Parallel Computing 36 (2010) 618–634

Contents lists available at ScienceDirect

Parallel Computing

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

Porting Estimation of Distribution Algorithms to the Cell Broadband Engine

Carlos Pérez-Miguel *, Jose Miguel-Alonso, Alexander Mendiburu

Intelligent Systems Group, Department of Computer Architecture and Technology, The University of the Basque Country, UPV/EHU Paseo Manuel de Lardizabal, 1, 20018 San Sebastian-Donostia, Spain

ARTICLE INFO

Article history:
Available online 5 August 2010

Keywords:
Parallel computing
Cell Broadband Engine
Estimation of Distribution Algorithms

ABSTRACT

Current consumer-grade computers and game devices incorporate very powerful processors that can be used to accelerate many classes of scientific codes. In this paper we explore the ability of the Cell Broadband Engine to run two similar Estimation of Distribution Algorithms, one for the discrete domain and the other for the continuous domain. Starting from initial, sequential versions, we develop multi-threaded programs for symmetric multiprocessors that are afterwards reworked to run on a Cell-based system. In most cases, the parallel programs significantly accelerate execution times, compared with the sequential counterparts. Additional acceleration is achieved using vector (instead of scalar) operations, which are supported by all the tested platforms. We describe the process of parallelizing and porting the programs, and analyze the results obtained taking into account the EDAs under study, the problems solved with them, and the platform in which programs run. We conclude that EDAs are not right targets to be ported to the Cell Broadband Engine.

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 this chip 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 use efficiently 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.

The availability of increasingly powerful computing platforms has encouraged the design and implementation of non-trivial algorithms capable of solving different kinds of complex optimization problems. Some of these problems can be tackled 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 general frameworks 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) [1,2], Evolution Strategies [3], Evolutionary Programming [4], Estimation of Distribution Algorithms (EDAs) [5] and Genetic Programming [6]. Even though processing speeds grow fast, the

* Corresponding author.
E-mail addresses: carlos.perezm@ehu.es (C. Pérez-Miguel), j.miguel@ehu.es (J. Miguel-Alonso), alexander.mendiburu@ehu.es (A. Mendiburu).

0167-8191/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2010.07.003

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 complete.

Complex algorithms could run much faster in current machines if the implementations were 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 reworking and porting to parallel platforms two instances of Estimation of Distribution Algorithms: the Univariate Marginal Distribution Algorithm for the discrete and for the continuous domains ($UMDA_d$ and $UMDA_c$, respectively). We start from initial, sequential versions and develop first a parallel version for multi-core, symmetric systems (such as quad-core Intel Xeon). These programs are, afterwards, reworked to run on a multi-core, hybrid system: the Cell Broadband Engine. The degree of success of each of the approaches, in terms of reduction of execution times, has been very different, depending mainly on the characteristics of the problem being solved. In this paper we describe the porting process, and discuss the causes of this disparity of results. We also provide some hints about the peculiarities of the Cell processor, and some best practices to take advantage of this platform.

Portions of this work have been presented in [7,8]. In this paper we present a unified view of those preliminary works, adding new insights into the reasons behind the success (or failure) of our developments in parallel EDAs. In particular, we incorporate a discussion about influence of compilers on the obtained results, and also include some performance figures obtained on IBM's Cell simulator [9].

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 with focus on the family of Estimation of Distribution Algorithms. Section 4 introduces the Univariate Marginal Distribution Algorithm (UMDA) with its variants for the discrete and the continuous domains. In Section 5 we describe the approach we have selected to parallelize UMDA. In Section 6 we evaluate a parallel implementation that targets symmetric multiprocessors. In Section 7 we do the same with a porting to the Cell. Section 8 discusses the impact of program vectorization on execution speed. We end with some conclusions in Section 9.

2. The Cell Broadband Engine

The Cell is a microprocessor system that integrates, into a single chip, a PowerPC core (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 [10] (see Fig. 1). A programmer who wants to take full advantage of a Cell has to work with two different instruction sets: one for the PPE and another 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 poor.

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 small (256 KB) Local Store (LS). All the data processed by a SPE has to be previously transferred to its LS, and the resulting data, if needed by the PPE or another SPE, has to be explicitly transferred too. To that purpose, each SPE has a companion Memory Flow Controller (MFC) that takes

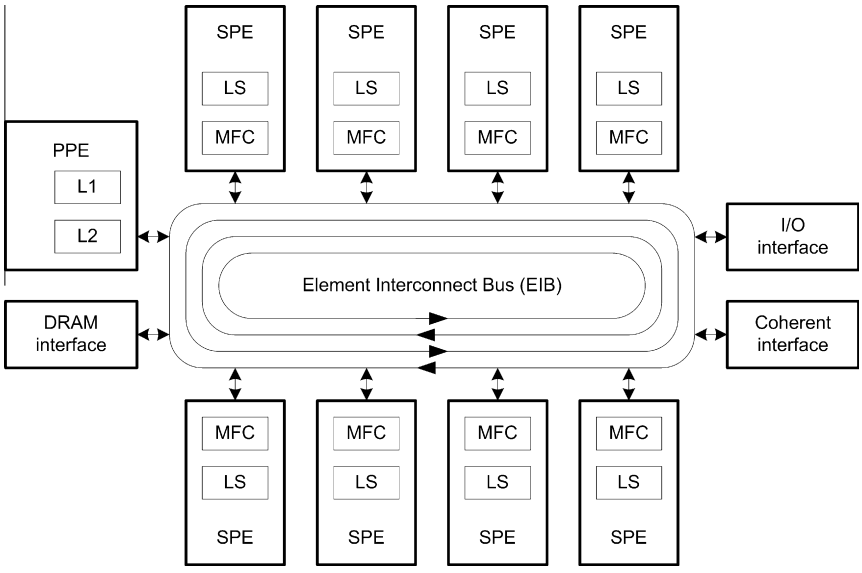


Fig. 1. Architecture of the Cell Broadband Engine.

care of data transfers using Direct Memory Access (DMA). A good programmer can manage to make a SPE and its MFC work at the same time, processing some pieces of data while transferring other pieces. A careless programmer may try to simultaneously move too much data through the interconnection fabric, which would become a bottleneck because of its limited capacity.

Challenges for the programmer are, therefore, manyfold: different instruction sets; real necessity of working with vector instructions; limited size of the LS; explicit data transfers 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 [11]. The most compute-intensive sections of the original PPE-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. In summary, using this computing platform will require the programmer to deal with unfamiliar techniques.

Regarding hardware platforms, IBM sells Cell-based blades for use in 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 into 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 distribution [12].

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 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 [13]. In addition, GAs show a poor performance in some 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 [2] 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 [5], although similar approaches can be previously found in [14]. 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 presented in Fig. 2.

Steps 3–5 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 EDAs from the point of view of complexity and performance. Depending on the used model, we can classify EDAs into these classes:

- *Without dependencies:* We assume that all the variables are independent.
- *Bivariate dependencies:* We consider only the dependencies between pairs of variables. Therefore, the process of estimating the joint probability is still simple and fast.
- *Multiple dependencies:* There are no restrictions on the dependencies to be considered. Obviously, this is a more complex process.

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. Obtain from D_l^{Se} an n dimensional model that shows the (in)dependencies between variables.
Step 5. Generate a new population D_{l+1} of M individuals sampled from the model learnt in the previous step.

Fig. 2. Outline of the Estimation of Distribution Algorithms (EDAs).

For detailed information about the characteristics of EDAs, and the algorithms that form part of this family, the interested reader can see [15–18].

4. The UMDA algorithm

In this work we focus on two variants of the Univariate Marginal Distribution Algorithm (UMDA), a very simple EDA. UMDA follows the general scheme shown in Fig. 2. In the previous section we have explained how the complexity of EDAs depends strongly on the procedure used to learn the model. UMDA uses the simplest way to estimate this model, because it assumes that all the variables in the problem are independent. Depending on whether the addressed problem fits on the discrete or on the continuous domain, these estimators will be calculated in a different way. In the next subsections we explain the differences between the continuous and the discrete versions of UMDA (UMDA$_c$ and UMDA$_d$, respectively) and we show the problems that we solve with each algorithm. These problems are common optimization problems, selected by its simplicity.

4.1. UMDA$_d$ – UMDA on the discrete domain

The algorithm UMDA$_d$ introduced in [19] considers that the model to be learnt is a probability distribution expressed as a product of n univariate and independent probability distributions:

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

where each Univariate Marginal Distribution is estimated from marginal frequencies:

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

being

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

We have applied this algorithm to solve *OneMax*, a well known optimization problem with a very simple objective function. This problem consists of maximizing:

$$OneMax(\boldsymbol{x}) = \sum_{i=1}^{n} x_i \tag{4}$$

where $x_i \in \{0,1\}$. That is, the best solution is reached when all the variables of the individual take value 1.

4.2. UMDA$_c$ – UMDA on the continuous domain

The variation of UMDA for the continuous domain (UMDA$_c$) [20,21] considers that the model to be learnt is a joint density function that follows a n-dimensional normal distribution, which is factorized by a product of one-dimensional and independent normal densities. In every generation and for every variable, UMDA$_c$ performs some statistical tests in order to find the density function that best fits the sampling of that variable.

UMDA$_c$ is a structure identification algorithm, because the density components of the model to be learnt are identified via hypothesis tests. This estimation of parameters is performed, after the densities are identified, by their maximum likelihood estimates. If all the univariate distributions are normal, then the two parameters to be estimated at each generation l and for

each variable are the mean, μ_i^l, and the standard deviation, σ_i^l. It is well known that their respective maximum likelihood estimates are:

$$\widehat{\mu_i^l} = \overline{X_i^l} = \frac{1}{N}\sum_{r=1}^{N} x_{i,r}^l \tag{5}$$

$$\widehat{\sigma_i^l} = \sqrt{\frac{1}{N}\sum_{r=1}^{N}(x_{i,r}^l - \overline{X_i^l})^2} \tag{6}$$

being N the number of individuals in the population and $x_{i,r}^l$ the different variables which compound each individual. These two parameters, μ and σ, will be used later to generate new individuals. For this purpose, an adaptation of the Probabilistic Logic Sampling (PLS) proposed in [22] is used. With PLS the instances are generated one variable at a time in a forward way. For the generation of a univariate normal distribution, a simple method based on the sum of 12 uniform variables is applied [23].

In this work we have tested UMDA$_c$ applied to the resolution of the *Sphere model* optimization problem. This is a simple minimization problem, defined so that each variable takes values from a range $-M \leqslant x_i \leqslant M$, $i = 1,\ldots,n$ and the fitness function for each individual is as follows:

$$F(x) = \sum_{i=1} x_i^2 \tag{7}$$

As we can see, the fittest individual is the one in which all components are 0, which corresponds to the fitness value 0.

5. Parallelizing UMDA

The resolution of problems by means of programs that implement Evolutionary Algorithms requires, in general, long execution times. For this reason, researchers often apply parallel techniques to reduce running times. The same techniques can also be used to improve solution accuracy, exploring a wider part of the solution space, or to manage larger problems within the same time budget [2,24]. In this paper we describe our experience with the parallelization of UMDA, considering the two versions of the algorithm. They are not identical, but the approach followed in both cases is basically the same – we will state the differences when necessary.

There are two basic approaches to parallelize EAs (including EDAs and, therefore, UMDA): acceleration of program sections, and division of the population into several independent sub-populations (islands model) [25–27]. When using the islands-based approach, the single population used in sequential algorithms is split into several sub-populations (islands). These islands evolve independently, and exchange information about their individuals with a specified frequency. These models are particularly suitable for distributed systems, because each island can be mapped onto a separate processor, and the amount of communications required between islands is not very large.

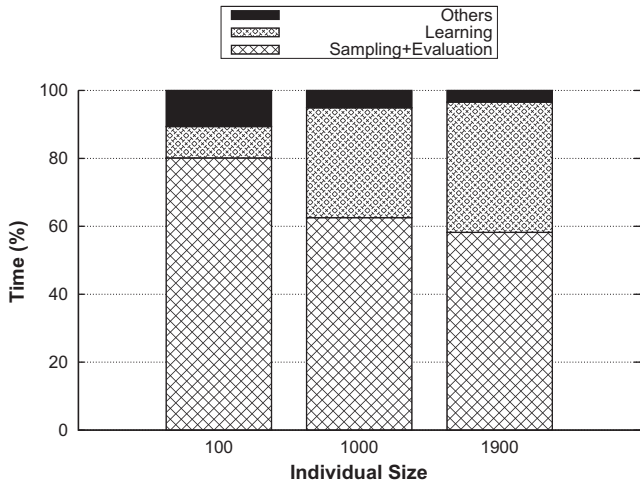
A more conservative approach starts with a sequential algorithm, parallelizing parts of it in order to reduce the execution time but without changing the semantics of the algorithm. The most time-consuming portions of the code are identified and rewritten to take advantage of a parallel computer. Among the techniques to parallelize the code, or portions of it, the Manager–Worker model is a popular one: a Manager task runs the main program, and delegates CPU-intensive parts to a collection of Worker tasks. For the interested reader, in the literature we can find different parallelization proposals following this approach, for Genetic Algorithms [28] and also for EDAs [29–31].

The selection of the parallelization paradigm has to be done taking into account the characteristics of the target computing platform. We will not work with a cluster of computers, but with on-chip multiprocessors including a Cell system. The limited memory of the Cell's LSs does not allow us to run a complete EDA (an island) on each SPE. Therefore, we have opted for a Manager–Worker approach in which the Manager task runs on the PPE and the Workers on the SPEs.

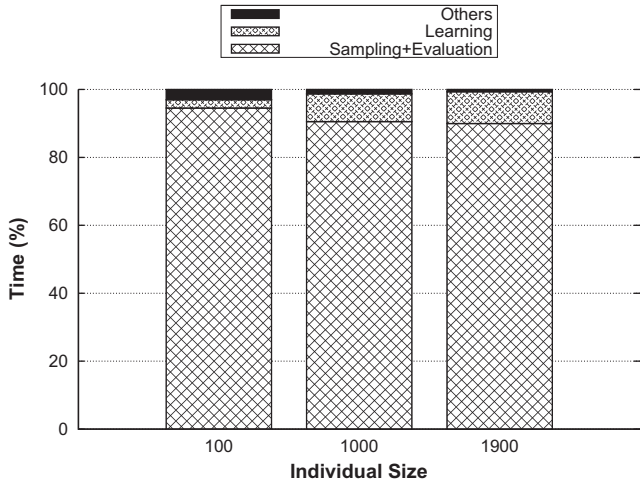
The starting point of our work is a sequential version of UMDA written in C++. We completed a profile of the program, identifying those phases in which it spends most of the CPU time – thus, the candidates for acceleration. We have split running time into three portions, related to the main parts of the algorithm: Sampling + Evaluation of new individuals, Learning of the new model, and other tasks. In Fig. 3 we have plotted the results of this profiling. As we can see, the most expensive part is always the Sampling + Evaluation phase, which takes between 60% and 95% of the execution; actual percentage depends on the algorithm, the problem being solved and the individual size. The second most-costly part is the Learning phase.

The degree of success we can expect from our acceleration approaches depend on the parts of the program we are dealing with. The larger the non-parallelized portions, the lesser the expected reduction of time. For example, a parallelization of the Sampling + Evaluation and the Learning phases should be more successful than a parallelization of the Sampling + Evaluation phase only.

Note that in this paper we are dealing with very simple EDAs solving very simple problems. In other scenarios the relative cost of the Learning phase would be higher because of the complexity of Learning a model that takes into consideration the dependencies between variables. Similarly, we are dealing with problems (*OneMax* and *Sphere model*) with extremely simple evaluation functions; in any other problem the evaluation of individuals would be more costly, increasing the relative effort devoted to the Sampling + Evaluation phase. Therefore the reader should understand that, to a certain extent, this work could



(a) UMDA$_d$



(b) UMDA$_c$

Fig. 3. Profiling of UMDA.

be considered a worst-case study on the parallelization of EDAs. As we will see later, for the Cell platform, there are other limitations that could impede the parallelization of costly EDA/problem combinations.

Although our final objective is to test the Cell platform, we considered of great interest for comparison purposes the development of parallel versions of our programs targeting state-of-the-art, multi-core x86 desktop computers. In addition to this, as all the tested platforms (x86-based machines and the Cell) include vector instructions, we manually reworked all the programs to take advantage of this feature in order to further accelerate the programs.

Regarding the specific platforms, we have used:

- A quad-core desktop computer with an Intel Xeon E5420 processor at 2.50 GHz, 4 GB of RAM and GNU/Gentoo Linux operating system. In this platform, programs have been compiled using GNU's *gcc* (version 4.3.2) [32].
- The PS3 installation of GNU-Linux described before. Tested compilers are GNU's *gcc* (version 4.1.1) and IBM's *xlc* (version V10.1) [33].

For tuning and debugging tasks, we have also used GNU's *gprof* profiling tool [34], and the Full-System Simulator for the Cell Broadband Engine Processor [9].

6. Parallel UMDA on a quad-core Xeon

As stated before, our approach to the parallelization of UMDA has been the implementation of a Manager–Worker scheme. For the desktop PC platform, Manager and Workers are implemented by means of Posix Threads (Pthreads) [35].

Using this API all the threads can share the main data structures (they can have private variables too) and have mechanisms to synchronize if required. The UMDA algorithm is executed by the main thread (Manager) and, when necessary, it asks the Workers for help. In particular, Workers collaborate in these phases:

- *Learning the model:* The Manager will ask the Workers to obtain parts of the model for a subset of the selected population. Once all Workers have finished, the Manager creates the main model based on the partial values.
- *Sampling + Evaluation*: Based on the model learnt in the previous step, new individuals will be created and evaluated. Note that this is the most expensive phase of UMDA, as can be seen through the execution profile of the sequential programs. 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. When the evaluation of individuals takes always the same computing time, a static assignment would be enough to guarantee a good balance of the computational workload among the Workers. However, if the time required to evaluate an individual depends on the values it takes, the on-demand scheme would be preferable. In our experiments we use a static distribution of the workload as the fitness evaluation takes a constant time.

Both $UMDA_d$ and $UMDA_c$ have been parallelized using this approach. In order to test the parallel implementations, we have run and compared them against the sequential counterparts when solving the test problems discussed in Section 4 (*OneMax* for $UMDA_d$ and *Sphere model* for $UMDA_c$). We used different individual sizes (L) for the target problems, ranging from 100 to 1900 variables, using a population size of $2.5L$. The stopping criterion was the evaluation of 50 generations. The target machine in all cases was the quad-core Xeon, with a number of Worker threads that varied from 1 to 4. The results of all these experiments have been summarized in Figs. 4 and 5, in which we can see the execution times for both versions when the individual size grows.

Note that the running times of the sequential runs are different to those of the 1-Worker runs. This is because in the 1-Worker cases there are two threads running in different cores: the Manager and a Worker – although they never run simultaneously. The use of more resources (in particular, of more cache memory) provides this slight improvement in efficiency.

The most relevant result is that the parallel programs are always faster than the sequential ones, and they scale very well with the number of Workers. Efficiency levels are close to the theoretical limit. Application speedups, relative to the sequential versions, are summarized in Table 1.

7. 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 hybrid multiprocessor.

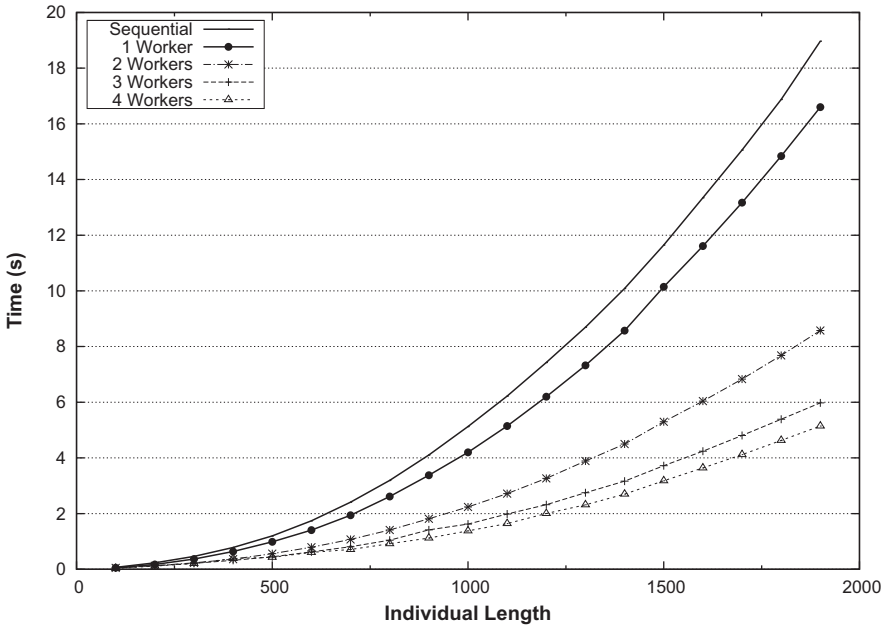


Fig. 4. $UMDA_d$ solving the *OneMax* problem. Execution times on the quad-core Xeon.

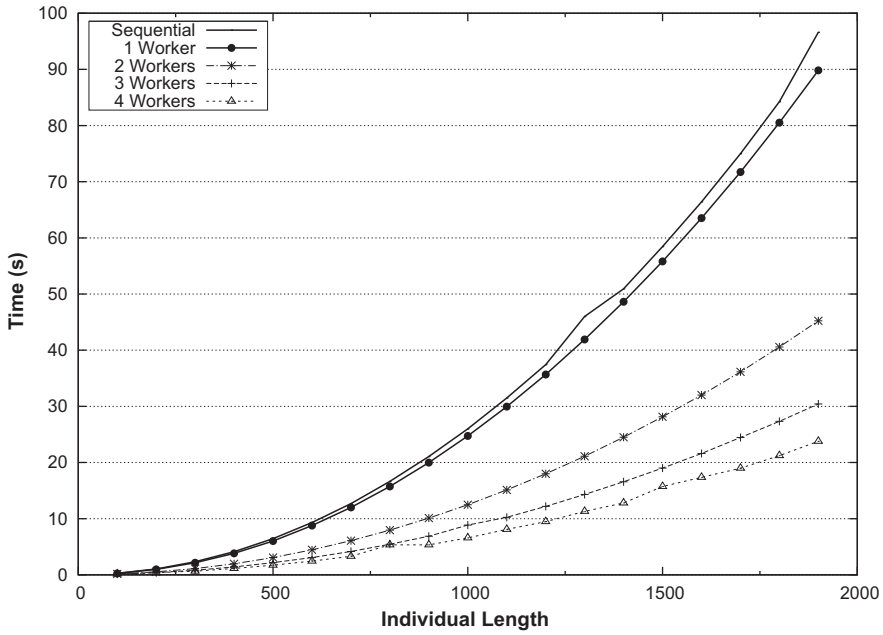


Fig. 5. UMDA$_c$ solving the *Sphere model* problem. Execution times on the quad-core Xeon.

Table 1
Speedups of UMDA on the quad-core Xeon, for different individual lengths.

Num. Workers	UMDA$_d$			UMDA$_c$		
	100	1000	1900	100	1000	1900
1	1.33	1.22	1.14	1.11	1.05	1.07
2	1.44	2.29	2.21	1.86	2.08	2.13
3	1.59	3.16	3.17	2.16	2.93	3.17
4	2.18	3.71	3.92	1.96	3.95	4.06

7.1. Modifications to the program structure

Firstly we have to deal with heterogeneity: the Master thread will run on the PPE, and the Worker threads will be separate programs running on the SPEs. In the quad-core Xeon we took advantage of a large, shared memory space, which simplifies communication among tasks avoiding 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 LSs and vice versa. This process is costly not only in terms of programming difficulty, but also in terms of application running times.

The profiling of UMDA helped us identifying those parts that should be delegated to the Workers: Learning the probability model, and Sampling + Evaluation. That is what we did for the quad-core Xeon platform. For the Cell, though, we made the Workers deal only with the latter. During the Learning phase, in the multi-thread version, the Manager asks the Workers to obtain the partial models for a subset of the selected individuals, a subset to which Workers have access by means of the shared memory. In the case of the Cell, it would be necessary to explicitly send the subsets to the SPEs using DMA transfers. Taking into account that the model learnt by UMDA is very simple, the cost of computing a part of the model is smaller than the cost of the transfers – therefore, efficiency would be penalized.

In Listings 1 and 2 we can see how a SPE uses DMA transfers to fetch a new model at the beginning of each new generation, and how explicit transfers are used again to send the recently generated individuals.

For the Sampling + Evaluation phase, we repeat for the Cell the same scheme used in the multi-thread code. The Manager asks the Workers to create and evaluate a given number of individuals. The Workers (executed in the SPEs), 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 Figs. 6 and 7 we can see the pseudo-codes for the parallelized version of UMDA: the Manager running in the PPE, and the Workers running in the SPEs. These figures can be directly compared with Fig. 2, which corresponds to the sequential version.

The decisions about the program phases that are delegated to the Workers, together with the profiling information from the sequential UMDA (see again Fig. 3), allows us to predict that the levels of scalability of the Cell ports will not be as good as

```
tag = spu_read_in_mbox ();
// Synchronization via Mailboxes

mfc_get (model, model_address, model_size, tag, 0, 0);
// DMA transfer initiation. Parameters are:
// model: A floats vector where the model is stored
// model_address: A pointer to destination at Main Memory
// model_size: Model's size
// tag: Channel to be used

mfc_write_tag_mask (tag_mask);
mfc_read_tag_status_all ();
// Waiting for the transfer finalization at channel stored in tag
```

Listing 1. DMA model transfer in the SPE.

```
mfc_write_tag_mask (1<<z);
spu_mfcstat (MFC_TAG_UPDATE_ALL);
spu_write_out_mbox (z);
// Waiting to the last transfer end over the channel z

z = spu_read_in_mbox ();
// Signaling availability of the channel from the PPE

delete inds [z];
inds [z] = ind;
// Updating the buffer

mfc_put ( inds [z], inds [z].address, inds [z].size, z, 0, 0);
// Transfering the new individual over channel z
```

Listing 2. Transfer of an individual in the SPE.

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. Obtain from D_l^{Se} a model that shows the (in)dependencies between variables.
Synchronization via Mailboxes.
Step 5. Ask the SPEs to generate and evaluate N individuals.
Synchronization via Mailboxes.

Fig. 6. Pseudo-code of UMDA on the Cell – PPE part.

Synchronization via Mailboxes.
Step 1. Transfer the model from the PPE to each SPE.
Step 2. For $i = 1..M/6$ do
Step 3. Generate a new individual sampled from the model learnt in the previous step.
Step 4. Evaluate the individual.
Step 5. Transfer the individual via DMA. from the SPE's LS to the PPE's Main Memory.
Synchronization via Mailboxes.

Fig. 7. Pseudo-code of UMDA on the Cell – SPE part.

those obtained with the quad-core platform. In particular, we expect the parallel version of $UMDA_c$ scaling much better than the parallel version of $UMDA_d$.

When dealing with the Cell, a good parallelization scheme is not the only key to obtain an effective program. Due to the limited size of each SPE's LS (256 KB for *both* code and data), it is recommendable (even compulsory) to reduce the size of the code as much as possible. In the following lines we discuss some ideas that can help obtaining binaries of small size:

Optimization flags: The optimization level used when compiling a program affects not only the speed of the executable, but also its size. In Tables 2 and 3 we show the impact that different optimization flags for GNU *gcc* and IBM *xlc* have on the overall execution times of our UMDA codes, and also the size of the Worker part (the binary that runs on the SPEs). In this particular case all options for *gcc* result in very similar execution times, but we can notice significant differences in sizes. However, with *xlc* there are notable differences in both execution times and code sizes. The right size/speed ratio has to be found, taking into account that a smaller code can free more space for larger data structures, something that conveniently exploited can result in shorter running times. For the experiments, we have used the -O3 optimization flag with the *gcc* compiler.

C++ exception handling: This mechanism increases in about a 10% the size of the code. We can avoid this system (for well debugged code) using the flag *-fno-exceptions* with *gcc*, or *-qnoeh* with *xlc*.

Libraries: It is also important to be careful with the libraries linked with the SPE code. As dynamic linking is not available for the SPEs, all the required libraries must be linked statically, increasing the final size of the executable. For example, an implementation of UMDA populations using the C++ Standard Template Library makes final code 70 KB larger than an alternative implementation using arrays.

Ignoring these suggestions can result in a code that does not fit into the SPEs' Local Stores, or a code that leaves no room for the local data structures. This can be an important drawback when designing and adapting code for the Cell.

7.2. Performance of parallel UMDA on the Cell

In this section we analyze the behavior of the Cell ports of UMDA. Tests have been made varying the length of the individuals from 100 to 1900 variables; the number of Worker threads (used SPEs) has been varied from 1 to 6, and the stopping criterion was to complete 50 generations for each problem. This criterion has been selected to guarantee a fair comparison between program versions – although it may make programs stop before reaching an optimal solution.

Results are summarized in Figs. 8 and 9 and Table 4. The first conclusion is that $UMDA_d$ does not make an efficient usage of the Cell architecture. The 1-Worker version (involving the PPE plus a single SPE) is slightly faster than the sequential version (running only on the PPE). Adding additional resources (more Workers) does not significantly accelerate running times.

Table 2
Code size and execution times (solving *OneMax* and *Sphere Model*) for different *gcc* compilation flags. Individual size = 1000. Fifty generations. Six SPEs.

Flags	$UMDA_d$		$UMDA_c$	
	Size (bytes)	Exec. time (s)	Size (bytes)	Exec. time (s)
-Os	120,860	65.65	120,796	29.99
-O0	144,628	141.19	144,300	72.10
-O1	124,076	49.59	123,964	20.37
-O2	123,164	54.33	123,068	20.06
-O3	126,772	49.56	126,676	20.21

Table 3
Code size and execution times for different *xlc* compilation flags. Individual size = 1000. Fifty generations. Six SPEs.

Flags	$UMDA_d$		$UMDA_c$	
	Size (bytes)	Exec. time (s)	Size (bytes)	Exec. time (s)
-O0	147,944	140.60	147,752	69.38
-O2	156,520	48.84	156,136	19.72
-O2 -qcompact	150,056	95.83	149,800	43.75
-O3	184,856	48.41	190,120	19.44
-O3 -qcompact	159,960	93.44	162,024	43.03
-O4	98,896	46.60	106,448	19.14
-O4 -qcompact	101,936	555.50	104,320	276.19
-O5	102,352	45.66	103,504	18.18
-O5 -qcompact	97,080	545.49	99,064	269.69

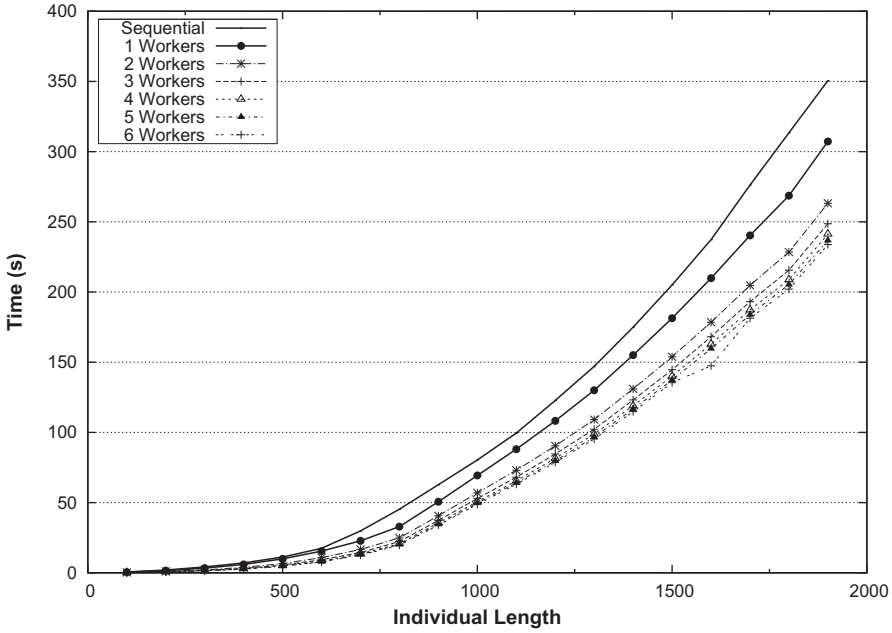


Fig. 8. UMDA$_d$ solving the *OneMax* problem. Execution times on the Cell.

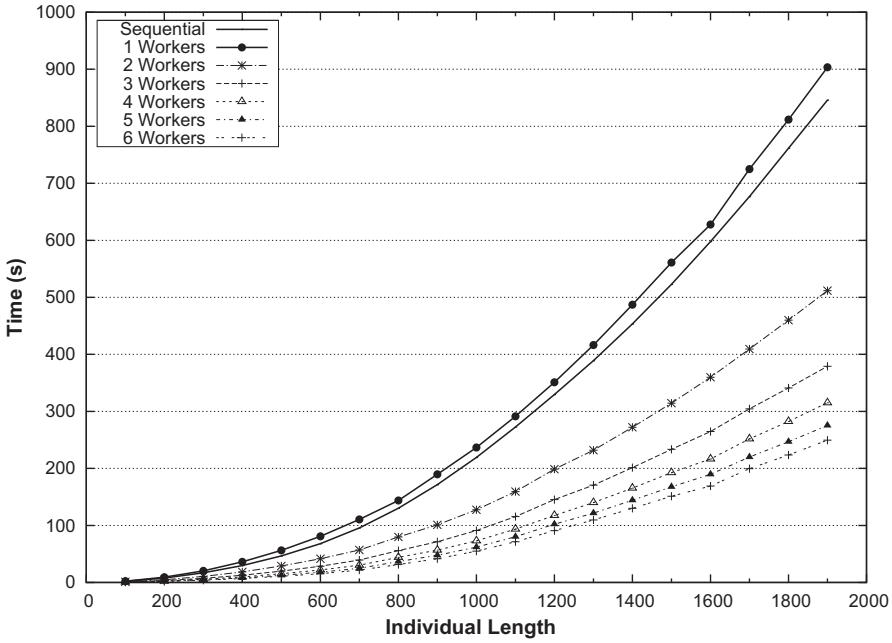


Fig. 9. UMDA$_c$ solving the *Sphere Model* problem. Execution times on the Cell.

Table 4
Speedups of UMDA on the Cell, for different individual lengths.

Num. Workers	UMDA$_d$			UMDA$_c$		
	100	1000	1900	100	1000	1900
1	1.47	1.15	1.14	0.87	0.92	0.93
2	2.32	1.41	1.33	1.67	1.71	1.65
3	2.91	1.52	1.40	2.44	2.40	2.23
4	2.97	1.58	1.45	3.03	3.00	2.68
5	2.86	1.62	1.48	3.70	3.53	3.07
6	2.84	1.64	1.49	4.07	3.98	3.38

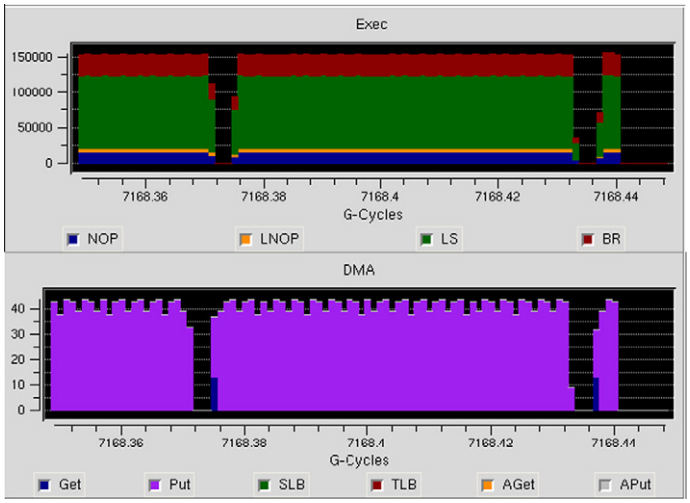


Fig. 10. *OneMax* on the Cell simulator. Individual length = 200, one SPE.

Fig. 11. *OneMax* on the Cell simulator. Individual length = 200, six SPEs.

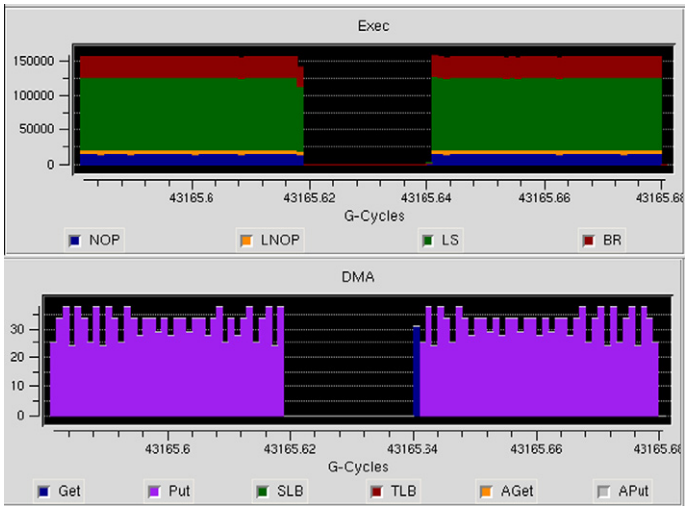
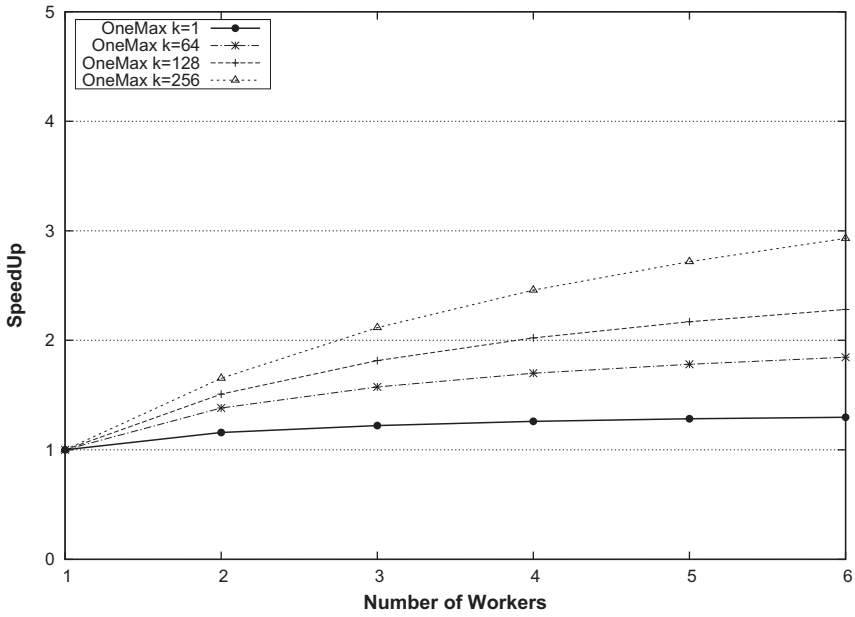


Fig. 12. *OneMax* on the Cell simulator. Individual length = 500, one SPE.

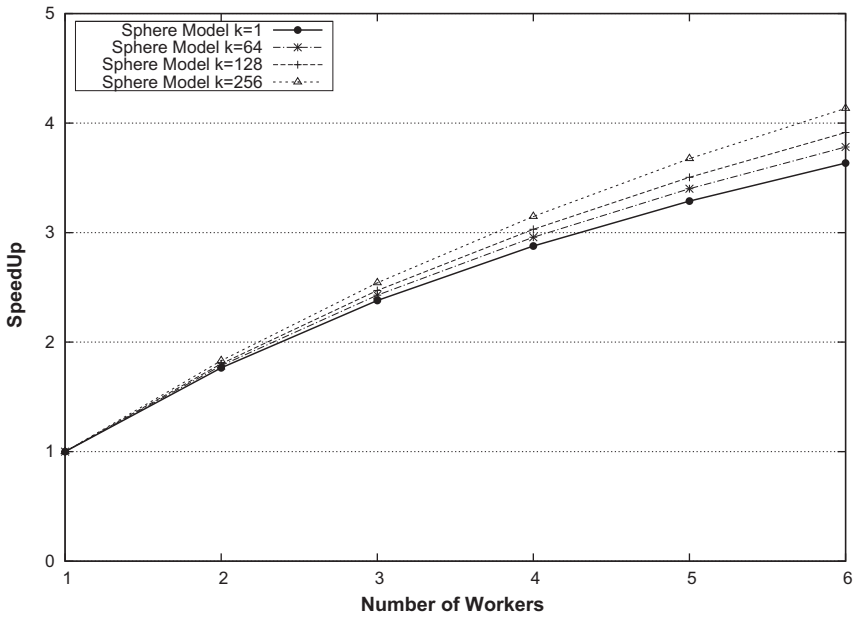
For UMDA$_c$ the picture is different: sequential and 1-Worker versions take approximately the same time, and adding Workers always result in smaller times; the maximum speedup achieved is around 3.5 for 6 SPEs, independently of the individual size. Note that, from the profiling information, it is not surprising that UMDA$_c$ scales better than UMDA$_d$, because the parallelized portion of the program (the Sampling + Evaluation phase) is significantly larger.

In order to better understand the reasons behind the scalability properties of the programs, we have used IBM's Full-System Simulator for the Cell [9]. We ran the programs inside this environment, making measurements of resource utilization on the SPEs. Some of these measurements are depicted in Figs. 10–12. Each figure comprises two parts: the upper one shows the CPU utilization of one SPE, while the lower one depicts the DMA transfers carried out in the same SPE.

Figs. 10 and 11 correspond to the execution of UMDA$_d$ with individuals of 200 variables, using one and six SPEs, respectively. The gaps between execution blocks correspond to the execution of the non-parallelized parts of the program on the



(a) UMDA$_d$

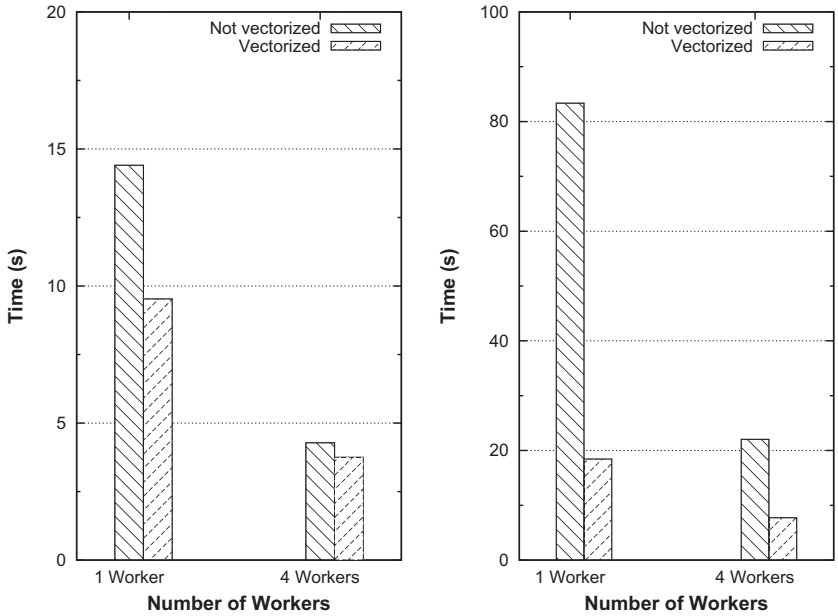


(b) UMDA$_c$

Fig. 13. Speedups of parallel UMDA on the Cell for different values of k. Ind. length = 1900

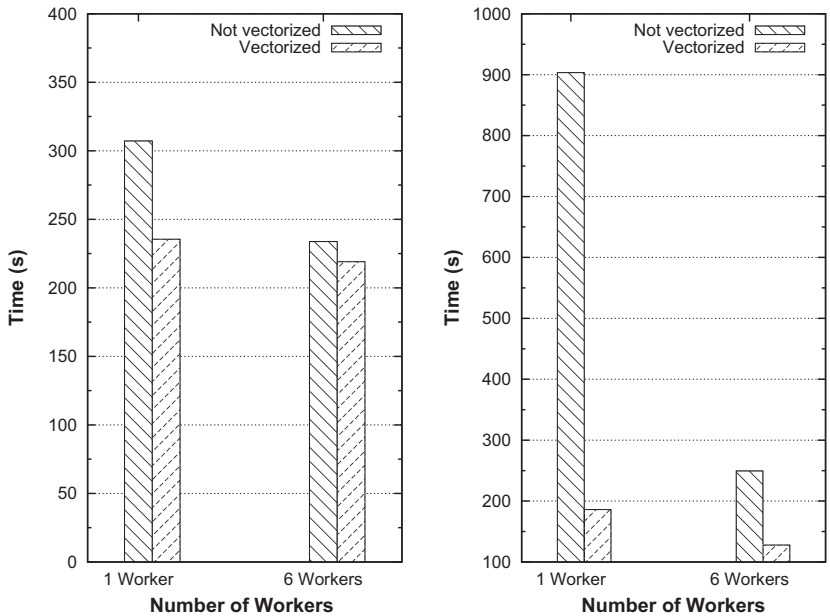
PPE. It can be seen that, with a single SPE, these gaps are relatively small compared to the Worker execution blocks. This is obviously not the case when there are six Workers, because execution blocks are notably shorter and the sequential code gains relative weight.

When individual size grows, the non-parallelized part of UMDA$_d$ increases too (Fig. 3). Therefore, the weight of the PPE part should augment. We have confirmed this behavior running the program in the simulator using an individual with 500 variables: see Fig. 12.



(a) UMDA$_d$ (b) UMDA$_c$

Fig. 14. Impact of vectorization of UMDA for the quad-core Xeon.



(a) UMDA$_d$ (b) UMDA$_c$

Fig. 15. Impact of vectorization of UMDA for the Cell.

For the sake of brevity we have not included the figures for the $UMDA_c$ program, as they do not provide additional insights into the program's behavior. Finally, we want to remark that the graphs allow us to verify that the double-buffer technique works well, allowing the SPEs to run while data transfers are taking place.

In the Introduction we discussed how the utilization of simple evaluation functions can be considered a stress-test for the parallel UMDAs, because the part to run in parallel is precisely the Sampling + Evaluation phase. What would happen if these functions were not that simple?

Instead of using other, more costly, evaluation functions, we decided to run additional tests artificially increasing the evaluation cost. To that extent, we added a parameter k to the programs, which indicates a number of times the evaluation function will be (unnecessarily) executed. This way we can manually tune the relative weight of the Sampling + Evaluation phase. Results for different values of k, for an individual size of 1900, are summarized in Fig. 13. As expected, now $UMDA_d$ shows better scalability figures, because SPEs are used more efficiently (run for more time between sequential phases). It is not surprising that with $UMDA_c$ additional benefits are minor, because the non-parallelized phases of the algorithm were small from the beginning.

8. Impact of manual vectorization

Vector processors give us the possibility to operate over multiple data elements with a single instruction. As the two platforms used in our experiments integrate support for vector operations, we decided to modify our UMDA implementations to take advantage of this feature.

We already know that the most expensive phase of UMDA is the sampling and evaluation of new individuals. To accelerate the sampling step we have adapted (vectorized) the random number generation function, making it capable of generating values in groups of four. Similarly, the function that evaluates individuals was rewritten to carry out the different arithmetic operations in groups of four.

The vectorized version of the Cell implementation was made using SPE intrinsics [36] and the *libmisc* library included in the IBM Cell SDK [37], which provides a vectorized version of a uniform random number generator. For the Intel platform we used SSE intrinsics [38] and implemented our own vectorized version of a linear congruential generator [39].

We repeated the experiments with the manually vectorized implementations of UMDA when managing problems with individuals of length 1900. In Fig. 14 we summarize the execution times on the quad-core Xeon, comparing vectorized vs. non-vectorized code using one and four Workers; this is done to verify that benefits of vectorization add to those of parallelization. Results for the Cell are in Fig. 15, using one and six Workers.

We can observe that manual vectorization is always beneficial, and that the acceleration levels for $UMDA_c$ are impressive. While this is true for both platforms, benefits on the Cell are higher. This is because the SPEs are vector processors, designed specifically for this form of computing, and the cost of a scalar operation with a single data element can be *superior* to that of a vector operation over four data elements.

These figures also allow us to do a direct platform-to-platform comparison. They show clearly that, for the class of codes we are dealing with, the quad-core Xeon can be up to one order of magnitude faster than the Cell platform. We have to take into consideration that the Xeon platform is a high-end desktop computer, considerably more expensive than a PS3 (but not ten times more expensive). Additionally, in our experience, the homogeneous quad-core is much easier to program than the heterogeneous Cell – except the part related to vectorization, which is harder to use in the Xeon. The promise of huge computing power for a budget using consumer gadgets can be true, but the programming effort required to take advantage of these platforms is still too high.

9. Conclusions

In this paper we have implemented and evaluated several parallel ports of UMDA, an instance of the class of Estimation of Distribution Algorithms, for two different platforms: a homogeneous quad-core Intel Xeon and a hybrid Cell processor that fuels a Sony PlayStation 3. A profile of the sequential program was the key to identifying the portions to be accelerated, via parallelism based on a Manager–Worker scheme. In the quad-core Xeon Manager and Workers were implemented using threads that communicate by means of shared memory structures. In the Cell the implementation was more complex for different causes, being the main one the lack of a global memory shared by all the SPEs. In this platform the Manager runs on the PPE and the Workers on the SPEs. DMA transfers are required to move data (individuals) back and forth the main memory.

Experimental work shows that the degree of success of these programs depends on the platform on which they run. The Xeon versions have been a complete success: for large-enough problems (in terms of complexity of the functions to evaluate, and also in terms of individual sizes) the utilization of multiple processing cores immediately results in program acceleration. Achieved speedups are very close to the theoretical ones. And this can be accomplished without major programming efforts.

The Cell platform, main theme of this paper, presents a different picture. Programming the Cell is a challenge for many reasons, from which we stress one: the lack of a large, shared memory. SPEs are only able to process data and instructions from their small LSs. The programmer has to include instructions to move data back and forth, and SPE programs have to be

small. These restrictions limited the amount of computational work delegated to the SPEs, and also the complexity of the problems being solved. Regarding program acceleration, achieved speedups have always been less than four, when using the six available SPEs.

As all the platforms used in our experiments integrate support for vector operations, we have tested manually vectorized versions of the programs, implemented using different repertoires of intrinsics. The vectorized programs perform exceedingly well, especially on the Cell. The lesson to learn is that if we ignore the vector nature of this platform we will get poor performance figures.

To summarize this work, we can affirm that a Cell system is full of potential, but this potential is not easily extracted. The required programming effort is considerably high, and returns are not always good. Still, once a programmer has familiarity with the platform, the task of efficiently using it becomes easier, giving us the possibility of using a low-cost but powerful machine.

Acknowledgements

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

References

[1] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, MA, 1989.
[2] J.H. Holland, Adaptation in Natural and Artificial Systems, The University of Michigan Press, Ann Arbor, MI, 1992.
[3] I. Rechenberg, Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, Frommann-Holzboog, Stuttgart, 1973.
[4] L.J. Fogel, Autonomous automata, Industrial Research 4 (1962) 14–19.
[5] H. Mühlenbein, G. Paaß, From recombination of genes to the estimation of distributions I. Binary parameters, in: H.M. Voigt, W. Ebeling, I. Rechenberger, H.P. Schwefel (Eds.), PPSN IV, Lecture Notes in Computer Science, vol. 1141, Springer, 1996, pp. 178–187.
[6] J.R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, 1992.
[7] C. Pérez-Miguel, J. Miguel-Alonso, A. Mendiburu, Evaluating the cell broadband engine as a platform to run estimation of distribution algorithms, in: F. Rothlauf (Ed.), GECCO (Companion), ACM, 2009, pp. 2491–2498.
[8] C. Pérez-Miguel, J. Miguel-Alonso, A. Mendiburu, Porting estimation of distribution algorithms to the cell broadband engine, in: Workshop on Parallel Architectures and Bioinspired Algorithms (WPABA) in Conjunction with PACT 2009, Raleigh, North Carolina, September 12–16, 2009.
[9] IBM Corp. – IBM Full-System Simulator for the Cell Broadband Engine Processor, <http://www.alphaworks.ibm.com/tech/cellsystemsim>.
[10] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, D.J. Shippy, Introduction to the cell multiprocessor, IBM Journal of Research and Development 49 (4–5) (2005) 589–604.
[11] IBM Corp., Software Development Kit for Multicore Acceleration, Programming Tutorial, version 3.1, 2008, <http://publib.boulder.ibm.com/infocenter/systems/topic/eiccj/tutorial/CBE_Programming_Tutorial_v3.1.pdf?Open&S_TACT=105AGX16&S_CMP=LP>.
[12] Fixstars Corp., <http://www.fixstars.com/>.
[13] J.J. Grefenstette, Optimization of control parameters for Genetic Algorithms, IEEE Transactions on Systems, Man, and Cybernetics 16 (1) (1986) 122–128.
[14] A.A. Zhigljavsky, Theory of Global Random Search, Kluwer Academic Publishers, 1991.
[15] P. Larrañaga, J.A. Lozano, Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation, Kluwer Academic Publishers, 2002.
[16] M. Pelikan, D.E. Goldberg, F. Lobo, A survey of optimization by building and using probabilistic models, Computational Optimization and Applications 21 (1) (2002) 5–20.
[17] J.A. Lozano, P. Larrañaga, I. Inza, E. Bengoetxea (Eds.), Towards a New Evolutionary Computation. Advances on Estimation of Distribution Algorithms, Studies in Fuzziness and Soft Computing, vol. 192, Springer, 2005.
[18] M. Pelikan, K. Sastry, 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.
[19] H. Mühlenbein, The equation for response to selection and its use for prediction, Evolutionary Computation 5 (1998) 303–346.
[20] P. Larrañaga, R. Etxeberria, J. Lozano, J. Peña, Optimization by learning and simulation of Bayesian and Gaussian networks, Tech. Rep. KZZA-IK-4-99, Department of Computer Science and Artificial Intelligence, University of the Basque Country, 1999.
[21] P. Larrañaga, R. Etxeberria, J. Lozano, J. Peña, Optimization in continuous domains by learning and simulation of Gaussian networks, in: L.D. Whitley, D.E. Goldberg, E. Cantú-Paz, L. Spector, I.C. Parmee, H.G. Beyer (Eds.), GECCO, Morgan Kaufmann, 2000, pp. 201–204.
[22] M. Henrion, Propagating uncertainty in Bayesian networks by probabilistic logic sampling, in: R.D. Shachter, T.S. Levitt, L.N. Kanal, J.F. Lemmer (Eds.), UAI, North-Holland, 1988, pp. 149–163.
[23] B.D. Ripley, Stochastic Simulation, John Wiley and Sons, 1987.
[24] W. Bossert, Mathematical optimization: are there abstract limits on natural selection?, in: P.S. Moorehead, M.M. Kaplan (Eds.), Mathematical Challenges to the Neo-Darwinian Interpretation of Evolution, The Wistar Institute Press, Philadelphia, PA, 1967, pp. 35–46.
[25] J. Madera, E. Alba, A. Ochoa, A parallel island model for estimation of distribution algorithms, in: J.A. Lozano, P. Larrañaga, I. Inza, E. Bengoetxea (Eds.), Towards a New Evolutionary Computation. Advances on Estimation of Distribution Algorithms, Studies in Fuzziness and Soft Computing, vol. 192, Springer, 2005, pp. 159–186.
[26] L. De la Ossa, J.A. Gámez, J.M. Puerta, Initial approaches to the application of islands-based parallel EDAs in continuous domains, in: T. Skie, C.S. Yang (Eds.), ICPP Workshops, IEEE Computer Society, 2005, pp. 580–587.
[27] J. Ocenasek, Parallel estimation of distribution algorithms, Ph.D. Thesis, Faculty of Information Technology, Brno University of Technology, 2002.
[28] E. Cantú-Paz, Efficient and Accurate Parallel Genetic Algorithms, Kluwer Academic Publishers, 2000.
[29] J. Ocenasek, J. Schwarz, The parallel Bayesian optimization algorithm, in: Proceedings of the European Symposium on Computational Intelligence, 2000, pp. 61–67.
[30] J. Ocenasek, J. Schwarz, The distributed Bayesian optimization algorithm for combinatorial optimization, in: EUROGEN – Evolutionary Methods for Design, Optimisation and Control, CIMNE, 2001, pp. 115–120.
[31] A. Mendiburu, J.A. Lozano, J. Miguel-Alonso, Parallel implementation of EDAs based on probabilistic graphical models, IEEE Transactions on Evolutionary Computation 9 (4) (2005) 406–423.
[32] The GNU Project – The GNU Compiler Collection, <http://gcc.gnu.org/>.

[33] IBM Corp. – XL C/C++ for Multicore Acceleration for Linux, <http://www-01.ibm.com/software/awdtools/xlcpp/multicore/>.
[34] M. Honeyford, Speed Your Code with the GNU Profiler, <http://www.ibm.com/developerworks/library/l-gnuprof.html>.
[35] D.R. Butenhof, Programming with POSIX Threads, Addison-Wesley Professional Computing Series, 1997.
[36] IBM Corp. – PPU and SPU C/C++ Language Extension Specification, <http://www.01.ibm.com/chips/techlib/techlib.nsf/techdocs/30B3520C93F437AB87257060006FFE5E>.
[37] IBM Corp. – Cell SDK Example Library API Reference, <http://www.01.ibm.com/chips/techlib/techlib.nsf/techdocs/3B6ED257EE6235D900257353006E0F6A?Open&S_TACT=105AGX01&S_CMP=LP>.
[38] Intel Corporation – Intel(R) C++ Compiler Intrinsics Reference, <http://download.intel.com/support/performancetools/c/linux/v9/intref_cls.pdf>.
[39] S.K. Park, K.W. Miller, Random number generators: good ones are hard to find, Communications of the ACM 31 (10) (1988) 1192–1201.