Parallel Constraint-based Local Search on the Cell/BE Multicore Architecture

Daniel Diaz, Salvador Abreu, and Philippe Codognet

Abstract We investigate the use of the Cell Broadband Engine (Cell/BE) for Combinatorial Optimization applications. We present a parallel version of a constraint-based Local Search algorithm which was chosen because it fits very well the Cell/BE architecture since it requires neither shared memory nor communication between processors. The performance study on several large optimization benchmarks shows mostly linear time speedups, even sometimes super-linear. These experiments were done on a Dual-Cell IBM Blade with 16 processors. Besides getting speedups, the execution times exhibit a much smaller variance, which benefits applications where a timely reply is critical.

1 Introduction

The Cell Broadband Engine (Cell/BE) has proved its suitability for graphical applications and scientific calculations [17], due to an innovative multicore architecture with its 8 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 IBM's Power6 and 7. 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.

We have developed a parallel extension of a constraint-based Local Search algorithm called Adaptive Search (AS) [4]. This metaheuristic is quite efficient in prac-

Daniel Diaz
University of Paris 1-Sorbonne, France, e-mail: Daniel.Diaz@univ-paris1.fr

Salvador Abreu
Universidade de Évora and CENTRIA FCT/UNL, Portugal, e-mail: spa@di.uevora.pt

Philippe Codognet
JFLI, CNRS / UPMC / University of Tokyo, Japan, e-mail: Philippe.Codognet@lip6.fr

tice, compared to classical propagation-based constraint solvers, especially for large problems. We implemented a parallel version of Adaptive Search for Cell (AS/Cell). To assess the viability of this approach, we experimented AS/Cell on several classical benchmarks from CSPLib [8]. 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 Adaptive Search 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 to the sequential algorithm. Because Local Search (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 w.r.t. 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 Local Search. Section 3 discusses the Adaptive Search 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 6. A short conclusion ends this paper.

2 Parallel Local Search

Parallel implementation of local search metaheuristics have been studied since the early 90's, when multiprocessor machines started to become widely available, see [19] for a general survey and concepts, or [15] for basic parallel version of Tabu search, simulated annealing, GRASP and genetic algorithms. With the availability of clusters in the early 2000's, this domain became active again [5, 2]. Apart from domain-decomposition methods and population-based method (such as genetic algorithms), one usually distinguish between single-walk and multiple-walk methods for LS. Single-walk methods consist in using parallelism inside a single search process, e.g. for parallelizing the exploration of the neighborhood, see for instance [10] for such a method making use of GPUs for the parallel phase. Multiple-walk methods (also called multi-start methods) consist in developing concurrent explorations of the search space, either independently or cooperatively with some communication between 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 speed-up if the solutions are uniformly distributed in the search space and if the method is able to diversify correctly [1]. Sophisticated cooperative strategies

for multiple-walk methods can be devised by using solution pools [6], but requires shared-memory or emulation of central memory in distributed clusters, impacting thus on performances.

In Artificial Intelligence, parallel implementation of search algorithms has a long history [7]. For Constraint Satisfaction Problems (CSP), early work has been done in the context of Distributed Artificial Intelligence and multi-agent systems [20], 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, most notably [16] for shared-memory architectures and recently [12] who proposes a distributed extension of the Comet LS solver for clusters of PCs. In the domain of SAT solving (boolean satisfiability), most parallel implementations have targeted multi-core architectures with shared memory [9, 3, 18]. Very recently, [14] extended a solver for PC cluster architectures by using a hierarchical shared memory model in order to to minimize communication between independent machines.

3 The Adaptive Search Algorithm

Over the last decade, the application of LS techniques in the CSP community has started to draw some interest. A generic, domain-independent LS method named Adaptive Search (AS) was proposed in [4]. AS is a meta-heuristic 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, can be applied to a large class of constraints (e.g. linear and non-linear arithmetic constraints, symbolic constraints...) and intrinsically copes with over-constrained problems. 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” needs to be defined; it will give, for each tuple of variable values, a quantification of how much the constraint is violated. For instance, the error function associated with an arithmetic constraint $X < c$, for a given constant $c \geq 0$, could be $max(0, X - c)$. 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 [13] 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. To prevent being trapped in local minima, the Adaptive Search 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).

For instance, 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 function from $D_1 \times \cdots \times D_n$ such that $f_c(X_1, \cdots X_n)$ has value zero iff $c(X_1, \cdots X_n)$ 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 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. Adaptive Search is a simple algorithm (see Algorithm 1) but it turns out to be quite effective in practice [4]. 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.

4 A Parallel Version of Adaptive Search on Cell/BE

We now present AS/Cell: our implementation of the Adaptive Search algorithm on the Cell/BE. We do not detail the Cell/BE processor here, but some features deserve mention since they strongly shape what applications stand a chance to succeed when ported:

- A hybrid multicore architecture, with a general-purpose "controller" processor (the PPE, a PowerPC instance) and eight specialized streaming SIMD processors ("Synergistics Processing Elements" or SPEs).
- Two Cell/BE processor chips may be linked on a single system board to appear as a multiprocessor with 16 SPEs.
- The SPEs are connected via a very high-bandwidth internal bus, the EIB.
- The SPEs may only perform computations on their local store (256KB for code+data).
- The SPEs may access main memory and each other's local store via DMA operations.

The interested reader can refer to the IBM Redbook [17] for further information, as well as the performance and capacity tradeoffs which affect Cell/BE programs. A-priori, the Adaptive Search algorithm seems to fit the requirements fairly well, justifying 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, giving thus an independent multiple-search version of the sequential

Algorithm 1 Adaptive Search Base Algorithm

Input: problem given in CSP format:
- set of variables X_i with their domains
- set of constraints C_j with error functions
- function to project constraint errors on vars
- (positive) cost function to minimize

some tuning parameters:
- TT: number of iterations a variable is frozen
- RL: number of frozen variables triggering a reset
- RP: percentage of variables to reset
- 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 ← 0
2:  repeat
3:      Restart ← Restart + 1
4:      Iteration ← Tabu_Nb ← 0
5:      Compute a random assignment A of variables in V
6:      Opt_Sol ← A
7:      Opt_Cost ← cost(A)
8:      repeat
9:          Iteration ← 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:             Tabu_Nb ← Tabu_Nb + 1
17:             if Tabu_Nb ≥ RL then
18:                 randomly reset RP % variables in V (and unmark those Tabu)
19:             end if
20:         else
21:             select the best move and change X, yielding the next configuration A′
22:             if cost(A′) < Opt_Cost then
23:                 Opt_Sol ← A ← A′
24:                 Opt_Cost ← cost(A′)
25:             end if
26:         end if
27:     until a solution is found or Iteration ≥ MI
28: until a solution is found or Restart ≥ MR
29: output(Opt_Sol, Opt_Cost)
```

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 thread on the PPE and a worker thread on each SPE.

1. The PPE gets the real time $T0$, launches a given number of threads (SPEs), each with an identical SPE context, and then waits for a solution.
2. 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.

3. As soon as an SPE finds a solution, it sends it to main memory (using a DMA operation) and informs the PPE.
4. The PPE then instructs all other SPEs to stop, and waits until all SPEs have done 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 between SPEs of some information about partial solutions, but always with the aim of limiting data communication as much as possible. It remained 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. We managed to fit both the program and the data in the 256KB of local store of each SPE, even for large benchmarks. This turned out 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, 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 hundred thousand literals.

In short, the Adaptive Search method requirements make a good match for the Cell/BE architecture: *little data but a lot of computation*.

5 Performance Evaluation

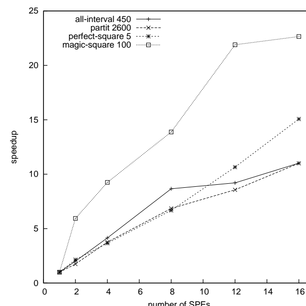
We now present and discuss our assessment of the performance of the initial implementation of AS/Cell. The code running on each SPE is a port of that used in [4], an implementation of Adaptive Search specialized for permutation problems. It should be noted that no code optimizations have been made to benefit from the peculiarities of Cell/BE (namely SIMD vectorization, loop unfolding and parallelization, branch removal...). Our initial 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 [8] consisting of:

- `all-interval`: the All-Interval Series problem (`prob007` in CSPLib).
- `perfect-square`: the Perfect-Square placement problem (`prob009` in CSPLib).
- `partit`: the Number Partitioning problem (`prob049` 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, e.g. the 100×100 magic square requires 10000 variables whose domains range over 10000 values.

The experiment has been executed on an IBM QS21 dual-Cell blade system (times are measured in seconds). Since Adaptive Search uses random configurations and progression, each benchmark was executed 50 times. The resulting execution time is the average of the 50 runs after removing both the lowest and the highest times.

We ran a series of tests, varying the number of SPEs from 1 to 16. The figure depicts the evolution of the speedup w.r.t the number of SPEs for the hardest instance of each problem.



Concerning raw performance, it is worth noticing that AS/Cell performs very well even on difficult problems. As case study, consider the magic squares problem (one of the most challenging: most solvers cannot even handle instances larger than 20×20). With 1 SPE, the average time to solve magic square 100×100 is around 3 minutes. Using 16 SPEs, AS/Cell drastically reduces the computation time, bringing it to barely over 8 seconds. Moreover, these results are obtained with a 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 initial port yields a CPI of `2.05`, which puts us at about one third of the performance which we can expect to attain. Approximately half the cycles in SPE executions are spent in stalls, of which about 60 % are branch misses while the remaining 40 % are spent on dependency stalls. Again, these results are not surprising, since 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 techniques to further improve performance.

Our first assessment clearly shows that the Adaptive Search 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 how AS/Cell can influence it. 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 since it represents the "worst case" one can encounter: too high a value can even inhibit use in time-critical situations.

This table summarizes the results, focusing on the worst case.[1] 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 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: the table below charts the evolution of the standard deviation of the execution times for the hardest instance of each problem varying the number of SPEs.

Problem Instance	Speedup for *k* SPEs				
	2	4	8	12	16
`all-interval 450`	2.4	4.7	15	11	**26**
`partit 2600`	1.2	3.6	7.2	12	**17**
`perfect-square 5`	2.1	3.9	7.1	9.7	**17**
`magic-square 100`	63	150	160	540	**510**

The standard deviation rapidly decreases as more SPEs are used. For instance, considering 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 1 SPE execution needs 2.5 hours at worst. When using 16 SPEs, this drops to 18 seconds.

Problem Instance	Standard deviation with *k* SPEs					
	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

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 which 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 take advantage of the parallel search in an efficient manner.

[1] Due to space limitation we only show the values for the hardest instance of each problem

7 Conclusion

We presented a simple yet effective parallel adaptation of the Adaptive Search 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 between 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 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 uses, for instance interactive games.

Recall that Adaptive Search is an "anytime" method: it is always possible to interrupt the execution and obtain the best pseudo-solution 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 pseudo-solution (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 a SPE needs to pass, when copying, is the current configuration (a small array of integers) and its associated cost, which can be done 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 [17] 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 MPI or the RDMA-based Fraunhofer Virtual Machine [11].

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 Executes sur MultiProcesseurs."

References

1. Renata M. Aiex, Mauricio G. C. Resende, and Celso C. Ribeiro. Probability distribution of solution time in grasp: An experimental investigation. *Journal of Heuristics*, 8(3):343–373, 2002.
2. Enrique Alba. Special issue on new advances on parallel meta-heuristics for complex problems. *Journal of Heuristics*, 10(3):239–380, 2004.
3. Geoffrey Chu and Peter Stuckey. A parallelization of minisat 2.0. In *SAT race*, 2008.
4. Philippe Codognet and Daniel Diaz. An efficient library for solving csp with local search. In T. Ibaraki, editor, *MIC'03, 5th International Conference on Metaheuristics*, 2003.
5. Teodor Crainic and Michel Toulouse. Special issue on parallel meta-heuristics. *Journal of Heuristics*, 8(3):247–388, 2002.
6. Teodor Gabriel Crainic, Michel Gendreau, Pierre Hansen, and Nenad Mladenovic. Cooperative parallel variable neighborhood search for the -median. *Journal of Heuristics*, 10(3):293–314, 2004.
7. Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1994.
8. Ian P. Gent and Toby Walsh. Csplib: A benchmark library for constraints. In *CP*, pages 480–481, 1999.
9. Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2009.
10. The Van Luong, Nouredine Melad, and El-Ghazali Talbi. Parallel local search on gpu. Technical Report RR 6915, INRIA, Lille, France, 2009.
11. R. Machado and C. Lojewski. The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. *Computer Science-Research and Development*, 23(3):125–132, 2009.
12. Laurent Michel, Andrew See, and Pascal Van Hentenryck. Distributed constraint-based local search. In Frédéric Benhamou, editor, *proceedings of CP'06, 12th Int. Conf. on Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 2006.
13. Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artif. Intell.*, 58(1-3):161–205, 1992.
14. Kei Ohmura and Kazunori Ueda. c-sat: A parallel sat solver for clusters. In *SAT*, pages 524–537. Springer Verlag, 2009.
15. Panos M. Pardalos, Leonidas S. Pitsoulis, Thelma D. Mavridou, and Mauricio G. C. Resende. Parallel search for combinatorial optimization: Genetic algorithms, simulated annealing, tabu search and grasp. In *IRREGULAR*, pages 317–331, 1995.
16. Laurent Perron. Search procedures and parallelism in constraint programming. In *CP*, pages 346–360, 1999.
17. IBM Redbooks. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. Vervante, 2008.
18. Tobias Schubert, Matthew D. T. Lewis, and Bernd Becker. Pamiraxt: Parallel sat solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:203–222, 2009.
19. M. Verhoeven and E. Aarts. Parallel local search. *Journal of Heuristics*, 1(1):43–65, 1995.
20. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.