A Parallel Solution to the Cutting Stock Problem for a Cluster of Workstations

Lisa D. Nicklas[†] Robert W. Atkins, Sanjeev K. Setia[†] and Pearl Y. Wang
Department of Computer Science
George Mason University
Fairfax, Virginia 22030

Abstract

This paper describes the design and implementation of a solution to the constrained 2-D cutting stock problem on a cluster of workstations. The constrained 2-D cutting stock problem is an irregular problem with a dynamically modified global data set and irregular amounts and patterns of communication. A replicated data structure is used for the parallel solution since the ratio of reads to writes is known to be large. Mutual exclusion and consistency are maintained using a token-based lazy consistency mechanism, and a randomized protocol for dynamically balancing the distributed work queue is employed. Speedups are reported for three benchmark problems executed on a cluster of workstations interconnected by a 10 Mbps Ethernet.

1. Introduction

Cutting and packing are well-known problems in Operations Research [4, 8, 9, 11] that are concerned with finding the optimal allocation of stock materials so that customer orders for given products are satisfied in a cost-effective manner. One long-standing OR problem, the two-dimensional (2-D) cutting stock problem, has numerous applications in the paper, glass, and lumber industries. Here, cutting patterns which 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 large practical instances, serial heuristic algorithms can execute for hours. The preferred computing platform used by OR practitioners for solving these problems is typically a microcomputer or workstation environment. Thus, the development of fast parallel solutions that solve 2-D cutting stock problems on a cost-effective cluster of workstations is a significant contribution towards the OR practice.

Parallelizing classical algorithms that solve a constrained 2-D cutting stock problem poses several challenges arising from the irregular nature of the problem [14]. First, the amount of computation involved is quite sensitive to the problem input and is thus unpredictable. As a result, both the amount and pattern of communication in a parallel solution is also unpredictable. These characteristics make it difficult to obtain an efficient parallel solution – especially in the environment considered in this paper: a cluster of workstations interconnected by a 10 Mbps Ethernet.

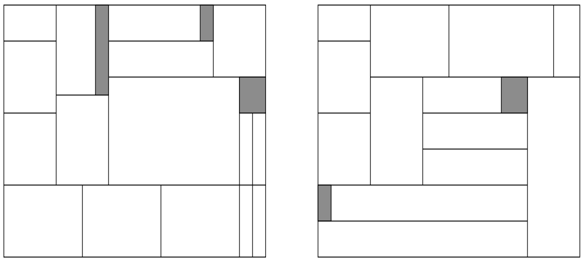
There are, however, aspects of the problem that a parallel solution can exploit. First, as in the case of many combinatorial optimization problems, the problem has a significant amount of parallelism. Second, the data structures used for this problem have no relationship to any physical domain. As such, there is no notion of spatial locality in the program which makes it easier to balance the computational load among the processors involved in the computation. Third, properties of the problem make it possible to use relaxed consistency mechanisms which reduce the communication and synchronization overheads.

We have implemented a parallel solution to the cutting stock problem on a cluster of Sparc workstations. While there have been previous solutions to the cutting stock problem for dedicated multiprocessors [17], this paper focuses on issues that arise in parallelizing this problem on a cluster of workstations.

A key element of our approach is that we replicate an important data structure on all processors. In contrast, previous approaches [17] to this problem have partitioned this data structure among the processors. By using a token-based protocol that maintains consistency while minimizing communication, and by overlapping communication with computation, we are able to obtain reasonable speedups on up to 14 processors even in an environment where the interconnection network has relatively low bandwidth.

This paper is organized as follows. The constrained cutting stock problem and a sequential algorithm for solving it are described in Section 2. Section 3 discusses the design

[†] These authors were partially supported by NSF grant CCR-9409697.



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

Figure 1. Cutting Patterns

and implementation of our parallel algorithm. In Section 4, we present results and discuss the factors that contribute to performance. Section 5 reviews related work, and Section 6 presents conclusions.

2. Problem Statement

The constrained 2-D cutting stock problem is concerned with finding 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.

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:

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

The value of a pattern is the sum of the values of the rectangles 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 [10], the constrained 2-D problems have used two classic approaches [7, 19]. 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 [13, 18]. These introduce upper bound estimates (and 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. These “good" patterns are frequently used by another optimization algorithm to fill customer orders for large quantities of these pieces [11, 15, 19]. 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 Building Algorithm.

2.2. The Serial Building Algorithm

In this approach, guillotine cutting patterns are successively generated by combining partial patterns into larger ones. An input parameter β , $0 \le \beta \le 1$, 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 [19].

The serial algorithm is shown in Figure 2. Here, the procedure $ADD(p, q)$ takes two input patterns p and q and constructs two new patterns r_1 and r_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.

Each newly constructed pattern r is then examined against all current patterns, B, in the $TEST(r, B)$ 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. Suppose 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

Input: A set of rectangles $F = \{p_1, p_2, \ldots, p_n\}$, Tolerance β, 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
  $r_1, r_2 \leftarrow ADD(p, q)$ // creates at most two new patterns if waste $\leq \beta LW$ and all constraints in C are met
  for each $r_i \neq \emptyset$
    $TEST(r_i, B)$ // r_i compared against all elements of B for duplication and dominance
  endfor
  for each r_i that passes TEST
    $UPDATE : B \leftarrow B \cup \{r_i\} - \{s \mid s \in B, s$ is dominated by $r_i\}$
    $UPDATE\_WQ : \forall\, t \in B : \ WQ \leftarrow WQ \cup \{(r_i, t)\} - \{(s, t) \mid s \in B, s$ is dominated by $r_i\}$
  endfor
endwhile

Figure 2. Serial Algorithm

q are the same size, and p cuts a subset of the rectangles cut by q. Either of these two cases imply that p cuts the same quantities of the 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 β parameter is two-fold. They reduce the potentially exponential number of patterns generated by the algorithm, and the β parameter permits the algorithm to keep generated patterns that are not necessarily optimal, but still meet an acceptable waste criterion.

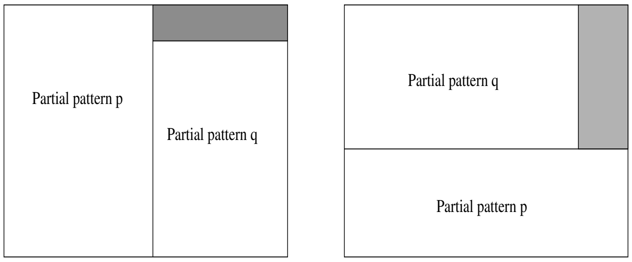
If the new pattern is a duplicate or is dominated by any existing pattern, it is discarded. Otherwise, it is added to B, and any current patterns that dominate the new pattern are removed from B. The $UPDATE$ procedure processes the new pattern subject to the outcome of these duplication and dominance tests. Further, the newly accepted patterns augment the work queue. The new work represents the horizontal and vertical combination with current existing patterns that have not been invalidated.

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. The majority of the computation time occurs when we combine two patterns to create a new pattern and when we test the 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 both the ADD and $TEST$ steps is highly variable and unpredictable. In addition, the amount of new work generated in step $UPDATE\_WQ$ is unpredictable. This necessitates that any approach to parallelization must incorporate mechanisms for *dynamic* load balancing. In our parallel approach, we partition the work queue WQ among the processors involved in the computation and employ a load balancing protocol for keeping the work queues balanced.



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

Figure 3. Possible Combinations

Another challenge in parallelizing the serial algorithm in Figure 2 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 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 r and test r 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 second 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 consistent replicas.

This tradeoff between replication and partitioning arises frequently in parallel computation [2, 6]. While neither approach can completely avoid communication overhead, replicating an object is preferable to partitioning the object if the ratio of reads to updates of the object is large. As we show in Section 4, this ratio is quite large for the algorithm.

A potential problem with replicating a data structure on several processors when the application's consistency requirements are strict (e.g., sequential consistency is required) is the potentially very high overhead for maintaining consistency. Fortunately, the properties of the 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.

Another 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 implementation. Thus, the main data structures used by our algorithm are a replicated set B and a distributed work queue, and the main sources of communication overhead are the protocols for balancing the work queue and maintaining B's consistency.

3.2. Algorithm and Data Structure Design

Figure 4 shows a parallel version of the serial algorithm in Figure 2. We discuss the design of the algorithm and its data structures below.

Distributed Work Queue Each processor obtains work from its local work queue LWQ. A processor's local work queue is modified in step $UPDATE\_WQ$. When a processor runs out of work it attempts to obtain work from the queue of another processor using randomized work stealing [5]. Since the target environment for parallelization is a cluster of possibly heterogeneous workstations, the load balancing algorithm takes the relative speeds of the workstations into account when deciding the amount of work to give another processor.

Consistency Protocol As discussed in the previous section, we replicated B on all the nodes. Since B is logically shared by all the processors, we must use a concurrency control mechanism to prevent simultaneous reads and updates of B by different processors. We must also use a protocol that ensures consistent copies of B.

A straightforward but inefficient way of enforcing these conditions would require each processor to acquire a global lock before the $TEST$ operation, releasing the lock after the $UPDATE\_WQ$ operation. 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 r. If the pattern r passes these tests, then it is added to B in the $UPDATE$ step. Also, any pre-existing patterns in B dominated by r are deleted. Thus, there are three possibilities:(i) r fails the test in which case no modification is made to B, (ii) r passes the test and is added to B, or (iii) r passes the test and is added to B while some existing pattern in B (say s) is deleted.

In the first case above, when $TEST$ fails, there is no conflict between $TEST$ on one processor and an $UPDATE$ on another processor. This is because the pattern r that is rejected by the processor executing $TEST$ would be rejected regardless of any elements that may be added to or deleted from B. In the second and third cases, though, it is possible that the pattern r 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 r 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, r is compared against all the elements in the local (possibly stale) copy of B on a processor. If r passes the local test, it is placed in a local update set U. Before r can be moved from U to B, however, the processor has to obtain any

[1] There is some potential for doing extra work in the third case 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.

Input: A set of rectangles $F = \{p_1, p_2, \ldots, p_n\}$, Tolerance β, 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
    endif
    else
        LWQ ← GETWORK(WQ)
    endelse
    r1, r2 ← ADD(f,g)        // creates at most two new patterns if waste ≤ βLW and all constraints in C are met
    for each ri ≠ ∅
        LOCAL_TEST(ri, B)   // ri compared against elements in local copy of B for duplication and dominance
        U ← U ∪ {ri | ri passes LOCAL_TEST}
    endfor
    GET_LOCK()
    UPDATE_LOCAL(B)          // Bring the local copy of B up to date
    for each ri ∈ U
        FINISH_TEST(ri, B) // ri compared against new elements added to B in UPDATE_LOCAL
        UPDATE: B ← B ∪ {ri} − {s | s ∈ B, s is dominated by ri}
    endfor
    RELEASE_LOCK()
    for each ri ∈ U copied to B
        UPDATE_WQ: ∀ t ∈ B : LWQ ← LWQ ∪ {(ri, t)} − {(s,t) | s ∈ B, s is dominated by ri}
    endfor
endwhile
```

Figure 4. High-level Parallel Algorithm

elements of B that are missing and compare r to these missing elements in the $FINISH\_TEST$ step shown in Figure 4. To prevent another processor from adding new elements to B after entering the $FINISH\_TEST$ step, we use a global lock.

An important consideration in the implementation of the algorithm is the method used to propagate the elements being added to B in the $UPDATE$ step to the other processors. The actual method best used 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 $FINISH\_TEST$.

3.3. Algorithm and Data Structure Implementation

Task Scheduling In the above parallel algorithm, the work consists of combining a pair of patterns from the set B to produce a new pattern and testing the produced pattern for dominance and duplication. The actual amount of computation in these steps is quite small relative to the cost of acquiring a lock. To obtain good performance we must amortize the cost of obtaining a lock by testing several patterns before acquiring the lock.

While its local work queue has tasks, each processor performs the ADD and $LOCAL\_TEST$ steps. If a task creates a new pattern that passes the local testing, the new pattern is placed in a local list U. Once a processor obtains the lock, it tests the patterns in U for dominance and duplication against the current B. The accepted patterns are added to B before releasing the lock. In a sense, we suspend tasks after the ADD and $LOCAL\_TEST$ steps until the processor can obtain the lock to enter the $FINISH\_TEST$ step. Since a processor never blocks for obtaining a lock, we overlap computation and communication.

Mutual Exclusion and Replica Consistency Protocol Since B is logically shared among the processors involved in the computation, we must prevent conflicting accesses to B on different processors. We needed a protocol for acquiring and releasing locks and a consistency mechanism to ensure updates made by a processor are propagated to other processors.

The primary consideration in implementing the locking and consistency protocols on a cluster of workstations is keeping the communication overhead as low as possible. We must minimize the number of messages by combining messages wherever possible and using weak consistency mechanisms [1].

Fortunately, the properties of the cutting stock problem permit the use of relaxed consistency mechanisms. Specifically, the properties of the problem allow a processor to delay update of its B to just before entering the $FINISH\_TEST$ step. Since a processor must acquire the lock before it enters $FINISH\_TEST$, we can reduce the number of messages by combining the message granting a lock with the message propagating the updates. This is similar to the lazy release consistency mechanism used in implementing distributed shared memory on a cluster of workstations [12].

We implemented a lazy release consistency protocol tailored to minimize the number of messages for the cutting stock problem. Updates to B occur throughout the execution of the program. In our parallel implementation with load balancing, the updates to B are distributed evenly among the processors involved in the computation.

Based on these observations, we designed a token-based lazy release consistency protocol with the compute nodes organized in a logical ring that combined the granting of a lock and propagation of updates. 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 all the other nodes involved in the computation the last time they had the token, i.e., in the current round.

When node i receives the token, it processes a fixed number of patterns from U (i.e., patterns that have passed local tests for dominance and duplication). In other words, node i compares the patterns in U against any new patterns received with the token. We add any patterns that pass this test to the local copy of B. In addition, we invalidate any patterns dominated by a newly accepted pattern from the local copy of B. We append these updates to B onto the message containing the token. Node i then transmits the token to the next node in the logical ring (node $i + 1$). When node $i + 1$ receives the token, it carries out the same sequence of steps. Thus, when node $i + 1$ transmits the token to its neighbor (node $i + 2$), along with the lock, node $i + 2$ receives updates made by both i and $i + 1$. When node i receives the token again, it removes the updates (new patterns and invalidations) 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 because we require no message to request a lock. Eagerly propagating the lock to all processors in turn performs well when the work is well balanced among the processors. In this case, each processor generates possible patterns for 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 nodes can do *useful work* in the ADD and $LOCAL\_TEST$ steps, even if they do not have the token.

Dynamic Load Balancing We used a randomized work stealing approach [5] to perform dynamic load balancing. When a processor's work queue is empty, the processor attempts to steal work from another processor picked at random. If the response is negative, the processor attempts to steal work from another randomly chosen processor.

A processor that receives a request for work gives a fraction of its work to the requesting processor provided it has enough remaining work. Along with a work steal request, a processor sends information about its pattern processing rate. The processor giving work uses this rate to estimate the relative speeds of the processor stealing work and the processor giving work. The processor giving work determines the fraction of work to give based on the rate estimates.

A potential problem that can occur during work stealing is a processor may receive work that references patterns it does not yet have in its local copy of B. Recall that in our implementation, a unit of work corresponds to a pair of patterns from the set B. Possibly, 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.

To avoid this situation, the node trying to steal work sends the current number of patterns in its local copy of B along with its steal request[2]. If the work being sent over involves additional patterns, the node giving work sends the needed patterns with the work.

When a processor is unsuccessful in stealing work from all other processors, it sends a message to all other processors indicating that it is finished with the computation. We have optimized the load balancing protocol so that a processor looking for work does not attempt to steal work from processors that have completed.

Program Structure The program is written in the SPMD style with a worker process on each node. One of the worker

[2] Deleting a pattern from the set B involves invalidating it in the array storing B. This array never shrinks in size; thus we need only send a count of the current number of patterns (both valid and invalid) along with the work request, for the node giving work to determine whether to send any patterns along with work.

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, 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.

4. Performance Results

4.1. Platform

Our distributed computing environment consisted of a cluster of SPARC LX and SPARC 5 workstations interconnected by a 10 Mbps Ethernet. The implementation used the message passing library, Parallel Virtual Machine (PVM) [16] Version 3.3.10. All the programs were written in C using `gcc-2.7.1` with optimization level `O3`. The timing measurements were taken when basically no foreign jobs were running on the workstations.

4.2. Benchmarks

We tested our implementation on three input data sets: the first two were obtained from the OR-library of benchmark cutting stock problems [3] and the third is a concatenation of the first two. (See Appendix for these instances.) We used a range of tolerances (β) to control the total number of patterns that could be generated. It is of note that serial (and parallel) versions of the algorithm can have differing execution times for the same input instance, depending on how the input rectangles are ordered. Ordering the input rectangles from smallest to largest value yields the fastest execution times. Input order has the greatest effect on execution time when the number of processors is small. As the number of processors increases, the order of the data is disturbed by the load balancing protocol making the order of the input data less important. In our implementation, we sorted the input values by increasing value.

PROBLEM	(n, L, W, β)	$P = 1$ time
cgcut2	$(10, 40, 70, 4\%)$	862 secs
cgcut3	$(20, 40, 70, 14\%)$	803 secs
concat	$(30, 80, 70, 1\%)$	8496 secs

Table 1. Execution Times on a Sparc LX

As mentioned before in Section 3.1, this cutting stock algorithm has a large read to write ratio for B. Table 2 shows this ratio for the three benchmarks. This table supports our decision to replicate instead of partition the data structure B. With such large read to write ratios, we can maintain the consistency of a replicated B with low communication overhead.

PROBLEM	READ/WRITE RATIO
cgcut2	33,426
cgcut3	26,788
concat	89,394

Table 2. Read to Write Ratio when P=1

4.3. Metrics

To study the performance of our implementation, we present total execution times and speedups where we use the parallel program running on one processor as the speedup baseline. As in the case of the Gröbner basis computation (see Section 5) [6], it is difficult to find the execution time of the best serial algorithm for solving the constrained cutting stock problem. Execution times for serial versions of the Building Algorithm will differ depending on the heuristics used for selecting the pairs of rectangles to be combined next. These heuristics work well for some input instances and poorly for others.

P	cgcut2	cgcut3	concat
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

Table 3. Percentage of serial work performed by parallel version

The normalized speedup suggested in [6] is perhaps a better measure that eliminates the effects of non-determinism in order to analyze the benefits of parallelism. While we do not report normalized speedups, our measurements indicated that the *total work done* 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

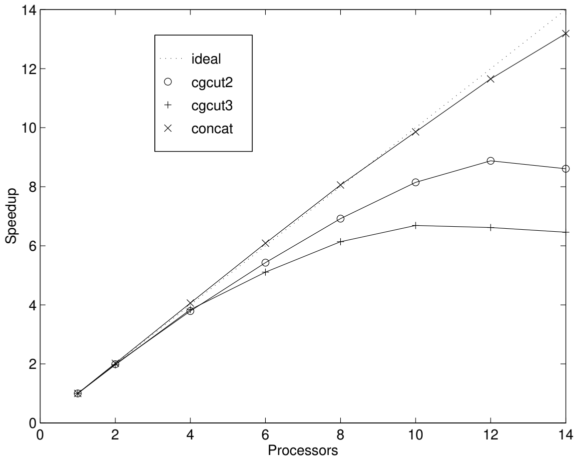


Figure 5. Speedups for the three benchmarks on a cluster of Sparc LX workstations.

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. As such, the speedups reported give a fair indication of the efficacy of our parallelization.

4.4. Findings

Speedup Figure 5 plots the speedups measured for the three benchmark problems considered. We achieve almost-linear speedups for `cgcut2` and `cgcut3` for up to 6 processors, and for `concat` up to 14 processors. We achieved speedups of 8.61, 6.46, and 13.19 for the three problems on 14 processors.

Table 4 shows where the parallel solution spends time in the three benchmarks. 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 corresponds to 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. The "Token" column corresponds to token communication (receiving, packing, unpacking, and sending messages) spent in the $FINISH\_TEST$ and $UPDATE$ steps. The "Load" column corresponds to the processing and communication time spent for running the work stealing protocol. Table 4 reports averages across all processors over several different runs; there was not much deviation of these times across processors.

From the table, we can see that token processing takes between 1-5% of the total time on each processor, and stays roughly constant even as the number of processors increases. Load balancing accounts for between 0-5% of the total time and increases as the number of processors increases. `Cgcut3` (as well as `cgcut2` to a lesser extent) do not generate enough work to keep 14 processors busy. This is reflected in the "idle" column where `cgcut3` spends 43% and `cgcut2` spends 28% of its time idling when $P = 14$. The larger case, `concat`, spends 3% of its time idling and therefore generates enough work to keep all 14 processors busy throughout execution.

Effect of Token Speed Another factor, 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 in turn depends upon two factors: (i) the time each processor "holds on" to the token, and (ii) the communication latency imposed by the underlying software and network. 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.

To study the tradeoffs arising from the speed of the token, we experimented with several different upper limits on the number of patterns a processor was allowed to process when holding the token. Table 4.4 shows the impact of varying these upper limits on the execution time of `cgcut2` when $P = 6$. Note the huge increase in execution time when a processor 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, allowing a processor to process all its patterns could result in lengthy delays between successive visits of the token, leading to poor performance. We found that setting this upper limit between 20 and 175 updates resulted in the best performance for all benchmarks. In general, for good performance, we had to lower this limit as the number of processors increased. However, lowering the limit too far, also increased execution time because a node was not able to generate enough work to keep busy until the next token arrival.

Dynamic Load Balancing We also ran some experiments on a heterogeneous cluster of Sparc LX and Sparc 5 workstations to test the efficacy of our dynamic load balancing scheme. From measurements of a P=1 run, a Sparc 5 workstation is roughly 66% faster than a Sparc LX for the `cgcut2` benchmark. Thus, given a cluster of five Sparc LXs and one Sparc 5, we expect the Sparc 5 to do 24.9% of

P	cgcut2				cgcut3				concat			
	Work	Idle	Token	Load	Work	Idle	Token	Load	Work	Idle	Token	Load
2	98.4	0.0	1.6	0.0	96.1	2.1	1.8	0.0	98.8	0.0	1.2	0.0
4	95.8	2.1	2.0	0.1	93.4	4.0	2.4	0.2	98.9	0.0	1.1	0.0
6	91.0	6.1	2.5	0.4	84.4	12.2	2.8	0.4	98.8	0.0	1.2	0.0
8	87.3	8.9	3.2	0.6	78.3	17.9	3.4	0.4	98.4	0.0	1.5	0.1
10	82.5	12.6	3.7	1.2	67.7	27.8	3.7	0.8	98.1	0.0	1.7	0.2
12	75.2	18.3	4.2	2.3	57.0	37.5	3.8	1.7	97.1	0.5	2.1	0.3
14	63.2	28.0	4.1	4.7	48.9	42.8	3.8	4.5	94.2	2.9	2.3	0.6

Table 4. Percentages of time spent on work, idling, token processing, and load balancing

the tasks (each task corresponds to combining and testing a pair of patterns) while each Sparc LX would do 15.0% of the tasks. From several runs, we found that the Sparc 5 did 25.0% of the tasks, while the five Sparc LXs did 12.8%,16.6%,16.2%,14.1%, and 15.3% of the tasks respectively. Given the variation in execution time for each task and the management work assigned to the first Sparc LX processor, these measurements indicate that our load balancing implementation was reasonably effective.

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 in U processed in the $FINISH\_TEST$ step.

Memory Considerations A disadvantage of replicating the set B is the increase in memory requirements at each node. The total amount of memory required for data at each node did not exceed 5.4 Mbytes while the workstations used were equipped with 32 Mbytes.

5. Related Work

Dynamically generated and modified data sets and dynamically changing patterns of communication that potentially involve *all* the data at each iteration characterize the irregular nature of the cutting stock computation. The calculation of Gröbner bases is an application with similar characteristics. A distributed memory CM5 implementation of the Gröbner basis algorithm was reported in [6]. Their implementation replicated a key data structure (the basis) and used a dynamically balanced distributed work queue. Relaxed consistency mechanisms and application level multi-threading were used to achieve high efficiencies. Our decision to replicate the set B was influenced by the results reported in their paper. However, we tailored the mutual exclusion and lazy consistency mechanisms used in our implementation to exploit the properties of the cutting stock problem and to execute on a workstation cluster with a 10 Mbps Ethernet.

Tschöke and Holthöfer [17] took a partitioned data set approach for a branch-and-bound modification of the Wang Building Algorithm [18]. In contrast, we use a replication-based approach in our implementation. Our experience with this approach suggests that a partitioned approach will not work well on a cluster of workstations interconnected by an Ethernet because of the low communication bandwidth available to the application. In addition, based on the results reported in [6], we conjecture that a replication approach would outperform a partitioned approach even on a system with a high-bandwidth interconnection network.

6. Conclusions

We have implemented an efficient solution to a highly irregular optimization problem on a workstation cluster. The key element of our solution is that we replicate an important data structure on all processors. To maintain mutual exclusion and consistency, we used a token-based lazy consistency mechanism. We employed a randomized work-stealing protocol for dynamic load balancing of the distributed work queue. We obtain good speedups for three benchmark problems using up to 14 processors on a Ethernet connected workstation cluster.

We are currently investigating the applicability of our replication-based approach to the constrained 2-D cutting stock problem in environments where the interconnection network has higher bandwidth, specifically the Intel Paragon and a workstation cluster with an ATM LAN. Since the memory considerations arising from replication may prevent very large instances from being solved, we also plan to examine partial replication schemes.

References

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

[2] 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.

[3] J. 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).

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

[5] R. D. Blumofe and C. 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.

[6] S. Chakrabarti and K. 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.

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

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

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

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

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

[12] 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.

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

[14] S. Setia and P. 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.

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

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

[17] 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.

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

[19] P. 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
β	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)

Table 6. Test Input