Arc-Consistency Algorithms on a Software DSM Platform

Marluce Rodrigues Pereira and Inês de Castro Dutra
Department of Systems Engineering and Computer Science
COPPE/UFRJ, Federal University of Rio de Janeiro, Brazil
{*marluce,ines*}*@cos.ufrj.br*

Maria Clicia Stelling de Castro
Department of Computer Science
Federal University of Juiz de Fora, Minas Gerais, Brazil
clicia@cos.ufrj.br

Abstract

This work presents the parallelisation of the AC-5 arc-consistency algorithm for distributed-shared memory platforms. We conducted our experiments using an adapted version of the PCSOS parallel system, over finite domains, running on top of Treadmarks, a software DSM system, on a cluster of 8 PCs connected via a Fast-Ethernet network. We ran four benchmarks used by the original PCSOS to debug and assess the performance of the system. Our results show that arc-consistency algorithms have a great potential for parallelisation on low cost distributed-shared memory platforms. One of the applications achieves superlinear speedups due to distributed labeling. Speedups for the other applications are limited by the write invalidate cache coherence protocol used by TreadMarks and extra synchronisation required by its memory consistency model, size of the problem, and kind of distribution of indexicals and labeling.

Keywords: arc-consistency, constraint logic programming, parallelism, software distributed shared memory.

1 Introduction

Finite domain Constraint Satisfaction Problems (CSPs) usually describe NP-complete search problems. Algorithms exist, such as arc-consistency algorithms, that help to eliminate inconsistent values from the solution space. They can be used to reduce the size of the search space, allowing to find solutions for large CSPs.

Still, there are problems whose instance size make it impossible to find a solution with sequential algorithms. Concurrency and parallelisation can help to minimise this problem because a constraint network generated by a constraint program can be split among processes in order to speed up the arc-consistency procedure.

Parallelisation of constraint satisfaction algorithms brings two advantages: (1) programs can run faster, and (2) large instances of problems can be dealt with because of the amount of resources (memory and cpus).

Several works have been done on the parallelisation and/or distribution of constraint systems. In [13, 14], authors describe logic gate level arc-consistency parallel algorithms, extending similar approaches of [23] and [7]. Gu and Sosic [12] implemented a parallel arc-consistency algorithm on a fine-grained, massively parallel hardware computer architecture. Fabiunke[9] presented a parallel framework to solve constraint satisfaction problems based on connectionist ideas of distributed information processing. Hermenegildo [16] introduced some of the problems faced by parallelising compilers for logic and constraint programs. Zhang and Mackworth developed two parallel algorithms to solve finite domain constraint satisfaction problems [24] and presented some results on parallel complexity generalising previous results achieved by Kasif and Delcher [17]. Nguyen and Deville presented a distributed arc-consistency algorithm based on the AC-4 algorithm for a distributed platform using message passing [20]. Baudot and Deville [3] proposed distributed versions of the AC-3 and AC-6 algorithms with static scheduling. Luo *et al* [19] presented heuristics to guide the search for a solution in a CSP to improve the execution time of arc-consistency algorithms. Ferris and Mangasarian [10] presented a particular technique to parallelise execution of constraints in mathematical problems. Gregory and Yang [11] have shown that good speedups can be achieved in shared-memory platforms for solving finite domain CSPs. Andino *et al* [1, 2] implemented a parallel version of the AC-5 algorithm [15] for a logically shared memory architecture, the Cray T3E, a high cost parallel platform.

Our work is based on Andino's *et al* implementation. We seek to obtain good performance on on-the-shelf low cost clusters of PCs. We adapted their algorithms and data structures to run on a distributed-shared memory platform using TreadMarks, a software Distributed Shared-Memory (DSM) system [18].

Our results show that arc-consistency algorithms can achieve good speedups on distributed-shared memory systems. One of our applications achieves superlinear speedups due to the distributed labeling.

The paper is organised as follows. Section 2 describes the AC-5 arc-consistency algorithm and its parallelisation. Section 3 describes succintly the porting of the PCSOS original system to our TreadMarks version. In section 4 we present the methodology used in our experiments. In section 5 we present our results, and finally, in section 6 we draw our conclusions and future work.

2 The AC-5 Arc-Consistency Algorithm

A CSP can be modeled through constraints that can be expressed as equations or inequations that relate variables. Each variable has an associated domain that can be finite or infinite. Arc-consistency algorithms solve CSPs over finite domains. There are several arc-consistency algorithms in the literature. Our work is based on the AC-5 algorithm that was firstly presented by Hentenryck [15]. AC-5 is a generalisation of other arc-consistency algorithms and it is parametrised on two procedures that are specified, but whose implementation are left open. It implements a constraint graph, where each edge and each node represents, respectively, a constraint and a variable.

Andino *et al* [2] implemented a parallel version of the AC-5 algorithm, called PCSOS (Parallel Constraint Sthocastic Optimisation Solver), using the *indexical scheme* [5, 21]. In this scheme, a constraint is translated into a set of *indexicals* that relate only two different variables. The execution of an indexical triggers changes in the domains of its set of related variables. These domains must be updated. The set of finite domains that keeps the current domain of each variable is called *store*.

The AC-5 algorithm presents two main steps. In the first step, all indexicals are considered once and the node-consistency is realised for each one. The store is updated and the variable related with this indexical is queued in a propagation queue. In the second step, while the propagation queue is not empty a variable is dequeued and arc-consistency is executed for all indexicals that depend on it.

The algorithm terminates when all variables are ground or when no solution is found or when we reach a fixed-point without being able to prune any more variable domain. At each arc-consistency step, if some variable is not yet ground, the algorithm enters the labeling phase to choose one non-ground variable and one of its values. During propagation, if an inconsistency is found, it is necessary to backtrack and choose another value to the non-ground variable.

3 PCSOS and PCSOS-TMK

PCSOS was implemented on a logically shared memory platform, the Cray T3E.

Our first step on porting the PCSOS to a DSM platform was to study its data structures, understand them, and separate private data from shared data. We also needed to adapt the data structures to the software DSM we used, since the PCSOS system relied on the SHMEM library [22] to access logically shared data on remote nodes. We then established the right synchronisation points in the source code in order to obtain a parallel correct code.

Besides porting the original PCSOS code to a software DSM platform, we implemented two kinds of labeling: *sequential* and *distributed*, and two kinds of partitioning of indexicals: *round-robin*, and *block*. The sequential labeling assumes that each processor can apply the labeling procedure over any variable. The distributed labeling partitions the set of variables in subsets of equal size, and each processor can execute the labeling procedure over its own subset. The round-robin partitioning of indexicals assumes that each indexical is allocated to each process at a time. The partition of indexicals in blocks assumes that a block of consecutive indexicals is allocated to each process. This partitioning is done in the beginning of the computation upon reading the input data file that contains the constraint network.

4 Methodology and Applications

Our parallel algorithm has been developed using TreadMarks, a software DSM system that implements Lazy Release Consistency (LRC), allows multiple writers, uses the page as coherence unit, and keeps caches coherent using an invalidate-based protocol. For our experiments we used a network of 8 PCs dual-pentium III with 650 Mhz of clock, 512 MBytes of main memory, 256 KBytes of cache (L2), connected by a Fast-Ethernet switch with 100 Mbits per second of bandwith, and Linux Red Hat 6.2. Although each node has two processors, we only used one processor per node. We ran each experiment 10 times and presented the average execution time in seconds. The standard deviation was less than 5%.

We have used a set of four benchmarks: Arithmetic, Sudoku, N-Queens and Parametrizable Binary Constraint Satisfaction Problem (PBCSP). These applications are the same used by Andino *et al* in their experiments.

- Arithmetic is a synthetic benchmark. It is formed by sixteen blocks of arithmetic relations, $\{B_1, \ldots, B_{16}\}$. Each block contains fifteen equations and inequations relating six variables. Blocks B_i and B_{i+1} are connected by an additional equation between a pair of variables, one from B_i and the other one from B_{i+1}. Coefficients were randomly generated. The goal is to find an integer solution vector. This kind of constraint programming is very much used for decomposition of large optimisation problems.

- Sudoku is a crypto-arithmetic Japanese problem. Given a grid of 25x25 squares, where 317 of them are filled with a number between 1 and 25, fill the rest of squares such that each row and column is a permutation of numbers 1 to 25. Furthermore, each of the twenty-five 5x5 squares starting in columns (rows) 1, 6, 11, 16, 21 must also be a permutation of numbers 1 to 25. The 25x25 grid is shown in figure 1. Filled circles represent ground variables.

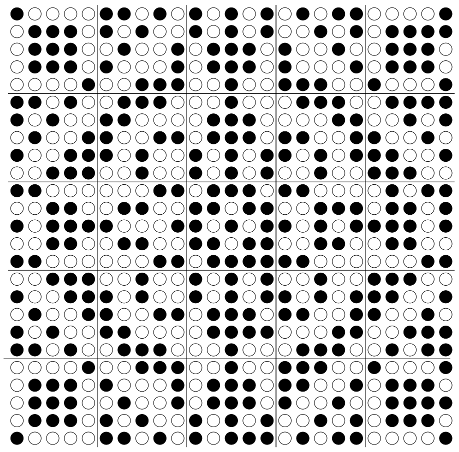

Figure 1: *Sudoku*: matrix representing the variables of the problem

- The N-Queens problem consists of placing N queens in an NxN chess board in such a way that no queen attacks each other in the same row, column and diagonal. The instance presented corresponds to N = 111, size which leads to a significant execution time.

- The Parametrizable Binary Constraint Satisfaction Problem (PBCSP) is another synthetic benchmark. Instances of this problem are randomly generated given four parameters: number of variables, the size of the initial domains, density, and tightness. Density and tightness are defined as follows: $\frac{nc}{nv-1}$ and $1 - \frac{np}{ds^2}$, respectively, where nv is the number of variables, nc is the number of constraints involving one variable (it is the same for all variables), np is the number of pairs that satisfies the constraint, and ds is the size of the initial domains.

  In our experiments, we used two different sets of variables containing 100 and 200 variables, with tightness 75% and domain size equals to 20.

Table 1 presents a summary of the main characteristics of our applications. They spend from 100 to 10,000 times more time executing the arc-consistency algorithm than executing the labeling procedure. We consider this benchmark as a representative set of CSPs. Their constraint graphs range from weakly connected to totally connected.

Characteristics	Applications				
	Arithmetic	PBCSP_1	PBCSP_2	*Queens*	*Sudoku*
Variables	126	100	200	111	308
Constraints	254	3,713	7,463	6,105	13,942
Indexicals	1,468	7,426	14,925	12,210	27,884

Table 1: Applications Characteristics

5 Results

For each application we show execution times in seconds, speedups, total of acquires, messages, and barriers executed, and total of indexicals and failures executed per each processor. The total number of acquires, barriers and messages were obtained from the TreadMarks statistics. The number of acquires and barriers correspond to the number of calls to the *Tmk_lock_acquire* and *Tmk_barrier* functions, respectively. The total number of messages corresponds to the number of exchange of bytes related to application data or to the TreadMarks protocol. The total number of failures corresponds to the number of times backtracking was called after the labeling phase.

In the next sections, we analyse the performance of each application, in decreasing order of performance.

5.1 *Arithmetic*

This application has a constraint graph weakly connected whose pattern of connection is regular, i.e., for each pair of blocks there are only two edges connecting them: one edge connecting a block of equations to the next, and one edge connecting the next block to the previous. We explored the structure of this graph in order to do a better distribution of indexicals and labeling among processors. Since the problem is structured in blocks of equations, we distributed the indexicals in blocks. Besides, each processor labels its own subset of variables, communicating only when pruning a variable which connects different blocks allocated to different processors.

Table 2 shows the execution times of the application *Arithmetic* for 1, 2, 4 and 8 processors. Figure 2 shows the speedup curve related to this table.

Number of Processors	Execution Times (sec.)
1	4.74
2	0.39
4	0.37
8	0.47

Table 2: *Arithmetic*: Execution Times for 1, 2, 4 and 8 Processors

This application presents superlinear speedups when we explore the constraint graph and distribute the labeling among processors. Note that on parallelising the sequential algorithm we naturally achieve distributed labeling, because we take advantage of the constraint graph structure.

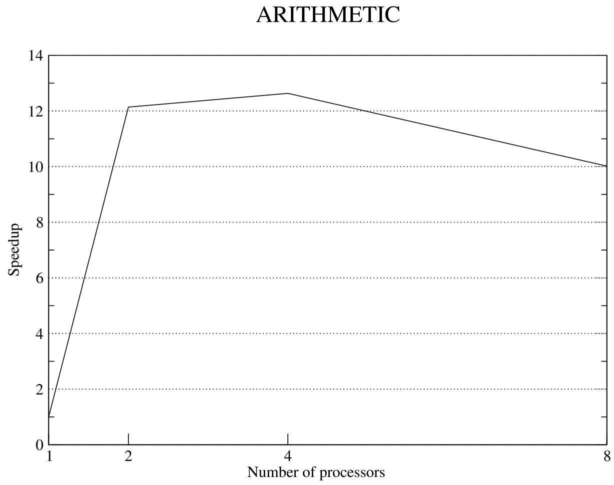

Figure 2: *Arithmetic*: Speedups

For two processors, the execution time is 12.14 times less than the execution time for one processor. The lowest speedup we obtain with this application is 10.01, for 8 processors related to the execution time for one processor. From 2 to 4 processors we obtain a discrete improvement in performance, and from 4 to 8 processors we obtain a slowdown. Because of the cache coherence protocol (invalidate) and the memory consistency model (LRC) used by TreadMarks, the overheads for this application are too high, when we increase the number of processors.

The cache coherence protocol employed by TreadMarks forces the invalidation of connecting variables that will probably be touched later. This means that the next processor to touch the variable will have to wait for its most recent value. As we increase the number of processors, the number of connecting variables also increases, the invalidation effect will become worse with the addition of more processors.

The memory consistency model forces the parallel programmer to introduce extra synchronisation (when compared with a sequential consistency model) in order to obtain updated copies of the touched data. As this extra synchronisation is located in the main loop of the arc-consistency source code, this causes more network traffic as we increase the number of processors.

Yet another factor that contributes to the low scalability of this application is the load balancing produced by the distribution of the indexicals and labeling. As mentioned before, distributed labeling partitions the set of variables in subsets of equal size. This may cause load imbalance. Table 3 shows the total number of indexicals executed per processor, and illustrates the load imbalance caused for 4 and 8 processors.

Proc#	1	2	4	8
0	1,953,660	89,255	23,363	2,392
1	-	44,071	25,457	20,917
2	-	-	4,607	15,001
3	-	-	13,665	10,437
4	-	-	-	3,237
5	-	-	-	2,069
6	-	-	-	19,566
7	-	-	-	3,543
Total of ind.	1,953,660	133,326	62,892	77,162

Table 3: *Arithmetic*: Total number of indexicals executed per processor

This load imbalance can only be removed by doing a more sophisticated distribution of labeling among the processors.

The last factor that impedes a better performance of this application is the size of the problem. We believe that if we increase the size of this problem, we could have more scalable results.

In order to perform a fairer comparison between one processor and more than one processor, we ran this application concurrently in just one processor, in order to isolate the benefits caused by distributed labeling alone. Table 4 shows the execution times in seconds for this experiment. We compare the results of labeling with 2 processes in one processor with 2 processes on 2 processors, 4 processes on one processor with 4 processes on 4 processors, and so forth. This experiment shows two aspects of the labeling: (1) as expected and confirmed by other works in the literature, arc-consistency algorithms produce much better results when we partition the set of variables to be labeled, (2) parallelisation of labeling is worthwhile. Our best result shows that we can obtain an improvement of 42% when we parallelise the labeling partitioned into four sets.

Number of processes	1 processor	n processors	*Speedup*
2	0.55	0.39	1.40
4	0.53	0.37	1.42
8	0.59	0.47	1.24

Table 4: *Arithmetic*: 1 processor and n processors

5.2 PBCSP

Our second application, *Parametrizable Binary Constraint Satisfaction Problem* (PBCSP), has a constraint graph that is strongly connected, and does not present a regular connection pattern, i.e., the edges are randomly generated to form the final constraint graph. Therefore, most of our experiments with this application, were done using sequential labeling. Our choice for the partitioning of indexicals was round-robin. We ran this application with two input data, containing 100 variables (PBCSP_1), and 200 variables (PBCSP_2).

Table 5 and figure 3 show the execution times and speedups, respectively of PBCSP_1 and PBCSP_2.

Number of Processors	Execution Times (sec.)	
	PBCSP_1	PBCSP_2
1	26.55	39.36
2	20.13	29.54
4	26.38	26.03
8	65.22	50.41

Table 5: PBCSP_1 and PBCSP_2: Execution Times for 1, 2, 4 and 8 Processors

The minimum execution time for PBCSP_1 was achieved with 2 processors, which yielded a speedup of 1.31 related to the execution time for one processor. We manage to keep a discrete speedup related to one processor up to 4 processors. The performance degrades for 8 processors.

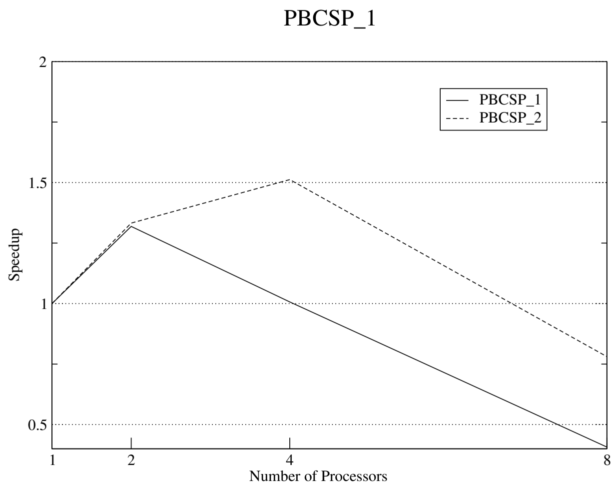

Figure 3: PBCSP_1 and PBCSP_2: Speedups

PBCSP_2 scales well up to 4 processors, yielding a speedup of 1.51, and starts to degrade for 8 processors. In this case, the parallel time is 28% greater than the sequential time.

We can observe that, as we increase the data size, we obtain an improvement in performance. The lack of scalability of this application is caused by the same reasons mentioned for the application *Arithmetic*: cache coherence protocol of TreadMarks, extra synchronisation, input data sizes, and load imbalance.

As the constraint network graph for this application is strongly connected, several indexicals distributed among processors share the same variables. This causes an extra network traffic in the cluster, which degrades performance as we increase the number of processors.

For this application, the load imbalance caused by our indexical distribution and by the sequential labeling is very high for 8 processors, for PBCSP_2, as can be seen in table 6.

Proc	4		8	
	PBCSP_1	PBCSP_2	PBCSP_1	PBCSP_2
0	143,907	197,983	73,377	89,371
1	144,645	226,786	73,583	135,042
2	137,993	196,468	73,490	83,699
3	139,652	193,257	71,782	127,350
4	-	-	75,023	86,453
5	-	-	75,785	135,846
6	-	-	73,469	87,415
7	-	-	75,612	128,080
Tot.	566,197	814,494	592,121	873,256

Table 6: PBCSP_1 e PBCSP_2: Total number of indexicals executed per processor

Although the graph connection pattern for this application is not regular, we managed to obtain results for 2 processors with PBCSP_1, experimenting with distributed labeling and different kinds of partitioning of indexicals. Table 7 shows the results for these experiments. As can be observed, we improved the average execution time from 20.13 seconds, from our original implementation using sequential labeling and partitioning of indexicals round-robin (LSPRR), to 19.70 seconds combining sequential labeling with partitioning of indexicals in blocks (LSPB). Our best improvement of 86% was achieved when combining distributed labeling with partitioning of indexicals round-robin (LDPRR). By combining distributed labeling with partitioning of indexicals in blocks (LDPB) we obtained an improvement of 77% related to our original implementation LSPRR.

Combinations	Execution Times (sec.)
LSPRR	20.13
LSPB	19.70
LDPRR	10.81
LDPB	11.31

Table 7: PBCSP_1: 2 processors, different kinds of labeling and partitioning of indexicals

5.3 *Queens*

The *Queens* application has a totally connected constraint graph. Therefore, its execution times increase with the increase of processors, because of the excessive communication overhead. Table

8 shows the execution times for this application, for 1, 2, 4 and 8 processors.

Number of Processors	Execution Times (sec.)
1	9.26
2	104.97
4	381.27
8	1,243.21

Table 8: *Queens*: Execution Times for 1, 2, 4 and 8 Processors

The labeling for this application is done sequentially, because its constraint graph is not trivially partitioned among the processors. In order to obtain a good partitioning for the labeling we need a more sophisticated analysis of the constraint graph. The partitioning of indexicals chosen was round-robin.

The communication overhead and the increase in the amount of work done in parallel compared to the uniprocessor version are the main performance limiting factors. Table 9 shows the total number of acquires, messages, barriers and failures for this application for 1, 2, 4 and 8 processors. The number of barriers executed and number of failures are the same for all numbers of processors, because the labeling is done sequentially. The number of messages for one processor is zero, because there is no message circulating in the network for one processor. As we can observe, the number of messages and acquires increase when we increase the number of processors.

Statistics	Processors			
	1	2	4	8
Acquires	157,296	1,918,354	5,856,510	7,353,994
Messages	0	1,635,691	6,996,274	28,645,600
Barriers	13,142	13,142	13,142	13,142
Failures	4,473	4,473	4,473	4,473

Table 9: *Queens*: Total number of acquires, messages, barriers and failures

The amount of work increases from 4 to 8 processors. The total number of indexicals executed increased from 808,497 to 843,695. This increase combined with the communication overheads contributes to higher execution times.

5.4 *Sudoku*

The last application studied, *Sudoku*, has a constraint graph that is strongly connected, whose pattern is not regular. As the execution time for this application increased almost 50 times from 1 to 2 processors, we just ran it up to 4 processors. Table 10 shows the execution times for this application with 1, 2 and 4 processors.

One of the reasons to the high execution time for 2 processors is that the number of indexicals executed increased in more than 600,000 for 2 processors in contrast to 1 processor. This increase definitely augmented the search space related to 1 processor. Table 11 shows the distribution of indexicals executed per processor.

Another reason for the poor performance of this application is the huge amount of network traffic generated by the TreadMarks protocol and extra synchronisation, as shown in table 12. This application is the only one with a high number of barriers executed, because its execution

Number of Processors	Execution Times (sec.)
1	35.03
2	1,489.83
4	1,670.99

Table 10: *Sudoku*: Execution Times for 1, 2 and 4 Processors

Proc#	1	2	4
0	9,765,277	5,593,249	2,609,039
1	-	4,776,026	2,551,971
2	-	-	2,673,049
3	-	-	2,354,200
Total of ind.	9,765,277	10,369,275	10,188,259

Table 11: *Sudoku*: Total number of indexicals executed per processor

time and search space is huge. The number of failures for this application is also very high what suggests that this application could benefit from an or-parallel search.

Statistics	Processors		
	1	2	4
Acquires	353,216	24,002,551	22,491,934
Messages	0	21,810,801	127,997,156
Barriers	108,297	108,297	108,297
Failures	36,092	36,097	36,085

Table 12: *Sudoku*: Total number of acquires, messages, barriers and failures

Yet another factor that limits the performance of this application, as was mentioned for the other applications, is the kind of partitioning and labeling performed. As the constraint graph of this application needs more sophisticated partitioning, we did not perform distributed labeling for this application. However, we experimented changing the kind of partitioning of this application from round-robin to block partitioning for 2 processors and obtained an improvement of 24% related to the 1 processor execution time. This improvement is not only caused by the kind of partitioning of indexicals, but also because of a decrease in the total amount of parallel work.

5.5 Discussion

Andino *et al* managed to obtain speedups for all applications without using distributed labeling. For the application *Arithmetic*, the maximum speedup was 3.0 for 8 processors. PBCSP_1 presented maximum speedup of 18.0 for 34 processors. *Queens* presented maximum speedup of 4.0 for 16 processors, and *Sudoku* presented speedup of 2.3 for 18 processors. Their experiments were done in a high performance architecture with a high bandwidth and low latency processor interconnection network. The Cray T3E interprocessor network has 480 MBytes per second of nominal capacity (230 MBytes per second in practice) while our interprocessor network achieves only 100 Mbits per second. The Cray T3E also provides efficient user library facilities for shared memory programming through the SHMEM library. This library provides fast atomic reads and writes to remote memories. This counts as an advantage to Andino *et al*'s experiments. We believe that different kinds of labeling and partitioning would produce better results in their architecture. They only used round-robin partitioning of indexicals and sequential labeling. Given our hardware platform restrictions and kind of software DSM system we employed, we consider our results very positive.

We achieved superlinear speedups for the application *Arithmetic*, and achieved best speedup for the application PBCSP_1 of 1.86, with 2 processors, using distributed labeling and partitioning of indexicals round-robin, which leads to 93% of efficiency. We still have several opportunities for improvement.

The poor performance produced by some applications is a direct consequence of the cache coherence protocol used by TreadMarks, its memory consistency model that forces extra synchronisation, size of applications, load imbalance, and kinds of labeling and distribution of indexicals among the processors.

Regarding the cache coherence protocol, previous experiments on hardware DSM system has shown that hybrid protocols, where update-based protocols are switched to invalidate-based protocols dynamically, can yield better performance than pure invalidate-based protocols [4, 8]. Other experiments on simulation of adaptive software DSMs also suggests that other kind of platform can improve our results [6].

6 Conclusions and Future Work

This work presented the parallelisation of the arc-consistency algorithm AC-5 for a software DSM platform on a cluster of 8 PCs. We ran four benchmarks already used in the literature to assess our experiments. We implemented two kinds of partitioning of indexicals: round-robin and block, and two kinds of labeling: sequential and distributed. For one application we achieved superlinear speedups, because of the distributed labeling. Our best speedup was achieved for the PBCSP_1 application using distributed labeling and partitioning of indexicals round-robin, for 2 processors. The reasons for the low performance of some experiments are directly related to the excessive communication overheads caused by the cache coherence protocol used by TreadMarks, its memory consistency model that forces extra synchronisation, size of the input data, and load imbalance caused by the kinds of labeling and distribution of indexicals performed in this work.

We still have many opportunities for improvements. Better organisation of shared data structures as well as algorithm restructuring could help to improve performance. Suitable software or hardware DSM systems that uses hybrid cache coherence protocols also could help to improve performance. Better analysis of the constraint graph could lead to a better distribution of labeling and indexicals.

Work is under way to accomplish these tasks and produce higher speedups for these and other constraint satisfaction applications, including experiments on faster interprocessor networks such as Giganet, Gigabit and Myrinet.

Acknowledgments

The authors would like to thank Dr. Ruiz-Andino for his help with the CSOS system used in this paper. He kindly gave us his C source code and benchmarks. We would also like to thank Dr. Amorim who granted us the utilisation of his cluster of PCs to realise our experiments. This work benefitted from useful comments from Dr. Santos Costa. Inês Dutra would like to thank the CNPq CLoP^n project.

This work is partially supported by the Brazilian Research Council CNPq, and by Capes.

References

[1] A. R. Andino, L. Araujo, and J. Ruz. Parallel Solver for Finite Domain Constraints. Technical Report SIP 71.98, Department of Computer Science. Universidad Complutense de Madrid, 1998.

[2] A. R. Andino, L. Araujo, F. Sáenz, and J. Ruz. Parallel Execution Models for Constraint Programming Over Finite Domains. In *Principles and Practice of Declarative Programming*, pages 134–151, 1999.

[3] B. Baudot and Y. Deville. Analysis of Distributed Arc-Consistency Algorithms. Technical Report RR 97-07, University of Louvain, Belgium, 1997.

[4] V. M. Calegario and I. C. Dutra. Performance Evaluation of Or-Parallel Logic Programming Systems on Distributed Shared Memory Architectures. In *Proceedings of the EUROPAR'99*, pages 1484–1491, Aug-Sep 1999.

[5] B. Carlson. *Compiling and Executing Finite Domain Constraints*. PhD thesis, University of Uppsala, Department of Computer Science, 1995.

[6] M. C. S. Castro and C. L. Amorim. Efficient categorization of memory sharing patterns in software dsm systems. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, page 63, April 2001.

[7] Z. Collin, R. Dechter, and S. Kartz. On the Feasibility of Distributed Constraint Satisfaction. In *Proceedings of the 12th IJCAI*, pages 318–324, 1991.

[8] I. C. Dutra, V. Santos Costa, and R. Bianchini. The Impact of Cache Coherence Protocols on Parallel Logic Programming Systems. In *International Conference on Computational Logic*, volume 1861, pages 1285–1299, 2000. Lecture Notes in Artificial Intelligence.

[9] M. Fabiunke. Parallel Distributed Constraint Satisfaction. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 99) - Las Vegas*, pages 1585–1591, June 1999.

[10] M. C. Ferris and O. L. Mangasarian. Parallel Constraint Distribution. *SIAM Journal on Optmization*, 4:487–500, 1991.

[11] S. Gregory and R. Yang. Parallel constraint solving in andorra-i. In *Proceeding of the Fifth Generation Computer Systems*, pages 843–850, June 1992.

[12] J. Gu and R. Sosic. A Parallel Architeture for Constraint Satisfaction. In *International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 229–237, June, 1991.

[13] H. W. Guesguen. Connectionist Networks for Constraint Satisfaction. In *AAAI Spring Symposium on Constraint-based Reasoning*, pages 182–190, 1991.

[14] H. W. Guesguen and J. Hertzberg. A Perspective of Constraint-Based Reasoning. In *Lecture Notes on Artificial Intelligence, Springer-Verlag*, 1992.

[15] P. V. Hentenryck, Y. Deville, and C. M. Teng. A Generic Arc-Consistency Algorithm and its Specifications. *Artificial Intelligence*, 57(2–3):291–321, October 1992.

[16] M. Hermenegildo. Parallelizing Irregular and Pointer-Based Computations Automatically: Perpectives from Logic and Constraint Programming. *Parallel Computing*, 26(13-14), 2000.

[17] S. Kasif. On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *Artificial Inteligence*, 45:275–328, 1990.

[18] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix*, pages 115–131, 1994.

[19] Q. Y. Luo, P. G. Hendry, and J. T. Buchanan. Heuristic search for distributed constraint satisfaction problems. Research Report KEG-6-93, Department of Computer Science. University of Strathclyde, 1993.

[20] T. Nguyen and Y. Deville. A Distributed Arc-Consistency Algorithm. In *Science of Computer Programming*, pages 227–250, 1998.

[21] V. A. Saraswat. Concurrent Constraint Programming Languages. In *MIT Press*, 1993.

[22] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *ASPLOS7*, pages 26–36, October 1996.

[23] M. J. Swain and P. R. Cooper. Parallel Hardware for Constraint Satisfaction. In *Proceedings of the National Conference on Artificial Intelligence*, pages 682–686, 1988.

[24] Y. Zhang and A. K. Mackworth. Parallel and distributed algorithms for constraint networks. Technical Report 91.6, Department of Computer Science. The University of British Columbia, Canada, 1991.