CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1855

SPECIAL ISSUE PAPER

Targeting the Cell Broadband Engine for constraint-based local search

Daniel Diaz [1], Salvador Abreu [2,*,†] and Philippe Codognet [3]

[1] *University of Paris 1-Sorbonne, Paris, France*
[2] *Universidade de Évora and CENTRIA, Évora, Portugal*
[3] *JFLI, CNRS/UPMC/University of Tokyo, Tokyo, Japan*

SUMMARY

We investigated the use of the Cell Broadband Engine (Cell/BE) for constraint-based local search and combinatorial optimization applications. We presented a parallel version of a constraint-based local search algorithm that was chosen because it fits very well the Cell/BE architecture because it requires neither shared memory nor communication among processors. The performance study on several large optimization benchmarks shows mostly linear time speedups, sometimes even super linear. These experiments were carried out on a dual-Cell IBM (Armonk, NY, USA) blade with 16 processors. Besides getting speedups, the execution times exhibit a much smaller variance that benefits applications where a timely reply is critical. Copyright © 2011 John Wiley & Sons, Ltd.

Received 29 June 2011; Accepted 22 August 2011

KEY WORDS: constraint satisfaction; local search; Cell/BE; parallel programming

1. INTRODUCTION

The Cell Broadband Engine (Cell/BE) has proved suitable for graphical applications and scientific calculations [1], because of its innovative multicore architecture with its eight independent, specialized processing units. However, its ability to perform well for general-purpose applications is questionable, as it is very different from classical homogeneous multicore processors from Intel, AMD, or Sun (Niagara), or even from the IBM (Armonk, NY, USA) Power6 and Power7. In this paper, we investigate the use of Cell/BE for combinatorial optimization. This is a first step towards a large-scale implementation on a massively parallel architecture with reduced communication.

It is worth noticing that, in these domains, most of the attempts to take advantage the parallelism available in modern multicore architectures have targeted homogeneous systems, for instance, Intel or AMD-based machines, and make use of shared memory [2–4]. The different cores are working on shared data structures that somehow represent a global environment in which the subcomputations are taking place. Such an approach cannot be used for Cell-based machines, because heavy use of shared memory would degrade the overall performance of this particular multicore system. In order to extend the use of the Cell processor for combinatorial optimization and constraint-based problem solving, new approaches have to be investigated, in particular, those that can lead to independent subcomputations requiring little or no communication among processing units and limited or even no accesses to the main (shared) memory. We decided to focus on local search (LS) algorithms, also called 'metaheuristics', which have attracted much attention over the last decade from

*Correspondence to: Salvador Abreu, Universidade de Évora and CENTRIA, Évora, Portugal.
†E-mail: spa@di.uevora.pt

both the operations research and the artificial intelligence communities, in order to tackle very large combinatorial problems that are out of range for the classical exhaustive search methods. LS and metaheuristics have been used in combinatorial optimization for finding optimal or near-optimal solutions and have been applied to many different types of problems such as resource allocation, scheduling, packing, layout design, and frequency allocation.

We have developed a parallel extension of a constraint-based LS algorithm called adaptive search (AS) [5]. This metaheuristic is quite efficient, in practice, compared with classical propagation-based constraint solvers, especially for large problems. We implemented a parallel version of AS for Cell (AS/Cell). To assess the viability of this approach, we experimented AS/Cell on several classical benchmarks from CSPLib [6]. These structured problems are abstractions of real problems and therefore representative of real-life applications; they are classically used for benchmarking new methods. The results for the parallel AS method show a good behavior when scaling up the number of cores: speedups are practically linear, especially for large-scale problems, and sometimes we reach super linear speedups because the simultaneous exploration of different subparts of the search space may converge faster towards a solution.

Another interesting aspect is that all experiments show a better robustness of the results when compared with the sequential algorithm. Because LS makes use of randomness for the diversification of the search, execution times vary from one run to another. When benchmarking such methods, execution times have to be averaged over many runs (we take the average of 50 runs). Our implementation results show that with AS/Cell running on 16 cores, the difference between the minimum and maximum execution times, as well as the overall variance of the results, decreases significantly with respect to the reference sequential implementation. Execution times become more predictable: this is a clear advantage for real-time applications with bounded response time requirements.

The rest of this article is organized as follows. Section 2 presents Parallel LS. Section 3 discusses the AS algorithm, and its parallel version is presented in Section 4. A performance analysis is shown in Section 5. The robustness of the method is studied in Section 5.5. A short conclusion ends this paper.

2. PARALLEL LOCAL SEARCH

Parallel implementations of LS metaheuristics have been studied since the early 1990s, when multiprocessor machines started to become widely available; see [7] for a general survey and concepts or [8] for the basic parallel version of tabu search, simulated annealing, greedy randomized AS procedure, and genetic algorithms. With the availability of clusters in the early 2000s, this domain became active again [9, 10]. Apart from domain-decomposition methods and population-based method (such as genetic algorithms), one usually distinguishes between single-walk and multiple-walk methods for LS. Single-walk methods consist in using parallelism inside a single-search process, for example, for parallelizing the exploration of the neighborhood; see, for instance, [11] for such a method making use of graphics processing units for the parallel phase. Multiple-walk methods (also called multistart methods) consist in developing concurrent explorations of the search space, either independently or cooperatively, with some communication among concurrent processes. A key point is that independent multiple-walk methods are the most easy to implement on parallel computers and can lead to linear speedup if the solutions are uniformly distributed in the search space and if the method is able to diversify correctly [12]. Sophisticated cooperative strategies for multiple-walk methods can be devised by using solution pools [13] but requires shared memory or emulation of central memory in distributed clusters, impacting thus on performance.

In artificial intelligence, parallel implementation of search algorithms has a long history [14]. For constraint satisfaction problems (CSP), early work has been carried out in the context of distributed artificial intelligence and multiagent systems [15], but these methods cannot lead to efficient algorithms and cannot compete with good sequential implementations. Only very few implementations of efficient constraint solvers on parallel machines have been reported, for instance, [2], for a shared-memory architectures with eight-core CPUs. The Comet system [16] has been parallelized for small clusters of PCs, both for its LS solver [17] and its propagation-based constraint solver [18]. Recent experiments have been carried out with up to 12 processors [19], and speedups tend somehow to level after 10 processors. In the domain of SAT solving (Boolean satisfiability), most parallel implementations have targeted multicore architectures with shared memory [4, 20, 21]. Very recently, [22] extended a solver for PC cluster architectures by using a hierarchical shared memory model in order to minimize communication between independent machines.

3. THE ADAPTIVE SEARCH ALGORITHM

A CSP is defined as a triple (X, D, C), where X is a set of variables, D is a set of domains, that is, finite sets of possible values (one domain for each variable), and C is a set of constraints restricting the values that the variables can simultaneously take. Classical CSPs usually consider finite domains for the variables (integers or naturals) and solvers based on arc-consistency techniques originating from artificial intelligence research carried out in the 1970s and now generalized under the term of propagation-based methods [23, 24]. Such solvers keep an internal representation of variable domains in order to handle all types of constraints. However, during the last decade, the application of LS techniques in the CSP community has started to draw some interest [16, 25–27].

A generic, domain-independent LS method named AS was proposed in [5, 26]. AS is a metaheuristic that takes advantage of the structure of the problem to guide the search more precisely than a unique global cost function like the number of violated constraints. This method is generic, that is, can be applied to a large class of constraints (e.g., linear and nonlinear arithmetic constraints and symbolic constraints) and intrinsically copes with overconstrained problems. It can deal with satisfaction problems such as classical CSPs (i.e., searching for an assignment of the problem variables satisfying all constraints) or with optimization problems requiring the minimization of an objective function.

The input is a problem in CSP format, that is, a set of variables with their domains of possible values and a set of constraints over these variables. For each constraint, an 'error function' also needs to be defined; it will give, for each tuple of variable values, a quantification of how much the constraint is violated. Consider an n-ary constraint $c(X_1, \ldots, X_n)$ and associated variable domains $D_1, \ldots, D_n$. An error function f_c associated to the constraint c is a function from $D_1 \times \cdots \times D_n$ such that $f_c(X_1, \ldots, X_n)$ has a value zero if and only if $c(X_1, \ldots, X_n)$ is satisfied. The error function is used as a heuristic to represent the degree of satisfaction of a constraint and thus gives an indication on how much the constraint is violated. This idea has also been proposed independently by Galinier and Hao [25], where it is called 'penalty function', and then reused by the Comet System [16], where it is called 'violation'. This error function can be seen as (an approximation of) the distance of the current configuration to the closest satisfiable region of the constraint domain. For instance, the error function associated with an arithmetic constraint $X < c$, for a given constant $c \geqslant 0$, could be $\max(0, X - c)$. Finally a 'global cost function' must be provided to capture the distance of the current assignment (configuration) to a solution. The AS algorithm performs an iterative improvement, by trying to minimize this global cost. For pure CSP instances, this cost is zero if and only if a solution is found. For problems involving an objective function to minimize, the cost function will encompass both the satisfiability of the configuration and the objective function (e.g., by a simple sum, a weighted sum, or any other combination operator).

Adaptive search relies on *iterative repair*, based on variable and constraint errors, seeking to reduce the error on the worst variable. The basic idea is to compute the error function for each constraint and then combine, for each variable, the errors of all constraints in which it appears, thereby projecting constraint errors onto the relevant variables. Finally, the variable with the highest error will be designated as the 'culprit', and its value will be modified. In this second step, the *min-conflict* heuristic [28] is used to select the value in the variable domain that is most promising, that is, the value for which the global cost of the next configuration is minimal. To prevent being trapped in local minima, the AS method includes a short-term memory mechanism in the spirit of tabu search (variables can be marked tabu and 'frozen' for a number of iterations). It also integrates reset transitions to escape stagnation around local minima. A reset consists in assigning fresh random values to some variables (randomly chosen) and is guided by the number of variables being marked tabu. It is also possible to restart from scratch when the number of iterations becomes too large (this sort of 'reset over all variables' is guided by the number of iterations).

Algorithm 1 Adaptive Search Base Algorithm

Input: problem given in CSP format:	some tuning parameters:
• set of variables X_i with their domains	• TT: number of iterations a variable is frozen
• set of constraints C_j with error functions	• *RL*: number of frozen variables triggering a reset
• function to project constraint errors on vars	• *RP*: percentage of variables to reset
• (positive) cost function to minimize	• *MI*: maximal number of iterations before restart
	• *MR*: maximal number of restarts

Output: a solution if the CSP is satisfied or a quasi-solution of minimal cost otherwise.

```
1:  $Restart \leftarrow 0$
2:  repeat
3:      $Restart \leftarrow Restart + 1$
4:      $Iteration \leftarrow 0$
5:      Compute a random assignment $A$ of variables in $V$
6:      $Opt\_Sol \leftarrow A$
7:      $Opt\_Cost \leftarrow cost(A)$
8:      repeat
9:          $Iteration \leftarrow Iteration + 1$
10:         Compute errors of all constraints in $C$ and combine errors on each variable
11:                         ▷ (by considering only the constraints in which a variable appears)
12:         select the variable $X$ (not marked Tabu) with highest error
13:         evaluate costs of possible moves from $X$
14:         if no improvement move exists then
15:             mark $X$ as Tabu until iteration number: $Iteration + TT$
16:             if the number of variables marked Tabu $\geq RL$ then
17:                 randomly reset $RP$ % variables in $V$ (and unmark those Tabu)
18:             end if
19:         else
20:             select the best move and change $X$, yielding the next configuration $A'$
21:             if $cost(A') < Opt\_Cost$ then
22:                 $Opt\_Sol \leftarrow A \leftarrow A'$
23:                 $Opt\_Cost \leftarrow cost(A')$
24:             end if
25:         end if
26:     until $Opt\_Cost = 0$ (a solution is found) or $Iteration \geq MI$
27: until $Opt\_Cost = 0$ (a solution is found) or $Restart \geq MR$
28: $output(Opt\_Sol, Opt\_Cost)$
```

The core ideas of AS can be summarized as follows:

- To consider for each constraint a heuristic function that is able to compute an approximate degree of violation (the current ‘error’ on the constraint).
- To aggregate constraints on each variable and project the constraint errors on each variables. This will allow focus on the ‘worst’ variable and to try to repair it with the most promising value.
- To define a ‘global cost’ function to approximate the distance from the current configuration to a solution. This can be simply the sum of the absolute values of all constraint errors or a more complicated operator (e.g., weighted sum and sum of squares). This function will be used as a success criterion (i.e., the algorithm will iteratively try to minimize it). In case of optimization problems, the ‘global cost’ must include the objective function to minimize.
- To keep a short-term memory (tabu list) to avoid looping, together with a reset mechanism.

Adaptive search is a simple algorithm (Algorithm 1) but turns out to be quite effective in practice [5]. For example, on CSPs such as the *magic square problem* that will be detailed in Section 5, the AS algorithm of [5] is about 15–40 times faster than dialectic search [27], (which is about as fast as the initial AS algorithm from 2001 [26]) and about 100–400 times faster than a tabu search algorithm implemented in the Comet System [16] (see [27] and [5] for timings carried out on the same machine). Considering the complexity/efficiency ratio, this can be a very effective way to implement constraint solving techniques, especially for ‘anytime’ algorithms where (approximate) solutions have to be computed within a bounded amount of time.

In order to deal with optimization problems, the AS algorithm memorizes the best configuration (see variables (*Opt_Sol*, *Opt_Cost*) in Algorithm 1) each time an improvement is found. Observe that, for problems dealing with the satisfaction of CSP instances (for which only a solution is required), this bookkeeping is not necessary because a solution is found if and only if the global cost is zero. In that case, *Opt_Sol* will contain the last configuration computed and therefore does not need to be stored. However, for optimization problems or overconstrained CSPs, the returned pair (*Opt_Sol*, *Opt_Cost*) corresponds to the ‘best’ configuration found. In the case of optimization problems, we clearly cannot guarantee that the solution encountered is the optimum, as it happens with any incomplete method. However, a good modeling and parameter fine tuning can greatly improve the quality of the solution found. Current experiments on the quadratic assignment problem [29] provide already encouraging results and seems to support this claim.

4. A PARALLEL VERSION OF ADAPTIVE SEARCH ON CELL BROADBAND ENGINE

We now present AS/Cell: our implementation of the AS algorithm on the Cell/BE. Some features of the Cell/BE processor deserve mention because they strongly shape what applications stand a chance to succeed when ported on such architecture.

- It is a hybrid multicore architecture, with a general-purpose ‘controller’ processor (the Power Processor Element [PPE], a PowerPC instance) and eight specialized streaming single-instruction multiple-data (SIMD) processors (Synergistic Processing Elements or SPEs), as can be seen on Figure 1.
- The SPEs are connected via a very high-bandwidth internal bus, the element interconnect bus, with an aggregate peak performance of about 200 GB/s.
- The SPEs may only perform computations on their local store (256 kB for code and data).
- The SPEs may access the main memory (RAM in Figure 1) and each other’s local store via atomic direct memory access (DMA) operations.
- Moreover, two Cell/BE processor chips may be linked via the input/output controllers, on a single system board, to appear as a multiprocessor with 16 SPEs, a feature that we exploited in our performance analysis.

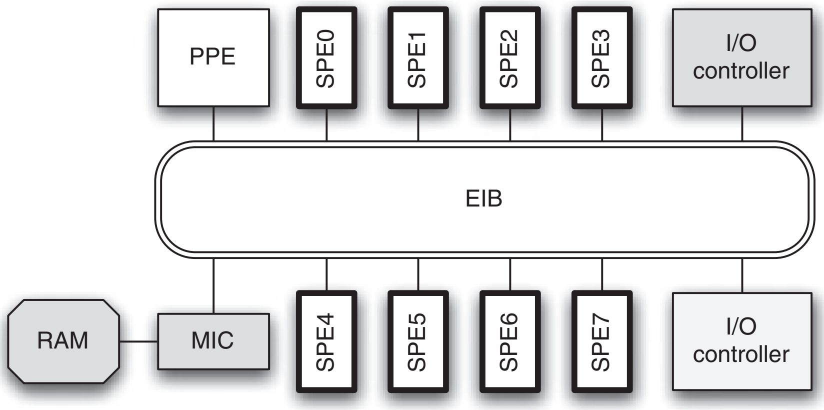

Figure 1. The Cell Broadband Engine processor organization. PPE, Power Processor Element; SPE, Synergistic Processing Element; I/O, input/output; EIB, element interconnect bus; MIC, memory interface controller.

The interested reader may refer to the IBM Redbook [1] for further information, as well as for the performance and capacity trade-offs that affect Cell/BE programs. The AS algorithm seems to *a priori* fit the requirements fairly well, justifying thus our choice.

The basic idea in extending AS for parallel implementation is to have several distinct parallel search engines for exploring simultaneously different parts of the search space, thus giving an independent multiple-search version of the sequential AS. This is very natural to achieve with the AS algorithm: one just needs to start each engine with a different, randomly computed, initial configuration, that is, a different assignment of values to variables. Subsequently, each 'AS engine' can perform the sequential algorithm independently, and, as soon as one processor finds a solution or when all have reached the maximum number of allowed iterations, all processors halt, and the algorithm finishes. The Cell/BE processor architecture is mapped onto the task structure, with a controller system thread on the PPE and a worker thread on each SPE.

- The system processor (a PPE) obtains the present real time $T0$, launches a given number of threads (SPEs),[‡] each with an identical SPE context, and then waits for a solution.
- Each SPE starts with a random configuration (held entirely in its local storage) and improves it step by step, applying Algorithm 1.
- As soon as one SPE finds a solution, it stores it in the main memory (using a DMA operation) and informs the PPE.
- The PPE then instructs all other SPEs to stop and waits until all SPEs have performed so (join). After that, it gets the real time $T1$ and provides both the solution and the execution time $T = T1 - T0$ (this is the real elapsed time since the beginning of the program until the join including SPEs initialization and termination).

This simple parallel version seems feasible and does not require complex data structures (e.g., shared tables). Note that SPEs do not communicate, only doing so with the PPE upon termination: each SPE works blindly on its configuration until it reaches a solution or fails. We plan to further extend this algorithm with communication among SPEs of some information about partial solutions but always with the aim of limiting data communication as much as possible.

It remains to be seen whether the space limitations of the Cell/BE processor, in particular, the size of the SPE local stores, are not crippling in terms of problem applicability. It turns out we managed to fit both the program and the data in the 256-kB local store of each SPE, even for large benchmarks. This was possible for two reasons: the relative simplicity of the AS algorithm and the compactness of the encoding of combinatorial search problems as a CSP, that is, variables with finite domains and many predefined constraints, including arithmetic ones. This is especially relevant when compared, for instance, with SAT encodings where only Boolean variables can be used, and each constraint has to be explicitly decomposed into a set of Boolean formulas, thereby yielding problem formulations that easily reach several hundred thousand literals.

In short, the AS method is very thrifty in its resource requirements, which makes it a good candidate for running on Cell/BE: *little data and long computations*.

5. PERFORMANCE EVALUATION

We now present and discuss our assessment of the performance of the implementation of AS/Cell. The code running on each SPE is a port of that used in [5], an implementation of AS specialized for CSP permutation problems. It should be noted that no code optimizations have been made to benefit from the peculiarities of Cell/BE (namely SIMD vectorization, loop unfolding and parallelization, and branch removal). Our measurements, made on the IBM system simulator, lead us to believe it is reasonable to expect a significant speedup when these aspects are taken into account, as the SPEs are frequently stalled when executing the present code. In order to assess the performance of AS/Cell, we use a pertinent set of classical benchmarks from CSPLib [6] consisting of the following:

[‡]It may be the case that the *second* Cell/BE processor in the dual-CPU blade may be involved in setting up the SPE thread: this happens for the ninth and subsequent threads.

- `all-interval`: the all-interval series problem (`prob007` in CSPLib).
- `partit`: the number partitioning problem (`prob049` in CSPLib).
- `perfect-square`: the perfect-square placement problem (`prob009` in CSPLib).
- `magic-square`: the magic square problem (`prob019` in CSPLib).

Although these are academic benchmarks, they are abstractions of real-world problems and representative of actual combinatorial optimization applications. These benchmarks involve a very large combinatorial search space, for example, the 100×100 magic square requires 10,000 variables whose domains range over 10,000 values. No polynomial-time solution is known for any of the problems we are addressing as benchmarks. The search spaces we are pushing AS into are so large as to be off limits for most complete, propagation-based constraint solvers, which are bound to do an exhaustive search, even when resorting to heuristics to speed up the process. It is also worth noting that, here, we tackle pure CSP instances and thus each solution returned by AS is exact (i.e., the global cost is zero if and only if a solution is found).

The rest of this section evaluates the performances of AS/Cell on the benchmarks. For each problem, we provide a short description and its AS modeling. We then analyze the behavior of AS/Cell when both the size of the problem and the number of used SPEs vary. The experiment has been executed on an IBM QS21 dual-Cell blade system (times are measured in seconds). Because AS uses random configurations and progression, we sampled each benchmark 50 times. The displayed execution time is the average of the 50 runs after removing both the lowest and the highest times.

5.1. *The All-Interval series problem*

This problem is described as `prob007` in the CSPLib. This benchmark is in fact a well-known exercise in music [30] where the goal is to compose a sequence of N notes such that all are different and tonal intervals between consecutive notes are also distinct. This problem is equivalent to finding a permutation of the N first integers such that the absolute difference between two consecutive pairs of numbers are all different. This amounts to finding a permutation $(X1, \dots, X_N)$ of $\{0, \dots, N-1\}$ such that the list $(\mathrm{abs}(X_1 - X_2), \mathrm{abs}(X_2 - X_3), \dots, \mathrm{abs}(X_{N-1} - x_N))$ is a permutation of $1, \dots, N-1$. A possible solution for $N = 8$ is $(3, 6, 0, 7, 2, 4, 5, 1)$ because all consecutive distances are different:

$$3\,{}^{\underline{3}}\,6\,{}^{\underline{6}}\,0\,{}^{\underline{7}}\,7\,{}^{\underline{5}}\,2\,{}^{\underline{2}}\,4\,{}^{\underline{1}}\,5\,{}^{\underline{4}}\,1.$$

Adaptive search for Cell modeling only maintains the list of N variables X_i and ensures it forms a permutation by swapping values inside the list. It is worth noticing that the constraint on the distances (absolute values between each $(X_i - X_{i+1})$ is not encoded as a data structure but is ensured via the cost function (this further reduces the amount of data in the local storage). The cost function of a configuration is the largest missing distance (largest distances are the hardest to place and thus should be privileged). Obviously, a solution is found when this cost is zero.

Table I presents the average time of 50 runs for several instances of this benchmark together with the speedup obtained when using different numbers of SPEs. From this data, one can conclude that the speedup increases linearly with the number of SPEs to reach about 11 with 16 SPEs. The speedup appears to be constant whatever the size of the problem.

5.2. *Number partitioning*

This problem is described as `prob049` in the CSPLib and consists in finding a partition of numbers $\{1, \dots, N\}$ into two groups A and B of the same cardinality such that the sum of numbers in A is equal to the sum of numbers in B and the sum of squares of numbers in A is equal to the sum of squares of numbers in B. A solution for $N = 8$ is $A = (1, 4, 6, 7)$ and $B = (2, 3, 5, 8)$.

Table I. Timings (s) and speedups for all-interval series.

		Speedup with k SPEs					
Problem instance	Time 1 SPE	2	4	8	12	16	Time 16 SPEs
`all-interval 100`	1.392	1.6	3.3	5.0	5.7	**7.4**	0.189
`all-interval 150`	9.496	2.3	4.4	6.3	9.0	**10.4**	0.910
`all-interval 200`	28.165	1.5	3.0	6.1	7.8	**9.0**	3.139
`all-interval 250`	61.437	1.8	3.8	5.1	6.5	**9.8**	6.282
`all-interval 300`	147.178	1.7	2.9	5.6	7.3	**9.2**	15.920
`all-interval 350`	346.790	2.3	4.4	5.6	9.6	**12.2**	28.359
`all-interval 400`	508.819	1.6	3.3	7.6	8.8	**10.8**	46.989
`all-interval 450`	946.860	2.0	4.1	8.7	9.2	**11.0**	85.936

SPE, Synergistic Processing Element.

The problem is modeled with N variables $V_i \in \{1, \ldots, N\}$ that form a permutation of $\{1, \ldots, N\}$. The first $N/2$ variables form the group A, and the $N/2$ last variables the group B. There are two constraints:

$$\Sigma_{i=1}^{N/2} V_i \;=\; \Sigma_{i=N/2+1}^{N} V_i \;=\; N(N+1)/4,$$

$$\Sigma_{i=1}^{N/2} V_i^2 \;=\; \Sigma_{i=N/2+1}^{N} V_i^2 \;=\; N(N+1)(2N+1)/12.$$

The possible moves from one configuration consist in all possible swaps exchanging one value in the first subset with another one in the second subset. The errors on the two equality constraints are computed as the absolute value of the difference between the actual sum and the expected constant (e.g., $N(N+1)/4$). In this problem, as for the `all-interval` example, all variables play the same role, and there is no need to project errors on variables. The global cost of a configuration is the sum of the absolute values of both constraint errors. A solution is found when the global cost is zero.

Table II details the average runtime (in seconds) for several instances of this problem together with the speedup obtained when using different numbers of SPEs. Similarly to what occurred with the all-interval series, the speedup increases linearly up to a factor of 11. Again, the speedup appears to be independent from the size of the problem.

5.3. *The perfect-square placement problem*

This problem is cataloged as `prob009` in CSPLib. The perfect square placement problem (also called the *squared square problem* [31]) is to pack a set of squares with given integer sizes into a bigger square of given integer size in such a way that no squares overlap each other and all square

Table II. Timings (s) and speedups for number partitioning.

		Speedup with k SPEs					
Problem instance	Time 1 SPE	2	4	8	12	16	Time 16 SPEs
`partit 1400`	6.227	2.7	3.7	6.0	7.2	**11.2**	0.556
`partit 1600`	7.328	1.8	3.4	6.0	7.5	**10.1**	0.727
`partit 1800`	11.559	2.0	3.7	6.4	9.4	**10.9**	1.062
`partit 2000`	13.802	1.7	3.1	6.1	9.5	**10.6**	1.303
`partit 2200`	18.702	2.3	3.5	6.2	10.0	**10.8**	1.735
`partit 2400`	21.757	2.1	3.3	5.5	7.1	**10.2**	2.129
`partit 2600`	29.890	1.8	3.8	6.9	8.6	**11.0**	2.716

SPE, Synergistic Processing Element.

borders are parallel to the border of the big square (the term ‘perfect’ means all squares have different sizes). The sum of the square surfaces is equal to the surface of the packing square, so that there is no spare capacity. The smallest solution involves 21 squares that must be packed into a master square of size 112 (see figure).

Because the implementation we are basing our work on (AS) only deals with permutation problems, we have modeled this problem as a set of N variables whose values correspond to the sizes of the squares to be placed in order—this is not the best modeling but complies with the requirements of the available implementation. Each square in a configuration is placed in the lowest and leftmost possible slot. The aforementioned solution corresponds to the configuration $(33, 37, 42, 29, 4, 25, 16, 18, 24, 9, 7, 2, 17, 6, 50, 15, 11, 19, 35, 8, 27)$.

Moving from a configuration to another consists in swapping two variables. For the cost of a configuration to be computed, the squares are packed as explained earlier. As soon as a square does not fit in the lowest/leftmost slot, the placement stops. The cost of the configuration is a formula depending on the set of unplaced squares (number of unplaced squares and size of the largest) and on the remaining slots in the master square (sum of heights, largest height, and sum of widths). As usual, a configuration is a solution when its cost drops to zero.

We tried five different instances of this problem, taken from CSPLib and [32] whose input data are summarized in Table III.

Table IV presents the data associated with the average case for these instances. Running 16 SPEs, the speedup ranges from 14 to 16, depending on the instance.

5.4. *Magic squares*

The magic square problem is listed as `prob019` in CSPLib and consists in placing the numbers $\{1, 2, \ldots, N^2\}$ on an $N \times N$ square such that each row, column, and main diagonal equal the same sum (the constant $N(N^2 + 1)/2$). For instance, here is a well-known solution for $N = 4$ (depicted by Albrecht Dürer in his engraving *Melancholia I*, 1514).

The modeling for AS/Cell involves N^2 variables $X_1, \ldots, X_{N^2}$. The error function of an equation $X_1 + X_2 + \ldots + X_k = b$ is defined as the value of $X_1 + X_2 + \cdots + X_k - b$. The combination operation is the sum of the absolute values of the errors. The global cost function is the addition of absolute values of the errors of all constraints. A configuration with zero cost is a solution.

Table III. Perfect-square instance data.

		Squares to place	
Problem instance	Master square size	Number	Largest
`perfect-square 1`	112 × 112	21	50 × 50
`perfect-square 2`	228 × 228	23	99 × 99
`perfect-square 3`	326 × 326	24	142 × 142
`perfect-square 4`	479 × 479	24	175 × 175
`perfect-square 5`	524 × 524	25	220 × 220

Table IV. Timings (s) and speedups for perfect square.

		Speedup with k SPEs					
Problem instance	Time 1 SPE	2	4	8	12	16	Time 16 SPEs
`perfect-square 1`	14.844	1.9	4.9	8.1	11.3	**16.6**	0.894
`perfect-square 2`	30.395	1.5	4.4	6.7	10.0	**14.4**	2.105
`perfect-square 3`	55.973	1.6	2.9	6.5	12.7	**14.1**	3.963
`perfect-square 4`	75.915	1.8	3.0	5.4	9.3	**15.4**	4.933
`perfect-square 5`	143.436	2.1	3.7	6.7	10.7	**15.1**	9.517

SPE, Synergistic Processing Element.

Table V. Timings (s) and speedups for magic squares.

		Speedup with k SPEs					
Problem instance	Time 1 SPE	2	4	8	12	16	Time 16 SPEs
`magic-square 30`	0.855	2.2	3.3	4.4	5.9	**6.9**	0.125
`magic-square 40`	2.496	2.0	3.6	5.7	6.1	**7.4**	0.335
`magic-square 50`	3.903	1.8	2.5	3.8	5.2	**5.6**	0.702
`magic-square 60`	9.834	2.2	3.8	5.6	7.2	**6.8**	1.441
`magic-square 70`	17.571	2.2	3.4	4.8	6.6	**8.5**	2.065
`magic-square 80`	31.889	3.0	4.3	5.8	7.6	**8.6**	3.689
`magic-square 90`	57.746	2.9	3.8	7.2	9.3	**10.8**	5.323
`magic-square 100`	189.957	5.9	9.3	13.9	21.9	**22.6**	8.387

SPE, Synergistic Processing Element.

Table V details the average running times (in seconds) for several instances of this problem together with the speedup obtained when using different numbers of SPEs. With the use of 16 SPEs, the obtained speedup increases with the size of the problem to reach 22 for the largest instance.

5.5. *Performance evaluation summary*

The aforementioned performance evaluation has shown that the AS method is a good match for the Cell/BE architecture, proving its suitability for effectively solving highly combinatorial problems. All the problems we tested clearly benefited from using several SPEs. For three of the problems, the ultimate speedup obtained with 16 SPEs seems constant whatever the size of the problem, which is very promising. Moreover, for the *magic square* benchmark, the speedup tends to increase as the problem becomes larger, which is also a very useful property. Figure 2 provides a global view of the measured speedups, as applied to the hardest instances of each problem. We observe that increasing the number of SPEs always results in a speed increase, which is one of the basic goals we had for this implementation.

Concerning raw performance, it is worth noticing that AS/Cell performs very well even on difficult problems. As a case study, consider the magic square problem (one of the most challenging problem: most solvers cannot even handle instances larger than 20×20). With one SPE, the average time to solve magic square 100×100 is around 3 min. Using 16 SPEs, AS/Cell drastically reduces the computation time, bringing it to barely over 8 s. Moreover, these results are obtained with a

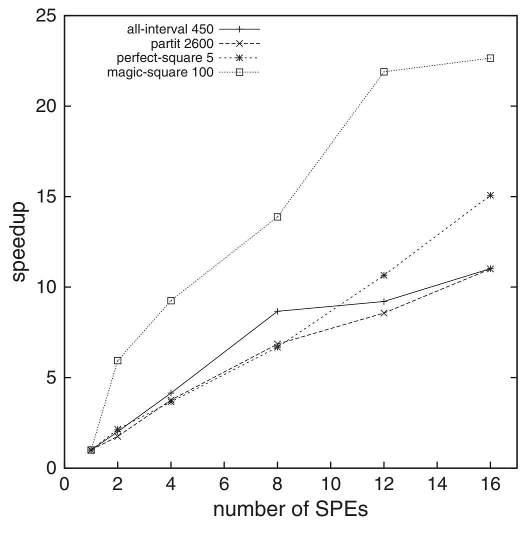

Figure 2. Evolution of the speedup depending on the number of Synergistic Processing Elements (SPEs).

straightforward port of the AS sequential code, not yet specialized to benefit from the potential of Cell/BE.

We performed some initial code profiling and datapath-level simulations to check the low-level performance and effective hardware utilization. At the SPE level, our first port yields a CPI (cycles per instruction) of `2.05`, which puts us at about one-third of the performance that we can expect to attain. Approximately half the cycles in SPE executions are spent in stalls, of which about 60% are branch misses whereas the remaining 40 % are spent on dependency stalls. Again, these results are not surprising, because we did a direct port of the existing, sequential code. This leads us to expect a significant performance increase from execution analysis and code reorganization. We also plan on experimenting with SIMD vectorization, loop unrolling, branch removal, and other known techniques to further improve performance.

Our first assessment clearly shows that the AS method is a good match for the Cell/BE architecture: this processor is definitely effective in solving nontrivial highly combinatorial problems.

6. ROBUSTNESS EVALUATION

In the previous section, each benchmark was run 50 times, and the average time was taken, leaving the extreme values out. This is a classical approach and gives us a precise idea of the general behavior of AS/Cell. In this section, we study the degree of variation between the various runs and what influence do AS/Cell runtime parameters have. One way to measure dispersion is to consider, for each instance, the longest execution time among the 50 runs. This is interesting for situations such as real-time applications because it represents the ‘worst case’ one can encounter: too high a value can even preclude its use in time-critical situations. Figure 3 depicts the graph of the 50 executions for `all-interval 450` both with 1 and 16 SPEs—we only show this particular benchmark, but a similar study exists for all the other problems, and they all display similar characteristics. This graph clearly exposes the difference, in terms of runtime dispersion, using between only 1 SPE and all 16 SPEs.

Table VI summarizes the results, focusing on the worst case—meaning we only took the highest time across all runs. We also only show the values for the hardest instance of each problem, so as to unclutter the charts. This evaluation uncovers a significant improvement: the obtained speedup is always better than the one obtained in the average case. For instance, with 16 SPEs, the worst case is improved by a factor 26 for `all-interval 450` and by a factor 500 for `magic-square 100` (the corresponding average time speedups are 11 and 22.6). Clearly, AS/Cell greatly narrows the range of possible execution times for a given problem.

Another way to measure this is to consider the standard deviation: Table VII charts the evolution of the standard deviation of the execution times for the hardest instance of each problem, varying the number of SPEs. The standard deviation rapidly decreases as more SPEs are used. For instance,

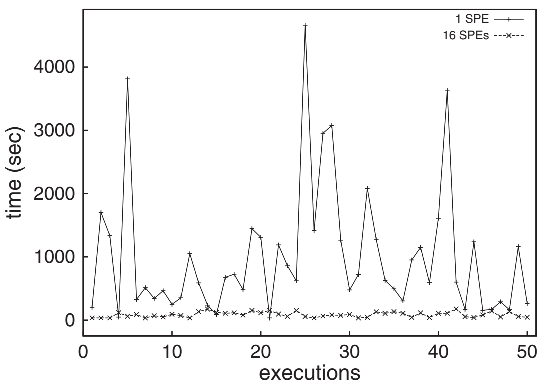

Figure 3. Dispersion analysis for the 50 runs of `all-interval 450`. SPE, Synergistic Processing Element.

Table VI. Timings (s) and speedups for the worst case (max of 50 runs).

		Speedup with k SPEs					
Problem instance	Time 1 SPE	2	4	8	12	16	Time 16 SPEs
`all-interval 450`	4661.870	2.4	4.7	15.3	10.5	**26.3**	177.210
`partit 2600`	105.030	1.2	3.6	7.2	11.9	**17.1**	6.160
`perfect-square 5`	456.470	2.1	3.9	7.1	9.7	**16.6**	27.550
`magic-square 100`	9013.330	62.9	152.3	154.5	534.6	**505.5**	17.830

SPE, Synergistic Processing Element.

Table VII. Standard deviation for the worst case.

	Standard deviation with k SPEs					
Problem instance	1	2	4	8	12	16
`all-interval 450`	891	460	224	65	61	40
`partit 2600`	24	16	7	3	2	2
`perfect-square 5`	122	54	27	16	10	6
`magic-square 100`	916	19	13	10	3	4

SPE, Synergistic Processing Element.

if we take magic square 100 × 100, the standard deviation decreases from 916 to less than 4. This amounts to a dramatic performance improvement for the worst-case scenario. Take magic square 100 × 100. With one SPE, execution takes 2.5 h, at worst, to complete. When using 16 SPEs, this drops to 18 s.

AS/Cell limits the dispersion of the execution times: we say that the multicore version is *more robust* than the sequential one. The execution time is more predictable from one run to another in the multicore version, and more cores means more robustness. This is a crucial usability feature for real-time systems or even for some interactive applications such as games. To further test this idea, we tried a slight variation of the method that consists in starting all SPEs with the *same* initial configuration (instead of a random one). Each SPE then diverges according to its own internal random choices. The results of this experiment show that the overall behavior is practically the same as the original method, only a little slower: on the average less than 10%. This slowdown was to be expected because the search has less diversity to start with and therefore might take longer to explore a search space that contains a solution. However, this limited slowdown shows that the method is intrinsically robust, can restore diversification, and again, takes advantage of the parallel search in an efficient manner.

7. CONCLUSION

We presented a simple yet effective parallel adaptation of the AS algorithm to the Cell/BE architecture to solve combinatorial problems. We chose to target the Cell/BE because of its promise of high performance and, in particular, to experiment with its heterogeneous architecture, which departs significantly from most multicore architectures with shared memory. The implementation drove us to minimize communication of data among processors and between a processor and the main memory. We view this experiment as the first step towards a large-scale implementation on a massively parallel architecture, where communication costs are at a premium and should be eschewed.

The experimental evaluation we carried out on a dual-CPU blade with 16 SPE cores indicates that linear speedups can be expected in most cases, and even some situations of super linear speedups are possible. Scaling the problem size seems never to degrade the speedups, even when dealing with very difficult problems. We even ran a reputedly very hard benchmark with increasing speedups when the problem size grows. An important, if somewhat unexpected, fringe benefit is that the worst-case execution time becomes even higher speedups than the average case. This characteristic opens up several domains of application to the use of combinatorial search problem formulations:

this is particularly true of real-time applications and other time-sensitive uses, for instance, interactive games.

Recall that AS is an 'anytime' method: it is always possible to interrupt the execution and obtain the best pseudosolution computed so far. Regarding the issue, this method can easily benefit from an architecture such as Cell/BE: when running several SPEs in parallel, the PPE simply queries each SPE to obtain its best pseudosolution (together with the corresponding cost) and then chooses the best of these best. Another good property of the Cell/BE organization is that the only data an SPE needs to pass, when copying, is the current configuration (a small array of integers) and its associated cost, which can be carried out very efficiently.

Our early results clearly demonstrate that heterogeneous architectures with internal parallelism, such as Cell/BE, have a significant potential to make good on solving combinatorial problems. Concerning further development, we plan to work along two directions: to optimize the Cell/BE code following the recommendations of [1] and to experiment with several forms of communication among the processors involved in a computation. The extension of the algorithm beyond a single dual-Cell blade is also in our plans, in particular, to extend our approach with cluster communication mechanisms, such as message passing interface or the remote DMA-based global address space programming interface[33].

ACKNOWLEDGEMENTS

The equipment used in this work was provided as a grant from the IBM Corporation Shared University Research (SUR) program awarded to CENTRIA and U. of Évora. Acknowledgements are due to the FCT/Egide Pessoa grant 'CONTEMP—CONTraintes Executées sur MultiProcesseurs'.

REFERENCES

1. Redbooks IBM. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. Vervante, 2008.
2. Perron L. Search procedures and parallelism in constraint programming. In *proceedings of CP'99*. Springer Verlag, 1999; 346–360.
3. Holzmann GJ, Bosnacki D. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering* 2007; **33**(10):659–674.
4. Hamadi Y, Jabbour S, Sais L. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation* 2009; **6**:245–262.
5. Codognet P, Diaz D. An efficient library for solving CSP with local search. In *MIC'03, 5th International Conference on Metaheuristics*, Ibaraki T (ed.), 2003.
6. Gent IP, Toby W. CSPLIB: a benchmark library for constraints. In *CP*, 1999; 480–481.
7. Verhoeven M, Aarts E. Parallel local search. *Journal of Heuristics* 1995; **1**(1):43–65.
8. Pardalos PM, Pitsoulis LS, Mavridou TD, Resende MGC. Parallel search for combinatorial optimization: genetic algorithms, simulated annealing, tabu search and grasp. In *proceedings IRREGULAR'95*, 1995; 317–331.
9. Crainic TG, Toulouse M. Special issue on parallel meta-heuristics. *Journal of Heuristics* 2002; **8**(3):247–388.
10. Alba E. Special issue on new advances on parallel meta-heuristics for complex problems. *Journal of Heuristics* 2004; **10**(3):239–380.
11. Luong TV, Melad N, Talbi E-G. Parallel local search on GPU. *Technical Report RR 6915*, INRIA, Lille, France, 2009.
12. Aiex RM, Resende MGC, Ribeiro CC. Probability distribution of solution time in GRASP: an experimental investigation. *Journal of Heuristics* 2002; **8**(3):343–373.
13. Crainic TG, Gendreau M, Hansen P, Mladenovic N. Cooperative parallel variable neighborhood search for the -median. *Journal of Heuristics* 2004; **1**(3):293–314.
14. Kergommeaux JCD, Codognet P. Parallel logic programming systems. *ACM Computing Surveys* 1994; **26**(3):295–336.
15. Yokoo M, Durfee EH, Ishida T, Kuwabara K. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering* 1998; **10**(5):673–685.
16. Hentenryck PV, Michel L. *Constraint-Based Local Search*. The MIT Press, 2005.
17. Michel L, See A, Hentenryck PV. Distributed constraint-based local search. In *proceedings of CP'06, 12th International Conference on Principles and Practice of Constraint Programming*, Vol. 4204, Benhamou F (ed.), Lecture Notes in Computer Science. Springer, 2006; 344–358.
18. Michel L, See A, Hentenryck PV. Parallelizing constraint programs transparently. In *proceedings of CP'07*, Bessiere C (ed.). Springer Verlag, 2007; 514–528.

19. Michel L, See A, Hentenryck PV. Parallel and distribited local search in Comet. *Computers and Operations Research* 2009; **36**:2357–2375.
20. Chu G, Stuckey P. A parallelization of MiniSAT 2.0. In *Proceedings of SAT Race*, 2008.
21. Schubert T, Lewis MDT, Bernd B. PaMiraXT: parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation* 2009; **6**:203–222.
22. Ohmura K, Ueda K. c-sat: a parallel SAT solver for clusters. In *Proceedings of SAT'09*. Springer Verlag, 2009; 524–537.
23. Hentenryck PV. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.
24. Bessiere C. Constraint propagation. In *Handbook of Constraint Programming*, Rossi F, Beek PV, Walsh T (eds). Elsevier, 2006; 29–83.
25. Galinier P, Hao J-K. A general approach for constraint solving by local search. In *2nd Workshop CP-AI-OR'00*, Paderborn, Germany, 2000.
26. Codognet P, Diaz D. Yet another local search method for constraint solving. In *Proceedings of SAGA'01*. Springer Verlag, 2001; 73–90.
27. Kadioglu S, Sellmann M. Dialectic search. In *CP 2009, International Conference on Principles and Practice of Constraint Programming*. Springer Verlag: Lisbon, Portugal, 2009.
28. Minton S, Johnston MD, Philips AB, Laird P. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence* 1992; **58**(1-3):161–205.
29. Koopmans TC, Beckmann M. Assignment problems and the location of economic activities. *Econometrica: Journal of the Econometric Society* 1957; **25**(1):53–76.
30. Truchet C, Codognet P. Musical constraint satisfaction problems solved with adaptive search. *Soft Computing - A Fusion of Foundations, Methodologies and Applications* 2004; **8**(9):633–640.
31. van Lint JH, Wilson RM. *A Course in Combinatorics*. Cambridge University Press, 1992.
32. Bouwkamp CJ, Duijvestijn AJW. Catalogue of simple perfect squared squares of orders 21 through 25. *Technical Report Technical Report 92 WSK 03*, University of Technology, Department of Mathematics and Computer Science, Eindhoven, The Netherlands, 1992.
33. Machado R, Lojewski C. The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. *Computer Science-Research and Development* 2009; **23**(3):125–132.