Function Based Benchmarks to Abstract Parallel Hardware and Predict Efficient Code Partitioning

Ioannis Zgeras, Jürgen Brehm, Mark Akselrod

Leibniz University of Hannover
Institute of Systems Engineering
Appelstr. 4, 30167 Hannover, Germany
[zgeras | brehm]@sra.uni-hannover.de

Abstract. *To increase the performance of a program, developers have to parallelize their code due to trends in modern hardware development. Since the parallelization of source code is paired with additional programming effort, it is desirable to know if a parallelization would result in an advantage in performance before implementing it. This paper examines the use of benchmarks for estimating the performance gain looking at the parallelization of Population Based Algorithms (PBAs) like Genetic Algorithms (GAs) and Particle Swarm Optimization Algorithms (PSOs) to be implemented on multi- and many-cores. These benchmarks are named function based benchmarks due to their dependence on the PBAs' functions. Furthermore, the software-hardware mapping with the most performance gain is suggested.*

Keywords: Benchmark, Parallel Programming, Multi-Core Architecture, Genetic Algorithm, Particle Swarm Optimization, GPGPU (General Purpose Graphics Processing Unit)

1 Introduction

In the last years, microprocessor development has arrived to a point where smaller integrated circuits and higher clock speeds are no longer feasible due to physical phenomena. To still increase performance, multi- and many-core architectures have become more and more important where performance gain is not only achieved by higher frequencies but primarily through parallelization. As a result, shorter execution times of application programs depend on the capability of the software engineers to parallelize their programs and thus take advantage of the parallel structures of modern computer hardware. The use of massively parallel GPGPUs as accelerators has added another level of complexity to the hardware.

One of the main challenges at achieving the best possible speedup with parallel software is to identify the best partitioning of the executable code on the hardware. This requires knowledge whether a piece of code performs better running single threaded or hyper threaded on one core of the CPU, multi threaded

ISBN 978-3-8007-3492-4

on several cores of a multi-core CPU or multi threaded on a many-core machine such as a GPGPU. This decision is nor trivial neither automatically accomplishable. It depends on static and dynamic constraints like e.g. input size, branching factor of the code or complexity of computational operations. In this paper, we introduce an approach for identifying the best partitioning of the code with the support of function based benchmarks (FBBs) especially designed for a class of Population Based Algorithms (PBAs) (2.1) represented by Genetic Algorithms (GAs) (2.1) and Particle Swarm Optimization Algorithms (PSOs) (2.1). The benchmarks are designed by abstracting the characteristic functions of population based algorithms.

Benchmarks to evaluate the performance of hardware have a long history in computer architecture. Starting with Dhrystone [1] and Whetstone [2] to evaluate the number of instructions and the number of operations per time unit they have evolved since. Nowadays, they are used to evaluate the performance of GPUs [3] or the performance of whole computer systems. We are introducing FBBs designed to predict the performance of PBAs on different hardware architectures like single threaded CPUs, multi-core CPUs and GPGPUs.

The rest of the paper is organized as follows: Section 2 contains a short survey of related work. In Section 3, we describe the concept and implementation of the FBBs for PBAs. Section 4 presents evaluation results. Finally, Section 5 concludes with future research opportunities.

2 Related Work

Benchmarks are designed to evaluate the performance of hardware. One of the first known synthetic benchmarks is *Whetstone* [2]. The basic idea is to look at the operation mix of many typically scientific programs written in *Algol 60* [4] and use it as characteristic load to benchmark the hardware. One of the first benchmarks that measured the performance on parallel machines were the *NAS Parallel Benchmarks*[5] that were developed to determine performance in *Computational Fluid Dynamics* and are further developed by NASA [6]. In [7] the authors describe a benchmark suite for parallel computers focusing on multi-core architectures (OpenMP). In [8] a benchmark suite specialized on AMD GPUs is described. The authors in [9] describe in their paper an interesting approach of benchmarking complex heterogeneous architectures. However, they do cover a more general purpose scenario than the focus of this paper.

2.1 Population Based Algorithms

PBAs are nature inspired heuristics, all PBAs have similar structures. The main part is the population that consists of a set of solutions for a given problem. These solutions are called individuals, particles or, more general, agents. These agents execute in each iteration of the algorithm different kinds of operations to improve their solution. The quality of a solution is called *fitness*. The function or problem that the agents have to optimize (or solve) is called fitness function. Two representatives of PBAs the so called GAs and PSOs that are used for this paper are now described in more detail.

Genetic Algorithms: GAs [10][11] are heuristics based on the idea of natural selection. The population of GAs consists of individuals that represent a solution of a given problem where every solution consists of single chromosomes. E.g. the solution of an individual for an n-dimensional function would consist of n values of this function. The vector representing these values is the chromosome of this individual and every value of this vector is called *gene*. The outer iteration loop of GAs consists of three main operations - *Crossover*, *Mutation* and *Selection* - that are performed after a random initialization process until a break condition (e.g. *finding the minimum*) is achieved.

The crossover operation uses two individuals (parents) to generate new individuals (children) by crossing the solutions of the two parents. There are many different crossover operations, some popular examples can be found in [10]. In the next step, some chromosomes of the individuals are randomly changed, this operation is called mutation. Again, there are different methods for implementing mutation [10]. The individuals are now ranked based on their *fitness value* that indicates the quality of their solution. In the last step, the individuals are selected to become part of the population for the next iteration. Once again, there are many different ways to select the individuals [10]. There are many parallel implementations for GAs. Some of them can be found in [12][13][14].

Particle Swarm Optimization: PSO algorithms [15][16] have similarities to GAs but the approach is different. PSO is based on the natural behavior of birds. The population of a PSO algorithm is called *swarm* and the individuals are called *particles*. Every particle is represented by a position and a velocity where the position represents a solution of the problem and velocity the speed and direction this particle changes it's position.

In each iteration step, the particles try to approximate better solutions by detecting the best neighbor and updating their velocity and position value taking into account the velocity and position values of the best neighbor. The iteration loop is repeated until a break condition is fulfilled. Like for GAs, different parallel PSO algorithms can be found in the literature [17][18].

CUDA: As GPUs became more and more complex, their use was no longer limited to tasks belonging to graphical programming. Different programming languages were developed to facilitate the GPUs towards more general purpose processing. The company NVIDIA published CUDA (Compute Unified Device Architecture) as a programming environment for their GPUs. CUDA became a common programming language extending the C language by a small set of instructions, allowing the programmer to develop code running in parallel. The parallel code is executed on so-called CUDA kernels. More about CUDA can be found in [19][20].

3 Function Based Parallel Benchmarks for PBAs

This section describes the concept and implementation of FBBs for PBAs such as GAs and PSOs. Subsection 3.1 contains the concept of the FBBs, in 3.2 we give a more detailed insight of the single benchmark kernels while the parallelization techniques of the kernels are explained in Subsection 3.3. The runtime parameter dependent software-hardware mapping (partitioning) is shown in 3.4.

3.1 Concept

The developed FBBs cover a variety of implementations of population based algorithms without becoming too generic. PBAs like shown in 2.1 consist of a sequence of characteristic operations (mutation, selection, ...) whereby the impact of the single operations can vary depending on the concrete implementation. For that reason, we implemented the benchmarks using these sequences of operations generalizing them as much as necessary without loosing their characteristic properties.

Single operations can have multiple appearances, which cannot all be implemented in separate benchmarks. To face this problem, the operations are divided into complexity classes. We chose common complexity classes like ($\mathcal{O}(1)$, $\mathcal{O}(n)$, ...) and implemented an individual benchmark kernel for each class as described in 3.2. If necessary, our implementation is easily expandable for other complexity classes. In the evaluation (Section 4), we show that this kind of generalization is suitable for PBAs such as GAs and PSOs. We have used kernel functions from PBAs for the crossover and mutation benchmarks while for the selection benchmarks we have abstracted the fitness functions (kernels vs. abstract kernels). The reason behind is an infinite number of fitness functions. Abstraction is necessary to cover them without writing an infinite number of test cases. Furthermore, the selection part consumes in almost all cases the overwhelming part of the computation time, so it is more important to treat the selection part separately to achieve accurate results. The fitness computation is considered as part of the selection. In the following, we show the concept of the GA- and PSO-benchmarks.

Genetic Algorithms: The main parameters of GAs are:

– Population size → Number of individuals
– Individual size → Number of chromosomes per individual
– Iteration size → Number of iterations to be executed
– Fitness function → Kind of operations

The first two parameters are easily adjustable, if the population is implemented as an array. Every individual is represented as a field of a dynamic array (the population). In addition, the third parameter is also easily adjustable, it is the number of executions of the outer loop of the program. The fourth parameter is much more complicated to be described by a FBB. The single operations have to be analyzed and classified into complexity classes. In the following subsection (3.2), we show kernels with different complexity classes.

Particle Swarm Optimization: PSO algorithms have a similar structure as GAs:

– Size of the swarm → Number of particles

– Dimension size → Dimension of the solution space of the given problem
– Iteration size → Number of iterations to be executed
– Fitness function → Kind of operations

The first two parameters of the algorithms are comparable with GAs by replacing individuals by particles and chromosomes by dimension. The last two parameters are on par with the last two parameters of GAs.

The function based benchmarks for PBAs evaluate the most time consuming operations using the parameters described above. These operations are summarized into kernels. Each kernel evaluates the runtime behavior of parts of the PBAs. To get more accurate results, we do not just simulate the execution times of population based algorithms but implement kernels for the concrete operations on sequential and parallel hardware and order the results by execution times.

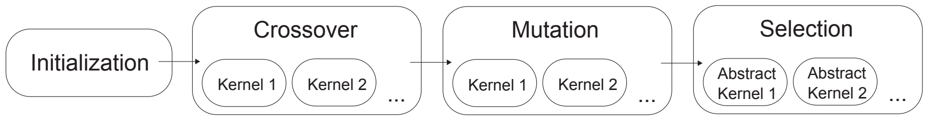

Fig. 1. Function Based Benchmark Model GA

Figure 1 shows a model of a simple GA benchmark. There are different kernels for each genetic operation. The reason for that are the many different variations of GAs and PBAs. We have tried to cover the most common ones and have held the extension interfaces quite open, so that new variations can easily be added. In the following, we give a short overview of the GA kernels used for evaluation.

3.2 Benchmark Kernels

Crossover: This benchmark kernel evaluates the behavior and performance of the crossover operations in GAs. Crossover, in this case, is always between two parent individuals and there are always two new individuals generated. We have considered the two mainly used crossover methods, *1-point-crossover* and *uniform crossover*[10]. 1-point-crossover is a simple method where the parents' chromosomes are cut at one point to produce new individuals (childs). The first child is generated by merging the first part of the first parent's chromosome and the second part of the second parent's chromosome, the second child is generated vice versa. This method has the time complexity of $\mathcal{O}(M)$ for one individual, $\mathcal{O}(N * M)$ (M - chromosome length; N - population size) for the whole population, respectively. Other methods (like 2-point-crossover) have similar structures and the same complexity and thus can be represented by this fairly simple kernel.

Another very important crossover implementation is uniform crossover. Here, for every position of the individuals (i.e. for every chromosome) a random number z is generated. If $(z = 0)$, the chromosome is filled with the value of the chromosome in this position from the first parent, if $(z! = 0)$ from the second parent. For the second individual, the procedure is used vice versa. This method generates $N*M$ random numbers and differs significantly from 1-point-crossover performance wise. More information about crossover methods can be found in [10].

Mutation: We have chosen three different mutation operations to cover most implementations. The first one is *mutation of one chromosome* with a time complexity of $\mathcal{O}(N)$, the second is *mutation of all chromosomes* with a complexity of $\mathcal{O}(N*M)$ and, additionally, generating of $\mathcal{O}(N*M)$ random values. The last mutation method is *mutation of a permutation* with a complexity of $\mathcal{O}(N*M)$ but with less random number generating steps as in mutation of all chromosomes. For more information about the detailed procedure of these methods please consider [10].

Selection: Selection consists of two operations. Firstly, the fitness of each individual has to be computed. Secondly, the individuals have to be chosen to become part of the new population in the next iteration step. We have implemented three common fitness functions to cover a larger set of functions. These functions are:

- $f_1(x) = \sum_{i=0}^{M-1} (x_i)^2$ (complexity $\mathcal{O}(M)$)
- $f_2(x) = \sum_{i=1}^{M-1} (\sum_{j=0}^{i} x_i)^2$ (complexity $\mathcal{O}(M^2)$)
- $f_3(x) = \sum_{i=0}^{M-1} \cos(x_i)^2$ (complexity $\mathcal{O}(M*\cos(1))$

The first two functions are popular and often used in benchmark functions for different problems [21]. The third function is chosen to evaluate the impact of trigonometric functions. Especially in GPGPU environments, functions with trigonometric operations often perform worse than functions without.

For the second part, selecting the individuals, we have implemented three of the most common methods. *Tournament selection* with a complexity of $\mathcal{O}(N)$, *roulette selection* with a complexity of $\mathcal{O}(N^2)$ and *rank selection* with a complexity $\mathcal{O}(N^2)$. Roulette selection and rank selection are similar whereby an additional sort operation in the roulette selection algorithm has to be performed. A detailed explanation of selection algorithms can be found in [10].

3.3 Parallelization

The described FBBs are designed to not only benchmark sequential machines but also multi- and many-core machines. Thus, the benchmarks have to be parallelized. Note that the investigated sequential program is not parallelized but the benchmarks. This procedure has to be done only once and can be used afterwards for benchmarking various different PBAs. Parallelization of sequential programs is difficult and not always obvious. The parallelization on multi-core machines is comparatively straight forward. Most parallelizable code segments in PBAs are loops that can easily be parallelized using e.g. OpenMP. The individuals are divided evenly on the threads which execute the operations in parallel.

Actualization of positions in PSOs (2.1) and sorting in rank election have to be considered separately. In position actualization the global best position may only be accessed by one particle simultaneously to prevent errors. OpenMP offers a simple way to do that by denoting a code segment as *critical*.

For sorting in rank election we have used a Mergesort implementation [22]. This implementation does not generate new threads if the subfields of the arrays that are generated in recursive sorting fall below a given threshold. Thereby, it is guaranteed that the recursive generation of new threads has no negative impact on system performance.

Parallelization on many-core machines is realized with CUDA (2.1). The whole population of the GA (the swarm in PSO algorithms, respectively) is represented by a grid where each block of the grid is arranged one dimensionally and represents the individuals (particles in PSO) one after another. Each chromosome (dimension) of the individuals is represented by a thread. If the number of chromosomes (dimension) becomes less than a given threshold, the whole individual is represented by a single thread. The benchmarks represent groups of operations with characteristic complexities.

3.4 Mapping

Software-hardware mapping is the last step in the benchmarking process. The benchmark programs need some information about the considered program as shown in Figure 2. Most of the values are trivial like population size or number of iterations. These values are normally static and known to the programmer. To accurately evaluate the complexity of operations the programmer has to find the correct complexity class. Alternatively, we are planning a function parser to help estimate the parameters:

– Population size	– Swarm size
– Individual size	– Number dimensions
– Complexity of genetic operations	– Complexity of fitness function
– Number iterations	– Number iterations

Fig. 2. Necessary parameters for GAs and PSOs

The corresponding benchmarks to the given parameters are executed several times to exclude fluctuations as a result from workload on the machine and the mean values are used. The benchmark kernels are executed single threaded, multi threaded and parallelized on the GPU using CUDA. Allocation time t_{alloc} of CUDA memory has to be considered separately. Furthermore, the time to copy the population from main memory into GPU memory and back ($t_{CPU} \rightarrow t_{GPU}$, $t_{GPU} \rightarrow t_{CPU}$) has also to be considered. Execution time of genetic algorithms is computed as:

$$t_{execGA} = t_{alloc} + Iterations * (t_{mean}(Crossover) + t_{mean}(Mutation) + t_{mean}(Selection)) + n * (t_{CPU} \rightarrow t_{GPU}) + m * (t_{GPU} \rightarrow t_{CPU}).$$

It is possible that data are copied more than once between CPU memory and GPU memory, this is represented by the values n and m. For PSO, the execution time is computed as follows:

$t_{execPSO} = t_{alloc} + Iterations * (t_{mean}(VelocityComputation) + t_{mean}(PositionComputation)) + n*(t_{CPU} \rightarrow t_{GPU}) + m*(t_{GPU} \rightarrow t_{CPU})$

If the GPU is not used, t_{alloc} is 0 and if the algorithm is a CPU/GPU implementation (some parts on the CPU and some on the GPU), the values $t_{CPU} \rightarrow t_{GPU}$, $t_{GPU} \rightarrow t_{CPU}$ have to be added in each iteration between the operations. This leads to 27 (selection,crossover,mutation (3)+data copy between operations(3)+CPU,multi-core,GPU(3) $\rightarrow 3*3*3$) different execution times for genetic algorithms and 9 different execution times for PSO. The implementation with the smallest execution time is suggested by the benchmark program as the best mapping.

4 Evaluation

To evaluate the accuracy of the benchmarks, we have implemented different PBAs (GAs and PSOs written in C++) and compared the execution times of the real algorithms with the execution times predicted by the benchmarks. The test programs were implemented in a single threaded, multi threaded (using OpenMP) and GPGPU version using different parameters. The results presented in this paper are restricted to GAs due to page limitations. The used parameters of the test GA are:

- *Crossover:* 1-point crossover (complexity $\mathcal{O}(N * M)$)
- *Mutation:* Mutation of all chromosomes (complexity $\mathcal{O}(N * M)$)
- *Selection:* Tournament selection (complexity $\mathcal{O}(N)$)
- *Fitness function 1:* $f(c_1, ..., c_M) = |c_1 - c_2| + |c_2 - c_3| + ... + |c_{M-1} - c_M| + |c_M - c_1|$
- *Fitness function 2:* $f(c_1, ..., c_M) = \sum_{i=1}^{M}\sum_{i=1}^{M} c_i$

The first fitness function has linear complexity while the second has a quadratic complexity, both are represented by the benchmarks of their related class of complexity

The test bench consists of a AMD Phenom X6 1090T (6 cores) with a NVIDIA GTX 580 (driver version 295.41, SKD version 4.2.9) running Ubuntu 11.04 (32bit). The gcc compiler is version 4.5.2-8ubuntu4. The evaluation results consist of the measured execution times for different iteration numbers. Furthermore, we have evaluated different population (N) and chromosome (M) sizes denoted as ($N * M$).

4.1 Genetic Algorithm - Linear Fitness Function:

Figure 3 shows the predicted execution times (left) and the real execution times (right) of the test program. For the multi-thread implementations there are two curves (multi-thread (min) and multi-thread (max)) which are caused by background processes of the operating system affecting the programs. The single threaded version is not affected by these processes because one core is reserved for the program execution exclusively and the processes run on the other cores, similar with the GPU implementation. The results of the benchmarks for 2000 iterations show a slightly benefit of the multi-core implementation over the GPU implementation. The single threaded implementation only performs well for small populations ($N * M$). The speedup of the GPU implementation is about 5.2 to 5.8 while the speedup of the multi-core (max) implementation is about 3.0 to 4.8 and multi-core (min) even 4.5 to 6.5. The speedup of the OpenMP implementation exceeds the estimated value of 6 (6 cores), that means the implementation achieves a super scalar speedup. The reason for that is the cache hierarchy of the Phenom CPU which consists of L1-,L2- and L3-Caches. L1 and L2 are located on the single cores while L3 is shared. By using all 6 cores more faster L2 cache is available. That leads to a super-linear speedup. Different from the GPU implementation, the multi-core implementations achieve also reasonable speedup with small populations, because of the relatively high overhead of the GPU implementation and the small execution times using small populations. Whereas the populations size increases the execution times increase also and the overhead is negligible. Table 1 summarizes the speedup comparison results whereby the *Benchmarked speedup* the predicted speedup by our benchmarks shows and *Real speedup* the achieved speedup by the test application.

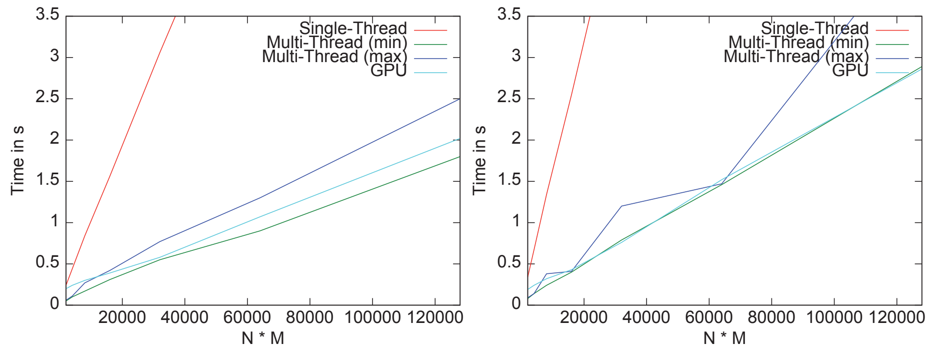

Fig. 3. Runtime of benchmarks and test program - 2000 iterations

Implementation	Benchmarked speedup	Real speedup
Multi-Thread (min)	4.5 to 6.5	6.5 to 7.2
Multi-Thread (max)	3.0 to 4.8	4.0 to 7.2
GPU	5.2 to 5.8	6.5 to 7.2

Table 1. Speedup comparison 2000 iterations - linear fitness function

We have also investigated GAs with greater iteration sizes. On Table 2 the results of 8000 iterations are shown. Again, the expected values differ slightly from the real measured values but obviously show that the GPU implementation performs best as proven by the test program.

Implementation	Benchmarked speedup	Real speedup
Multi-Thread (min)	5.5 to 6.6	6.8 to 7.1
Multi-Thread (max)	4.9 to 5.2	4.5 to 6.5
GPU	6.8 to 7.2	7.5 to 8.2

Table 2. Speedup comparison 8000 iterations - linear fitness function

The shown results imply that a mixed implementation using different hardware does not perform better than a multi-core or GPU only implementation. Single threaded implementations do only outperform multi-and many-core implementations in small population sizes while multi-and many-core implementations are too close in terms of performance that the copy overhead between main memory and GPU memory is too big compared to the benefit of a hybrid CPU-GPU implementation.

4.2 Genetic Algorithm - Quadratic Fitness Function:

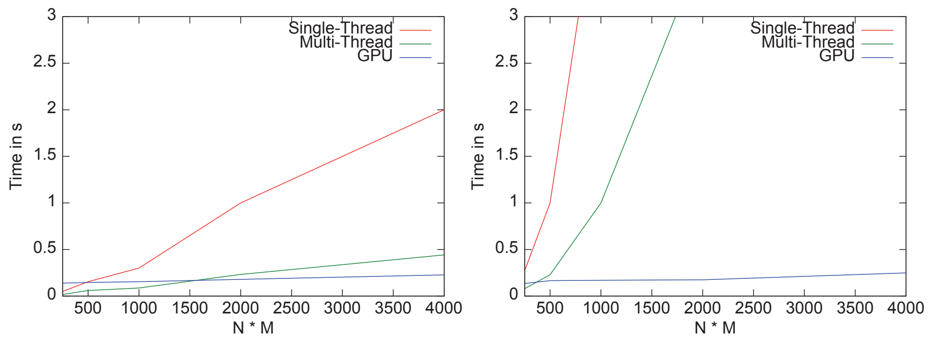

Fig. 4. Runtime of benchmarks and test program - quadratic GA - 1000 iterations

In the evaluation graphs we have relinquished to separate multi-thread (min) and multi-thread (max) results because of the large performance gap between multi- and many-core implementation results. In Figure 4, one drawback of the benchmark method is shown. The behavior of the test program (right graph) differs from the behavior of the benchmark program (left graph) in terms of absolute values, while the progression of the speedup curves are similar. The reason for that is the fitness function. In the benchmarks the fitness function executes $\frac{M*(M+1)}{2}$ operations but the fitness function in the test program executes M^2 operations. Both functions are within the complexity class $\mathcal{O}(M^2)$ but it makes a huge difference which to choose in real world scenarios.

The multi-core implementation does not reach speedup values as in the GA with a linear function (Figure 5). Again, the reason is the architecture of the Phenom CPU. When using just one core, the CPU clocks the core with 500 MHz more than when using all cores. As the quadratic function needs more computational power, a higher clocked core has a huge impact on the performance of the single core implementation.

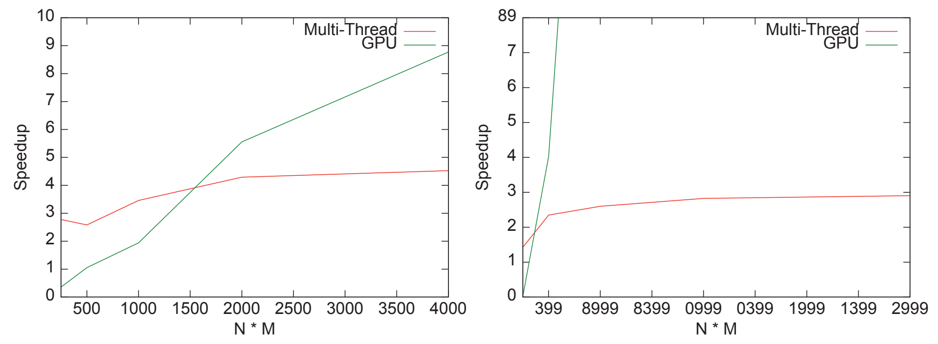

Fig. 5. Speedup of benchmarks and test program - quadratic GA - 1000 iterations

Overall, FBBs are well suited to predict correct partitioning of PBAs. The benchmarks have predicted that it is worthwhile to parallelize the GA with linear function at any point except very small population sizes regardless of the mapping. This assumption has been proven by the results with our test program. Furthermore, the benchmarks suggest a mapping on the GPU if the iteration size increases (over 8000 iterations), this assumption has been proven by the test program as well. For the GA with quadratic function, the benchmarks recommend a mapping of the GA on the GPU if the population size exceed 1500 while the test results show that a mapping on the GPU is worthwhile if the population size exceeds 450. While in this case the accuracy of the prediction is not as high as in the first evaluation scenario, it is still a helpful result to map the program on the best performing hardware.

There are some inaccuracies by using complexity classes as the main classification method looking at the total execution times. Nevertheless, the progression of the speedup curves is predicted correctly which leads to a good partitioning of the code.

5 Conclusion and Future Work

In this paper, we have presented an approach for speedup estimation and partitioning of PBAs on multi-and many-core architectures using FBBs. The structure of the benchmarks are based on the structure of the PBAs by implementing characteristic operations. The different operations are divided into complexity classes to enable the FBBs to simulate their behavior on parallel machines. First evaluation results are promising and show correct trends of achieved speedups. The evaluation results have also shown some drawbacks by using complexity classes for classification. This is apparent from the fact that differences in operations with equal complexity classes can have a large impact on execution times. We want to address this problem by a more precise classification of the operations. Furthermore, we are working on methods to extend the benchmarks by more features - e.g. more extensively analysis of memory operations. Also, we are working on more general kernels for the other functions (e.g. selection, mutation, ...). Finally, we plan to extend the FBBs to supercomputers with more than one compute node using MPI.

References

1. Weicker, R.P.: Dhrystone: a synthetic systems programming benchmark. Commun. ACM **27**(10) (October 1984) 1013–1030
2. H. J. Curnow, B. A. Wichmann, T.S.: A synthetic benchmark. In: The Computer Journal. (1973) 43–49
3. Papadopoulou, Sadooghi-Alvandi, Wong, H.: Micro-benchmarking the gt200 gpu. Processing (2009) 235–246
4. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Report on the algorithmic language algol 60. Numerische Mathematik **2** (1960) 106–136 10.1007/BF01386216.
5. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The nas parallel benchmarkssummary and preliminary results. In: Proceedings of the 1991 ACM/IEEE conference on Supercomputing. Supercomputing '91, New York, NY, USA, ACM (1991) 158–165
6. NASA, A.S.D.: Nas parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html (2012)
7. Aslot, V., Domeika, M., Eigenmann, R., Gaertner, G., Jones, W., Parady, B.: Specomp: A new benchmark suite for measuring parallel computer performance. In Eigenmann, R., Voss, M., eds.: OpenMP Shared Memory Parallel Programming. Volume 2104 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2001) 1–10
8. Taylor, R., Li, X.: A micro-benchmark suite for amd gpus. In: Parallel Processing Workshops (ICPPW), 2010 39th International Conference on. (sept. 2010) 387 –396
9. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC), IEEE (October 2009)
10. Mitchell, M.: An Introduction to Genetic Algorithms (Complex Adaptive Systems). Third printing edn. A Bradford Book (February 1998)
11. Goldberg, D.E., Holland, J.H.: Genetic algorithms and machine learning. Machine Learning **3** (1988) 95–99 10.1023/A:1022602019183.
12. Luong, T.V., Melab, N., Talbi, E.G.: Gpu-based island model for evolutionary algorithms. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10, New York, New York, USA, ACM Press (July 2010) 1089
13. Luong, T.V., Melab, N., Talbi, E.G.: Parallel hybrid evolutionary algorithms on gpu. IEEE Congress on Evolutionary Computation CEC (2010)
14. Parrilla, M., Ar, J., Dormido-canto, S.: Parallel evolutionary computation: Application of an ea to controller design
15. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Neural Networks, 1995. Proceedings., IEEE International Conference on. Volume 4. (nov/dec 1995) 1942 –1948 vol.4
16. Poli, R., Kennedy, J., Blackwell, T.: Particle swarm optimization. Swarm Intelligence **1** (2007) 33–57 10.1007/s11721-007-0002-0.

17. Zhan, Z.h., Zhang, J.: An parallel particle swarm optimization approach for multi-objective optimization problems. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10, New York, New York, USA, ACM Press (July 2010) 81
18. Zhou, Y., Tan, Y.: Gpu-based parallel particle swarm optimization. In: Evolutionary Computation, 2009. CEC '09. IEEE Congress on. (may 2009) 1493 –1500
19. NVIDIA: NVIDIA CUDA Compute Unified Device Architecture - Programming Guide. (2007)
20. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: Nvidia tesla: A unified graphics and computing architecture. In: IEEE Micro. Volume 28., Los Alamitos, CA, USA, IEEE Computer Society Press (2008) 39–55
21. Vesterstrom, J., Thomsen, R.: A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In: Evolutionary Computation, 2004. CEC2004. Congress on. Volume 2. (june 2004) 1980 – 1987 Vol.2
22. Pieloth, C.: Paralleles mergesort mit hilfe von openmp. Master's thesis (2010)