Distributed, Scalable, and Static Parallel Arc Consistency Algorithms on Private Memory Machines*

James M. Conrad and Dharma P. Agrawal
Electrical and Computer Engineering
North Carolina State University, Box 7911, Raleigh, NC 27695-7911
conrad/dpa@csl.36h.csl.ncsu.edu

Abstract

The use of straightforward backtracking algorithms in handling constraint satisfaction is inadequate because they exhibit an excessive amount of thrashing for large problems. This can be avoided if pre-processing is performed to eliminate variable assignments which can never participate in a solution. Arc consistency algorithms can pre-process a network of constraints before the tree search. This paper introduces three scalable Static Parallel Arc Consistency algorithms (DSPAC-1, DSPAC-2 and DSPAC-3) designed for any general-purpose distributed memory computer, or for a network of computers. Through actual machine experimentation we measure time required for the DSPAC algorithms and compare it with existing sequential algorithms. Results indicate that our parallel arc consistency algorithms are very effective and scalability can be efficiently maintained.

1 Introduction

Constraint Satisfaction Problems (CSPs) are prevalent in artificial intelligence applications. Versions of CSPs occur in areas such as image processing, resource scheduling, semantic information processing, and database retrieval. Mackworth [5] describes CSPs as “those [problems] in which one has a set of variables each to be instantiated in an associated domain and a set of Boolean constraints limiting the set of allowed values for specified subsets of the variables”. Back-tracking is often used to solve these CSPs. Straightforward backtracking algorithms, however, are inadequate for handling large constraint networks found in typical AI applications because they exhibit an excessive amount of thrashing. To avoid this, one can pre-process the problem by eliminating variable assignments which can never participate in a solution. This can be achieved by using node, arc, and path consistency algorithms [5] to filter a network of constraints before the tree search.

No one, to our knowledge, has proposed and demonstrated a distributed arc consistency algorithm. In this paper we introduce three new scalable Static Parallel Arc Consistency algorithms designed for any general-purpose distributed memory computer or network of computers. We measure time required to execute our algorithms on an Intel iPSC/2 multicomputer, and results show that our parallel arc consistency algorithm can be effectively used to pre-process a constraint network. We also address scaling our algorithm to solve larger Artificial Intelligence problems.

2 Consistency Algorithms

To describe consistency algorithms, we refer to the network model of a CSP. An example is shown in Figure 1. We also rely on the following definitions:

Definition 1: A variable, \( i \), is a unit which is assigned a meaning, or value. There are \( n \) variables in a problem.

Definition 2: A finite domain, \( D_i \), is a list of possible values that variable \( i \) can be given. We assume that each domain has the same cardinality (size). The size of each domain is \( a \).

Definition 3: A value, \( l_i \), also called a label, is a specific assignment given to variable \( i \) from a list of possible items, that is, \( l_i \in D_i \).

Definition 4: A binary constraint, \( R_{ij} \), is a relation by which if a value \( l_i \) from domain \( D_i \) is assigned to variable \( i \), and value \( l_j \) from domain \( D_j \) is assigned to variable \( j \), then the constraint \( R_{ij} \) is true if the relation is satisfied, that is \( (l_i, l_j) \in R_{ij} \).

---

*This work has been supported in part by the Army Research Office, under grant DAAL03-89-K-0142.
Definition 5: An arc $arc(i, j)$ is arc consistent if, for each value in the domain $D_i$ for variable $i$, there is at least one value in $D_j$ that satisfies the constraint of the arc. Arcs can be made consistent by performing the operation $D_i := D_i \cap \{h_j \mid \exists (h_i, h_j) \in R_{ij} \land h_j \in D_j\}$ for every specific assignment $h_i$ given to variable $i$ from a domain $D_i$ of possible items, where $R_{ij}$ is a binary constraint relation containing tuples of values satisfying the constraint. The number of arc of a problem is $2e$.

Definition 6: A consistency check is a comparison of a possible value of one variable with the possible value of another variable. We use consistency checks to analyze algorithm performance.

Definition 7: The complexity of a sequential algorithm is the maximum number of steps the algorithm may take. These steps are identified as a basic unit of work, like an addition operation or a comparison of two values. Our step is a consistency check.

Definition 8: The time complexity of a parallel algorithm is the maximum number of steps, as a measure of time, that are required to solve the problem. This can be viewed as "start-to-stop" time. Some or all of the processors running a parallel algorithm may be active at any one time. Therefore, the computational complexity is the maximum number of steps, as a measure of work, that a parallel algorithm will perform. This is often the time complexity times the number of processors.

2.1 Sequential Arc Consistency Algorithms

Two popular sequential arc consistency (AC) algorithms are Mackworth’s AC-1 and AC-3 [5]. Both the AC-1 and AC-3 algorithms use a Revise subroutine. Revise examines an arc by choosing a variable value and checking if the other variable attached to the arc has at least one value which is consistent with the value of the first variable. This is done for each value in the first variable’s domain.

AC-1 calls Revise for every arc of the constraint network graph. If Revise removes a value from any variable’s domain, a change bit is set. Once AC-1 has finished checking each arc, it examines the change bit. If the bit is set, AC-1 rechecks all of its arcs again. AC-1 ends when it checks all of its arcs and does not remove any values.

AC-1 is a simple but inefficient algorithm. It is clear that AC-1’s inefficiencies lie in the fact that when only one value is removed, all variables are checked again. To avoid the shortcomings of AC-1, AC-3 builds a queue by adding all of the arcs of the constraint network $G$ to the queue, $Q$. If Revise removes a value from a variable, AC-3 rechecks the arcs which are attached to that variable (except the current arc and node consistent $arc(i, i)$) by adding the arcs to the queue. This means, given that a value in $D_i$ is removed when processing $arc(i, j)$, then $Q := Q \cup \{arc(m, i) \mid arc(m, i) \in G, m \neq i, m \neq j\}$. AC-3 terminates when the queue is empty.

2.2 Existing Parallel Arc Consistency Algorithms

Samal [7] introduced several parallel versions of arc consistency algorithms for a shared memory environment with an infinite number of processors. Although his algorithms can not be directly applied to distributed-memory computers, we have used many of his ideas while implementing our algorithms.

Samal employs a parallel version of Mackworth’s Revise, called Previse. In Previse, all consistency checks of a single arc are performed concurrently.

Samal’s parallel version of Mackworth’s AC-1 is Parallel Arc Consistency algorithm #1 (PAC-1). Although this algorithm requires a large number of processors, PAC-1 ensures consistency quickly since all
possible variable assignments of the 2e network arcs are checked simultaneously. Previser executes up to \( O(2e) \) consistency checks concurrently, therefore each iteration of the PAC-1 loop requires \( O(2e) \) processors.

The parallel version of AC-3 is PAC-3. PAC-3 builds a globally available queue like AC-3, and uses Previser like AC-1. Again, only arcs in the queue are checked, and so we can check up to 2e arcs simultaneously. Generally, the PAC-3 queue is much smaller than the set of all edges, while in the worst case PAC-3 behaves just like PAC-1. A drawback of PAC-3 is that the time to build the queue is not negligible and may affect the total computational time.

We have also introduced several parallel arc consistency algorithms for shared memory machines [1, 2, 3] and in this paper we consider similar distributed algorithms.

3 Distributed Static Parallel Arc Consistency

We have developed three parallel arc consistency algorithms which are similar to Mackworth’s AC-1 and AC-3 algorithms. Unlike previous approaches, our algorithms uniquely assign large parts of a CSP to specific processors. This unique characteristic of our algorithms defines a relationship between constraint variables and parallel processor architectures. These assignments allow our algorithms to run on any MIMD computer, and allow us to scale a problem simply by adding more processors. Each processor of our target multiprocessor ensures arc consistency for the assigned part of the network. In the following discussions, we interpret a distributed memory computer to be a distributed memory multiprocessor or a network of computers.

We statically assign one or more constraint variables to each processor. With one constraint variable per processor the distribution of work is not necessarily equitable. If we assign several constraint variables to each processor, we can ensure that the number of arcs per processor is approximately equal.

Since we can not share variables, we designed our algorithms to best exploit the nature of a distributed memory architecture as follows:

- Read-only variables are stored on each processor. Only those variables needed by each processor are stored.
- Variables which are read by all processors, but written by one processor are stored on each processor. Each processor starts with the same variable values, but these are changed once the processing commences. If a processor changes one of these variables, it sends a message with the changes to all those processors which use that variable.
- Each processor has one variable which is written by all processors, but read only by itself. This variable is easily maintained by receiving messages from other processors with instructions to change the variable. Since this is a logical variable, we only need to perform OR or AND operations.
- One variable is read and written by all processors and indicates when all processors are done. It is maintained on the host processor. Messages from the node processors to the host processor instruct the host to increment or decrement the variable’s value. When the variable indicates that all processors are done, the host gathers the results from node processors.

Our three Distributed Static Parallel Arc Consistency Algorithms, DSPAC-1, DSPAC-2, and DSPAC-3, utilize a procedure called Static Parallel Revise Boolean function (Spreviser), which is similar to Mackworth’s Revise function. Spreviser checks an arc and eliminates all variable values which could never participate in a solution. If a value is removed, it informs the calling DSPAC algorithm of a change. Spreviser also checks to ensure that the variable domain still has at least one value. If it removes the last value from a domain, it sends a no_solution message to all processors, and the DSPAC algorithm stops.

3.1 DSPAC Algorithm #1

The DSPAC-1 algorithm (Figure 2) has been split into two components, the host algorithm and the node algorithm. The host component resides in only one processor and assigns work to other processors. The host maintains the change bit and determines if the arcs need to be rechecked. The host component of the DSPAC-1 algorithm starts the node component of the algorithm. Each processor works with one or more assigned constraint network variables and the arcs which emanate from those variables. Thus, arcs assigned to a processor are checked for consistency.

The DSPAC-1 algorithm is similar to Mackworth’s AC-1 algorithm. One difference between DSPAC-1 and AC-1 is that our algorithm ends only when all processors have verified local arc consistency. Once each processor has verified local arc consistency of the variables and arcs it was assigned, it waits for the host
procedure DSPAC-1-HOST()
begin
assign variables to processors;
change := TRUE; no_solution := FALSE
while (change AND no_solution) do
begin
change := FALSE;
for each processing node p, change := change OR DSPAC-1-NODE(p);
gather results;
end

procedure DSPAC-1-NODE(p)
begin
for each assigned variable i do
for each directed arc(i,j) do
begin
local_change := FALSE;
local_change := Sprevise(arc(i,j));
change := change OR local_change;
if (local_change)
for each arc(i,j), where j is on processor m
send(Dj) to m;
end
return(change);
end

Figure 2: DSPAC-1 Algorithm: Host Processor and Node Processor Components.

to check consistency of the entire network. If any processor has set the change bit in the host to “on,” all processors again ensure local arc consistency by checking all of its arcs.

The versatility of this parallel algorithm lies in the fact that domain changes are available to all processors almost immediately. Each node processor has an interrupt-driven receive procedure which reads domain change messages and updates its own local memory. Therefore, values which are removed are never checked again by the Sprevise function. One disadvantage, however, is that the processors are somewhat loosely synchronized. Finished processors sit idle until all processors have completed checking their arcs.

Theorem 1: The time complexity of DSPAC-1 is \( O(a^3ne/p) \).

Proof: Assume that we run the DSPAC-1 algorithm on a computer with \( p \) processors, each processor is assigned \( 2e/p \) constraint arcs. The possibility exists that only one value is removed from the entire constraint problem during each DSPAC-1-NODE call, and that being during the consistency check of the last variable’s value. Since each DSPAC-1-NODE call makes up to \( a^2 \) consistency checks in Sprevise, each call in each processor takes up to \( O(a^2e/p) \) time units. If one value is removed from one of the \( p \) processors each call, \( O(na) \) values can be removed from the entire constraint network. Therefore, the time complexity is \( O(a^3ne/p) \). \( \square \)

Theorem 2: The computational complexity of DSPAC-1 is \( O(a^3ne) \).

Proof: If the worst case time complexity is \( O(a^3ne/p) \), there is a possibility that all processors are active all the time. In this case, the complexity of work performed is \( p \) times the time complexity, or \( O(a^3ne) \). \( \square \)

These complexities represent the extreme worse case. On the average (as we show with the experimental results in Section 4.3), the actual time and work performed is substantially smaller.

3.2 DSPAC Algorithm #2

DSPAC-1 also operates in the same manner as PAC-1 and AC-1, where all arcs are checked if one value in the network has been removed from a variable’s domain. Therefore, DSPAC-1 still suffers from the inefficiencies of the other algorithms. To better exploit the distributed nature of AC-1 and reduce the amount of unnecessary processing, we have developed another parallel arc consistency algorithm, DSPAC-2.

The DSPAC-2 algorithm (Figure 3) has been split into host and node components. The host part executes in only one processor and assigns work to other processors. The host maintains the active_procs variable. This variable is used by all processors to verify network arc consistency. The host component of the DSPAC-2 algorithm starts the node component of the algorithm. Each node processor has an interrupt-driven receive procedure which reads domain change messages, updates its own memory, and updates its own change variable.

The DSPAC-2 algorithm is similar to DSPAC-1 algorithm with two exceptions. First, each processor has its own change bit. If processor \( x \) is assigned variable \( i \) and DSPAC-2 running on processor \( x \) removes a value from the domain of \( D_i \), processor \( x \) sends messages to set to “on” only the change bit of all processors \( y \) where \( arc(i,j) \in E \) and variable \( j \) has been assigned to processor \( y \). Therefore, processor \( x \) does not cause processor \( z \) to perform work when the results of processor \( z \) do not depend on the results of processor \( x \).

The second difference is the active_procs variable. This counter, which is maintained by the host, is used to signify the number of processors starting and stopping local arc consistency. When a processor starts (stops) its local arc consistency, it sends a message to increment (decrement) the counter. When the host finds the active_procs variable is zero, it tells the node processors to send the results.

With the DSPAC-2 algorithm we see that changes are “localized,” so only processors which have an arc
connected to a changing variable are informed of the change. This differs from the classic AC-1, Samal's PAC-1, and the DSPAC-1 algorithms. In the DSPAC-2 algorithm, all processors with a variable connected by a constraint to the changed variable rechecks all of its arcs. This selective rechecking still prevents all processors from checking their arcs, even if no changes to any connected variables are made. Processors which have verified local arc consistency, therefore, can immediately start useful processing when they are supplied new variable domain data.

**Theorem 3:** The time complexity of DSPAC-2 is $O(a^2 n c / p)$ and the computational complexity of DSPAC-2 is $O(a^2 n e)$.

**Proof:** The DSPAC-2 complexity analysis is identical to that for DSPAC-1. If DSPAC-2 removes only one value from some domain, that being on the last consistency check of the last active processor, DSPAC-2 performs identically to DSPAC-1.

We should notice a slight improvement in speed over DSPAC-1 in the average case because DSPAC-2 in not synchronized and updates change bits frequently.

---

### 3.3 DSPAC Algorithm #3

All parallel and sequential arc consistency algorithms based on AC-1 operate in the same manner, where all or many arcs are checked if one value in the network has been removed from a variable’s domain. Therefore, even DSPAC-2 still suffers from the inefficiencies of these AC-1 based algorithms. We have developed another parallel arc consistency algorithm to exploit the efficiency of AC-3 and reduce the amount of unnecessary processing. The Distributed Static Parallel Arc Consistency #3 algorithm (DSPAC-3) is listed in Figure 4.

Our algorithm differs from PAC-3 with respect to the queue of arcs to be processed. Samal assumes a shared memory parallel processing model with one queue available for all processors to use. In DSPAC-3 each processor maintains its own queue of arcs that have been assigned to that processor, i.e., only work involving assigned variables and all arcs that emanate from those variables. If Sprevise indicates that a value has been deleted from a variable’s domain, the
Table 1: Comparison of AC, PAC, and DSPAC algorithms.

<table>
<thead>
<tr>
<th></th>
<th>AC-1</th>
<th>AC-3</th>
<th>PAC-1</th>
<th>PAC-3</th>
<th>DSPAC-1</th>
<th>DSPAC-2</th>
<th>DSPAC-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Based on:</td>
<td></td>
<td></td>
<td>AC-1</td>
<td>AC-3</td>
<td>AC-1</td>
<td>AC-1</td>
<td>AC-3</td>
</tr>
<tr>
<td>Processors</td>
<td>1</td>
<td>1</td>
<td>2eα^2</td>
<td>2eα^2</td>
<td>2 to n</td>
<td>2 to n</td>
<td>2 to n</td>
</tr>
<tr>
<td>Computer</td>
<td>Sequential</td>
<td>Sequential</td>
<td>Shared</td>
<td>Shared</td>
<td>Distrib.</td>
<td>Distrib.</td>
<td>Distrib.</td>
</tr>
<tr>
<td>Architectures</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(Shared)</td>
<td>(Shared)</td>
<td>(Shared)</td>
</tr>
<tr>
<td>Time Complexity</td>
<td>O(a^2 ne)</td>
<td>O(a^2 e)</td>
<td>O(nα)</td>
<td>O(nα)</td>
<td>O(a^2 ne/π)</td>
<td>O(a^2 ne/π)</td>
<td>O(a^2 e)</td>
</tr>
<tr>
<td>Computational</td>
<td>O(a^2 ne)</td>
<td>O(a^2 e)</td>
<td>O(a^2 ne)</td>
<td>O(a^2 ne)</td>
<td>O(a^2 ne)</td>
<td>O(a^2 ne)</td>
<td>O(a^2 e)</td>
</tr>
<tr>
<td>Complexity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

queue of arcs of only those processors that utilize
the variable are modified according to the equation
Q ← Q ∪ {arc(m, i)|arc(m, i) ∈ G, m ≠ i, m ≠ j}.

Like DSPAC-1 and DSPAC-2, DSPAC-3 has host
and node components. The host part executes in only
one processor and assigns work to other processors.
The host maintains the active_procs variable, and oper-
ates the same as DSPAC-2. The host component of
the DSPAC-3 algorithm starts the node component of
the algorithm. Each node processor has an interrupt-
driven receive procedure which reads domain change
messages, updates its own memory, and updates its
own queue of arcs (Q[p]).

Theorem 4: The time complexity of DSPAC-2 is
O(a^2 e/p).

Proof: The complexity of a DSPAC-3 is similar to
the complexity of AC-3. Assume that we run the algo-
rithm on a computer with p processors, each proc-
essor is assigned 2e/p arcs (and the variables associated
with those arcs). The queue of arcs on each processor
dynamically grows and shrinks during execution. The
possibility exists that only one value is removed from
each successful call to Sprevise on any of the proc-
sors. Mackworth and Freuden [6] show that the largest
number of arcs added to the queues is a(2e−n). These
arcs are added to the queues of p processors, so the
greatest number of original and added arcs checked
on any of the processors is (2e + a(2e − n))/2p. Spre-
vise makes up to a^2 consistency checks. For all con-
straint networks, e ≥ n − 1, so time complexity be-
comes O(a^2 e/p). □

Theorem 5: The computational complexity of
DSPAC-3 is O(a^2 e).

Proof: If the worst case time complexity occurs,
there is a possibility that all processors are active all
the time. In this case, the complexity of work per-
formed is p times the time complexity, or O(a^2 e). □

These complexities represent the extreme worse
case. On the average (as we show with the experimen-
tial results in Section 4.3), the actual work performed
is substantially smaller.

Table 1 shows a summary and comparison of the
sequential and parallel arc consistency algorithms we
have discussed.

4 Implementation and Measurement

4.1 Execution on the Intel iPSC/2

We have implemented the DSPAC algorithms on
an Intel iPSC/2 distributed memory hypercube com-
puter (a loosely-coupled MIMD computer). The inter-
processor communication is handled via a hypercube
configured network, although Intel specifies that the
network actually works like a fully connected network
due to wormhole routing techniques [4]. Each pro-
cessing node contains an Intel 80386 processor and an
ample amount of memory.

The iPSC/2 operating system allows a host com-
puter to control a set of processing nodes. Our host
distributes and loads the work into the processing
nodes, instructs the processing nodes to start, moni-
tors the work performed, and gathers the node results
and desired statistics. Each processor is assigned a
number of constraint network variables and the asso-
ciated constraint data.

We have built a simulator [2] to examine the impact
of communications frequency with the amount of work
performed by our algorithms. A value-based simula-
tion sends a message every time a value is removed
from a domain. An arc-based simulation sends a mes-
sage if at least one value is removed after one entire arc
has been checked. A variable-based simulation sends a
message if at least one value is removed after all the arc
originating from a single variable have been checked.
A processor-based simulation sends a message after all constraint variables assigned to a processor have been checked. A message takes time equivalent to one consistency check to reach its destination or to receive the data. Although we simulated four message frequencies, we only programmed the arc-based message paradigm, which usually represented the best speedup possible [2]. This is also indicated in the algorithms we presented in Section 3.

One global variable, the no_solution variable, signifies when processing should continue. This is implemented by a broadcast message. If a processor finds that a variable is empty (and the constraint network has no solution), it sends a message to all processors informing them to stop work. Each node processor has an interrupt-driven receive procedure which acknowledges this message. Another interrupt-driven receive procedure indicates a processing-completion message from the host, whereafter nodes then send their results to the host.

## 4.2 Network Data

We created many sample constraint networks for experimental trials. While gathering the experimental data, ten random constraint networks were run for each of the configurations. The networks are computer-generated and have some attributes determined by random processes as follows:

- **Size:** problems contain 20, 40, and 60 variables.
- **Domains:** each variable has a domain of 10 values.
- **Solutions:** Each problem contains one specific solution. That is, if the constraint network were solved using a tree search algorithm, only one consistent assignment could be made.
- **Connectivity:** The connectivity (degree) of each variable is a random number, with a minimum of 2, a maximum of 10, and an average of 6.
- **Satisfiability:** The relative satisfiability is the success rate of a consistency check. More specifically, it is the ratio of relation value pairs to the total possible pairs (domain \( \times \) domain). The satisfiability of our networks are \((15 \text{ relation pairs})/(10 \text{ domain values} \times 10 \text{ domain values}) = 0.15\).

## 4.3 Metrics and Results

The results of our measurements suggest that our parallel algorithms provide good speedup. We compute DSPAC-1, DSPAC-2, and DSPAC-3 speedup based on the AC-3 sequential algorithm. Actual speedup measurements are shown in Figure 5.

Execution of DSPAC-1 on a four-processor computer configuration yields speedup of between 64 and 74% of linear speedup, DSPAC-2 yields a range of 60 to 76%, and DSPAC-3 yields a range of 71 to 84%. Execution of DSPAC-1 on a ten-processor computer configuration yields speedup of between 44 and 51% of linear speedup, DSPAC-2 yields a range of 45 to 63%, and DSPAC-3 yields a range of 46 to 68%. Execution of DSPAC-1 on a twenty-processor computer configuration yields speedup of between 31 and 48% of linear speedup, DSPAC-2 yields a range of 33 to 52%, and DSPAC-3 yields a range of 35 to 56%.

## 5 Scaling of CSPs using Parallelism

Since arc consistency algorithms can greatly reduce the search space of a CSP, they can be used as an effective pre-processor for many AI applications which are viewed as CSPs. CSPs include resource scheduling, detection of graph and subgraph isomorphism, the graph coloring problem, semantic information pro-
cessing, theorem proving, image processing and object recognition, VLSI design, neural network computing, and database consistency checking.

The constraint networks examined here represent small problems with a single solution. These networks can be scaled to larger problems by processing more constraint variables on each processor. From the measurements one can see that our algorithms perform best when each processor is assigned many constraint variables. Our best speedup is measured when 60 constraint variables are assigned to four processors. Also, due to the distributed nature of the algorithms, problems can also be scaled to a greater number of processors.

As an example of scaling of a problem, we created several large constraint networks and compare results of the DSPAC-2 algorithm on five different multiprocessor configurations. In particular, our network has 120 variables, the domain size is 16 values, the connectivity of each variable is from 2 to 30, and the satisfiability is 15%. When executed on an Intel iPSC/2 with 4, 10, 20, 32, and 64 processors, we still realize an appreciable speedup (although speedup and utilization decreases with configuration size). Speedup and utilization of these large networks as compared to the 60 variable networks is shown in Figure 6.

6 Conclusions

We have reviewed several arc consistency algorithms for sequential and parallel processing computers. We introduced three distributed parallel arc consistency algorithms: DSPAC-1, DSPAC-2, and DSPAC-3, and compared our algorithms with existing algorithms. We ran the algorithms in a parallel processing environment and compared their performances with that of Mackworth's AC-3 algorithm.

We have learned that our algorithms can be effectively used to preprocess a constraint network, but we must ensure that each processor has a large amount of data to process. We also learned that the SPAC-3 algorithm is the best for our distributed environment. Although SPAC-3 messages are four bytes longer and receiving these messages require more processing, we save time because we recheck fewer arcs.

References


