Architecture-based Performance Evaluation of Genetic Algorithms on Multi/Many-core Systems

Long Zheng[†§], Yanchao Lu[‡], Mengwei Ding[‡], Yao Shen[‡], [*]Minyi Guo[‡] and Song Guo[§]
[†]*School of Computer Science and Technology*
Huazhong University of Science and Technology, Wuhan, 430074, China
[§]*School of Computer Science and Engineering*
The University of Aizu, Aizu-wakamatsu, Fukushima-ken, 965-8580, Japan
[‡]*Department of Computer Science and Engineering*
Shanghai Jiao Tong University, Shanghai, 200240, China
[*]*Email:guo-my@cs.sjtu.edu.cn*

***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 GAs on multi-core CPUs and on GPUs, with solution quality considered. We describe widely-used parallel GA schemes based on Master-Slave, Island and Cellular models. Then, based on the multi-core and many-core architectures, especially the thread organization, memory hierarchy, and core utilization, we analyze the execution speed and solution quality of different GA schemes theoretically. Finally, we can point to the best approach to use on multi-core and many-core systems to execute GAs, so that we can obtain the highest quality solution at a cost of the shortest execution time.**

Furthermore, there are three extra contributions. Firstly, during our analysis and evaluation, we not only focus on the execution speed of different schemes, but also take the solution quality into account, so that our findings will be more useful in practice. Secondly, during our optimization of an Island scheme on GPUs, we find that the GPU architecture actually alters the scheme, making it become the Cellular scheme, which leads to big changes in solution quality and optimization results. Finally, we calculate the GPU speedup based on a comparison between the best scheme on a GPU and the best one on a CPU, rather than between an optimized one on the GPU and the worst one on a CPU, so that the speedup we calculate is more reasonable and a better guide to practical decisions.

***Keywords*-Genetic Algorithm; multi-core; GPU; accuracy; architecture; speedup;**

I. Introduction

Nowadays, multi-core processors and GPUs have entered the mainstream of microprocessor development. By putting several cores on a chip, multi-core processors based on Multiple Instruction Multiple Data (MIMD) parallelism can easily improve the performance. Unlike a multi-core processor, a GPU or many-core co-processor organizes hundreds of cores into Streaming Multiprocessors (SMs) to support Single Instruction Multiple Data (SIMD), which is well-suited to data-dense computing. 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 any more. 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 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 and MapReduce [3] to interact with hundreds of cores. 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, which is widely used in business, engineering and science [4], [5], [6], [7]. Because GAs require a huge amount of computation, each time a new parallel computing mechanism or new type of parallel hardware emerges, GAs take advantage of it. Multi-core processors and GPUs have attracted much interest from the GA field, and many GA applications have been ported to these systems [8], [9], [10], [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 GAs on multi-core and many-core systems are done on a case by case basis. These implementations have more commonalities than differences but previous work emphasizes the differences caused by the different GAs. The commonalities have not been much discussed.

(2) Many implementations do not consider the architecture issues deeply; some work even misunderstands the fundamentals of implementation on the new architectures.

(3) Most work focuses exclusively on the relationship between the speedup and architecture, ignoring the solution quality. Since GAs mostly find only approximate solutions, more attention should be paid to the quality of solutions. The relationship between speedup and architecture should be discussed along with the solution quality.

(4) In most previous work, the speedup of GPUs is calculated by comparing the execution speed on GPUs to a serial implementation on CPUs. The Serial GA is the worst scheme in execution speed, as it does not take any advantage of a multi-core architecture. Multi-core processors are now in the mainstream, so a speedup calculated in this way neither practical nor reasonable.

The drawbacks above 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 several popular parallel GA schemes: Master-Slave, Synchronous Island, Asynchronous Island, and Cellular. We abstract the implementation of different parallel GA schemes to the thread model. Then we can analyze the performance of different schemes on multi-core and many-core architectures theoretically, showing how the architecture impacts the different schemes in different ways. During our analysis, we also compare the performance of different schemes on different architectures, so that we can find out which scheme can fully take advantage of each architecture. Finally we choose a real problem solved by GA to evaluate our analysis.

Unlike previous work on evaluating the performance of Parallel GAs, we emphasize the solution quality. We evaluate the execution time that GAs take to reach a fixed-accuracy solution. With this evaluation, experimental results validate our theoretical analysis.

Furthermore, we find that the GPU architecture changes the Island scheme. The Island scheme on a many-core architecture becomes a complicated Cellular scheme with a migration mechanism. The change improves the solution quality, and also makes the Island scheme on a many-core architecture not follow the normal theory of Island schemes.

Although our discussion and analysis of multi-core and many-core architecture are based on GAs, similar conclusions also apply to Evolutionary Algorithms, Neural Networks and Machine learning on these systems. The general considerations about these architectures also can be used in other research fields using multi-core and many-core system.

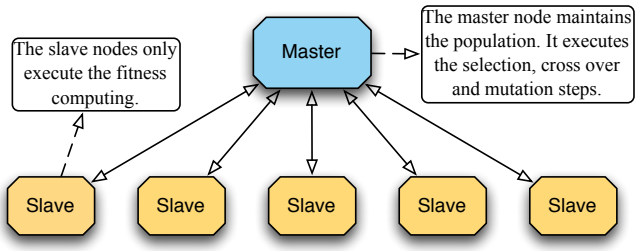


Figure 1. Diagram of Master-Slave scheme

The remainder of this paper is structured as follows. Section II gives a short overview of GA. Sections III and IV offer an architecture based analysis of GA on multi-core and many-core systems, respectively. We evaluate our analysis in V. Section VI summarizes our findings.

II. 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.

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 composed of 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].

GAs can be effective for finding approximate solutions to optimization and search problems in an acceptable amount of time, so the technique is successfully used in business, engineering and science. However, since a GA uses huge numbers of individuals that compose a population to search for probable solutions over many generations of evolution, GA applications use a lot of computation. New parallel

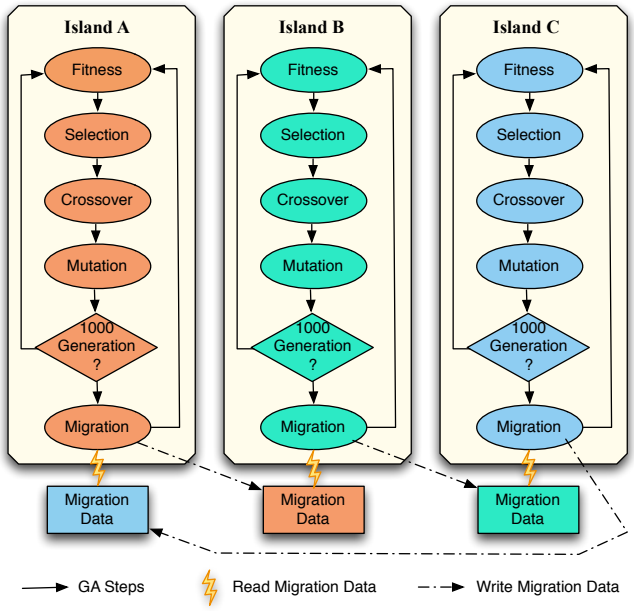


Figure 2. Diagram of Island scheme

and distributed computing systems can provide the computing power GAs require. Several GA schemes have been proposed to make GAs parallel, such as the Master-Slave scheme, the Synchronous Island scheme, the Asynchronous Island scheme and the Cellular scheme [12].

Fig. 1 illustrates how the Master-Slave scheme works. First 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 assign 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*.

Unlike the Master-Slave scheme which is a tightly coupled structure, the Synchronous and Asynchronous Island schemes are loosely coupled. In the Island scheme, which is illustrated in Fig. 2, the population of a GA is divided into several sub-populations, called islands. The populations in islands evolve in isolation from each other. After a fixed number of generations, islands communicate and exchange some of their best individuals; this is called *migration*. With *migration* the local best solutions in islands can spread to other islands, so that the population can evolve to better solutions. The Island scheme is perfectly suitable for the parallel and distributed environment. With the island scheme, the computation load can be distributed to different computing devices without huge overheads caused by the communication between islands. Fig. 2 also illustrates a widely used implementation of migration, based on a ring topology. Islands are connected by migration in a ring topology. Each island maintains a Migration Data structure to support the migration. When an island does a *migration*,

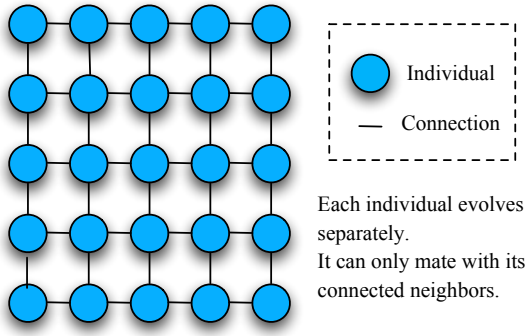


Figure 3. Cellular scheme with mesh topology

it puts a fixed number of individuals into its neighbour's Migration Data structure. After that, each island gets the individuals from its Migration Data structure. Then the *migration* step is complete.

The difference between the Synchronous Island and Asynchronous Island schemes lies only in the method of *migration*. In the Synchronous scheme, migration takes place when all islands achieve a fixed number of generations, usually 1000 generations. A faster island has to wait for slower islands to evolve to the fixed number of generations. The scheme guarantees that the migration occurs only when all islands achieve the same generation, which means there must be a synchronization operation for all islands in the migration. In the Asynchronous Island scheme, although all islands do the migration every fixed number of generations, islands do not have to wait for each other. An island just puts its best individuals into its neighbour's Migration Data structure, and gets the individuals from its own Migration Data structure. In this scheme, an island may get individuals that were sent by its neighbour in the previous migration, or get the same individuals as the the ones in the last migration, or even get no individuals. The Asynchronous Island schemes avoids synchronization operations among islands in the migration.

The Cellular scheme is another popular parallel implementation of GA, which is illustrated in Fig. 3. 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 neighbors of individuals are decided by the connection topology. In Fig. 3, the mesh topology is applied. Unlike the Master-Slave and Island schemes which are coarse grained, the Cellular scheme is a fine grained parallel scheme.

From the previous work on GAs, the solution qualities of the three schemes should be different. The serial and Master-Slave schemes do the *selection*, *crossover* and *mutation* steps on the whole population. On the other hand, the Island scheme does these steps in the island, that is, on a subset of the population. The Serial and Master-slave schemes select

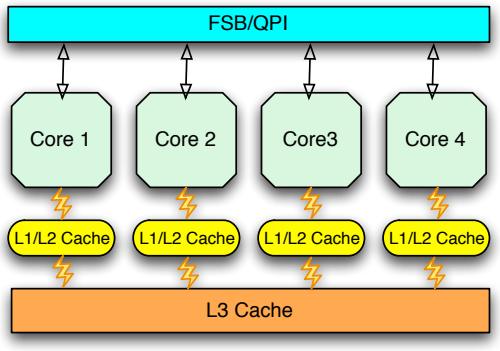


Figure 4. The multi-core architecture

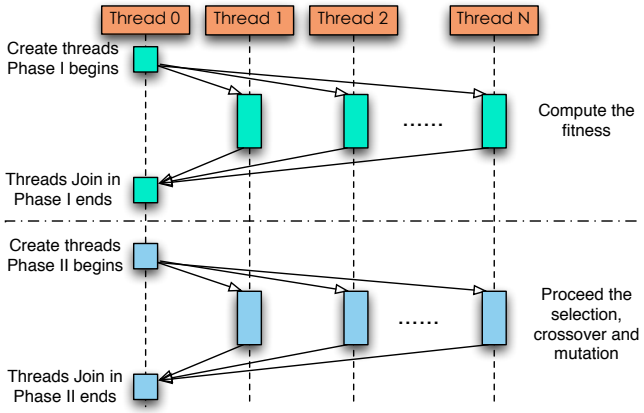


Figure 5. The Master-Slave scheme on the multi-core architecture

the globally best individuals from the whole population, but the Island scheme only gets the locally best individuals in each island. Also, in other schemes an individual can choose its mate globally, but in Island schemes the choice is restricted to the island. Therefore, the solution qualities of the Serial and Master-Slave schemes are better than those the Island scheme, which is quite similar to the species selection and evolution in nature. However which solution is better between the Synchronous Island and Asynchronous Island schemes is unknown, which is an issue we explore in this paper.

III. Architecture based Analysis of GA on Multi-core Systems

Multi-core chips doing Symmetric Multi-Processing (SMP) are now in the mainstream. They bring the parallelism in microprocessors from Instruction Level Parallelism (ILP) to Thread Level Parallelism (TLP). In a multi-core system, when a hyper-thread is not enabled, each core executes a thread simultaneously, which improves the performance. However, the way to organize the threads is the key that affects the performance of a multi-core system. Therefore, a multi-core architecture puts constraints on the implementations of different GA schemes, and affects the performance of GAs both in speed and in quality of solutions. In this section we first give a short description of the multi-core architecture and multi-threading on it, then we analyze how that affects the different GA schemes.

A. 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. In order to take advantage of multi-core processors, programmers must implement multi-threaded programs. With the help of the operating system, threads are assigned to cores. The multi-core architecture is quite similar to the previous multiprocessor architecture in which they are both SMP, and the operating system treats them almost the same. However, as a multi-core processor integrates cores into a chip, cores on the chip share the Level 3 cache or even Level 2 cache. The communication between cores is much faster than between processors in a multiprocessor architecture. The cache coherence mechanism is more important, easy and efficient. Furthermore, since threads share their virtual address space, they can get a performance improvement from the shared caches between cores. A multi-core architecture fits the multi-threaded programs and those programs can get more performance benefit than on previous multiprocessor architectures. A typical multi-core architecture is illustrated as Fig. 4.

However, it is not easy to use the power of multi-core processors. First of all, programmers have to write multi-thraeded code. Then the communication between threads, for example, synchronization, locks and so on, will influence the performance. Also, how the threads are organized in a program can affect the performance. Therefore, we will analyze the performance of GAs from the multi-core architecture view.

Several libraries can help programmers write multi-threaded programs, for example Pthread, OpenMP, Cilk and so on. In our work, we choose Pthread to implement different schemes of the GA.

B. The Master-Slave Scheme on Multi-core Architecture

A Serial GA can run on the 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

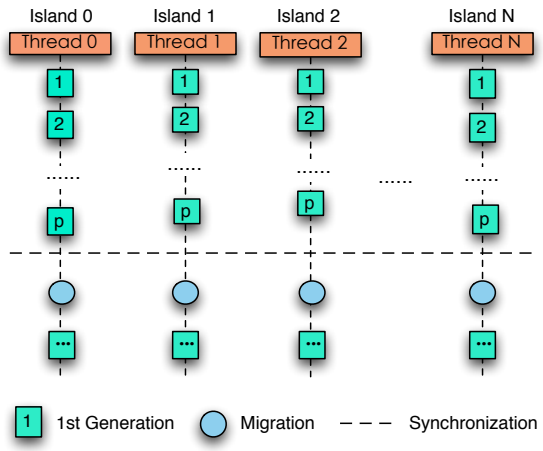


Figure 6. The Synchronous Island scheme on a multi-core architecture

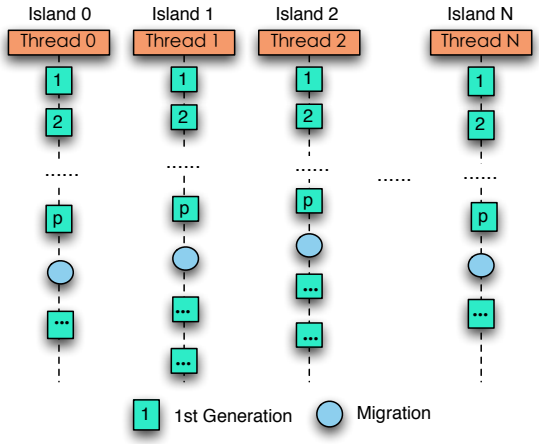


Figure 7. The Asynchronous Island scheme on a multi-core architecture

cores, which means we need to decentralize the Master-Slave scheme on the multi-core architecture.

With the 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 multi-core architecture is illustrated in Fig. 5. 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' 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 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 the N and c do not follow the rule displayed in Equation (1), the performance decreases. For example, if Thread 0 creates 5 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's obvious that the execution time that the 4-core processor takes to executes 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 Equation (1), or returns 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 Equation (1). In practice, f_1^{max} and f_2^{max} are very small, because the computation in Phases I and II is small. Therefor the $mod(N, c) \cdot r$ part can affect the performance, which requires us to keep threads aligned to avoid wasting mutli-core computation capacity.

C. Island Scheme on Multi-core Architecture

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. 6 and 7 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.

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

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 Equation (3) can be rewritten as

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

With Equation (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. 7. 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

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

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 Equation (4) is bigger than d_i^{avg} in Equation (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.

D. The Mixed Scheme on Multi-core Architecture

A Mixed scheme combines the Master-Slave and Island schemes. It uses islands and the relationship between islands follows the Island scheme. However, there are several threads in each island following the Master-Slave scheme.

Fig. 8 shows the Mixed scheme. In the figure, there are two islands and each island consists of two threads. It is noted that Thread 0 is the control thread, which is

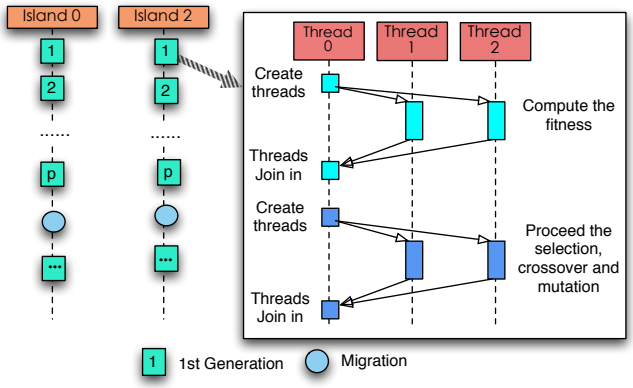


Figure 8. The Mixed scheme on the multi-core architecture

omitted. We define the number of threads in a island as the Parallelism Degree (PD).

The value of PD affects the solution quality. With a fixed size of population and fixed number of threads, as the value of PD increases, the size of each island increases and the number of islands decreases. As we mentioned in Section II, with the same size of population, more islands mean worse solution quality. Therefore, as the PD increases the solution quality gets worse and worse. Let's consider the extreme case. When PD is one, the Mixed scheme is the Master-Slave scheme, which can get the best quality of solutions; on the other hand, when PD is equal to the total number of threads, the Mixed scheme is the Island scheme, which leads to the worst quality of solutions.

The value of PD also affects the execution time. The more threads are in an island, the more synchronization operations are needed every generation. Although all threads of all islands are parallel, the synchronization operations are serial. We divide the implementation of the Mixed scheme into two parts-serial ($\mathcal{S}$) and parallel ($\mathcal{P}$) parts. As the value of PD increases, the serial part increases but the parallel part stays the same, so the ratio of $\mathcal{S}$ to $(\mathcal{S} + \mathcal{P})$ increases. According to Amdahl's Law, the speedup of the multi-core processor decreases, which implies the execution time of the implementaton of the Mixed scheme gets longer. According to this analysis, we can also conclude that the execution time of the Master-Slave scheme is longer than that of the Island scheme.

Summarizing, The Master-Slave scheme has the best solution quality, but the worst execution speed and the Island Scheme is the opposite; it has the best execution speed, but the worst solution quality. For the Synchronous and Asynchronous Island schemes, we are sure the Asynchronous one is faster, but the solution quality cannot be decided by our analysis above; this is left to our experiments. The Mixed Scheme is a transitional form between the Master-Slave and Island scheme. As the value of PD varies from one to the

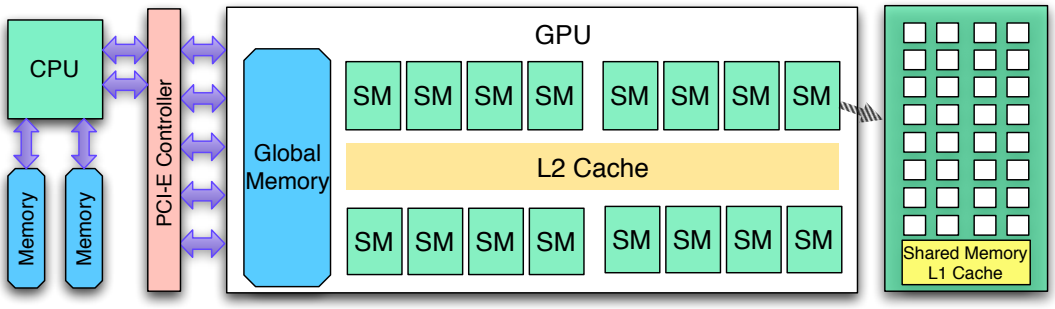


Figure 9. The GPU architecture with a CPU

number of threads, the execution speed of the Mixed Scheme gets slower, however, the solution quality gets better. There is a trade-off between execution speed and solution quality. If the vaule of PD is set appropriately, the Mixed scheme should get a best solution within the shortest time among all schemes we analyzed above.

IV. Architecture based Analysis of GA on Many-core Systems

A GPU is composed of hundreds or even thousands of cores, so it is also called a many-core system. Since the GPU can offer a powerful computation capacity for data-parallel computing, it has attracted much attention from GA researchers. In this section, we will first introduce the implementation of a GA on a many-core system, then analyze the execution time and solution quality of GAs from the viewpoint of many-core architecture. Since some analysis issues about GAs are similar to the multi-core architecture, we focus on the differences between many-core and multi-core architecture, as well as the unique characteristics of GAs on a many-core architecture.

A. Many-core Architecture and Constraints

Although the multi-core and many-core architectures both advance the performance with parallel cores and are TLP, the fundamentals are quite different. A multi-core CPU is an MIMD (Multiple Instructions, Multiple Data) system, while a many-core system is SIMD (Single Instruction, Multiple Data). Therefore, the implementations of GAs on the two kinds of architecture are quite different. Also, the implementation has to consider the constraints of GPUs.

The GPU is considered as an accelerated co-processor in the computer system, which implies that the GPU has to be controlled by a CPU. The architecture of a GPU and its corresponding controller CPU is illustrated in Fig. 9. In this figure, we show the latest Fermi GPU architecture. The CPU can access main memory via QPI or FSB, and communicate with the GPU via a PCI-E controller. The GPU has its own memory hierarchy. There are many cores in a GPU. Numbers of cores are organized into a Streaming Multiprocessor (SM) and a GPU is composed of several SMs.

The Fermi architecture has evolved considerably beyond its predecessor. In Fermi architecture, there are 32 cores in a SM. All SMs have shared Level 2 cache, and each SM has a configurable shared memory and Level 1 cache whose capacity is 64KB in total. The shared memory and Level 1 cache are actually implemented by the same hardware; and they can organized into either 16KB/48KB or 48KB/16KB. In the previous architecture, there are only 8 cores and 16KB shared memory in a SM. Although the shared memory has a very small access latency; however it is totally exposed to programmer. An inappropriate use of shared memory leads to a big performance penalty. The new Fermi architecture alters part of shared memory into the Level 1 cache, so that programmers can get the performance improvement even if they do not use any shared memory at all. Besides, the Level 2 cache in the Fermi architecture can reduce the overhead of access from GPU cores to the global memory.

A GPU task is called a *kernel*, and is launched by the CPU. When a kernel is about to be launched, the CPU transfers the data from main memory to GPU global memory, then the kernel runs. After the kernel completes, the CPU needs to copy the result data from GPU global memory back to main memory. Because main memory and GPU memory are independent of each other, the data transfer is expensive, and should be avoided.

The organization of threads for a GPU is quite different. Threads for the CPU are managed by the operating system, but threads for a GPU are managed by the GPU driver. Threads for GPUs are first grouped into wraps. A wrap is the smallest scheduling unit of threads for a GPU. Threads in a wrap execute the same instruction exactly concurrently. Wraps are further grouped into blocks. A block is the smallest resource unit of threads for a GPU. Threads in a block can access the shared block data in the shared memory. Blocks are assigned to a particular SM by the GPU driver. Several blocks compose a grid. All threads in a grid will run on the same GPU. When we are coding a program on GPU, we must follow the basic rules below, otherwise the performance of the GPU degrades badly.

(1) The shared memory of a SM should at least meet the

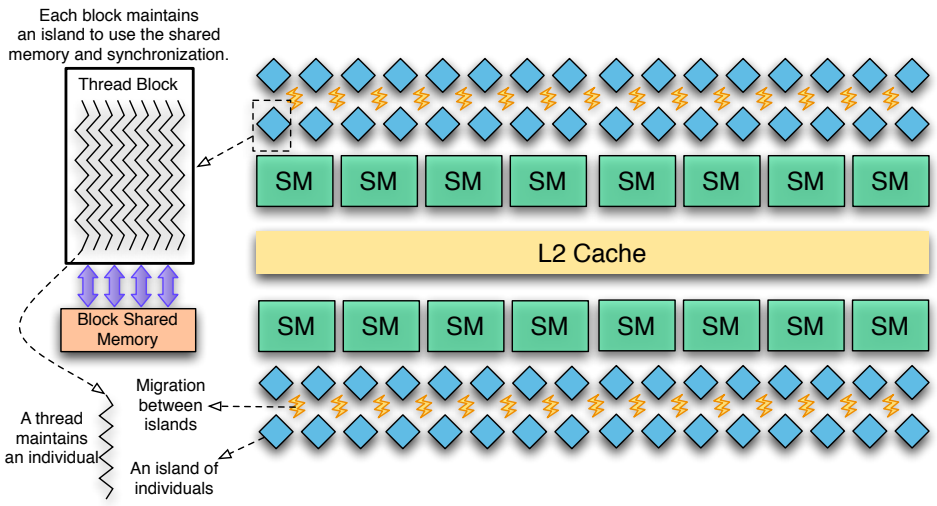


Figure 10. The Island scheme on a GPU

requirement of one block, so that the kernel can be compiled and executed.

(2) There should be at least 32 threads in a block, which is the size of a wrap, 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 the 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.

B. Analysis of GA Implementations on Many-core System

As we analyzed above, the many-core architecture presents several rules for us to implement different schemes. In this subsection, we will analyze the implementations of Master-Slave, Synchronous Island and Asynchronous Island schemes on a many-core architecture.

It is very simple to implement the Master-Slave scheme on a many-core system, as the relationship between CPU and GPU is master and slave. In the Master-Slave scheme, the CPU does the selection, crossover, and mutation; and the GPU calculates the fitness. However, the Master-Slave scheme would be very inefficient for two reasons. One is that it needs to transfer the population data between main memory and the GPU global memory every generation. And the computation for each generation is very little, so the communication between CPU and GPU takes most of the execution time. The other is that the GPU only calculates the fitness, but the CPU needs to do the other steps of GAs. The CPU will probably be the bottleneck of the whole system, since the GPU computation computation capacity is much more powerful than that of the CPU, and the workload of selection, crossover and mutation is even heavier than that of fitness calculation in some cases. Therefore, although the many-core system is naturally a master-slave model, it is not a good choice to adopt the Master-Slave scheme.

According to our description and analysis in Section IV-A, threads in a block can use the shared memory to communicate with little overhead; threads in different blocks can communicate but at cost of high overhead. Hence, from the viewpoint of architecture, 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. 10.

The communication between individuals in an island can use the synchronization operation provided by CUDA. However because there is no synchronization operation among blocks in a GPU, the synchronization operation in the Synchronous Island scheme must be implemented by programmers using the global memory. Efficient synchronization among blocks is discussed in detail in [13].

As on the multi-core architecture, the Asynchronous Island scheme should be faster than the Synchronous Island scheme. There are two reasons. One is that with an Asyn-

chronous Island scheme, the faster blocks do not need to wait for the slower blocks, so that the faster blocks can continue to evolve without any interruptions, but with the Synchronous Island scheme, the faster ones need to wait for the slower ones. The other is that the Synchronous Island scheme needs to do synchronization among blocks leading to extra accesses to the global memory. Compared to the previous architecture, the Fermi architecture has Level 2 cache that can improve the performance of access to the global memory. Besides, the synchronization operations among blocks are not frequent, so the overhead caused by synchronization among blocks in the Fermi architecture will be much smaller than in the previous architecture.

As we mentioned above, in order to fully make use of all SMs, we must set the number of islands to at least the number of SMs. In order to fill the wrap, each island must have at least 32 individuals. When the Island scheme is implemented, the shared memory should be used, but the size of shared memory required by our code can not exceed 48KB. Furthermore, there are not many accesses to the global memory, as the communication between blocks only occurs when migration is done. Therefore, the implementation of the Island scheme does not have to use a large number of threads to overlap the latency of access to the global memory.

C. Argument for Island Scheme on Many-core Systems

In implementing the Island scheme on the many-core architecture, the thread model of GPUs changes the scheme.

In every scheme of GAs, the random generation sequence is a key factor that influences the solution quality, so it can be considered a key feature of GAs. In the Master-Slave scheme, all individuals of a population use the same random generation sequence. In the Island scheme, each island maintains a random generation sequence. The Master-Slave and Island schemes on the multi-core architecture can exactly obey the rules above using a lock mechanism to get a safe parallel random generation sequence. When the Master-Slave scheme is applied to many-core systems, all operations that need random numbers run on the CPU, so the system exactly follows the rules.

Whereas, when the Island scheme is implemented on a many-core system, each thread executes the evolution of an individual. The threads in a wrap execute each instruction concurrently, if we use a lock mechanism to generate safe random numbers, threads getting random numbers will be forced to be serialized, which degrades the GPU performance dramatically.

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 call 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 16GB memory capacity. The largest global memory capacity of GPUs now is 6GB. 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.

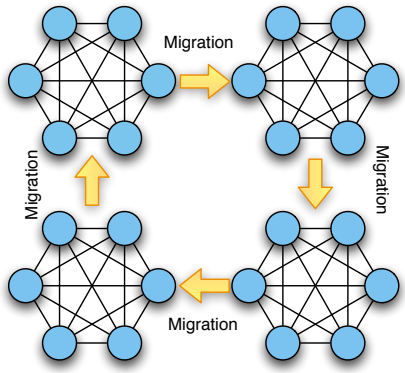


Figure 11. The Island scheme on the many-core architecture is actually a complicated Cellular scheme

Therefore, we have to make each thread maintain an independent random generation sequence; this is called dynamic random generation. It violates the Island scheme. Each individual is independent, and all individuals in an island communicate for selection, crossover and mutation. It is similar to the Cellular scheme with a fully connected topology. Therefore, a population that uses an Island scheme on a many-core architecture is composed of several populations each using the Cellular scheme. However these populations do migration after a fixed number generations. The Island scheme on a many-core architecture cannot be considered a conventional Island scheme; it is a complicated Cellular scheme, which is illustrated in Fig. 11.

In this section we began with an analysis of the many-core architecture and presented several rules that should be followed when GAs are implemented. Then we analyzed the performance of the Master-Slave and Island schemes on a many-core architecture. Most importantly, because of the features of the many-core architecture, we found that the Island scheme on the many-core architecture should be a Cellular scheme with a migration mechanism. Our experiments will explore the performance of the Island scheme on a many-core architecture.

V. Performance Evaluation

With the analysis presented in Sections III and IV, we get several theoretical results. In experiments, we choose a real GA problem which is widely used in engineering as our benchmark to validate our theoretical analysis. Besides,

although the speedup by GPUs has been exploited in the previous work, the solution quality has not been taken into account. In our work, we measure the speedup of GPUs compared to the best parallelized GA implementation on a multi-core processor, considering solution quality. Our results on the speedup of GPUs are more practical and useful for researchers and engineers.

A. Experiment Setup

The experiments in the previous work on GAs on GPUs usually only measure the execution time, and then compare it with that on the CPU. The accuracy of the solution, the importance of solution quality, is always underestimated. In our experiments, we not only measure the execution time, but also the accuracy. We use the execution time to reach a fixed solution as our main metric to evaluate both the execution speed and solution quality. This metric can tell us which scheme can get the best solution during the same time.

Also, in most previous work, when comparing the performance between GPU and CPU, the implementation of GAs on the CPU is serial, without any parallelization or optimization. This is not fair to the CPU, since multi-core processors are now common. In our experiments, we choose the best GA scheme on CPUs in our analysis above as the baseline, so that the speedup of GPUs are more convincing and reasonable.

In the evaluation, we use GA to solve a Nonlinear Programming (NLP) problem, which is used as the benchmark to evaluate the performance of different schemes on both the multi-core and many-core architectures. NLP techniques are widely used in industrial management, engineering design, scientific computation, the military and so on [7]. The NLP problem we used is as follows. This problem is widely used as the test problem in optimization test collections. The optimum solution is that $f = 7049.330923$ [14]. The experimental environments are listed in Table I.

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

Subject to

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

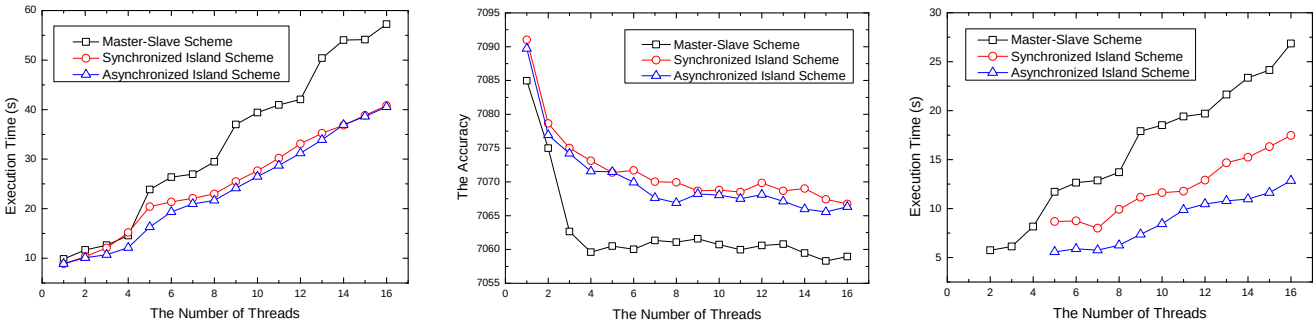
Table I
THE EXPERIMENTAL ENVIRONMENT

CPU	Intel i7-2600 Quard-core Processor
Hyper-Thread on CPU	Disabled
Turbo Boost on CPU	Disabled
Main Memory	6GB
GPU	Nvidia GTX580, 1.5GB Global Memory
OS	Ubuntu 10.04 Server 64bit
Kernel Version	2.6.38-8
GCC	4.5.2 with -O2 optimization option
CUDA Version	SDK 3.2
GPU Driver	270.41.06-x86_64 for Linux

B. Experimental Results

We first compare the Master-Slave, Synchronous Island and Asynchronous Island schemes on a multi-core architecture. Fig. 12 shows the execution time, solution quality and the execution time to reach the fixed solution. In experiments in Fig. 12, 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. From Fig. 12(a), we find that the Master-Slave scheme is the slowest, while the Asynchronous Island scheme is the fastest. Also, the execution time of the Master-Slave scheme shows the impact of thread align. However, for solution quality, the Master-Slave scheme is the best, while the Synchronous Island scheme is the worst, which is depicted in Fig. 12(b). We find that the Asynchronous Island scheme can reach the fixed solution quality fastest from Fig. 12(c). We set the fixed solution quality as within 0.2% of the optimum solution. Therefore, the Asynchronous Island scheme is the best one on a multi-core architecture. However, it is noted that when size of population is smaller than 768, the Island scheme can not reach the fixed solution but the Master-Slave one can.

Fig. 13 shows the performance of the Mixed scheme on a multi-core architecture. Fig. 13(a) expresses that the execution time increases as the value of PD increases. Fig. 13(b) shows that the solution is getting closer to the optimum solution as the PD varies incrementally. So we need to find an appropriate value of PD that use the least time to reach the fixed solution. From Fig. 13(c), the appropriate value of PD is 4. As we analyzed before, when PD is equal to 1, it is the Asychronized Island scheme, so that the Mixed scheme in which each island uses 4 threads is the best scheme on the multi-core architecture. We also evaluate the performance of the Master-Slave, Synchronous Island and Asynchronous Island schemes on the many-core architecture. Table II shows the execution time for different sizes of population evolving to 50,000 generation using different schemes. In this experiment, since GTX580 GPU has 16 SMs, we use 16 blocks to implement 16 islands. As the size of population increases, we make the number of individuals in each island increase from 32 to 512. It is clear that the Asynchronous Island scheme is the fastest. We also see that the Asynchronous Island scheme gives the best



(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 (7064, 0.2% close to the optimum solution)

Figure 12. Comparison among Master-Slave, Synchronous Island and Asynchronous Island schemes on a multi-core architecture.

(a) Execution time with different values of PD (50,000 Generations)

(b) Accuracy of solutions with different values of PD (50,000 Generations)

(c) Execution time to reach a fixed solution with different values of PD (7064, 0.2% close to the optimum solution)

Figure 13. Execution time, solution quality and time to a fixed solution with different values of PD, for Mixed scheme on multi-core architecture

solution among these GA schemes from Table III. Finally, the execution time to reach the fixed solution of these GA schemes shown in Table IV implies that the Asynchronous Island scheme is the best considering solution quality and execution speed.

In our analysis, we mentioned that the new Fermi architecture can reduce the access to the global memory because it adds Level 2 cache between SMs and the global memory. In our experiments, we execute the Synchronous and Asynchronous Island schemes on both GTX580 and 9800GT GPUs. The GTX580 is the Fermi architecture, consisting of 16 SMs. On the other hand, the 9800GT is the previous architecture, consisting of 14 SMs.

Compared to the Asynchronous Island scheme, the Synchronous Island one need the extra access to the global memory to implement the synchronization among islands. In the experiment, each block maintains a island. We keep the number of blocks less than the number SMs. Therefore as the number of islands increases, the time that the population takes to evolve should be the same, however the Synchronous Island scheme needs to access more data in global memory to synchronize islands.

The difference of execution time between the Synchronous and Asynchronous Island schemes mainly represents the overhead of access to the global memory, which is depicted in Fig. 14. The x-axis is the number of islands, which is from 1 to 14. Each island has 256 individuals. So as the number of islands increases along the x-axis, the overhead of access to the global memory increases.

Fig. 14(a) shows the difference when a 9800GT GPU is used. As the number of islands increases, the difference between the Synchronous and Asynchronous Island schemes goes up. This means the overhead of access to the global memory becomes bigger and bigger. On the other hand, the difference is very small and stays unchanged when a GTX580 GPU is used, illustrated in Fig 14(b). In Fig. 14(c), it is obvious that the difference for the GTX580 GPU is much smaller than for the 9800GT. It is because GTX580 has Level 2 cache, so that most of overhead of access t the global memory is eliminated by the Level 2 cache; while 9800GT does not have Level 2 cache, so that as the overhead of access to the global memory increases, the execution time increases dramatically.

As we mentioned in our analysis, the Island scheme on the many-core architecture has some unique characteristics different from the typical Island scheme.

In Fig. 15, we evaluate the execution time and solution quality when the number of islands varies. The lines with

Table II
THE EXECUTION TIME OF DIFFERENT SCHEMES ON GPU WITH 50,000 GENERATIONS

Size of Population	512	1024	2048	4096	8192
Master-Slave Scheme	20.037 sec	39.227 sec	77.907 sec	154.248 sec	308.139 sec
Synchronous Island Scheme	7.433 sec	8.116 sec	10.29 sec	16.642	32.263 sec
Asynchronous Island Scheme	7.409 sec	8.086 sec	10.243 sec	16.602 sec	32.217 sec

Table III
THE SOLUTION QUALITY OF DIFFERENT SCHEMES ON GPU WITH 50,000 GENERATIONS

Size of Population	512	1024	2048	4096	8192
Master-Slave Scheme	7065.778293	7062.501734	7061.25624	7059.324437	7057.884367
Synchronous Island Scheme	7055.3655138	7054.8607042	7054.6954771	7054.1094868	7053.6442773
Asynchronous Island Scheme	7054.7561047	7054.3709347	7054.340533	7053.8333257	7053.406822

Table IV
EXECUTION TIME TO REACH THE FIXED ACCURACY (0.1%, 7056) OF DIFFERENT SCHEMES ON GPU

Size of Population	512	1024	2048	4096	8192
Master-Slave Scheme	∞	125.49 sec	221.834 sec	239.582 sec	476.919 sec
Synchronous Island Scheme	4.517 sec	4.819 sec	7.300 sec	12.938 sec	24.712 sec
Asynchronous Island Scheme	4.320 sec	4.532 sec	7,120 sec	12.424 sec	21.173 sec

square symbols represent the execution time; the lines with circle symbols represent the solutions. We keep the size of the population and the number of generations the same. According to the typical Island scheme, with the same size of population, more islands give lower solution quality.

Figs. 15(a) and 15(b) show the Island scheme on the multi-core and many-core architectures, respectively. The variation of execution time in both figures occurs because by the core utilization varies as the number of islands increases. We focus on the solution quality. The solution quality in Fig. 15(a) gets worse as the number of island increases, which perfectly obeys the typical theory of the Island scheme. However, the solution quality in Fig. 15(b) improves when the number of islands increases, which is totally opposite to the typical theory of the Island scheme. This proves our theoretical analysis that the Island scheme on a many-core architecture is actually a complicated Cellular scheme, not a pure Island scheme. This result also tells us that, although the Asynchronous Island scheme is the best on the many-core architecture, we can create more blocks to implement more islands to further improve its performance.

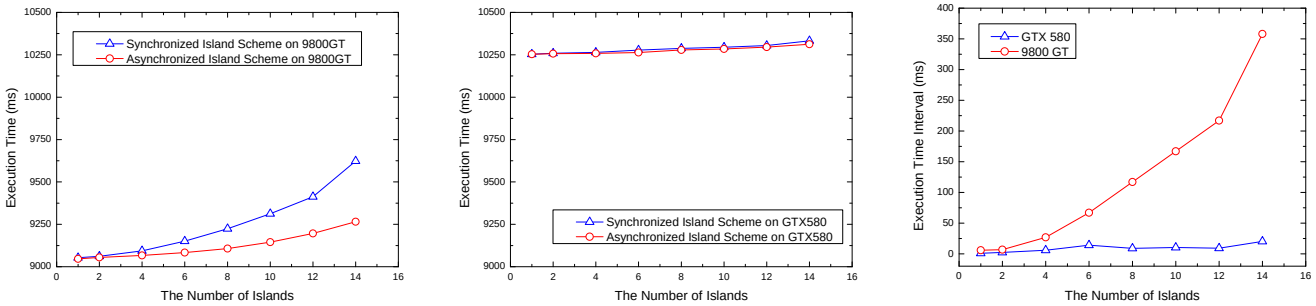
So now we know, from theoretical analysis and experiments, that the Mixed scheme works best on a multi-core machine while the Asynchronous Island scheme is best on a many-core device. We will choose the best scheme with the best optimization configuration on each architecture to calculate the speedup.

The conventional GPU speedup is calculated by comparing the execution time on the GPU with that on the CPU without considering solution quality. Many papers use the Serial scheme on the CPU to calculate the execution time. The serial scheme cannot take advantage of multi-core architecture. Therefore, the conventional calculation of speedup is not reasonable. From Table V, we find that the speedup can be over 10x if we use the conventional calculation of speedup, while the speedup is actually around 4x if we fully take advantage of multi-core architecture.

Besides, the conventional calculation of speedup ignores the solution quality. In our work, we use the execution time to reach a fixed solution quality as the metric to evaluate the performance of GAs on multi-core and many-core architecture, so that we can get the GPU speedup considering solution quality, which is illustrated in Table VI. From the table, we find that speedup from using a GPU is larger than without considering solution quality. It implies that the Asynchronous Island scheme on the many-core architecture has higher solution quality, because the requirements of GPU architecture turn the Aynchornized Island model into a Cellular model. Furthermore, as the accuracy becomes higher, the GPU speedup increases, which implies that we can use the GPU to gain more benefit of execution speed when we need more accurate solutions.

VI. CONCLUSION

In this paper, we analyze the performance of GAs on multi-core and many-core systems. We offer detailed theoretical analysis and use experiments to validate them. We also take the solution quality into account to find which GA scheme is the fastest for the multi-core and many-core systems. We find that the Mixed scheme can fully take advantage of the architecture to offer the best performance on a multi-core system, and that the Asynchronous Island scheme is the best scheme on many-core systems. We also point our that the global access to the global memory is effectively improved because of the Level 2 cache in the Fermi architecture GPU.

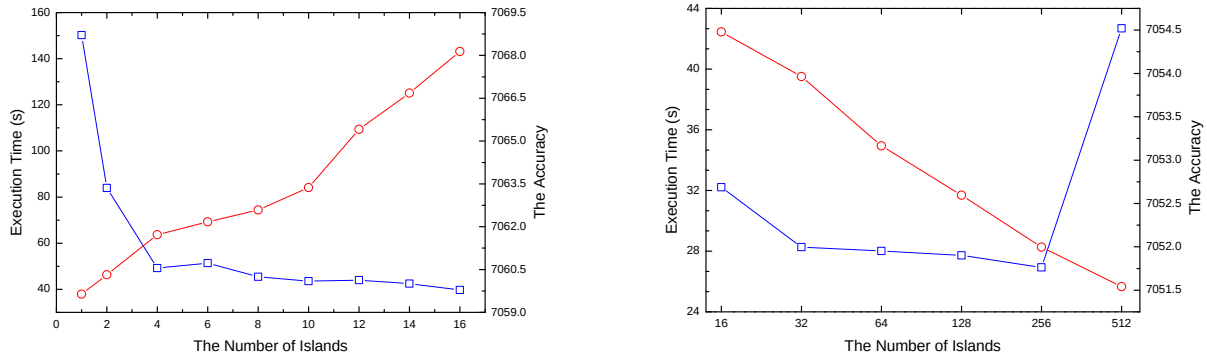


(a) Comparison of the execution time using Synchronous Island and Asynchronous Island schemes on Nvidia 9800GT GPU (50,000 Generations)

(b) Comparison of the execution time using Synchronous Island and Asynchronous Island schemes on Nvidia GTX580 GPU (50,000 Generations)

(c) Difference of execution time using Synchronous Island and Asynchronous Island schemes (50,000 Generations)

Figure 14. Effect of L2 Cache in the newest Fermi architecture, reducing the penalty of access to the global memory compared to the old architecture



(a) Execution time and solution accuracy with different number of islands on a multi-core architecture (50,000 Generations)

(b) Execution time and solution accuracy with different number of islands on the many-core architecture (50,000 Generations)

Figure 15. The Island scheme on the multi-core architecture exactly follows the classic Island scheme; while the Island scheme on the many-core architecture does not, proving that it actually is a complicated Cellular scheme.

Furthermore, we find that the Island scheme on a many-core architecture is not actually the Island model of GAs, but the Cellular model. If we follow the typical Island model theory, we should decrease the number of islands to improve the solution quality. However, our findings tell us that what we should do is exactly the opposite.

The speedup of the GPU compared to the CPU is also evaluated in our work. We offer a reasonable comparison method in which we choose the best implementations of GAs on both multi-core and many-core architectures to compare. Moreover, we take the solution quality into account. The speedup of GPUs compared to CPUs increases because the Island scheme on the many-core architecture can offer the better solution quality. Our findings are not only useful for GAs, but also for general optimizations on multi-core and many-core systems, as we do much deep architecture analysis.

ACKNOWLEDGMENT

This work was supported in part by NFSC (Grant No. 60811130528, 61003012), and the National 973 Basic Research Program (No. 2007CB310900) of China. Minyi Guo is the corresponding author.

REFERENCES

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55 – 69, 1996.

[2] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24.

[3] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 260–269.

[4] A. Beham, S. Winkler, S. Wagner, and M. Affenzeller, "A genetic programming approach to solve scheduling problems with parallel simulation," in *IEEE International Symposium*

Table V
MANY-CORE ARCHITECTURE SPEEDUP COMPARED TO THE MULTI-CORE ARCHITECTURE, NOT CONSIDERING SOLUTION QUALITY

Generation	10,000	20,000	40,000	80,000	100,000
Serial on CPU	27.494 sec	54.851 sec	109.860 sec	219.613 sec	274.904 sec
Mixed Scheme on CPU	10.798 sec	21.022 sec	41.406 sec	82.137 sec	102.673 sec
Asynchronous Island Scheme on GPU	2.745 sec	5.454 sec	10.832 sec	21.503 sec	26.868 sec
Speedup (Compared to Serial on CPU)	10.02	10.06	10.14	10.21	10.23
Speedup (Compared to Mixed Scheme on CPU)	3.93	3.85	3.82	3.82	3.82

Table VI
MANY-CORE ARCHITECTURE SPEEDUP COMPARED TO THE MULTI-CORE ARCHITECTURE CONSIDERING SOLUTION QUALITY

Accuracy	7064.0 (0.2%)	7060.0 (0.15%)	7056.0 (0.1%)
Serial on CPU	88.058 sec	122.072 sec	222.115 sec
Mixed Scheme on CPU	36.351 sec	47.370 sec	160.423 sec
Asynchronous Island Scheme on GPU	2.994 sec	3.822 sec	5.933 sec
Speedup	13.50	13.66	15.72

on Parallel and Distributed Processing, 2008. IPDPS 2008., april 2008, pp. 1–5.

[5] A. Markham and N. Trigoni, "Discrete gene regulatory networks dgrns: A novel approach to configuring sensor networks," in *Proceedings IEEE INFOCOM, 2010*, march 2010, pp. 1 –9.

[6] M. Lahiri and M. Cebrian, "The genetic algorithm as a general diffusion model for social networks," in *Proc. of the 24th AAAI Conference on Artificial Intelligence*, 2010, pp. 494–499.

[7] G. Renner and A. Ekart, "Genetic algorithms in computer aided design," *Computer-Aided Design*, vol. 35, no. 8, pp. 709–726, 2003.

[8] R. Arora, R. Tulshyan, and K. Deb, "Parallelization of binary and real-coded genetic algorithms on gpu using cuda," in *2010 IEEE Congress on Evolutionary Computation (CEC)*, july 2010, pp. 1 –8.

[9] Z. Konfrt, "Parallel genetic algorithms: Advances, computing trends, applications and perspectives," *Parallel and Distributed Processing Symposium, International*, vol. 7, p. 162b, 2004.

[10] T. Luong, N. Melab, and E. Talbi, "Gpu-based island model for evolutionary algorithms," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, ser. GECCO '10. New York, NY, USA: ACM, 2010, pp. 1089–1096.

[11] P. Vidal and E. Alba, "A multi-gpu implementation of a cellular genetic algorithm," in *2010 IEEE Congress on Evolutionary Computation (CEC)*, july 2010, pp. 1 –7.

[12] E. Cantú-Paz, "A survey of parallel genetic algorithms," *Calculateurs paralleles, reseaux et systems repartis*, vol. 10, no. 2, pp. 141–171, 1998.

[13] S. Xiao and W. Feng, "Inter-block gpu communication via fast barrier synchronization," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, pp. 1–12.

[14] Z. Michalewicz, "Genetic algorithmsnumerical optimizationand constraints," in *Proceedings of the 6th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 151–158.