Evaluating the Cell Broadband Engine as a Platform to Run Estimation of Distribution Algorithms

Carlos Perez-Miguel, Jose Miguel-Alonso and Alexander Mendiburu
Department of Computer Architecture and Technology,
The University of the Basque Country
P. Manuel de Lardizabal, 1 (20018) Donostia-San Sebastian, Spain
{cperez027, j.miguel, alexander.mendiburu}@ehu.es

ABSTRACT

Current consumer-grade computers and game devices incorporate very powerful processors that can be used to accelerate many classes of scientific codes. However, programming multi-core chips, hybrid multi-processors or graphical processing units is not an easy task for those programmers that deal mainly with sequential codes. In this paper, we explore the ability of the Cell Broadband Engine to run a particular Estimation of Distribution Algorithm. From an initial sequential version, we develop a multi-threaded one that is afterwards reworked to run on a Cell. The multi-threaded version is capable of efficiently use current multi-core chips, such as those used in desktop PCs. However, the efficiency of the Cell version is very low. We analyze the causes of these discouraging results, and provide some clues about the class of problems that could be efficiently ported to the Cell.

Categories and Subject Descriptors

J.2 [**Physical Sciences and Engineering**]: Mathematics and statistics; G.4 [**Mathematical Software**]: Parallel and vector implementations

General Terms

Algorithms, Design, Measurement, Performance

Keywords

Cell Broadband Engine, Estimation of Distribution Algorithms, Parallel programming

1. INTRODUCTION

A current hybrid, on-chip multiprocessor, such as the Cell Broadband Engine, promises an enormous computing power (up to 200 GFlops) for a budget. We can find these chips on game consoles and other consumer devices. At the same time, the computational power available from desktop PCs continues growing at an incredible pace, and we should not forget the number-crunching abilities of graphical processing units. Users that run scientific codes are willing to take advantage of this power, but this is not an easy task. Programs have to be reworked in order to take advantage of parallel and hybrid processors. These machines require sophisticated programming models that are not easy for the casual programmer. Parallelism, a challenge by itself, is not the only issue. Unfamiliar memory models, limited instruction sets, explicit communications, etc. combine to make really hard the effective exploitation of theoretically powerful machines.

Still, users have powerful desktop computers, or departmental servers, with large amounts of memory that can be used to solve problems of growing complexity. This fact has encouraged the design and implementation of non-trivial algorithms to solve different kinds of complex optimization problems. Some of these problems can be solved via an exhaustive search over the solution space, but in most cases this brute force approach is unaffordable. In these situations, heuristic methods (deterministic or non deterministic) are often used, which search inside the space of promising solutions. Some heuristic approaches are specifically designed to find good solutions for a particular problem, but others are presented as a general framework adaptable to many different situations. Among this second group (general designs), there is a family of algorithms that has been widely used in the last decades: Evolutionary Algorithms (EAs). This family comprises, as main paradigms, Genetic Algorithms (GAs) [11, 13], Evolution Strategies [26], Evolutionary Programming [9] and Genetic Programming [16].

Even though processing speeds grow fast, the requirements of this class of algorithms grow even faster. No matter the computing power available, we can always find a harder problem that cannot run in our machines, or can do so but takes too long to run.

Complex algorithms could run much faster in current machines if the implementations are adapted to the system's characteristics. That is a fact. Programming to take full advantage of a parallel, hybrid, machine is a difficult task. That is another fact. In this paper we report our experiences porting a particular Estimation of Distribution Algorithm, which belongs to the class of Evolutionary Algorithms, from an initial sequential version to a parallelized version capable of running on a multi-core, symmetric system (such as a Quad-Core Intel processor). The parallel version was, afterwards, reworked to run on a multi-core, hybrid system (the Cell Broadband Engine). The degree of success, in terms

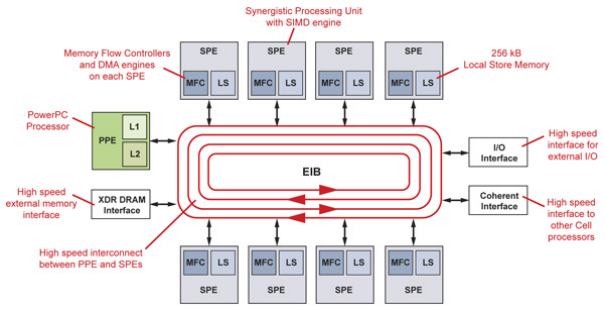

Figure 1: Cell Broadband Engine Architecture.

of obtained acceleration, of each of the approaches has been very different. We discuss the process, and the causes of this disparity of results. We also provide some clues about the kind of program modifications that would be required to take advantage of a Cell-based system. For the interested reader, in the literature we can find several references about accelerating Evolutionary Algorithms on the Cell [4, 29, 2] and on graphics processing units (GPUs) [10, 30].

The rest of the paper is organized as follows. Section 2 discusses the architecture of the Cell Broadband Engine, with special emphasis on those characteristics visible to the programmer. Section 3 summarizes the main characteristics of Evolutionary Algorithms and, in particular, the family of Estimation of Distribution Algorithms. Section 4 introduces the parallel models commonly used in EAs and analyzes a sequential implementation of the particular EDA (Univariate Marginal Distribution Algorithm, UMDA), that will be ported to a multi-core system. Section 5 is devoted to explain the porting of the parallel UMDA to the Cell, with an analysis of the obtained results and a discussion of the suitability of this platform for this class of algorithms. We end with some conclusions in Section 6.

2. CELL BROADBAND ENGINE

The Cell is a microprocessor system that integrates, into a single chip, a Power-based processor (Power Processing Element,PPE), eight vector co-processors (Synergistic Processing Elements, SPEs), a memory interface, input/output interfaces and a high-speed ring that acts as the interconnection fabric for the remaining elements [15] (see Figure 1, extracted from [27]). A programmer that wants to take full advantage of a Cell has to deal with two different instruction sets: one for the PPE and another one for the SPEs. This usually means using two compilers, and dealing with different strategies to optimize code running in different processors. To mention just an example, the PPE can deal with vector operations to accelerate parts of the program, but the SPEs must work with vectors – its efficiency with scalars is not brilliant.

Another peculiarity of the Cell is its memory organization. From a programmer's point of view, the PPE has full, direct access to the system's main memory. However, the SPEs have direct access only to a very limited sized (256 KB) local store. All the data processed by an SPE has to be previously transferred to its local store, and the resulting data, if required by the PPE or another SPE, hast to be explicitly transferred too. To that purpose, each SPE has a companion Memory Flow Controller (MFC) that can take care of this transfer. A good programmer can manage to make an SPE and its MFC work at the same time, processing pieces of data while transferring new ones. A careless programmer may try to simultaneously move too much data trough the interconnection fabric, which would become a bottleneck because of limited capacity.

Challenges for the programmer are, therefore, manifold: different instruction sets; real necessity of working with vector instructions; limited memory size; explicit transfer of data between different memory blocks, etc. Adaptation of an application to this architecture is not trivial: a few applications may have a natural mapping, but most require exhaustive reworking. A common model for organizing Cell applications, but by no means the only one, is to use the PPE as a main processor running most of the application's logic, using the SPEs as acceleration co-processors [14]. The most compute-intensive sections of the originalPPE-only code are identified, and reworked to make them run in parallel in the available SPEs. As just mentioned, this is a complex task that requires careful organization of data structures, data movement, synchronization, vectorization, etc.

Regarding hardware platforms, IBM sells Cell-based systems for use as general-purpose systems, but the most popular, consumer-available platform to get acquaintance with this processor is Sony's PlayStation 3 game console. The PS3 can be easily converted in a GNU/Linux "computer" with all the necessary toolkits to develop and run Cell applications, by means of any of the several available Linux distributions. The main limitation of this platform is that only six SPEs are available: Sony guarantees just seven working SPEs (to increase manufacture yield), and one is always reserved for the operating system. In this work we use a PS3 running Fixstars' Yellow Dog Linux [1].

3. EVOLUTIONARY ALGORITHMS

The main characteristic of Evolutionary Algorithms is that they use techniques inspired by the natural evolution of the species. In nature, species change across time; individuals evolve, adapting to the characteristics of the environment. This evolution leads to individuals with better characteristics. This idea can be translated to the world of computation, using similar concepts:

Individual: Represents a possible solution for the problem to be solved. Each individual has a set of characteristics (genes) and a fitness value (based on its genes) that denotes the quality of the solution it represents.

Population: In order to look for the best solution, a group of individuals is managed. An initial population is created randomly, and will change across time, evolving towards members with different (and supposedly better) characteristics.

Breeding: Several operators can be used to emulate the breeding process present in nature: mixing different individuals (crossover) or changing a particular one (mutation). These operators are used to obtain new

Pseudo-code for the EDA framework.

Step 1. Generate the first population D_0 of M individuals and evaluate all of them
Step 2. **Repeat** at each generation l until a stopping criterion is fulfilled
Step 3. Select N individuals (D_l^{Se}) from the D_l population following a selection method
Step 4. Induce from D_l^{Se} an n (size of the individual) dimensional probability model that shows the interdependencies between variables
Step 5. Generate a new population D_{l+1} of M individuals based on the sampling of the probability distribution $p_l(\boldsymbol{x})$ learnt in the previous step

Figure 2: Common outline for all Estimation of Distribution Algorithms (EDAs).

individuals, expected to be better than the previous ones.

In the last two decades, Genetic Algorithms have been widely used to solve different problems, improving in many cases the results obtained by previous approaches. However, GAs require a large number of parameters (for example, those that control the creation of new individuals) that need to be correctly tuned in order to obtain good results. Generally, only experienced users can do this correctly and, moreover, the task of selecting the best choice of values for all these parameters has been suggested to constitute itself an optimization problem [12]. In addition, GAs show a poor performance in some problems (deceptive and separable problems) in which the existing crossover and mutation operators do not guarantee that better individuals will be obtained changing or combining existing ones.

Some authors [13] have pointed out that making use of the relations between genes can be useful to drive a more "intelligent" search through the solution space. This concept, together with the limitations of GAs, motivated the creation of a new type of algorithms grouped under the name of Estimation of Distribution Algorithms (EDAs).

EDAs were introduced in the field of Evolutionary Computation in [21], although similar approaches can be previously found in [31]. In EDAs there are neither crossover nor mutation operators. Instead, the new population of individuals is sampled from a probability distribution, which is estimated from a database that contains the selected individuals from the current generation. Thus, the interrelations between the different variables that represent the individuals are explicitly expressed through the joint probability distribution associated with the individuals selected at each generation. A common pseudo-code for all EDAs is described in Fig. 2.

Steps 3, 4 and 5 will be repeated until a certain stop criterion is met (e.g., a maximum number of generations, a homogeneous population or no improvement after a specified number of generations). The probabilistic model learnt at step 4 has a significant influence on the behavior of the EDA from the point of view of complexity and performance.

For detailed information about the characteristics of EDAs, and the algorithms that form part of this family, see [17, 24, 18, 25].

4. PARALLEL MODELS FOR EAS

Evolutionary Algorithms require, in general, long execution times. For this reason, researchers often apply parallel techniques to reduce running times. These techniques are also useful to improve accuracy or to manage larger problems with the same time budget (see [5, 13]).

There are two basic approaches to parallelize EAs: parallelization of program loops, and division of the population into several independent subpopulations.

In the second approach, known as islands model, the single population used in sequential algorithms is split into several sub-populations (islands). These islands evolve independently, and exchange information about their best individuals with a predefined frequency. These models are suitable to distributed systems, because each island can be mapped onto a separate processor, and the amount of communications required between islands is not very large. Several works exist about this subject (see for example [28, 3]). In [8] authors proposed several island-based topologies and made several experiments using different models of migration between islands for EDAs over a discrete domain. Experiments proved that these island-based EDAs obtain better solutions than single-population sequential EDAs, because of the possibility of searching concurrently over different points of the solution space.

A more conservative approach starts with a sequential algorithm like that described in Figure 2, parallelizing parts of it in order to speed-up the execution time but without changing the semantics of the algorithm. There exist different proposals for GAs [7] or EDAs [22, 23, 19]. The most time-consuming portions of the code are identified and rewritten to take advantage of a parallel computer. Among the techniques to parallelize the code, or portions of it, the Master-Worker model is a popular one: a Master task runs the program, and delegates CPU-intensive parts to a collection of Worker tasks. Note that, as program semantics is not changed, obtained solutions are not improved.

The selection of the parallelization paradigm has to be done taking into account the characteristics of the available computing platform. In this work we will not work with a cluster of computers, but with on-chip multiprocessors – in particular, with a Cell system. The limited memory of the Cell's SPEs does not allow us to run a complete EDA (an island) in each SPE. Therefore, we have to opt for the conservative approach. We will use a Master-Worker approach to parallelize our code. When porting it to the Cell, the Master task will run on the PPE and the Workers on the SPEs.

4.1 Parallel implementation of UMDA

In this work we focus on the Univariate Marginal Distribution Algorithm (UMDA), a simple EDA introduced in [20]. It follows the general scheme of EDAs shown in Figure 2.

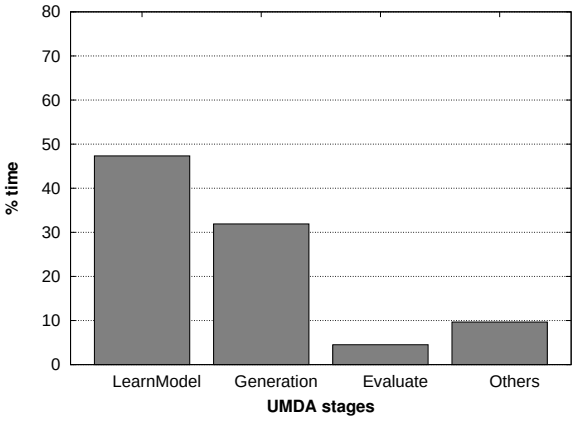

Figure 3: Principal functions in UMDA.

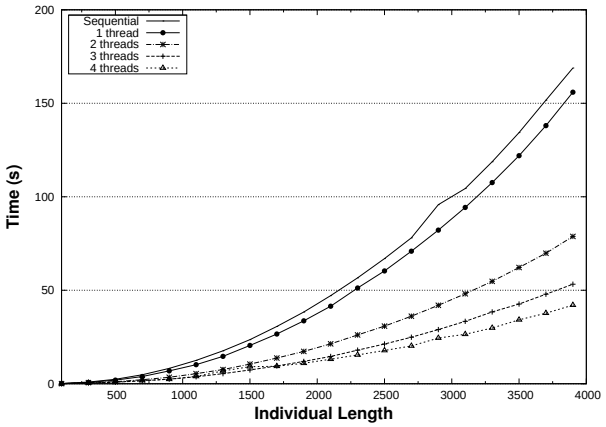

Figure 4: Execution times for sequential and threaded version.

As explained in the previous section, the complexity of the probabilistic model learnt at each step has a significant influence in terms of performance and complexity. UMDA uses the simplest way to estimate the joint probability distribution, considering that all the variables are independent. This model can be expressed as a product of the probabilities of the variables:

$$p_l(\boldsymbol{x}) = p(\boldsymbol{x}|D_{l-1}^{Se}) = \prod_{i=1}^{n} p_l(x_i) \tag{1}$$

where each univariate marginal distribution is estimated from marginal frequencies:

$$p_l(x_i) = \frac{\sum_{j=1}^{N} \delta_j(X_i = x_i|D_{l-1}^{Se})}{N} \tag{2}$$

being

$$\delta_j(X_i = x_i|D_{l-1}^{Se}) = \begin{cases} 1 & \text{if in the } j^{th} \text{ case of } D_{l-1}^{Se},\ X_i = x_i \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The parallel approach has been done starting from a sequential version made in C++. Before designing the parallel version for the Cell Broadband Engine, we considered interesting, for comparison purposes, to design a parallel version based on the Posix Threads (pthreads) library [6], that could be executed on a multi-core personal computer.

When facing the parallelization of a sequential algorithm, it is mandatory to obtain execution statistics, identifying which are the most expensive parts in terms of computation time. For this purpose, we executed UMDA to solve *OneMax*, a well-known problem, that has a very simple objective function. This problem consists of maximizing:

$$OneMax(\boldsymbol{x}) = \sum_{i=1}^{n} x_i$$

where $x_i \in \{0, 1\}$.

That is, the best solution is reached when all the variables of the individual take value 1. The size of the individual is 1,000, the population size (POP_SIZE) is 2,500 individuals, and the algorithm was stopped at the 50th generation. The rest of the parameters are the same for all the experiments: To learn the model, truncation was used, selecting the best half of the population. At each generation, POP_SIZE new individuals are created, mixed with the current population, and the best POP_SIZE individuals will be selected to constitute the next generation.

As can be seen in Figure 3, the most time-consuming steps are the first three. That is, learning the model and, creating and evaluating the new population. It must be taken into account that evaluation is problem dependent and few problems are as simple as the *OneMax*.

Our parallel approach follows the well-known Manager-Worker scheme, where the UMDA algorithm will be executed by the main thread (Manager), and when necessary, it will ask the Workers for help. In particular, workers will collaborate in these phases:

- Learning the probabilistic model: As introduced previously, UMDA uses a very simple probabilistic model, that assumes that there is no relation between the variables. Therefore, once the population has been selected, the Manager will ask the workers to obtain the marginal frequencies for a subset of the selected population. Once each Worker has finished, the Manager creates the main model based on the partial values.

- Sampling and Evaluation: These two steps usually come together. That is, based on the model learnt in the previous step, new individuals will be created (and evaluated). Again, the Manager will ask the workers to create (and evaluate) a subset of individuals. The number of individuals to be managed by each worker can be established statically, or assigned dynamically using an on-demand scheme. That is, when the evaluation of the individuals takes always the same computing time, a static assignation can be done. However, if the time required to evaluate the individual depends on the values it takes, the on-demand scheme is preferable. In these experiments, a static distribution was

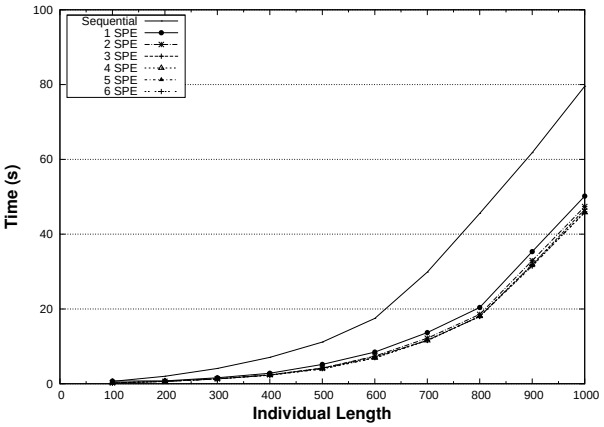

Figure 5: Execution times for sequential and threaded version in CBE.

used (evaluation time is constant for $OneMax$).

In order to test the performance of this multi-threaded version, we completed several experiments on an Intel Xeon Quad-Core computer. Different individual sizes were used (ranging from 10 to 4,000 variables), using a population size of $2,5L$, being L the number of variables of the individual. The number of threads was also changed from 1 to 4. UMDA was stopped after computing 50 generations. The results of these experiments are shown in Figure 4.

According to the results, it can be seen that the multi-threaded version has an adequate behavior from the point of view of scalability. Therefore, the second step of this work would be to test the suitability of the Cell for the execution of EDAs.

5. PORTING UMDA TO THE CELL

Once we have the multi-threaded parallel version of UMDA running on the Quad-Core Xeon symmetric multiprocessor, we are ready to port it to the Cell. To do so, we have to rewrite the code to adapt it to the characteristics of this processor. First, we have to deal with heterogeneity: the Master thread will run on the PPE, and the worker threads will be separate programs that will run on the SPEs. In the Quad-Core we took advantage of a large, shared memory space, which simplifies communication among tasks and avoided explicit data movements. This memory model is not valid for the Cell. Portions of code have to be introduced in order to explicitly move data, using DMA, from the main memory to the SPE's local store and vice-versa. This process is costly not only in terms of programming difficulty, but also in terms of application running times.

In the previous section we identified those parts to be delegated to the workers: learning the probability model, and sampling and evaluation. For the Cell, we will make the workers deal only with the latter.

Regarding the learning the probability model phase, in the multi-thread version the manager asks the workers to obtain the marginal probabilities for a subset of the selected individuals. In the case of the Cell, in order to complete this step, it would be necessary to previously send a subset of selected individuals to each of the SPEs. Taking into account that the probability model learnt by UMDA is very simple, it is not worth to complete this step following a parallel scheme, because the time required by the communication (send individuals) is notably higher than the time needed to compute the marginal probabilities.

Flags	Size (bytes)	Exec. Time (s)
-Os	120,028	247.999
-O0	142,092	252.366
-O1	122,924	247.665
-O2	122,116	248.097
-O3	125,724	248.030

Table 1: Code size and execution times for different compilation flags. Individual size = 3,000. Six SPEs.

For the sampling and evaluation phase, we repeat for the Cell the same scheme used in the multi-thread code. That is, the manager will ask the workers to create and evaluate a given number of individuals. The workers (executed in the SPEs), will use a double-buffer technique to create, evaluate, and send the individuals to the PPE using DMA. This technique consists on working on an individual while, in parallel, another one is being transferred to the manager.

In addition to the general design aspects explained for the two phases of the algorithm, we consider interesting to point out an important aspect when adapting a code for the Cell Broadband Engine. Due to the size restrictions in the SPEs (256KB of memory, for code and data), it is recommendable to reduce the size of the code as much as possible. In the following lines, we present some ideas that can help to obtain a small-sized code:

- Optimization flags: The optimization level affects the size and execution time of the program. In Table 1 we show different optimization options together with the sizes and execution times for our UMDA implementation. In this particular case all options result in very similar execution times but we can notice significant differences in the size of the code that runs on the SPEs. Therefore when preparing code to the SPEs it is important to choose the right speed/size relation.

- C++ exception handling: this mechanism increases in about a 10% the code's size. We can avoid this system (for well debugged code) using the flag *-fno-exceptions*.

- Libraries: Is is also important to be careful with the libraries linked with the SPE code. As there is not dynamic linking in the SPE, all the libraries must be linked statically, increasing the final size of the executable. For example, using the C++ Standard Template Library to manage sets of individuals in a vector implementing a population object makes code notably larger in around 70 KB.

In summary, leaving behind these suggestions can result in a code that does not fit in the SPEs' Local Stores, or a code

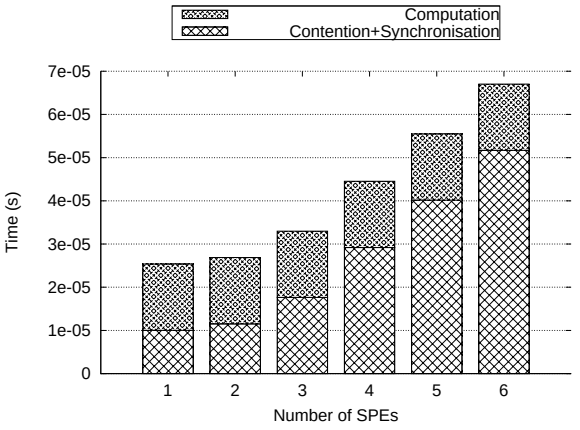

Figure 6: Relation between computation and communication using a different number of SPEs.

that makes no room for the local structures to be created. Therefore, this can be an important drawback when designing and adapting code for the Cell.

5.1 Evaluation and analysis of results in CBE

In Figure 5, the execution time of the sequential UMDA version (using only the PPE), and the execution times of the parallel versions (using a different number of SPEs) is shown. UMDA was executed for the *OneMax* problem, using the same parameters selected for the multi-threaded version, and changing the size of the individual (number of variables). It can be seen that the efficiency and scalability of this parallel version is really poor. In fact, increasing the number of SPEs does not provide any improvement.

Looking for the reasons for such a bad behavior, it must be noted that the learning of the model, which requires about the 47% of the total execution time was not parallelized (due to the DMA communication overload). Related to this, the creation of individuals and the posterior evaluation of the *OneMax* function is very fast (just an addition of the values of the variables), and once the individual has been created, it must be sent to the PPE, which gathers all the new individuals, creates the next population and continues with the next step. In Figure 6, a graphical explanation of this aspect is presented showing the average time to create and send one individual per SPE. It can be seen that as the number of used SPEs increases, the communication becomes more costly, due to contention accessing the Cell's internal network.

5.2 Additional experiments with more complex evaluation functions

As explained when discussing the implementation of the learning the probability model step, the communication between the PPE and the SPEs may become an important bottleneck for Manager-Worker approaches, particularly when the data movement cost is not amortized with a hard "number crunching" over that data. As *OneMax* is a very simple function, we decided to complete an additional set of experiments introducing a multiplicative factor that artificially makes the evaluation of the *OneMax* function harder (see Figure 7).

```
for i in 1 to MultiplicativeFactor
    Result = Evaluate_OneMax()
end for
return Result
```

Figure 7: Making the evaluation of *OneMax* more complex.

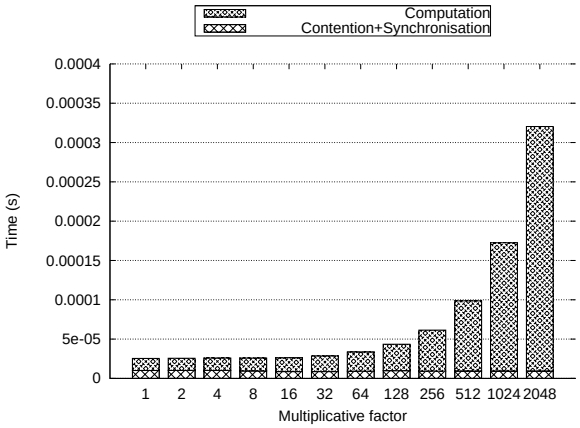

Figure 8: Computation/communication ratio for different evaluation costs.

In Figure 8, the results of these experiments are shown. Executions were completed using a single SPE, and varying the multiplicative factor from 1 to 2,048. The figure shows that, even if the communication time remains constant, the higher complexity of this new artificial function improves clearly the computation to communication ratio. Remember that, when using more that one SPE, DMA is even more costly, because several concurrent communication have to share the Cell's internal interconnection network, see Figure 6. In Figure 5 we could see that the UMDA on the Cell running *OneMax* cannot take advantage of the 6 available SPEs: the running times are almost the same using 1 or 6 SPEs. The scenario changes drastically when the evaluation function is costly: communication is no longer a bottleneck, and workers are busy in parallel doing useful work. The harder this work, the better our program scales. We can see this in Figure 9, in which we plot the program speedup (comparing execution times for 1 SPE with those for 6 SPEs) for different values of the multiplicative factor.

6. CONCLUSIONS

The effective utilization of current chip multiprocessors require careful reworking of applications, or of parts of them. In some cases, some experience with multi-thread programming is enough to exploit the computer power available from the chip, as we have shown with the thread-based implementation for the Quad-Core Xeon. Other platforms, such as the Cell, require the utilization of a series of programming artifacts that are hard to handle, and impose some limits (in terms, for example, of memory size and communication capacity) that can totally impede the successful implemen-

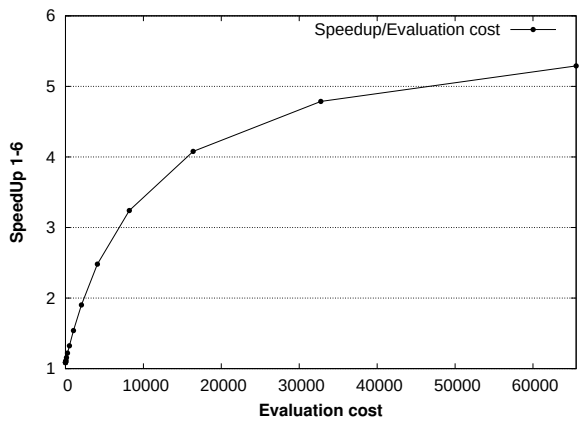

Figure 9: SpeedUp 1-6 for different evaluation costs.

tation or execution of an application.

In this paper we have discussed the case of the UMDA algorithm solving the *OneMax* problem on the Cell. The necessity of working, at each SPE, with a limited-sized Local Store forces the program to spend most of its time moving data through a limited-bandwidth network. Once data is at the LS, the effort required by the SPE to process it is too small. The application is communication-biased and, therefore, cannot fully exploit the processing power of the SPEs. The main lesson is that both the algorithm and the problem to solve are too simple. A harder problem, more computation-biased, would fit better on the Cell. We have demonstrated so by means of testing the Cell implementation of UMDA with artificially large problems (multiple repetitions of the *OneMax* problem). For the future we plan to study the portability of more complex EDAs, for example those based on Bayesian networks (discrete domains) or those based on Gaussian networks (continuous domains).

However, we have to take into account that more complex problems may mean more complex programs, with larger data structures and more code. And both have to fit into the LS. Therefore, important programming effort must be devoted to code data structures compactly, and to write compact programs. Local memory is a treasure that must not be wasted.

One might wonder if the effort of porting to the Cell is worthwhile. The answer is, definitely, yes: if we check the Top500 list of most powerful computers in the world, we see that the current "champion" (Roadrunner, #1 in list 11/2008) incorporate Cells as co-processors. Others will probably follow. We need to be ready to use these chips.

7. ACKNOWLEDGEMENTS

This work has been supported by the Ministry of Education and Science (Spain), grant TIN2007-68023-C02-02, by Saiotek and Research Groups 2007-2012 (IT-242-07) programs from the Basque Governement, TIN2008-06815-C02-01 and Consolider Ingenio 2010 - CSD2007-00018 projects (Spanish Ministry of Science and Innovation) and COMBIOMED network in computational biomedicine (Carlos III Health Institute).

8. REFERENCES

[1] Fixstars Corp. home page. `http://www.fixstars.com/`.

[2] GA-CBE home page. `http://ga-cbe.sourceforge.net/`.

[3] E. Alba and J. M. Troya. A survey of parallel distributed Genetic Algorithms. *Complex.*, 4(4):31–52, 1999.

[4] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos. Dynamic multigrain parallelization on the cell broadband engine. In *PPOPP*, pages 90–100, 2007.

[5] W. Bossert. Mathematical optimization: Are there abstract limits on natural selection? In P. S. Moorehead and M. M. Kaplan, editors, *Mathematical Challenges to the Neo-Darwinian Interpretation of Evolution*, pages 35–46. The Wistar Institute Press, Philadelphia, PA, 1967.

[6] D. R. Butenhof. *Programming with POSIXÂő Threads*. Addison–Wesley Professional Computing Series, 1997.

[7] E. Cantú-Paz. *Efficient and accurate parallel genetic algorithms*. Kluwer Academic Publishers, 2000.

[8] L. De la Ossa, J. A. Gámez, and J. M. Puerta. Migration of probability models instead of individuals: An alternative when applying the island model to EDAs. In X. Yao, E. K. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. E. Rowe, P. Tiño, A. Kabán, and H. P. Schwefel, editors, *PPSN*, volume 3242 of *Lecture Notes in Computer Science*, pages 242–252. Springer, 2004.

[9] L. J. Fogel. Autonomous automata. *Industrial Research*, 4:14–19, 1962.

[10] K.-L. Fok, T.-T. Wong, and M. L. Wong. Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems*, 22(2):69–78, 2007.

[11] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison–Wesley, Reading MA, 1989.

[12] J. J. Grefenstette. Optimization of control parameters for Genetic Algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122–128, 1986.

[13] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, 1992.

[14] IBM. *Software Development Kit for Multicore Acceleration. Programming Tutorial. Version 3.1*. 2008.

[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. J. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4-5):589–604, 2005.

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

[17] P. Larrañaga and J. A. Lozano. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, 2002.

[18] J. A. Lozano, P. Larrañaga, I. Inza, and E. Bengoetxea, editors. *Towards a New Evolutionary Computation. Advances on Estimation of Distribution Algorithms*, volume 192 of *Studies in Fuzzines and*

Soft Computing. Springer, 2005.

[19] A. Mendiburu, J. A. Lozano, and J. Miguel-Alonso. Parallel implementation of EDAs based on probabilistic graphical models. *IEEE Transactions on Evolutionary Computation*, 9(4):406–423, 2005.

[20] H. Mühlenbein. The equation for response to selection and its use for prediction. *Evolutionary Computation*, 5:303–346, 1998.

[21] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions I. Binary parameters. In H. M. Voigt, W. Ebeling, I. Rechenberger, and H. P. Schwefel, editors, *PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 178–187. Springer, 1996.

[22] J. Ocenasek and J. Schwarz. The parallel Bayesian optimization algorithm. In *Proceedings of the European Symposium on Computational Intelligence*, pages 61–67, 2000.

[23] J. Ocenasek and J. Schwarz. The distributed Bayesian optimization algorithm for combinatorial optimization. In *EUROGEN - Evolutionary Methods for Design, Optimisation and Control, CIMNE*, pages 115–120, 2001.

[24] M. Pelikan, D. E. Goldberg, and F. Lobo. A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications*, 21(1):5–20, 2002.

[25] M. Pelikan, K. Sastry, and E. Cantú-Paz. *Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications (Studies in Computational Intelligence)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[26] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann–Holzboog, Stuttgart, 1973.

[27] J. Rudin. Accelerating persistent surveillance radar with the cell broadband engine. *Embedded Technology Journal*, 2008.

[28] L. D. Whitley, S. B. Rana, and R. B. Heckendorn. Island model genetic algorithms and linearly separable problems. In D. Corne and J. L. Shapiro, editors, *Evolutionary Computing, AISB Workshop*, volume 1305 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 1997.

[29] A. Wirawan, K. C. Keong, and B. Schmidt. Parallel dna sequence alignment on the cell broadband engine. In *PPAM*, pages 1249–1256, 2007.

[30] M. L. Wong, T.-T. Wong, and K.-L. Fok. Parallel evolutionary algorithms on graphics processing unit. In *Congress on Evolutionary Computation*, pages 2286–2293, 2005.

[31] A. A. Zhigljavsky. *Theory of Global Random Search*. Kluwer Academic Publishers, 1991.