Parallelisation of Arc-Consistency Algorithms

[1] **M. R. Pereira and [1] I. Dutra and [2] M. C. S. de Castro**
[1] Department of Systems Engineering and Computer Science
COPPE/UFRJ, Federal University of Rio de Janeiro, Brazil
{*marluce,ines,clicia*}*@cos.ufrj.br*
[2]Department of Infomartics and Computer Science
Rio de Janeiro State University, Rio de Janeiro, Brazil

Abstract

This work presents the parallelisation of the AC-5 arc-consistency algorithm for two different parallel architectures. One is a cluster of PCs and the other is a centralised memory machine (CMM). We conducted our experiments using an adapted version of the PCSOS parallel constraint solving system, over finite domains. The implementation for the cluster of PCs uses Treadmarks (TMK), a software distributed shared memory (SDSM) system. On the CMM we use synchronisation based on atomic read-modify-write primitives supported in hardware. We ran four benchmarks used by the original PCSOS to assess the performance of the system. We implemented different kinds of partitioning for the constraints, and different kinds of distributed labeling that are not present in the original version. Our results show that arc-consistency algorithms have very good speedups on centralised memory systems, and have a great potential for parallelisation on low cost distributed-shared memory platforms. We showed that performance of the benchmarks are greatly affected by these kinds of partitioning and distributed labeling. One of our applications achieves superlinear speedups due to distributed labeling. Speedups for the cluster of PCs are limited by the write invalidate cache coherence protocol used by TMK and extra synchronisation required by its memory consistency model, size of the problem, and kind of distribution of indexicals and labeling. Speedups for the CMM are better than for the cluster, however this platform is more expensive than a cluster.

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

1 Introduction

Finite domain Constraint Satisfaction Problems (CSPs) usually describe NP-complete search problems. The arc-consistency algorithms 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 speedup 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. We cite some works close to our work. Sosic [8] implemented a parallel arc-consistency algorithm on a fine-grained, massively parallel hardware computer architecture. Zhang and Mackworth developed two parallel algorithms to solve finite domain CSPs [16] and presented some results on parallel complexity. Nguyen and Deville presented a distributed arc-consistency algorithm based on the AC-4 algorithm for a distributed platform using message passing [13]. Baudot and Deville [3] proposed distributed versions of the AC-3 and AC-6 algorithms with static scheduling. Luo *et al.* [12] presented heuristics to guide the search for a solution in a CSP to improve the execution time of arc-consistency algorithms. Ferris and Mangasarian [6] presented a particular technique to parallelise execution of constraints in mathematical problems. Gregory and Yang [7] 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 [9] for a logically shared memory architecture, the Cray T3E, a high cost parallel platform.

We seek to obtain good performance on a low cost clusters of PCs based on Andino *et al.*'s implementation. We adapted their algorithms and data structures to run on a distributed-shared memory platform using TreadMarks (TMK), a software Distributed Shared-Memory (SDSM) system [11]. We also performed some experiments on a higher cost architecture, a centralised memory machine (CMM), the Enterprise 4500, in order to assess the performance of our applications in both platforms.

Andino *et al.* implemented only sequential labeling and round-robin partitioning of indexicals. We added one more kind of labeling, distributed, and one more kind of partitioning, in blocks. We intended to show that different kinds of labeling and partitioning of indexicals contribute to improve performance.

Our results show that arc-consistency algorithms can achieve good speedups on both platforms. One of our applications achieves superlinear speedups due to the distributed labeling. Our best results achieve linear speedups at 7.97 out of 8 processors on the CMM version.

The paper is organised as follows. Section 2 describes the AC-5 arc-consistency algorithm and its parallelisation. Section 3 describes succinctly the porting of the PCSOS original system to our TMK and CMM versions. 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 relations involving variables with associated domain that can be finite or infinite.

Arc-consistency algorithms, such as the AC-5 [9], solve CSPs over finite domains. The AC-5 algorithm was first presented by Hentenryck [9]. 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* [4, 14]. 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: 1) all indexicals are considered once and the node-consistency is performed for each one, and 2) 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 finishes 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, PCSOS_TMK and PCSOS_SHM

Our first step on porting the PCSOS to a SDSM platform (PCSOS_TMK version) 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 SDSM we used, since the PCSOS system relied on the SHMEM library [15].

Also, 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.

The version we used for the CMM (PCSOS_SHM) is similar to the cluster version. Synchronisation in this platform is obtained through atomic read-modify-write primitives supported in hardware and taken from the Linux kernel. On the CMM, we removed the synchronisation for reading shared data, because the consistency memory model is sequential.

4 Methodology and Applications

We used two platforms to make our experiments. One is a network of 16 PCs dual-pentium III with 650 Mhz, 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. We used TMK, a SDSM system that implements Lazy Release Consistency (LRC), allows multiple writers, uses the page as coherence unit,and uses an invalidate-based protocol. We only used one processor per node. The second platform is a Sun Enterprise 4500 with 4 GBytes of memory, 14 Ultrasparc processors with 400 Mhz, connected by a Gigaplane bus at 100 Mhz, that delivers up to 3.2 GBytes per second, and Solaris 2.8. Synchronisation in this platform is obtained through atomic read-modify-write primitives supported in hardware and taken from the `spinlock.h` include file used in the Linux kernel for Ultrasparc processors.

Our experiments were run on 8 processors in both platforms. We ran each experiment 10 times and presented the average execution time in seconds. The standard deviation was less than 5% in both platforms. 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. More details about these applications can be found in [2]. **Arithmetic** is a synthetic benchmark. This kind of constraint programming is very much used for decomposition of large optimisation problems.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 **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.In our experiments, we used two different sets of variables containing 100 and 200 variables, with tightness 75% and domain size equals to 20. **Sudoku** is a crypto-arithmetic Japanese problem. Given a grid of 25x25 squares, there are some circles filled and others not. The 25x25 grid is shown in Figure 1. Filled circles represent ground variables.

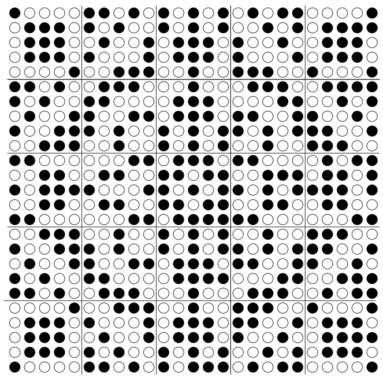

Figure 1: *Sudoku*: matrix representing the variables of the problem

Table 1 summarises 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 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

5.1 Results on the Distributed-Shared Memory Platform

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 TMK 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 TMK protocol. The total number of failures corresponds to the number of times backtracking was called.

Arithmetic. This application has a constraint graph weakly connected. 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. The processors communicate only when pruning a variable which connects different blocks allocated to different processors. Figure 2 shows the table execution times of the application *Arithmetic* and 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

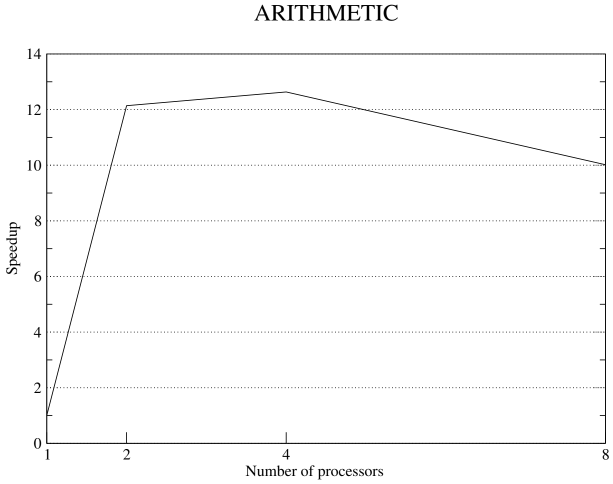

Figure 2: *Arithmetic*: Execution Times for 1, 2, 4 and 8 Processors and Speedups for the Cluster of PCs

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.

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. Because of the cache coherence protocol (invalidate) and the memory consistency model (LRC) used by TMK, it 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. Distributed labeling partitions the set of variables in subsets of equal size. This may cause load imbalance. Table 2 shows the total number of indexicals executed per processor, and illustrates the load imbalance caused for 4 and 8 processors.This load imbalance can only be removed by doing a more sophisticated distribution of labeling among the processors. 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 3 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.

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 2: *Arithmetic*: Total number of indexicals executed per processor for the Cluster of PCs

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 3: *Arithmetic*: 1 processor and n processors for the Cluster of PCs

PBCSP. Our second application PBCSP has a constraint graph that is strongly connected. 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, PBCSP_1 (100 variables), and PBCSP_2 (200 variables). Figure 3 shows their execution times and speedups.

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

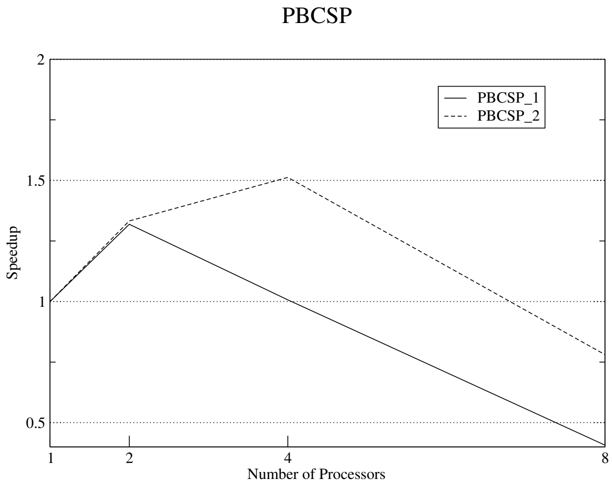

Figure 3: PBCSP_1 and PBCSP_2: Execution Times for 1, 2, 4 and 8 Processors and Speedups for the Cluster of PCs

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. PBCSP_2 scales well up to 4 processors, yielding a speedup of 1.51, and starts to degrade for 8 processors.

As the constraint network graph for PBCSP 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. Table 4 shows the very high load imbalance for 8 processors, for PBCSP_2, caused by our indexical distribution and by the sequential labeling.

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 4: PBCSP_1 e PBCSP_2: Total number of indexicals executed per processor for the Cluster of PCs

Although the PBCSP's graph connection pattern 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 5 shows the results for these experiments. 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 5: PBCSP_1: Different kinds of labeling and partitioning of indexicals for the Cluster of PCs

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 6, on the left, shows the execution times for *Queens*. On the right side, we can see the Table 7 that shows the total number of acquires, messages, barriers and failures for this application.

Number of Processors	Execution Times
1	9.26
2	104.97
4	381.27
8	1,243.21

Table 6: *Queens*: Execution Times for 1, 2, 4 and 8 Processors for the Cluster of PCs

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 7: *Queens*: Total number of acquires, messages, barriers and failures for the Cluster of PCs

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 7). 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. As we can observe, the number of messages and acquires increase when we increase the number of processors.

The amount of work increases from 4 to 8 processors. This increase combined with the communication overheads contributes to higher execution times.

Sudoku. The *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 8 shows the execution times for this application and Table 9 shows the distribution of indexicals executed per processor.

Number of Processors	Execution Times
1	35.03
2	1,489.83
4	1,670.99

Table 8: *Sudoku*: Execution Times for 1, 2 and 4 Processors for the Cluster of PCs

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 9: *Sudoku*: Total number of indexicals executed per processor for the Cluster of PCs

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.

Another reason for the poor performance of *Sudoku* is the huge amount of network traffic generated by the TreadMarks protocol and extra synchronisation (Table 10). *Sudoku* is the only one with a high number of barriers executed, because its execution 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 10: *Sudoku*: Total number of acquires, messages, barriers and failures for the Cluster of PCs

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.

5.2 Results on the Centralised Memory Platform

For each application we show execution times in seconds, speedups, and total of indexicals executed per each processor. The total number of acquires and barriers are similar to the implementation in the cluster, except that the implementation for the CMM does not need synchronisation for reading shared data.

Arithmetic. Figure 4 shows the execution times and the speedups achieved by *Arithmetic* application. Compared with the speedups achieved in the cluster of PCs, we obtained an extraordinary improvement. We managed to achieve much higher superlinear speedups (from a maximum of 12.6 for 4 processors in the cluster to a maximum of 83 for 8 processors on the CMM).

The speedups related to 1 processor improved from the cluster to the CMM version because the cluster has a very high synchronisation overhead compared with the CMM. This improvement is also due to the memory consistency model implemented by TMK. It requires extra synchronisation for reading shared data.

Num. of Processors	Execution Times (sec.)
1	8.57
2	0.41
4	0.12
8	0.10

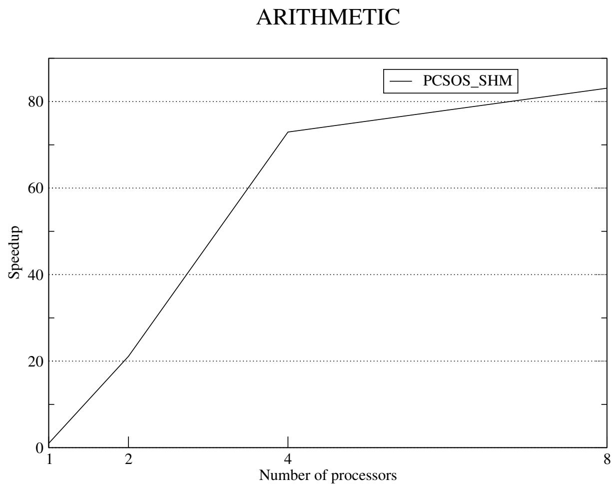

Figure 4: *Arithmetic*: Execution Times for 1, 2, 4 and 8 Processors and Speedups for the CMM

Observing the speedup graph, we may conclude that this application is not scalable up to 8 processors. However we still have room for improvements, because the input data size for this application can be increased in order to keep the processors busy, and the labeling can be better designed to obtain a better load balance. This issue is beyond the scope of this paper, since design of better labeling procedures is a hard problem. We have been investigating this problem in order to produce better speedups for our applications.

Table 11 shows the number of indexicals executed per processor. Compared with the cluster version, we can observe that the load balance is very similar. However on the CMM version we had a decrease in total load from 4 to 8 processors. The improvement in performance on the CMM for 8 processors is due to two factors: the decrease of load and the use of less synchronisation.

Proc#	1	2	4	8
0	1,953,660	89,255	23,363	2,392
1	-	44,077	25,459	9,227
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	133,332	67,094	65,472

Table 11: *Arithmetic*: Total number of indexicals executed per processor for the Enterprise

PBCSP. Figure 5 shows execution times and it has linear speedups in this platform, for the application PBCSP for the two input data of 100 (PBCSP_1) and 200 (PBCSP_2) variables. This application had limited performance in the cluster due to the overheads introduced by TMK. We can also observe from Table 12 that the load for 8 processors is much better balanced among the processors in this platform.

Queens. The *Queens* application produced better results on the CMM than on the cluster as can be seen in Table 13. As this application has a totally connected constraint graph, we did not manage to have a better performance using sequential labeling and round-robin partitioning of indexicals. The load balancing for this application running on the CMM is very similar to the one obtained on the cluster.

Sudoku. *Sudoku* produced very bad results, mainly because of the constraint graph that has a very irregular connection pattern. For both platforms, the performance was very poor. Table 14 shows the execution times for 1, 2, and 4 processors. This table confirms that the overheads introduced by TMK is very high. For

Number of Processors	Execution Times (sec.)	
	PBCSP_1	PBCSP_2
1	40.42	52.13
2	20.61	26.36
4	10.02	14.14
8	5.07	6.97

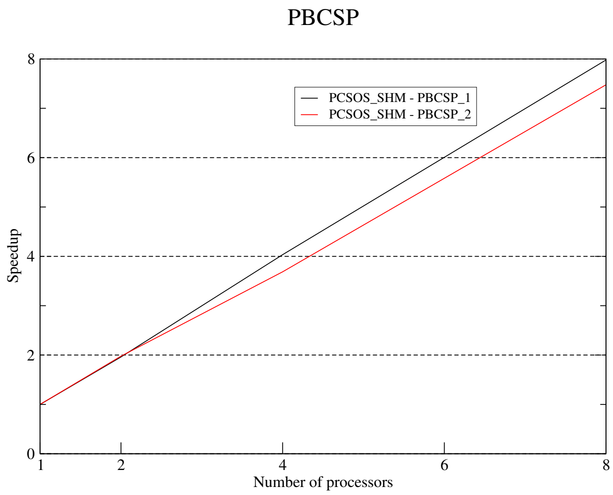

Figure 5: PBCSP_1 and PBCSP_2: Execution Times for 1, 2, 4 and 8 Processors and Speedups for the CMM

Proc	4		8	
	PBCSP_1	PBCSP_2	PBCSP_1	PBCSP_2
0	152,764	199,540	69,853	93,248
1	159,961	203,961	77,347	97,258
2	157,138	202,935	76,655	97,992
3	158,108	203,228	77,204	97,884
4	-	-	77,776	97,754
5	-	-	77,331	97,695
6	-	-	78,037	97,148
7	-	-	77,077	97,784
Tot.	627,971	809,664	611,290	776,763

Table 12: PBCSP_1 e PBCSP_2: Total number of indexicals executed per processor for the Enterprise

the same reasons presented for the *Queens* application, we need more sophisticated labeling and partitioning procedures in order to obtain better performance.

5.3 Discussion

In contrast with previous Andino *et al.*'s work, our parallel PCSOS versions are unique since they use different kinds of partitioning for the constraints, and different kinds of distributed labeling. For the application *Arithmetic*, Andino *et al.* obtained the maximum speedup 3.0 out of 8 processors, while we achieve superlinear speedups in both platforms because of our distributed labeling.

In our PBCSP_1 implementation we obtained almost linear speedups (7.97) up to 8 processors for the CMM version. We used sequential labeling and partitioning of indexicals round-robin, which practically

Number of Processors	Execution Times
1	10.63
2	10.45
4	8.22
8	8.62

Table 13: *Queens*: Execution Times for 1, 2, 4 and 8 Processors for the CMM

Number of Processors	Execution Times
1	60.39
2	95.77
4	106.44
8	121.19

Table 14: *Sudoku*: Execution Times for 1, 2 and 4 Processors for the CMM

corresponds to 100% of efficiency. We still have several opportunities for improvement.

We and Andino *et al.* did not achieve good speedups for *Queens* and *Sudoku* applications. Distributing the labeling and doing a better partitioning of indexicals is a hard problem that we have been investigating in order to obtain better load balancing and better performance for constraint applications.

Andino *et al.* experiments were done in a high performance architecture with a high bandwidth and low latency processor connection network, 3D-torus. This counts as an advantage to their experiments.We have shown that we can obtain better performance changing the labeling and partitioning algorithms. We believe that different kinds of labeling and partitioning would produce better results in their architecture.Given our hardware platforms restrictions and kind of SDSM system we employed, we consider our results very positive. We can still improve these results by implementing a more sophisticated labeling procedure and indexical distribution, and fine tuning data structures and the algorithms.

Regarding the cache coherence protocol, others experiments on simulation of adaptive SDSMs also suggests that other kind of platform can improve our results [5].

6 Conclusions and Future Work

This work presented the parallelisation of the arc-consistency algorithm AC-5. We conducted our experiments in two platforms: an CMM and a SDSM platform, a cluster of 8 PCs running TMK. We ran four benchmarks already used in the literature to assess our experiments up to 8 processors. 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 in both architectures, because of our distributed labeling. Our best speedup (7.97) was achieved for the PBCSP_1 application, on the CMM, using sequential labeling and partitioning of indexicals round-robin, for 8 processors. The reasons for the low performance of some experiments on the cluster are directly related to the excessive communication overheads caused by the cache coherence protocol used by TMK, 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. The last two factors also impedes better performance on the CMM version.

We still have many opportunities for improvements. Better organisation of shared data structures as well as algorithm restructuring could help to improve performance. Suitable SDSM 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 Myrinet.

Acknowledgments. The authors would like to thank Dr.Ruiz-Andino for his help with the PCSOS system. He kindly made available his C source code and benchmarks. We would also like to thank Dr. Amorim who granted us the utilisation of his cluster of PCs. Dr. Pontelli granted us the utilisation of the CMM available in New Mexico State University. We are very grateful for his kindness. 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 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] B. Carlson. *Compiling and Executing Finite Domain Constraints.* PhD thesis, University of Uppsala, Department of Computer Science, 1995.

[5] 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.

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

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

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

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

[16] 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.