Solving scheduling problems using parallel message-passing based constraint programming

Feng Xie
Department of Computing and Software,
McMaster University, 280 Main Street West,
Hamilton, L8S 4K1, Ontario, Canada
xief@mcmaster.ca

Andrew Davenport
Department of Business Analytics and Mathematical Science,
IBM T. J. Watson Research Center,
P.O. Box 218, Yorktown Heights, NY, 10598, USA
davenport@us.ibm.com

Abstract

We discuss some of the engineering challenges in implementing a constraint programming based scheduling engine to scale well on massively parallel computer hardware. In particular, we have been targeting our solvers to work in high performance computer environments such as the IBM BlueGene/P supercomputer. On such hardware parallelism is enabled at the software level using message passing, based on the MPI standard. We investigate a parallelization scheme based on a dynamic decomposition and reallocation of the search space during search. We present initial results of our research showing good scaling behaviour on challenging resource-constrained project scheduling problems up to 1024 processors on the IBM BlueGene/P supercomputer.

Introduction

Recent progress in computer processor design has seen a transition from improvements in clock speed to the use of parallel architectures. Whereas single core CPU speed remains within the 2-4 GHz range, multi-core architectures are giving us more computational power through more cores in a single computer: four core processors are now commonplace, eight core processors have recently been released and eighty core prototype processors are currently in development by Intel. In addition, developments in supercomputing are resulting in massively parallel distributed architectures, for example with up 65,536 processors on IBM's first generation BlueGene/L supercomputer and up to 1,048,576 cores on IBM's second generation BlueGene/P supercomputer (IBM 2006).

The effective exploitation of multi-core and parallel hardware architectures is critical for the next generation of combinatorial optimization based software. We identify two potential areas where parallel solvers can have a significant impact in the scheduling area:

- Using parallelism to solve very large-scale scheduling problems through (static or dynamic) problem decompostion. Modern constraint programming solvers are able to handle up to a few thousand activities. We have regularly encountered problems with tens or even hundreds of thousands of activities which cannot be solved using constraint programming.
- Using parallelism to control variability in solver run-time. End-users of scheduling systems based on mixed-integer programming or constraint programming technology often do not understand or appreciate the potentially high variability in run times, exhibited by “heavy tails” in the run time distribution (Gomes et al. 2000) and a consequence of many practical scheduling problems being NP-hard. Distributed computing environments, where computing capacity can be allocated on-demand, offer a potential solution for reducing this variability on an as-needed basis.

There has been significant research into designing constraint programming solvers for shared memory multi-core architectures (Perron 1999; Michel, See, and Van-Hentenryck 2007). There has been less work on developing solvers to work in distributed, message-passing environments (see (Michel, See, and Van-Hentenryck 2008; Duan, Gabrielsson, and Beck 2007) for some examples). The focus of the research presented in this paper is on designing constraint programming solvers for massively parallel distributed environments, where coordination of solvers is achieved through message passing. In particular, we investigate parallelizing tree search over a larger number of processors than is typically found in today's consumer multi-core hardware. The goal of our research is to develop scalable parallel constraint programming solvers for massively parallel computer environments.

In the sections which we follow we present an overview of BlueGene, the target hardware architecture of the research in this paper, and investigate a dynamic decomposition scheme for parallelizing search. We present some experimental results on BlueGene.

BlueGene

Blue Gene is an IBM Research project dedicated to exploring the frontiers in supercomputing: in computer architecture, in the software required to program and control massively parallel systems, and in the use of computation to advance our understanding of important biological processes such as protein folding. The Blue Gene/L supercomputer is unique in the following aspects:

- Trading the speed of processors for lower power consumption.

- Dual processors per node with two working modes: co-processor (1 user process/node: computation and communication work is shared by two processors) and virtual node (2 user processes/node)
- A large number of nodes (scalable in increments of 1024 up to at least 65,536)
- Three-dimensional torus interconnect with auxiliary networks for global communications, I/O, and management
- Lightweight OS per node for minimum system overhead.

Blue Gene/P, the second generation of the Blue Gene supercomputer, is designed to run continuously at 1 PFLOPS (petaFLOPS) and it can be configured to reach speeds in excess of 3 PFLOPS. It is at least seven times more energy efficient than any other supercomputer, accomplished by using many small, low-power chips connected through five specialized networks. Four 850 MHz PowerPC 450 processors are integrated on each Blue Gene/P chip. The 1-PFLOPS Blue Gene/P configuration is a 294,912-processor, 72-rack system harnessed to a high-speed, optical network. Blue Gene/P can be scaled to an 884,736-processor, 216-rack cluster to achieve 3-PFLOPS performance.

Parallelism in software targetted at BlueGene is achieved through message-passing, based on the open standards MPI library (Forum 1997). While message passing based communication is designed to be fast on BlueGene, it is still significantly slower than communication through shared memory. It is faster than in communication in distributed networks of computers (often two orders of magnitude faster).

Figure 1: IBM's BlueGene/P supercomputer

Overview of constraint programming environment

We have developed a parallel solver based on a constraint programming based scheduling library developed at IBM Research (currently named the Watson Scheduling Library). Scheduling problems are solved using tree search combined with constraint propagation at each node of the search tree. The focus of the research presented in this paper is how to allocate parts of the search tree to different processors in a parallel tree search. We parallelize a limited discrepancy search (Harvey and Ginsberg 1995), using the SetTimes branching heuristic and timetable and edge-finding resource constraint propagation (Baptiste, Pape, and Nuijten 2001). We evaluate the parallelization schemes on resource constrained project scheduling problems from PSPLIB (Kolisch and Sprecher 1996).

A dynamic parallelization scheme

Parallelization of search algorithms over a small number of processors or cores can often be achieved by statically decomposing the problem into a number of disjoint sub-problems as a preprocessing step, prior to search. This might be achieved by fixing some variables to different values in each sub-problem. This is the strategy used by some commercial mixed-integer programming and constraint programming solvers. The advantage of such a static decomposition scheme is that each processor can work independently on its assigned part of the search space and communication is only needed to terminate the solve. When scaling this static decomposition scheme to large numbers of processors, we have observed that the resulting computational effort required to explore each sub-problem can be very unbalanced, resulting in poor load-balancing and processor idle time. In a good load balancing scheme, we want to minimize processor idle time.

Dynamic work allocation schemes partition the search space among processors in response to the evolving search tree, for example by reassigning among processors during problem solving. As a result, processors are less likely to become idle for long periods of time, compared to a static decomposition scheme. Work-stealing is an example of a dynamic decomposition scheme that has been used in programming languages such as CILK (Blumofe et al. 1995), and in constraint programming (Michel, See, and Van-Hentenryck 2007) on shared memory architectures. We have developed a simple dynamic load balancing scheme for constraint programming based tree search, using message-passing based parallelism in distributed computing architectures. One of the goals of our scheme is to achieve linear scaling as much as possible by exploring nodes in the parallel scheme in as close as possible the same order that the serial algorithm would use. The basic idea of our approach is the following:

- The processors are divided into master and worker processes.
- Each worker processor is given a particular sub-tree to explore.
- A worker processor requests a sub-tree from a master processor.
- The master processors are responsible for coordinating the worker processors. A master process has a global view of the search tree and acts as a sub-problem dispatcher by assigning available sub-problems to worker processors. It keeps track of which sub-trees have been explored and which are to be explored.

There are a number of challenges in implementing such a dynamic problem decomposition scheme, each of which can have a major impact on performance:

- how to represent the pool of assigned and unassigned sub-problems at the master processors;
- how to communicate sub-problem information between the master and worker processes;
- how to initialize worker processors with new assigned sub-problems;
- how to generate enough work to keep all the workers occupied.

Master-worker communication

Given an initial model M of a constraint programming problem, a sub-problem of this model consists of the model itself augmented with some constraints C. These constraints correspond to a path from the root of the search tree to the root of the sub-problem search tree. In the context of scheduling, the constraints in C might be precedence constraints between activities with a demand for some shared resource (Cheng and Smith 1997).

Each worker processor is a constraint programming solver which implements a tree-based search algorithm on the constraint programming model M. Each worker is assigned a unique master processor. Typically a single master processor is coordinating many worker processors. The master processor assigns sub-problems of the model M to the worker, where each sub-problem is specified by a set of constraints C. These constraints are communicated in a serialized form using message-passing. A worker may receive a new sub-problem from its assigned master processor either at the beginning of problem solving, or during problem solving after exhausting tree search on its previously assigned sub-problem. On receiving a message from its master specifying a sub-problem as a set of constraints, a worker processor will establish an initial state to start tree search by creating and posting constraints to its constraint store based on this message.

Problem pool representation

A problem pool is used by the master to keep track of which parts of the search space have been explored by the worker processors, which parts are being explored and which parts are remaining to be explored.

Each master processor maintains a *job tree* to keep track of this information (see Figure 2). A job tree is a representation of the tree explored by the tree search algorithm generated by the worker processors. A node n in the job tree represents the state of exploration of the node, with respect to the master's worker processors. Each node can be in one of three states: *explored*, *exploring* and *unexplored*. Let T be the subtree rooted at n. Then the states are defined as:

- explored: T has been exhausted;
- exploring: a subtree of T or T itself is assigned to a worker, and no result is received;
- unexplored: neither a subtree of T nor T itself is assigned to a worker.

An edge in a job tree is labelled with a representation of a constraint posted at the corresponding branch in the search tree generated by the tree search algorithm executed by the worker processors. The set of constraints on the edges from the root node of the job tree to some child node represents a sub-problem.

A job tree is dynamic structure that indicates how the whole search tree is partitioned among the workers at a certain time point in problem solving. An unexplored node in the job tree corresponds to a unit of job that can be assigned to a worker process who will search the subtree rooted at the node. In order to minimize the memory use and shorten the search time for new jobs, a job tree is expanded and shrunk dynamically in response to communications with the worker processors.

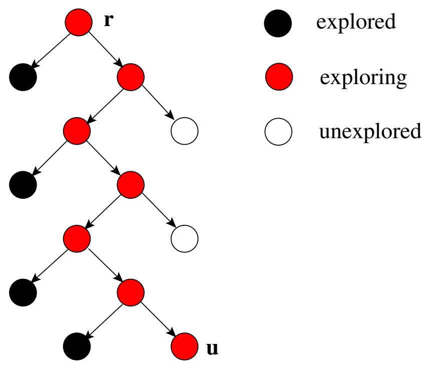

Figure 2: An illustration of a job tree

When a worker processor become idle (or at their initialization) they request work from their master processor. In response to such a request, a master processor will look up a node N in its job tree which is in an unexplored state, and send a message to the worker processor consisting of the sub-problem composed of the serialized set of constraints on the edges from the root node of the job tree to the node N.

Work generation

Work generation refers to creating and maintaining the job tree of the master processors so that new work can be assigned in response to requests from the worker processors. Work generation is an important issue: if there are not enough available unexplored nodes (work) on the job tree when a worker processor makes a job request of a master, then this worker must remain idle until new work is generated. In general, we want to minimize idle time of the worker processors. As we scale to larger numbers of worker processors, work generation can become an increasingly important issue.

Work generation occurs firstly during an initialization phase of the solve, and then dynamically during the solve itself. The initial phase of work generation involves creating the initial job tree for each of the master processors. The master processor creates its initial job tree by exploring some part of the search space of the problem, up to some (small) bound on the number of nodes to exlore. If during this initial search a solution is found, the master can terminate. Otherwise, the master initializes its job tree from the search tree explored during this phase. The master processor then enters into a job dispatching loop where it responds to requests for job assignments from the worker processors.

The second phase of work generation occurs as workers themselves explore the search space of their assigned subproblems and detect that they are exploring a large search tree which requires further parallelization. Job expansion is a mechanism for a worker to release free jobs if it detects that it is working on a large subtree. We use a simple scheme based on a threshold of searched nodes as a rough indicator of the “largeness” of the job subtree. If the number of nodes searched by a worker exceeds this threshold without exhausting the subtree or finding a solution, the worker will send a job expansion request to its master and pick a smaller part of the job to keep working on. Meanwhile, the master updates the job tree using the information offered by the worker, eventually dispatching the remaining parts of the original search tree to other worker processors.

In particular, suppose the tree is explored using depth-first search and a worker has reached the job expansion threshold and it is currently exploring node u (see Figure 2). Let P be the path from the root r of the job subtree to u, which can be built from the backtrack stack. It can be observed that all the nodes to left of P have been explored, and those to the right of P are to be explored. The worker can find out the number of branches of each node on the path, and forward this information to its master. Upon receiving the information, the master renders the nodes to the left of P as explored, and those to right as unexplored. After job expansion, the worker’s current job is changed to the subtree rooted at u.

Job expansion has two side effects. First, it introduces communication overhead because the job expansion information needs to be sent from the worker processor to the master processor. Secondly, the size of the job tree may become large, slowing down the search for unexplored nodes in response to worker job requests. The job tree can be pruned when all siblings of some explored node n are explored. In this case, the parent of node n can be rendered as explored and the siblings can be removed from the job tree.

Job dispatching

A master process employs a tree search algorithm to look for unexplored nodes in its job tree in response to job requests from the workers. The search algorithm used by the master to dispatch unexplored nodes in the job tree is customizable. It partially determines how the search tree is traversed as a whole. If a worker makes a job request and no unexplored nodes are available, the state of the worker is changed to idle. Once new jobs become available, the idle workers are woken up and dispatched these jobs.

MPI offers two types of communication: blocking and non-blocking. The process that initiates a blocking send or receive busy-waits for the communication to finish. Non-blocking operations enables a process to use the waiting time during communication for computation. The worker process benefits little from this feature because it cannot start working before finishing receiving all the information about the assigned job. However, applying non-blocking communication to the master process side can improve its throughput by using the waiting time for housekeeping and job caching. Our current implementation uses non-blocking communication only for termination signal receiving.

Multiple master processes

The parallelization scheme has been implemented on top of a constraint based scheduling system implemented at IBM Research. The results presented in this section are from execution runs on BlueGene/L and BlueGene/P (IBMBlueGeneSystemSolution) on resource constrained project scheduling problems from PSPLIB (Kolisch and Sprecher 1996). The job expansion threshold is set at 200.

Figure 3 shows the scaling performance of the parallelization scheme with a single master process, as we vary the number of processors from 64 to 1024 (on the 120 activity RCPSP instance 1-2 from PSPLIB). We manage to achieve good linear scaling up to 256 processors. However the single master process becomes a bottleneck when we have more than 256 worker processors, where we see overall execution time actually slow down as we increase the number of processor beyond 256.

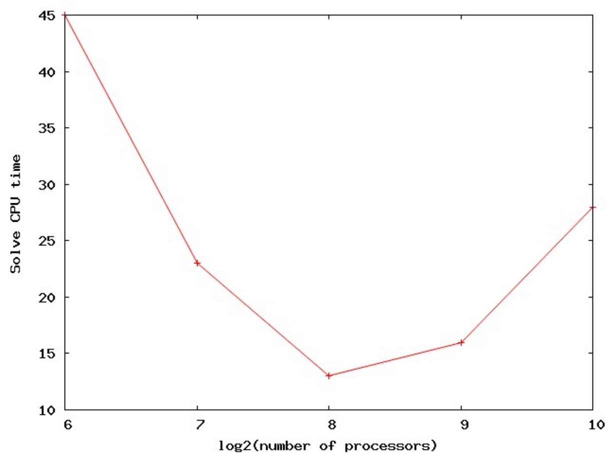

Figure 3: Scaling with one master process

The master processor can be a bottleneck as the number of workers assigned to it increases. In this case, multiple masters can be used to improve the scalability. In the multi-master mode, the search tree is divided among the masters in a static way. Each worker is associated with only one master.

We present here a static decomposition scheme to allocate sub-problems to each master process. The goal of the scheme is to partition the search space as evenly as possible so that each master processor is allocated a significant part of the search space and we minimize the number of processors that finish work early. In order to partition the search space, we consider possible branchings that can be made in the search tree. For example, in the context of resource constrained project scheduling, these could be possible sequencings of pairs of unsequenced activities sharing a common resource. In order to choose which branches to select in order to partition the search space as equally as possible between two processes, we evaluate a number of possible branchings by posting the corresponding constraints and evaluating the consequent reduction in search space size.

Algorithm 1 describes how we partition the space space between processors. The inputs to Algorithm 1 are the set of activities in the scheduling problem and a set processors. The output of the algorithm assigns to each processor a set of activity orderings. These orderings constitute sequencing constraints betwen activities that will be added by each processor to the constraint store as a preprocessing step before beginning tree search. Each processor is assigned a unique set of sequencings, such that the resulting search space assigned to each processor is disjoint from that assigned to all other processors.

Experimental investigation

Figure 4 plots the scaling performance of the scheduler (on the 120 activity RCPSP instance 1-2 from PSPLIB) with multiple master processes, as we vary the number of processors from 64 to 1024. With multiple master processors, we can achieve linear scaling up to 1024 processors. However the decomposition scheme used to distribute sub-problems over multiple masters can impact scaling. In our experiments we have not managed to achieve linear scaling with greater than 1024 processors and multiple masters. We believe that to scale well beyond 1024 processors requires developing techniques to dynamically allocate job trees between multiple masters.

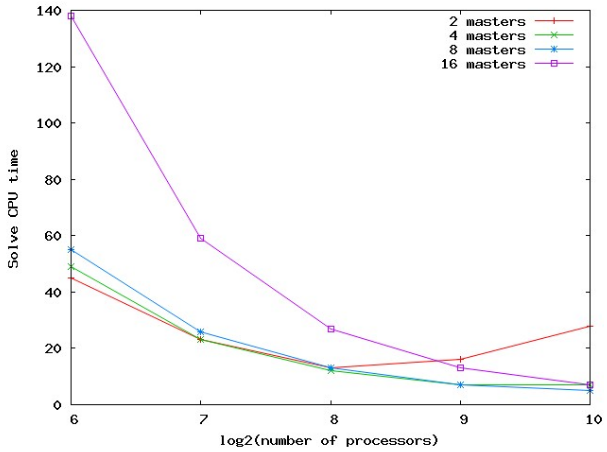

Figure 4: Scaling with multiple master processes

We present results showing execution time scaling for solving infeasible and feasible problems in Tables 1, 2 and 3 respectively. The number of processors ranges from 16 to

Algorithm 1: Static decomposition algorithm

```
input : a set of $n$ activities $A = \{A_1, \ldots, A_n\}$ with
        corresponding start time domains
        $\{min_{A_i}, \ldots, max_{A_i}\}$, a set of $m$ processors
        $P = \{P_1, \ldots, P_m\}$
output: an assignment of a set of orderings
        $orderings(i)$ of activities to each processor $i$
1  procedure static-decompose($A$, $P$)
2  begin
3      $minEval = 1.0$;
4      $minPair = ()$;
5      $sz = \prod_{A_i \in A} |max_{A_i} - min_{A_i} + 1|$;
6      foreach $A_i$ and $A_j$ which may be sequenced on a
       shared resource do
7          foreach sequencing $X \rightarrow Y$ of $A_i$ and $A_j$ do
8              post-constraint ($X \rightarrow Y$);
9              $sz_{X \rightarrow Y} = \prod_{A_j \in A} |max_{A_i} - min_{A_i} + 1|$
               after posting $X \rightarrow Y$;
10             retract-constraint ($X \rightarrow Y$);
11         end
12         $eval_{X,Y} = abs(\frac{sz_{X \rightarrow Y} - sz_{Y \rightarrow X}}{sz})$;
13         if $eval_{X,Y} < minEval$ then
14             $minEval = eval_{X,Y}$;
15             $minPair = (X, Y)$;
16         end
17     end
18     if $minPair \neq ()$ then
19         for $i = 1$ to $m/2$ do
20             $orderings(i)$.append(minPair.first →
               minPair.second)
21         end
22         static-decompose ($A$, $\{P_1, \ldots, P_{m/2}\}$);
23         for $i = m/2 + 1$ to $m$ do
24             $orderings(i)$.append(minPair.second →
               minPair.first)
25         end
26         static-decompose ($A$,
           $\{P_{m/2+1}, \ldots, P_m\}$)
27     end
28 end
```

512. Note that a single master is used for all the test cases except for 256 and 512 processes, in which a single master will become the bottleneck. The test results for 256 and 512 processes are obtained using two and four masters respectively.

Problem	np=16	32	64	128	256
5-9	20	9.1	5.0	3.0	2.0
14-1	78	38	18	9.0	5.0
14-4	65	31	15	8.0	5.0
14-10	91	43	21	11	6.0
26-3	69	33	16	9.1	4.1
26-6	24	11	5.1	3.0	2.1
30-5	204	98	48	26	13
30-2	98	47	23	12	6.0

Table 1: Scaling of execution time (in seconds) with varying number of processors for finding a proof of infeasibility for PSPLIB resource-constrained project scheduling problems with 60 activities. The infeasbility proof is when the makespan of the schedule is constrained to be 1 unit of time less than the optimal makespan.

Problem	np=16	32	64	128	256	512
14-4 (65)	30	14	7.0	3.1	2.1	2.0
26-3 (76)	>600	>600	90	75	24	10
26-6 (74)	63	18	8.1	5.0	2.0	1.0
30-10 (86)	>600	>600	>600	>600	216	88
42-3 (78)	>600	>600	>600	>600	256	81
46-3 (79)	148	27	13	6.0	3.1	2.0
46-4 (74)	>600	>600	>600	>600	104	77
46-6 (90)	>600	>600	477	419	275	122

Table 2: Execution time (in seconds) for solving feasible problems for PSPLIB resource-constrained project scheduling problems with 60 activities.

Problem	16	32	64	128	256	512
14-6 (76)	>600	371	218	142	48	25
26-2 (85)	294	142	86	35	16	9.0
22-3 (83)	50	24	12	5	3.0	0.07

Table 3: Execution time (in seconds) for solving feasible problems for PSPLIB resource-constrained project scheduling problems with 90 activities.

The experimental results show a good scaling for up to 512 processes, however the scaling deteriorates as the number of processes goes above that. This is mainly caused by the static load sharing among the masters, which is increasingly unbalanced as the number of masters increases.

Conclusions

We have presented a dynamic, message-passing based parallelization scheme for massively parallel constraint programming search. In experiments we have observed almost linear scaling on BlueGene up to 1024 processors on resource-constrained project scheduling problems. We have not managed to achieve linear scaling with greater than 1024 processors using multiple masters, where work is allocated statically to each master. We believe that to scale beyond 1024 processors we need to develop techniques to dynamically allocate sub-problems between masters.

References

Baptiste, P.; Pape, C. L.; and Nuijten, W. 2001. *Constraint-Based Scheduling - Applying Constraint Programming to Scheduling Problems*. International Series in Operations Research and Management Science. Springer.

Blumofe, R. D.; Joerg, C. F.; Kuszmaul, B. C.; Leiserson, C. E.; Randall, K. H.; and Zhou, Y. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 207–216.

Cheng, C., and Smith, S. 1997. Applying constraint satisfaction techniques to job-shop scheduling. *Annals of Operations Research, Special Issue on Scheduling: Theory and Practice* 70.

Duan, L.; Gabrielsson, S.; and Beck, J. 2007. Solving combinatorial problems with parallel cooperative solvers. In *Ninth International Workshop on Distributed Constraint Reasoning*.

Forum, M. P. I. 1997. MPI: A message-passing interface standard. Technical report.

Gomes, C. P.; Selman, B.; Crato, N.; and Kautz, H. A. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning* 24(1/2):67–100.

Harvey, W. D., and Ginsberg, M. L. 1995. Limited discrepancy search. In *14th International Joint Conference on Artificial Intelligence*.

2006. *IBM Journal of Research and Development: Special Issue on BlueGene*, volume 49 (2/3).

IBM System Blue Gene Solution: Application Development. IBM.

Kolisch, R., and Sprecher, A. 1996. Psplib - a project scheduling library. *European Journal of Operational Research* 96:205–216.

Michel, L.; See, A.; and Van-Hentenryck, P. 2007. Parallelizing constraint programs transparently. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*.

Michel, L.; See, A.; and Van-Hentenryck, P. 2008. Transparent parallelization of constraint programs on computer clusters.

Perron, L. 1999. Search procedures and parallelism in constraint programming. In *International Conference on the Principles and Practice of Constraint Programming*.