# The Design and Implementation of a Parallel Solution to the Cutting Stock Problem*

Lisa D. Nicklas Robert W. Atkins Sanjeev K. Setia Pearl Y. Wang
Department of Computer Science MS 4A5
George Mason University
Fairfax, VA 22030-4444

**Abstract**

The constrained 2-D cutting stock problem is an irregular problem with dynamic data structures, highly variable amounts of computation per task, and unpredictable amounts and patterns of communication. This paper describes the design and implementation of a parallel solution to this problem on a cluster of workstations and a distributed memory multicomputer. The key element of our parallel solution is the replication of an important data structure on all processors. By exploiting properties of the cutting stock problem which allow the use of relaxed consistency mechanisms, our approach is able to reduce the overheads for communication and synchronization in comparison to approaches that partition this data structure among processors. A token-based lazy release consistency protocol is used to ensure mutual exclusion and maintain consistency, and a randomized work-stealing protocol is employed to dynamically balance work among processors. Good speedups are reported for three benchmark problems executed on two distributed memory platforms: a cluster of workstations interconnected by a 10 Mbps Ethernet and an Intel Paragon.

**keywords:** two-dimensional cutting stock, irregular problems, workstation clusters, dynamic load balancing, relaxed consistency, distributed memory multicomputer

## 1 Introduction

Cutting and packing problems are well-known resource allocation problems in Operations Research [5, 9, 10, 12]. They are concerned with finding the optimal allocation of stock materials so that customer orders for given products are satisfied in a cost-efficient manner. One long-standing problem which has numerous applications in the paper, glass, and lumber industries is known as the two-dimensional (2-D) cutting stock problem. Here, cutting patterns which can be used to produce a given set of rectangles from stock sheets with minimum waste are to be determined. These patterns are then used to fill customer orders for paper, glass, or wood materials, subject to manufacturing and storage costs.

Cutting stock problems are usually difficult to solve optimally, and heuristic methods are commonly utilized. Even so, for the large problem instances that arise in practice, serial solution algorithms can take hours

*This work was partially supported by NSF grant CCR-9409697.

to execute, making it important to develop parallel solutions. This paper describes a parallel approach for solving 2-D cutting stock problems on distributed-memory parallel systems.

The cutting stock problem falls into a class of problems that have a common property: finding their solution requires taking an input data set and repeatedly performing calculations involving all pairs of items in the set. As these computations proceed, new data items are added to the set or old ones are removed. Thus, the size of the data set can increase or decrease at unknown rate. Another characteristic of the problem is that the time to perform the calculations for different pairs of data items is highly variable. Because of these characteristics, the cutting stock problem can be considered to be irregular in nature. Another well-known application that falls into this class of irregular problems is Gröbner bases calculation, an algebraic technique for solving nonlinear systems of equations [1].

Parallelizing classical algorithms that solve a constrained 2-D cutting stock problem poses several challenges arising from the irregular nature of the problem [16]. First, the amount of computation in the steps of the algorithm is highly variable, leading to the need for dynamic load balancing in a parallel solution. Second, the main data structure used by the algorithm is dynamic in nature, i.e., its size increases and decreases during execution in an unpredictable fashion. Third, as mentioned above, one of the steps in the algorithm potentially involves the entire data set, which results in data dependencies between the nodes storing the data set in a parallel solution. These characteristics make it difficult to obtain an efficient parallel solution to the cutting stock problem, especially on distributed memory parallel systems such as workstation clusters. However, as in the case of many combinatorial optimization problems, the problem has a significant amount of parallelism which can be exploited by a parallel solution.

In this paper, we describe a new approach for parallelizing Wang's algorithm [23], a classical algorithm used for solving the cutting stock problem. Under our approach, an important data structure used by the algorithm is replicated on all processors. In contrast, the method used in previous approaches to this problem [20, 14] partitioned this data structure among the processors. The key insight behind our approach is that the properties of the cutting stock problem make it possible to use relaxed consistency mechanisms in a parallel solution. The reduction in communication and synchronization overheads resulting from an approach based on replication makes it possible to obtain an efficient parallel solution, even on a cluster of workstations interconnected by a low-bandwidth LAN.

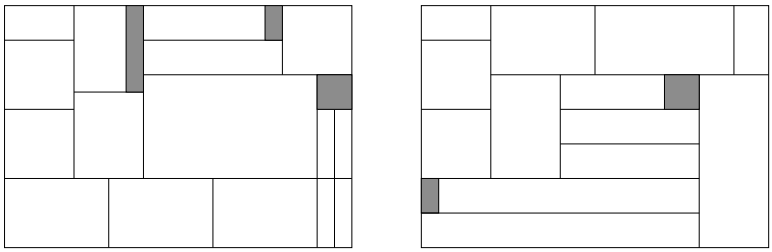
Our parallel implementation uses a token-based lazy release consistency protocol that is tailored to minimize communication and synchronization costs by exploiting the properties of the cutting stock problem. In addition, our implementation uses a randomized work stealing protocol for dynamic load balancing, and overlaps all communication with computation. We obtained good speedups for three benchmark problems on a workstation cluster with a 10 Mb/s Ethernet, and on the Intel Paragon.

The remainder of this paper is organized as follows. The constrained cutting stock problem and a classical solution algorithm are reviewed in Section 2. Section 3 discusses how we addressed the parallel design issues that must be considered in order to develop an efficient parallelization of the algorithm based on replication. In Section 4, we present performance results for three benchmark problems, and analyze the factors that affect the performance of our implementation. Finally, Section 5 presents conclusions.

## 2 Problem Statement

The constrained 2-D cutting stock problem is to find the optimal allocation of a set of $n$ input rectangles to a stock sheet of size $L \times W$. Each input rectangle $r_i$ has an associated length ($l_i$), width ($w_i$), value ($v_i$), and

constraint ($b_i$). The constraint, $b_i$, is the maximum number of times that rectangle $r_i$ may appear in the stock sheet.



(a) A Guillotine Pattern (b) A Non-guillotine Pattern

Figure 1: Cutting Patterns

In the paper and lumber industries, rectangles must be cut from stock sheets using *guillotine* or edge-to-edge cuts made parallel to the edges of the stock sheet. For example, Figure 1 shows two cutting patterns; the first uses guillotine cuts and the second does not. (The shaded areas of the patterns represent trim waste.) Thus, the goal of the constrained 2-D problem is to determine a guillotine cutting pattern which has maximum total value:

$$\begin{aligned} \text{Maximize}_G \quad & \sum_{i=1}^{n} v_i x_i \\ \text{subject to} \quad & 0 \leq x_i \leq b_i \\ & x_i \text{ integer and G a guillotine pattern} \end{aligned}$$

where $x_i$ is the number of times that $r_i$ is cut in $G$.

The value of a pattern is the sum of the rectangle values within the pattern. Hence, an optimal pattern has the largest total value of all possible cutting patterns. If the value of an input rectangle is its area, then the objective function corresponds to finding the guillotine pattern with minimum trim waste. Alternatively, the value of an input rectangle is often defined as a quantity that reflects demand or costs associated with cutting the rectangle from the stock piece.

## 2.1 Solution Strategies

While dynamic programming can solve the unconstrained 2-D cutting stock problem [11], the constrained 2-D problems have used two classic approaches [8, 23]. Christofides and Whitlock's approach uses a tree-based search to find an optimal solution, while Wang's approximation algorithm is combinatoric in nature. Recently, several modifications have been made to the latter in [15, 21, 22]. These introduce lower and upper bound estimates (i.e. pruning), and thus, in effect, extend the combinatoric method to a branch-and-bound algorithm.

By nature, tree-based searching methods that employ pruning terminate after finding an optimal solution, e.g., a guillotine pattern having maximal total value. In industrial settings, it is desirable to obtain multiple solutions, i.e., alternative cutting patterns that have the identical maximal total value or have values slightly less than optimal. Since the optimal cutting pattern could possibly exclude one or more input rectangles, it

is advantageous to generate several near optimal cutting patterns that will include a larger subset of the original rectangles. These "good" patterns are frequently used by another optimization algorithm to fill customer orders for large quantities of specific pieces [12, 18, 23]. Since we seek to develop an algorithm that generates a set of acceptable cutting patterns to be used in a larger practical context, we parallelized the original combinatoric version of Wang's algorithm.

## 2.2 Wang's Algorithm

Guillotine cutting patterns are successively generated by combining pairs of partial patterns into larger ones as shown in the algorithm in Figure 2. A work queue $WQ$ holds the pairs of patterns to be combined. A procedure $ADD(f, g)$ takes two input patterns $f$ and $g$ and constructs two new patterns $p_1$ and $p_2$ by combining the input patterns horizontally and vertically as shown in Figure 3. These newly constructed patterns are then tested for size, amount of waste, and the number of times any input piece appears in the new patterns. If they pass these tests in $ADD$, the new patterns are returned to the calling algorithm.

---

**Input:** A set of rectangles $F = \{r_1, r_2, \ldots, r_n\}$
Tolerance $\beta$
Set of constraints C
**Algorithm:**
$B \leftarrow F$
Work queue $WQ \leftarrow \{(f, g) |\ f, g \in B\}$
**while** ($WQ$ not empty) **do**
remove pair $(f, g)$ from $WQ$
$p_1, p_2 \leftarrow ADD(f, g)$ //creates at most 2 new patterns if waste $\leq \beta LW$ and constraints are met
**for** each $p_i \neq \emptyset$
$TEST(p_i, B)$ //$p_i$ compared against all elements of $B$ for duplication and dominance
**endfor**
**for** each $p_i$ that passes TEST
$UPDATE : B \leftarrow B \cup \{p_i\} - \{s \mid s \in B, s$ is dominated by $p_i\}$
$UPDATE_WQ : \forall\, t \in B :\ WQ \leftarrow WQ \cup \{(p_i, t)\} - \{(s, t) \mid s \in B, p_i$ dominates $s\}$
**endfor**
**endwhile**

Figure 2: Wang's Algorithm

---

Additionally, an input parameter $\beta$ , $0 \leq \beta \leq 1$, identifies those generated patterns having acceptable values. Adopting the convention that the value of an input rectangle is equal to its area, we retain only those constructed patterns with total waste less than $\beta \cdot LW$. (Using this rule, we may predict the difference in waste between the heuristic's best pattern and the optimal solution pattern [23].) Each newly constructed pattern $p$ is then examined against all current patterns, $B$, in the $TEST(p, B)$ procedure to detect occurrences of *duplication* and *dominance*. When a new pattern is compared with an existing pattern, the algorithm checks if:

- the new pattern is identical to the existing pattern,

- the existing pattern is dominated by the new pattern,
- or the new pattern is dominated by the existing pattern.

Patterns are considered identical if they have the same dimensions and contain (or cut) the same rectangles. If $p$ and $q$ represent two patterns, then $p$ is dominated by $q$ if (1) $p$ has length and width greater than $q$, and $p$ cuts the same rectangles as $q$ in identical amounts, or (2) $p$ and $q$ are the same size, and $p$ cuts a subset of the same rectangles cut by $q$. Either of these two cases imply that $p$ cuts the same input rectangles as $q$, but less efficiently, i.e. with more trim waste.

The motivation for using these duplication and dominance tests, and the controlling $\beta$ parameter is two-fold. They reduce the potentially exponential number of patterns generated by the algorithm, and the $\beta$ parameter permits the algorithm to keep generated patterns that are not necessarily optimal, but still meet an acceptable waste criterion.

The $UPDATE$ procedure processes the new pattern subject to the outcome of these duplication and dominance tests. If the new pattern is a duplicate or is dominated by any existing pattern, it is discarded. Otherwise, it is added to $B$, and any current patterns that are dominated by the new pattern are removed from $B$. Further, the newly accepted patterns augment the work queue in the $UPDATE_WQ$ procedure. The new work corresponds to horizontal and vertical builds of the new pattern with existing patterns.

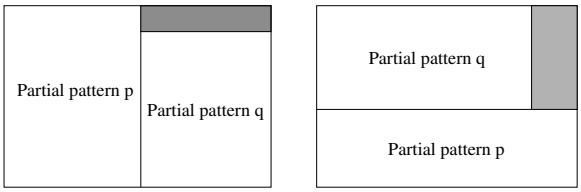
# 3 Parallel Approach

## 3.1 Design Considerations

An examination of the serial algorithm in Figure 2 reveals the potential for exploiting parallelism when solving the cutting stock problem. Execution time is dominated by the combining of two patterns to create a new pattern and the testing of a new pattern for dominance and duplication. By distributing the work queue $WQ$, thereby dividing the computational work among several processors, we can reduce the computation time.

One of the challenges in efficiently parallelizing the cutting stock problem is that the amount of computational work done in the $ADD$ and $TEST$ steps is highly variable and unpredictable. The amount of new work generated in step $UPDATE_WQ$ is also unpredictable. This necessitates that any approach to parallelization must incorporate mechanisms for *dynamic* load balancing. In our parallel approach, the work queue $WQ$ is partitioned among the processors involved in the computation and a load-balancing protocol is employed for keeping the work queues balanced.

Another challenge in parallelizing the serial algorithm is in the $TEST$ step. We must compare the newly generated pattern against all previously generated patterns stored in the set $B$. Two alternative approaches can




(a) A Horizontal Combination (b) A Vertical Combination

Figure 3: Possible Combinations

be taken to parallelize this step: partitioning or replicating $B$.

Under the first approach, the set $B$ is partitioned among the processors involved in the computation. This implies that in the $TEST$ step, the processors must circulate the pattern $p$ and test $p$ for dominance and duplication against the partial list of patterns (say $B_i$) stored at each processor $i$. In addition, since the list $B_i$ will grow and shrink dynamically, all the $B_i s$ must be kept balanced during the computation.

In the replicated approach, each processor has a copy of the set $B$. As a result, the $TEST$ step could be completed locally at a node without any communication. However, replicating $B$ entails communication both for concurrency control (to prevent simultaneous updates to $B$ on different nodes) and for ensuring up-to-date replicas.

This tradeoff between replication and partitioning arises frequently in parallel computation [3, 7, 17]. For the 2-D constrained cutting stock problem, the amount of communication when using replication depends on the number of updates to $B$ while the amount of communication using partitioning depends on the number of comparisons of new patterns being tested to existing patterns in the set $B$. As we show in Section 4, the ratio of the number of such comparisons to the number of updates to $B$ is quite large for Wang's algorithm.

A potential problem with replicating a data structure on several processors is the high overhead for maintaining consistency when the application's consistency requirements are strict (e.g., sequential consistency is required). Fortunately, as we show in Section 3.2, the properties of Wang's algorithm permit the use of relaxed consistency mechanisms. Specifically, each processor can do useful work in the $TEST$ step even without the latest copy of $B$. We only need to ensure that a processor has read the latest copy of $B$ before it updates $B$ and that different processors are prevented from making simultaneous updates of $B$.

There is another, more subtle, advantage of replication over partitioning for the cutting stock problem. With the sequential solution algorithm, the execution time for the duplication and dominance checks is greatly affected by the order in which $B$ was traversed. When $B$ is traversed in last-in first-out (LIFO) order, duplication and dominance checks are most efficient, i.e., fewer patterns in $B$ need to be checked. This is because the algorithm has a greater probability of making a positive comparison for duplication and dominance with a pattern in $B$ that has been recently generated, which is apt to have a similar length, width, and composition of input rectangles.

With a replication scheme, each node can simply traverse its local copy of $B$ in LIFO order for duplication and dominance checks. When $B$ is partitioned among nodes, however, there is no global order maintained for the elements in the partitions of $B$. As such, under this approach, new patterns are not compared against elements of $B$ in LIFO order leading to potentially higher execution times.

A potential disadvantage of replicating a data structure on all nodes is a limitation on the scalability of the approach, since the memory requirements of large problems may exceed the available memory on each node. However, as Section 4 discusses, this did not pose a problem for the (relatively) large benchmarks considered in this paper.

Because of the reasons outlined above, we chose to replicate the set $B$ in our parallel implementation. In contrast, to the best of our knowledge, previous parallel solutions to the cutting stock problem have been based on partitioning the set $B$ [20, 14]. We note that a replication-based approach has also been used successfully in parallelizing Gröbner bases calculation, a problem with structural similarities to the cutting stock problem [16], on a distributed-memory parallel computer [7].

### 3.2 Algorithm and Data Structure Design

Figure 4 shows a parallel version of Wang's algorithm. The main data structures used by our algorithm are a distributed work queue and the replicated set $B$. The main sources of communication overhead are the protocols for balancing the work queues and maintaining $B$'s consistency. We discuss the design of the algorithm and its data structures below.

---

**Input:** A set of rectangles $F = \{r_1, r_2, \ldots, r_n\}$
Tolerance $\beta$
Set of constraints C
**Algorithm:**

```
B ← F
Global Work Queue WQ ← {(f, g) | f, g ∈ B}
Local Work Queue LWQ ← GETWORK(WQ)
while ( WQ not empty) do
    if (LWQ not empty)
        remove pair (f, g) from LWQ
    else
        LWQ ← GETWORK(WQ)
    endif
    p1, p2 ← ADD(f, g)          //creates at most 2 new patterns if waste ≤ βLW and
                                  constraints are met
    for each p_i ≠ ∅
        LOCAL_TEST(p_i, B)      //p_i compared against local copy of B for
                                  duplication and dominance
        U ← U ∪ {p_i | p_i passes LOCAL_TEST}
    endfor
    GET_LOCK()
    UPDATE_LOCAL(B)             //Bring the local copy of B up to date
    for each p_i ∈ U
        FINISH_TEST(p_i, B)     //p_i compared against new elements added
                                  to B in UPDATE_LOCAL
        UPDATE: B ← B ∪ {p_i} − {s | s ∈ B, s is dominated by p_i}
    endfor
    RELEASE_LOCK()
    for each p_i ∈ U that passes FINISH_TEST
        UPDATE_WQ: ∀ t ∈ B : LWQ ← LWQ ∪ {(p_i, t)} − {(s, t) | s ∈ B, p_i dominates s}
    endfor
endwhile
```

Figure 4: A parallel version of Wang's algorithm

---

**Distributed Work Queue** Each processor obtains work from its local work queue $LWQ$. When a processor runs out of work, it attempts to obtain work from the queue of another processor using randomized work stealing [6]. Since one of the target environments for parallelization is a cluster of possibly heterogeneous

workstations, the load balancing algorithm takes the relative speeds of the processors into account when deciding the amount of work to give another processor.

**Exploiting Weak Consistency Requirements** Since $B$ is logically shared by all the processors, we must use a concurrency control mechanism to prevent simultaneous reads and updates of the replicas of $B$ on different processors. We must also use a protocol that ensures consistent copies of $B$.

A straightforward but inefficient way of enforcing these conditions would be to require each processor to acquire a global lock before the $TEST$ operation, releasing the lock after the $UPDATE$ step. To maintain consistency, if a processor updates its local copy of $B$ while holding this lock, this update has to be propagated to all other copies of $B$. Alternatively, all other copies of $B$ need to be invalidated and other processors would have to acquire the latest copy of $B$ before their next read or update of $B$.

While this critical section style of shared data access would ensure the correctness of the parallel algorithm, it would also severely restrict the possible concurrency and result in poor performance. To obtain good speedups, we must exploit the properties of the problem that allow relaxed consistency mechanisms.

Consider the $TEST$ and $UPDATE$ phases of the algorithm. In the $TEST$ step, each element of $B$ is compared for dominance and duplication against the new pattern $p$. If the pattern $p$ passes these tests, then it is added to $B$ in the $UPDATE$ step. Also, any pre-existing patterns in $B$ dominated by $p$ are deleted. Thus, there are three possibilities:(i) $p$ fails the test in which case $B$ is not modified, (ii) $p$ passes the test and is added to $B$, or (iii) $p$ passes the test and is added to $B$ causing some existing pattern in $B$ (say $s$) to be removed.

In case (i), when $TEST$ fails, there is no conflict between $TEST$ on one processor and $UPDATE$ on another processor. This is because the pattern $p$ that is rejected by the processor executing $TEST$ would be rejected regardless of any elements that may be added to or deleted from $B$ during $UPDATE$ on another processor. In cases (ii) and (iii), though, it is possible that the pattern $p$ being tested may pass *locally* but would be rejected if compared against the new elements being added at another processor during $UPDATE$. Most importantly, the work being done locally by the processor during $TEST$ with a stale copy of $B$ is *not useless*[1]. To complete $TEST$, we must compare $p$ with any new elements added to $B$ by other processors.

This observation shows the possibility of increasing the concurrency of the algorithm by splitting $TEST$ into two steps as shown in Figure 4. In the $LOCAL_TEST$ step, $p$ is compared against all the elements in the local (possibly stale) copy of $B$ on a processor. If $p$ passes the local test, it is placed in a local update set $U$. Before $p$ can be moved from $U$ to $B$, however, the processor has to obtain any missing elements of $B$ and compare $p$ to these missing elements in the $FINISH_TEST$ step shown in Figure 4. To prevent another processor from adding new elements to $B$ when executing $FINISH_TEST$, we use a global lock.

An important consideration in the implementation is the method used to propagate the elements added to $B$ in the $UPDATE$ step to the other processors. The best method to use in an implementation depends upon many factors, the most important being the cost of communication on the underlying system platform. For the correctness of the algorithm, we must only ensure that a processor has a current copy of $B$ before it enters $FINISH_TEST$.

[1]There is some potential for doing extra work in all cases when an existing pattern is deleted by another processor. In this case, the comparison between the pattern under consideration and the deleted pattern is extra work that would not be done if strict sequential consistency is maintained.

### 3.3 Algorithm and Data Structure Implementation

**Task Scheduling** In the parallel algorithm above, a unit of work, i.e., a logical "task" in the work queue, consists of combining a pair of patterns from the set $B$ to produce a new pattern (the $ADD$ step) and the testing of the produced pattern for dominance and duplication (the $LOCAL_TEST$, $FINISH_TEST$, and $UPDATE$ steps). The actual amount of computation in these steps is quite small relative to the cost of acquiring a lock. To obtain good performance it is necessary to amortize the cost of obtaining a lock by testing several patterns each time a processor acquires a lock.

While there are tasks in its work queue, each processor performs the $ADD$ and $LOCAL_TEST$ steps for these tasks. If the new pattern created by a task passes the local test, it is placed in a local list $U$. Once a lock is obtained, several patterns in $U$ are tested for dominance and duplication, and the accepted patterns are added to $B$ before releasing the lock. Since a processor never blocks for obtaining a lock, this allows us to overlap computation and communication.

**Representation of Work** In our implementation, the set $B$ is organized as a linear array of patterns[2]. A simple way to represent a unit of work would be to store the patterns that have to be combined as a tuple $(x, y)$, where $x$ and $y$ are the indices of the two patterns in the array $B$. However, since the number of pairs can grow exponentially during execution, this results in a large space overhead. A unit of work can be compactly represented by mapping each tuple $(x, y)$ from the two-dimensional space to an integer in one-dimensional space. Multiple units of work (or tasks) can now be represented by a subrange of the linear work space. The work queue thus becomes a list of subranges.

As discussed in Section 3.1, the work queue $WQ$ in the serial algorithm is partitioned among the nodes involved in the computation. Each node's local work queue initially has one subrange of work. As work migrates between processors, each processor may have several subranges in its local work queue. When a processor adds a new pattern to $B$, it also adds a new subrange to the end of its local work queue.

Deleting a pattern from $B$ is more problematic. Deleting a pattern should result in the removal of a subrange from a queue. However, the work involving the deleted pattern may already have migrated to other nodes. Keeping track of each subrange of work in a distributed work queue that is dynamically load balanced is difficult and unnecessary. Instead of trying to delete work from the distributed work queue, we simply invalidate deleted patterns in the set $B$. When a processor attempts a task $(x, y)$, it checks $B$ to make sure that both patterns $x$ and $y$ are valid.

**Mutual Exclusion and Replica Consistency Protocol** The primary consideration in implementing the locking and consistency protocols on a distributed memory system is to keep the communication overhead as low as possible. To achieve this, it is necessary to minimize the number of messages sent by a processor by combining messages wherever possible and using weak consistency mechanisms [2].

As discussed in Section 3.1, the properties of the cutting stock problem permit the use of relaxed consistency mechanisms. Specifically, the properties of the problem allow a processor to update its copy of $B$ just before it enters the $FINISH_TEST$ step. Since a processor has to acquire a lock before it enters $FINISH_TEST$, one way to reduce the number of messages is to combine the message granting a lock to

[2] Adding a pattern corresponds to extending this array. A pattern is deleted by invalidating it in the array; however, the pattern is not removed from the array.

a processor with the updates that have to be propagated to the processor. This is similar to the lazy release consistency mechanism used in implementing distributed shared memory on a cluster of workstations [13].

We implemented a lazy release consistency protocol that was tailored to minimize the number of messages for the cutting stock problem. It was observed that the updates of the set $B$ occur consistently across nodes throughout the execution of the program. Moreover, in a parallel implementation, these updates are distributed evenly among the processors involved in the computation.

Based on these observations, we designed a token-based lazy release consistency protocol that combined the granting of a lock and propagation of updates to a processor. Under this scheme, the nodes involved in the computation are organized in a logical ring. A logical token circulates among the nodes. Possessing the token is equivalent to possessing the lock needed to enter $FINISH_TEST$. Along with the token, a node also receives any updates to $B$ made by the all the other nodes involved in the computation the last time they had the token, i.e., in the current round.

When it receives the token, a node $i$ processes a number of patterns from $U$ (i.e., patterns that have passed local tests for dominance). In other words, it compares these patterns against any new patterns received along with the token. Any patterns that pass the test are then added to the local copy of $B$. In addition, any patterns that are dominated by a new pattern are deleted from the local copy of $B$. These updates to $B$ are then appended to the message containing the token. The token is then transmitted to the next node in the logical ring. When that node $i + 1$ receives the token, it carries out the same sequence of steps. Thus, when it transmits the token to its neighbor node $i + 2$, along with the token, node $i + 2$ will receive updates made by both $i$ and $i + 1$. When the token returns to node $i$, it removes the patterns it had added to the token during its last visit.

By merging the updates made by several nodes and a lock grant notification into a single message, this protocol minimizes the number of messages required for enforcing concurrency and consistency constraints. This protocol differs from implementations of lazy release consistency for distributed shared memory systems in that there is no message required for acquiring a lock. This eager method of propagating a lock to all processors in turn performs well if work is well balanced among the processors, since each processor will generate patterns that get added to the set $B$. However, there can be a relatively long delay between successive visits of the token. The key to the success of this protocol is that nodes can do useful work in the $ADD$ and $LOCAL_TEST$ steps, even if they do not have the token.

**Dynamic Load Balancing** A randomized work stealing approach [6] was used to perform dynamic load balancing. When a processor has an empty work queue, it attempts to steal work from another processor picked at random. If it receives a negative response to its work stealing request, it picks another processor and attempts to steal work from it.

A processor that receives a request for work gives a fraction of its work to the requesting processor provided it has enough work in its work queue. Along with a work steal request, a processor sends information about the rate at which it has been processing patterns. This rate is used to estimate the relative speeds of the processor stealing work and the processor giving work. The fraction of a processor's work that is given to the requesting processor is determined based on this estimate.

A potential problem that can occur during work stealing is that a processor may try to steal work that involves patterns it has not yet received. It is possible that the node giving work has generated work in the $UPDATE_WQ$ step that involves patterns which the stealing node does not yet know about. In this situation, the stealing node will not be able to process the work it has stolen.

In order to avoid this situation, the node trying to obtain work sends the current number of patterns in its local copy of $B$ along with its steal request. If the work being sent involves additional patterns which the requesting node does not have, these patterns are forwarded along with the work [3].

**Termination Detection** When a processor is unsuccessful in stealing work from any other processor, it sets a flag in the token indicating that it is finished with the computation. When all workers have set their flags, the computation is considered terminated. This protocol is optimized so that a processor looking for work does not try to steal work from other processors that have already set their completion flags.

**Program Structure** The program is written in the SPMD style with a worker process on each node. One of the worker processes doubles as a manager process, creating the processes on the other nodes, reading the input data, initializing the local work queues, and printing out the final results.

Communication between the worker processes for the token-based consistency protocol and the load balancing protocol is asynchronous. In our implementation on a cluster of workstations, we create a second process on each node for handling communication. This message handling process is always blocked waiting for messages. When a message arrives, the message handling process notifies the worker process of its arrival by setting a flag in shared memory.

In our Intel Paragon implementation, we eliminated the separate message handling process. Instead, the `hrecv` function in the NX message library was used. The `hrecv` function posts an asynchronous receive for a message and then returns control to the worker process. When the message is actually received, a user-supplied interrupt handler is executed. For our purposes, the interrupt handler updates a global flag. Both of these techniques eliminate expensive message probes replacing them with less expensive checks on a global variable.

# 4 Performance Evaluation

## 4.1 Platforms

The parallel algorithm implementation was evaluated on two distributed memory platforms – a cluster of workstations and a dedicated multiprocessor. The cluster environment consisted of Sun Sparc IPX, Sparc LX, Sparc 5, Sparc 20, and Ultra workstations interconnected by a 10 Mbps Ethernet. The cluster implementation used the message passing library, Parallel Virtual Machine (PVM) [19] Version 3.3.11, and the program was written in C using `gcc-2.7.1` with optimization level `O3`. The timing measurements were taken when primarily when no other user jobs were running on the workstations and network traffic was at a minimum. The dedicated multiprocessor environment was an Intel Paragon with GP compute nodes. The Paragon implementation used the Intel NX message-passing library and was also written in C using the `icc` compiler with optimization level `O4`.

[3] Note that deleting a pattern from the set $B$ involves invalidating it in the array storing $B$. This array never shrinks in size; thus a count of the current number of patterns (both valid and invalid) in the array is all that needs to be sent along with the work request for the node giving work to determine if it needs to send any patterns along with work.

## 4.2 Benchmarks

The parallel implementation was tested using three input data sets. `cgcut2` and `cgcut3` which were obtained from the OR-library of benchmark cutting stock problems [4]. The third test data set, `concat`, was a concatenation of the first two. (See Appendix for these instances.) The serial (and parallel) execution times of Wang's algorithm can vary depending on how the input rectangles are ordered because of its nondeterministic nature. For example, ordering the input rectangles by increasing area encourages the early creation of the basic building blocks used by the final patterns. Hence, in the results reported here, the input rectangles were pre-processed into ascending area order as this generally yields the faster execution times.

| Problem | Comparisons | Updates |
|---|---|---|
| cgcut2 | 117,038,030 | 12,599 |
| cgcut3 | 107,135,670 | 14,580 |
| concat | 905,158,455 | 37,749 |

Table 1: The total number of comparisons (in the $TEST$ step) and the total number of updates to $B$ (in the $UPDATE$ step) for the three benchmarks in a sequential run.

As discussed in Section 3.1, the decision to replicate versus partition the pattern list $B$ is guided by the frequency and amount of communication used by the algorithm for the input data sets. A partitioned approach necessitates communication in order to compare a new pattern that has passed local constraints against the existing patterns in $B$ for duplication and domination. The number of these comparisons is an indication of the communication costs incurred when using the partitioned method. When $B$ is replicated, the amount of communication is dependent on the number of updates, additions to, and invalidations performed on $B$ which must be communicated to all nodes to ensure $B$'s consistency. The cutting stock algorithm has many more comparisons to existing patterns in $B$ than updates of patterns in $B$. Table 1 shows the comparisons and updates statistics for the three benchmarks. This data supports the decision to replicate instead of partition the data structure $B$.

## 4.3 Metrics

The implementation's total execution times and speedups were used to study the performance of our implementation. Speedups are evaluated against the execution time of the parallel program running on one processor ($P = 1$) as shown in Table 2. This baseline was used since it is difficult to find the best serial algorithm for solving the constrained cutting stock problem. For example, the execution times for the serial version of the Wang's Algorithm will differ depending on the heuristics used for selecting the pairs of rectangles to be combined next.

The use of this speedup measure also provides a good indication of the efficacy of our parallelization – the measurements indicate that the total work performed in the parallel implementation was roughly equivalent to the work done by the serial implementation. Table 3 shows the ratio of the number of pattern pairs attempted by the parallel implementation to the number attempted by the serial implementation. Clearly, there is not much variation in the total amount of work done by the parallel and serial implementations.

| Problem | WORKSTATIONS | | | | | PARAGON |
|---|---|---|---|---|---|---|
| | Sparc IPX | Sparc LX | Sparc 5 | Sparc 20 | Ultra | compute node |
| cgcut2 | 1013 secs | 789 secs | 314 secs | 300 secs | 145 secs | 827 secs |
| cgcut3 | 921 secs | 721 secs | 286 secs | 275 secs | 135 secs | 786 secs |
| concat | 9942 secs | 7733 secs | 3077 secs | 2932 secs | 1568 secs | 7121 secs |

Table 2: Execution times for $P = 1$ on various platforms.

| $P$ | cgcut2 | cgcut3 | concat |
|---|---|---|---|
| 1 | 98.80 | 99.49 | 100.01 |
| 2 | 99.10 | 97.54 | 99.98 |
| 4 | 101.00 | 97.41 | 99.97 |
| 6 | 100.63 | 99.22 | 99.98 |
| 8 | 100.89 | 102.01 | 99.98 |
| 10 | 101.22 | 101.19 | 99.99 |
| 12 | 101.57 | 103.43 | 99.99 |
| 14 | 102.68 | 105.99 | 99.98 |
| 16 | 102.97 | 108.56 | 100.00 |
| 20 | 101.40 | 107.44 | 100.00 |
| 24 | 102.02 | 107.54 | 99.99 |
| 28 | 101.68 | 104.61 | 99.99 |
| 30 | 100.78 | 106.93 | 100.01 |

Table 3: Percentage of serial work performed by parallel version on a cluster of workstations

### 4.4 Performance Results

**Execution Times and Speedups** The measured execution time for the three benchmark problems on a cluster of workstations and an Intel Paragon are plotted in Figure 5. In the workstation cluster, the first 14 workstations are Sparc LXs and the remainder are a combination of Sparc IPXs, Sparc 5s, Sparc 20s and Ultras. For each of the three benchmarks, these curves have similar shape, with the minimum of the Paragon execution time curve occurring at a larger number of nodes because of the Paragon's higher communication bandwidth.

Figure 6(a) shows the speedups obtained for the cluster of workstations using up to 14 Sparc LX processors. Beyond this number of processors, the cluster is heterogeneous and speedup is difficult to define. For the cluster of workstations, near linear speedups for `cgcut2` and `cgcut3` were achieved for up to 6 processors; for the larger problem, `concat`, the behavior was similar for up to 12 processors. Figure 6(b) plots the speedup values obtained for the Intel Paragon. The maximum speedups achieved on the Paragon were 15.9, 13.6, and 31.0 for `cgcut2`, `cgcut3`, and `concat` respectively. Near linear speedups were achieved for `cgcut2` and `cgcut3` using up to 12 nodes and for `concat` using up to 24 nodes. Speedup for the cluster of workstations starts to fall away from linear at one-half the number of nodes for the same benchmark.

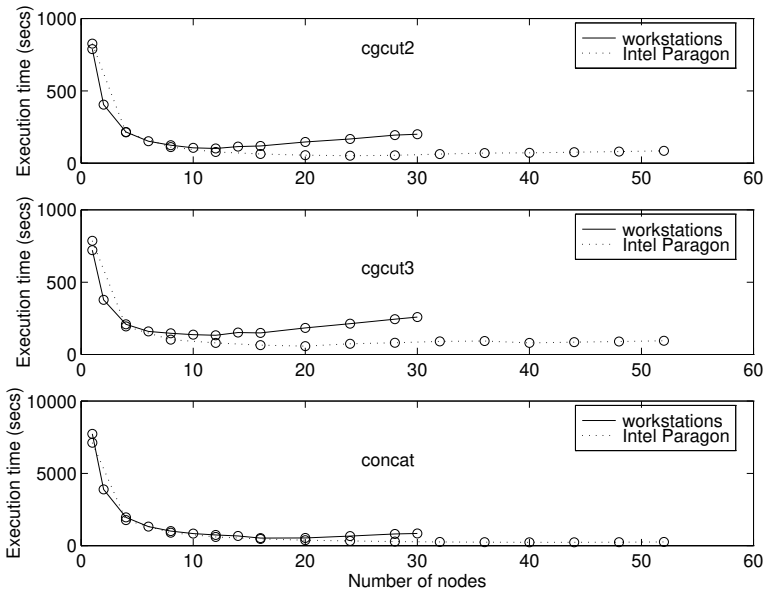



Figure 5: Execution Time for the Three Benchmarks

**Analysis of Parallelization Overheads** Table 4 indicates where the parallel solution spends time in each of the three benchmarks for both platforms. The *Work* column corresponds to the total time spent in the $GETWORK$, $ADD$, $LOCAL_TEST$, $FINISH_TEST$, $UPDATE$ and $UPDATE_WQ$ steps excluding the time spent processing the token. The *Idle* column indicates the total time spent spinning when a processor has no work in its local queue and is waiting for another processor to send it work or for the token to arrive so that it can generate its own local work. The *Token* column corresponds to token communication, i.e.

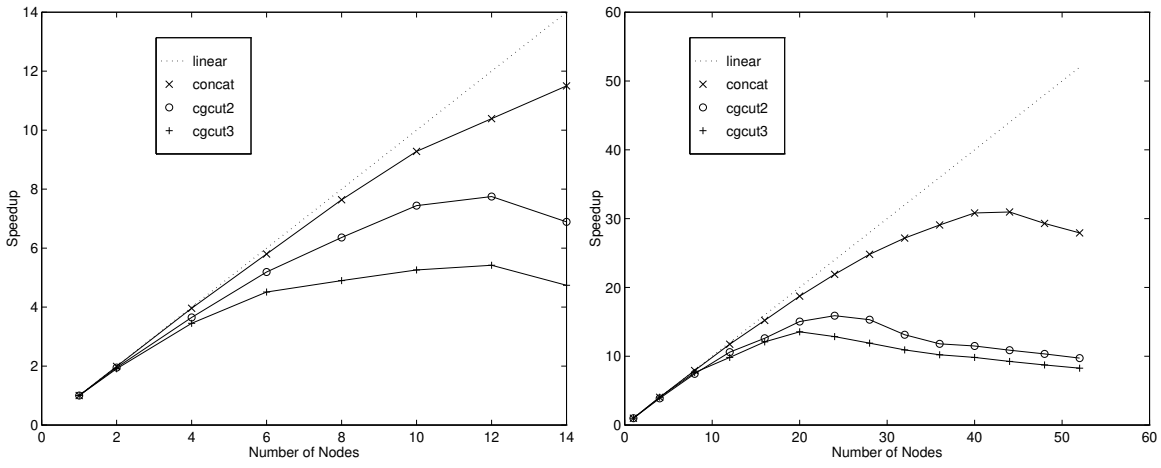



Figure 6: Speedups for the three benchmarks on (a) the workstation cluster, (b) an Intel Paragon

| $P$ | cgcut2 | | | | cgcut3 | | | | concat | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Work | Idle | Token | Load | Work | Idle | Token | Load | Work | Idle | Token | Load |
| Cluster of Workstations | | | | | | | | | | | | |
| 2 | 95.9 | 3.0 | 1.1 | 0.0 | 92.2 | 6.5 | 1.3 | 0.0 | 99.0 | 0.5 | 0.5 | 0.0 |
| 4 | 91.6 | 6.7 | 1.7 | 0.0 | 83.3 | 14.6 | 1.9 | 0.2 | 98.5 | 0.8 | 0.7 | 0.0 |
| 6 | 86.5 | 11.0 | 2.4 | 0.1 | 73.9 | 23.2 | 2.5 | 0.4 | 97.3 | 1.5 | 1.1 | 0.1 |
| 8 | 79.7 | 17.2 | 2.9 | 0.2 | 62.0 | 34.4 | 2.8 | 0.8 | 95.2 | 3.4 | 1.3 | 0.1 |
| 10 | 74.8 | 20.2 | 3.4 | 1.6 | 52.8 | 43.0 | 3.0 | 1.2 | 87.3 | 10.7 | 1.6 | 0.4 |
| 12 | 65.1 | 28.7 | 3.7 | 2.5 | 46.2 | 48.6 | 3.4 | 1.8 | 86.3 | 11.0 | 1.8 | 0.9 |
| 14 | 50.2 | 40.4 | 3.4 | 6.0 | 35.6 | 55.3 | 3.1 | 6.0 | 81.9 | 14.5 | 2.1 | 1.5 |
| 16 | 36.7 | 46.8 | 3.1 | 13.4 | 28.2 | 58.3 | 2.8 | 10.7 | 78.9 | 16.2 | 2.4 | 2.5 |
| 20 | 20.8 | 59.7 | 2.3 | 17.2 | 16.1 | 66.7 | 2.2 | 15.0 | 55.8 | 32.0 | 2.3 | 9.9 |
| 24 | 14.9 | 65.9 | 1.8 | 17.4 | 11.2 | 72.2 | 1.7 | 14.9 | 35.8 | 50.5 | 1.6 | 12.1 |
| 28 | 10.1 | 72.6 | 1.3 | 16.0 | 7.6 | 77.1 | 1.3 | 14.0 | 23.4 | 63.7 | 1.1 | 11.8 |
| 30 | 9.7 | 73.5 | 1.2 | 15.6 | 6.7 | 78.1 | 1.2 | 14.0 | 20.7 | 67.0 | 1.0 | 11.3 |
| Intel Paragon | | | | | | | | | | | | |
| 4 | 98.9 | 0.9 | 0.1 | 0.0 | 98.9 | 0.8 | 0.2 | 0.0 | 99.8 | 0.0 | 0.1 | 0.0 |
| 8 | 93.1 | 6.4 | 0.2 | 0.1 | 90.9 | 8.6 | 0.4 | 0.1 | 99.4 | 0.1 | 0.4 | 0.0 |
| 12 | 90.9 | 8.6 | 0.4 | 0.2 | 82.6 | 17.3 | 0.3 | 0.2 | 97.9 | 1.9 | 0.2 | 0.0 |
| 16 | 83.1 | 15.8 | 0.5 | 0.6 | 77.7 | 21.2 | 0.6 | 0.5 | 95.1 | 4.4 | 0.5 | 0.1 |
| 20 | 76.6 | 22.1 | 0.6 | 0.7 | 69.1 | 29.8 | 0.6 | 0.6 | 93.7 | 5.6 | 0.6 | 0.1 |
| 24 | 68.9 | 28.8 | 0.5 | 1.8 | 53.6 | 44.1 | 0.5 | 1.8 | 91.2 | 8.0 | 0.5 | 0.2 |
| 28 | 56.3 | 38.4 | 0.5 | 4.9 | 42.5 | 53.5 | 0.4 | 3.6 | 88.6 | 10.7 | 0.4 | 0.3 |
| 32 | 38.2 | 54.1 | 0.5 | 7.3 | 34.1 | 60.2 | 0.4 | 5.3 | 84.9 | 14.1 | 0.4 | 0.6 |
| 36 | 33.1 | 58.0 | 0.4 | 8.6 | 28.3 | 64.9 | 0.4 | 6.4 | 80.7 | 17.9 | 0.3 | 1.1 |
| 40 | 29.3 | 60.5 | 0.3 | 9.9 | 25.4 | 66.7 | 0.3 | 7.6 | 77.0 | 20.5 | 0.3 | 2.2 |
| 44 | 25.0 | 63.6 | 0.3 | 11.2 | 21.4 | 70.0 | 0.3 | 8.3 | 70.3 | 26.0 | 0.3 | 3.4 |
| 48 | 22.0 | 66.0 | 0.3 | 11.8 | 19.2 | 71.6 | 0.3 | 9.0 | 61.0 | 33.3 | 0.2 | 5.4 |
| 52 | 18.9 | 68.3 | 0.3 | 12.6 | 15.9 | 74.0 | 0.3 | 9.9 | 53.4 | 38.7 | 0.2 | 7.4 |

Table 4: Percentages of execution time spent in computation, idling, token processing, and load balancing.

(receiving, packing, unpacking, and sending messages) time spent in the $FINISH_TEST$ and $UPDATE$ steps. The *Load* column shows the processing and communication time spent for running the work stealing protocol.

The percentages reported in this table are based on average times across all processors measured over several different runs. Little variation in these times occurred across processors, except when $P > 14$ on the cluster of workstations. In these cases, the time spent token processing and load balancing varies with the pattern processing speed of the workstation.

From this table, we can clearly see the importance of communication performance in the overall execution time. Token processing takes between 1-4% of the total time on each workstation and less than 1% of the total time on each Paragon compute node. The percentage of time spent in token processing is less on the Paragon even though the token circulates more times around the logical ring for all values of $P$. The token processing percentage increases slowly as $P$ increases from $P = 1$ and then decreases slowly once the token is traveling as fast as possible around the ring ($P = 20$ on the cluster and $P = 16$ on the Intel Paragon). After this, the

addition of more nodes decreases the number of token visits to each node.

Load balancing accounts for between 0-18% of the total time on the workstation cluster and 0-13% of the total time on the Paragon for all values of $P$. The cluster's load balancing costs take a larger percentage of the total execution time because the slower communication performance of the local area network makes steal attempts more expensive.

The time for load balancing increases as the work is spread more thinly across an increasing number of processors. `Cgcut3` (and `cgcut2` to a lesser extent) do not generate enough work to keep more than 10 processors busy on the cluster and more than 16 nodes busy on the Paragon. This is reflected in the Idle column where `cgcut2` and `cgcut3` spend approximately one-half of their time idling and attempting to steal work when $P = 14$ on the cluster and one-third of their time idling and attempting to steal work when $P = 24$ on the Paragon. In contrast, `concat` spends 15% of its time idling when $P = 14$ on the cluster and 8% of its time idling when $P = 24$ on the Paragon and therefore generates enough work to keep all processors busy throughout the majority of execution.

## 4.5 Evaluation of Token-based Relaxed Consistency Protocol

### 4.5.1 Effect of Token Speed

The speed at which the token circulates among the processors has a significant impact on the performance of our implementation. Speeding up the token ensures that each processor receives the lock more frequently and that local copies of $B$ are not stale. On the other hand, speeding up the token increases the token processing overhead.

The token speed depends upon three factors: (i) the time each processor holds on to the token, (ii) the communication latency imposed by the underlying software and network, and (iii) how often the processors check for the token's arrival. Recall that each time a processor acquires the token, it can process the backlog of patterns locally tested and stored in $U$. The time each processor holds on to the token depends upon how many patterns from $U$ it processes before sending the token to the next processor. Processors can also slow down the token speed by allowing a token that has arrived to sit unreceived while continuing to process patterns.

| Upper limit | Execution time (in seconds) |
|---|---|
| None | 305 |
| 1000 | 209 |
| 500 | 171 |
| 100 | 162 |
| 60 | 159 |
| 20 | 164 |

Table 5: The effect of limiting the number of elements of $U$ processed in the $FINISH_TEST$ step.

To study the tradeoffs arising from the speed of the token, several experiments were performed by varying the maximum number of patterns a node was allowed to process when holding the token. Table 4.5.1 shows the impact of varying these upper limits on the execution time of `cgcut2` when $P = 6$ on the cluster.

There was a large increase in execution time when a node processed all patterns in $U$ before sending the token. Because of the wide variation in the number of patterns in $U$ at any given time during the execution,

lengthy delays between successive token vists could occur when the token was not forwarded until all of $U$ was processed. Setting this upper limit between 20 and 175 updates on the cluster and between 20 and 40 updates on the Paragon resulted in the best performance for all benchmarks. In general, for good performance, this limit had to be lowered as the number of processors increased. However, lowering the limit too far increased execution time when a node was not able to generate enough work locally (from the patterns processed in $U$) to keep busy until the next token arrival.

Towards the end of execution, there are relatively few patterns in $U$ to process. The maximum number of patterns that can be processed by a node holding the token has little effect on the token speed. To reduce the number of token trips around the ring, a limit on how often a processor checks for the token's arrival was employed. This limit corresponds to the minimum number of patterns that must be in $U$ before checking for the token's arrival. It was determined that values between 0 and 100 on the cluster and values between 0 and 30 on the Paragon worked best for our hardware platforms. Again, the more processors that were used, the lower this minimum had to be.

#### 4.5.2 Comparison to Alternative Approaches

**Static Lock Manager** The use of a central coordinator for lock management is an alternative to the circulating token approach. In this approach, a node would request the lock from the lock manager when it either has many local updates or it is out of work and has at least one local update that could generate local work. The lock manager would keep track of which node last requested the lock and would send a message to this node requesting that it forward the lock along with any updates the requesting node is missing when finished. We note that a static lock manager is used in a replication-based parallel approach to Gröbner basis calculation that exploited weak consistency [7], and also by distributed shared memory systems such as Treadmarks that support lazy release consistency [13].

An increase in execution time was detected on the Paragon for an implementation that used this approach as shown in Table 6. As the number of nodes increased, the lock manager became a bottleneck at the beginning of execution since all the nodes need to update the global pattern list on a regular basis. Our eager token-based protocol eliminates the need to request the lock, thus removing this bottleneck.

| P | Circulating Token Approach | Lock Manager Approach |
|---|---|---|
| 4 | 213 secs | 251 secs |
| 8 | 111 secs | 264 secs |
| 12 | 78 secs | 292 secs |
| 24 | 52 secs | 301 secs |

Table 6: A comparison of execution times for cgcut2 using the Static Lock Manager and Circulating Token approaches for maintaining consistency on the Paragon.

**Hybrid Protocol** Our experiments indicated that there were two phases to the parallel program. During the first phase, all nodes generate updates on a regular basis; passing the token around the ring works well. As time proceeds, however, the new pattern acceptance rate decreases. This indicates that after the initial phase,

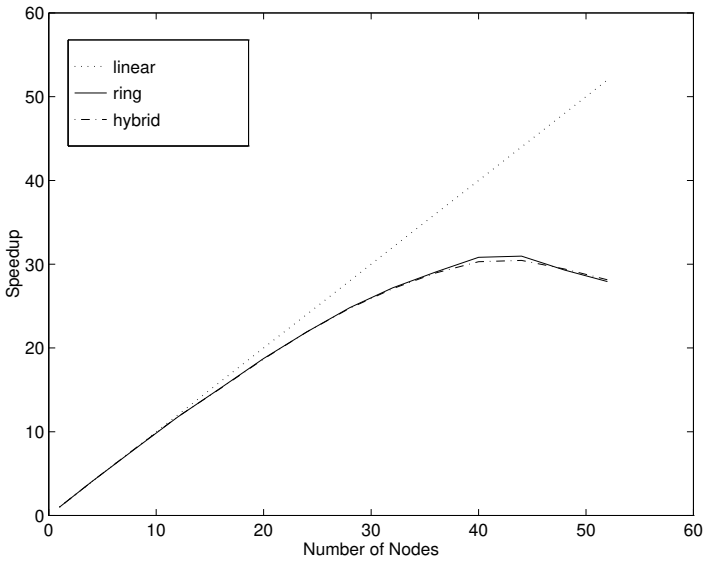



Figure 7: A comparison of the speedups obtained for concat using the hybrid and the eager release consistency protcols on the Intel Paragon.

nodes will require the token more irregularly. When $P$ is large, some nodes could be waiting for the token's arrival with local updates in $U$ while the token is being passed on by a series of nodes with no local updates.

To investigate whether the high idle times for large $P$ in Table 4 might correspond to long waits for the token's arrival, a hybrid approach was implemented that uses the eager token-based protocol for maintaining consistency during the first phase of the program, but switches to a central lock manager based protocol in the second phase. The program enters its second phase when the rate at which updates to $B$ are generated falls below a threshold. This hybrid method eliminates the unnecessary handling of the token by worker processes during the second phase of processing, thereby reducing the wait time for the token when $P$ is large, but adding two extra messages for a token request[4].

Figure 7 compares the speedup using the hybrid approach for the `concat` benchmark to the speedup using the circulating token approach on the Paragon. There is clearly no advantage or disadvantage to the hybrid approach. This indicates that the high idle times for large $P$ in Table 4 are due to a global lack of work and are not due to waiting for the token's arrival. Table 7 gives the percentage time breakdowns for `concat`. Again, the idle time becomes a major component as $P$ increases because the nodes are waiting for a positive response to an outstanding steal request.

## 4.6 Dynamic Load Balancing

During the first phase of the algorithm, it was observed that very little initial work was generated for the three benchmark data sets as they have a small number of input rectangles ($\leq$ 30). When the initial work associated with the input data was distributed among the processors, some nodes ran out of work quickly and

[4]A message to request the token from the central coordinator and a message from the central coordinator to the node currently holding the token.

| $P$ | concat | | | |
|---|---|---|---|---|
| | Work | Idle | Token | Load |
| 4 | 99.9 | 0.0 | 0.1 | 0.0 |
| 8 | 99.5 | 0.2 | 0.3 | 0.0 |
| 12 | 98.2 | 1.5 | 0.2 | 0.0 |
| 16 | 94.9 | 4.3 | 0.8 | 0.1 |
| 20 | 93.4 | 5.7 | 0.7 | 0.2 |
| 24 | 91.2 | 7.8 | 0.7 | 0.2 |
| 28 | 88.2 | 10.7 | 0.7 | 0.3 |
| 32 | 84.6 | 14.0 | 0.7 | 0.7 |
| 36 | 80.4 | 17.7 | 0.7 | 1.2 |
| 40 | 75.7 | 21.6 | 0.7 | 1.9 |
| 44 | 69.2 | 26.8 | 0.8 | 3.3 |
| 48 | 61.3 | 32.7 | 0.8 | 5.2 |
| 52 | 54.1 | 36.4 | 1.3 | 8.2 |

Table 7: Percentages of Time Spent on Computation, Idling, Token Processing, and Load Balancing for the hybrid approach on the Intel Paragon

immediately attempted to steal from other nodes before the token had made even one tour around the logical ring. To eliminate the initial work stealing, the program is run sequentially until a threshold amount of work is generated for each processor. At this point, the work is distributed and the parallel phase of the program begins.

The work stealing approach to dynamic load balancing was tested by performing some experiments on a heterogeneous cluster of Sparc LX and Sparc 5 workstations. The baseline runtimes in Table 2 indicate that a Sparc 5 workstation is roughly 60% faster than a Sparc LX for the benchmarks. Thus, given a cluster of five Sparc LXs and one Sparc 5, the Sparc 5 would be expected to do 24.2 percent of the work (i.e. tasks corresponding to combining and testing pairs of patterns), while each Sparc LX would do 15.1 percent of the tasks. From several runs of the `cgcut2` benchmark, it was found that the Sparc 5 did 25.0 percent of the tasks, while the five Sparc LXs did 12.8, 16.6, 16.2, 14.1, and 15.3 percent of the tasks, respectively. Given the variation in execution time for carrying out a single pattern combination and test step, these measurements indicate that our load balancing implementation is reasonably effective.

## 4.7 Scalability

Table 4 shows that the percentage of time spent on token processing remains below 1% of total execution time on the Intel Paragon. This indicates that the token-based lazy release consistency protocol does not limit the scalability of our approach, at least up to a moderate number of processors.

The primary restriction on scalability is the lack of parallelism in the algorithm during the final stages of execution of a problem. During this phase, we observe a sharp increase in work stealing attempts indicating a lack of work at several nodes. With the smaller problems, `cgcut2` and `cgcut3`, this final stage of execution is a large percentage of the total execution time, thus limiting the speedup achieved.

This is illustrated in Figure 8 where the number of nodes that have less than 9000 patterns pairs in their

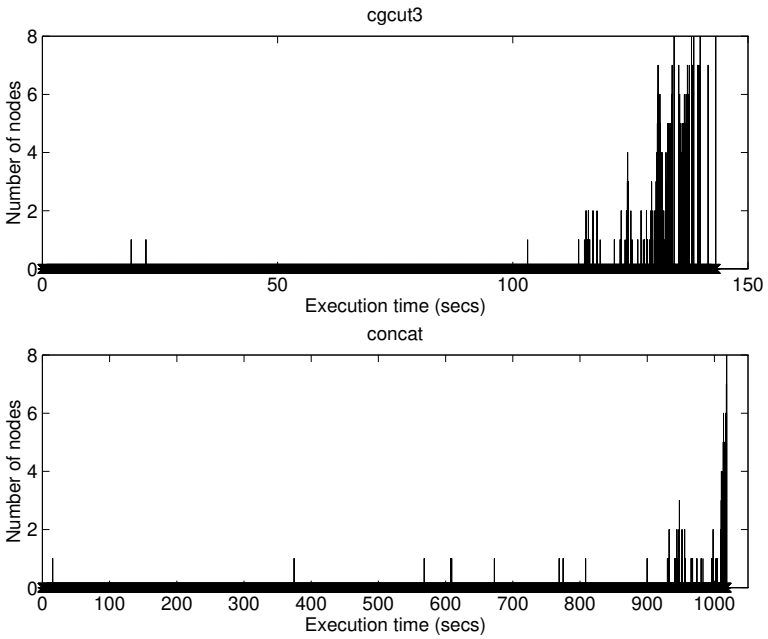



Figure 8: The number of nodes with less than 9000 pieces of work in their local work queue as a function of time for $P = 8$ on the cluster of workstations.

work queues[5] is plotted as a function of time for the `cgcut3` and `concat` benchmarks on an eight processor workstation cluster. For `cgcut3`, at least one-half of the nodes have less than one second of work for over 10% of its total execution time, i.e. from time 130" until 145". For `concat`, at least one-half of the nodes have less than one second of work for less than 1% of the total execution, from time 1010" until 1018".

**Memory Considerations** Another consideration that potentially limits the scalability of any approach based on replicating the set $B$ is the potentially large memory requirements at each node. For the three benchmark tests, the total amount of memory required for data at each node did not exceed $5.4$ Mbytes. This did not pose a problem as the cluster workstations have 32 Mbytes and the Intel Paragon GP nodes have 16 Mbytes of memory.

# 5 Conclusions

We have implemented an efficient parallel solution to a highly irregular optimization problem on a workstation cluster and a dedicated memory multicomputer. The key element of our solution is the replication of an important data structure on all processors. To maintain mutual exclusion and consistency, a token-based lazy consistency protocol is used. A randomized work-stealing protocol is employed for dynamic load balancing of the distributed work queue. Good speedups were obtained for three benchmark problems on the Intel

[5]This number of patterns is equivalent to one second of computation on average.

Paragon. Reasonable speedups were also obtained on the 10 Mbps Ethernet connected cluster, despite the low communication bandwidth of the network.

As future work, we plan to use our replication-based approach for parallelizing branch and bound modifications of Wang's original algorithm [15, 21, 22]. Since the memory considerations arising from replication may prevent very large instances from being solved, we also plan to examine partial replication schemes. We also plan to consider the applicabilty of our approach to other combinatorial optimization problems with similar characteristics to the cutting stock problem.

# References


[1] W. W. Adams and P. Loustanau. *An Introduction to Gröbner Bases*. American Mathematical Society, Providence, Rhode Island, 1994.

[2] Sarita Adve and Mark Hill. Weak ordering: A new definition. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 2–14, May 1990.

[3] H. E. Bal, M. F. Kaashoek, A. S. Tannenbaum, and J. Jansen. Replication techniques for speeding up parallel applications on distributed systems. *Concurrency: Practice and Experience*, 4(5):337–355, August 1992.

[4] J.E. Beasley. OR-Library: Distributing Test Problems by Electronic Mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990. (http://mscmga.ms.ic.ac.uk/info.html).

[5] E.E. Bischoff and G. Wäscher. Cutting and Packing. *European Journal of Operational Research*, 84:503–505, 1995.

[6] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, Santa Fe, New Mexico, November 1994.

[7] Soumen Chakrabarti and Kathleen Yelick. Implementing an Irregular Application on a Distributed Memory Multiprocessor. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 28(7). ACM SIGPLAN Notices, July 1993.

[8] N. Christofides and C. Whitlock. An Algorithm for Two-Dimensional Cutting Problems. *Operations Research*, 25:30–44, 1975.

[9] K.A. Dowsland and W.B. Dowsland. Packing Problems. *European Journal of Operational Research*, 56:2–14, 1992.

[10] H. Dyckhoff. A Typology of Cutting and Packing Problems. *European Journal of Operational Research*, 44:145–149, 1990.

[11] P.C. Gilmore and R.E. Gomory. The Theory and Computation of Knapsack Functions. *Operations Research*, 4:1045–1075, 1966.

[12] R.W. Haessler and P.E. Sweeney. Cutting stock problems and solution procedures. *European Journal of Operational Research*, 54:141–150, 1991.

[13] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for software distributed shared memory. In *Proc. of the 19th International Sympsium on Computer Architecture*, pages 13–21, May 1992.

[14] B. Kröger and O. Vornberger. Solving a Cutting Stock Problem on a Transputer Network. In *Proceedings of the Second Conference of the North American Transputer Users Group*, pages 393–404, Durham, North Carolina, 1990.

[15] J.F. Oliveira and J.S. Ferreira. An Improved Version of Wang's Algorithm for Two-Dimensional Cutting Problems. *European Journal of Operational Research*, 44:256–266, 1990.

[16] S. Setia and P.Y. Wang. Solving A Highly Irregular Problem on Distributed Memory Computers. In *Proceedings of the 1995 International Conference on Parallel Processing Workshop on Challenges for Parallel Processing*, pages 155–162, Oconomowoc, WI, 1995.

[17] Hemal V. Shah and Jose A.B. Fortes. A Quasi-barrier Technique to Improve Performance of an Irregular Application. In *Proceedings of Frontiers '96 The Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 263–270. IEEE Computer Society Press, October 1996.

[18] B.A. Skalbeck and H.K. Schultz. Reducing Trim Waste in Panel Cutting Using Integer and Linear Programming. In *Proceedings of the Western AIDS Conference*, March 1976.

[19] V. S. Sundaram. PVM: A Framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[20] S. Tschöke and N. Holthöfer. A New Parallel Approach to the Constrained Two-Dimensional Cutting Stock Problem. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems*, pages 285–300. Springer-Verlag, Berlin, Germany, 1995.

[21] Francis J. Vasko. A Computational Improvement to Wang's Two-Dimensional Cutting Stock Algorithm. *Computers ind. Engng*, 16(1):109–115, 1989.

[22] K.V. Viswanathan and A. Bagchi. Best-First Search Methods for Constrained Two-Dimensional Cutting Stock Problems. *Operations Research*, 41(4):256–776, 1993.

[23] P.Y. Wang. Two Algorithms for Constrained Two-Dimensional Cutting Stock Problems. *Operations Research*, 31(3):573–586, 1983.

# Appendix

| | cgcut2 | cgcut3 | concat |
|---|---|---|---|
| $n$ | 10 | 20 | 30 |
| $L$ | 40 | 40 | 80 |
| $W$ | 70 | 70 | 70 |
| $\beta$ | 4% | 14% | 1% |
| $(l_i, w_i, b_i)$ | (21,22,1) | (31,43,4) | (21,22,1) |
| | (31,13,1) | (30,41,2) | (31,13,1) |
| | ( 9,35,3) | (29,39,4) | ( 9,35,3) |
| | ( 9,24,3) | (28,38,4) | ( 9,24,3) |
| | (30, 7,2) | (27,37,3) | (30, 7,2) |
| | (11,13,3) | (26,36,4) | (11,13,3) |
| | (10,14,1) | (25,35,3) | (10,14,1) |
| | (14, 8,3) | (24,34,4) | (14, 8,3) |
| | (12, 8,3) | (33,23,4) | (12, 8,3) |
| | (13, 7,3) | (22,32,3) | (13, 7,3) |
| | | (31,21,3) | (31,43,4) |
| | | (29,18,3) | (30,41,2) |
| | | (17,27,2) | (29,39,4) |
| | | (16,25,4) | (28,38,4) |
| | | (15,24,2) | (27,37,3) |
| | | (15,24,1) | (26,36,4) |
| | | (23,14,4) | (25,35,3) |
| | | (21,12,3) | (24,34,4) |
| | | (19,11,4) | (33,23,4) |
| | | ( 9,17,1) | (22,32,3) |
| | | | (31,21,3) |
| | | | (29,18,3) |
| | | | (17,27,2) |
| | | | (16,25,4) |
| | | | (15,24,2) |
| | | | (15,24,1) |
| | | | (23,13,4) |
| | | | (21,12,3) |
| | | | (19,11,4) |
| | | | ( 9,17,1) |

The input rectangles for the three benchmark applications.