Future Generation Computer Systems 38 (2014) 75–91

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Architecture-based design and optimization of genetic algorithms on multi- and many-core systems

Long Zheng [a,b], Yanchao Lu [a], Minyi Guo [a,*], Song Guo [b], Cheng-Zhong Xu [c]

[a] University of Shanghai Jiao Tong University, Shanghai, 200240, China
[b] University of Aizu, Aizu-Wakamatsu, 965-8580, Japan
[c] Wayne State University, Detroit, MI 48202, United States

HIGHLIGHTS

- Evaluate different PGA schemes and propose the best one for each architecture.
- General optimization approaches and rules are analyzed and proposed.
- Practical comparisons of GA performance on multi-core and many-core are discussed.
- Besides execution speed, solution quality is also concentrated on and analyzed.
- All work is based on features of architectures, instead of specific GA problems.

ARTICLE INFO

Article history:
Received 27 April 2012
Received in revised form
5 April 2013
Accepted 14 September 2013
Available online 10 October 2013

Keywords:
Genetic algorithm
Multi-core
GPU
Accuracy
Architecture
Speedup

ABSTRACT

A Genetic Algorithm (GA) is a heuristic to find exact or approximate solutions to optimization and search problems within an acceptable time. We discuss GAs from an architectural perspective, offering a general analysis of performance of GAs on multi-core CPUs and on many-core GPUs. Based on the widely used Parallel GA (PGA) schemes, we propose the best one for each architecture. More specifically, the Asynchronous Island scheme, Island/Master–Slave Hierarchy PGA and Island/Cellular Hierarchy PGA are the best for multi-core, multi-socket multi-core and many-core architectures, respectively. Optimization approaches and rules based on a deep understanding of multi- and many-core architectures are also analyzed and proposed. Finally, the comparison of GA performance on multi-core and many-core architectures are discussed. Three real GA problems are used as benchmarks to evaluate our analysis and findings.

There are three extra contributions compared to previous work. Firstly, our findings based on deeply analyzing architectures can be applied to all GA problems, even for other parallel computing, not for a particular GA problem. Secondly, the performance of GAs in our work not only concerns execution speed, also the solution quality has not been considered seriously enough. Thirdly, we propose the theoretical performance and optimization models of PGA on multi-core and many-core architectures, finding a more practical result of the performance comparison of the GA on these architectures, so that the speedup presented in this work is more reasonable and is a better guide to practical decisions.

1. Introduction

Nowadays, multi-core processors and many-core GPUs have entered the mainstream of microprocessor development. The multi-core and many-core architecture both successfully make use of Thread Level Parallelism (TLP) to improve the performance rather than just Instruction Level Parallelism (ILP).

When microprocessors use ILP to improve the performance, all parallel mechanisms are hidden by compilers and the architecture of microprocessors. However with TLP, parallel mechanisms cannot be hidden. Users have to know about the architecture of multi-core and many-core systems, and implement their parallel programs explicitly. For example, users need to write multi-threaded code to improve the parallelism of their programs. For GPUs, users even have to be familiar with the details of the architecture, because they must assign threads to different SMs, use hundreds of cores efficiently, and consider the choice of different types of memory. With multi-core systems, several existing or new programming models and environments can help users. For example, Pthread, OpenMP, Cilk [1], and even MapReduce [2] can be considered tools to help users implement programs on multi-core

* Corresponding author.
E-mail addresses: d8112104@u-aizu.ac.jp (L. Zheng), chzblych@sjtu.edu.cn (Y. Lu), guo-my@cs.sjtu.edu.cn (M. Guo), sguo@u-aizu.ac.jp (S. Guo), czxu@wayne.edu (C.-Z. Xu).

0167-739X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.future.2013.09.029

systems. GPU-based systems are newer than multi-core systems, but work has been done in both industry and academia. Users can use CUDA, OpenCL to interact with hundreds of cores. Recently, a framework called MARS [3] is proposed, so that MapReduce can also help to harness the power of many-core GPUs. These systems do not try to hide the parallelism; they expose it to users, which requires users to know about architectural details and parallel mechanisms; otherwise, the systems cannot perform efficiently.

A Genetic Algorithm (GA) is a heuristic to find exact or approximate solutions to optimization and search problems within an acceptable time; the technique is widely used in business, engineering and science [4–7]. Actually, many real world engineering problems that solved by GA require quite a long computation time. Furthermore, GA is integrated in many other applications, and stands as a key component (such as scheduling) which has a significant influence on the overall performance. These facts are the motivation that multi- and many-core architectures are adopted, focused and researched [8–11].

Although some transplanting of GAs from CPUs to GPUs has been done, the lack of understanding of detailed multi-core and many-core architectures leads to the following shortcomings in the previous work:

1. Many GA which work on multi-core and many-core systems are done on a case by case basis, describing how to use multi-core and many-core systems to accelerate the specific GA problems. Some work even misunderstands the fundamentals of implementation on the new architecture. The different GA problems have more commonalities than differences. Features of multi- and many-core architectures actually determine which Parallel GA (PGA) is the best for both execution speed and solution quality, rather than features of different GA problems.
2. Most work focuses exclusively on the relationship between the speedup on the multi-core and many-core architecture, pursuing a fast execution time, but ignoring the solution quality. Since GAs mostly find only approximate solutions, more attention should be paid to solution quality. The relationship between speedup and architecture should be discussed along with the solution quality.
3. Most previous work demonstrates the GPU performance by comparing the execution speed on GPUs to a serial implementation on CPUs. Although this metric is reasonable, the serial implementation can only use a single CPU core. Now multi-core CPU is the mainstream. The lack of expressing the speedup of GPUs compared to multi-core CPUs makes the speedup in previous work less practical to GA users.

These problems motivate us to discuss GAs from an architecture perspective, to offer a general analysis of GAs on multi-core and many-core architectures, considering the quality of solutions. We first introduce the fundamentals of GAs followed by several popular PGA schemes; then we make an architecture-based analysis of the different PGA models on multi-core and many-core architectures. The best PGA schemes for multi-core, multi-socket multi-core and many-core architectures are proposed as well. Also, we propose some approaches and rules to optimize the performance of GAs on multi- and many-core architectures. Finally, three real and widely used GA problems are used as benchmarks, so that our findings can be validated.

Especially, our work concentrates on features of architectures which are considered as the key points to improve GA performance. Hence our work aims at all kinds of GAs not a specific GA problem. We consider the GAs from the abstract level of a parallel model; performance improvements in GA fundamentals – for example, migration topology, mutation strategy or selection method – will not affect our findings. These fundamental GA improvements can get more performance benefit from the architecture with our findings. Existing multi- and many-core acceleration work for GAs mostly aims to reduce the execution time, ignoring the solution quality. We take the solution quality into account, so that we try to make GAs get the best solutions in the shortest time, since a solution is what the engineers and researchers who use GAs actually need. As we propose the best approaches to make GA use multi-core and many-core architectures, we can make a fairer comparison of GA performance between multi-core and many-core. Besides the speedup of execution time, a speedup with consideration of solution quality is also proposed. The speedup based on our experimental results is more reasonable and is a better guide to practical decisions when engineers and researchers use GAs to solve their problems.

We highlight our contributions compared to previous work as follows.

1. Analyze the performance of generic GAs on both multi-core and many-core architectures in depth, and proposing the best PGA scheme and implementation on each architecture.
2. Consider a more reasonable performance metric that combines the execution speed and the solution quality to compare the performance of different PGA schemes on different architectures.
3. Propose the theoretical performance and optimization models of PGA on multi-core and many-core architectures and evaluate them with three real world engineering problems, finding a more practical and bias result of the performance comparison of the GA on these architectures.

The remainder of this paper is structured as follows. Section 2 gives an essential overview of GA. Sections 3 and 4 offer an architecture-based theoretical analysis of GAs and propose the best approaches and optimization rules for multi-core and many-core systems. We evaluate our analysis and findings in Section 5. Related work is discussed in Section 6. Section 7 summarizes our findings.

2. Background of GA

Before analyzing GA on multi-core and many-core architectures, we give a quick overview of GAs. In this section, we begin with a review of species selection and evolution in nature, which is a good way to understand the fundamentals of GAs. Based on this, we present several GA schemes for parallel and distributed computing environments.

2.1. Fundamental of GA

In nature, individuals compete with each other and adapt to the environment. Only the strongest ones can survive in a tough environment. The survivors mate more-or-less randomly and produce the next generation. During reproduction, mutation always occurs, which makes some individuals of the next generation better fitted for the environment.

GAs are heuristic search algorithms that mimic natural species selection and evolution as described above. The problem that a GA intends to solve is the tough environment. Each individual in the population of a GA is a candidate solution for the problem.

A generation of a GA is generated by the following steps—*fitness computation*, *selection*, *crossover* and *mutation*. The *fitness computation* is the competition of individuals, and can tell which individual is good for the problem; the *selection* chooses good individuals to survive and eliminates bad ones; the *crossover* mates two individuals to produce the next generation individuals; and the *mutation* occurs after *crossover*, so that the next generation can be more diverse. With enough generations, GAs can evolve an individual that is the optimal solution to the problem. This is the classic serial GA scheme [12]. Since GAs are so similar to the

biological species evolution, many theories of GAs are motivated and explained by biological theory.

2.2. Overview of parallel GA

Following the GA philosophy, the Parallel GA (PGA) is proposed to take advantage of parallel and distributed systems to speedup GA computing. Although PGA makes some essential modification to fit parallel and distributed systems, the effect of PGA is equivalent to GA—all applications/algorithms implemented in GA can also be implemented in PGA. With the perspective of effect, PGA is only different from GA in execution speed and solution quality. A quick overview and discussion of various PGAs are presented as follows.

Course-grained PGA has three variations—Master–Slave, Synchronous Island and Asynchronous Island schemes.

In the Master–Slave scheme, one node is chosen as the master node and the other nodes become the slaves. The master node maintains the population of a GA. The master node assigns individuals to slave nodes to parallelize the calculation of the fitness. After the fitness of each individual is sent back to the master node, the master node does the *selection*, *crossover* and *mutation*. The Master–Slave scheme is a tightly coupled structure. In this scheme, the master node spreads the best results of population of each generation to all individuals. Actually, Master–Slave exactly follows the original fundamental of GA. Others transform the original GA more or less.

In the Island scheme, the population of a GA is divided into several sub-populations, called islands. The populations in islands evolve in isolation from each other. The Island model isolates the population into several islands, hence, an extra step – *migration* – is introduced to spread the best results across all islands. Generally, each island migrates their best results every fixed generation. The difference between the Synchronous Island and Asynchronous Island schemes lies only in the method of *migration*. In the Synchronous Island scheme, migration takes place when all islands achieve a fixed number of generations, but migration in the Asynchronous Island scheme does not. The Island schemes are loosely coupled.

Fine-grained PGA is another popular parallel variation of GA, also called the Cellular scheme. With the Cellular scheme, each individual is executed by a computing device separately. When crossover and mutation occur, individuals only mate with their connected neighbors. The form of how individuals connected is called connection topology. It is obvious that the way to spread the result of each individual is decided by the structure of topology, so that the connection topology is a key factor that affect the effect of the Cellular scheme.

Hierarchy PGA is actually the mixture of the schemes above. Generally, in a Hierarchy PGA, a GA population is divided into several parts. Each part uses a particular scheme, which is called the lower layer; and each part acts as an individual in the top layer which follows another scheme. Fig. 1 shows an example of a Hierarchy PGA which combines the Island scheme (the lower layer) and Cellular scheme (the top layer). A Hierarchy PGA is actually a trade-off or compromise of different schemes, and is very flexible, so that Hierarchy PGA is designed to fit different computing environments.

2.3. Discussion and analysis of different PGAs

GAs are used to get approximate solutions for engineering problems within an acceptable time, so there are two aspect to evaluate GA performance – solution quality and execution time.

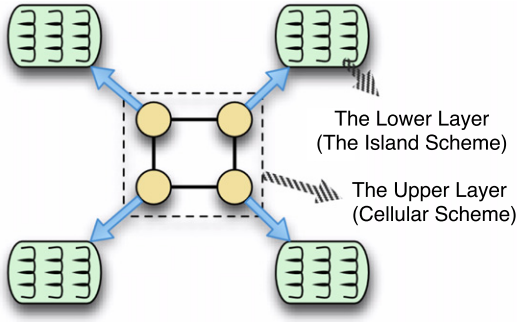


Fig. 1. The Hierarchy PGA.

Different PGAs are more or less inspired by biological theory, so much theoretical analysis of GAs is based on biological theory not on more precise mathematics. Generally, from biological theory there are two aspects affecting the GA solution quality—isolation and diversity.

It is clear that inbreeding leads to worse and worse generations in nature. Therefore, it is empirically indicated that the Master–Slave scheme can get better solutions than the Island scheme, since the Island scheme isolates the population into several islands and individuals in islands only communicate during migration. It is very slow for the Island scheme to spread the best individual to the whole population.

It is also clear that heterosis, which depends on diversity, can make a better generation. Although isolation indeed exists in the Cellular scheme, where individuals can only mate with their neighbors, there is also diversity because individuals are definitely independent of each other, more than in the Master–Slave or Island schemes. The difference in diversity among Master–Slave, Island, and Cellular schemes is explained in more detail Section 4.2. The solution quality depends on the competition of the effect of isolation and diversity. Generally, we can make a conclusion that the Cellular scheme can get the best solution, followed by the Master–Slave one, and the Island one is the worst.

The computation quantity of PGA schemes varies by the number of individuals and generations. With the same number of individuals and generations, execution time depends on how to map the different PGA scheme to architectures with consideration of memory hierarchy, thread model and communication mechanism and so on. In the following sections, we will analyze different PGAs based on the architecture of multi-core and many-core architectures, make optimization from the architecture view, in order to find out how PGA can provide a best performance in both execution time and solution quality, offering practical advice to help tune PGAs on these architectures.

3. Architecture based analysis of GA on multi-core systems

In order to achieve the peak performance of multi-core systems, users should consider many performance optimization issues. Thus, a multi-core architecture puts constraints on the implementation of different GA schemes, and affects the performance of GAs both in speed and in quality of solutions. In this section, we will first give an essential description of multi-core systems and then analyze different schemes for GAs on multi-core systems, finally we summarize the rules for implementing GAs on multi-core systems.

3.1. Overview of multi-core processors

By placing several cores on a chip, multi-core processors offer a new way to improve the performance of microprocessors. The state-of-the-art multi-core systems usually use multi-socket and multi-core integration. Fig. 2(a) shows the detailed architecture of

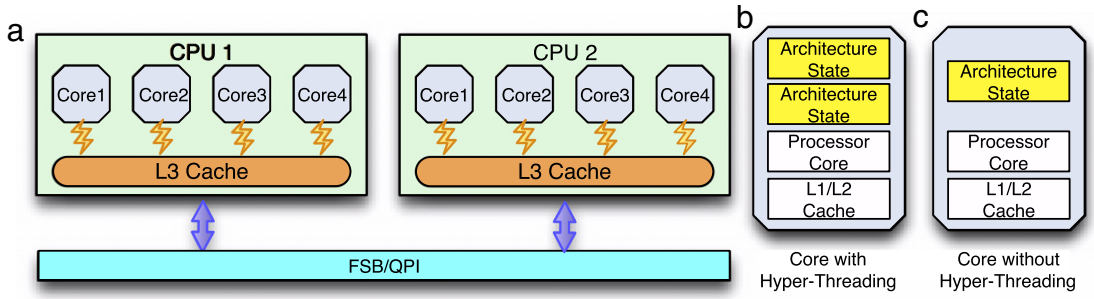


Fig. 2. The multi-core architecture.

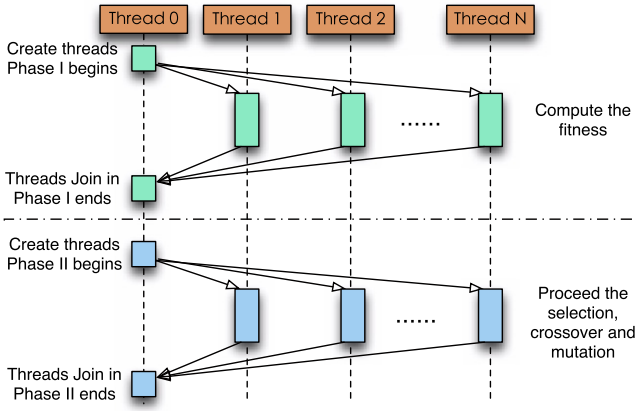


Fig. 3. The Master–Slave scheme on the multi-core architecture.

a 2-socket 8-core system. Each chip has a shared L3 cache, and two chips can communicate via Front Side Bus (FSB) or QuickPath Interconnect (QPI). Multi-threaded programming is needed to take advantage of multi-core processors. In our work, we choose Pthread to implement different schemes of the GA. The performance of a multi-threaded program will be affected by three main factors—the thread model, the cache miss rate and the CPU utilization.

First, threads within an multi-threaded application are often close-coupled; and the communication among these threads occurs quite often, which decrease the parallelism. Thus, the design of a appropriate thread model—such as threads organization, communication and data sharing among different threads, can affect the performance. An appropriate thread model can improve the performance significantly, and the wrong model will decrease the performance dramatically.

Second, since the cache coherency is maintained in multi-core systems, maintaining the cache coherence causes cache evictions, which may lead to excessive cache misses. Excessive cache misses can lead to thread halts, so that the CPU utilization decreases. If a particular variable is shared by threads that are executed on cores within a same CPU, the cache miss occurs in the L1 and L2 cache. However, if the variable is shared by the threads that are executed on cores of different CPUs, the cache miss occurs in L1, L2 and L3 caches of these CPUs. Thus, the overhead of cache coherence among different CPUs is dramatically bigger than within a CPU. Therefore, multi-threaded programs must avoid sharing cache lines among cores, especially among cores of different CPUs.

Third, the CPU utilization is very important for the performance of a multi-threaded program. Intel Hyper-threading (HT) is a powerful way to increase the CPU utilization. Fig. 2(b) and (c) illustrate the details of cores with and without the HT technique. A core consists of three main parts which are Process State, Processor Core and Cache. A Process State tracks the flow of a program or thread. A traditional CPU has only one Process State in a core. A CPU with Intel HT technology duplicates the state that is the same as the Cyclops64 [13]; others such as UltraSPARC [14] and MIT Alewife Machine [15] may support more than two. When the operating system schedules threads, it treats the two separate states as two separate logical cores. When a thread on a logical core stalls, the HT technology allows the thread on the other logical core to be executed immediately. Thus, the HT technology helps to keep the CPU busy and increase the utilization of cores efficiently.

In order to implement a well-optimized GA, all aspects of both multi-core architecture and multi-threaded software above should be carefully considered, so that the GA can get the best solution quality within the shortest execution time. It is noticed that in the analysis, we assume that L1/L2 caches are private for each core and L3 caches are shared across the cores on the same chip; and the HT technique is supported.

3.2. Analysis of GA schemes on multi-core architecture

A serial GA can run on a multi-core architecture without any modification. However, since a serial GA only has one thread, such that only one core can be utilized, there is no difference between serial GAs on single-core and multi-core processors.

The traditional Master–Slave scheme, which uses a centralized organization, needs to choose a node as the master node, and others as the slaves. Therefore, in a straightforward implementation, we should use a thread as the master thread, and create a number of threads as slave threads. However, a multi-core CPU is symmetric; each core is equal to any other. We should keep the load balance among cores, which means we need to decentralize the Master–Slave scheme on a multi-core architecture.

In a multi-core architecture, all threads are symmetric. We divide these threads into two phases. The threads in Phase I act as slaves, and the threads in Phase II act as the master. The detailed design and implementation of the Master–Slave scheme on a multi-core architecture is illustrated in Fig. 3. In the figure, Thread 0 is the father thread, which is a control thread, so that it creates other threads and waits for other threads to join in. In Phase I, Thread 0 first creates threads 1 to N, to compute the fitness. Then Threads 1 to N join in to Thread 0. After Phase I completes, Phase II begins with Thread 0 creating N threads. They process the selection, crossover and mutation, to produce the next generation. Then Threads 1 to N join in to Thread 0. Phases I and II are looped to make individuals evolve. During both Phase I and II, except for Thread 0, each thread evolves an equal number of individuals. With the help of the operating system, threads are scheduled to cores to be executed.

The performance of the Master–Slave scheme on a multi-core architecture is affected by the synchronization operations and the number of threads. The *join-in* operation in Pthread can actually be considered as a synchronization operation. Hence, there are two synchronization operations in a generation. The GA usually needs thousands of generations to evolve a good solution so there are

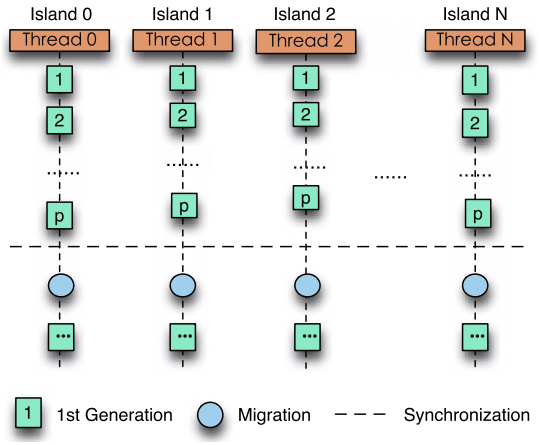


Fig. 4. The synchronous island scheme on a multi-core architecture.

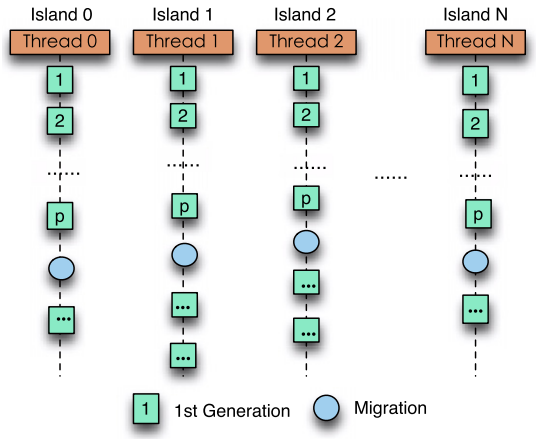


Fig. 5. The Asynchronous Island scheme on a multi-core architecture.

a huge number of synchronization operations. When we create the threads from Thread 0, we must consider the influence of synchronization operations. The following rule should be followed:

$$N\%c \equiv 0 \tag{1}$$

where N is number of threads that Thread 0 creates, and c is the number of cores. If N is not a multiple of c, performance decreases. For example, if Thread 0 creates five threads on a 4-core processor, four threads execute each time. There are two synchronization operations during a generation, therefore the first four threads that achieve the synchronization operation have to wait for the last thread to reach it, which leads to wasting three cores. It is obvious that the time that a 4-core processor takes to execute 5, 6, 7, or 8 threads should be almost the same. The execution time T can be expressed as

$$T = ((f_1^{\max} + f_2^{\max}) \cdot N/c + 2 \cdot S + mod(N, c) \cdot r) \cdot G \tag{2}$$

where $f_1^{\max}$ and $f_2^{\max}$ are the maximum sum of the execution times of Phases I and II of all threads, S is the time that thread creation and join-in need, r is the time slice of the operating system, G is the number of generations the population evolves, and the function $mod(N, c)$ returns 0 if N and c follow Eq. (1), or 1 otherwise.

If the size of the population stays unchanged, which means $(f_1^{\max} + f_2^{\max})$ is a constant value. When $N\%c$ is not equal to 0, then when there are 5, 6, 7 or 8 threads in the example above, extra execution time r is needed. We call it *thread align* that the numbers of threads and cores follow Eq. (1). In practice, $f_1^{\max}$ and $f_2^{\max}$ are very small, because the computation in Phases I and II is small. Therefore the $mod(N, c) \cdot r$ part can affect the performance, which requires us to keep threads aligned to avoid wasting multi-core computation capacity.

The Island scheme is naturally suitable for the multi-core architecture. An island can be implemented in a thread and the migration operation is very efficient and easy in memory with the help of shared caches. Although the communication between threads is expensive, the communication caused by the migration is not frequent in the Island scheme, since the migration occurs every fixed number of generations.

It is straightforward to use one thread for the evolution of each island. Figs. 4 and 5 illustrate the Synchronous and Asynchronous Island schemes respectively. In the figures, we set that the migration occurs every p generations.

When the population using a Synchronous Island scheme evolves G generations on a processor with c cores and the migration occurs every p generations, the execution time T can be expressed as follows:

$$T = \sum_{i=1}^{G} (d_i^{\max} \cdot N/c) + (m + mod(N, c) \cdot r) \cdot G/p \tag{3}$$

where $d_i^{\max}$ is the maximum execution time of all islands to evolve to the i-th generation, and m is the overhead of putting individuals in and getting individuals from the Migration Data structure. However, the $\sum_{i=1}^{G}(d_i^{\max} \cdot N/c)$ part is far greater than the $(mod(N, c) \cdot r) \cdot G/p$ part, since p is usually a big number to decrease the frequency of migrations. Hence, the $(mod(N, c) \cdot r)$ can be ignored, and Eq. (3) can be rewritten as

$$T = \sum_{i=1}^{G} (d_i^{\max} \cdot N/c) + m \cdot G/p. \tag{4}$$

With Eq. (4), we find that we do not need to keep threads aligned when the Synchronous Island scheme is used.

In the Asynchronous Island scheme, we do not have to synchronize the migration operation. This implies that the faster threads do not have to wait for the slower threads for migration, which is illustrated in Fig. 5. When a thread that executes an island puts and gets individuals, it just needs to lock the Migration Data structure to avoid a read/write conflict. As for the Synchronous Island scheme, when a population using the Asynchronous Island scheme evolves G generations on a processor with c cores and the migration occurs every p generations, the execution time T can be calculated as

$$T = \sum_{i=1}^{G} (d_i^{avg} \cdot N/c) + m \cdot G/p \tag{5}$$

where d_i^{avg} is the average execution time of all islands to evolve to the i-th generation. It is obvious that $d_i^{\max}$ in Eq. (4) is bigger than d_i^{avg} in Eq. (5), so that with the same size of population and number of generations, the Asynchronous Island scheme should be faster than the Synchronous Island one.

3.3. Optimization of GAs for multi-socket multi-core architecture

Multi-socket multi-core processors are usually considered the scaling of single-chip multi-core processors. Since an operating system also naturally supports the multi-socket multi-core processor, users' multi-threaded programs can be executed on it without modifications. Typically, a multi-socket multi-core processor can consist of up to 64 logical cores with HT support. Hence, during this section we will focus on how to design an appropriate PGA that has a good scalability. Besides, we also discuss the architecture-based optimization featured in the trade-off between cache coherence overhead and load balancing.

It is definitely true that we can directly use Master–Slave or Island schemes on multi-socket multi-core architectures without any modification. However, threads in a Master–Slave scheme

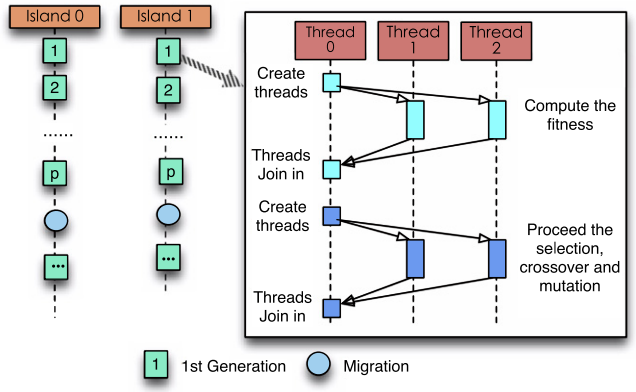


Fig. 6. The Island/Master–Slave Hierarchy scheme on the multi-socket multi-core architecture.

share the whole GA population. Threads are close-coupled, so that sharing of cache lines among cores cannot be avoided at all. When the Master–Slave scheme is applied to multi-core processors, the overhead of cache evictions can be ignored, since the shared L3 cache can maintain the coherency. However, things in a multi-socket multi-core architecture are totally different. The overhead of maintaining cache coherency among chips is far larger than within a chip, so we cannot ignore it any more. As the number of cores increases, i.e., the number of chips increases, the overhead of cache coherence among chips increases. As the number of cores increases in a multi-socket multi-core architecture, the Master–Slave scheme degrades the GA performance in execution time.

On the other hand, threads in Island schemes are isolated from each other, except for the migration that occurs every fixed generation, so that the problem of cache line sharing only may occur in the migration, which can be ignored. However, as there are many cores in a multi-socket multi-core architecture, more threads are needed to take advantage of the parallelism. More threads in Island schemes mean more islands to divide the GA population into. With the same population size, the solution quality decreases as the number of islands increases. Therefore the Island schemes degrade the GA performance in solution quality when the multi-socket multi-core architecture is scaled up.

In order to avoid the overhead caused by cache line sharing among cores and the degradation of solution quality caused by excessive islands, we should try to use a Hierarchy PGA to combine Master–Slave and Island schemes, to achieve good performance both in execution speed and solution quality.

A Hierarchy PGA on a multi-socket multi-core architecture is illustrated by Fig. 6. The upper layer uses an Island scheme and the lower layer uses the Master–Slave scheme. Threads on cores of a chip follow the Master–Slave scheme that is further coupled following the Island scheme. With this Hierarchy PGA, fault sharing only occurs within a chip, which eliminates the overhead of cache coherence among chips, so that the execution speed is faster than a pure Master–Slave scheme on this architecture. Besides, both the Master–Slave scheme in the lower layer and the smaller number of islands, equal to the number of chips, improve the solution quality, which is better than the one of pure Island schemes on this architecture. As a result, the Hierarchy PGA does not have much cache coherence overhead and its solution quality does not degrade when the number of sockets is increased.

Generally operating systems are not aware of the relationship of threads. Some operating systems – for example, CentOS – even initially put consecutive threads to different chips equally to balance the load between chips. If threads are close-coupled, this policy achieves load balancing at the cost of cross-chip cache line sharing. To implement a Hierarchy PGA on multi-socket multi-core architectures, a thread affiliation mechanism is needed. With this mechanism, we can manually arrange particular threads on particular cores or chips. Although the appropriate manual alteration of thread affiliation can avoid or reduce the overhead of false sharing and improve the reuse of cache and the cache hit ratio, it also leads to a load balance problem. There are two extreme options. One is loose affiliation, with which threads are not affiliated with any particular cores, so that the operating system can schedule them at runtime to achieve the best load balance. The best load balance can keep all cores as busy as possible, hence the execution speed can get benefit, however, the fault sharing cannot be avoided at all which slows the execution speed at the same time. The other one is restricted affiliation, with which each thread is affiliated to a particular core to avoid the false sharing and improve the reuse of cache. For example, with HT support, the logical cores that are emulated by the same physical core have the highest priority to be assigned the threads sharing data, then the cores within a chip, and the cores across chips are the last option. The logical cores that are actually on the same physical core do not lead to any false sharing overhead and can reuse the data they share in the cache since they use the same cache hierarchy. Shared caches of cores within a chip reduce the overhead of false sharing and also can reuse some shared data. Threads on different chips can do nothing but cache miss when false sharing occurs or threads share data. A well-designed restricted affiliation can reduce the cache coherence overhead as much as possible, however there is no room left for the operating system to schedule threads at runtime to balance the workload leading to performance loss. Therefore, a trade-off between the extreme options above should be made.

Considering that false sharing within a chip is not so critical, we try to avoid false sharing across chips. A moderate affiliation should be appropriate, with which threads of the Master–Slave scheme in the lower layer will be affiliated with a particular chip not a particular core. Since multi-core processors have at least four logical cores with HT support, the moderate affiliation is able to avoid the overhead caused by false sharing across chips, reuse the data residing in shared caches and offer enough room for the operating system to migrate threads to cores within a chip at runtime to balance the workload.

Summarizing, with the features of multi-socket multi-core architecture, especially the scalability, the Island/Master–Slave Hierarchy PGA can deal with the execution speed and solution quality well when the number of sockets increases. In the implementation, threads in the Master–Slave scheme should be manually affiliated to a particular chip but not a particular core to achieve the fastest execution time.

4. Architecture based analysis of GA on many-core systems

In this section, we first introduce the GPU architecture, and then analyze the choice of different PGA schemes to map the GPU architecture from perspectives of both execution time and solution quality. Finally, we offer the design principle of the PGA with deep architecture-based optimizations.

4.1. Analysis of many-core architecture

The Graphics Processor (GPU) is considered as an accelerated co-processor in the computer system, which implies that the GPU has to be controlled by a CPU. In recent years, GPUs are widely used for general-purpose computation with the help of flexible programming frameworks, e.g. CUDA, OpenCL. With these programming frameworks, programmers write two parts of codes, the kernel code and the host code. The host code runs on the CPU to control data transfers between the CPU and GPU, and start the execution on the GPU. The kernel code is executed by massive

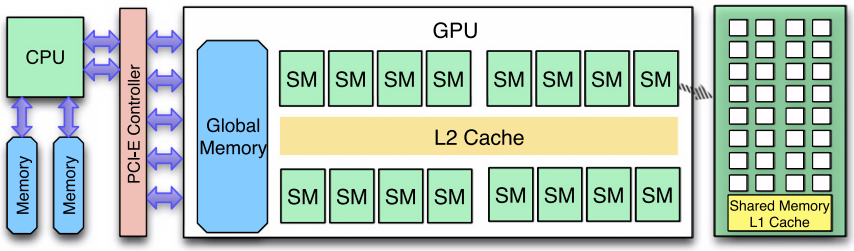


Fig. 7. The NVIDIA GPU architecture with a CPU.

threads on the GPU. At high level, a GPU comprises several SIMD multiprocessors. Processors within a multiprocessors execute the same instruction, but operate on different data, and share computation resources. The GPU has a large off-chip device memory with high bandwidth as well as high latency. This architecture is a common design for both NVIDIA and AMD GPUs. For efficiency, we use NVIDIA GPU architecture and CUDA for our analysis and implementation.

The architecture model of a NVIDIA GPU is illustrated in Fig. 7. Each Streaming Multiprocessors (SMs) has 32 Streaming Processors (SPs) that are also called GPU cores in the latest NVIDIA Fermi architecture. Massive threads are executed on cores, and 32 threads are organized into a warp, in which the exact same instructions are issued to these threads. Warps are further organized into a block that is the basic unit mapping to an SM. Several blocks composed the grid which is the top unit of threads in GPUs. Compared to the CPU architecture, the GPUs have a complicated memory hierarchy in order to leverage the power of huge numbers of cores. The memory hierarchy stands as a key point for the performance. Programmers usually use registers, constant memory, shared memory, L1 cache, L2 cache and off-chip global memory for general purpose computation. Registers are divided equally and statically among all threads during the kernel initialization phase. Constant memory is read-only and has uniform access behavior for all SMs. L1 cache and shared memory in each SM are shared by threads running on the same SM. Moreover, the total capacity of L1 cache and shared memory is always 64 KB for each SM. Programmers can statically initialize them as 16 KB vs. 48 KB or 48 KB vs. 16 KB. L2 cache and Global memory is dynamically shared by all threads at runtime. Data transfer between Global memory and main memory is controlled by the CPU via the PCI-E channel. Access to registers is the fastest at only one cycle, followed by constant memory, L1 cache and shared memory consume 1–2 cycles if bank conflict does not happen. The slowest is the global memory, consuming hundreds of cycles.

The interaction between execution units and memory hierarchy is more complicated than above. We focus on the following three aspects—registers, shared memory and interaction between main memory and global memory.

As register files are divided statically among all threads, a thread context switch causes no overhead except when the total registers all threads need exceed the size of the register file. On the other hand, GPUs need massive simultaneous threads running a SM, so that if a warp of threads accesses global memory, threads can switch out and another warp switch in to overlap the memory access. To make sure a GPU has enough simultaneous threads, the number of registers all threads need must be less than the size of the register file. Otherwise some threads must access the global memory to recover their registers during a context switch, leading to hundreds of cycles of overhead. We call threads that do not need to recover their context flying threads, otherwise non-flying threads. However the Fermi architecture introduces L1 cache, mostly the non-flying threads only need to access L1 cache to recover their contexts. The difference between flying and non-flying threads blurs in the new Fermi architecture, so that the number of registers all threads need is not as critical as in the previous GPU architectures.

Shared memory is actually critical in the GPU architecture. The shared memory of each SM is divided into blocks that run on this SM. Hence, how much shared memory the blocks require decides how many blocks can be executed on a SM concurrently. Besides, input data are transferred from main memory to global memory, before starting the kernel execution. After the kernel completes, results are transferred from global memory back to main memory. Data transfer between main memory and global memory is expensive, especially for frequent small data transfers.

Summarizing, since the complexity of many-core architecture is exposed to end-users in the current programming framework, three challenges should be overcome to fully use the power of many-core architecture—the deep understanding of architecture features, the appropriate parallel model and the fine-grained optimization to avoid performance degradation. In the following sections, we focus on how we make GAs overcome the latter two challenges.

4.2. Analysis of GA implementations on many-core system

Based on the analysis of many-core architecture, we analyze different PGA schemes and the best choice on many-core systems, focusing on the performance both in execution time and solution quality. Since some issues of GAs on many-core architectures are similar to those on multi-core architectures, we concentrate the unique features of GAs on many-core systems.

The Master–Slave scheme. As a GPU is designed as a co-processor, it is natural to implement the Master–Slave scheme. More specifically, the CPU does the selection, crossover and mutation as masters; and the GPU calculates the fitness as slaves. However, it is very inefficient for two main reasons. First, it needs to transfer the population data between main memory and global memory every generation, resulting in a large communication overhead. Second, the CPU will probably be the bottleneck of the whole system, since the GPU computation capacity is much more powerful than that of the CPU, and the workload of GA selection, crossover and mutation is even heavier than that of fitness calculation in some cases. Therefore, it is not a good choice to adopt the Master–Slave scheme on the GPU. The execution speed will be very slow.

Island scheme. According to the analysis in Section 4.1, threads in a block can use the shared memory to communicate with little overhead; threads in different blocks can communicate but at high cost. Hence, the GPU should use the Island scheme, since individuals in an island communicate a lot and the communication between islands is rare. With the Island scheme on a GPU, each thread executes an individual, and each block executes an island, which is illustrated in Fig. 8. It is obvious that the solution quality of the Island scheme on GPUs is exactly the same as on multi-core CPUs, except that the execution speed should be faster.

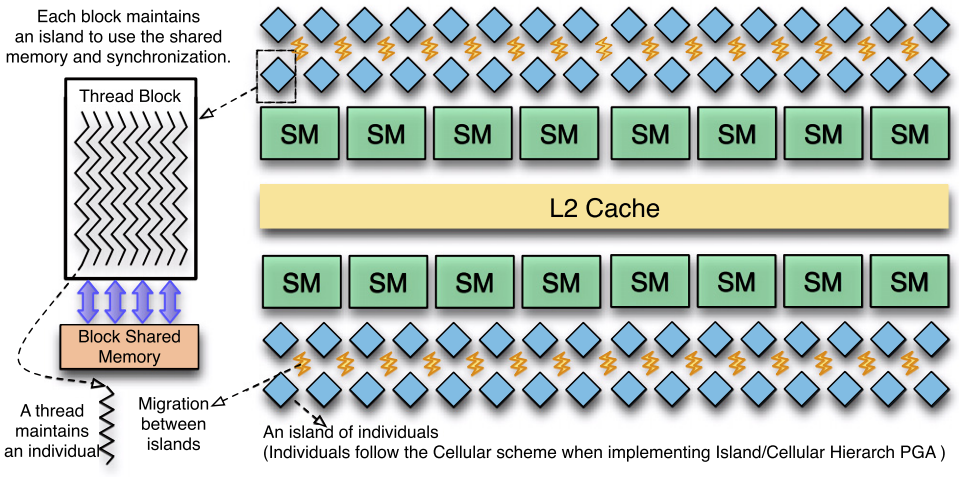


Fig. 8. The Island and Island/Cellular PGA schemes on many-core architectures.

The communication between individuals in an island can use the synchronization operation provided by CUDA. However, the synchronization operation in the Synchronous Island scheme must be implemented by programmers using the global memory, because there is no synchronization operation among blocks in a GPU. Efficient synchronization among blocks is discussed in detail in [16]. The Asynchronous Island scheme should be faster than the Synchronous Island scheme on the many-core architecture. There are two major reasons. First, faster blocks do not need to wait for slower blocks in an Asynchronous Island scheme, so that faster blocks can continue to evolve without any interruptions. Second, the Synchronous Island scheme needs to do synchronization among all blocks leading to extra accesses to the global memory.

Hierarchy scheme. The Cellular scheme can be applied to many-core architectures easily. Each thread acts as an individual, and communicates with its neighbors every generation to evolve the solution. The communication overhead of threads within a block is low with the help of in-block shared memory. However, threads across blocks can only communicate via global memory, which results in significant performance degradation. In order to reduce the communication overhead, we can construct an Island/Cellular Hierarchy PGA. With this model, threads within a block follow the Cellular scheme, and all blocks are composed with the Island scheme (see Fig. 8). The communication overhead across blocks is reduced dramatically because the Island scheme only needs to do communication during the migrations that occurs usually every thousand of generations, i.e., the communication overhead across blocks decreases to less than 0.1% of overhead of the pure Cellular scheme.

Individuals in a typical Cellular scheme communicate with their neighbors with a particular topology in order to keep the communication overhead low. Only communicating with particular neighbors leads to the isolation problem that further degrades the solution quality. Thanks to shared memory, the communications overhead of individuals within a block is always the same, no matter how many neighbors an individual has. Thus, we apply the fully-connected mesh topology in the Cellular scheme on a many-core architecture, which is illustrated in Fig. 9. The fully-connected mesh topology improves the solution quality but does not cause any extra overhead. The proposed Island/Cellular Hierarchy PGA is very similar to the pure Island scheme, since the fully-connected mesh topology is used. However the Island/Cellular Hierarchy PGA outperforms the Master–Slave or Island schemes in scalability and solution quality. The main reason is that they have different choices on a random number generation sequence.

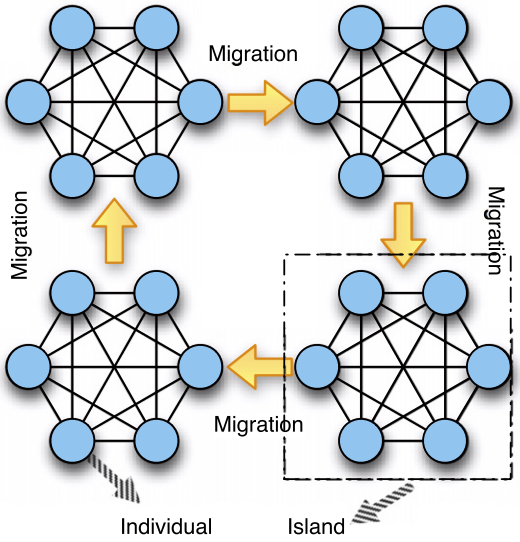


Fig. 9. Details of upper and lower layer of Island/Cellular Hierarchy PGA on many-core architectures.

In the Master–Slave scheme, all operations needing random numbers are in the master part, which implies that the random numbers are all generated by the CPU, the GPU only computes the fitness. Thread safe random generators are provided by many programming language with APIs. Although most thread safe random generators are lock-free, the sequential correctness is guaranteed by primitive operations. When the number of concurrent threads accessing a random generator is large, threads also are halted by the processor to avoid Read After Write or Write After Read data conflicts. However threads on a multi-core architecture are rarely halted to avoid the data conflicts, since there are not so many threads to access the random generator at exactly the same time.

When the Island scheme is implemented on a many-core system, each thread represents an individual that needs random numbers to do crossover and mutation. The random numbers that all threads need should be in the same random sequence. A similar lock-free thread safe random generator will lead to extra overhead, since there are thousands of threads that access the generator. To avoid the data conflict actually forces a lock-free thread safe random generator to be serialized. Besides, because all threads need to share the random sequence, the data that the random generator needs must reside in the global memory. Each access to the random generator by each thread means a

long latency access to global memory. Some previous work just generates random sequences for each island in advance, and puts them in the global memory to follow the rule, which is called static random generation. However, as we mentioned time and time again, access to global memory needs a long latency. The number of random numbers needed in the evolution is large, since each individual needs at least ten random numbers every generation. For 4096 individuals and 50,000 generations, it needs about 16 GB memory capacity. The largest global memory capacity of GPUs now is 6 GB. Even if the global memory capacity of GPUs were large enough, it is a performance disaster for GPUs to read large data from global memory; this can ruin the GPU performance entirely. Static random generation cannot actually be used in practice. In a word, the random generator that the Island scheme needs hardly fits the many-core architecture and the frequency of access to the random generator is very expensive, which obviously degrades performance.

In the Island/Cellular Hierarchy PGA, the Cellular scheme is applied in each block. The random generator requested by the Cellular scheme is perfect for features of many-core architectures. Since with the Cellular scheme, all individuals are independent, each thread maintains its own random sequence with different seeds generated by the main random number generator. All random generators of threads are independent, so threads do not need to wait for each other even if the number of threads is huge and the frequency of access to the random generator is very high. Besides, since the data that random generator needs is private for each thread, it can reside in the shared memory. When the fully-connected mesh topology is used in an Island/Cellular Hierarchy PGA, the computation is actually almost the same as in the Island scheme. However, there is no thread competition caused by the random generator and the random generator uses shared memory rather than global memory. The execution speed of an Island/Cellular Hierarchy PGA exceeds that of the Island scheme and of course the Master–Slave scheme. Regarding solution quality, because with the fully-connected mesh topology all individuals in the Cellular scheme can do selection, crossover and mutation together rather than only with neighbors, we decrease the isolation effect of the Island/Cellular Hierarchy PGA, which is exactly the same as the Island scheme. Moreover, because each thread uses an independent random sequence, the diversity is much better than in either the Master–Slave or Island scheme. The Island/Cellular combines the advantages of both Island and Cellular schemes, hence it can provide a more precise solution than the Master–Slave or Island scheme. In a word, on the many-core architecture, the Island/Cellular Hierarchy PGA fits the architecture best and can offer the best performance both in execution speed and solution quality.

4.3. Architecture-based detailed design of GAs

After analyzing different PGA models on the many-core architecture and proposing the most appropriate one, the architecture-based optimization should be followed. As the many-core architecture is very complicated and all exposed to users, only careful optimization can fully use the power of GPUs. In this section, we first conclude the general GPU optimization fundamentally related to GAs, then we offer the detailed optimization rules for the Island/Cellular Hierarchy PGA, which is effective for all cases.

Since GPUs have a complicated memory hierarchy and huge numbers of cores, the optimization focuses on the appropriate use of different types of memory and how to increase the efficiency of cores. Based on the description in Section 4.1, the optimization fundamentals related to GAs are as follows.

1. The shared memory of a SM should at least meet the requirements of one block, so that the kernel can be compiled and executed.
2. There should be at least 64 threads in a block, which is the size of a warp, so that all cores in a SM can be fully utilized.
3. The number of blocks should be at least equal to the number of SMs in a GPU, so that all SMs in a GPU can be fully utilized.
4. If there are many accesses to global memory, the total number of threads should be as large as possible with constraint of (1), to overlap accesses to global memory and therefore reduce latency.
5. The communication between threads in a block should be implemented in shared memory. Otherwise, this uses global memory by default, which leads to a big access latency.
6. The communication between blocks should be minimized, since it must use the global memory.
7. The data transfer between CPU and GPU should be minimized during kernel execution, as moving data from main memory to GPU global memory is always the bottleneck of GPU computing.

In order to harness the computation power of GPUs, the design of GAs, especially the thread-block configuration and efficient use of memory hierarchy are decided by the GPU's resource constraints. A good optimization on GPUs is from design to final implementation, hence we offer a quantitative analysis to construct a GA on a many-core architecture, focusing on the Island/Cellular Hierarchy PGA; that is the best PGA on the many-core architecture as we showed in the last section.

To construct a GA on a many-core architecture, we begin with the analysis of the essential variables of the GA implementation. The location of variables is based on how many threads access the variable and how often, so that the GA can get the most efficient access to the memory hierarchy. Table 1 shows the variable names, and their location.

Some GA problems have several constraint conditions, each of which could be stored in the *Eq_Constraint*. The *Configuration* stores information on the size of the population, the number of generations that GA evolves, how to organize the Island/Cellular Hierarchy and so on. Since the constraint conditions and configuration are unchanged during the evolution, they can be stored in the Constant Memory. Each thread stores *Best_Fitness* and *Best_Penalty*, that is values of fitness and penalty of the best individual in its island. *Random_Seed* is used for each thread to generate its own random numbers. During the selection, crossover and mutation, some exchange occurs among individuals, hence with an *Ex_Individual*, a descriptor of an individual, threads can accomplish the exchange efficiently. The *Assist_Var* represent all other general variables that a program needs such as iterator, temporary data and index. These variables are accessed several times in each generation, so that it is very efficient to use registers to store them. The *Block_Individual* is an array of which each element is an individual descriptor, so that the *Block_Individual* can record all individuals of an island. The individual description is actually a structure in which there is an array to record the value of each parameter of the GA problem, values of fitness and penalty and several statistics data. The *Best_Individual* is an individual descriptor that records the best individual in the island. *Block_Individual* and *Best_Individual* are shared by all threads in an island and each thread needs to access them once every generation, so they should reside in the Shared Memory. The *All_Individual* is similar to the *Block_Individual*, except that *All_Individual* records all individuals of the population. It is used for the migration. It is shared by all threads and only accessed when migration occurs, so that it is very suitable to be in the global memory. The *True_Random* is used to initialize *Random_Seed* of each thread. It is generated by the CPU because a GPU cannot offer true random numbers. It is transferred from main memory before the GPU kernel is launched. The *Statistic_Data* records the solution to the GA problem and other static information users needs at the end of the GPU kernel, and is transferred back to main memory when the GPU kernel ends. *All_Individual*, *True_Random*

Table 1
Variables and their locations.

Location	Variable name	Description
Constant memory	Eq_Constraint	The constraint condition for GA problem.
	Configuration	The configuration data for the GA.
Register files	Best_Fitness	The fitness value of the best individual of its island.
	Best_Penalty	The Penalty value of the best individual of its island.
	Random_Seed	Random Generator related data.
	Ex_Individual	Supporting to exchange two individuals during the selection operation
	Assist_Var	The variables that general programs needs.
Shared memory	Block_Individual	The array of descriptors of individuals in an island.
	Best_Individual	The descriptor of the best individuals in an island.
Global memory	All_Individual	The array of descriptors of individuals of the population.
	Statistic_Data	The statistic data that users need.
	True_Random	The true random numbers generated by CPU.

and *Statistic_Data* are all shared by all threads and accessed rarely, so that it is quite appropriate for them to reside in the Global Memory.

The shared memory capacity is the strongest resource constraint in the Fermi architecture, so we begin with the design of how many threads should be in a block with the constraint of shared memory. Each thread represents an individual and each block represents an island as designed in Section 4.2. In the following we will not distinguish thread from individual or block from island.

Several blocks can be assigned to an SM, only if the shared memory they need does not exceed the capacity of shared memory, which is expressed as

$$\left(S_x \cdot n + S_f + S_p + \sum_{i=1}^{\theta} S_s^i\right) \cdot (T_B + 1) \cdot \rho \le M_s \tag{6}$$

where $(S_x \cdot n + S_f + S_p + \sum_{i=1}^{\theta} S_s^i)$ calculates the size of a individual descriptor. More specifically, S_x is the size of each parameter of the GA problem, n is the number of parameters, S_f and S_p are the sizes of the variables that record fitness and penalty, S_s^i is the size of the i-th static data item in the individual descriptor and θ is the number of static data. Each block needs a *Block_Individual* that consists of T_B individuals, and a *Best_Individual*, hence there are $(T_B + 1)$ individual descriptors. ρ indicates how many blocks are assigned to an SM. M_s is the capacity of shared memory of an SM.

With (6), we can calculate the number of threads a block can handle, T_B. However, 32 threads organized into a warp execute concurrently. Hence, we must make T_B a multiple of 32, otherwise the last warp cannot fully use cores in a SM, which degrades the performance. Besides, because the SM issues instructions to 64 threads at once, there are at least 64 threads on an SM. Finally, T_B can be expressed as

$$\begin{cases} T_B = \eta \cdot 32, \quad (\eta = 1, 2, \ldots, k) \\ k = \lfloor M_s / (S \cdot \rho - 1) / 32 \rfloor \\ S = S_x \cdot n + S_f + S_p + \sum_{i=1}^{\theta} S_s^i. \end{cases} \tag{7}$$

Obviously, the number of the blocks on the GPU, N_B is

$$N_B = \rho \cdot N_{SM}. \tag{8}$$

The number of individuals of the population, i.e., the number of threads on the GPU, N_{pop} can be calculated as

$$N_{Pop} = \rho \cdot N_{SM} \cdot T_B. \tag{9}$$

With T_B, N_B and $N_P op$, and the variables in Table 1, we can construct a GA on the GPU with preliminary architecture-based optimization. However, how to choose ρ and η is very important for the performance, and there are further optimization issues.

Firstly, we must choose the appropriate ρ to make sure $k \ge 1$. $k = 0$ means the size of shared memory blocks exceeds the capacity of shared memory of an SM, which leads to the compiler failure. When $k = 0$, even with $\rho = 1$, we must try to move *Best_Individual*, *Block_Individual* or even both to the global memory to shrink the S to make the compiler successful.

Then, we must make sure that the sizes of variables in Constant Memory and Global Memory do not exceed the corresponding capacity, which can be expressed as

$$\sum_{i=0}^{\lambda} C_{eq}^i + C_{con} \le M_c \tag{10}$$

and

$$\begin{cases} G_A + \sum_{i=1}^{\gamma} G_s^i + G_R \le M_G \\ G_A = S \cdot \rho \cdot N_{SM} \cdot T_B \\ G_R = g_{seed} \cdot \rho \cdot N_{SM} \cdot T_B \end{cases} \tag{11}$$

where C_{eq}^i is the size of *Eq_Constraint* for the i-th constraint condition of the GA problem, C_{con} are the sizes of *Configuration*, λ is the number of constraint conditions of the GA problem, and M_c is the capacity of the Constant Memory of the GPU; G_A is the size of *All_Individual*, G_R is the size of *True_Random*, G_s^i is the size of the i-th *Statistic_Data*, γ is the number of *Statistic_Data* in the GA, M_G is the capacity of the Global Memory of the GPU, and g_{seed} is the size of a random seed for each thread.

Eq. (11) can be shortened to

$$(S + g_{seed}) \cdot \rho \cdot N_{SM} \cdot T_B + \sum_{i=1}^{\gamma} G_s^i \le M_G. \tag{12}$$

If (10) is not tenable, which implies too many constraint conditions in the GA problem, we can do nothing but put all *Eq_Constraint* in global memory. And we must adjust the value T_B that is further decided by η to make (12) tenable. If either equation is not tenable, the code fails to compile.

So far, we can make our code compile successfully. The GPU needs many threads to overlap the access to the memory hierarchy especially access to the global memory. Therefore, with a particular k in (7), we should set η equal to k. However, besides (12), there are two issues that constrain the value of η.

Register usage is a very important optimization point. In the Fermi architecture, if the registers that all threads require exceed the capacity of register files of the GPU, when a context switch occurs registers are evicted to L1 cache rather than to global memory. Different from the previous CUDA version, the current CUDA has very strong compiler optimization; we can use NVCC to profile the GPU code to know how many registers a thread actually uses, which is denoted as R. R is mostly less than the sum of the

Table 2
The experimental environment.

CPU1	Intel i7-2600 (4 cores)
CPU2	Intel E5650 × 2 (12 cores)
Main memory	6 GB(CPU1)/24 GB(CPU2)
GPU	NVIDIA GTX580
OS	Ubuntu 10.04 Server 64 bit
Kernel version	2.6.38-8
GCC	4.4.3 with -O2 option
CUDA version	SDK 4.0

size of *Best_Fitness*, *Best_Penalty*, *Random_Seed*, *Ex_Individual* and *Assist_Var*. We should adjust η to make sure (13) is tenable:

$$R \cdot \rho \cdot N_{SM} \cdot T_B \leq M_R + M_{L1} \tag{13}$$

where M_R and M_{L1} are the size of Register Files and L1 cache of the GPU, respectively. If (13) is not tenable, it implies that some registers are evicted into global memory, which decreases performance.

When (13) is tenable, we can further consider the dual-issue to improve the performance. In the Fermi architecture, each SM has two independent warp schedulers and two instruction dispatch units, so that each SM can execute two different warps on 32 cores to achieve near peak hardware performance, which is called dual-issue. Only when there are two warps in a block, the GPU can have the opportunity to take advantage of the dual-issue, which can be expressed as

$$\rho \cdot T_B \geq 64. \tag{14}$$

Finally, we can choose appropriate η and ρ to calculate the number of threads in a block T_B, the number of blocks N_B, and the size of population N_{Pop}. With the design of Island/Cellular Hierarchy PGA in Section 4.2, a well-optimized GA can be implemented on the many-core architecture.

5. Performance evaluation

We get several theoretical results in Sections 3 and 4. In this section, we validate our findings and analysis, and evaluate the performance of different PGA schemes on the multi-core and many-core architectures, respectively. We focus on both the execution time and the solution quality. Finally, we present the performance speedup of PGAs on multi-core and many-core architectures from the viewpoint of execution time and solution quality, respectively.

5.1. Experiment setup

Experiments in the previous work on PGAs on many-core GPUs usually only measure the execution time and compare with those on CPUs. The importance of solution quality is always underestimated. In our experiments, we propose the execution time to reach a fixed solution as our main metric to evaluate both the execution speed and solution quality. Meanwhile, when comparing the performance between GPU and CPU in most previous work, the implementation of GAs on the CPU is serial, without any parallelization or architecture-based optimization. In this paper, we choose the best PGA scheme on CPUs as the baseline, so that the speedup claimed for GPUs is more reasonable.

We choose three Nonlinear Programming (NLP) problems from [17] for evaluation, which are widely used to evaluate the performance of different PGA schemes or other optimization algorithms. The NLP techniques are widely used in industrial management, engineering design, and scientific computation [7]. This is different from previous related work which only used one or two mini-problems, i.e., problems with few variables and constraints. The benchmarks we chose are both complicated and real world engineering problems. Moreover, experimental results are quite stable among three different test problems, which further demonstrates our analysis and findings are not relative to a particular problem. The details of the three benchmarks are described in the Appendix.

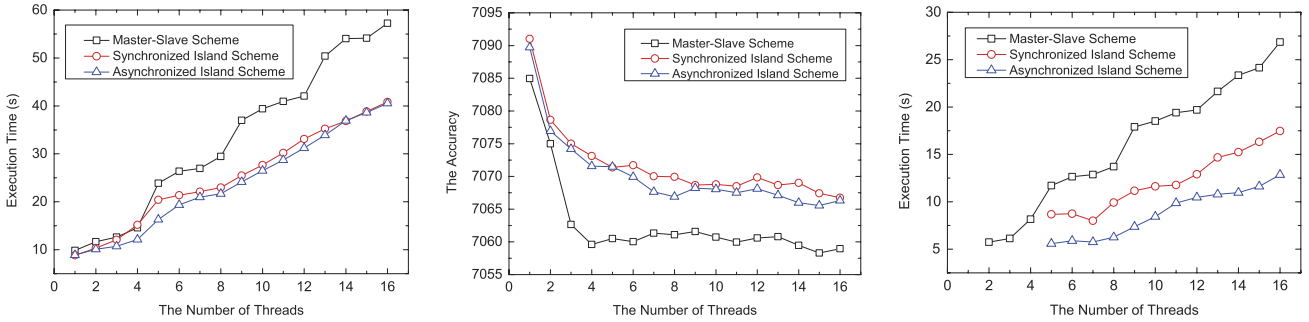
The experimental environment is listed in Table 2. We compare the performance of PGAs on i7-2600 and NVIDIA GTX 580 by following the industry standard that compares one PCI-E slot vs. one CPU socket. And we take two Intel E5650 to construct a 2-socket 6-core system as a target multi-socket multi-core system.

5.2. Evaluation of GA on multi-core systems

We first compare the Master–Slave, Synchronous Island and Asynchronous Island schemes on a multi-core architecture (Intel i7-2600). Due to space limitations, we only report the result for Test Problem 1. The behavior of Test Problems 1 and 2 are very similar. In Fig. 10, each thread evolves 256 individuals. The size of the population varies from 256 to 4096 as the number of threads varies from 1 to 16. Fig. 10(a) shows the execution time of Test Problem 1. We find that the Master–Slave scheme is the slowest, while the Asynchronous Island scheme is the fastest. Also Eqs. (2), (4) and (5) indicate that the Master–Slave scheme needs to keep threads aligned, but Island schemes do not. Because there are 4 cores in this experiment, we can find that the overhead of the Master–Slave scheme increases rapidly when the number of threads goes from 4–5, 8–9 and 12–13. The execution time of Island schemes, Synchronous or Asynchronous, are not affected by the thread alignment.

The solution quality illustrated in Fig. 10(b) shows an opposite result compared to that of the execution time. The Master–Slave scheme is the best followed by the Asynchronous Island scheme, and the Synchronous Island scheme is the worst. Thus, the execution time to reach a fixed solution is used to tell which scheme is the best for both execution speed and solution quality. The result is illustrated in Fig. 10(c). In this figure, we set the fixed solution quality as within 0.3% of the optimum solution. We observe that the Asynchronous Island scheme is much better than either Master–Slave or Synchronous Island schemes. However, when the population size is smaller than 768, Island schemes cannot reach the fixed solution but the Master–Slave one can. In a word, the Asynchronous Island scheme can get the most accurate solution in the shortest time with a large enough population size on the multi-core architecture.

Fig. 11 shows the execution time of Island/Master–Slave Hierarchy PGAs on a multi-socket multi-core architecture with different affiliation policies—No Affiliation, Affiliation to Core and Affiliation to Chip. In this experiment, we use 2-socket 6-core processors, total 12 cores, and 24 logical cores if HT is enabled. There are 48 threads, twice the number of logical cores, on the system so that the parallelism and load balancing can be guaranteed. When No Affiliation is used, all threads are scheduled entirely by the operating system. In Affiliation to Core, we try our best to reduce the overhead of cache coherence, for example, we assign threads sharing data to the same physical core or logical cores emulated by the same physical core as much as we can. When Affiliation to Chip is used, threads within the same island are assigned to the same chip to avoid the false sharing across chips. The three affiliation policies are evaluated with HT on and off. All results are normalized to that of No Affiliation. We observe that the execution time of Affiliation to Core is the worst, which implies that although Affiliation to Core can avoid the overhead of cache coherence, the load balancing of threads impacts the performance more significantly. Affiliation to Chip is the best; it avoids fault sharing across chips and also allows threads migrating among cores. With the Affiliation to Chip policy,



(a) The execution time with different sizes of population (50,000 generations).

(b) The accuracy of solutions with different sizes of population (50,000 generations).

(c) The execution time to reach the fixed solution (7070.5, 0.3% close to the optimum solution).

Fig. 10. Comparison among Master–Slave, Synchronous Island and Asynchronous Island schemes on a multi-core architecture.

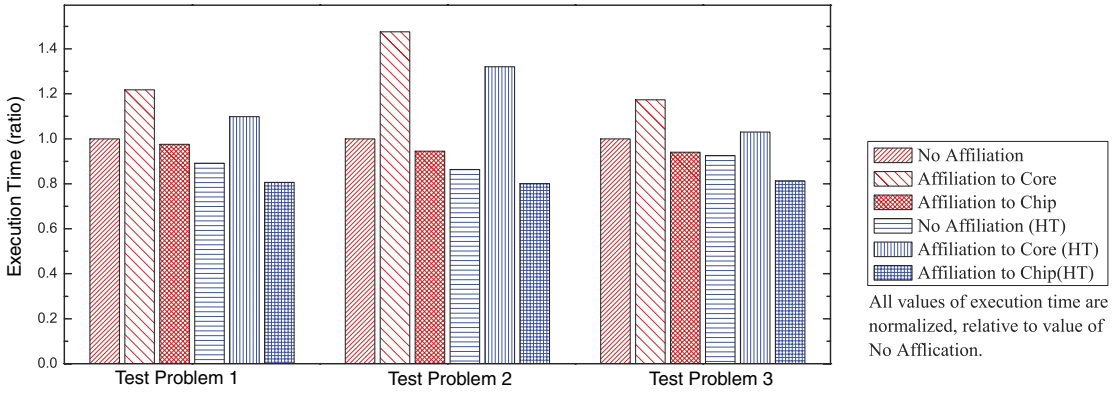
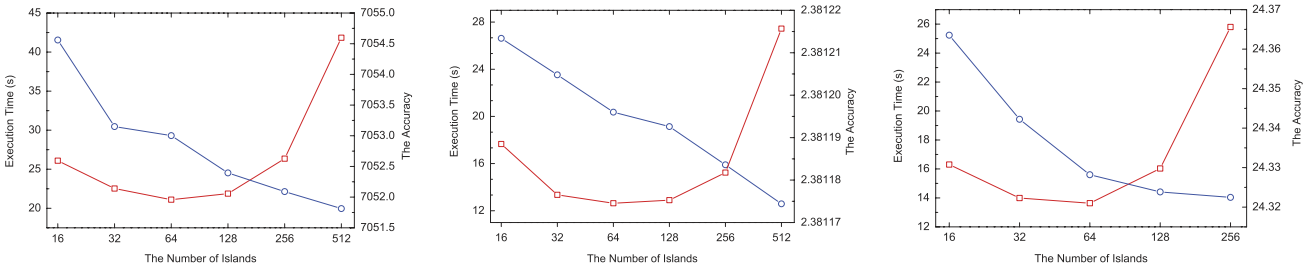


Fig. 11. The execution time of Island/Master–Slave Hierarchy PGAs on a multi-socket multi-core architecture.



(a) Test Problem 1.

(b) Test Problem 2.

(c) Test Problem 3.

Fig. 12. The execution time and solution accuracy with different numbers of islands on a many-core architecture (50,000 generations).

the more cores a chip integrates, the better workload balance can be achieved. Thus, when HT is enabled (i.e., the number of available cores is effectively doubled), the performance improvement of the Affiliation to Chip policy (with HT enabled) increases. The Affiliation to Chip policy can improve the execution speed up to 6% and 14% over those of No Affiliation policy when HT is on and off, respectively. Besides, the HT brings 15% performance improvement on average for all affiliation policies.

5.3. Evaluation of GA on many-core systems

With our analysis in Section 4.2, the behavior of the Island scheme on a many-core architecture is exactly the same as the one on a multi-core architecture. Thus, the Island scheme is ignored. We evaluate the performance of the Master–Slave scheme, the Island/Cellular Hierarchy scheme with synchronous migration (represented by Hierarchy (Sync) Scheme), and the Island/Cellular Hierarchy scheme with asynchronous migration (represented by Hierarchy (Async) Scheme), respectively.

Table 3 shows the execution time in seconds for different sizes of population evolving to 50,000 generations using three different PGA schemes. We keep 16 islands, and increase the number of individuals in each island from 32 to 512 except for Test Problem 3, since it has more variables and constraints so that a single block cannot hold more than 256 threads limited by the shared memory usage. We observe that the Asynchronous Hierarchy scheme is the fastest. We also see that the Asynchronous Hierarchy scheme gives the best solution among these PGA schemes from Table 4. Finally, from Table 5, we find that the execution time to reach a fixed solution of the Asynchronous Hierarchy scheme is the best, which implies it is the best scheme for considering both the solution quality and execution speed. The fixed solution for Test Problems 1, 2, and 3 are 7056.0 (0.1%), 2.38140 (0.1%), and 24.80 (2.0%) respectively. Infinite time means that the program cannot reach the fixed solution within 200,000 generations.

We further study the Hierarchy scheme, and evaluate the impact of the number of islands in Fig. 12. We keep the total population size unchanged, and increase the number of islands (i.e., the

Table 3
The execution time of different PGA schemes on a GPU (50,000 generations).

Test Problem	Size of population	512	1024	2048	4096	8192
Problem 1	Master–Slave scheme	20.038	39.227	77.908	154.248	308.140
	Hierarchy (Sync) scheme	6.862	7.499	9.068	13.951	26.135
	Hierarchy (Async) scheme	6.794	7.440	8.994	13.902	26.091
Problem 2	Master–Slave scheme	14.367	28.138	54.807	108.726	218.579
	Hierarchy (Sync) scheme	4.128	4.615	5.963	9.516	17.708
	Hierarchy (Async) scheme	4.088	4.578	5.920	9.468	17.655
Problem 3	Master–Slave scheme	22.917	44.825	89.037	176.489	–
	Hierarchy (Sync) scheme	8.071	8.773	10.535	16.174	–
	Hierarchy (Async) scheme	8.060	8.760	10.524	16.158	–

Table 4
The solution quality of different PGA schemes on a GPU (50,000 generations).

Test Problem	Size of population	512	1024	2048	4096	8192
Problem 1	Master–Slave scheme	7065.778293	7062.501734	7061.256240	7059.324437	7057.884367
	Hierarchy (Sync) scheme	7055.365514	7054.860704	7054.695477	7054.109487	7053.644277
	Hierarchy (Async) scheme	7054.756105	7054.370935	7054.340533	7053.833326	7053.406822
Problem 2	Master–Slave scheme	2.38125	2.38125	2.38124	2.38123	2.38121
	Hierarchy (Sync) scheme	2.38125	2.38124	2.38123	2.38121	2.38121
	Hierarchy (Async) scheme	2.38123	2.38122	2.38122	2.38121	2.38120
Problem 3	Master–Slave scheme	25.2011692	25.1846270	25.1784330	25.1760074	–
	Hierarchy (Sync) scheme	24.3676946	24.3454162	24.3342626	24.3290792	–
	Hierarchy (Async) scheme	24.3650734	24.3405024	24.3318092	24.3280018	–

Table 5
The execution time to reach a fixed accuracy of different PGA schemes on a GPU.

Test Problem	Size of population	512	1024	2048	4096	8192
Problem 1	Master–Slave scheme	∞	125.490	221.835	239.582	476.919
	Hierarchy (Sync) scheme	5.133	4.820	5.952	10.572	20.916
	Hierarchy (Async) scheme	4.877	4.533	5.654	10.023	18.810
Problem 2	Master–Slave scheme	8.836	12.441	26.994	34.741	76.782
	Hierarchy (Sync) scheme	2.247	2.052	2.767	4.178	6.101
	Hierarchy (Async) scheme	1.992	1.893	2.320	3.899	5.109
Problem 3	Master–Slave scheme	45.651	89.270	177.021	352.185	–
	Hierarchy (Sync) scheme	0.350	0.459	0.623	1.129	–
	Hierarchy (Async) scheme	0.328	0.368	0.578	1.058	–

number of blocks). The lines with square symbols represent the execution time; the lines with circle symbols represent the solution quality. We have made the following observations on the result of solution quality and execution time. First, as the number of islands increases, the solution quality for the Hierarchy scheme becomes better. This observation is opposite to the typical theory of the pure Island scheme, which proves our theoretical analysis that the Hierarchy scheme on a many-core architecture is actually a complicated combination of Island and Cellular schemes, not a pure Island scheme. Furthermore, thanks to a fully-connected mesh topology and independent random generator which increase the isolation and diversity, the Hierarchy PGA on a GPU has more accurate results than other PGA schemes on a CPU (see Fig. 10(b). Second, the execution time decreases as the number of islands increases at first, however it stops decreasing when the number of islands (or blocks) is more than 64 or 128 (i.e., the number of threads in each block is less than 128 or 64). This is because each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently in current NVIDIA GPU architecture. The dual-warp scheduler selects two warps, and issues one instruction from each warp to a group of sixteen cores each time in order to decrease the probability of program branching happening. So the number of threads in a block should be large enough to provide necessary active warps so that the dual-issue can work. Thus, we should keep the number of threads in a block to at least 64.

5.4. Performance comparison of GA on multi- and many-core systems

From theoretical analysis and experimental results, we find that the Asynchronous Island scheme works best on a multi-core machine while the Island/Cellular Hierarchy scheme with asynchronous migrations is the best one on a many-core device. We choose these two PGA schemes to compare the performance of PGAs on multi-core and many-core systems without bias. Most previous works evaluate a GPU's speedup by comparing the execution time of a parallel program on a GPU to a sequential version on a CPU. Our comparison is more fair and reasonable.

Table 6 shows the result of the conventional performance evaluation approach; the execution time is measured in seconds. Each scheme's program is set with the best parameters. We keep the population size at 4096, and change the number of total generations. We observe that the GPU's speedup is over 10× if we use the conventional calculation method in all three test problems, while the speedup is actually around 3×–3.7× if we consider the best optimized scheme on the multi-core architecture. The speedup is quite stable among the three different problems and different number of total generations. This result is more reasonable when compared with other similar evaluations [8,10,11]. We further calculate the speedup of the Island (Async) Scheme compared to the serial Scheme on the multi-core CPU, which is 3.44× on average. As the multi-core CPU we used has 4 cores, the theoretical linear speedup is 4×. Thus, our approaches for implementing PGAs on

Table 6
The many-core architecture speedup compared to the multi-core architecture, conventional comparison approach.

Blank	Problem 1		Problem 2		Problem 3	
	40,000	100,000	40,000	100,000	40,000	100,000
Serial scheme on CPU	109.860	274.904	85.5956	215.7539	135.345	337.968
Island (Async) scheme on CPU	32.816	82.246	23.7205	60.3814	39.570	100.448
Hierarchy (Async) scheme on GPU	9.070	22.498	7.8311	19.4361	10.913	27.212
Speedup (Compared to serial on CPU)	12.11	12.22	10.93	11.10	12.40	12.42
Speedup (Compared to Island (Async) on CPU)	3.62	3.66	3.03	3.11	3.63	3.69

Table 7
The many-core architecture speedup compared to the multi-core architecture, considering solution quality.

Blank	Problem 1		Problem 2		Problem 3	
	0.2%	0.1%	0.02%	0.01%	0.2%	0.1%
Island (Async) scheme on CPU	40.377	∞	7.283	11.604	32.771	87.657
Hierarchy (Async) scheme on GPU	2.994	5.933	1.098	1.536	5.803	9.261
Speedup (Compared to Island (Async) on CPU)	13.49	∞	6.64	7.56	5.65	9.47

multi-core and many-core architectures are both well optimized and efficient.

Results in Sections 5.2 and 5.3 have demonstrated that different PGA schemes on different architectures have different solutions even if they are set with the same GA parameters. Thus, it is impractical and biased to evaluate the performance of different PGA schemes without taking into account the solution accuracy. We then use the execution time to reach a fixed solution quality as another metric to compare the performance of different PGA schemes. The result is shown in Table 7. This comparison is more practical and reasonable in real engineering fields. We use the same PGA configurations as those we used in Table 6, except setting a fixed solution accuracy for each run. The fixed solution accuracy for Test Problems 1–3 is 7064.0 (0.2%), 7056.0 (0.1%), and 2.38160 (0.02%), 2.38140 (0.1%), and 24.355 (0.2%), 24.331 (0.1%), respectively. The infinite time means that the program cannot reach the fixed solution within 200,000 generations. We observe that speedups of the GPU improve significantly when the solution quality is considered. Speedups shown in Table 7 are about 2–4 times higher than those in Table 6. This improvement benefits from the Hierarchy scheme, which has higher solution quality as described in Section 4. Moreover, as the accuracy increases, the GPU speedup becomes larger. It implies that we can gain more benefits of the execution time on a many-core architecture when needing more accurate solutions. The speedup shown in Table 7 provides a better guidance for scientists and engineers to make decisions on how to implement PGA applications on state-of-the-art hardware.

6. Related work

Researchers have been continuously studying the Genetic Algorithm (GA) in terms of new applications, new variants of the algorithm itself, and new implementations on different hardware. GAs have shown capabilities for solving many real world problems [18], including optimization [19], machine learning [20], scheduling [21,22], network analysis [23], hardware design [24,25], etc. Our work is not aimed at a particular GA application. We provide a generic performance analysis of GAs on different state-of-the-art hardware architectures, and present the GA model with the best performance on each architecture. These applications can directly take advantage of our techniques to reduce runtime overheads and achieve more accurate solutions, especially for large problems.

Parallel Genetic Algorithms (PGA) have attracted more interest as parallel architectures have become more popular. There has been much research on improving the performance of PGAs. The first category is focused on theoretical analysis of PGAs, including different PGAs operators [26,27], migration mechanisms [28] and population topologies [29,30]. They proposed new variants of the genetic algorithm itself to improve the convergence speed or the quality of solutions. This work is orthogonal to our research; we do not focus on a particular type of GAs, and provide a architecture-based design and optimization for implementing PGAs on different architectures, while others provide a low-level foundation for PGAs. These new GA variants can get further performance improvement by leveraging our design and optimization on different architectures. Cantú-Paz [12] documented a comprehensive survey of PGAs from theoretical analysis to implementation, and also analyzed the behavior of different PGAs on different architectures. But it focused on distributed-memory MIMD machines. We rethink the taxonomy of PGAs on state-of-the-art parallel architectures, and revisit the design and optimization of PGAs based on multi-core and many-core architecture features.

The second category is to explore potential benefits by using new hardware. The first PGA implementations relied on multiprocessor parallel machines or cluster computing systems [31,32], while we focus on multi- and many-core systems. With the introduction of multi- and many-core architectures, research shifted towards PGAs on multi-core CPUs and GPUs to alleviate multiprocessors and cluster systems inefficiencies, such as network overhead, and distributed memory access delay. Luong et al. [10] proposed an island-based PGA implementation on a GPU, and showed a speedup factor of thousands compared to a serial version on a CPU. We perform the comparison in a more reasonable way. Arora et al. [8] proposed parallelized versions of binary and real-coded GAs using CUDA, and studied the effect of GA parameters. We study the impact of different architecture features. Jaros et al. [33] proposed a multi-GPU island-based GAs for solving the knapsack problem, while we focus on making full usage of a single powerful GPU device. Tsutsui et al. [34] designed a coarse-grained PGA implementation on a GPU to solve the quadratic assignment problem, showing a speedup ratio from 3 to 12 times when compared to a four-core CPU. However, it does not take the solution quality into account like other works. The existing work has not been shown to give a general optimization mechanism based on multi- and many-core architecture features. More importantly, their comparison is unfair; they do not make the best of multi-core CPUs. Our work gives a deep analysis of the performance of PGAs on multi- core CPUs and many-core GPUs, and proposes generic optimization methods to achieve the best performance on different architectures. We show a reasonable speedup comparison between multi- and many-core systems, assessing both the execution speed and solution accuracy.

There are many researches in performance analysis and optimization on multi- and many-core systems. General analytical performance models and evaluations are proposed. Hong et al. [35]

showed an analytical model for a GPU architecture to estimate the execution time of massively parallel programs. Sim et al. [36] gave a performance analysis framework for identifying potential benefits in GPGPU applications. Our work differs from theirs, and studies GAs on multi- and many-core architectures in detail to improve the performance of both the execution speed and solution accuracy. Besides targeting at general analysis, other researchers have also proposed architecture-based performance optimization on specific applications, such as data processing [37], numerical algorithms [38] and execution models [39], for multi- and many-core systems. Like these work, our work emphasizes the necessity of careful design and optimization of parallel applications when considering architecture characteristics in depth.

7. Conclusion

In this paper, we have analyzed the performance of GAs on multi-core and many-core architectures, and constructed the relationship between GA performance and architecture features. We consider a combination of the execution speed and the solution quality as a more reasonable performance metric to compare the performance of different PGA schemes on different architectures. This is a distinct feature of performance evaluation for PGAs, in comparison with the previous work. The best PGA scheme for different architectures is explored. More specifically, the Asynchronous Island scheme, Island/Master–Slave Hierarchy scheme and Island/Cellular Hierarchy scheme has the best performance on multi-core, multi-socket multi-core and many-core architectures, respectively. We evaluate our proposed PGA schemes with three real-world GA problems. Our experimental results demonstrate the proposed Island/Cellular Hierarchy scheme on a many-core GPU outperforms all other PGA schemes. It has faster execution speed as well as better solution quality. Moreover, the speedup of the Island/Cellular Hierarchy scheme on a many-core architecture compared with the Asynchronous Island scheme on a multi-core architecture improves significantly when taking into account the solution accuracy. Our findings offer GA engineers and researchers practical guidelines to take advantage of the multi-core and the emerging many-core architectures.

Acknowledgments

This work was supported in part by NFSC (Grant Nos. 60811130528, and 61003012), Shanghai Excellent Academic Leaders Plan (No. 11XD1402900), Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT) China, and the Japan Society for the Promotion of Science (JSPS).

Appendix. Benchmarks description

The following terms describe the three testing benchmarks used in the performance evaluation which can be found at [17,40].

Test Problem 1. This problem has eight variables and six inequality constraints:

Minimize $f(\vec{x}) = x_1 + x_2 + x_3$

Subject to

$$g_1(\vec{x}) \equiv 1 - 0.0025(x_4 + x_6) \geq 0,$$
$$g_2(\vec{x}) \equiv 1 - 0.0025(x_5 + x_7 - x_4) \geq 0,$$
$$g_3(\vec{x}) \equiv 1 - 0.01(x_8 - x_5) \geq 0,$$
$$g_4(\vec{x}) \equiv x_1x_6 - 833.33252x_4 - 100x_1 + 83333.333 \geq 0,$$
$$g_5(\vec{x}) \equiv x_2x_7 - 1250x_5 - x_2x_4 + 1250x_4 \geq 0,$$
$$g_6(\vec{x}) \equiv x_3x_8 - x_3x_5 + 2500x_5 - 1250000 \geq 0,$$
$$100 \leq x_1 \leq 10000,$$
$$1000 \leq (x_2, x_3) \leq 10000,$$
$$10 \leq x_i \leq 1000, \quad i = 4, \ldots, 8.$$

The optimum solution is $f^*(\vec{x}) = 7049.330923$, and the best solution obtained by any genetic algorithm method in the literature is $f_G(\vec{x}) = 7060.221$.

Test Problem 2. This problem has four variables and five inequality constraints, and is known as the welded beam design problem (WBD):

Minimize $f(\vec{x}) = 1.10471h^2l + 0.04811tb(14.0 + l)$

Subject to

$$g_1(\vec{x}) \equiv 13600 - \tau(\vec{x}) \geq 0,$$
$$g_2(\vec{x}) \equiv 30000 - \sigma(\vec{x}) \geq 0,$$
$$g_3(\vec{x}) \equiv b - h \geq 0,$$
$$g_4(\vec{x}) \equiv P_c(\vec{x}) - 6000 \geq 0,$$
$$g_5(\vec{x}) \equiv 0.25 - \delta(\vec{x}) \geq 0,$$
$$0.125 \leq h \leq 10,$$
$$0.1 \leq l, t, b \leq 10,$$

The terms $\tau(\vec{x}), \sigma(\vec{x}), P_c(\vec{x}), \delta(\vec{x})$ are given below:

$$\tau(\vec{x}) = \left((\tau'(\vec{x}))^2 + (\tau''(\vec{x}))^2 + \frac{l\tau'(\vec{x})\tau''(\vec{x})}{\sqrt{0.25(l^2 + (h+t)^2)}}\right)^{\frac{1}{2}},$$
$$\sigma(\vec{x}) = \frac{504000}{t^2b},$$
$$P_c(\vec{x}) = 64746.022(1 - 0.0282346t)tb^3,$$
$$\delta(\vec{x}) = \frac{2.1952}{t^3b},$$

where

$$\tau'(\vec{x}) = \frac{6000}{\sqrt{2}hl},$$
$$\tau''(\vec{x}) = \frac{6000(14 + 0.5l)\sqrt{0.25(l^2 + (h+l)^2)}}{2\{0.707hl(l^2/12 + 0.25(h+t)^2)\}}.$$

The optimum solution is $f^*(\vec{x}) = 2.38116$, and the best solution obtained by any genetic algorithm method in the literature is $f_G(\vec{x}) = 2.38119$.

Test Problem 3. This problem has ten variables and eight inequality constraints:

Minimize

$$f(\vec{x}) = x_1^2 + x_2^2 + x_1x_2 - 14x_1 - 16x_2 + (x_3 - 10)^2 + 4(x_4 - 5)^2 + (x_5 - 3)^2 + 2(x_6 - 1)^2 + 5x_7^2 + 7(x_8 - 11)^2 + 2(x_9 - 10)^2 + (x_{10} - 7)^2 + 45.$$

Subject to

$$g_1(\vec{x}) \equiv 105 - 4x_1 - 5x_2 + 3x_7 - 9x_8 \geq 0,$$
$$g_2(\vec{x}) \equiv -10x_1 + 8x_2 + 17x_7 - 2x_8 \geq 0,$$
$$g_3(\vec{x}) \equiv 8x_1 - 2x_2 - 5x_9 + 2x_{10} + 12 \geq 0,$$
$$g_4(\vec{x}) \equiv -3(x_1 - 2)^2 - 4(x_2 - 3)^2 - 2x_3^2 + 7x_4 + 120 \geq 0,$$
$$g_5(\vec{x}) \equiv -5x_1^2 - 8x_2 - (x_3 - 6)^2 + 2x_4 + 40 \geq 0,$$
$$g_6(\vec{x}) \equiv -x_1^2 - 2(x_2 - 2)^2 + 2x_1x_2 - 14x_5 + 6x_6 \geq 0,$$
$$g_7(\vec{x}) \equiv -0.5(x_1 - 8)^2 - 2(x_2 - 4)^2 - 3x_5^2 + x_6 + 30 \geq 0,$$
$$g_8(\vec{x}) \equiv 3x_1 - 6x_2 - 12(x_9 - 8)^2 + 7x_{10} \geq 0,$$
$$-10 \leq x_i \leq 10, \quad i = 1, \ldots, 10.$$

The optimum solution is $f^*(\vec{x}) = 24.3062091$, and the best solution obtained by any genetic algorithm method in the literature is $f_G(\vec{x}) = 24.37248$.

References

[1] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou, Cilk: An efficient multithreaded runtime system, J. Parallel Distrib. Comput. 37 (1) (1996) 55–69.
[2] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, Evaluating mapreduce for multi-core and multiprocessor systems, in: ISCA '07, 2007.
[3] B. He, W. Fang, Q. Luo, N.K. Govindaraju, T. Wang, Mars: a mapreduce framework on graphics processors, in: PACT '08, 2008.
[4] A. Beham, S. Winkler, S. Wagner, M. Affenzeller, A genetic programming approach to solve scheduling problems with parallel simulation, in: IPDPS '08, 2008.
[5] A. Markham, N. Trigoni, Discrete gene regulatory networks dgrns: A novel approach to configuring sensor networks, in: INFOCOM '10, 2010.
[6] M. Lahiri, M. Cebrian, The genetic algorithm as a general diffusion model for social networks, in: AAAI '10, 2010.
[7] G. Renner, A. Ekart, Genetic algorithms in computer aided design, Comput. Aided Design 35 (8) (2003) 709–726.
[8] R. Arora, R. Tulshyan, K. Deb, Parallelization of binary and real-coded genetic algorithms on gpu using cuda, in: CEC '10, 2010.
[9] Z. Konfrt, Parallel genetic algorithms: Advances, computing trends, applications and perspectives, in: IPDPS '04, 2004.
[10] T. Luong, N. Melab, E. Talbi, Gpu-based island model for evolutionary algorithms, in: GECCO '10, 2010.
[11] P. Vidal, E. Alba, A multi-gpu implementation of a cellular genetic algorithm, in: CEC '10, 2010.
[12] E. Cantú-Paz, A survey of parallel genetic algorithms, Calculat. Parall., Reseauxet Syst. Repartis 10 (2) (1998) 141–171.
[13] J. del Cuvillo, W. Zhu, Z. Hu, G.R. Gao, Toward a software infrastructure for the cyclops-64 cellular architecture, in: HPCS '06, 2006.
[14] J. Shin, D. Huang, B. Petrick, C. Hwang, A. Leon, A. Strong, A 40 nm 16-core 128-thread sparc soc processor, in: A-SSCC '10, 2010.
[15] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, D. Yeung, The mit alewife machine: architecture and performance, in: ISCA '95, 1995.
[16] S. Xiao, W. Feng, Inter-block gpu communication via fast barrier synchronization, in: IPDPS '10, 2010.
[17] K. Deb, An efficient constraint handling method for genetic algorithms, Comput. Methods. Appl. Math. 186 (2-4) (2000) 311–338.
[18] E. Sanchez, G. Squillero, A. Tonda, Industrial applications of evolutionary algorithms, in: Intelligent Systems Reference Library, 2012.
[19] R. Battiti, A. Passerini, Brain-computer evolutionary multiobjective optimization: A genetic algorithm adapting to the decision maker, IEEE Trans. Evol. Comput. 14 (5) (2010) 671–687.
[20] S. Li, H. Wu, D. Wan, J. Zhu, An effective feature selection method for hyperspectral image classification based on genetic algorithm and support vector machine, Knowl.-Based Syst. 24 (1) (2011) 40–48.
[21] B.D. Frédéric Pinela, P. Bouvry, Solving very large instances of the scheduling of independent tasks problem on the gpu, J. Parallel Distrib. Comput. 73 (1) (2013) 101–110.
[22] M. Mezmaz, N. Melab, Y. Kessaci, Y. Lee, E.-G. Talbi, A. Zomaya, D. Tuyttens, A parallel bi-objective hybrid metaheuristic for energy-aware scheduling for cloud computing systems, J. Parallel Distrib. Comput. 71 (11) (2011) 1497–1508.
[23] S. Rahmani, S. Mousavi, M. Kamali, Modeling of road-traffic noise with the use of genetic algorithm, Appl. Soft Comput. 11 (1) (2011) 1008–1013.
[24] C.-C. Tsai, H.-C. Huang, C.-K. Chan, Parallel elite genetic algorithm and its application to global path planning for autonomous robot navigation, IEEE Trans. Ind. Electron. 58 (10) (2011) 4813–4821.
[25] D. Datta, A.R. Amaral, J.R. Figueira, Single row facility layout problem using a permutation-based genetic algorithm, European J. Oper. Res. 213 (2) (2011) 388–394.
[26] D. Whitley, D. Hains, A. Howe, A hybrid genetic algorithm for the traveling salesman problem using generalized partition crossover, in: PPSN XI, 2010.
[27] M. Giacobini, M. Tomassini, A. Tettamanzi, E. Alba, Selection intensity in cellular evolutionary algorithms for regular lattices, IEEE Trans. Evol. Comput. 9 (5) (2005) 489–505.
[28] D.-X. Chang, X.-D. Zhang, C.-W. Zheng, D.-M. Zhang, A robust dynamic niching genetic algorithm with niche migration for automatic clustering problem, Pattern Recognit. 43 (4) (2010) 1346–1360.
[29] N. Xiao, M.P. Armstrong, A specialized island model and its application in multiobjective optimization, in: GECCO '03, 2003.
[30] C. Gagné, M. Parizeau, M. Dubreuil, The master–slave architecture for evolutionary computations revisited, in: GECCO '03, 2003.
[31] T.C. Belding, The distributed genetic algorithm revisited, in: ICGA '95, 1995.
[32] E. Alba, J.M. Troya, Analyzing synchronous and asynchronous parallel distributed genetic algorithms, Future Gener. Comput. Syst. 17 (4) (2001) 451–465.
[33] J. Jaros, Multi-gpu island-based genetic algorithm for solving the knapsack problem, in: CEC '12, 2012.
[34] S. Tsutsui, N. Fujimoto, Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study, in: GECCO '09, 2009.
[35] S. Hong, H. Kim, An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness, in: ISCA '09, 2009.
[36] J. Sim, A. Dasgupta, H. Kim, R. Vuduc, A performance analysis framework for identifying potential benefits in gpgpu applications, in: PPoPP '12, 11–22.
[37] B. He, M. Lu, K. Yang, R. Fang, N.K. Govindaraju, Q. Luo, P.V. Sander, Relational query coprocessing on graphics processors, ACM Trans. Database Syst. 34 (4) (2009) 21:1–21:39.
[38] L.-W. Chang, J.A. Stratton, H.-S. Kim, W.-M.W. Hwu, A scalable, numerically stable, high-performance tridiagonal solver using gpus, in: SC '12, 2012.
[39] W. Fang, B. He, Q. Luo, N.K. Govindaraju, Mars: Accelerating mapreduce with graphics processors, IEEE Trans. Parallel Distrib. Syst. 22 (4) (2011) 608–620.
[40] Z. Michalewicz, Genetic algorithms numerical optimization and constraints, in: ICGA '95, 1995.

Long Zheng received the Ph.D. degree in computer science and engineering from the University of Aizu, Aizu-wakamatsu, Japan in 2012; the M.S. degree in computer science and engineering from the University of Aizu in 2009, the M.S. degree in computer science and technology from Huazhong University of Science and Technology (HUST), Wuhan, China in 2010; and the B.S. degree in computer science and technology from HUST in 2006. He was a visiting scholar at the Embedded Pervasive Computing Center, Shanghai Jiao Tong University, Shanghai, China in 2011. He was rewarded for the Grant for Non-Japanese Researchers by NEC C&C Foundation, Japan during his doctoral program. He was a recipient of the Fellowship for Young Scientists of the Japan Society for the Promotion of Science (JSPS) in 2012. He is currently the recipient of the Post Doctoral Fellowships of JSPS. His research interests include multi- and many-core architectures, parallel and distributed processing and cloud computing.

Yanchao Lu received the B.S. degree in computer science and technology from Beijing Institute of Technology (BIT), China, in 2010. He is currently a second year Ph.D. student at the Department of Computer Science and Engineering, Shanghai Jiao Tong University (SJTU), China. His research interests include GPGPU, parallel and distributed systems, cloud computing, performance evaluation of computer systems and genetic algorithms.

Minyi Guo received his Ph.D. degree in computer science from the University of Tsukuba, Japan. Before 2000, Dr. Guo had been a research scientist of NEC Corp., Japan, and a professor in the School of Computer Science and Engineering, University of Aizu, Japan. Currently, he is also a guest professor at Nanjing University, Huazhong University of Science and Technology, and Central South University, China.

Dr. Guo is a senior member of the IEEE and the IEEE Computer Society, and a member of the ACM, IPSJ, CCF, and IEICE. He has served as general chair, program committee, and organizing committee chair for many international conferences. He is the founder of the International Conference on Parallel and Distributed Processing and Applications (ISPA) and the International Conference on Embedded and Ubiquitous Computing (EUC). He is the editor-in-chief of the Journal of Embedded Systems. He is also on the editorial board of IEEE Transaction on Parallel and Distributed Systems, the Journal of Computer Science and Technology, the Journal of Pervasive Computing and Communications, the International Journal of High Performance Computing and Networking, the Journal of Embedded Computing, the Journal of Parallel and Distributed Scientific and Engineering Computing, and the International Journal of Computer and Applications.

Dr. Guo conducted research in parallel and distributed processing, parallelizing compilers, pervasive computing and embedded systems software optimization. Nowadays, his research interests shifted to high performance computing and cloud computing.

Song Guo received the Ph.D. degree in computer science from the University of Ottawa, Canada in 2005. He is currently a Senior Associate Professor at the School of Computer Science and Engineering, University of Aizu, Japan. His research interests are mainly in the areas of protocol design and performance analysis for computer and telecommunication networks, presently focusing on network modeling, cross-layer design and optimization, security analysis, and performance evaluation of wireless, mobile, and personal networks. Dr. Guo is an associate editor of the IEEE Transactions on Parallel and Distributed Systems and an associate editor of Wireless Communications and Mobile Computing. He is a senior member of the IEEE.

Cheng-Zhong Xu received the B.S. and M.S. degrees from Nanjing University in 1986 and 1989, respectively, and the Ph.D. degree in computer science from the University of Hong Kong in 1993. He is currently a professor in the Department of Electrical and Computer Engineering, Wayne State University and the director of Sun's Center of Excellence in Open Source Computing and Applications. His research interests are mainly in distributed and parallel systems, particularly in scalable and secure Internet services, autonomic cloud management, energy aware task scheduling in wireless embedded systems, and high performance cluster and grid computing. He has published more than 160 articles in peer-reviewed journals and conferences in these areas. He is the author of Scalable and Secure Internet Services and Architecture (Chapman & Hall/CRC Press, 2005) and the coauthor of Load Balancing in Parallel Computers: Theory and Practice (Kluwer Academic/Springer, 1997). He serves on five journal editorial boards including IEEE TPDS and JPDC. He was a program chair or general chair of a number of conferences, including Infoscale '08, EUC '08, and GCC '07. He is a recipient of the Faculty Research Award of Wayne State University in 2000, the President's Award for Excellence in Teaching in 2002, and the Career Development Chair Award in 2003. He is a senior member of the IEEE.