Load Balancing in a Distributed-Memory Or-Parallel System

Bo-Ming Tong
Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
email: bmtong@cs.arizona.edu

Ho-Fung Leung
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, New Territories, Hong Kong.
email: lhf@cs.cuhk.hk

December 4, 1996

Abstract

We consider or-parallel logic programming implementations on parallel machines with no shared-memory. Traditional implementation techniques as employed in Aurora and Muse are not applicable. In our or-parallel execution model, all processors perform identical work initially. At each choice point, processors are divided evenly among alternatives of the choice point. Backtracking is employed if there are not enough processors for such a division. As execution proceeds, the division of processors among alternatives becomes uneven. In this paper, we present two different methods of load balancing called *equalization* and *apportion*, aimed at improving the degree of parallelism. Equalization and apportion reallocates all processors to the or-parallel branches by copying heaps through a interprocessor communication network.

1 Introduction

We consider or-parallel logic programming implementations on parallel machines with no shared-memory. Traditional implementation techniques as employed in Aurora [Lusk *et al.*, 1990] and Muse [Ali and Karlsson, 1990b] are not applicable. In our or-parallel execution model, all processors perform identical work initially. At each choice point, processors are divided evenly among alternatives of the choice point. Backtracking is employed if there are not enough processors for such a division. As execution proceeds, the division of processors become uneven. Some or-parallel branches possess a lot of processors, while some other or-parallel branches run out of processors and have to resort to backtracking. To address this problem, we present two different methods of load balancing called *equalization* and *apportion*, aimed at improving the degree of parallelism. Equalization and apportion reallocates all processors to the or-parallel branches by copying heaps through a interprocessor communication network. We implemented equalization and apportion in *Firebird* [Tong and Leung, 1993; Tong and Leung, 1994; Tong and Leung, 1995], a data-parallel concurrent constraint programming system for DECmpp and Maspar massively parallel computers. The next section is a review of Firebird. Equalization and apportion, together with some preliminary performance figures, are presented in section 3. Section 4 is a comparison to previous research.

2 Firebird: A Review

2.1 The Firebird Computation Model

In Firebird, a program consists of a number of clauses and every clause is divided into a *guard* part and a *body* part by a commit operator in the same way as the concurrent logic programming language *flat GHC* [Ueda, 1985]. Execution consists of two alternating derivation steps, *indeterministic derivation* and *nondeterministic derivation*. In an *indeterministic derivation step*, execution consists of guard tests, commitment, output unification and spawning in the same manner as committed-choice logic programming languages. In a *nondeterministic derivation step*, a choice point based on one of the domain variables in the system is set up and all possible values in its domain are attempted in an or-parallel manner. The domain variable used in a nondeterministic derivation step is said to be *labeled*[1] [Van Hentenryck, 1989] and each or-parallel branch is called a *partition*.

2.2 Exploitation of Data-Parallelism in Firebird

In a nondeterministic derivation step, the labeled domain variable becomes a vector of all possible values of its domain. Goals and constraints take these argument vectors arising from the labeling operation for the exploitation of data-parallelism.

To illustrate how Firebird exploits data-parallelism, it is helpful to trace the execution of 4-queens using the query `queen(4,[X1,X2,X3,X4])`. Here

[1] *Labeling* a domain-variable means instantiating a domain-variable by attempting each value in its domain one by one or in parallel.

X_i represents the position of the queen in the i-th column. We assume that all atoms have been reduced by indeterministic derivation. Only constraints remain in the system and they are shown in Figure 1. At this point, all domain variables have the same initial domain {1,2,3,4}.

X1 $\neq$ X2	X1 $\neq$ X2 + 1	X1 $\neq$ X2 - 1
X1 $\neq$ X3	X1 $\neq$ X3 + 2	X1 $\neq$ X3 - 2
X1 $\neq$ X4	X1 $\neq$ X4 + 3	X1 $\neq$ X4 - 3
X2 $\neq$ X3	X2 $\neq$ X3 + 1	X2 $\neq$ X3 - 1
X2 $\neq$ X4	X2 $\neq$ X4 + 2	X2 $\neq$ X4 - 2
X3 $\neq$ X4	X3 $\neq$ X4 + 1	X3 $\neq$ X4 - 1

Figure 1: Constraints remaining just before the first nondeterministic derivation

If we label X1 using nondeterministic derivation, form a vector with the 4 possible values of X1 and try the 4 possible values in a data-parallel fashion, we can evaluate the first 9 constraints with an ideal 4-times speedup[2] on a SIMD machine. Because the value of X1 is now known, the domains of other variables can be deduced using the constraints in Figure 1. Then, a second nondeterministic derivation will occur. If every branch chooses to create a choice point on X2, there will be $2+1+1+2 = 6$ branches (see Figure 2). Thus the next 9 constraints can be solved with an ideal speedup of 6. Thousands of processor elements can be fully utilized easily in this way because many problems are combinatorial in nature.

2.3 Mapping Partitions to Processor Elements

In order to avoid data movement among the processor elements, a single *logical partition*, or simply a *partition*, is mapped to a number of identical *physical partitions*. Each physical partition corresponds to a single processor element of a data-parallel computer, and we use the two terms interchangeably. Initially, all processor elements execute exactly the same initial logical partition. If the data-parallel computer has N processor elements and a choice point with 4 alternatives is created, $\frac{N}{4}$ processor elements will be allocated to each alternative. A trace of the execution of 4-queens on a machine with 8,192 processor elements is shown in Figure 2. The processor elements mapped to each partition are shown under each chessboard. Since a *processor-id* is associated with each processor element, each processor element can compute which alternative it should take autonomously.

[2] This is just a rough approximation. Amdahl's Law dictates that such an "ideal" speedup cannot be obtained in practice.

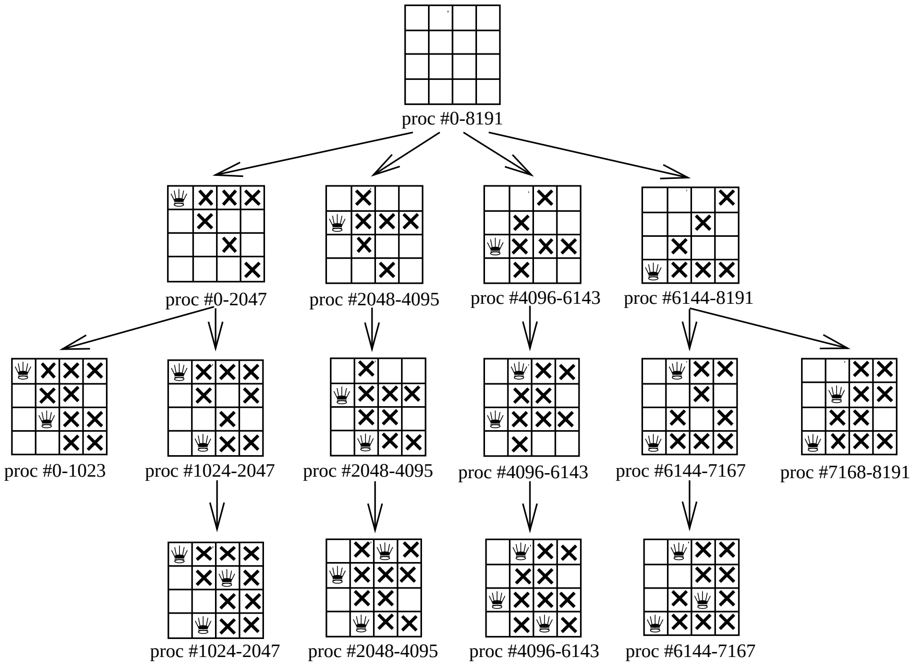

Figure 2: Example: execution of 4-queens

3 Load Balancing

Firebird avoids inter-processor communication with a processor allocation strategy which divides processor elements among possible alternatives. However, this heuristics is much a matter of guesswork. It is difficult to estimate the exact number of processor elements needed before execution. Consequently, after several nondeterministic derivation steps, some partitions may still have a lot of active processor elements while some other partitions have very few active processor elements. On the other hand, some partitions may have succeeded or failed and its processor elements can be freed up and reallocated to other partitions in need. The reallocation is achieved by an (optional) machine-dependent operation called *equalization*. In an *equalization* operation, processor elements of the machine are reallocated and divided evenly among the logical partitions which have not yet succeeded or failed. Equalization and branch-and-bound are the only two operations in the Firebird system which require interprocessor communication. Compared to *SIMD MultiLog* [Smith, 1993], Firebird has the advantage of lower inter-processor communication overhead.

3.1 The DECmpp Massively Parallel Computer

The Firebird language is implemented on a DECmpp [Blank, 1990], which consists of a front-end UNIX workstation and a back-end *data-parallel unit*. The data-parallel unit in turn consists of an *array control unit* (ACU), a *processor element array* (PE) and an inter-processor communication network which supports both mesh and arbitrary communication patterns. The ACU dispatches a single instruction stream to the processor elements. In addition, it broadcasts data to the processor elements and receives the logical or-ing of data from the processor elements. A processor element may choose to execute or ignore an instruction based on its *contingent bit*. Each processor element has its own local memory and the processor elements must use the inter-processor communication networks to communicate with each other.

3.2 Equalization

We describe the DECmpp-specific implementation of the *equalization* operation. Before equalization, some partitions have a lot of processor elements, some other partitions have only a few processor elements, and some partitions have already succeeded or failed. The *equalization* operation frees up the processor elements originally allocated to the succeeded or failed logical partitions, and reallocates them to active logical partitions. The detailed operations are as follows.

1. We count and number the active partitions using the DECmpp system library call *enumerate()* (Figure 3a). The width of a partition in the figure represents the number of processor elements it has. The leftmost processor element of each active partition is called a *source*.

2. Processor elements are divided evenly among the active partitions (Figure 3b). The leftmost processor element of each resulting chunk is called a *destination*. Each processor element checks (in parallel) if it is a destination by dividing its *processor-id* by the total count of active partitions obtained from the *enumerate()* call.

3. After the sources and destinations are identified, each *destination* processor element fetches the entire heap and argument stack from a *source* processor element (Figure 3c) through DECmpp's *router* hardware. The router network allows an arbitrary pattern of point-to-point interprocessor communication. For clarity, the sources and destinations are shown as two separate boxes but in fact they are the same set of processor elements.

4. Upon receiving each word of data, each *destination* processor element is responsible for distributing the data to the processor elements to its right with DECmpp's *xnet pipelined copy* operation (Figure 3d). *Xnet* is a two-dimensional mesh network which connects each processor element with its eight nearest neighbors, N, NE, E, SE, S, SW, W, NW. *Pipelined copy* sends data along some direction for some distance d, and leaves a copy at each intermediate processor element the data travels through.

Equalization is applied there is a choice point but there are not enough processor elements for all possible alternatives in the choice point, except if there have been backtrackings before. Equalization after backtrackings is future work and the main difficulty is that currently we copy entire heaps from processor elements to processor elements, overwriting old heaps, but the old heaps must be restored upon backtracking.

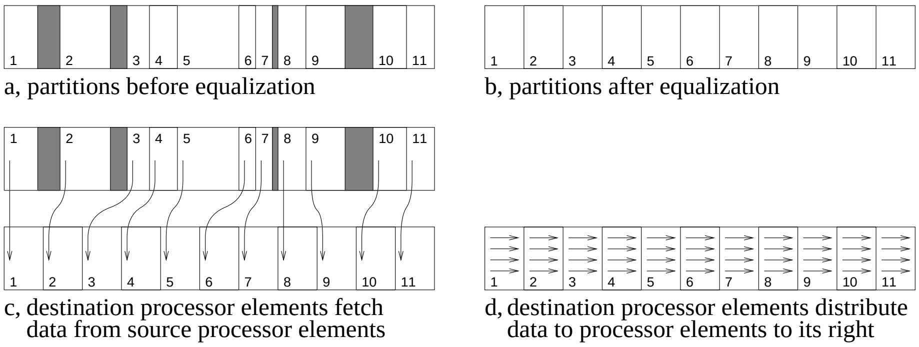

Figure 3: Equalization on DECmpp

3.3 Apportion

Apportion is a variation of equalization. Rather than sharing processor elements among partitions equally, apportion allocates processor elements according to some heuristics. Each partition asks for a number of *shares*. For example, if there are 5 partitions and they ask for 3, 5, 4, 6 and 2 shares respectively, a total of 20 shares are needed. The processor elements are divided into 20 shares and the shares are allocated to the 5 partitions accordingly. The implementation of apportion is similar to that of equalization.

1. The DECmpp system library call *scanAdd()* is used in place of *enumerate()*. *scanAdd()* numbers each partition by adding up the needs of partitions to its left. In the last example, *scanAdd()* numbers the 5 partitions as 0, 3, 8, 12 and 18 respectively.

2. The processor elements are evenly divided into shares. The source processor elements send their processor-id to the destination processor elements of share 0, share 3, share 8, share 12 and share 18 respectively.

3. The destination processor elements fetch data from source processor elements and distribute the data to its right as in *equalization*. The length of distribution for each destination processor element is different (because now the size of each resulting partition is different) but the DECmpp hardware will stop the distribution automatically as soon as it hits the next destination processor element.

How does each partition determine how many shares it asks for ? The apportion operation is performed when there is a choice point but there are not sufficient processor elements. Therefore, each partition knows the number of alternatives in the choice point. If there are n alternatives, the partition asks for n shares. Compared to equalization, apportion looks one step ahead and ensures that after the next choice point the partitions will have equal numbers of processor elements. If there are not enough processor elements to satisfy the requests of all partitions, the apportion operation is aborted and the system resorts to backtracking.

3.4 Preliminary Results

The performance of equalization and apportion is shown in Table 1. The *overhead* column indicates the percentage of time spent on equalization or apportion. P is the number of partitions measured at the end of the execution and BT is the number of backtrackings. `eq10` is a program taken from the *clp(FD)* [Diaz and Codognet, 1993] distribution, which solves 10 linear equations over 7 variables. t_1, t_2, t_3 are execution time in seconds to find all solutions.

- Although equalization and apportion are heavy operations which involve copying entire heaps and stacks through the interprocessor communication

benchmark	using apportion t_1	overhead	t_3/t_1	using equalization t_2	overhead	t_3/t_2	no equalization or apportion t_3
eq10	.293	10.2%	1.17	.288	8.5%	1.19	.344
queen(9)	.202	22.6%	.90	.199	22.5%	.91	.182
queen(10)	1.066	5.7%	1.44	1.089	6.1%	1.41	1.540
queen(11)	15.545	.007%	1.00	15.154	.7%	1.03	15.544
queen(12)	106.229	.001%	1.00	104.437	.2%	1.02	106.227

	using apportion		using equalization		no equalization or apportion	
benchmark	P	BT	P	BT	P	BT
eq10	99	0	99	0	181	8
queen(9)	770	0	770	0	2313	3
queen(10)	3936	15	3924	16	2399	17
queen(11)	2763	158	2779	164	2763	158
queen(12)	4161	1201	4183	1154	4161	1201

Test conditions: $\#proc$=8,192, eager bit vector creation, eager nondeterministic derivation, no solitary memory access, no priority scheduling.

Table 1: Benchmark: Equalization and Apportion

network, the overhead is as serious as one would expect because they are used only once. Applying equalization or apportion more than once does not further reduce the number of backtrackings or execution time significantly, and we left out the uninteresting results.

- For most programs, equalization and apportion does not significantly affect performance. For `queen(11)` and `queen(12)`, the number of shares requested by partitions in an apportion is greater than the total number of processor elements (8,192). Therefore, the apportion operation is aborted, leading to very low overhead figures (the only overhead is that of checking) but no effect on execution time and number of backtrackings.
- For `queen(10)`, the degree of parallelism is improved by equalization and apportion as can be seen in P, the number of partitions at the end of execution. As a result, the number of backtrackings is reduced and execution time is improved.
- For `queen(9)`, although the unoptimized benchmark shows 2313 partitions, it is an indication of poor processor element utilization rather than an indication of a high degree of parallelism. Before equalization or apportion, there are 2312 partitions whose sizes ranges from 1 to 13 processor elements, but only 600 of them have not succeeded or failed yet. An equalization or an apportion frees up the 1712 useless partitions (totaling 5953

processor elements). Similarly, `eq10` has a lot of failed partitions which got freed up in an equalization or an apportion. For all the other programs, most partitions are still active at the point of equalization or apportion, and the number of partitions freed is insignificant.

4 Comparison

Traditional or-parallel logic programming research focuses on the allocation of a small number of processor elements called *workers* to a large number of or-parallel branches called *jobs*. Work on scheduling and load balancing strategies for this style of or-parallelism include [Butler *et al.*, 1988; Calderwood and Szeredi, 1989; Hausman, 1989; Szeredi and Carlsson, 1990; Ali and Karlsson, 1990a; Ali and Karlsson, 1991; Szeredi *et al.*, 1991; Karlsson and Ali, 1992; Ali and Karlsson, 1992; Beaumont and Warren, 1993; Sindaha, 1993]. These schedulers assign jobs to idle workers who have finished their jobs at hand. Some aim at minimizing context switching overhead, and others allow *speculative work* to be scheduled, the principle of using idle processors to work on jobs which we are not sure whether are useful in the future. Speculative work may become useless if the or-parallel branch subsequently gets pruned by the Prolog cut operator, but otherwise, some execution time will be saved.

In all of the above papers, it is assumed that the processor elements have shared-memory and the overhead for a processor to switch from one job to another is low. The assumptions of our work are different. Processor elements are plentiful on a massively parallel computer, but memory is distributed. A processor element cannot switch from one job to another efficiently. As a result, we propose an execution model in which work is duplicated on many processor elements. At each choice point, processor elements are evenly divided among alternatives of the choice point. This strategy is not effective at balancing loads to processor elements because some or-parallel branches get a lot of processor elements but others starve for processor elements, but this strategy successfully avoids interprocessor communication overhead. The work described in this paper is operations to even out the number of processor elements for each or-parallel branch, on our implementation platform, a DECmpp 12000 Sx massively parallel computer.

5 Conclusion

We present two different load balancing methods, called *equalization* and *apportion*, for Firebird, describe their implementations and show some preliminary performance results. We found that although both *equalization* and *apportion* are heavy weight operations they incur an acceptable overhead because they are used sparingly. In most cases they do not significantly affect performance but in some special cases they help. We are surprised that sometimes *equalization* gives better results than *apportion* given that intuitively the latter seems to be the better of the two methods.

References

[Ali and Karlsson, 1990a] K. A. M. Ali and R. Karlsson. Full Prolog and scheduling or-parallelism in MUSE. *International Journal of Parallel Programming*, 19(6):445–475, December 1990.

[Ali and Karlsson, 1990b] K. A. M. Ali and R. Karlsson. The MUSE approach to or-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.

[Ali and Karlsson, 1991] K. A. M. Ali and R. Karlsson. Scheduling Or-parallelism in MUSE. In K. Furukawa, editor, *ICLP'91, International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 807–821, Paris, 1991. The MIT Press.

[Ali and Karlsson, 1992] K. A. M. Ali and R. Karlsson. Scheduling speculative work in MUSE and performance results. *International Journal of Parallel Programming*, 21(6), December 1992.

[Beaumont and Warren, 1993] T. Beaumont and D. H. D. Warren. Scheduling speculative work in or-parallel Prolog systems. In D. S. Warren, editor, *Logic Programming: Proceedings of the Tenth International Conference*, pages 135–149, Budapest, Hungary, 1993. The MIT Press.

[Blank, 1990] T. Blank. The Maspar MP-1 architecture. In *Proceedings of the IEEE COMPCON Spring 1990*, pages 20–24, San Francisco, U.S.A., February 1990. IEEE.

[Butler *et al.*, 1988] R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek, and R. Stevens. Scheduling OR-Parallelism: An argonne perspective. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 1590–1605, Seattle, U.S.A., 1988. ALP, IEEE, The MIT Press.

[Calderwood and Szeredi, 1989] A. Calderwood and P. Szeredi. Scheduling or-parallelism in aurora: The manchester scheduler. In G. Levi and M. Martelli, editors, *Logic Programming: Proceedings of the Sixth International Conference*, pages 419–435, Lisbon, 1989. The MIT Press.

[Diaz and Codognet, 1993] D. Diaz and P. Codognet. A minimal extension of the WAM for clp(FD). In D. S. Warren, editor, *Logic Programming: Proceedings of the Tenth International Conference*, pages 774–790, Budapest, Hungary, 1993. The MIT Press.

[Hausman, 1989] B. Hausman. Pruning and scheduling speculative work in or-parallel Prolog. In *PARLE'89, Conference on Parallel Architectures and Languages Europe*, 1989.

[Karlsson and Ali, 1992] R. Karlsson and K. A. M. Ali. The engine-scheduler interface used in the MUSE or-parallel Prolog system. SICS Research Report R92:04, Swedish Institute of Computer Science, March 1992.

[Lusk *et al.*, 1990] E. Lusk, D. H. Warren, S. Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[Sindaha, 1993] R. Y. Sindaha. Branch-level scheduling in Aurora: The Dharma scheduler. In D. Miller, editor, *Logic Programming: Proceedings of the 1993 International Symposium*, pages 403–419, Vancouver, Canada, October 1993. The MIT Press.

[Smith, 1993] D. A. Smith. MultiLog: Data or-parallel logic programming. In D. S. Warren, editor, *Logic Programming: Proceedings of the Tenth International Conference*, pages 314–331, Budapest, Hungary, 1993. The MIT Press.

[Szeredi and Carlsson, 1990] P. Szeredi and M. Carlsson. The engine-scheduler interface in the Aurora or-parallel Prolog system. Technical Report TR-90-09, University of Bristol, Computer Science Department, April 1990.

[Szeredi *et al.*, 1991] P. Szeredi, M. Carlsson, and R. Yang. Interfacing engines and schedulers in or-parallel Prolog systems. In *PARLE'91, Conference on Parallel Architectures and Languages Europe*, number 506 in Lecture Notes in Computer Science, 1991.

[Tong and Leung, 1993] B. M. Tong and H. F. Leung. Concurrent constraint logic programming on massively parallel SIMD computers. Technical Report CS-TR-93/04, The Chinese University of Hong Kong, July 1993.

[Tong and Leung, 1994] B. M. Tong and H. F. Leung. Implementation of a data-parallel concurrent constraint programming system. In *Proceedings of the First International Symposium on Parallel Symbolic Computation*, pages 382–393, Linz, Austria, September 1994. World Scientific.

[Tong and Leung, 1995] B. M. Tong and H. F. Leung. Performance of a data-parallel concurrent constraint programming system. In *Proceedings of the 1995 Asian Computing Science Conference*, Pathumthani, Thailand, December 1995. Springer-Verlag. Volume 1032 of Lecture Notes in Computer Science.

[Ueda, 1985] K. Ueda. Guarded horn clauses. In E. Wada, editor, *Logic Programming '85 — Proceedings of the 4th Conference*, Lecture Notes in Computer Science 221, pages 168–179, Tokyo, Japan, July 1985. Springer-Verlag.

[Van Hentenryck, 1989] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.