Abstract. The availability of multi-processor machines and computer clusters offer significant opportunities for constraint programming. They also present a fundamental challenge: how to exploit parallelism transparently to speed up constraint programs. Our recent research showed how to parallelize constraint programs transparently on parallel computers. This paper generalizes the approach for cluster of computers and provides preliminary experimental evidence of its benefits. Indeed, on some difficult job-shop scheduling problems, the transparent parallelization provides dramatic, superlinear speed-ups ranging from 61 to 42,090 on 16 processors for depth-first search and from 9 to 48 for limited discrepancy search.

1 Introduction

Even in modest computing environments, networks of commodity “personal computers” are now standard and are being upgraded to multi-core CPUs. Together, these resources offer significant computing power which is often under-utilized. At the same time, Constraint programming (CP) search naturally offers significant opportunities for parallel computing, yet very little research has been devoted to parallel constraint programming implementations. Notable exceptions include CHIP/PEPSys [8] and its successors ECLiPSe [5], Parallel Solver [6], and Mozart [7]. One of the difficulties here is the rich search languages typically supported by modern constraint programming languages. Indeed, CP search is best described by the equation

\[
\text{CP Search} = \text{Nondeterministic Program} + \text{Exploration Strategy}
\]

indicating that a CP search procedure consists of a nondeterministic program implicitly describing the search tree and an exploration strategy specifying how to explore the search space. Our recent work in [2] addressed exactly that issue. It showed that constraint programs in COMET, organized along the equation

\[
\text{CP Search in \text{COMET}} = \text{Nondeterministic Program} + \text{Search Controller},
\]

can be parallelized transparently by lifting the search controller into a parallel search controller using work stealing to explore the search space. The resulting implementation exhibited excellent speed-ups on parallel computers for a variety
ProblemPool<CP> pool(["m1","m2"]);
parall<CP>(i in 1..pool.getSize()) {
    DistributedSolver<CP> cp(pool);
    LDS lds(cp);
    range S = 1..8;
    var<CP>{int} q[S](cp,S);
    exploreall<cp> {
        cp.post(alldifferent(all(i in S) q[i]+i));
        cp.post(alldifferent(all(i in S) q[i]-i));
        cp.post(alldifferent(q));
    } using {
        forall(i in S) by q[i].getSize()
        tryall<cp>(v in S : q[i].memberOf(v))
        label(q[i],v);
    }
}

Fig. 1. Moving from a sequential to distributed constraint program.

of applications. The CP'07 paper also mentioned that the approach should apply
to clusters of computers, but left open the feasibility and performance of the
resulting distributed implementation.

This paper settles this question. It demonstrates that a transparent parallel-
ization is also possible for networks of computers which induce much higher
communication latency and offer no support for shared memory. Moreover, the
implementation was evaluated on some difficult job-shop scheduling problems
and shows excellent speedups. Optimality proofs, which are not subject to su-
perlinear speedups, often exhibits linear speedups. When searching for optimal
solutions and proving optimality, dramatic super linear speedups are obtained.
Indeed, the speedups range from 61 to 42,090 on 16 processors for depth-first
search and from 9 to 48 for limited discrepancy search.

2 Transparent Distributed Constraint Programming

Figure 1(left) shows a simple program for the Queens problem. Line 2 creates
the solver, while line 3 creates the LDS search controller. Lines 4–9 create the
variables and post the constraints. Finally, lines 11–14 describe the search pro-
procedure: the forall loop iterates over the variables using the first-fail principle
and the tryall instruction specifies the non-deterministic choices for the values.
The right side of Figure 1 depicts the distributed version. The first three
lines have changed, but the rest of the program is left unchanged. Line 0 creates the ProblemPool<CP> responsible for storing the sub-problems shared by the
COMET processes executing on machines m1 and m2. The explicit definition of the workers could be replaced with a resource discovery
protocol. This facility has not been implemented yet but is standard.
loop with as many iterations as there are machines in the pool. (Distributed loops were discussed in the context of local search in [3]). Each iteration forks a COMET process (copying the entire COMET address space on the other machine) and executes the body of the loop. The body instantiates a distributed solver (on line 2) before executing the constraint program which is left unchanged with respect to the sequential implementation. The difference between sequential, parallel, and distributed solvers lies in their search controllers only. In the sequential case, the search controller implements the search strategy only. In the parallel and distributed cases, it implements the search strategy too but also manages work stealing and communication between the workers. The entire architecture was described in [2] and we review some of the basic concepts below.

3 Architecture and Implementation

As in [2], the architecture relies on work stealing and consists of several workers and a centralized master. The master starts remote processes for the workers and maintains a ProblemPool which is shared by all the processes and holds unexplored sub-problems. The workers alternate between exploring and publishing sub-problems. When workers exhaust their local set of open sub-problems they request more work from the ProblemPool. When the size of the pool becomes too low, the master asks workers to publish additional work. A de-centralized approach is also possible but our experiments suggested that it was not necessary for 16 workers.

The main difference between the parallel and distributed implementation is the implementation of the pool. Indeed, sub-problems produced by workers can no longer simply be shared in virtual memory since they operate on different machines. Instead, they must be shipped over the network back to the master (using sockets in our implementation) and then transmitted to starving workers (also across the network). Sub-problems are defined by the choices along a path from the root of the search tree and are thus reasonably compact. When a worker receives a new subproblem to solve, it restores the subproblem before executing the search procedure. This is the idea of semantic decomposition proposed in [4]. Note also that, on a parallel machine, all the workers can probe the problem pool regularly to decide whether to push new sub-problems into the pool. Since they share the same address space, it induces no overhead. On a distributed platform, however, the higher communication latency induces significant delays and mandates a pull model in which the master initiates the communication to obtain problems whenever it wishes to replenish its pool.

4 Experimental Results

The distributed implementation was evaluated on jobshop scheduling problems using instances LA21-26 [1] and either a DFS or LDS search strategy. The hardware is a network of AMD Athlon at 2Ghz (Athlon 3800+) with a 100 Mbit switched LAN. The heuristic in the constraint-based scheduler first selects the
machine with the smallest local slack and breaks ties using the global slack. The heuristics also ranks the resources by selecting first those tasks with the earliest completion times. The constraint-based scheduler only uses the basic edge-finding algorithm.

Table 1 (top) depicts the times and speedups for the optimality proofs. This computation is interesting since it is not subject to superlinear speedups, e.g., speedups due to finding higher-quality solutions sooner in the search. Only LA-21, LA-24, and LA-25 are shown, the proof of optimality being trivial for the other problems. The speedups are excellent, ranging from 9.25 to almost 15 on 16 machines. It is also interesting to compare the speedups of the parallel and distributed implementations. On a multicore parallel machines with 4 workers, the speed-ups are 3.28 and 3.78 on LA-24 and LA-25, while they are 3.20 and 3.53 on the distributed implementation.

Table 1 (bottom) is particularly interesting. It describes the experimental results for finding the optimal solution and proving optimality. In sequential, this search takes a long time for LA-22 and LA-24 when DFS is used, so the table reports the LDS times to give a lower bound on the speedups. (The sequential times for LDS are given in italics). The results indicates speedups ranging from 61 to 42,090 for 16 machines: the transparent parallelization thus transforms a sequential implementation that takes hours or days into a distributed implementation that takes a few seconds or a couple of minutes. Readers may wonder why the speedup stalls from 8 to 16 workers on LA-23. Shipping the address space of the Comet virtual machine to one peer over a 100Mbit network takes about 0.5 second. The running times for LA-23 with 16 workers is therefore bounded from below by this initial communication (e.g., 16 · 0.5). To improve efficiency one must switch to a gigabit network or use a multicast IP protocol.

Table 2 shows the results for finding optimal solutions and proving optimality using LDS. As expected, sequential LDS performs better than DFS. For the most part, the results show steady improvements as the number of workers increases with significant super-linear speed-ups on LA-24. The speedups on LA-23 stall once again for reasons explained above, while LA-22 and LA-25 exhibit speedups of 16 and 20.

An interesting observation about these preliminary results is that DFS outperforms LDS in a distributed environment, although this is clearly not the case in sequential. The availability of parallel and distributed hardware, and the transparent parallelization of constraint programs, may alter the benefits and limitations of exploration strategies.

5 Conclusion

This paper demonstrates that the techniques introduced in [2] to parallelize constraint programs transparently and automatically smoothly scales to a distributed environment consisting of a network of commodity hardware. The experimental results on difficult scheduling instances using two search strategies

---

4 multicast involves invasive changes to network routing and is unreliable.
Table 1. Experimental Results on Jobshop Scheduling: Proof of Optimality and Full Search for DFS. Italics indicate a comparison against the sequential LDS rather than the much slower DFS.

<table>
<thead>
<tr>
<th>Runtime (seconds)</th>
<th>Speedup (vs seq)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>4w</td>
</tr>
<tr>
<td>LA-21-P</td>
<td>5.713</td>
</tr>
<tr>
<td>LA-24-P</td>
<td>292.15</td>
</tr>
<tr>
<td>LA-25-P</td>
<td>269.58</td>
</tr>
<tr>
<td>LA-22</td>
<td>10.37</td>
</tr>
<tr>
<td>LA-23</td>
<td>464254</td>
</tr>
<tr>
<td>LA-24</td>
<td>42874</td>
</tr>
<tr>
<td>LA-25</td>
<td>23337</td>
</tr>
</tbody>
</table>

Table 2. Experimental Results on Jobshop Scheduling: Full Search (LDS).

<table>
<thead>
<tr>
<th>Runtime (seconds)</th>
<th>Speedup (vs seq)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>4w</td>
</tr>
<tr>
<td>LA-22</td>
<td>1037.29</td>
</tr>
<tr>
<td>LA-23</td>
<td>219.02</td>
</tr>
<tr>
<td>LA-24</td>
<td>42873.71</td>
</tr>
<tr>
<td>LA-25</td>
<td>3050.96</td>
</tr>
</tbody>
</table>

demonstrates the approach produces close to linear speedups on optimality proofs and remarkable speedups for finding and proving optimal solutions.

References