Three Implementations of Branch-and-Bound in CLP

Steven Prestwich
Nortel technology, London Road, Harlow,
Essex, CM17 9NA, England Tel. +44 1279 40 2719
Email steven@nortel.co.uk

September 9, 1996

Constraint logic programming systems are becoming increasingly popular for solving real-life combinatorial optimization problems. An important component of such systems is therefore an optimization function, and one or more implementations of the branch-and-bound method are typically provided. However, there seems to be no available methodology for choosing an implementation, and making the wrong choice can cause disastrous performance problems. This paper examines the robustness of three known implementations by testing their performance under extreme conditions. One implementation performs badly when constraint propagation is poor and also interacts poorly with parallelization in three distinct ways. Another performs badly on certain cost functions. The third is not always the fastest but it emerges as more robust than the other two. We therefore recommend that only the third or something like it be provided, thus relieving the programmer of a difficult but critical decision.
Keywords: *constraint logic programming, or-parallelism, branch-and-bound*

1. Introduction

Constraint Logic Programming (CLP) has been successfully applied to combinatorial optimization problems in Operations Research and Artificial Intelligence [3, 20, 21], some implementations approaching the efficiency of specialized, hand-coded algorithms [2]. The main advantage of the CLP approach is its ease of programming and its flexibility. For well-studied problems Operations Research techniques are the best, but for new problems with complex and non-standard constraints, the flexibility of CLP makes it the tool of choice.

Most CLP systems provide functions for optimization based on the branch-and-bound method. The first two implementations were in CHIP [5]. Various implementations are also provided by commercial versions of CHIP [1] and related constraint solving tools, such as ECL^{i}PS^e [6], CHARME [11] and DECISIONPOWER [4]; and by ILOG SOLVER [7], a constraint solver based on C++ instead of Prolog.

Users of such systems, when programming a new problem, must try all the available implementations to see which works best. However, there are no guiding principles for selecting an implementation, and even after tests on examples there remains the possibility that another implementation would be much better with different data or in parallel.

In this paper we test the behaviour of three implementations under various conditions, both sequential and parallel. The aim is to highlight the dangers of each and to see whether one emerges as more robust than the others. The examples we construct to do this are also useful as benchmarks for new implementations. First we provide some background in Sections 1.1. and 1.2.. In Section 2. we describe the three implementations, in Section 3. we compare them, and in Section 4. we draw some conclusions.

1.1. Or-parallel branch-and-bound

The basic paradigm used in CLP is that of "constrain and generate". Since even after the constrain phase the remaining search space can be quite large, or-parallel CLP systems attempt to explore the remaining search space in parallel. This combination of or-parallelism and CLP was first suggested in [17]. The first real system to combine finite domain constraints with or-parallelism was Elipsys [19]. Or-parallelism has now been added to ECL^{i}PS^e [6] on which the results in this paper have been obtained.

Or-parallelism works as follows. At certain nodes of the search tree each branch is tried in parallel on different processors instead of in sequence by backtracking. These nodes are usually selected by a run-time scheduler from a superset specified by the programmer in the program source. Actually, the branches need not be explored by physically different processors, only by different *workers* (or processes). Several workers can be started on a single machine by multitasking, but in this paper we shall assume that each physical processor executes exactly one process, and that a worker is equivalent to a processor.

A number of researchers have worked on exploiting or-parallelism within branch-and-bound algorithms, both inside and outside the logic programming framework [8, 9, 10, 15, 16]. In particular, Szeredi [16] investigated the use of or-parallel Prolog for such problems and had some encouraging preliminary performance figures using Aurora [12].

1.2. Terminology

We use the standard Prolog syntax whereby predicates and constants are denoted by lower-case and variables by upper-case characters, plus some constraint operators. The `=/2` operator (that is the `=` operator with arity 2) denotes unification while `#=/2` makes the value of two domain variables equal. The `::/2` operator associates a domain with a domain variable or a list of domain variables. The `../2` operator defines a closed integer interval in a domain. So for example `[X,Y]::0..3, Z#=X` sets the domains of the variables `X`, `Y` and `Z` to $\{0, 1, 2, 3\}$. The syntax is fully defined in the ECL^{i}PS^e manual [6]. We also allow sub- and superscripts to facilitate the description of parameterized families of programs. A call to a branch-and-bound implementation s with goal `Goal` and cost function `Cost` will be denoted by $\mathtt{bb}_s$`(Goal,Cost)`. We allow `Goal` to be a conjunction $(\mathtt{Goal}_1, \mathtt{Goal}_2, \ldots)$.

2. Three branch-and-bound implementations

Combinatorial optimization problems typically have many solutions. A cost is assigned to each solution and we are usually interested in finding any one of the optimal solutions, that is one with the lowest possible cost. This is done by searching for a solution to the problem and then adding a further constraint that any new solution must be better than the current best. The way the new constraint is added and checked may affect performance enormously. We now describe three implementations which we shall call $\mathtt{bb}_1$, $\mathtt{bb}_2$ and $\mathtt{bb}_3$.

2.1. Implementation $\texttt{bb}_1$

The $\texttt{bb}_1$ implementation appears to be the most popular, probably because it is easily implemented and it usually behaves quite well. It is provided by CHIP, ElipSys, ECL^{i}PS^e, DECISIONPOWER (called `min_max` in each), CHARME (`minim`), and ILOG SOLVER (`IlcMinimize`) and works as follows. First find any solution by standard backtracking search, using the initial upper and lower cost bounds H and L as a constraint $L \leq C \leq H$ to prune the search, where C is the cost function being constructed. Whenever a new solution is found with cost C_i the search is halted and *restarted* using the tighter constraint $L \leq C < C_i$ to prune the search. Eventually no further solutions can be found (or $C_i = L$) and the solution found in the previous search is optimal.

2.2. Implementation $\texttt{bb}_2$

The $\texttt{bb}_2$ implementation is provided by CHIP, ElipSys, ECL^{i}PS^e and DECISIONPOWER (called `minimize` in each). It does not restart the search when a new cost bound is established. Instead it maintains a global variable which records the lowest cost found so far. On finding a solution with cost C_i the global variable is updated. Whenever the domain of C changes during the search this variable is consulted, and a new constraint $C \leq C_i$ added (which is undone on backtracking). The idea behind $\texttt{bb}_2$ is to avoid the restarting overhead of $\texttt{bb}_1$.

2.3. Implementation $\texttt{bb}_3$

This implementation is provided by ILOG SOLVER (`IlcMinim`), CHARME (`minimize`) and was proposed for ECL^{i}PS^e in [14] (where it was called `min`). It works in a similar way to $\texttt{bb}_2$ except that the variable is consulted at the start of every branch of every node in the search tree, and a new constraint $C \leq C_i$ added as in $\texttt{bb}_2$ (actually, the new constraint need only be added if the upper bound on the cost function is greater than C_i). At a node with many branches, if the new constraint fails at some branch then $\texttt{bb}_3$ should not try the remaining branches (though it currently does in CHARME [11]).

2.4. Discussion

There are other possibilities than the three implementations described here. For example when a new upper cost bound is found, intelligent backtracking can be used to jump back to an appropriate place in the search tree [18]. However, in this paper we only consider the three implementations $\texttt{bb}_{1,2,3}$ currently available in ECL^{i}PS^e. They each have pros and cons which we discuss here before testing them on examples.

Firstly, note that $\texttt{bb}_3$ always uses an up-to-date value of C_i because it consults C_i at every labeling; $\texttt{bb}_1$ is also always up-to-date because as soon as C_i is changed the entire search is restarted. In contrast $\texttt{bb}_2$ will perform some search using an outdated C_i if the domain of C does not change immediately after finding a new solution. Therefore $\texttt{bb}_2$ may perform redundant search. Secondly, note that $\texttt{bb}_1$ may explore some leftmost part of the search space on more than one iteration, depending upon how efficient the constraint propagation is. Neither of $\texttt{bb}_{2,3}$ restart the search so they never explore any part of the space more than once. Hence $\texttt{bb}_{1,2}$ perform at least as much search as $\texttt{bb}_3$, though for different reasons. However, the extra checking on the part of $\texttt{bb}_3$ may make it slower than $\texttt{bb}_{1,2}$.

Each of the three implementations perform better than the other two on some programs, and parallelism adds a new dimension to the problem; are any of the implementations more suited to parallel execution than the others? We know of no methodology for selecting an implementation, yet it is clearly unsatisfactory to force users to test them on each new problem, especially as the

```
p_{s,n}  :-  L = [V0,V1,V2,V3,V4,V5,V6,V7,V8,V9],
             [V|L] :: 0 .. 1,  Cost :: 0 .. 1023,
             Cost #= 2^9 * V0 + ... + 2^(9-n) * Vn + ... + 2^0 * V9,
             bb_s((label10(L), V = Vn), Cost).
```

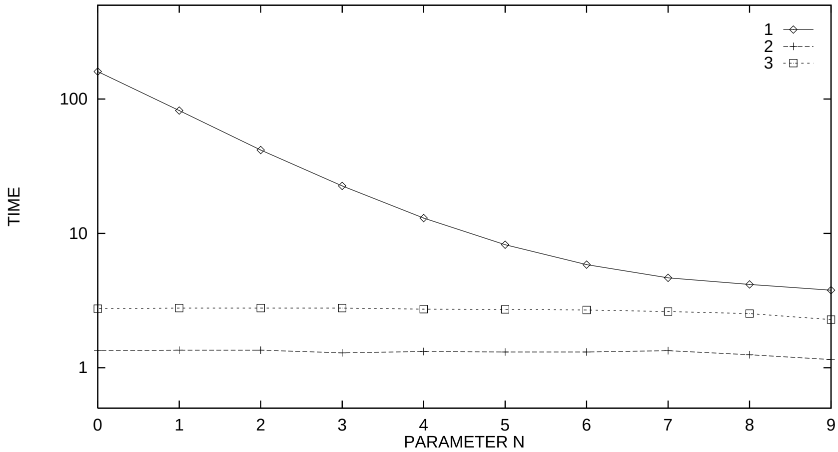

Figure 1: Example showing the effects of poor constraint propagation

characteristics of a single problem may depend on its input. We shall therefore examine the behaviour of the implementations under extreme conditions to see if one is more robust than the others.

3. Six pathological examples

In this section we examine the behaviour of the three implementations $\mathtt{bb}_{1,2,3}$ by constructing pathological examples. The aim is to test their robustness under extreme conditions. The examples are constructed for clarity so that their behaviour can be understood. They are rather artificial in nature, but the fact that we can construct such examples means that they may occur in real programs. Moreover, we have observed at least some of these effects in real applications.

All timings were obtained on ECL^{i}PS^e version 3.5.2 [6]. Examples which are purely sequential were executed on a Sun SparcStation 5 while parallel examples were executed on a 4-processor ICL DRS6000. Timings are elapsed times (not CPU times) and are given in seconds.

3.1. Poor constraint propagation

Here we show that $\mathtt{bb}_1$ can suffer from a severe overhead for certain cost functions. Consider a program whose constraint propagation is poor in the sense that some early labeling(s) do not affect propagation immediately. We construct a family of programs with two parameters in Figure 1. The variable $\mathtt{V}_n$ takes no part in the constraint propagation. A lower value of n implies worse constraint propagation, because a $\mathtt{V}_n$ with low n has a higher coefficient in the cost function. The graph in Figure 1 compares the implementations for different values of n, with the query `?- p`$_{s,n}$.

The graph shows that decreasing n has little effect on $\mathtt{bb}_{2,3}$ but an exponentially bad effect on $\mathtt{bb}_1$. The reason is that a large left portion of the search space is re-traversed on each iteration.

$\mathtt{p}_{s,n}$:- L = [$\mathtt{V}_0$,$\mathtt{V}_1$,$\mathtt{V}_2$,$\mathtt{V}_3$,$\mathtt{V}_4$,$\mathtt{V}_5$,$\mathtt{V}_6$,$\mathtt{V}_7$,$\mathtt{V}_8$,$\mathtt{V}_9$],
L :: 0 .. 1, Cost :: 0 .. 1023,
Cost #= $2^0 * \mathtt{V}_0 + \ldots + 2^n * \mathtt{V}_n$,
$\mathtt{bb}_s$((label10(L), work(10000)), Cost).

```
label10([]).
label10([V|L])  :-  member(V,[1,0]), label10(L).
```

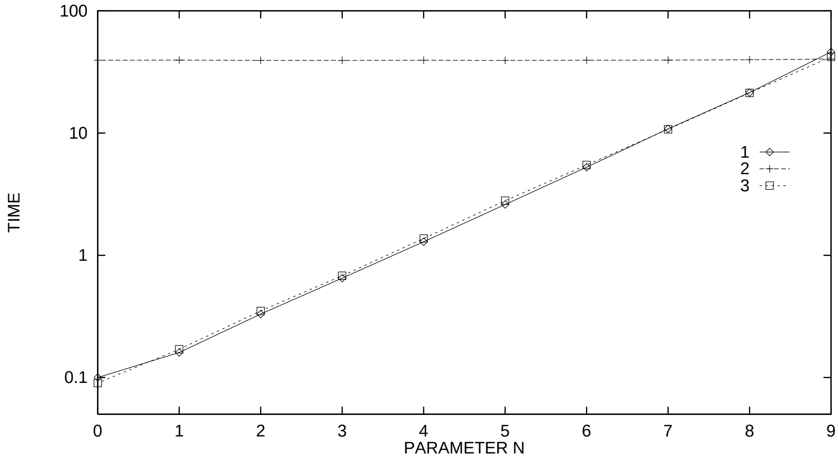

Figure 2: Example showing the effects of insufficient cost checking

3.2. Insufficient cost checking

Here we show that under certain conditions $\mathtt{bb}_2$ can be unacceptably inefficient compared to $\mathtt{bb}_{1,3}$, because it does not check the cost bound often enough and therefore performs redundant search; that is, it searches part of the space which cannot possibly contain a cheaper solution than the current best.

First we construct a family of programs with two parameters s and n in Figure 2. The predicate `label10/1` labels the domain variables, selecting the value `1` before the value `0`. It uses the predicate `member/2` which simply enumerates list members in textual order.

The predicate `work/1` is a simple predicate used to artificially increase the amount of work performed, so that timings can be made more accurately. This predicate will also be used below, and is defined by:

```
work(0)  :-  !.
work(N)  :-  N1 is N - 1, work(N1).
```

The graph in figure 2 compares the behaviour of the implementations with the query ?- $\mathtt{p}_{s,n}$ as $n \longrightarrow 0$. Recall that Cost #= $2^0 * \mathtt{V}_0 + \ldots + 2^n * \mathtt{V}_n$ so as $n \longrightarrow 0$ the number of possible costs also tends to zero. Both $\mathtt{bb}_{1,3}$ speed up exponentially as $n \longrightarrow 0$ but $\mathtt{bb}_2$ hardly speeds up at all. This is because the later labeling events ($n+1 \ \ldots\ 9$) do not affect the cost function, so $\mathtt{bb}_2$ does not check

```
p_s(N)  :-  Cost :: 1 .. 1000,
            bb_s(g(Cost, N), Cost).

g(Cost, N)  :-  reverse_indomain(Cost),
                work(N),  Cost #= 1.
```

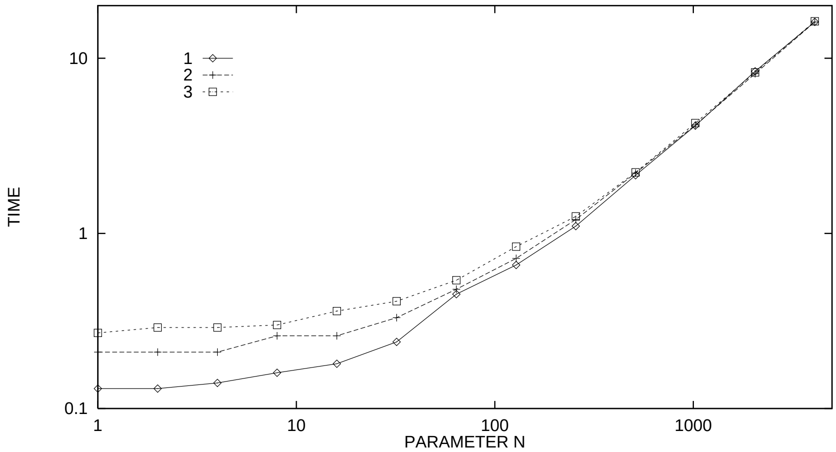

Figure 3: Example showing the effects of excessive cost checking

the global cost bound often enough. For large portions of the search, $\mathtt{bb}_2$ is using an out-of-date cost bound, which makes it extremely inefficient for this example.

This behaviour has been observed in more realistic problems. In [14] it was noted that a set of job-shop scheduling problems behaved well under $\mathtt{bb}_{1,3}$ but was orders of magnitude slower under $\mathtt{bb}_2$. For these reasons (we believe that) $\mathtt{bb}_2$ is used less often than $\mathtt{bb}_1$.

3.3. Excessive cost checking

Here we consider the overhead of cost bound checking in each implementation, with the family of programs in Figure 3. The argument `N` controls the amount of work performed after each labeling, and labeling is performed in reverse order by `reverse_indomain`. The idea of this example is that `N` simulates constraint propagation. As `N` $\longrightarrow$ `0` the simulated propagation becomes negligible, and times are increasingly dominated by other overheads such as cost bound checking, resolution and so on. There is only one solution with cost `1` so that the timings are not distorted by frequently updating the cost bound.

Executing this program with the query `?-` $\mathtt{p}_s$ we obtain the graph shown in Figure 3. As `N` $\longrightarrow$ `0` $\mathtt{bb}_3$ becomes worse than $\mathtt{bb}_2$ which in turn becomes worse than $\mathtt{bb}_1$. In this example $\mathtt{bb}_1$ has the best times, at least partly because it has negligible cost bound checking overhead.

Even in the worst case $\mathtt{bb}_3$ is only twice as slow as $\mathtt{bb}_1$. Although this is not conclusive evidence, and moreover applies only to these particular implementations, it is at least evidence that it is hard to make $\mathtt{bb}_3$ much slower than $\mathtt{bb}_{1,2}$. Hence the cost bound checking overhead is not nearly as much of a potential problem as the overheads described in the previous two sections.

```
ps    :-  L = [V0,V1,V2,V3,V4],
          L :: 0 .. 1,  Cost :: 0 .. 31,
          Cost #= 16*V0 + 8*V1 + 4*V2 + 2*V3 + V4,
          bbs(g(L, Cost), Cost).

g(L, Cost)  :-  label10(L),
                N is 2000 * (31 - Cost) + 20000,
                work(N).
```

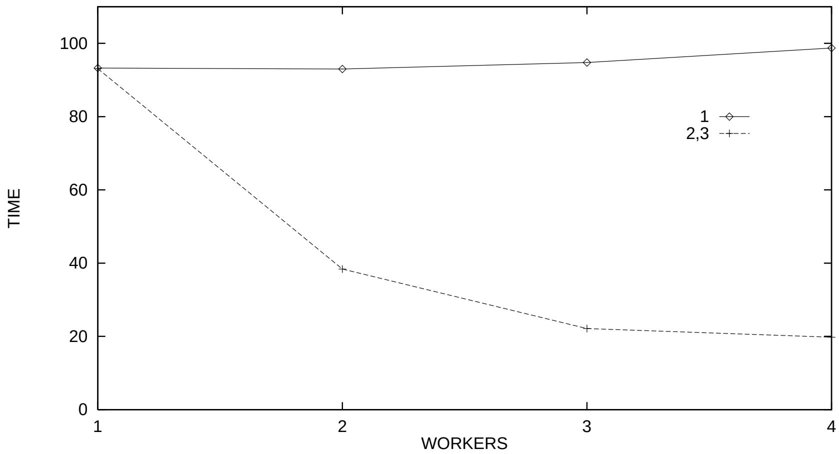

Figure 4: Example showing the effects of wasted speculative parallelism

3.4. Wasted speculative parallelism

The bottlenecks described above apply to both sequential and parallel optimization. In this and subsequent sections we describe bottlenecks associated only with parallel optimization.

A well-known problem with certain types of parallel program is that of *wasted speculative parallelism*. Speculative parallelism is parallel work which may turn out to have been unnecessary — that is, which would not be performed during a sequential execution of the program. If so, it is called *wasted*. Speculative parallelism may lead to superlinear speedups, but if some or all workers perform wasted speculative work then parallel speedup may be poor. All three implementations may perform wasted speculative work, but here we show that $\mathtt{bb}_1$ is more prone to this problem than $\mathtt{bb}_{2,3}$.

We construct a family of programs with a parameter s in Figure 4. Executing this program with 1–4 workers and the query `?- p`$_s$ we obtain the graph shown in Figure 4. Since $\mathtt{bb}_{2,3}$ behave in the same way for this example we merge their results. We see that $\mathtt{bb}_1$ has no speedup whereas $\mathtt{bb}_{2,3}$ have good (in fact superlinear) speedup (for example 2.42 for 2 workers). The reason is that, in this example, solutions near the left of the search space are found in a shorter time than those to the right (this is a consequence of the argument of the `work` predicate). Therefore $\mathtt{bb}_1$ never finds a solution faster in parallel than sequentially. However, $\mathtt{bb}_{2,3}$ continue to search and in a short time find a better solution.

This behaviour is quite likely to occur, especially if the program has been written with sequen-

```
p_s  :-  L = [V0,V1,V2,V3,V4],
         L :: 0 .. 1,  Cost :: 0 .. 31,
         Cost #= 16*V0 + 8*V1 + 4*V2 + 2*V3 + V4,
         bb_s(label01(16,L), Cost).

label01(N,[]).
label01(N,[V|L])  :-  W is 50000 * N * (2 - V),
                      N1 is N // 2,
                      par_member(V,[0,1]),
                      work(W),  label01(N1,L).
```

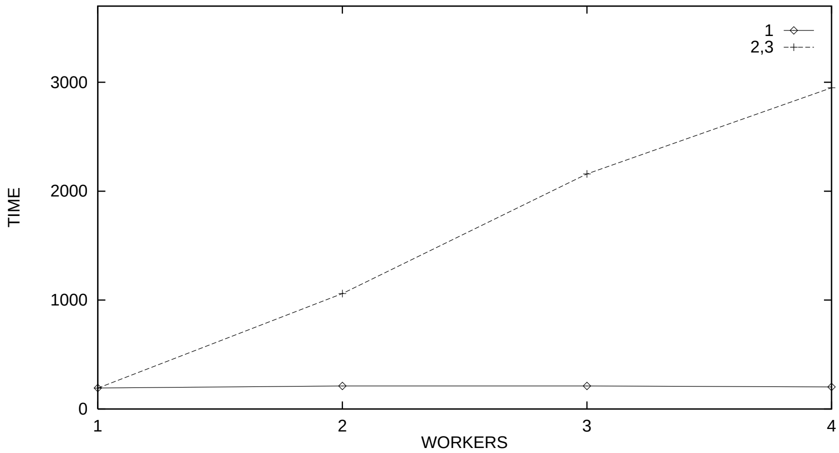

Figure 5: Example showing the effects of an unusual cost distribution

tial execution in mind. If heuristics such as dynamic value-ordering are successfully employed then solutions occur to the left of the space, and parallelism yields little benefit.

3.5. Unusual cost distributions

A logic program may be slower when executed in parallel than sequentially even if the parallelism overhead is negligible [13]. This is possible if what is computed depends upon the history of the parallel computation, as in branch-and-bound programs.

We construct a family of programs with a parameter s in Figure 5. The predicate `par_member/2` is identical to the `member/2` predicate, except that it tries alternatives in parallel where possible (and by backtracking where processors are unavailable). In this example, the more expensive the solution the easier it is to find. This is typical of optimization problems: there are often many expensive solutions, and only a few cheap ones. However, what is unusual about this example is that the cheaper the solution, the closer it is to the left of the search space. This may sound unlikely but it is possible, and we have observed somewhat similar behaviour in an application. In fact heuristics designed to speed up sequential programs may make it more likely, by moving cheap solutions left in the space.

```
p_s(N)  :-  Cost :: 1 .. 1000,
            bb_s(random_grain(N,Cost), Cost).
```

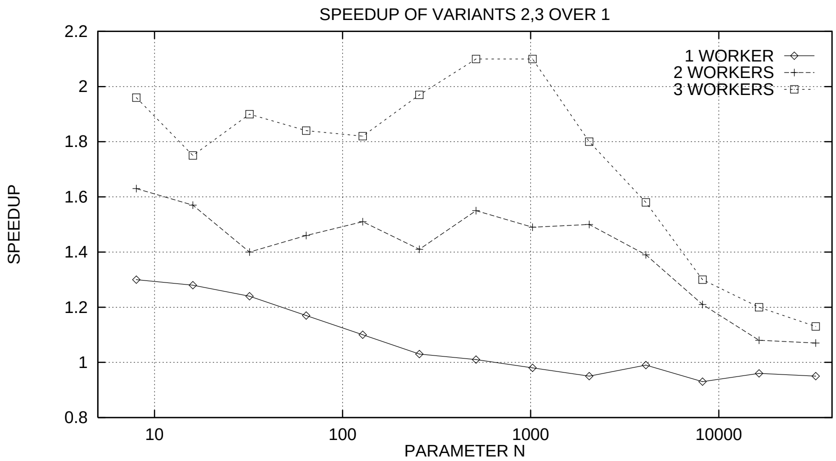

Figure 6: Example showing the effects of parallel overhead

Executing this program with 1, 2, 3 and 4 workers and the query `?- p`$_s$, we obtain the graph shown in Figure 5. Since bb$_{2,3}$ behave in the same way for this example we merge their results. None of the implementations find the optimal solution faster in parallel, though they all benefit from parallelism in the sense that they find *a* solution in a shorter time. The problem with bb$_1$ is that when a processor searching to the right finds a solution, restarting the search interrupts the processors which were about to find better solutions to the left. Parallelism therefore increases the number of iterations and hence the total execution time.

3.6. Parallel overhead

A common problem with parallel execution is the overhead caused by creating parallel tasks. In this section we compare the behaviour of the implementations as the grain size of the parallel tasks tends to zero. We construct a family of programs with a parameter s in Figure 6. The goal `random_grain(N,Cost)` generates 1000 parallel tasks, each leading to a solution. The costs of these solutions, from left to right, are `1000`, `999`, ... `2`, `1` and the tasks have random grain sizes in the range `0`–`N`.

We are interested in the behaviour of restarting implementations with non-restarting implementations for fine-grained parallel tasks. Since `random_grain` is not a “real” predicate we must simulate the effect of cost bound checking in bb$_{2,3}$. We use a bb$_{2,3}$ which checks the current cost bound after regular intervals equivalent to a grain size of `N` = 128.

Figures 6 shows the speedup of bb$_{2,3}$ relative to bb$_1$ (the execution times of bb$_1$ divided by those of bb$_{2,3}$) as a function of grain size (controlled by the parameter `N`) for the query `?- p`$_s$`(N)`, plotted for 1, 2 and 3 workers. The graph shows that as the grain size shrinks and the number of workers grows, bb$_{2,3}$ performs increasingly better than bb$_1$. The precise reasons for this are not easy to prove, because monitoring fine-grained parallel programs distorts the very effects we are trying to observe. However, cautious profiling indicates that for small-grained tasks bb$_1$ starts more searches than bb$_{2,3}$. The reason for this seems to be that at each iteration bb$_1$ starts a task for each worker, but when tasks are fine-grained the leftmost task is usually completed before the others have chance to communicate their results. So $\mathtt{bb}_1$ starts more parallel tasks than necessary and incurs greater parallel overhead. The difference is not great, but because it increases with each extra worker we believe that $\mathtt{bb}_1$ will behave poorly on highly parallel machines.

4. Conclusion

We compared the behaviour of three branch-and-bound implementations under a variety of extreme conditions, and we conclude that one ($\mathtt{bb}_3$) is superior to the others ($\mathtt{bb}_{1,2}$) in several ways. In the context of sequential execution $\mathtt{bb}_1$ may be very slow if early variable labelings cannot be used for constraint propagation, while $\mathtt{bb}_2$ may be very slow if the cost function is not affected by later variable labelings; in both cases $\mathtt{bb}_3$ behaves well. In fact, $\mathtt{bb}_3$ is unlikely to be much slower than $\mathtt{bb}_{1,2}$ and it may be much faster. We therefore recommend that, unless these problems can be predicted and avoided, only $\mathtt{bb}_3$ or something like it be used.

In the context of or-parallel execution $\mathtt{bb}_1$ may exhibit additional undesirable effects: more wasted speculative work than parallel $\mathtt{bb}_{2,3}$, more overhead than parallel $\mathtt{bb}_{2,3}$, and more restarts than sequential $\mathtt{bb}_1$. Parallel overhead may be avoided by the use of abstract interpreters at compilation time, or detected at runtime by monitoring tools. Considerable research has been done recently in this area, for example in Esprit Project EP6707, ParForce (Parallel Formal Computing Environment). However, we know of no way to detect the other problems at compilation time. Nor can they be easily detected at runtime by monitoring tools, because they are not related to general properties such as communication overhead or load balancing. We therefore conclude that the widely-used $\mathtt{bb}_1$ is particularly unsuitable for parallel combinatorial optimization.

We should note that $\mathtt{bb}_3$ is also not without problems, and we have not considered how frequently it should consult the current upper cost bound. Here we chose a simple approach: it is checked after each variable labeling (or other nondeterministic step). In problems where propagation is cheap this may be too often, causing high overheads. Conversely, if propagation is expensive it may be too seldom — in a parallel execution workers should check each others' results frequently, to ensure that their cost bounds are not out of date. We provided empirical evidence that the cost checking overhead is not a significant problem, but future work should include a study of this issue.

References

[1] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. In *Proc. of JFPL'92*, 1992.

[2] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. In *Proc. of 11th Int. Conf. on Logic Programming*, pages 369–383. The MIT Press, 1994.

[3] D. Clark, C. Rawlings, J. Shirazi, L.-L. Li, M. Reeve, K. Schuermann, and A. Veron. Solving large combinatorial problems in molecular biology using the ElipSys parallel constraint logic programming system. *Computer Journal*, 36(4), 1993.

[4] J.-Y. Cras. A review of industrial constraint solving tools. Survey, AI Perspectives, 1994.

[5] M. Dincbas et al. The constraint logic programming language CHIP. In *Proc. of the Int. Conf. on 5th Generation Computer Systems (FGCS'88)*, pages 693–702. ICOT, 1988.

[6] European Computer-Industry Research Centre, Munich, Germany. *ECLiPSe 3.5.2 User Manual*, December 1995.

[7] ILOG. ILOG SOLVER reference manual version 3.0. Technical report, 1995.

[8] J. M. Jansen and F. W. Sijstermans. Parallel branch-and-bound algorithms. *Future Generation Computer Systems*, 4:271–279, 1988–89.

[9] V. Kumar and V. N. Rao. Parallel depth first search — Part I. Implementation. *Int. Journal of Parallel Programming*, 16(6), 1987.

[10] V. Kumar and V. N. Rao. Parallel depth first search — Part II. Analysis. *Int. Journal of Parallel Programming*, 16(6), 1987.

[11] M. Leconte. Personal communication, March 1996.

[12] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szerdi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora Or-Parallel Prolog system. In *Proc. of the Int. Conf. on 5th Generation Computer Systems (FGCS'88).* ICOT, 1988.

[13] S. D. Prestwich. On parallelization strategies for logic programs. In *Proc. of the Int. Conf. on Parallel Processing*, number 854 in LNCS, pages 289–300. Springer-Verlag, 1994.

[14] S. D. Prestwich and S. Mudambi. Improved branch and bound in constraint logic programming. In *Proc. of the 1st Int. Conf. on Principles and Practice of Constraint Programming*, pages 533–548, 1995.

[15] T. J. Reynold and P. Kefalas. Or-parallel prolog and search problems in AI applications. In *Logic Programming: Proc. of the 7th Int Conf.*, pages 340–354. MIT Press, 1990.

[16] P. Szeredi. Exploiting or-parallelism in optimisation problems. In *Proc. of the JICSLP'92*, pages 703–716. MIT Press, 1992.

[17] P. van Hentenryck. Parallel constraint satisfaction in logic programming: Preliminary results of chip within PEPSys. In Giorgio Levi and Maurizio Martelli, editors, *Proc. of the 6th Int. Conf. on Logic Programming*, pages 165–180, Lisbon, 1989. The MIT Press.

[18] P. van Hentenryck. Personal communication, January 1995.

[19] A. Veron, K. Schuerman, M. Reeve, and L. Li. Why and how in the ElipSys Or-Parallel CLP system. In *PARLE'93: Parallel Architectures and Languages Europe*, pages 291–302. Springer-Verlag, 1993.

[20] A. Veron, K. Schuerman, M. Reeve, and L.-L. Li. APPLAUSE: Applications using the ElipSys parallel CLP system. In *Proc. of the 10th Int. Conf. on Logic Programming.* MIT Press, June 1993.

[21] M. Wallace. Constraints in planning, scheduling and placement problems. In *Constraint Programming.* Springer-Verlag, 1994.