INT. J. GEOGRAPHICAL INFORMATION SYSTEMS, 1996, VOL. 10, NO. 6, 699–712

Research Article

Developing parallel procedures for line simplification

JAMES E. MOWER
Department of Geography and Planning and Laboratory for Geographic Information Systems and Remote Sensing Earth Sciences 218, State University of New York at Albany, Albany, N.Y. 12222 U.S.A.
email: jmower@geog.albany.edu

Abstract. This paper explores parallel programming issues that are relevant to the efficient implementation of spatial data handling procedures on current parallel computers through sample implementations of the Douglas line simplification procedure. Using source code-equivalent implementations of the Douglas procedure, this paper analyses the performance characteristics of two parallel implementations, compares their performance characteristics to those of a sequential implementation, and identifies critical components of the parallel implementations that enhance or inhibit their overall performance values. The results of this work show that the selection of appropriate interprocessor communication and load balancing strategies are crucial to obtaining large speedup values over comparable sequential implementations.

1. Introduction

Parallel computing has been applied to a wide array of problems in spatial data handling, ranging from early work in image processing (Rohrbacher and Potter 1977) and drainage basin analysis (Peucker and Douglas 1975) to recent papers on network analysis (Ding *et al.* 1992), cartographic name placement (Mower 1993), line intersection detection (Franklin *et al.* 1989, Hopkins *et al.* 1992), viewshed analysis (Mills *et al.* 1992), and other related topics. Collectively, these papers represent divergent approaches to parallel computing, each drawing upon a specific computing architecture and software development environment to meet the data modelling requirements of its application.

The success of a parallel implementation is most often measured by its speedup factor; the reduction in execution time that it gains over a comparable sequential implementation. Finding the combination of hardware and software environments that generates the greatest speedup factor for an implementation can be quite difficult, especially for the novice parallel developer. Moreover, manufacturers frequently distribute parallel hardware, programming languages, compilers, and operating systems in pre-release versions. Undergoing almost constant revision, these environments rarely provide source code optimization tools. Users are burdened with the task of finding and avoiding expensive or unimplemented programming features, often by gleaning software release notes or by learning from the experiences of other programmers.

This paper is intended to serve the novice parallel programmer by illustrating key programming issues that are relevant to the efficient implementation of spatial data handling procedures on current parallel computers. The discussion of these

0269–3798/96 $12·00 © 1996 Taylor & Francis Ltd.

issues will centre on the implementation of the Douglas procedure for line simplification in a procedure-level programming environment. In summary, this paper:

1. Identifies critical aspects of procedure-level parallel programming environments that effect execution performance.
2. Illustrates these aspects by implementing them in sample parallel code.
3. Quantifies the effects of these aspects on performance by providing an analysis of the run-time characteristics of the example implementations (including comparisons to the characteristics of a functionally-equivalent sequential implementation).
4. Suggests strategies for increasing the performance of a procedure-level parallel implementation.

The results of this work show that the selection of appropriate interprocessor communication and load balancing strategies are crucial to obtaining large speedup values over comparable sequential implementations. Inappropriate selection strategies can result in execution times that, at best, equal or barely surpass those of functionally-equivalent sequential programs.

2. Efficiency issues in parallel processing

A parallel approach achieves a time reduction over a functionally-equivalent sequential approach by dividing a problem into components that can be computed simultaneously, with or without communication among the components.

As an analogy, suppose that 25 identical jigsaw puzzles in unopened boxes were to be solved by one or more people. Clearly, 25 people of equal ability, each working on his or her own puzzle simultaneously, should solve all of the puzzles in the time required for one person to complete a single puzzle. Now suppose that just one of the puzzles were to be solved by the same 25 people working together. Any strategy that the group adopts will require extensive communication to coordinate their activity. It is very unlikely that the group will complete the puzzle in 1/25 of the time required for a single person!

What happens if the puzzles differ in size and complexity? Those who are working independently on small, simple puzzles will finish well before those working on large puzzles. Certainly, the group can reduce the total time required to solve all the puzzles (measured from the starting clock time to the time that the last puzzle is finished) if those who finish early help the remaining workers.

Communication and work partitioning (load balancing) strategies are similarly crucial to the success of a parallel algorithm. Just as a worker can be distracted by a continuous series of instructions from a supervisor or coworker, the instruction stream on a parallel processor is interrupted by interactions with other processors. A good supervisor shifts job assignments to accommodate changing work loads; a parallel algorithm must similarly assure that expensive computing resources are being fully utilized.

2.1. *Parallel approach used in this paper*

To illustrate the effects that varying communication and load balancing strategies imposed upon implementation execution statistics, two procedure-level parallel variants of the Douglas line simplification algorithm will be presented, implemented, run, and analysed. The performance of each implementation will be compared to a functionally-equivalent sequential implementation.

Procedure-level parallel programs are generally implemented on multiple instruction stream, multiple data stream (MIMD) computers. Processors execute independent instruction streams asynchronously using programming constructs that resemble concurrent programming on sequential computers. (Smith 1993).

3. The Douglas procedure for line simplification

Using a vector representation of a cartographic line, the Douglas procedure locates significant vertices among the set occurring between the starting and ending nodes of the line (figure 1). A vertex is considered to be significant if its orthogonal distance from a baseline constructed between the starting and ending nodes is greater than that of any other vertex in the set and greater than a predefined tolerance. The procedure is applied recursively to segments bounded by significant vertices until no new significant vertices are found.

The Douglas procedure (Douglas and Peucker 1973) was selected to demonstrate and evaluate parallel programming techniques because it is simple, effective, and well known. In this regard, the parallel variants that follow retain the simplicity of the original sequential procedure and purposely avoid enhancements suggested by later authors.

Readers interested in implementing line simplification in sequential production environments are encouraged to see Hershberger and Snoeyink (1992). The authors propose an adaptation to the Douglas procedure that operates on the convex hull of a cartographic line, reducing the worst-case running time from $O(n^2)$ in the original procedure to $O(n \log_2 n)$.

4. Implementing the Douglas procedure

Wherever possible, the sequential implementation and the parallel implementations share common source code written in the C programming language for the Thinking Machines CM-5 computer. The CM-5 consists of from 32 to 1024 RISC microprocessors that can be programmed in data-level or procedure-level parallel modes (Thinking Machines Corporation 1991). The sequential and parallel implementations were run on data from modified U.S.G.S. 1:2 000 000 series DLG files

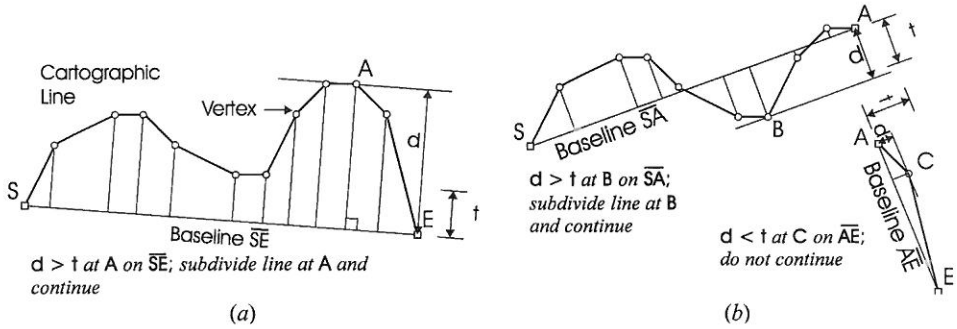

Figure 1. Douglas algorithm. In *a*, the maximum distance between a vertex on a cartographic line and the baseline SE occurs at A. Since the distance *d* at A is greater than the tolerance *t*, vertex A is retained on the simplified line. The cartographic line is subdivided at A and processing continues in *b*. Here, *d* is greater than *t* at B on baseline SA; B is retained and this segment of the cartographic line is subdivided again. On AE, however, *d* is less than *t* so C is not retained and no further subdivision of this segment is performed.

covering the continental United States. Only the hydrography coverages were used for this paper. To enhance the search for lines that fall within the user's window, the author modified the DLG files by prefacing each line with the coordinates of its minimum bounding rectangle (MBR) in internal DLG coordinates. To accommodate user windows extending over DLG boundaries, the MBR for each DLG was calculated and stored in a metafile. At run time, the implementations convert the bounding coordinates of the user's rectangular window, entered in degrees, minutes, and seconds of latitude and longitude, to DLG internal coordinates (figure 2). They compare the user's window against the MBRs of the converted DLGs, flagging intersecting files for processing. For those that intersect, the implementations check the MBR of each line in the DLG for intersection with the user's window, rejecting lines that fall outside it.

4.1. *The sequential implementation*

After determining the user's input window and assembling the list of pertinent DLG files, the sequential implementation scans each DLG in turn, looking for lines that fall within the input window. When it finds a valid line, it sends it to the simplification procedure which recurses until no new significant vertices are found.

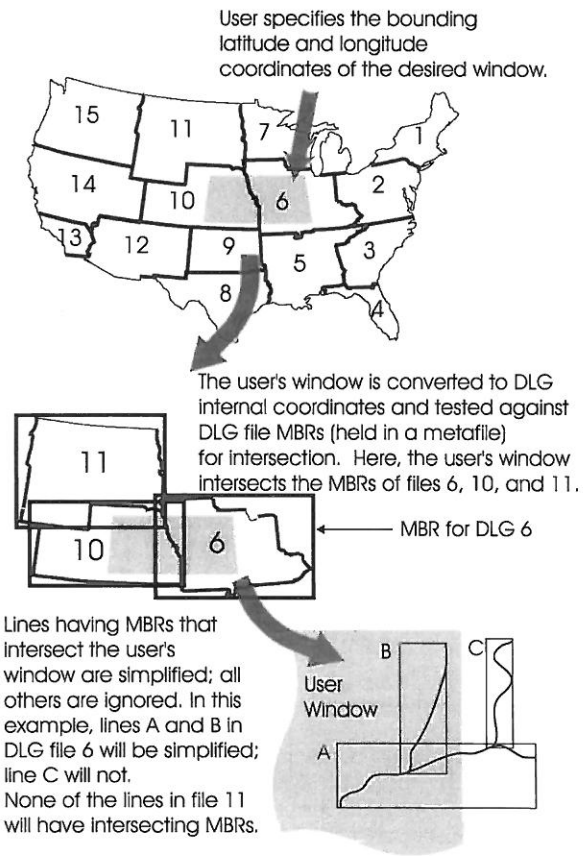

Figure 2. The user's view of the sequential and procedure-parallel implementations of the Douglas algorithm.

The significant vertices are written to the output file and control is returned to the reading procedure. Execution continues until all lines in the relevant DLGs have been simplified.

4.2. *The parallel implementations*

Vaughan, *et al.* (1991) propose three parallel implementations of the Douglas procedure, the first exploiting the parallelism found in the repetitive calculation of distance offsets, the second in the application of the simplification procedure to subdivisions of line segments, and the third which combines elements of the first two implementations. Although their article addresses load balancing issues, it does not provide an explicit indication of the role that interprocessor communication costs play in overall performance characteristics.

To isolate the contribution of these costs in support of a load balancing strategy, the two parallel implementations presented in this paper take a different approach. The first implementation (procedure-parallel-by-file or PPBF) keeps interprocessor communication costs as low as possible by running identical copies of the sequential implementation on each worker processor asynchronously, each processing its own DLG (figure 3). This approach typically causes processor work load imbalances in late processing stages. The second implementation (procedure-parallel-by-line or PPBL) addresses this imbalance by distributing work to processors on demand. To do so, this implementation must incur higher interprocessor communication costs.

Both the PPBF and PPBL implementations use a master-worker model in which a master processor coordinates the distribution of line segments to worker processors. In the PPBF implementation, the master assigns each worker a DLG through a message passing operation. No messages pass between the workers. Each worker reads its own DLG independently and asynchronously from a parallel disk array and writes its output (a list of retained vertices) to a global file, also located on the disk array. Although each worker could have written its output asynchronously to its own file on the parallel disk array, it was found in preliminary testing that this operation took longer to perform than writing to a global file. Hence, both PPBF and PPBL write their output to global files. The PPBF procedure is as follows:
On the master:

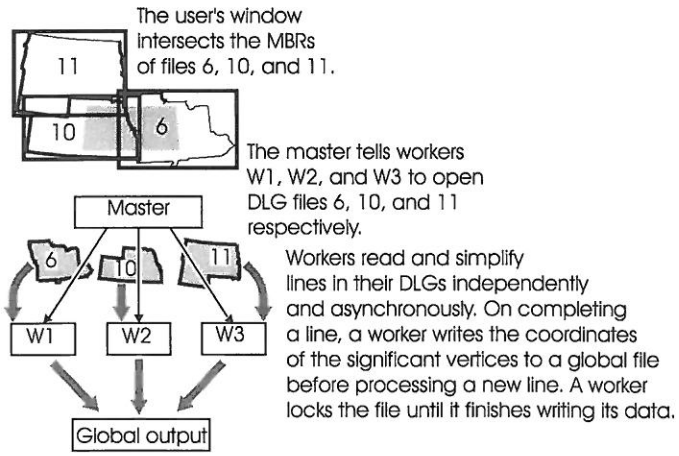

Figure 3. Structure of the procedure-parallel-by-file (PPBF) implementation.

Determine the user's geographical window;
Assign each DLG intersecting the user's window to a unique worker processor;
On each worker:
While unprocessed lines remain in the worker's DLG:
Read a line from the DLG;
If this line intersects the user's window:
Apply the Douglas procedure recursively to the line and all its children;
Save any significant points to a buffer;
Write the buffer of significant points to disk.

Each PPBF worker operates on the lines in its file independently. Therefore, the implementation will continue to run as long as one of the workers still has lines to simplify. It will achieve its greatest efficiency and speedup over a comparable sequential implementation when the data is divided equally among the workers.

Figure 4 shows a user window that falls completely within 1 of 9 rectangular DLGs with identical amounts and distributions of line vertices. Since all of the lines in this example are handled by one worker, the implementation will behave sequentially. Figure 5 shows another user window overlapping all 9 DLGs. Each worker

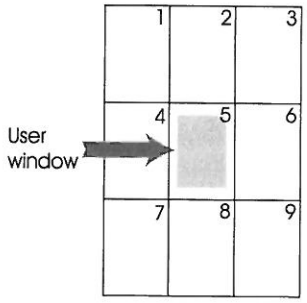

Figure 4. Expected speedup for PPBF when the user's window falls within 1 of 9 rectangular DLGs with identical amounts and distributions of line vertices. Since all of the lines will be simplified by the processor associated with DLG 5, the implementation will perform no better than the sequential implementation on the same window.

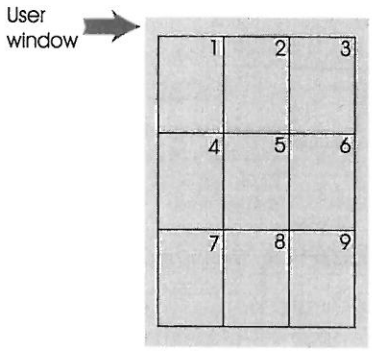

Figure 5. Expected speedup for PPBF when the user's window completely covers all 9 DLGs. Each worker will simultaneously process 1/9 of the lines in the user's window. In this case, the Principle of Unitary Speedup limits the maximum attainable speedup for PPBF over the sequential implementation to a factor of 9.

will require the same amount of time to simplify its lines; and, if each begins its work at the same time, the implementation will complete all of the lines in the 9 DLGs in 1/9 of the time required for the sequential implementation. Figure 6 shows another user window superimposed on the same data, centred on the central DLG and intersecting small portions of the surrounding DLGs. In this example, the time required to process the central DLG will be much greater than that required to process any of the surrounding DLGs. The expected speedup for the PPBF implementation, disregarding I/O costs, can be stated as $S\frac{L}{W_{LMAX}}$ where S is the expected speedup over a comparable sequential implementation, L is the total number of lines in the user window, and W_{LMAX} is the maximum number of lines handled by a single processor. For figure 4, L and W_{LMAX} are identical, giving an expected speedup of 1. In figure 5, W_{LMAX} is 1/9 of the value of L, providing an expected speedup of 9. The expected speedup for figure 6 will not be much greater than 1 since W_{LMAX} is approaching L in this example.

To maintain a higher level of efficiency throughout program execution, the second parallel version (PPBL) distributes valid lines in packets to workers as they become available (figure 7). A worker simplifies all the children of any segment it receives from the master. Each processor writes its output to a global file on the disk array, and waits for another packet of lines from the master processor. Three types of messages must pass between the master and a worker to transmit a packet:

1. the worker requests a line packet from the master;
2. the master transfers a line packet vector to the worker; and
3. the master transfers packet display parameters to the worker.

Both the sending and receiving processors synchronize during message transfer, requiring the sending processor to wait until the receiving processor is ready to accept the transmission. Increasing the number of lines sent in each packet reduces the number of packets, and hence the number of messages that must be sent to the workers. This also allows the master to read ahead of the workers, eliminating an I/O bottleneck. If the packet size is very large, however, load imbalances will occur late in processing as the total number of remaining lines falls below the packet size.

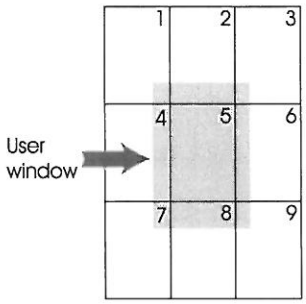

Figure 6. Expected speedup for PPBF when the user's window overlaps one of the DLGs completely, but overlaps the remainder by a small amount. Since most of the lines in the user window fall in DLG 5, its processor will remain active long after the processors for the other windows have completed simplifying their lines. The overall execution time for this PPBF run will be only slightly less than that of the sequential implementation on the same window.

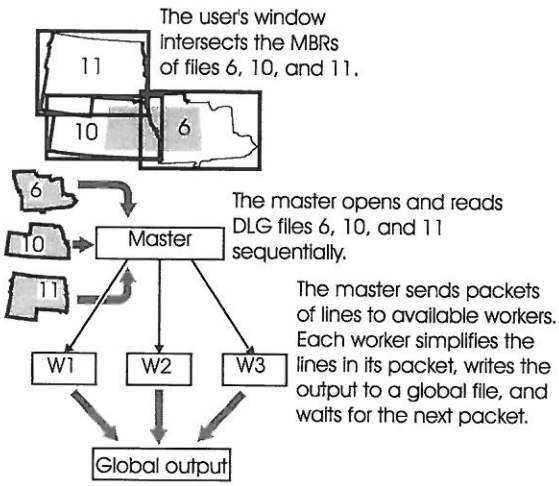

Figure 7. Structure of the procedure-parallel-by-line (PPBL) implementation.

Preliminary testing of the implementation on the largest user window showed that a packet size of 40 lines (approximately 5 per cent of the total number of lines sent to each processor for the largest user window) reduced the running time of the implementation by 38 per cent over the same window using a packet size of 1 (figure 8). The expected speedup of the PPBL implementation, disregarding synchronization and I/O costs, is directly proportional to the number of workers. The PPBL procedure is summarized as follows:

On the master:

Determine the user's geographical window;
Determine which DLGs intersect the user's window;
While unprocessed lines remain in the DLGs;
Read a packet of lines from the current DLG into a buffer;
Send the line buffer to the next available worker;

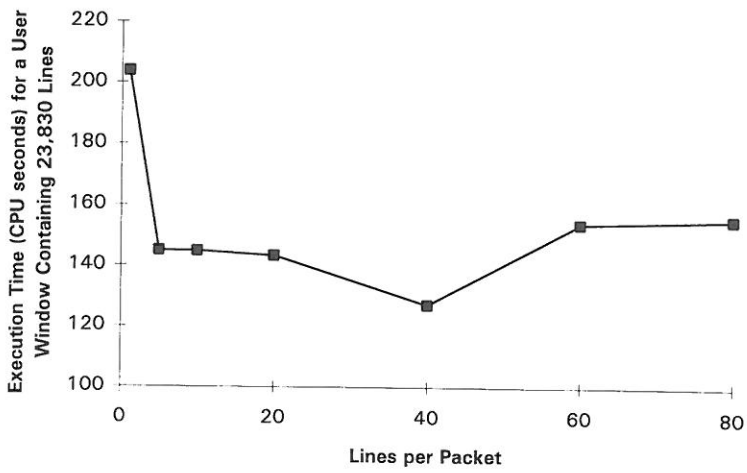

Figure 8. Execution times for the largest window of the PPBL implementation with varying line packet sizes.

```
On each worker:
  Receive a line buffer from the master;
  While unprocessed lines remain in the buffer;
    Get the next line from the buffer;
    If this line intersects the user's window:
      Apply the Douglas procedure recursively to the line and all its
      children;
        Save any significant points to a buffer;
      Write the buffer of significant points to disk.
```

The second implementation of Vaughan, *et al.* (1991) also partitions the workload by line segment. Their implementation differs from PPBL, however, in that they store all unprocessed segments by their endpoints on a global stack, visible to all processors. The stack must be locked so that only one processor can read or write one segment at a time. Whether locking presents a significant processing bottleneck for their implementation is unclear.

5. Testing the performance characteristics of the implementations

The author established the performance characteristics of the sequential and parallel implementations by running them over a series of 25 user windows containing between 16 and 23 830 lines. All of the implementations were executed on a Thinking Machines CM-5 computer at the Northeast Parallel Architectures Center (NPAC) at Syracuse University. The NPAC CM-5 is installed with 32 Sun Microsystems SPARC processors. The sequential implementation was compiled and run as a single-processor procedure-level parallel program on the CM-5 to provide an execution environment comparable to that of the PPBF and PPBL implementations.

Timing data for the sequential and parallel implementations were gathered from calls to timing functions within CMMD, the CM-5 message passing library. Each processor implements its own timing functions and records only times for local operations. No synchronization is required among the workers to collect timing data.

Reported timings for the parallel implementations are broken down by procedure in some of the following figures. Unless otherwise noted, the reported values represent accumulated timings across the processors. The time to process a read operation, for example, would be reported as the accumulated time that it took to perform the operation on all processors. The author calculated accumulated times by summing the collected timer data for an operation across all processors. There is currently no available method to record the real time of a procedure, starting with the wall-clock time at the execution of its first instruction on some processor and ending at the time of the execution of its last instruction on the last remaining processor.

6. Results

Figure 9 shows execution statistics for the PPBF, PPBL, and sequential implementations. The PPBF implementation produced the lowest execution times of the three implementations. The following discussion will explore the characteristics of each implementation that led to these results.

6.1. *Procedure-parallel-by-file (PPBF)*

The PPBF implementation was expected to achieve a speedup over the sequential implementation proportional to the ratio of the total number of lines handled by all workers to the maximum number of lines handled by an individual worker. Figure 10

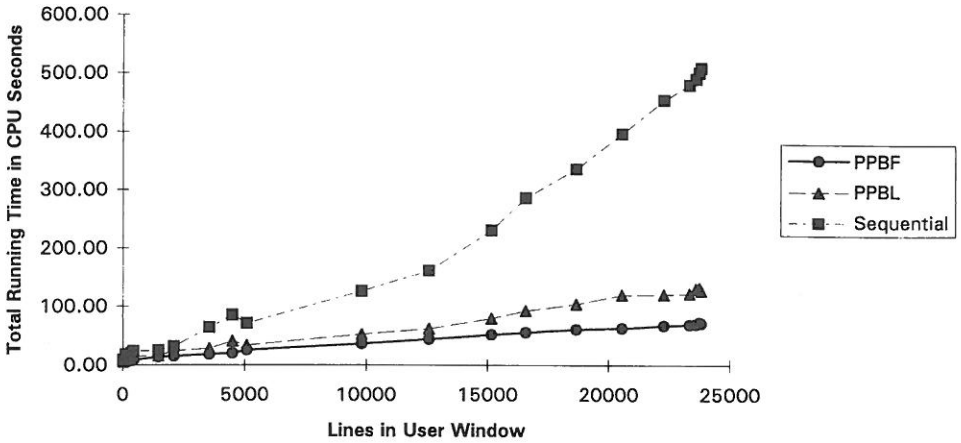

Figure 9. Overall execution times for PPBF, PPBL, and sequential implementations for each user window.

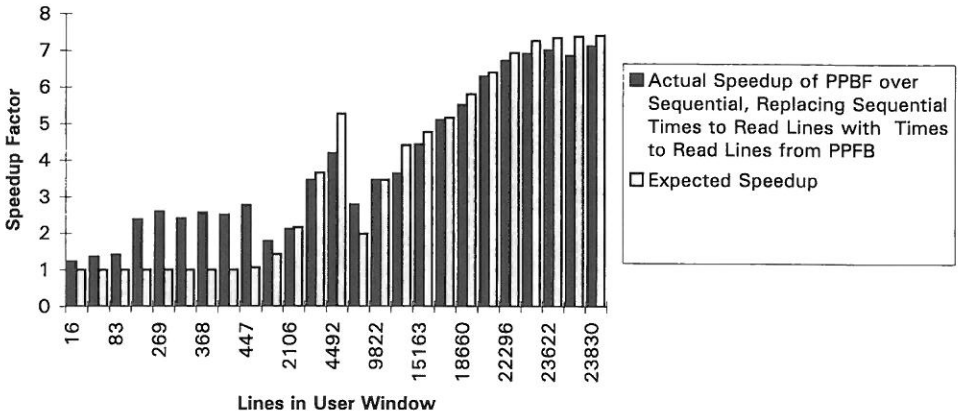

Figure 10. Adjusted PPBF speedup values for each user window. Execution times for reading lines on the sequential implementation were replaced with times to read lines on PPBF.

compares the expected and actual speedup of the PPBF implementation over the sequential implementation. To compensate for very poor parallel I/O performance on read operations in the PPBF implementation, timings for read operations in the sequential implementation were substituted with the parallel I/O timings for the corresponding runs. Actual speedup values generally correspond to expected speedup values for all user windows.

6.2. *Procedure-parallel-by-line (PPBL)*

As expected, the PPBL procedure distributed work evenly across the processors, sending from 768 to 770 lines to each of 31 worker processors for the largest user window (figure 11). In contrast, the PPBF implementation sent 0 lines to 14 of the processors (those with no assigned DLG file) and from 379 to 3212 lines to the rest. Unfortunately, figure 12 shows that the PPBL implementation achieved low overall speedup values.

The poor performance of this implementation is due predominantly to the cost

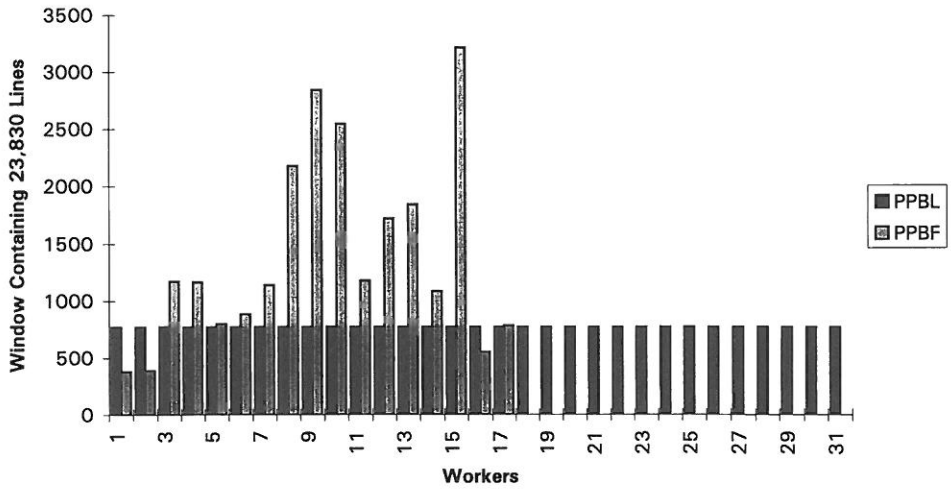

Figure 11. Comparison of the processing load carried by PPBF and PPBL workers for a user window containing 23 830 lines. PPBF workers 18 through 31 did not process any lines.

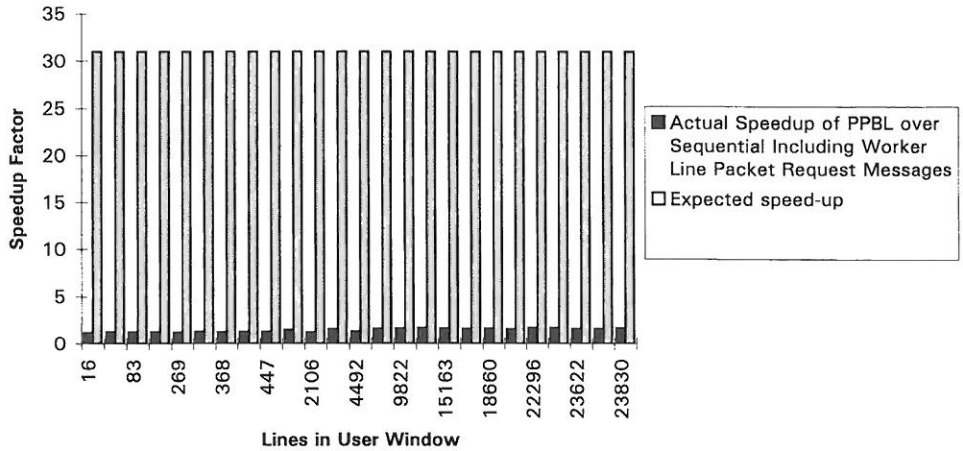

Figure 12. Expected and actual speedup values for PPBL over the sequential implementation for each user window.

of performing message passing operations. Figure 13 compares the time that worker nodes spent performing message passing operations to all other worker operations for the PPBF and PPBL implementations. Although the costs of implementing message passing operations were high for both implementations, they especially dominated the PPBL implementation.

Of the three types of message passing operations performed by a PPBL worker, the first type, requesting a line packet from the master processor, required the longest execution times (figure 14). Most of this time is spent waiting for the master to respond to the request. The amount of time that the worker remains idle depends upon the amount of time that the master processor spends within its message polling loop. For the PPBL implementation, the tasks within the loop were limited to serving requests from the workers.

How much of the difference between the expected and actual speedup values for

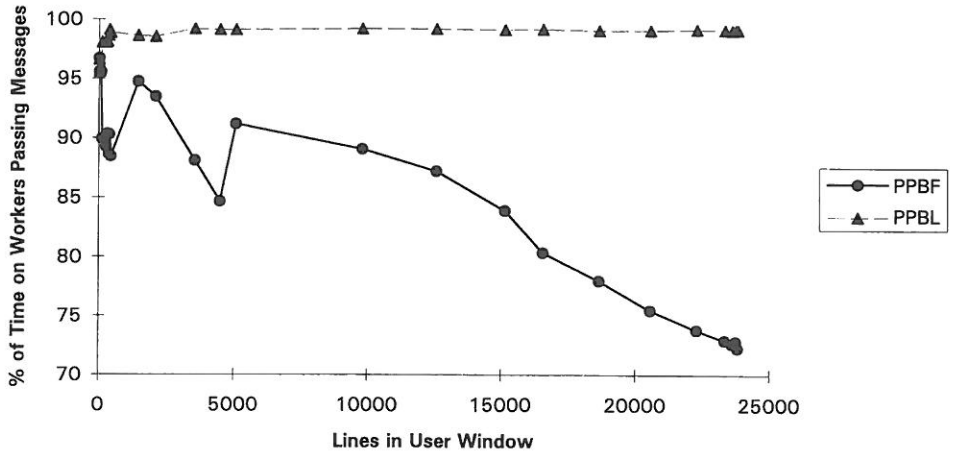

Figure 13. Percentage of accumulated worker execution times spent on passing messages for PPBF and PPBL for each user window.

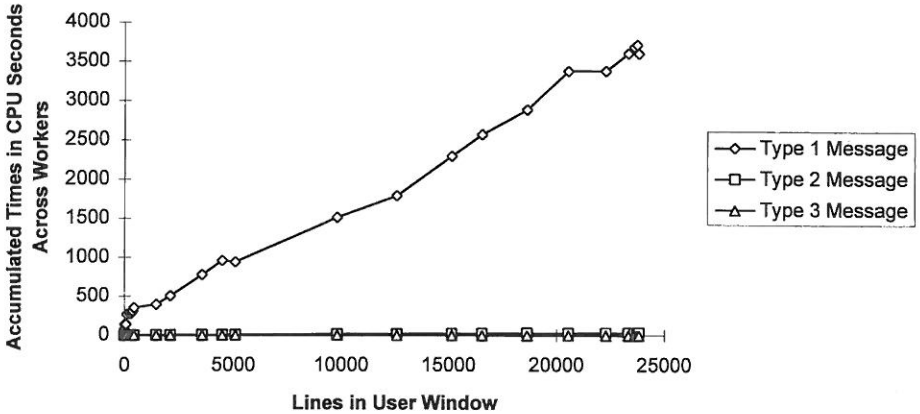

Figure 14. Accumulated time spent on PPBL workers performing message passing by type. Type 1 is a worker request for a packet from the master; type 2 is a line packet transfer to a worker; and type 3 is a line packet display parameter transfer to a worker.

PPBL can be accounted for by message passing operation execution times? Figure 15 shows the speedup values of PPBL over the sequential implementation excluding the cost of line packet request messages on the worker processors. Speedup values increase considerably for all runs but only one exceeded the expected speedup value of 31. The others attained values ranging from 3.32 to 25.36.

It is likely that much of the remaining gap between the expected and actual speedup values for the PPBL implementation could be closed by excluding message passing operations on the master processor, which occupied from 2 to 29 per cent of its running time over the 25 user windows. However, message passing operations on the master processor overlap temporally with those on the workers. If the amount of overlap is unknown, subtracting the execution times for both the master and worker message passing operations from the overall program execution time will double-count overlapping periods. The amount of overlap can be determined by comparing the absolute starting times and running times of the message passing

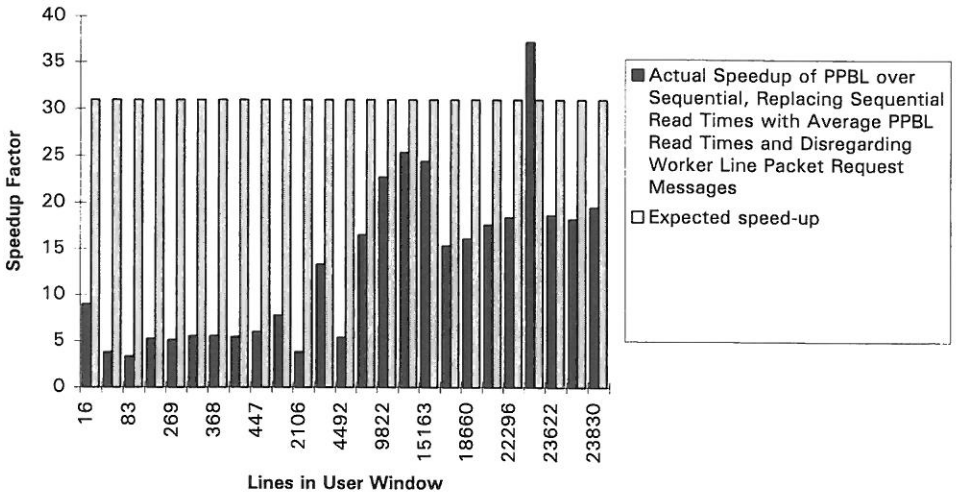

Figure 15. Adjusted PPBL speedup values for each user window. Execution times for reading lines on the sequential implementation were replaced with times to read lines on PPBL. Execution times for worker line packet request messages were disregarded.

functions on the master and worker processors. Reliable absolute timing data could not be collected for this project; therefore timings for message passing operations on the master processor were not deducted from the overall execution times.

7. Conclusions

This paper has shown that the successful development of a parallel implementation must be based upon knowledge of the underlying hardware and software of the target platform. Without it, measures taken to enhance system performance can be thwarted by the cost of their implementation. The PPBF implementation achieved shorter execution times and greater speedup values than the PPBL implementation by sacrificing load balancing for reductions in communication overhead. Most of the costs incurred in the PPBL message passing operations were due to blocking—forcing a processor to wait until another catches up with it to exchange data. Many parallel environments now support asynchronous message passing operations without blocking. Unfortunately, these operations were not fully implemented on the CM-5 at the time of this writing. Their incorporation would have greatly reduced the amount of idle time in the PPBL implementation.

Disk I/O operations, whether performed on parallel or sequential computers, create processing bottlenecks. Parallel read operations in the PPBF implementation performed particularly slowly. Although the speed of these operations can change dramatically with hardware and operating system upgrades, the developer is always advised to keep such operations to an absolute minimum.

The continual, rapid evolution of parallel systems is likely to continue the perception, even among developers, that implementing a parallel algorithm is a black art, a combination of trial and error, blind luck and just a bit of theory. As parallel computing matures, stability and standardization across platforms will eventually bring the theory closer to practice.

Acknowledgments

This work was conducted using the computational resources of the Northeast Parallel Architectures Center (NPAC) at Syracuse University.

References

DING, Y., DENSHAM, P. J. and ARMSTRONG, M. P., 1992, Parallel processing for network analysis: decomposing shortest path algorithms for MIMD computers. In *Proceedings of the 5th International Symposium on Spatial Data Handling* (Columbia: International Geographical Union), pp. 682–691.

DOUGLAS, D. H. and PEUCKER, T. K., 1973, Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, **10,** 112–122.

FRANKLIN, W. R., NARAYANASWAMI, C., KANKANHALLI, M., SUN, D., ZHOU, M. and WU, P. YF., 1989, Uniform grids: a technique for intersection detection on serial and parallel machines. In *Proceedings of Autocarto 9* (Falls Church, Virginia: American Congress for Surveying and Mapping), pp. 100–109.

HERSHBERGER, J. and SNOEYINK, J., 1992, Speeding up the Douglas-Peucker line-simplification algorithm. In *Proceedings of the 5th International Symposium on Spatial Data Handling* (Columbia: International Geographical Union), pp. 134–143.

HOPKINS, S., HEALEY, R. G. and WAUGH, T. C., 1992, Algorithm scalability for line intersection detection in parallel polygon overlay. In *Proceedings of the 5th International Symposium on Spatial Data Handling* (Columbia: International Geographical Union), pp. 210–218.

MILLS, K., FOX, G. and HEIMBACH, R., 1992, Implementing an intervisibility analysis model on a parallel computing system. *Computers & Geosciences*, **18,** 1047–1054.

MOWER, J. E., 1993, Automated feature and name placement on parallel computers. *Cartography and Geographic Information Systems*, **20,** 69–82.

PEUCKER, T. K. and DOUGLAS, D. H., 1975, Detection of surface specific points by local parallel processing of discrete terrain elevation data. *Computer Graphics and Image Processing*, **4,** 375–387.

ROHRBACHER, D. and POTTER, J. L., 1977, Image processing with the Staran parallel computer. *Computer*, **10,** 54–59.

SMITH, J. R., 1993, *The Design And Analysis Of Parallel Algorithms* (Oxford: Oxford University Press).

THINKING MACHINES CORPORATION, 1991, *The Connection Machine CM-5 Technical Summary* (Cambridge, MA: Thinking Machines Corporation).

VAUGHAN, J., Whyatt, J. D. and BROOKES, G., 1991, A parallel implementation of the Douglas-Peucker line simplification algorithm. *Software—Practice and Experience*, **22,** 331–336.