Parallel local search for solving Constraint Problems on the Cell Broadband Engine (Preliminary Results)

Salvador Abreu
Universidade de Évora and CENTRIA FCT/UNL
Portugal
spa@di.uevora.pt

Daniel Diaz
University of Paris 1-Sorbonne
France
Daniel.Diaz@univ-paris1.fr

Philippe Codognet
JFLI, CNRS / UPMC / University of Tokyo,
Japan
Philippe.Codognet@lip6.fr

We explore the use of the Cell Broadband Engine (Cell/BE for short) for combinatorial optimization applications: we present a parallel version of a constraint-based local search algorithm that has been implemented on a multiprocessor BladeCenter machine with twin Cell/BE processors (total of 16 SPUs per blade). This algorithm was chosen because it fits very well the Cell/BE architecture and requires neither shared memory nor communication between processors, while retaining a compact memory footprint. We study the performance on several large optimization benchmarks and show that this achieves mostly linear time speedups, even sometimes super-linear. This is possible because the parallel implementation might explore simultaneously different parts of the search space and therefore converge faster towards the best sub-space and thus towards a solution. Besides getting speedups, the resulting times exhibit a much smaller variance, which benefits applications where a timely reply is critical.

1 Introduction

The Cell processor has shown its power for graphic and server applications, and more recently has been considered as a good candidate for scientific calculations [15]. Its floating point arithmetic performance and energy efficiency make it useful as a basic block for building super-computers, cf. the “Roadrunner” machine based on Cell processors which is currently the fastest supercomputer. However, its ability to perform well for general-purpose applications has been questioned, and Cell programming has always been considered as very challenging. We investigate in this paper the use of the Cell/BE for combinatorial optimization applications and constraint-based problem solving. It is worth noticing that in these domains most of the attempts to take advantage of the parallelism available in modern multi-core architectures have targeted homogeneous systems, for instance Intel or AMD-based machines and make use of shared memory, e.g. [14, 8, 7]. The different cores are working on shared data-structures which 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 multi-core 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 between processing units and limited or even no accesses to the main (shared) memory. We decided to focus on Local Search algorithms, also called “metaheuristics”, which have attracted much attention over the last decade from both the Operations Research and the Artificial Intelligence communities, in order to tackle very large combinatorial problems which are out of range for the classical exhaustive search methods. Local search

Submitted to:
LSCS’09

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, frequency allocation, etc.

To enable the use of the Cell/BE for combinatorial optimization applications, we have developed a parallel extension of a constraint-based local search algorithm based on a method called “Adaptive Search” which was proposed a few years ago in [4, 5].

To assess the viability of this approach, we experimented on several classical benchmarks of the constraint programming community from CSPlib [6]. These structured problems are somehow abstractions of real problems and are therefore representative of real-life applications; they are classically used in the community for benchmarking new methods. The preliminary implementation results for the parallel Adaptive Search method show a good behavior when scaling up the number of cores (from one to sixteen): speedups are, most of the time, practically linear, especially for large-scale problems and our experiments even exhibit a few super-linear speedups because the simultaneous exploration of different subparts of the search space may converge faster towards a solution.

Another interesting point to mention is that all experiments show a better robustness of the results on the multi-core version when compared to the sequential algorithm, as will be explained below. Because local search methods make use of randomness for the diversification of the search, execution times may vary from one run to another. This is why, when benchmarking such methods, execution times have to be averaged on many runs (in our experiments, we always take the average of 50 runs). Our implementation results show that for a parallel version running on 16 cores, the difference between the minimal and maximal execution times, as well as the overall variance of the results, decreases significantly with respect to the reference sequential implementation. The main result of this is that execution times become more predictable and this is, of course, an advantage in real-time applications with bounded response time requirements.

The remainder of this article is organized as follows: after an introduction, section 3 discusses the Adaptive Search algorithm and its parallel version is presented in section 4. We proceed with a performance analysis in section 5, which is analyzed and commented on in section 6. Finally, we conclude in section 7 and present our lines for related future research.

2 Parallelizing Constraint Solvers

Parallel implementation of search algorithms has a long history, especially in the domain of logic programming, see [9] for an overview. Most of the proposed implementations are based on the so-called OR-parallelism, splitting the search space between different processors but making used of a shared or duplicated stack for coping with the adequate execution environment and they rely on a Shared Memory Multiprocessor (SMP) architecture for the parallel execution support. For some years, similar techniques have been used for Model Checkers, which are used as verification tools for hardware and software, such as the SPIN software [1, 8]. These implementations are also based on some kind of OR-parallelism and, again, these approaches are well-suited for multi-core architecture with shared memory. More recently, there have been several initiatives to extend SAT solvers for parallel machines, in particular multi-cores [7, 3, 16]. However these frameworks require a shared memory model and will thus not be scalable to massively parallel machines or architectures for which communication through (distributed) shared memory is costly such as a cluster system or for hetherogeneous multicore processors such as the Cell/BE. It is worth noticing that now some authors are also extending SAT solvers to PC cluster architectures [13], using a hierarchical shared memory and trying to minimize communication between clusters. While moving from traditional SMP machines to multi-core systems is a relatively straightforward change, it is not necessarily so for more exotic architectures, such as the hetherogeneous multicore chips which include the Cell/BE.

For Constraint Satisfaction Problems, early work has been done in the context of Distributed Artificial Intelligence and multi-agent systems, see for instance [18], but these methods, even if interesting from a theoretical point of view, cannot lead to efficient algorithms and cannot compete with good sequential implementations. Moreover, the focus is usually not on performance but on the formulating a problem in a distributed fashion. Only very few implementations of efficient constraint solvers on parallel machines have been reported, most notably [14], which again is aimed at shared-memory architectures and recently [11] which proposes a distribted extendion of the Comet local search solver for clusters of PCs.

3 The Adaptive Search Algorithm

Over the last decade, the application of local search techniques for constraint solving in general (and not only for combinatorial optimization) has started to draw some interest in the CSP community. A generic, domain-independent local search method named Adaptive Search was proposed by [4, 5], a new meta-heuristic that takes advantage of the structure of the problem in terms of constraints and variables and can guide the search more precisely than a global cost function to optimize (such as for instance the number of violated constraints). The algorithm also uses an short-term adaptive memory in the spirit of Tabu Search in order to prevent stagnation in local minima and loops. This method is generic, can be applied to a large class of constraints (e.g. linear and non-linear arithmetic constraints, symbolic constraints, etc) and naturally copes with over-constrained problems. The input of the method is a problem in CSP format, that is, a set of variables with their (finite) domains of possible values and a set of constraints over these variables. For each constraint, an "error function" needs to be defined; it will give, for each tuple of variable values, an indication of how much the constraint is violated. For instance, the error function associated with an arithmetic constraint $|X - Y| < c$, for a given constant $c \geq 0$, can be $max(0, |X - Y| - c)$. Adaptive Search relies on iterative repair, based on variable and constraint error information, seeking to reduce the error on the worst variable so far. The basic idea is to compute the error function for each constraint, 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 well known min-conflict heuristic [12] is used to select the value in the variable domain which is the most promising, that is, the value for which the total error in the next configuration is minimal. In order to prevent being trapped in local minima, the Adaptive Search method also includes a short-term memory mechanism to store configurations to avoid (variables can be marked Tabu and "frozen" for a number of iterations), and also integrates restart-based transitions to escape stagnation around local minima. Restarts are partial and are guided by the number of variables being marked Tabu. The core ideas of adaptive search can be summarized as follows:

- to consider for each constraint a heuristic function that is able to compute an approximated degree of satisfaction of the goals (the current "error" on the constraint);
- to aggregate constraints on each variable and project the error on variables thus trying to repair the "worst" variable with the most promising value;
- to keep a short-term memory of bad configurations to avoid looping (i.e. some sort of "tabu list").

Algorithm

Consider an n-ary constraint $c(X_1, \cdots X_n)$ and associated variable domains $D_1, \cdots D_n$. An error function f_c associated to the constraint c is a real-valued function from $D_1 \times \cdots \times D_n$ such that $f_c(X_1, \cdots X_n)$ has value zero if $c(X_1, \cdots X_n)$ is satisfied. The error function will in fact be used as a heuristic value to represent the degree of satisfaction of a constraint and will thus give an indication on how much the constraint is violated. This is very similar to the notion of "penalty functions" used in continuous global optimization. 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. Since the error is only used to heuristically guide the search, we can use any approximation when the exact distance is difficult (or even impossible) to compute.

Input

Problem given in CSP format:
- a set of variables $V = \{V_1, V_2, ..., V_n\}$ with associated domains of values
- a set of constraints $C = \{C_1, C_2, ..., C_k\}$ with associated error functions
- a combination function to project constraint errors on variables
- a (positive) cost function to minimize

Some tuning parameters:
- T : Tabu tenure (number of iterations a variable is frozen)
- RL : reset limit (number of frozen variables triggering a reset)
- RP : reset percentage (percentage of variables to reset)
- Max_I : maximal number of iterations before restart
- Max_R : maximal number of restarts

Output

A solution (configuration where all constraints are satisfied) if the CSP is satisfied or to a quasi-solution of minimal cost otherwise.

Algorithm

Iteration = 0
Restart = 0
Repeat
  Restart = Restart + 1
  Iteration = Iteration +1
  Tabu_Nb = 0
  Compute a random assignment A of variables in V
  Opt_Sol = A
  Opt_Cost = cost(A)
  Repeat
    1. Compute errors of all constraints in C
       and combine errors on each variable
       (by considering only the constraints in which a variable appears)
    2. select the variable X (not marked Tabu) with highest error

3. evaluate costs of possible moves from X
4. **if** no improvment move exists
   then mark X as Tabu until iteration number: Iteration + T
      Tabu_Nb = Tabu_Nb + 1
      if Tabu_Nb > RL
      then randomly reset RP variables in V
         (and unmark those which are Tabu)
   else select the best move and change the value of X
      accordingly to produce next configuration A'
      if cost(A') < Opt_Cost
      then Opt_Sol = A'
         Opt_Cost = cost(A')

until a solution is found or Iteration > Max_I
until a solution is found or Restart > Max_R
output (Opt_Sol, Opt_Cost)

Adaptive Search is a simple algorithm but it turns out to be quite efficient in practice [5]. Considering the complexity/efficiency ratio, it can be a very effective way to implement constraint solving techniques in larger software tools, especially for anytime algorithms where (approximate) solutions have to be computed within a limited amount of time.

4 Parallel Algorithm on the Cell/BE

We will not present the Cell/BE processor architecture here, some features of this architecture, however, deserve mention because they strongly shape what applications may succeed when ported:

- A hybrid multicore architecture, with a general-purpose "controller" processor (the PPE, a PowerPC instance) and eight specialized processors (SPEs.)
- Two Cell/BE processor chips may be linked to appear as a multiprocessor with 16 SPEs.
- The PPEs are connected via a very high-bandwidth internal bus, the EIB.
- The PPEs may only perform operations on their local store, which contains both code and data and is limited to 256KB.
- The PPEs may access system memory and each other's private memory by means of DMA operations.

The interested reader can refer to the IBM Redbook [15] for further information on this architecture, as well as the performance and capacity tradeoffs which affect Cell/BE programs.

The basic idea in extending the algorithm for parallel implementation is to have several distinct parallel search engines for exploring simultaneously different parts of the search space, and to start each such engine a a different processor. This is very natural to achieve with the Adaptive Search 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 "Adaptive Search engine" can perform the sequential algorithm described in the previous section independently. As soon as one process finds a solution, or when all processors reach the maximal number of iterations allowed, all processors are halted and the algorithm finishes with the condition that led to the termination (solution found or maximum iterations reached in all processors).

The Parallel Algorithm

The Cell/BE processor architecture is reflected on the task structure, in which a controller thread resides in the PPE and each SPE has a worker thread.

- The PPE gets the real time $T0$, launches a given number of threads, each with an identical SPU 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 the algorithm of section 3.
- As soon as an SPE finds a solution, it sends it to the main memory (using a DMA operation) and informs the PPE.
- The PPE then propagates this information to all other SPEs to stop their job and waits until all SPUs have finished (join). After that, it gets the real time $T1$ and provides both the solution and the execution time [1] $T = T1 - T0$.

It is worth pointing out that SPEs do not communicate among themselves and only do so with the PPE upon termination: each SPE can work blindly on its configuration until it reaches an outcome (solution or failure). Indeed, we managed to fit both the program and the data in the 256KB of local store of each SPU, even for admittedly large benchmarks. This turns out to be possible for two reasons: (1) the simplicity and compactness of the algorithm, (2) the compactness of the encoding of the combinatorial problem as a CSP, that is, variables with finite domains and many predefined constraints, including arithmetics. This is especially true when compared, for instance, to a SAT encoding where only boolean variables can be used and each constraint has to be explicitly decomposed into a set of boolean formulas yielding problem formulations which easily reach several thousands of literals.

To summarize, the Adaptive Search method requirements are a good match for the Cell/BE architecture: *not much data but a lot of computation*.

5 Performance Evaluation

We now present and discuss the performance of our implementation of AS/Cell. The code running on each SPU is derived from the code used in [4, 5] which is an implementation of the Adaptive Search for permutation problems. It is worth noting that no code specialization has been made to benefit from the full potential of the Cell processor (namely vectorization, branch removing, ...) It is reasonable to expect a significant speedup when these aspects are taken into account.

Since the Adaptive Search uses random configurations and progression, each benchmark has been executed 50 times. There are two interesting ways for aggregating those results: considering the average case (average of 50 executions times after removing both the lowest and the highest times) and considering the worst case (maximum of the 50 executions). On one hand, the former is classical and gives a precise idea of the behavior of the AS/Cell. On the other hand, the latter is also interesting for real-time applications since it represents the “worst-case” one can encounter (too high a value can even prevent the use in time-critical applications). Interestingly, AS/Cell improves both cases, achieving linear speedups and sometimes even super-linear speedups.

[1] the execution time is then the real elapsed time since the beginning of the program until the join (thus including SPUs initialization and termination phases).

5.1 The All-Interval series

Although looking like a pure combinatorial search problem, this benchmark is in fact a well-known exercise in music composition [17]. The idea is to compose a sequence of N notes such that all are different and tonal intervals between consecutive notes are also distinct (see Figure 1).

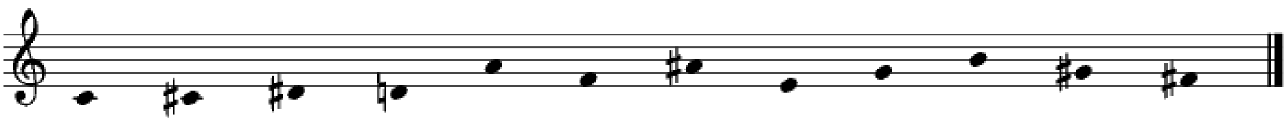

Figure 1: an example of all-interval in music

This problem is described as `prob007` in the CSPLib [6]. It 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 $(X1, \ldots X_N)$ of $\{0, \ldots N-1\}$ such that the list $(abs(X_1 - X_2), abs(X_2 - X_3) \ldots abs(X_{N-1} - x_N))$ is a permutation of $1, \ldots, N-1$.

Table 1 presents the average time of 50 running (in seconds) for several instances of this benchmark together with the speedup obtained when using different number of SPUs. From this data one can conclude the speedup linearly increases with the number of SPUs to reach 11 with 16 SPUs. This factor appears to be constant whatever the size of problem.

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

Table 1: timings (sec) and speedups for all-interval series

It is worth noticing that state-of-art constraint solvers (e.g. Gecode) are able to find the trivial solution $(0, N-1, 1, N-2, 2, N-3, \ldots)$ in a reasonable amount of time but fail to find an interesting solution for $N \geq 20$. The AS/Cell implementation is able to find solutions for $N = 450$ in 1.5 minute with 16 SPUs. Table 2 detail this instance providing information on both the average case and the worst case (together with the associated speedups). In this problem, linear speedups are obtained: with 16 SPUs the average time is 11 times faster. When discussing worst cases, the time is divided by a factor 26.

5.2 Number partitioning

This problem consists in finding a partition of numbers $\{1, \ldots N\}$ into two groups A and B such that:

- A and B have the same cardinality

#SPUs	average case		worst case	
	time (sec)	speedup	time (sec)	speedup
1	946.860	1.0	4661.870	1.0
2	470.421	2.0	1912.200	2.4
4	228.294	4.1	988.220	4.7
8	109.322	8.7	304.160	15.3
12	102.831	9.2	443.550	10.5
16	85.936	11.0	177.210	26.3

Table 2: average and worst times for all-interval 450

- the sum of numbers in A is equal to the sum of numbers in B
- 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 $B = (2,3,5,8)$ since:

$$1+4+6+7 = 18 = 2+3+5+8$$
$$1^2+4^2+6^2+7^2 = 102 = 2^2+3^2+5^2+8^2$$

This problem admits a solution iff N is a multiple of 8 and is modeled with N variables $V_i \in \{1 \ldots N\}$ which form a permutation of $\{1 \ldots N\}$. The first $N/2$ variables form the group A, the $N/2$ last variables the group B. There are two constraints:

$$\Sigma_{i=1}^{N/2} V_i \;=\; N(N+1)/4 \;=\; \Sigma_{i=N/2+1}^{N} V_i$$
$$\Sigma_{i=1}^{N/2} V_i^2 \;=\; N(N+1)(2N+1)/12 \;=\; \Sigma_{i=N/2+1}^{N} V_i^2$$

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 2 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, like for the all-intervals example, all variables play the same role and there is no need to project errors on variables. The total cost of a configuration is the sum of the absolute values of both constraint errors. A solution is found when the total cost is zero.

Table 3 details the average running times (in seconds) for several instances of this problem together with the speedup obtained when using different numbers of SPUs. Similarly to what occured 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.

Once more, it is worth noticing that Constraint Programming systems such as GNU Prolog cannot solve this problem for instances larger than 128. On the other hand the AS/Cell implementation is able to find solutions for $N = 2600$ in few seconds with 16 SPUs: this problem scales very well and it is possible to solve even larger instances. Table 4 details the largest instance both for the average case and the worst case (together with the associated speedups). For this example the speedups are linear: with 16 SPUs the average time is divided by 11 while the worst case is divided by 17.

5.3 The Perfect-Square placement problem

This problem is described as `prob009` in CSPLib [6]. It is also called the *squared square* problem [10] and consists in packing a set of squares into a master square in such a way that no squares overlap each other. All squares have different sizes and they fully cover the master square (there is no spare

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

Table 3: timings (sec) and speedups for number partitioning

#SPUs	average case		worst case	
	time (sec)	speedup	time (sec)	speedup
1	29.890	1.0	105.030	1.0
2	17.071	1.8	84.750	1.2
4	7.941	3.8	29.540	3.6
8	4.362	6.9	14.590	7.2
12	3.490	8.6	8.830	11.9
16	2.716	11.0	6.160	17.1

Table 4: average and worst times for partit 2600

capacity). The smallest solution involves 21 squares which must be packed into a master square of size 112.

Since the system we are basing our work on (Adaptive Search) only deals with permutation problems, we have modeled this problem as a set of N variables whose values corresponds 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.

Moving from a configuration to another consists in swapping 2 variables. To compute the cost of a configuration, the squares are packed as explained above. 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 several criteria on the set of non placed squares (number of non-placed squares and the size of the biggest) and on remaining slots in the master square (sum of heights, largest height, sum of widths). As usual, a configuration is a solution when its cost drops to zero.

We tried 5 different instances of this problem taken from [6, 2] whose input data are summarized in table 5. Table 6 presents the data associated to the average case for these instances. Running 16 SPUs, the speedup ranges from 11 to 16 depending on the instance.

As previously explained, our modeling is not the best one: a modeling explicitely using variables to encode X and Y coordinates of each square would be clearly better as done in a Constraint Programming modeling. Nevertheless, AS/Cell performs rather well and the most difficult instance (number 5) is solved in less than 10 seconds with 16 SPUs. Table 7 provides more information for this instance both for the average case and the worst case (together with the associated speedups). In this problem linear speedups are obtained: with 16 SPUs, both the average and worst case times are about 16 times

problem instance	master square size	squares to place	
		number	largest
1	112 × 112	21	50 × 50
2	228 × 228	23	99 × 99
3	326 × 326	24	142 × 142
4	479 × 479	24	175 × 175
5	524 × 524	25	220 × 220

Table 5: perfect-square instances

size	time 1 SPU	speedup with k SPUs					time 16 SPUs
		2	4	8	12	16	
1	14.844	1.9	4.9	8.1	11.3	16.6	0.894
2	30.395	1.5	4.4	6.7	10.0	14.4	2.105
3	55.973	1.6	2.9	6.5	12.7	14.1	3.963
4	75.915	1.8	3.0	5.4	9.3	15.4	4.933
5	143.436	2.1	3.7	6.7	10.7	15.1	9.517

Table 6: timings (sec) and speedups for perfect square

lower.

#SPUs	average case		worst case	
	time (sec)	speedup	time (sec)	speedup
1	143.436	1.0	456.470	1.0
2	66.775	2.1	217.330	2.1
4	39.180	3.7	117.410	3.9
8	21.481	6.7	64.330	7.1
12	13.467	10.7	47.170	9.7
16	9.517	15.1	27.550	16.6

Table 7: average and worst times for perfect square #5

5.4 Magic squares

The magic square problem is listed as `prob019` in CSPLib [6] and consists in placing the numbers $\{1,2\cdots N^2\}$ on an $N \times N$ square, such that the sum of the numbers in all rows, columns and the two diagonal is the same. The constant value that should be the sum of all rows, columns and the two diagonals can be easily computed to be $N(N^2+1)/2$.

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 $X_1+X_2+\ldots+X_k-b$. The combination operation is the sum of the absolute values of the errors. The overall cost function is the addition of absolute values of the errors of all constraints. A configuration with zero cost is a solution.

Table 8 details the average running times (in seconds) for several instances of this problem together with the speedup obtained when using different numbers of SPUs. Using 16 SPUs, the obtained speedup increases with the size of the problem to reach 22 for the largest instance.

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

Table 8: timings (sec) and speedups for magic squares

It is worth noticing that this benchmarks is one of the most challenging: Constraint programming systems such as GNU-Prolog or ILOG Solver perform poorly on this benchmark and cannot solve instances greater than 10×10. On the other hand AS/Cell is able to solve 100×100 in only few seconds with 16 SPUs. Table 9 details the largest instance both for the average case and the worst case with associated speedups. For this example the speedups are super-linear: with 16 SPUs the average time is divided by 22 while the worst case is divided by 500!

#SPUs	average case		worst case	
	time (sec)	speedup	time (sec)	speedup
1	189.957	1.0	9013.330	1.0
2	31.975	5.9	143.270	62.9
4	20.532	9.3	59.170	152.3
8	13.686	13.9	58.350	154.5
12	8.677	21.9	16.860	534.6
16	8.387	22.6	17.830	505.5

Table 9: average and worst times for magic squares 100×100

6 Analysis: Performance and Robustness

The performance evaluation of section 5 has shown that the Adaptive Search method is a good match for the Cell/BE architecture. This processor is clearly a serious candidate to effectively solve highly combinatorial problems. All problems tested were accelerated when using several SPUs. For 3 of the problems the ultimate speedup obtained with 16 SPUs seems constant whatever the size of the problem which is very promising. Moreover, for magic squares the speedup tends to increase as the problem becomes more difficult which is also a very interesting property.

This evaluation has also uncovered an even more significant improvment on the worst case: the obtained speedup is always better than the one obtained in the average case. Like this, AS/Cell greatly narrows the range of possible execution times for a given problem. Figure 3 depicts the graph of the 50 executions for the all-interval 450 benchmark, both with 1 and 16 SPUs (due to space limitation we only show this one but a similar graph exists for all other problems). This graph clearly reveals the

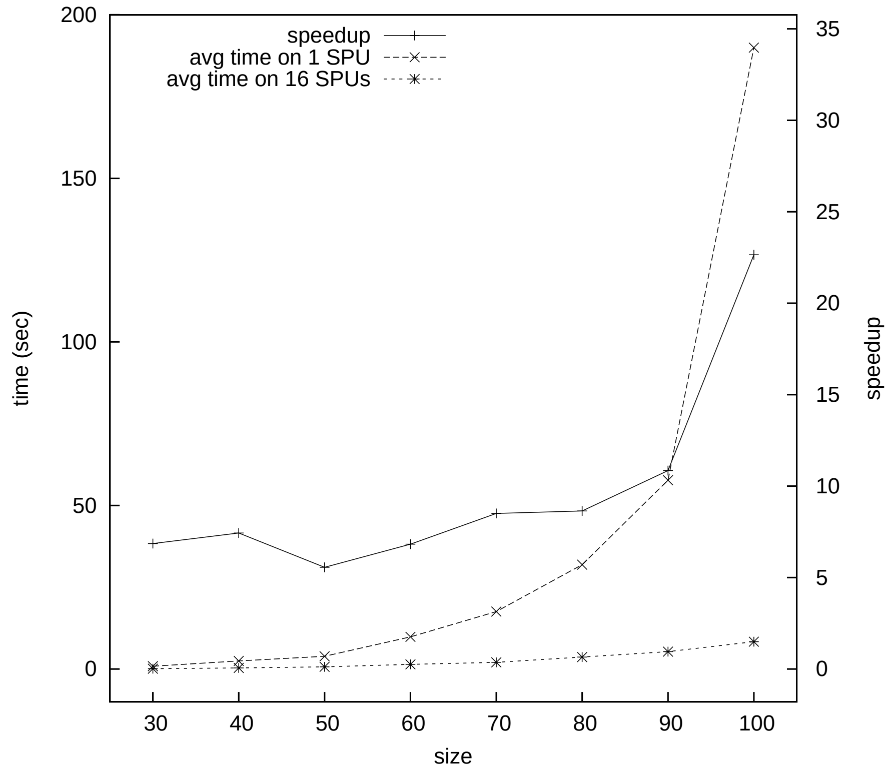

Figure 2: average time with 1 and 16 SPUs for magic squares

difference of dispersion when using 1 SPU or 16 SPUs. Table 10 charts the evolution of the standard deviation of the execution times for the largest instance of each problem depending on the number of SPUs. The standard deviation rapidly decreases when more SPUs are used (the most spectacular case being magic square 100×100 where the standard deviation decreases from 915.7 to 3.8). AS/Cell limits the dispersion of the execution times. We can say that the multicore version is more robust than the sequential one in the sense that the difference between the minimum and maximum execution times, as well as the overall variance of the results, decreases significantly. Therefore, the execution time is more predictable from one run to another in the multicore version, and more cores means more robustness. This is crucial for real-time systems or even some interactive applications.

SPUs	all interval 450	number partit 2600	perfect square 5	magic square 100
1	891.2	24.0	122.0	915.7
2	459.6	16.0	54.4	18.5
4	223.9	6.9	26.6	12.7
8	65.4	2.8	16.3	10.0
12	61.0	1.9	10.4	3.4
16	40.0	1.5	6.3	3.8

Table 10: evolution of the standard deviation (50 execution times)

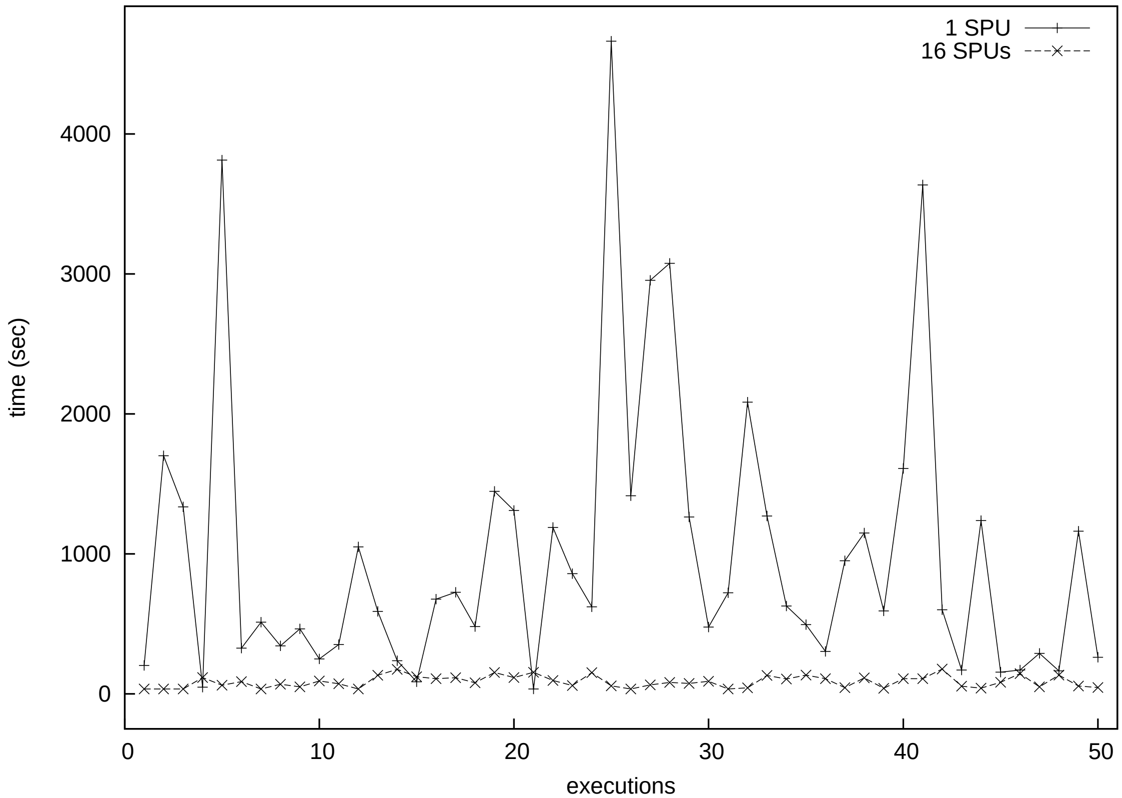

Figure 3: dispersion analysis for the 50 runs of all-interval 450

An interesting experiment can be made to further develop this idea. We experimented with a slight variation of the method which consists in starting all parallel processes with the *same* initial configuration (instead of a random one). Each SPU will then diverge according to its own internal random choices. We implemented this variant and the results show that the overall behavior is practically the same as the original method, just a bit slower on average by about 10%. This slowdown was to be expected because in this case the search has less diversity to start with, and therefore might take longer to explore a portion of the search space that contains a solution. However the fact that this slowdown is only 10% shows that the method is intrinsically quite robust, can restore diversification and take again advantage of the parallel search in a quite efficient manner.

Finally, it is worth noticing that Adaptive Search is an "anytime" method: it is always possible to interrupt the execution and to obtain the best pseudo-solution computed so far. On this point too this method can benefit easily from the Cell: when running several SPUs in parallel, the PPE simply has to ask each SPU to obtain its best pseudo-solution (together with the corresponding cost) and then to chose the best of these bests. Indeed, another good property regarding the Cell features, is the fact that the only data a SPU needs to pass is the current configuration (an array of integers) and the associated cost.

7 Concluding Remarks

We presented a simple yet effective initial port of the Adaptive Search algorithm to the Cell/BE architecture, which we used to solve combinatorial search problems. The experimental evaluation we carried out indicates that linear speedups are to be expected in most cases, and even some situations of superlinear 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 gets 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 usages, for instance interactive games.

Clearly the Cell/BE has a very significant potential to make good on combinatorial search problems. We plan to work on two separate directions: on one hand, to optimize the code as per the IBM guidelines [15] and on the other, to experiment with more sophisticated organizations and forms of communication among the processors involved in a computation.

Acknowledgements

The equipment used to perform the benchmarks described herein was provided by IBM Corporation, as a grant from the Shared University Research (SUR) program awarded to CENTRIA and U. of Évora.

References

[1] Jiri Barnat, Lubos Brim & Petr Rockai (2007): *Scalable Multi-core LTL Model-Checking*. In: *SPIN*. pp. 187–203. Available at `http://dx.doi.org/10.1007/978-3-540-73370-6_13`.

[2] C. J. Bouwkamp & A. J. W. Duijvestijn (1992): *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.

[3] Geoffrey Chu & Peter Stuckey (2008): *A parallelization of MiniSAT 2.0*. In: *SAT race*.

[4] Philippe Codognet & Daniel Diaz (2001): *Yet Another Local Search Method for Constraint Solving*. In: *SAGA*. pp. 73–90. Available at `http://link.springer.de/link/service/series/0558/bibs/2264/22640073.htm`.

[5] Philippe Codognet & Daniel Diaz (2003): *An Efficient Library for Solving CSP with Local Search*. In: T. Ibaraki, editor: *MIC'03, 5th International Conference on Metaheuristics*. `http://pauillac.inria.fr/~diaz/adaptive/`.

[6] Ian P. Gent & Toby Walsh (1999): *CSPLIB: A Benchmark Library for Constraints*. In: *CP*. pp. 480–481. `http://www.csplib.org`.

[7] Youssef Hamadi, Saïd Jabbour & Lakhdar Sais (2009): *ManySAT: a Parallel SAT Solver*. *Journal on Satisfiability, Boolean Modeling and Computation* 6, pp. 245–262.

[8] Gerard J. Holzmann & Dragan Bosnacki (2007): *The Design of a Multicore Extension of the SPIN Model Checker*. *IEEE Transactions on Software Engineering* 33(10), pp. 659–674. Available at `http://doi.ieeecomputersociety.org/10.1109/TSE.2007.70724`.

[9] Jacques Chassin de Kergommeaux & Philippe Codognet (1994): *Parallel Logic Programming Systems*. *ACM Computing Surveys* 26(3), pp. 295–336.

[10] J. H. van Lint & R.M. Wilson (1992): *A Course in Combinatorics*. Cambridge University Press.

[11] Laurent Michel, Andrew See & Pascal Van Hentenryck (2006): *Distributed Constraint-Based Local Search*. In: Frédéric Benhamou, editor: *proceedings of CP'06, 12th Int. Conf. on Principles and Practice of Constraint Programming*, *Lecture Notes in Computer Science* 4204. Springer, pp. 344–358.

[12] Steven Minton, Mark D. Johnston, Andrew B. Philips & Philip Laird (1992): *Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems*. *Artif. Intell.* 58(1-3), pp. 161–205.

[13] Kei Ohmura & Kazunori Ueda (2009): *c-sat: A Parallel SAT Solver for Clusters*. In: *SAT*. Springer Verlag, pp. 524–537. Available at `http://dx.doi.org/10.1007/978-3-642-02777-2_47`.

[14] Laurent Perron (1999): *Search Procedures and Parallelism in Constraint Programming*. In: *CP*. pp. 346–360.

[15] IBM Redbooks (2008): *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. Vervante.

[16] Tobias Schubert, Matthew D. T. Lewis & Bernd Becker (2009): *PaMiraXT: Parallel SAT Solving with Threads and Message Passing*. *Journal on Satisfiability, Boolean Modeling and Computation* 6, pp. 203–222.

[17] Charlotte Truchet & Philippe Codognet (2004): *Musical constraint satisfaction problems solved with adaptive search*. *Soft Comput.* 8(9), pp. 633–640. Available at `http://dx.doi.org/10.1007/s00500-004-0389-0`.

[18] Makoto Yokoo, Edmund H. Durfee, Toru Ishida & Kazuhiro Kuwabara (1998): *The Distributed Constraint Satisfaction Problem: Formalization and Algorithms*. *IEEE Transactions on Knowledge and Data Engineering* 10(5), pp. 673–685. Available at `db/journals/tkde/YokooDIK98.html`.