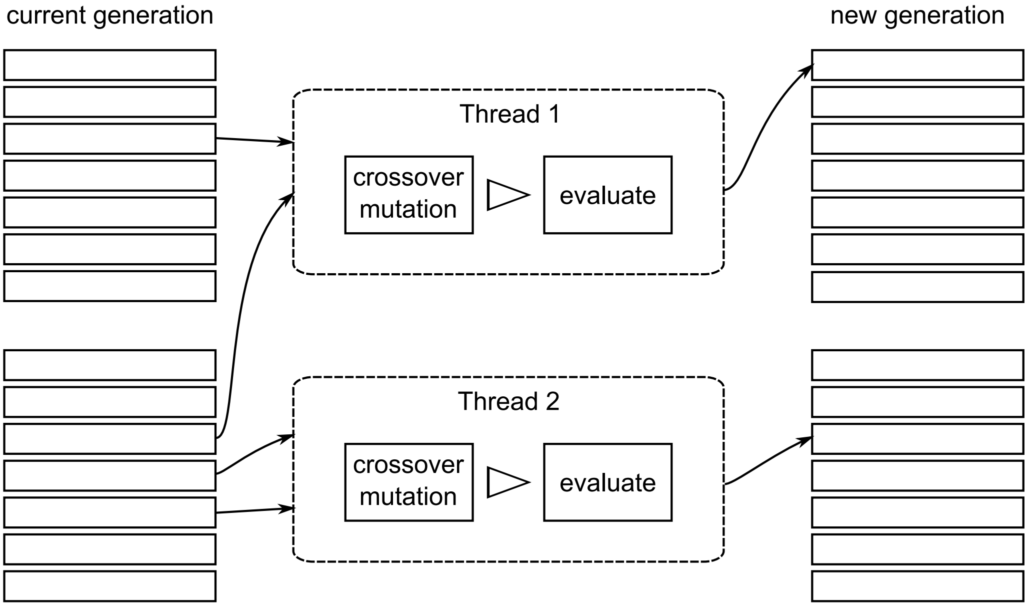

Figure 6.1: Schematic drawing of evolution process on system with two processors.

Since there is no communication, genomes are divided into small subpopulations of equal size and each thread stores new genomes into its own population to prevent genome loss due to overwrite. But as the memory is shared, selection of parents is done globally over all subpopulations with little penalties on SMP systems (the modification for NUMA systems is described in section 6.5). Because of that, these small subpopulations create together one global population, where each genome has the same chance to get selected by each thread as in serial GA.

The evolution process is divided into two phases - first is the *initialization phase*, where every thread initializes its population by randomizing all genomes and then evaluates every genome. The *evolution phase*, which comes after *initialization*, lasts until the ending condition is met.

Each thread has two storage areas - *current generation* and *new generation* (see Figure 6.1). Parents are selected from *current generation* and offsprings are stored to *new generation*. After all free places in *new generation* are filled, pointers to both storage areas are swapped making *new generation* act as *current generation*. Swapping is done in an asynchronous manner. The pseudocode for this process is in figure 6.2.

```
randomize all genomes in current_generation
evaluate all genomes in current_generation

while ( not ending_condition ):
   select 2 genomes
   create offspring
   evaluate offspring
   store offspring in new_generation
   if ( new_generation is full ):
      swap new_generation and current_generation
```

Figure 6.2: Pseudo-code for one thread running Symmetric Method.

6.3.1 Elitism

Elitism is very powerfull technique and helps to maintain the convergence of fitness values. In case of Symmetric Method, the elitism is implemented on each subpopulation as copy of the best genome in current subpopulation to its new generation.

6.3.2 Properties

A good method of parallelization should preserve any properties that sequential algorithm with the same genetic operators would have. It should also not introduce too many additional parameters whose values could significantly affect the GA performance. [11].

Symmetric Method has almost all of this properties except identity with serial GA due to its asynchronous matter. It is using the same genetic operators except selection operator, which needs to be changed to work on multiple subpopulations.

Furthermore, the communication and synchronization is eliminated, with sequential part used only for memory allocation and thread creation. Because of this, the sequential part is not scaling with the problem size and remains constant, therefore it tends to diminishes as the problem grows.

The description of some properties of SMP version below (for properties of NUMA version, see section 6.5):

Advantages

Global population - because the selection operates on all of the populations equally, every genome has a chance to compete with each other making it one global distributed population. This eliminates the need for migration.

No communication - with no migration, non-blocking write, shared memory and asynchronous approach, there is no need for threads to communicate to each other.

Non-blocking write - each thread stores new genomes only into its designated memory. Threads don't have to wait for each other to complete writing and one thread cannot overwrite other thread's offspring.

Everything is parallel - since whole evolution process is done in parallel, speed-up of asynchronous approach is limited only by memory bus bandwidth and cache coherence mechanism.

Localized memory usage - as each thread has its own private memory for writing and this memory is not read by other threads at the time of writing, the need for cache controllers communication to ensure cache coherence is minimal.

No additional parameters - global population doesn't require new parameters as migration rate and frequency needed for multi-deme PGA, and shared memory eliminates need for topologies and communication.

Easy implementation - implementation of the method is simple as only selection operator needs to be modified to work on whole population. This can be simplified even more by using tournament selection, which selects n genomes with equal chance (as it is no problem to choose uniformly from many population), thus minimizing amount of modifications.

Disadvantages

Shared memory - this method requires shared uniform access memory by design, as the selection operates on all genomes. As so, efficient implementation on computer architectures with non-uniform memory access is not possible without further modification of algorithm described in section 6.5. Also memory bus bandwidth could be bottleneck on some multiprocessor systems, as each processor accesses genomes from whole population.

Selection methods - distributing population into several subpopulations makes this method unsuitable for any kind of fitness-proportional selection because sorting of population after each asynchronous swap would be impossible without communication and synchronization.

Thread starvation - as each subpopulation has corresponding thread, the fitness of its genomes are dependant on flawless execution of this thread. In case the thread is deprived of processor, the subpopulation is stagnating and low-fitness genomes are leaking to the global population, thus slowing the convergence.

6.4 Asymmetric Method

Asymmetric Method is focusing on some problems of Symmetric Method - thread starvation particulary. This could be serious problem, as running time of GA could be measured in days, and it would be inconvenient to not be able to use the computer during this time.

Asymmetric Method is designed to minimize the damage in case that some thread will be denied of processor for few generation. To solve this problem, it divides the global population into many smaller subpopulations (more than number of threads). Due to this fact the probability of chosing parent genomes from the defect subpopulation is lower.

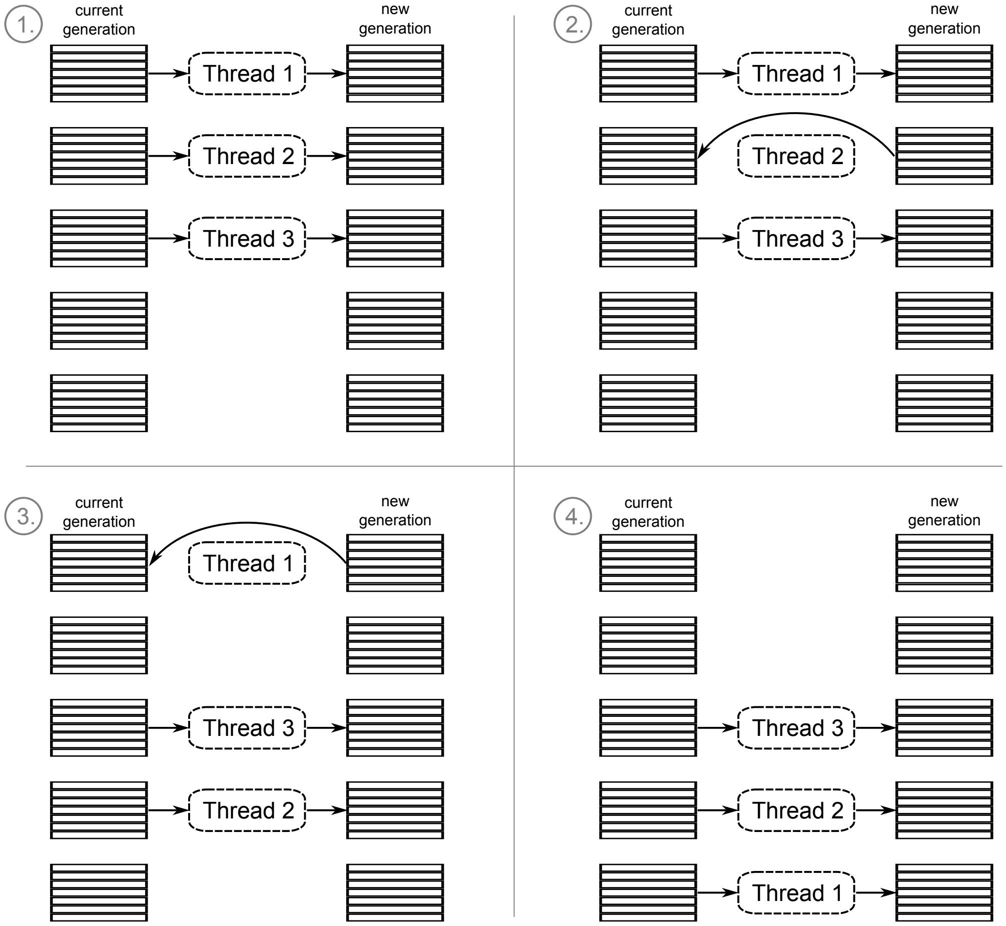

Figure 6.3: Consecutive steps of computation by Asymmetric Method. (1) Each thread is evolving corresponding subpopulation. (2) Thread 2 is finished, swapping new generation with current generation and moving to next free subpopulation. (3) Thread 2 is evolving the new subpopulation while Thread 1 finished evolution. (4) Thread 1 acquired the next free subpopulation.

As there are more subpopulations than thread, each thread will work on more than one subpopulation. When the thread finishes evolution of current subpopulation, it will claim the first "free" one and starts evolving it. Figure 6.3 displays few steps of the evolution process done by Asymmetric Method. Pseudo-algorithm form Asymmetric Method is on figure 6.4.

As there is possibility that each subpopulation will be evolved by each thread in some time, the damage done by the starving thread should be divided equally between all subpopulation. The crucial difference between Symmetric Method and Asymmetric Method is the acquisition of free subpopulation, because it cannot be done without sacrificing advantages like no thread communication or synchronization. Section 6.7.1 describes the implementation of this acquisition.

```
while ( not ending_condition ):
   pop = first free subpopulation
   if pop was not initialized:
      randomize all genomes in current_generation
      evaluate all genomes in current_generation
      continue to next iteration

   for each genome in new_generation:
      select 2 parents
      create offspring
      evaluate offspring
      store offspring in new_generation
   swap new_generation and current_generation
```

Figure 6.4: Pseudo-code for one thread running Asymmetric Method.

6.4.1 Properties

Experimental results in section 7.3 shows that the Asymmetric Method successfully eliminates the thread-starvation disadvantage of Symmetric Method. Additionally, Asymmetric Method retains some of the advantages from its symmetric counterpart:

- Global population
- Non-blocking write
- Whole evolution is parallel
- Localized memory usage

However, it needs additional parameter (number of subpopulations) and synchronization (albeit with little overhead, as seen on table 6.2). Also it is not suitable for NUMA architectures, as each thread could work on any subpopulation so it is impossible to associate the data with some processor. Asymmetric Method also retains some of the disadvantages of Symmetric Method such as it need memory with uniform access to achieve efficient performance, and the selection operator must be modified to work on subpopulations.

6.5 SMP vs NUMA

SMP technology is one of the most popular parallel architectures today. It is used in form of multicore CPU in personal computers, servers and even building blocks of supercomputer clusters [18]. SMP systems are also often used as nodes for hierarchical parallel genetic algorithms.

On the other hand, high-end computer systems used for scientific calculations tends to have higher processor count and so are using NUMA architecture because of its better scaling. Application programming for these systems is slightly different, because memory and processor location on chip must be taken into account when designing the parallel algorithm. On SMP system, the algorithm could count on uniform access time to any block of memory. This is not the case on NUMA architecture, where the data must be allocated carefully and simultaneous reading or writing to the same memory block by more than one processor must be avoided at all costs [9]. Thus algorithm designed to fully exploit all of the resources on SMP system could have poor results on NUMA system.

So to be effective on NUMA architecture, each thread should have associated separate data, and access to other data should be minimized, so OS can allocate data to physical memory more effectively. Luckily, in case of genetic algorithm, the selection function is used by thread to obtain data it will use in evolution process, so the modification is restricted to writing two different selection operators, one for SMP system and the other for NUMA system.

6.5.1 NUMA selection

As accesing genomes residing in non-local memory will slow down the execution of the thread, the selection operator should be inclined to choose genomes from thread's local subpopulation. This could be implemented in two ways:

1. for choosing genomes from non-local subpopulation with low probability
2. restrict selection to operate only on local subpopulation, ignoring non-local subpopulations entirely

To compare this two approaches, both were implemented and run on NUMA systems. Figure 6.5 shows speedup of each approach.

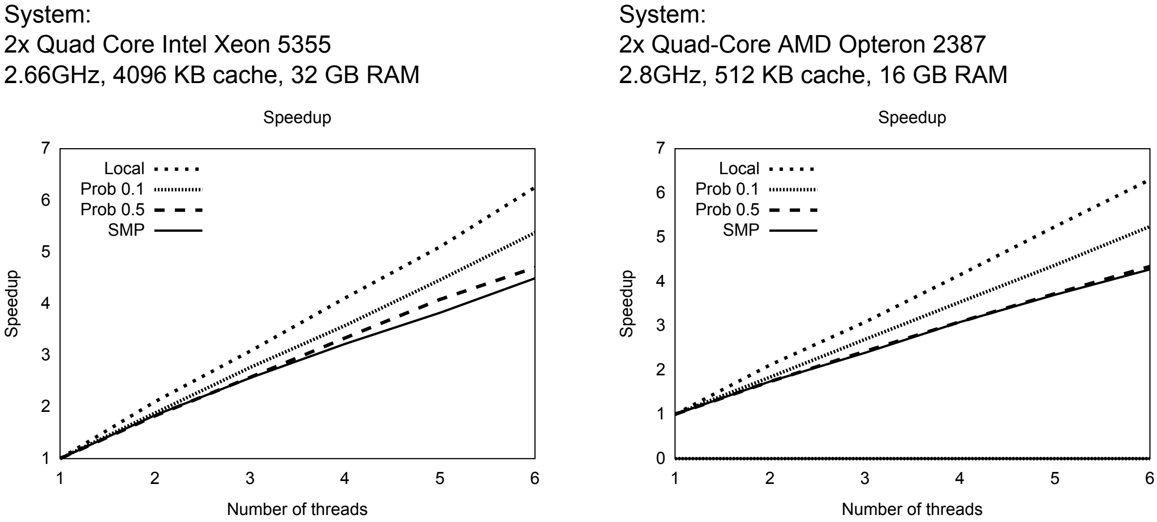

Figure 6.5: Speedup of NUMA selection approaches compared with UMA selection. *Local* describes selection restricted to local subpopulation, *Prob 0.1/0.5* describes selection from global population with 0.1/0.5 probability and *SMP* describes selection chosing all genomes with uniform probability. The tests were taken on 2 different NUMA systems and values are computed as arithmetic means of 20 independent runs.

The results shows that the selection restricted to operate only on local subpopulation has the best speedup. With local selection, there would be no means to get the genomes from other subpopulations, so some form of migration must be introduced to NUMA variant of GA. To minimise the operations on non-local data, the migration scheme displayed on figure 6.6 will be employed. Each thread takes best genome from all of the subpopulations and copies these genomes into new generation. Then it will continue the evolution process as in SMP version. Comarison of fitness convergence for SMP selection and NUMA selection with migration is described in section 7.2

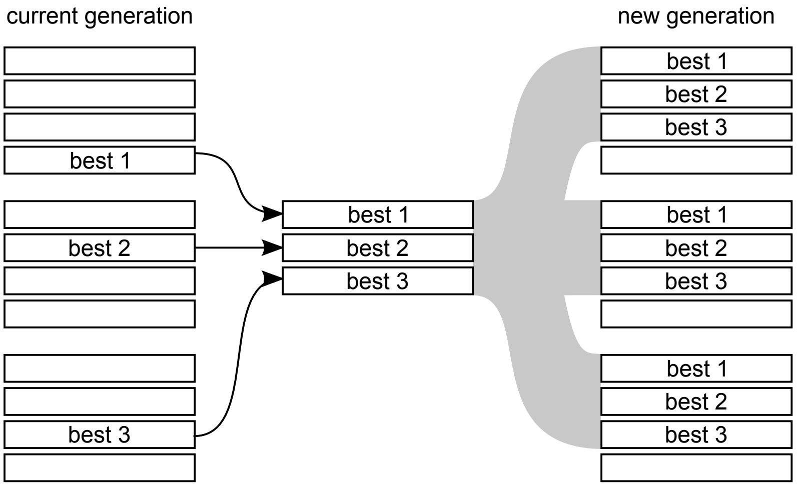

Figure 6.6: Diagram showing the migration in NUMA variation. Each thread copies the best genome from each subpopulation to its local new generation.

6.5.2 Disadvantages

The obvious disadvantage of NUMA selection is that it selects genomes only from local subpopulation, thus altering the function of GA. With UMA selection, the paralell version of GA behaves similary to serial GA - the evolution is done on whole population. But with NUMA selection, it is closer to the island model of multi-deme paralell GA. Another disadvantage is the need to implement migration schemes to mix the genomes of local subpopulations, and this brings additional parameters required to define the process of migration. These parameters are often very hard to optimize for achieving the best results [4].

6.6 Comparison

Although modification of the selection operation to work on NUMA systems is simple, it alters some of the properties of each method. Table 6.1 displays properties of each method and its corresponding variants.

6.7 Implementation

Prototypes of both methods were implemented in C language using the OpenMP[6] library for parallel computing. Some details of implementation are described below.

Method	Symmetric		Asymmetric	
	SMP	NUMA	SMP	NUMA
Global population	X		X	
Non-blocking write	X	X	X	X
No additional parameters	X			
Everything is parallel	X	X	X	X
No synchronization	X	X		
Special selection	X		X	

Table 6.1: Comparison of methods' properties

6.7.1 Synchronization

To minimize communication and synchronization, the best way to acquire the free subpopulation in Asymmetric Method is to keep index pointig to first free subpopulation, so the process of acquisition can be implemented as simple atomic increment operation. In the same atomic operation, we must also get the old value. There are several ways to do it, and as the prototype is implemented using OpenMP and GNU Compiler Collection[25], following options for atomic operations are available:

- OpenMP Atomic command
- OpenMP Critical section
- GNU gcc built-in function `__sync_fetch_and_add` [25]

Table on figure 6.2 shows running times of each operation. We can see that OpenMP Critical section is not suited for this kind of problem as the overhead is too high for one incrementation. Running time of OpenMP Atomic incrementation is acceptable, but it is impossible to increment and get value in one atomic operation using this approach[6], so GNU gcc built-in function `__sync_fetch_and_add` is the only choice that remains.

Method	Intel Xeon	AMD Opteron
Without synchronization	0.037	0.1651
OMP critical	9.56	12.6
OMP atomic	0.26614	0.29312
`__sync_fetch_and_add`	0.26934	0.2884

Table 6.2: Running times in seconds for various synchronization methods and computer systems. Values are calculated as arithmetic means from 50 independent runs. Each run consisted of 10 million corresponding operations. Source code of the benchmark program is in appendix A.

6.7.2 Fitness functions

To benchmark the efficiency and speedup of new methods, two fitness functions are used:

- Onemax
- Cartesian Genetic Programming [7]

Onemax

Onemax is one of the most used fitness function for evaluating GA performance. The calculation of fitness value is straightforward - it is the count of bits with value 1.

Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) is relatively new technique of genetic programming (GP). It was presented by J. F. Miller and P. Thompson in 2000 [23]. CGP represents computation algorithms by acyclic directed graphs rather than trees used by traditional GP pioneered by John R. Koza in [17]. Each node of the graph is encoded as three numbers: function index, node index for the first input and node index for the second input. To keep the graph acyclic, input of each node can be taken only from previous nodes. Genotype of CGP consists of nodes list followed by indexes of output nodes. Example of genotype and corresponding phenotype used for solving symbolic regression is on figure 6.7.

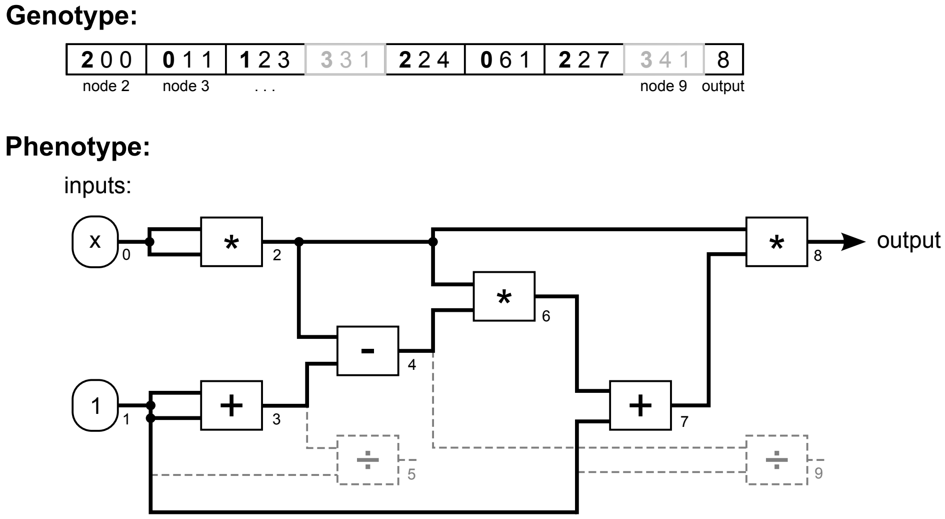

Figure 6.7: Example of relation between CGP genotype and corresponding phenotype. The phenotype represents computation of mathematical expression $x^6 - 2x^4 + x^2$. First number of each gene is function type with (0)+, (1)−, (2)∗ and (3)÷. Grey color denotes inactive nodes. Surce: [7]

CGP was usually implemented with mutation only, because the crossover used on integer encoding has negative impact on fitness convergence. However, in 2007, Miller and others introduced new encoding using floating point numbers in [7]. This encoding was designed to allow succesfull crossover operator for CGP, speeding-up the convergence considerably. Each node consisted of three floating point numbers from interval $\langle 0, 1 \rangle$. Each number corresponds with integer number in original genome so the purpose of each number is not changed. To get the index values from floating points number, each interval is mapped to corresponding integer range from 0 to maximum allowed index. The value of particular index is then produced by rounding the floating point number representing it. With this kind of encoding, the real-valued crossover can be applied to genome. This technique was shown to produce better results than original CGP [7].

For benchmarking the new methods, CGP was used to find the logical circuit realizing the 3x3 multiplier. The building blocks for this circuit was nodes with four logical functions: *and*, *or*, *xor* and *not*. Each node has two 1-bit inputs and one 1-bit output. The input of the whole circuit are 6 bits representig two 3-bit numbers and the output is one 6-bit number. The circuit containins 100 nodes, although not every node must be active. Fitness value of genome is defined as number of bits that are the same as bits in desired output. This difference is calculated for all of 2^6 input combination. In case of absolutely correct results, the number of inactive nodes are added to fitness, so the evolution can optimize also the size of the circuit.

6.7.3 Pseudorandom number generator

As GA are essentialy stochastical algorithms, the function implementing pseudorandom number generator (PRNP) is the most called function in the whole system. Large period and high efficiency are the main requirements that GA put on this generator. Although statistical quality of generated numbers can be helpfull, it was shown in [5] that it is not essential for good convergence. Based on these requirements, xorshift [21] was selected as PRNG for benchmarking. As its name suggests, only bitwise *xor* and *shift* instructions are used for generating random numbers. Because these instructions are among the fastest instruction supported by modern CPUs, xorshift can produce numbers at very high rates. These numbers have also sufficient statistical quality as xorshift has period $2^{128} - 1$ and passes all of die-hard tests designed to evaluate PRNG [21]. Source code used in prototype implementation is in appendix B.

Chapter 7

Experimental Results

This chapter presents results of various experiments used to benchmark the speedup and convergence of both methods, and conclusions based on acquired data.

Goal of the GA was to optimize two problems: simple onemax and more complex design of digital logical circuit using cartesian genetic programming. More info on implementation details of each method and fitness functions is in section 6.7.

7.1 Speedup

This section presents a comparison of Symmetric Method and Asymmetric Method. The speedup of each method with corresponding UMA and NUMA variant are calculated from data acquired from 50 independent runs. Each run consisted of 5000 generations. As the methods are essentialy asynchronous, the notion of *generation* is defined as number of evaluations equal to population size (with elitism and migration not counting towards this number)

Results are presented by graphs on figure 7.1. Tests were run on four multiprocessor systems with different configurations. Brief description of the systems is presented in the following list, with more detailed technical parameters described in appendix C.

edesign1, edesign2 – high-performance multiprocessor systems dedicated for research purposes. Hybrid NUMA architecture (2 Dual-Core chips for edesign1 and 2 Quad-Core chips for edesign2) based on AMD and Intel processor technologies. Both systems run Linux-based operating system.

merlin – computer server system set up for developing and testing student projects. As all of the students have access to this system (opposed to edesign systems with access restricted to authorized personell), it is under low to moderate load for most of the time. Tests on this system serves to benchmark performance on such systems. Server is built on NUMA architecture with two Quad-Core AMD Opteron chips.

pcjaros-gpu – high-performance private SMP system with single Quad-Core Intel CPU used for benchmarking algorithms accelerated on graphic cards. As all of the other systems represents hybrid NUMA architectures, pcjaros-gpu is used for obtaining test results on SMP architecture.

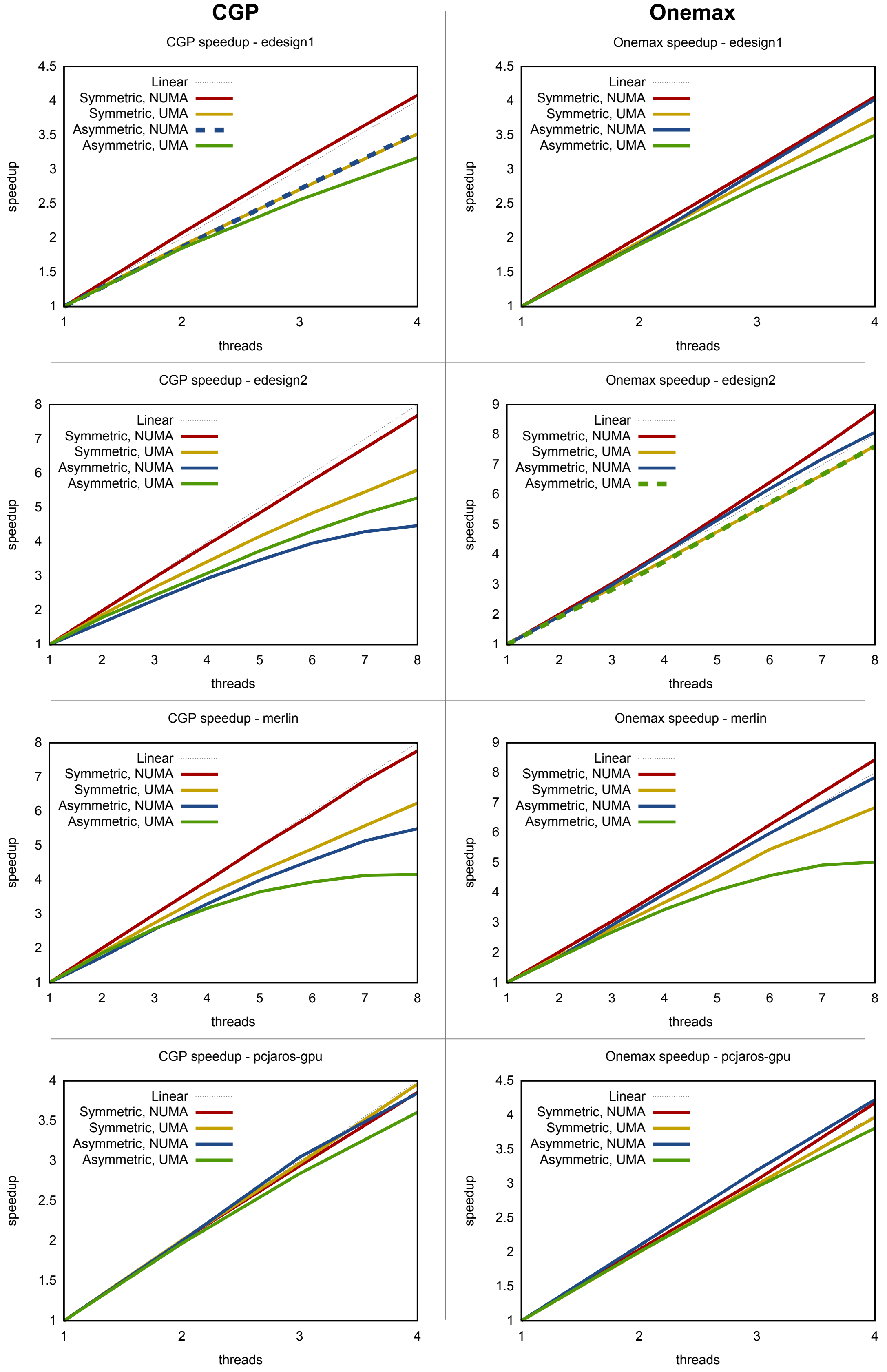

Figure 7.1: Graphs showing speedup for different systems and fitness functions.

The size of population was chosen to be 840 as this number is divisible by numbers from 1 to 8 (the numbers of threads), so each test run will be run on the population of same size. For Asymmetric version, the number of subpopulations was $s = 2t + 1$, where t is the number of threads. Appendix C describes all of the parameters used to configure genetic algorithm for the tests. Due to the elitism and migration, different number of genomes are copied to new generation (details in table 7.1). As these genomes have their fitness already calculated, they are not counted towards evaluations count.

7.1.1 Conclusion

The results plotted in figure 7.1 shows that the speedup of both methods is dependandant on many factors, including fitness function and even hardware configuration of multiprocessor system the test is being run on. Other than that, the results were quite consistent with the expectations except a few phenomena:

Superlinear speedup – on some systems, the onemax fitness function displays superlinear speedup. This could be the effect of larger cache memory being used – as more threads use more processor and each processor has its own cache, more data could be stored in high-speed low latency cache memory contributing to better speed. This assumption would also explain the fact that CGP fitness function doesn't display such large superlinear speedup, as the CGP genome is roughly 10 times larger than genome used for onemax problem (for details on parameters of GA see appendix C). This assumption should be the subject of further research.

UMA faster than NUMA – on edesign2, the UMA variant of Asymmetric version optimizing the CGP problem shows greater speedup than its corresponding NUMA variant. As the Asymmetric UMA variant is working with whole population all the time, it is expected to have the worse speedup. This holds true for all the other test cases.

Overall, the NUMA variant of the Symmetric Method shows nearly linear speedup in all of the test cases. Second place in performance is dependant on fitness function, as the Asymmetric NUMA variant shows better results for onemax, but Symmetric UMA variant beats it in the field of CGP problem optimization. All of the methods and variations display somewhat linear speedup on SMP system pcjaros-gpu.

7.2 Convergence

In previous chapter we measured the performance of GA by measuring the the time of run. Althoug this time is one of the indicators of performance, the effectivity of GAs also depends largley on fitness convergence. It would not be beneficial to have algorithm that runs at double speed, but need also twice the steps to reach the same fitness value, as we would have to wait the same time to get similar solutions.

Following experiments provides results showing the convergence of fitness for each of the methods and variants. Graph on figure 7.2 displays arithmetic means of best and average fitness of population after 50 independent runs of GA.

As the surface of CGP fitness function is much more complex than the simple Onemax problem, the fitness convergence is measured on CGP problem. Each run consisted of 1000 generations (as the methods are asynchronous, one generation is defined as number

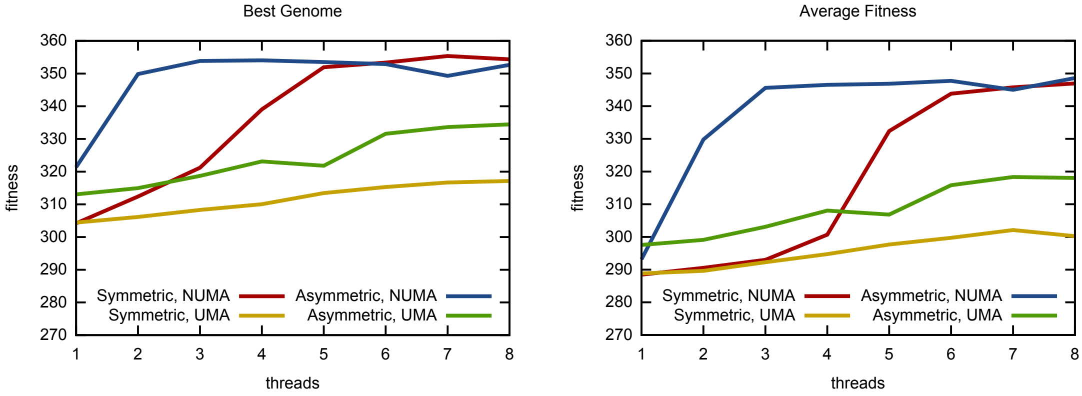

Figure 7.2: Graphs showing best and average fitness achieved on the end of the 1000th generation. The graph displays relation between number of threads and fitness values.

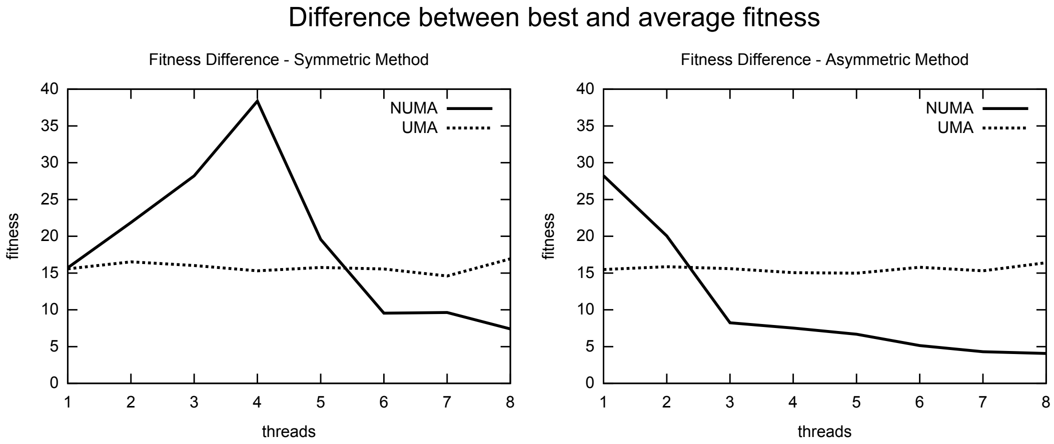

Figure 7.3: Difference between best and average fitness.

of evaluations equal to population size), because the solutions have converged by then and population fitness just slowly rose in the follofing generations.

We can see that the fitness achieved at the end of the evolution rises with the thread count. Difference between best and average fitness, which is displayed on figure 7.3, can be considered as one of the indicators of variance in the population.

Plot on figure 7.4 provides the values of average and best fitness during all the steps of evolution process. Both methods were run with 8 threads for 1000 generations

7.2.1 Conclusion

Result of the convergence tests shows that the fitness values tends to rise with rising thread count. Figure 7.2 also shows that NUMA variants are more effective than their UMA counterparts. Better convergence is mirrored by smaller variations in population displayed on figure 7.3. We can also notice that Asymmetric Method achieves better values and converge faster than Symmetric Method (figure 7.4).

This effect could be caused by the elitism and migration technique used in UMA and NUMA variants. In case of Symmetric Method, the number of subpopulations that global

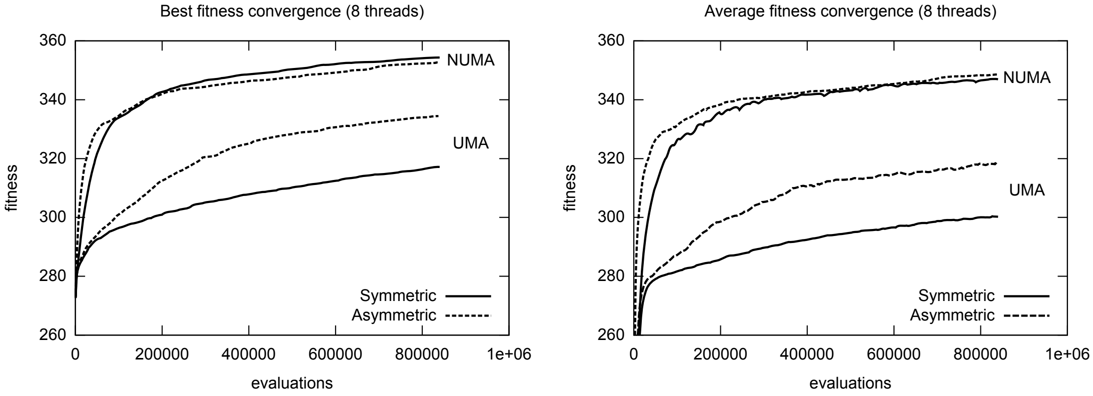

Figure 7.4: Convergence of best and average fitness values as it develops during the run of GA.

population is divided into equals to the number of threads t. As the elitism is carried out on subpopulation level, the number of locally optimal genomes copied into new generation is equal t in UMA variant and t^2 in NUMA variant. In Asymmetric Method, the number of subpopulations is even bigger: $2t + 1$. This migration is possible due to the fact that each subpopulation keeps local information about the best thread.

	Symmetric		Asymmetric	
Thr	SMP	NUMA	SMP	NUMA
1	1 (0%)	1 (0%)	3 (0%)	9 (1%)
2	2 (0%)	4 (0%)	5 (1%)	25 (3%)
3	3 (0%)	9 (1%)	7 (1%)	49 (6%)
4	4 (0%)	16 (2%)	9 (1%)	81 (10%)
5	5 (1%)	25 (3%)	11 (1%)	121 (14%)
6	6 (1%)	36 (4%)	13 (2%)	169 (20%)
7	7 (1%)	49 (6%)	15 (2%)	225 (27%)
8	8 (1%)	64 (8%)	17 (2%)	289 (34%)

Table 7.1: Number of genomes propagating to the new generation by each method.

Number of locally best genomes migrating to the new generation is displayed in table 7.1. We can see that as we increase the subpopulation count, the number of genomes copied from "old" generation increases – we are approaching kind of high-pressure variant of a steady-state genetic algorithm. SSGA are known to have faster convergence rates at the expense of smaller variance [28], so it could be possible explanation of acquired data.

Figures 7.2 and 7.3 also displays an unexpected raise in fitness for Symmetric NUMA Method between 4 and 5 threads. Comparable raise can be observed in Asymmetric NUMA Method between 2 and 3 threads. According to table 7.1, both of these two points share the number of migrating genomes (approximatly 3%-6% of whole population). It could represent a critical migration value to achieve the best convergence and together with NUMA's better convergence should be the subject of further research.

7.3 Heavy Load

Following experiments test the ability of each method to converge on system with moderate to heavy load. They are designed to test the assumption that Symmetric Method suffers from lack of computing resources. To simulate system with load, the tests were run with more threads than the number of processors.

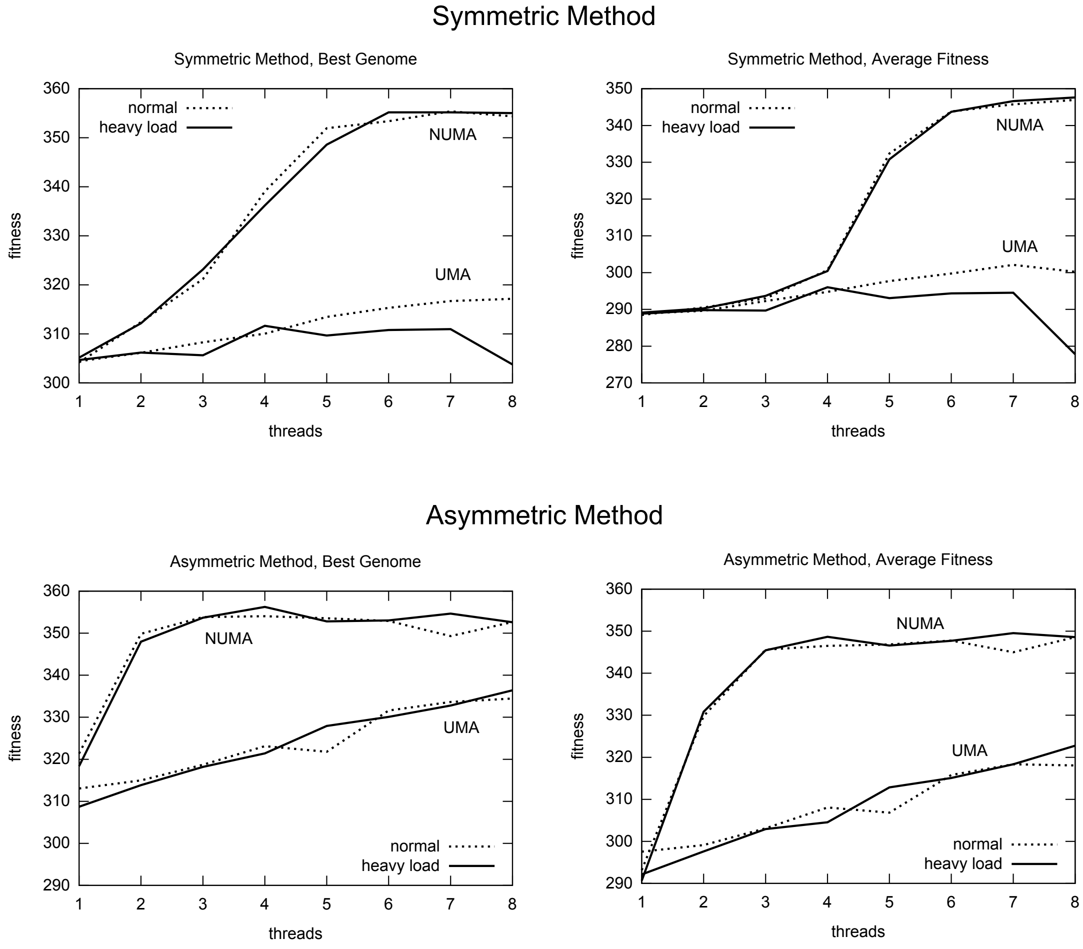

Figure 7.5: Graphs show the relation between fitness achieved on system with no load and system with moderate to heavy load. Plot label *normal* refers to runs on edesign2 system with while label *heavy load* describes data measured on edesign1.

Two systems were used: edesign2 with two Quad-Core CPUs to simulate normal behaviour and edesign1 with two Dual-Core CPUs to simulate system with moderate to heavy load (as it has only 4 cores, 5 to 8 threads could not run simultaneously). More informations about each system can be found in section 7.1 and in appendix C. Figure 7.5 displays average and best fitness values achieved by the GA under different conditions.

7.3.1 Conclusion

From graphs on figure 7.5 is obvious that the only variant suffering from lack of processor time is UMA variant of Symmetric Method. These results confirms the assumptions about the UMA selection. As the selection operates on whole population, sub-optimal genomes spread from poor subpopulations. On the other hand, UMA variant of Asymmetric Method is achieving the same convergence under heavy load as under normal conditions, thus the Asymmetric Method succeeded in solving the lack-of-resources issue of Symmetric Method and successfully accomplished its design goal.

However, the NUMA variants of both Methods produce almost identical results independenly of system load. This result is interesting because it renders the Asymmetric Method unnecessary as the heavy load problem of Symmetric Method can be solved by its NUMA variant, which has better results than UMA variant in all the aspects.

Chapter 8

Conclusion

This work is presenting comparison of two different approaches to paralellization of GA. These two methods - Symmetric Method and Asymmetric Method - are presented in chapter 6 and their performance is evaluated in chapter 7. Final chapter sums up the conclusions based on acquired results and presents possible direction for future research of this subject.

8.1 Results

The experiments measured three main aspects of each method:

- Speedup
- Convergence
- Performance under heavy load

The quality of each aspect is measured by running different optimization problems described in section 6.7.

8.1.1 Speedup

The ability to achieve effective speedup on multiple processors is essential to achieve good performance. Figure 7.1 displays speedups of each method and variant. As can be seen on graphs, the speedup depends on many factors as fitness function or hardware configuration and architecture. Overall, NUMA variant of Symmetric Method has the best performance, achieving linear or almost linear speedup on all of the architectures.

8.1.2 Convergence

Another important indicator of GA performance is its ability to find near optimal solution quickly. As the quality of solution is measured by the fitness function, this ability can be represented as convergence of fitness values in population toward optimal value. Figures 7.2, 7.3 and 7.4 in section 7.2 compare the development and final values of population fitness of each method.

We can see that NUMA variants are achieving slightly better convergence rates and fitness values for the CGP fitness function. This improvement of convergence is achieved at the expense of variance in the population. The relation between Asymmetric Method and steady-state genetic algorithm is discussed at the end of section 7.2. Overall, Asymmetric

Method (especially its NUMA variant) shows better convergence rates that Symmetric Method. Also NUMA selection is producing better results than its UMA counterpart.

8.1.3 Heavy Load

Final tests were dedicated to the performance under the system with moderate to heavy load. Graphs on figure 7.5 shows that the methods are not affected much by system load or lack of resources, with UMA variant of Symmetric Method being the exception (however, the difference in fitness values is only marginal).

8.2 Future work

The multi-deme approach represented by NUMA variant of Symmetric Method combined with shared memory of multiprocessor systems shows promising results in the area of performance enhancing for genetic algorithms. As SSGA-like aspect of Asymmetric Methods provides better convergence than Symmetric Method, so optimal parallelization of GA could be achieved by combination of these two properties.

Another interesting phenomenon is superlinear speedup achieved on some systems. The cause of this speedup should be analyzed and conclusion of this analysis should be used in design and implementation of this new combined method.

As genetic algorithms are complicated dynamic systems with many parameters, it is difficult to find general assumptions that would hold true in all the cases. Parallel GA are no different. There are always more test cases that can measure different combination of various parameters to quantify the performance more accurately. Unfortunately, it is out of scope of this work to perform all of these tests, so additional tests can be the subject of the future research. Following list provides some of the aspects that could be measured to assert the performance of both methods:

- Speedup dependance on the size of population and subpopulations count
- Relation between convergence and subpopulation count
- More fitness functions
- Comparison of convergence with other methods (SSGA, master-slave, . . .)
- More detailed statistics (population variance, standard deviation, . . .)
- More migration strategies

Bibliography

[1] Thomas Baeck, David Fogel, and Zbigniew Michalewicz. *The Handbook of Evolutionary Computation.* Oxford University Press, 1997.

[2] Glen Beane. The effects of microprocessor architecture on speedup in distributed memory supercomputers. Technical report, The University of Maine, 2004.

[3] Erick Cantu-Paz. A survey of parallel genetic algorithms. Technical report, Illinois Genetic Algorithms Laboratory, University of Illinois, 1998.

[4] Erick Cantu-Paz. *Efficient and Accurate Parallel Genetic Algorithms.* Springer, 2000.

[5] Erick Cantú-Paz. On random numbers and the performance of genetic algorithms. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference.* Morgan Kaufmann Publishers Inc., 2002.

[6] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming.* MIT Press, 2007.

[7] Janet Clegg, James Alfred Walker, and Julian Frances Miller. A new crossover technique for cartesian genetic programming. *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 2007.

[8] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, University of Michigan, 1975.

[9] Ulrich Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., 2007.

[10] Vaclav Dvorak. *Architektura a Programovani Paralelnich Systemu.* VUTIUM Brno, 2004.

[11] Marin Golub and Domagoj Jakobovic. A new model of global parallel genetic algorithm. Technical report, Faculty of Electrical Engineering and Computing, University of Zagreb, 2000.

[12] John Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 1988.

[13] Randy Haupt and Sue Ellen Haupt. *Practical Genetic Algorithms.* Wiley-Interscience, 2004.

[14] Intel. Multiprocessors, clusters, grids and parallel computing. Internet, http://www.intel.com/cd/ids/developer/asmo-na/eng/95581.htm.

[15] Intel. Optimizing software applications for numa. Technical report, Intel, 2009.

[16] Earl Joseph, Christopher G. Willard, and Nicholas J. Kaufmann. The amd opteron processor: A new alternative for technical computing. Technical report, AMD, 2003.

[17] John Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[18] Los Alamos National Laboratory. Roadrunner system overview.

[19] Manuel Lozano, Francisco Herrera, and José Ramón Cano. Replacement strategies to preserve useful diversity in steady-state genetic algorithms. *Information Sciences*, 2008.

[20] Adam Marczyk. Genetic algorithms and evolutionary computation. Internet, http://www.talkorigins.org/faqs/genalg/genalg.html, 2004.

[21] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 2003.

[22] Brad Miller and David Goldberg. Genetic algorithms, tournament selection and the effect od noise. Technical report, Department of General Engineeringm, University of Illinois, 1995.

[23] J. F. Millerand and P.Thomson. Cartesian genetic programming. *Proceedings of the 3rd European Conference on Genetic Programming (EuroGP2000)*, 2000.

[24] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1998.

[25] Collective of Authors. Gnu compiler collection manual.

[26] Collective of Authors. Wikipedia, the free encyclopedia. Internet, http://www.wikipedia.com.

[27] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE Micro*, 1997.

[28] Alex Rogers and Adam Prügel-Bennett. Modelling the dynamics of a steady state genetic algorithm. In *Foundations of Genetic Algorithms 5.* Morgan Kaufmann, 1999.

[29] Yuan Shi. Reevaluating amdahl's law and gustafson's law. Technical report, Computer and Information Sciences Department, Temple University, Philadelphia, 1996.

List of used abbreviations and symbols

CGP – Cartesian Genetic Programming

CPU – Central Processing Unit

GA – Genetic Algorithm

GP – Genetic Programming

NUMA – Non-Uniform Memory Access

PGA – Parallel Gentic Algorithm

PRNG – Pseudo-Random Number Generator

SSGA – Steady-state genetic algorithm

SMP – Symmetric Multiprocessing

UMA – Uniform Memory Access

Appendix A

Atomic Increments

Source code for atomic increments benchmark.

```
#define ITERS 10000000
#define T 8
void test() {
  #pragma omp parallel for shared(a) num_threads(T)
  for(int i = 0; i < ITERS; i++) {
     a++;
  }
}
void testsync() {
  #pragma omp parallel for shared(a) num_threads(T)
  for(int i = 0; i < ITERS; i++) {
     __sync_fetch_and_add(&a, 1);
  }
}
void testomp() {
  #pragma omp parallel for shared(a) num_threads(T)
  for(int i = 0; i < ITERS; i++) {
     #pragma omp atomic
       a++;
  }
}
void testompcrit() {
  #pragma omp parallel for shared(a) num_threads(T)
  for(int i = 0; i < ITERS; i++) {
     #pragma omp critical
       a++;
  }
}
```

Appendix B

Xorshift Pseudo-Random Number Generator

B.1 Implementation of xorshift PRNG

Source: [21]

```
static unsigned int x=123456789,y=362436069,z=521288629,w=88675123;

inline unsigned int xor128(void) {
    unsigned int t=x^(x<<11);
    x=y; y=z; z=w; return w=(w^(w>>19))^(t^(t>>8));
}
```

B.2 Initialization with seed value

```
inline void randseed(unsigned int seed)
{
  x ^= seed;  y ^= seed;  z ^= seed;  w ^= seed;
}
```

Appendix C

Speedup Data

C.1 System Specifications

```
edesign1
Hardware:
2xDual Core AMD Opteron 2220
1024 KB cache, 32 GB RAM
OS:
Linux version 2.6.32.12
(gcc version 4.3.4 (GCC) ) #1 SMP Tue Apr 27 15:10:42 CEST 2010

edesign2
Hardware:
2xQuad Core Intel Xeon 5355
4096 KB cache, 32 GB RAM
OS:
Linux version 2.6.32.12
(gcc version 4.3.4 (GCC) ) #1 SMP Tue Apr 27 15:10:42 CEST 2010

merlin
Hardware:
2x Quad-Core AMD Opteron 2387 2.8GHz
512 KB cache, 16 GB RAM
OS:
Linux version 2.6.32.12
(gcc version 4.3.4 (GCC) ) #1 SMP Tue Apr 27 15:10:42 CEST 2010

pcjaros-gpu
Hardware:
Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz overclocked to 3.32Ghz (Quad-Core)
8192 KB cache, 12 GB RAM
OS:
Linux version 2.6.31-21-generic
(gcc version 4.4.1 (Ubuntu 4.4.1-4ubuntu9) ) #59-Ubuntu SMP
```

C.2 Running times

Parameters of genetic algorithm used for speedup benchmark are in table C.1.

	CGP	Onemax
Population	840	840
Genome size (bits)	9792	1024
Generations	5000	5000

Table C.1: Parameters of GA used for benchmark

Values were calculated as an arithmetic mean and its standard deviation from 50 independent runs. Example table C.2 describes measured parameters.

System name						
Method name						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	19.75 (0.33)	4.70 (0.08)	1.00	13.01 (0.10)	3.10 (0.02)	1.00
2	9.54 (0.14)	2.27 (0.03)	2.07	6.44 (0.02)	1.53 (0.01)	2.02

Table C.2: Data table example. System nam is on top followed by Method name. *CGP* and *Onemax* are the names of fitness functions used for benchmark. *Thr* is number of threads, *Time* is totoal time of run in seconds, *Time/Eval* is total time divided by fitness evaluations count (virtual time of one evaluation) and *Spd* denotes speedup in comparison with single-thread version. Values in parentheses represents standard deviations.

edesign1						
Symmetric Method NUMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	19.75 (0.33)	4.70 (0.08)	1.00	13.01 (0.10)	3.10 (0.02)	1.00
2	9.54 (0.14)	2.27 (0.03)	2.07	6.44 (0.02)	1.53 (0.01)	2.02
3	6.37 (0.07)	1.52 (0.02)	3.10	4.30 (0.02)	1.02 (0.00)	3.03
4	4.84 (0.09)	1.15 (0.02)	4.08	3.20 (0.05)	0.76 (0.01)	4.06
Symmetric Method UMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	20.24 (0.31)	4.82 (0.07)	1.00	13.81 (0.01)	3.29 (0.00)	1.00
2	10.71 (0.16)	2.55 (0.04)	1.89	7.11 (0.01)	1.69 (0.00)	1.94
3	7.48 (0.09)	1.78 (0.02)	2.70	4.81 (0.01)	1.15 (0.00)	2.87
4	5.75 (0.06)	1.37 (0.01)	3.52	3.68 (0.01)	0.88 (0.00)	3.76
Asymmetric Method NUMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	19.49 (0.20)	4.64 (0.05)	1.00	12.91 (0.10)	3.07 (0.02)	1.00
2	10.45 (0.28)	2.49 (0.07)	1.87	6.76 (0.02)	1.61 (0.01)	1.91
3	7.18 (0.33)	1.71 (0.08)	2.71	4.34 (0.01)	1.03 (0.00)	2.98
4	5.50 (0.32)	1.31 (0.08)	3.53	3.20 (0.01)	0.76 (0.00)	4.02
Asymmetric Method UMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	20.00 (0.31)	4.76 (0.07)	1.00	13.64 (0.02)	3.25 (0.00)	1.00
2	10.81 (0.16)	2.57 (0.04)	1.85	7.18 (0.01)	1.71 (0.00)	1.90
3	7.83 (0.12)	1.86 (0.03)	2.56	4.98 (0.01)	1.19 (0.00)	2.74
4	6.29 (0.08)	1.50 (0.02)	3.17	3.89 (0.01)	0.93 (0.00)	3.50

Table C.3: Running times and speedup of both methods on edesign1 system

edesign2						
Symmetric Method NUMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	13.58 (0.21)	3.23 (0.05)	1.00	11.11 (0.01)	2.64 (0.00)	1.00
2	6.83 (0.06)	1.63 (0.02)	1.99	5.51 (0.01)	1.31 (0.00)	2.01
3	4.59 (0.04)	1.09 (0.01)	2.96	3.65 (0.01)	0.87 (0.00)	3.04
4	3.47 (0.05)	0.83 (0.01)	3.91	2.69 (0.01)	0.64 (0.00)	4.14
5	2.80 (0.10)	0.67 (0.02)	4.84	2.11 (0.01)	0.50 (0.00)	5.26
6	2.34 (0.08)	0.56 (0.02)	5.80	1.73 (0.01)	0.41 (0.00)	6.42
7	2.02 (0.09)	0.48 (0.02)	6.73	1.46 (0.01)	0.35 (0.00)	7.60
8	1.77 (0.08)	0.42 (0.02)	7.68	1.26 (0.00)	0.30 (0.00)	8.79
Symmetric Method UMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	14.28 (0.18)	3.40 (0.04)	1.00	11.33 (0.01)	2.70 (0.00)	1.00
2	7.64 (0.07)	1.82 (0.02)	1.87	5.83 (0.01)	1.39 (0.00)	1.94
3	5.35 (0.05)	1.27 (0.01)	2.67	3.95 (0.00)	0.94 (0.00)	2.87
4	4.18 (0.03)	1.00 (0.01)	3.41	2.99 (0.00)	0.71 (0.00)	3.79
5	3.43 (0.03)	0.82 (0.01)	4.16	2.40 (0.00)	0.57 (0.00)	4.72
6	2.95 (0.03)	0.70 (0.01)	4.84	2.00 (0.00)	0.48 (0.00)	5.68
7	2.62 (0.03)	0.62 (0.01)	5.45	1.71 (0.00)	0.41 (0.00)	6.62
8	2.34 (0.02)	0.56 (0.00)	6.09	1.49 (0.00)	0.36 (0.00)	7.59
Asymmetric Method NUMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	13.52 (0.21)	3.22 (0.05)	1.00	10.96 (0.04)	2.61 (0.01)	1.00
2	8.24 (0.19)	1.96 (0.05)	1.64	5.69 (0.02)	1.35 (0.01)	1.93
3	5.89 (0.13)	1.40 (0.03)	2.30	3.68 (0.01)	0.88 (0.00)	2.98
4	4.60 (0.14)	1.10 (0.03)	2.93	2.69 (0.01)	0.64 (0.00)	4.06
5	3.88 (0.11)	0.93 (0.03)	3.47	2.12 (0.00)	0.51 (0.00)	5.14
6	3.38 (0.11)	0.81 (0.03)	3.96	1.75 (0.00)	0.42 (0.00)	6.19
7	3.15 (0.07)	0.75 (0.02)	4.29	1.53 (0.00)	0.36 (0.00)	7.17
8	3.00 (0.06)	0.72 (0.01)	4.47	1.35 (0.00)	0.32 (0.00)	8.06
Asymmetric Method UMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	14.11 (0.17)	3.36 (0.04)	1.00	11.24 (0.02)	2.68 (0.00)	1.00
2	7.89 (0.09)	1.88 (0.02)	1.79	5.87 (0.01)	1.40 (0.00)	1.91
3	5.79 (0.07)	1.38 (0.02)	2.44	3.97 (0.01)	0.95 (0.00)	2.83
4	4.57 (0.04)	1.09 (0.01)	3.08	2.97 (0.01)	0.71 (0.00)	3.77
5	3.76 (0.03)	0.90 (0.01)	3.73	2.35 (0.01)	0.56 (0.00)	4.75
6	3.24 (0.03)	0.78 (0.01)	4.31	1.94 (0.00)	0.47 (0.00)	5.73
7	2.92 (0.03)	0.69 (0.01)	4.84	1.68 (0.00)	0.40 (0.00)	6.68
8	2.65 (0.02)	0.64 (0.00)	5.28	1.46 (0.00)	0.35 (0.00)	7.62

Table C.4: Running times and speedup of both methods on edesign2 system

merlin						
Symmetric Method NUMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	7.50 (0.15)	4.46 (0.09)	1.00	5.14 (0.05)	3.06 (0.03)	1.00
2	3.74 (0.05)	2.23 (0.03)	2.00	2.53 (0.01)	1.51 (0.01)	2.03
3	2.50 (0.03)	1.49 (0.02)	2.99	1.68 (0.01)	1.00 (0.01)	3.05
4	1.89 (0.03)	1.13 (0.02)	3.97	1.25 (0.01)	0.74 (0.00)	4.12
5	1.51 (0.03)	0.90 (0.02)	4.97	1.00 (0.01)	0.59 (0.01)	5.16
6	1.27 (0.03)	0.76 (0.02)	5.89	0.82 (0.01)	0.49 (0.01)	6.26
7	1.09 (0.04)	0.65 (0.02)	6.89	0.70 (0.00)	0.42 (0.00)	7.35
8	0.97 (0.04)	0.57 (0.02)	7.76	0.61 (0.01)	0.36 (0.01)	8.43
Symmetric Method UMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	7.77 (0.12)	4.63 (0.07)	1.00	5.38 (0.05)	3.20 (0.03)	1.00
2	4.12 (0.05)	2.45 (0.03)	1.89	2.82 (0.02)	1.68 (0.01)	1.91
3	2.83 (0.06)	1.69 (0.03)	2.74	1.92 (0.01)	1.14 (0.00)	2.80
4	2.18 (0.03)	1.30 (0.02)	3.57	1.46 (0.00)	0.87 (0.00)	3.68
5	1.83 (0.02)	1.09 (0.01)	4.25	1.19 (0.01)	0.71 (0.01)	4.51
6	1.59 (0.02)	0.94 (0.01)	4.90	0.99 (0.00)	0.59 (0.00)	5.44
7	1.39 (0.01)	0.83 (0.01)	5.58	0.88 (0.01)	0.52 (0.00)	6.12
8	1.25 (0.02)	0.74 (0.01)	6.24	0.79 (0.03)	0.47 (0.02)	6.84
Asymmetric Method NUMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	7.09 (0.21)	4.22 (0.12)	1.00	5.05 (0.05)	3.01 (0.03)	1.00
2	4.06 (0.10)	2.42 (0.06)	1.75	2.72 (0.02)	1.62 (0.01)	1.86
3	2.77 (0.09)	1.65 (0.05)	2.55	1.73 (0.01)	1.03 (0.00)	2.91
4	2.14 (0.06)	1.28 (0.04)	3.30	1.27 (0.01)	0.76 (0.00)	3.96
5	1.77 (0.04)	1.06 (0.02)	3.99	1.01 (0.01)	0.60 (0.00)	4.99
6	1.53 (0.07)	0.92 (0.04)	4.58	0.84 (0.00)	0.50 (0.00)	5.98
7	1.38 (0.03)	0.82 (0.02)	5.14	0.73 (0.01)	0.43 (0.00)	6.92
8	1.28 (0.06)	0.77 (0.03)	5.49	0.64 (0.01)	0.38 (0.00)	7.84
Asymmetric Method UMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	7.82 (0.15)	4.65 (0.09)	1.00	5.46 (0.06)	3.25 (0.04)	1.00
2	4.21 (0.07)	2.50 (0.04)	1.86	2.94 (0.01)	1.75 (0.01)	1.86
3	3.03 (0.05)	1.81 (0.03)	2.58	2.03 (0.01)	1.21 (0.01)	2.69
4	2.46 (0.04)	1.47 (0.03)	3.17	1.58 (0.01)	0.94 (0.00)	3.44
5	2.13 (0.03)	1.27 (0.02)	3.65	1.33 (0.01)	0.80 (0.00)	4.08
6	1.97 (0.03)	1.18 (0.02)	3.94	1.18 (0.01)	0.71 (0.01)	4.57
7	1.89 (0.04)	1.13 (0.02)	4.13	1.11 (0.00)	0.66 (0.00)	4.92
8	1.87 (0.03)	1.12 (0.02)	4.15	1.08 (0.02)	0.65 (0.01)	5.02

Table C.5: Running times and speedup of both methods on merlin system

pcjaros-gpu						
Symmetric Method NUMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	6.56 (0.14)	1.56 (0.03)	1.00	7.10 (0.01)	1.69 (0.00)	1.00
2	3.27 (0.05)	0.78 (0.01)	2.00	3.49 (0.01)	0.83 (0.00)	2.04
3	2.23 (0.03)	0.53 (0.01)	2.94	2.33 (0.02)	0.55 (0.00)	3.06
4	1.70 (0.05)	0.41 (0.01)	3.85	1.70 (0.02)	0.41 (0.00)	4.18
Symmetric Method UMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	6.58 (0.11)	1.57 (0.03)	1.00	7.24 (0.01)	1.72 (0.00)	1.00
2	3.28 (0.07)	0.78 (0.02)	2.01	3.62 (0.01)	0.86 (0.00)	2.00
3	2.22 (0.05)	0.53 (0.01)	2.97	2.42 (0.02)	0.58 (0.00)	2.99
4	1.66 (0.09)	0.40 (0.02)	3.96	1.83 (0.02)	0.43 (0.00)	3.97
Asymmetric Method NUMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	6.56 (0.12)	1.56 (0.03)	1.00	6.94 (0.05)	1.65 (0.01)	1.00
2	3.29 (0.17)	0.78 (0.04)	2.00	3.31 (0.02)	0.79 (0.01)	2.10
3	2.15 (0.14)	0.51 (0.03)	3.04	2.17 (0.02)	0.52 (0.00)	3.20
4	1.70 (0.14)	0.41 (0.03)	3.84	1.64 (0.03)	0.39 (0.01)	4.22
Asymmetric Method UMA						
	CGP			Onemax		
Thr	Time [s]	Time/Eval [ms]	Spd	Time [s]	Time/Eval [ms]	Spd
1	6.54 (0.15)	1.56 (0.04)	1.00	7.12 (0.02)	1.69 (0.00)	1.00
2	3.33 (0.08)	0.79 (0.02)	1.97	3.55 (0.02)	0.85 (0.00)	2.00
3	2.30 (0.06)	0.55 (0.01)	2.84	2.41 (0.02)	0.57 (0.00)	2.95
4	1.81 (0.05)	0.43 (0.01)	3.61	1.86 (0.02)	0.44 (0.00)	3.81

Table C.6: Running times and speedup of both methods on pcjaros-gpu system