Massively Parallel Constraint Programming for Supercomputers: Challenges and Initial Results

Feng Xie[1] and Andrew Davenport[2]

[1] Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada
xief@mcmaster.ca
[2] IBM T. J. Watson Research Center, Yorktown Heights, NY, USA
davenport@us.ibm.com

1 Introduction

In this paper we present initial results for implementing a constraint programming solver on a massively parallel supercomputer where coordination between processing elements is achieved through message passing. Previous work on message passing based constraint programming has been targeted towards clusters of computers (see [1,2] for some examples). Our target hardware platform is the IBM Blue Gene supercomputer. Blue Gene is designed to use a large number of relatively slow (800MHz) processors in order to achieve lower power consumption, compared to other supercomputing platforms. Blue Gene/P, the second generation of Blue Gene, can run continuously at 1 PFLOPS and can be scaled to 884,736-processors to achieve 3 PFLOPS performance. We present a dynamic scheme for allocating sub-problems to processors in a parallel, limited discrepancy tree search [3]. We evaluate this parallelization scheme on resource constrained project scheduling problems from PSPLIB [4].

2 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 (as is explored in [5]). 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, this approach may sometimes lead to poor load-balancing and processor idle time.

Dynamic work allocation schemes partition the search space among processors in response to the evolving search tree, for example by reassigning work among processors during problem solving. Work-stealing is an example of a dynamic decomposition scheme that has been used in programming languages such as CILK [6], and in constraint programming [7] on shared memory architectures. We have

A. Lodi, M. Milano, and P. Toth (Eds.): CPAIOR 2010, LNCS 6140, pp. 334–338, 2010.
© Springer-Verlag Berlin Heidelberg 2010

developed a simple dynamic load balancing scheme for distributed hardware environments based on message passing. The basic idea behind the approach is that (a) the processors are divided into master and worker processes; (b) each worker processor is assigned sub-trees to explore by a master; and (c) the master processors are responsible for coordinating the sub-trees assigned to worker processors. A master process has a global view of the full search tree. It keeps track of which sub-trees have been explored and which are to be explored. Typically a single master processor is coordinating many worker processors. Each worker processor implements a tree-based search. The master processor assigns sub-problems to each worker, where each sub-problem is specified by a set of constraints. 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.

2.1 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. A job tree is a representation of the tree explored by the tree search algorithm generated by the worker processors. A node 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*, where the sub tree has been exhausted; *exploring*, where the subtree itself is assigned to a worker, and no result is received; or *unexplored*, where the subtree has not been assigned to any 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. 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. 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. 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 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.

2.2 Work Generation

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 explore. 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 sub-problems 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.

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.

2.3 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.

2.4 Multiple Master Processes

The results presented in this section are from execution runs on BlueGene/L on instances of resource constrained project scheduling problems from PSPLIB [4]. The constraint programming solver executed by the worker processes uses the SetTimes branching heuristic and timetable and edge-finding resource constraint propagation [8]. The job expansion threshold is set at 200 nodes. Due to space limitations, we only present limited results in this section. However they are representative of what we see for other problems in PSPLIB.

Figure 1 (left) 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 processors beyond 256.

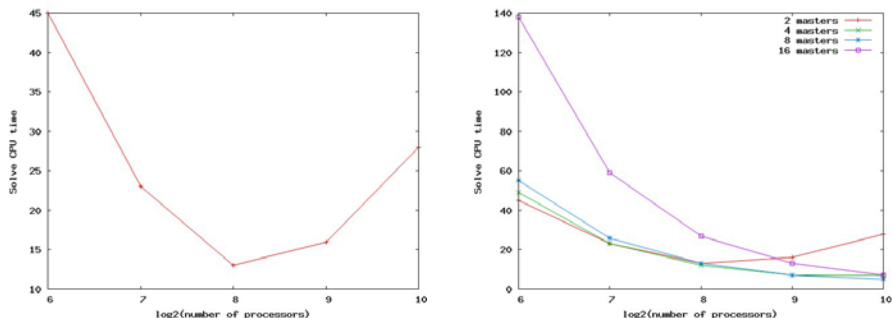

Fig. 1. Scaling with one master process (left) and multiple (right) master processes

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 scalability. In the multi-master mode, the full search tree is divided among the masters at the beginning of problem solving. This is a static decomposition scheme, in that sub-trees are not reallocated between masters dynamically during search. We implemented a simple static decomposition scheme based on measuring the reduction in search space size as we evaluate possible branches from the root node. We then distribute the sub-trees resulting from following these branches among the masters, so that as much as possible each master has a similar sized search space to explore. Figure 1 (right) 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. We see here that with multiple master processors, we can achieve good scaling up to 1024 processors. We also present results showing execution time scaling for solving feasible makespan satisfaction problems from PSPLIB in Table 1, for varying numbers of processors p. A single master is used for all the test cases except for 256 and 512 processes, where we used two and four masters respectively.

To summarize, with a single master processor, we are able to achieve good scaling up to 256 processors. With multiple masters, we achieve good scaling up to 512 and sometimes 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 good 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.

Table 1. Execution time (in seconds) for solving fixed makespan satisfaction PSPLIB resource-constrained project scheduling problems with 60 and 90 activities

Problem (makespan)	Size	CPU time					
		p=16	p=32	p=64	p=128	p=256	p=512
14-4 (65)	60	30	14	7.0	3.1	2.1	2.0
26-3 (76)	60	>600	>600	90	75	24	10
26-6 (74)	60	63	18	8.1	5.0	2.0	1.0
30-10 (86)	60	>600	>600	>600	>600	216	88
42-3 (78)	60	>600	>600	>600	>600	256	81
46-3 (79)	60	148	27	13	6.0	3.1	2.0
46-4 (74)	60	>600	>600	>600	>600	104	77
46-6 (90)	60	>600	>600	477	419	275	122
14-6 (76)	90	>600	371	218	142	48	25
26-2 (85)	90	294	142	86	35	16	9.0
22-3 (83)	90	50	24	12	5	3.0	0.07

References

1. Michel, L., See, A., Van Hentenryck, P.: Transparent parallelization of constraint programming. INFORMS Journal of Computing (2009)
2. Duan, L., Gabrielsson, S., Beck, J.: Solving combinatorial problems with parallel cooperative solvers. In: Ninth International Workshop on Distributed Constraint Reasoning (2007)
3. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: 14th International Joint Conference on Artificial Intelligence (1995)
4. Kolisch, R., Sprecher, A.: PSPLIB - a project scheduling library. European Journal of Operational Research 96, 205–216 (1996)
5. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Twenty-first International Joint Conference on Artificial Intelligence, IJCAI 2009 (2009)
6. Blumofe, R.D., Joerg, C.F., Bradley, C.K., Leiserson, C.E., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 207–216 (1995)
7. Michel, L., See, A., Van Hentenryck, P.: Parallelizing constraint programs transparently. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 514–528. Springer, Heidelberg (2007)
8. Baptiste, P., Pape, C.L., Nuijten, W.: Constraint-Based Scheduling - Applying Constraint Programming to Scheduling Problems. International Series in Operations Research and Management Science. Springer, Heidelberg (2001)