From Proceeedings of the Workshop on Communication and Architectural Support for Network-based Parallel Computing (CANPC' 97), San Antonio, TX, February 97.

Understanding the Performance of DSM Applications *

Wagner Meira Jr.[1] Thomas J. LeBlanc[1] Nikolaos Hardavellas[1]
Cláudio Amorim[2]

[1] Department of Computer Science, University of Rochester, Rochester – NY – 14627
{meira,leblanc,nikolaos}@cs.rochester.edu
[2] COPPE Systems Engineering, UFRJ, Rio de Janeiro, Brazil – 21945-970
amorim@cos.ufrj.br

Abstract. Carnival is a performance measurement and analysis tool that assists users in understanding the performance of DSM applications and protocols. Using traces of program executions, Carnival presents performance data as a hierarchy of execution profiles. During analysis, Carnival automates the inference process that relates performance phenomena to specific causes in the source code or DSM protocol using techniques that focus on the two most important sources of overhead in DSM systems: *waiting time analysis* identifies the causes of synchronization overhead, and produces an explanation for each source of waiting time in the program; *communication analysis* identifies the sequence of requests that result in invalidations, and produces an explanation for each source of communication. We describe these techniques and their implementation in TreadMarks, and show how to use waiting time analysis and communication analysis to improve the running time of two programs from the SPLASH application suite when executed on DEC Alphas connected by a DEC Memory Channel network.

1 Introduction

Shared memory is an attractive programming model because it is easier to use than a distributed-memory model. Software DSM (distributed shared memory) systems offer the simplicity of the shared-memory programming model on cost-effective distributed-memory architectures (including networks of workstations). Although early DSM systems could only provide good performance for a limited class of applications, recent advances at both the protocol level [8, 7, 9] and the architecture level [5, 2, 3] have made DSM a practical and cost effective approach to parallel computing. Nonetheless, synchronization and communication are still major sources of performance degradation in DSM systems.

* This research was supported by NSF grant CCR-9510173, an NSF CISE Institutional Infrastructure Grant No. CDA-9401142, and an equipment grant from Digital Equipment Corporation's External Research Program. Wagner Meira Jr. is supported by CNPq–Brazil, Grant 200.862/93-6. Cláudio Amorim is a visiting professor at the University of Rochester and is supported by CAPES, Brazil.

Reducing or eliminating synchronization and communication in DSM systems is complicated by several factors. First, communication in a DSM system is dictated by the details of the coherence protocol, and therefore is not under the direct control of the user. As a result, the relationship between shared-memory references in the source code and the resulting frequency of invalidations, page requests, and diffs may not be understood by the programmer. Second, DSM systems support the shared-memory model on a range of architectures, where the costs associated with synchronization and communication vary widely. Implicit tradeoffs between the costs of various operations are embedded in the source code (including the data layout scheme, the scheduling strategy, and the degree of parallelism to be exploited), and thus are difficult to discover and change when porting code from one architecture to another. Third, the dynamic nature of synchronization and communication makes it difficult to associate runtime overhead with specific code segments or data structures. Often the cost of an operation is distributed in time (a write operation by one processor causes a subsequent invalidation on another, but only at a later synchronization point) and space (a request by one processor must be satisfied by another), making it difficult to understand the cause of excessive overhead observed during runtime.

There are many tools that help the programmer in understanding and tuning the performance of parallel applications. Many tools identify the location of performance problems. For example, Paradyn [12] measures performance bottlenecks, and presents the resulting performance information in an abstraction hierarchy. MTool [6] measures the time spent by processors waiting for memory requests to be satisfied, and relates memory behavior to code segments. MemSpy [10] identifies the data structures that cause remote memory references, and classifies the misses into various categories, such as invalidation misses and replacement misses. All of these tools measure performance effects and assist the programmer in finding the causes of performance degradation; however, the programmer is responsible for most of the inference process that links an observed effect to a specific cause.

There are two tools that focus on cause-and-effect relationships. Rajamony and Cox [14] implemented a performance debugger that automatically detects unnecessary and excessive synchronization by verifying data accesses between synchronization intervals. StormWatch [4] is a visualization tool for memory system protocols that presents multiple views of memory access operations, including performance slices that capture relationships between individual memory events, exposing causality in memory operations.

In this paper we present two techniques that help to automate the inference process between observed performance phenomena and underlying causes in DSM systems. *Waiting time analysis* is used to understand the causes of synchronization overhead; *communication analysis* is used to understand page reference and invalidation behavior. These techniques, which have been implemented as part of the **Carnival** performance visualization tool, can be used to understand an application's performance and tune the implementation.

In the next section we present two automated techniques that relate ob-

served performance phenomena (synchronization and communication overhead) to underlying causes. Section 3 describes how these techniques are implemented within Treadmarks (a DSM system) and **Carnival** (a performance visualization tool). Section 4 shows how to use these techniques to understand and tune the performance of two Splash applications running under Treadmarks on a cluster of DEC Alpha stations connected by a DEC memory channel. Section 5 presents our conclusions and the directions of our future work.

2 Overview of Analysis Techniques

Waiting time analysis and communication analysis are both automated techniques that examine execution traces of DSM programs and produce explanations for parallel overheads in terms of the source code. Waiting time analysis examines traces to discover the set of basic blocks whose execution delayed one processor, causing another to wait at a synchronization point. Communication analysis examines the same traces to discover the access pattern that caused a page to be invalidated, and subsequently requested by another processor. Both techniques present the sources of parallel overheads in order of their relative contribution to the running time of the application, and highlight the portions of source code that must be modified to reduce the overheads, and hence improve running time.

2.1 Waiting Time Analysis

Many of the overheads associated with parallelism ultimately manifest themselves as *waiting time*; a processor is idle while it waits for another. Waiting time can be introduced at any synchronization point, such as locks and barriers, or whenever a request is issued by one processor that is served by another (e.g., page faults served remotely).

Consider two processors A and B that synchronize at a barrier, execute for some period of time, and then synchronize again at the barrier. Assume A arrives at the barrier before B. We can define the *cause* of waiting time suffered by processor A to be the differences in the execution paths of processor A and B since the last time they synchronized at the barrier. In order to understand why A must wait for B, we compare the execution paths of the processors leading up to the synchronization point, and determine why one path is longer than the other. Anything the two paths have in common is removed as a potential cause of waiting time, leaving only the differences between the two paths as an *explanation* for waiting time. These differences may represent code segments that were executed by one processor but not the other, or communication operations that were required by one processor but not the other.

Waiting time analysis is an automated technique that generates explanations for waiting time in an execution. (See [11] for a detailed description of waiting time analysis and its use in message-passing systems.) The implementation analyzes execution trace files, recording each occurrence of waiting time, and the

set of basic blocks traversed by each processor leading up to a synchronization point. The result of this process is (1) a global execution-time profile of the program, which describes how much time is devoted to various forms of overhead (e.g., load imbalance, contention, insufficient parallelism) that result in idle processors; (2) a *waiting time profile* for each basic block in the program, which helps to identify portions of the source code that deserve special attention; and (3) an explanation for each source of waiting time in terms of the basic blocks that must be modified to reduce it.

Waiting time analysis complements profiling, which focuses attention on the code that appears to dominate the execution, but which cannot capture or quantify indirect effects on waiting time. Since the source code line at which we observe idle time may be distant from the actual cause, we need both waiting time analysis and performance profiles to isolate and understand the behavior.

2.2 Communication Analysis

In DSM systems, communication occurs when a page (the granularity supported by the coherence protocol) is accessed by a processor and that page is not available locally. The page may not be available because (1) this is the first reference to the page (e.g., a cold start) or (2) the page was invalidated as a consequence of write operations by another processor and a subsequent synchronization point. In order to understand why a page reference results in a remote request, we must know the operations that preceded the request (e.g., reads, writes, invalidations); the ordering and type of operations on a page that precede a remote request are the *cause* for that remote request. Analyzing the causes of remote requests is particularly important in DSM systems employing release consistency, since the cause of a remote request can involve multiple processors executing different portions of source code asynchronously.

Communication analysis examines the causes for remote requests (either from the point of view of an individual page or set of pages, or from the point of view of an individual source code line) and from that information infers the access pattern exhibited by a page or source code line. The access patterns are: (1) single-producer-single-consumer, (2) single-producer-multiple-consumer, (3) multiple-producer-single-consumer, (4) multiple-producer-multiple-consumer, (5) migratory, and (6) cold start.

To infer these access patterns, communication analysis uses traces of program executions that contain a record of every page fault and synchronization operation, with a global timestamp for each. Each page fault records the source code line that generated the fault, the nature of the fault (read or write), and the page number. Each synchronization operation records the list of pages that were invalidated as part of the operation. From these traces, the *immediate* cause of each remote request is determined automatically, where an immediate cause is the invalidation that preceded the page fault, and the write faults that generated the write notices at the synchronization point.

Requests to a particular page are usually chained (i.e., one page fault is the cause of another that happens later in time), corresponding to the migration of

the page across processors. We represent causality between requests to a page as a directed graph, called the *communication graph*. We build this graph for each page while traversing the trace file and determining immediate causes for page faults. The nodes in the graph represent page faults (and their immediate cause), and edges in the graph represent causality relationships. There is an edge between two nodes if the write fault explained by one node generates a write notice that is an immediate cause for the fault in the other node. We assign weights to the edges according to the cumulative cost of the communication operations that the edge represents.

Since we are interested in learning why a processor faulted on a page that it owned in the past, we trace back through the edges in the graph until we arrive at a node representing the previous fault on the same page on the same processor. The explanation for the page fault is the set of paths leading back to the immediately previous page fault on the same processor.

We can merge explanations to understand reference behavior across pages. Similar explanations for different pages are combined, allowing us to generalize the reference behavior at a single source code location. The criteria for similarity takes into account the relative importance of each edge's weight in the graph.

The output of this process is, for each data structure, a list of sets of graphs that provide explanations for the page faults on that data structure. As described above, each set of graphs represents one or more pages that behave similarly. These explanations are augmented with communication profiles, which describe how the communication costs during execution are distributed among data structures, source code lines, and causes. With this information, the programmer can identify the source code, data structures, and access patterns that result in page requests, and thereby discover optimizations in data layout or scheduling to improve performance.

It is important to note that communication is a common source of waiting time, and therefore contributes to overhead both on the processor that performs the communication, and on any other processor that must wait for the communication to complete. Therefore, reducing the amount of communication can have the added benefit of reducing waiting time, so that the total savings during execution are much larger than the measured communication time. Waiting time analysis identifies the communication operations that contribute to waiting time; communication analysis identifies the access patterns (and associated pages and source code lines) that cause communication, so that both communication and waiting time can be reduced.

3 Instrumentation and Visualization

An implementation of waiting time analysis and communication analysis requires that we instrument an execution environment to capture the relevant trace information, and present the results of the analysis using an appropriate visualization. We instrumented the Treadmarks DSM system, and use **Carnival** for presentation and visualization. Treadmarks [1] is a DSM system for Unix

systems developed at Rice University that uses a lazy release consistency protocol [8] to reduce communication and false sharing. **Carnival** is a performance analysis and visualization tool developed at the University of Rochester. We first describe the **Carnival** framework, and then describe our implementations of waiting time analysis and communication analysis within Treadmarks.

3.1 Carnival

Carnival is a performance measurement and analysis tool that supports hierarchical abstraction in the presentation of performance data, maintains links between dynamic measurements and the source code, and automates cause-and-effect analysis of performance phenomena. Performance analysis with **Carnival** consists of four steps: (i) instrumentation, (ii) program execution, (iii) automated analysis, and (iv) visualization.

During the instrumentation phase a preprocessor uses static information [11] or user hints, which identify the portions of the code where computation is replicated, to insert instrumentation calls into the application code, Each call records the occurrence of an important event, a timestamp, and the basic block (or data structure) in the source code where the event occurred. We link the instrumented code to a library that generates events in a trace file when the application is executed. After execution, the **Carnival** preprocessor analyzes the trace files, producing a hierarchy of performance profiles, and explanations for various performance phenomena. The results of this analysis (both profiles and explanations) are examined via a Tcl/Tk [13] interface (see figures 1 and 2). More details about the visualization resources provided by **Carnival** can be found in [11].

The instrumentation library is the only architecture-dependent code in **Carnival**. To use **Carnival** with Treadmarks, we only had to implement a global clock within Treadmarks (for recording timestamps) and define the relevant protocol events for our analysis. We implemented a global clock by broadcasting one processor's clock value using the DEC Memory Channel [5]. The accuracy of this global clock is on the order of tens of microseconds.

The relevant events in Treadmarks include lock operations, barriers, page requests, and garbage collection. At the application level we record two types of computation: parallel computation represents parallelized code executed by each processor on different data; replicated computation represents redundant execution performed on each processor as a side-effect of parallelization. Time spent during execution is divided into four categories for analysis: (1) computation, (2) idle time (waiting time), (3) local protocol overhead, and (4) daemon overhead caused by remote requests satisfied locally.

3.2 Waiting Time Analysis

In Treadmarks, processors may become idle while performing any one of four operations: (1) lock acquire, (ii) barrier entry, (iii) diff/page requests, and (iv) garbage collection.

We implemented waiting time analysis in **Carnival** using a pipeline of three independent tools. The first tool in the pipeline takes in trace file information and produces as output a pair of execution paths for every instance of waiting time in the execution. Each pair of matching events (such as the beginning and end of a Treadmarks library call) becomes a execution step that is identified by the processor where the events happened, the profiling category and location in the source code, and has a duration associated with it. An execution path is a set of execution steps, so the size is bounded by the product $processors \times states \times basic\,blocks$.

Another tool takes as input the list of waiting time steps and the pair of paths representing each such step, and creates a set of equivalence classes of execution paths (i.e., merges equivalent paths). The output of this tool is a list of waiting time steps expressed in terms of the representative path for an equivalence class. The waiting time step defines the duration of waiting; the representative path defines the percentage of that duration associated with each execution step on the path.

Finally, for each instance of waiting time, another tool removes any redundant steps between the pair of paths that characterizes it. This tool also acts as a filter on the set of characterizations, allowing the user to select individual execution steps for analysis.

3.3 Communication Analysis

Diffs and page requests are the Treadmarks operations that are the focus of communication analysis, which is also implemented as a pipeline of tools. The first tool takes as input a trace containing all page-related events (i.e., page faults, diff requests, invalidations) and maintains a per-page record of the latest operations to affect a page on each processor. When the tool encounters a request event in the trace, it outputs a summary of the request (i.e., processor, source code location, time to satisfy) and its cause, which is defined as the invalidation location and the preceding write-fault information (i.e., processor and source code location).

Another tool takes as input the page request summaries and their causes, and creates, for each page, a causality graph, where the nodes are locations in the code and the edges are causal relations. There is an edge from one node to another if the source location (basic block) associated with the first node caused an access fault in the source location associated with the second node. Both nodes and edges have attributes; the nodes describe the set of processors that read or wrote the page, and the access patterns exhibited, while the edges contain the cumulative request costs and the location of the synchronization operation that produced the invalidation.

The last tool in the pipeline creates a database of causality graphs, which summarize the access patterns in the program and the causes of remote page references. The causality graph for the program merges page causality graphs that are similar in terms of access patterns and causes. This last tool also creates

the visualization interface for examining the access patterns to data structures and sets of pages.

3.4 Visualizing Performance with Carnival

Carnival presents performance profiles and waiting time analysis in the context of the source code. It helps the user quickly identify where in the source code a program spends the majority of its execution time, and where in the code important sources of parallel overhead are introduced.

The primary **Carnival** display window (Figure 1**a**) is divided into two parts. The source code (with line numbers) is presented on the right; information about each scope in the source code appears on the left. The line numbers are presented in a grey scale, where the intensity of the scale represents the percentage of execution time (summed across all processors) spent on a given line of source code. Users can quickly identify places in the code where the most time is spent by scrolling down the line numbers looking for the darkest portion of the scale. **Carnival** also provides other profiling displays, as described in [11].

Two pop-up windows are used for waiting time analysis. The **WT Map** (Figures 1**b** and 2**b**) provides a global perspective of all sources of waiting time; the **Characterization Map** (Figures 1**c** and 2**c**) presents an explanation for a single source of waiting time in terms of the two execution paths involved.

The WT Map lists each source of waiting time, the line number at which waiting occurred, the name of the scope involved, and the percentage of the total waiting time associated with that scope. The color-coded bar indicates the nature of the overheads that are causing the waiting time at that scope, such as load imbalance (LI), insufficient parallelism (IP), and communication and contention (CC). This map is used to navigate within the source code window and to initiate waiting time analysis. Clicking on an entry in the WT Map causes the main display window to shift to the relevant portion of the source code, and the WT Map presents statistics about each cause of that waiting time. These statistics include the percentage contribution of each cause to the total waiting time experienced at that statement, as well as the total waiting time explained by each cause.

Clicking on a characterization in the WT Map produces an explanation for that waiting time in the **Characterization Map**. Color-coded operations for the longer of the two paths are presented on the right side of the window, operations for the shorter path are on the left. The number of occurrences of each operations is given, as is the percentage of the waiting time associated with each scope. Clicking on an scope shifts the source code window to the relevant portion of the code.

The programmer inspects the results of communication analysis via the CA windows (Figure 2**a**), where information about a variable (or set of pages) is organized in a table form.

Each causality graph is represented as an incidence matrix, where the column header identifies the access pattern (using a color code), the source code location of the faults (R for request, I for invalidation, and W for preceding writes), and

the percentage of total page fault cost in the graph associated with that node. The entries quantify the relative frequency of transitions between nodes in the graph. It is also possible to obtain per-processor information by clicking on the top of each column.

4 Examples

In this section we present examples of how **Carnival** can be used to tune applications running on Treadmarks. All experiments were performed on a cluster of eight DEC Alpha Server 2100 nodes connected by a DEC Memory Channel [5]. Each Alpha Server node has four 233 MHz Alpha processors with 256 Mb of memory. Applications are linked to an instrumented implementation of Treadmarks (version 0.9.6), which employs DEC's implementation of TCP/IP on the Memory Channel.

4.1 Excessive synchronization in Water

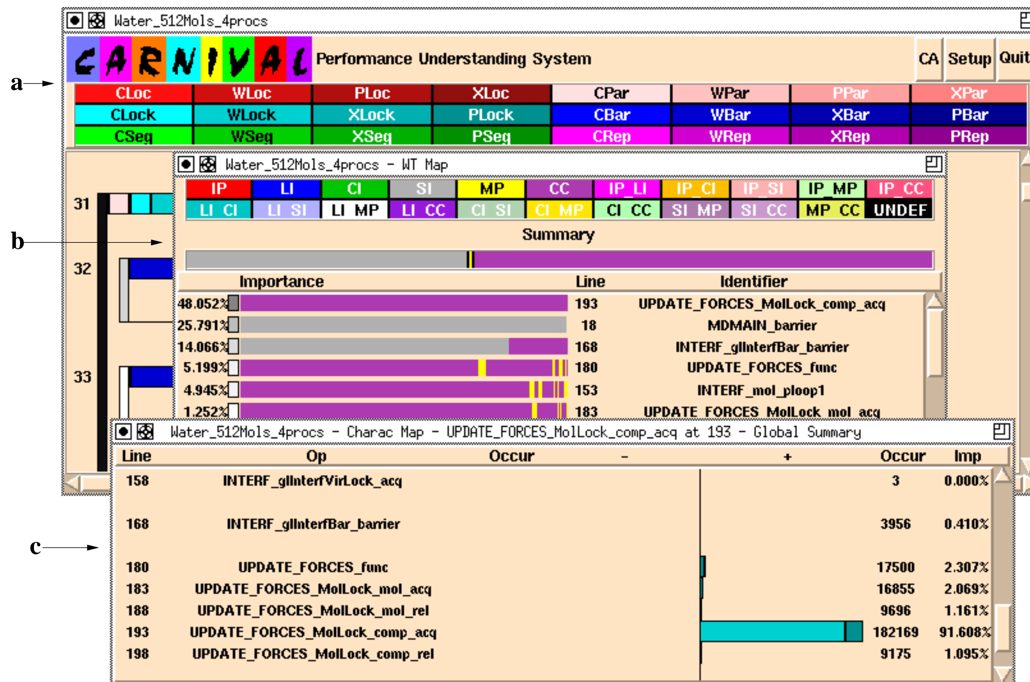

Fig. 1. Carnival visualization of Water

Our first example examines **Water**, a molecular dynamics simulation from the Splash suite [15] that is distributed as part of the Treadmarks release. **Wa-**

ter evaluates forces and potentials that occur over time in a system of water molecules. It uses one large, shared array to represent the molecules being simulated.

We executed three iterations of a 512 molecule simulation of **Water** on four processors and collected the execution traces. The execution took 272 seconds of real time, or 1088 processor seconds. The global execution-time profiles showed that almost 60% of the total processor time was spent waiting for locks and barriers. Waiting time analysis (as shown in the WT Map in Figure 1**b**) shows three major sources of waiting time, which together account for nearly 90% of all waiting time in the application (and thus over 50% of the execution time):

1. Nearly half of the total waiting time is associated with the lock acquire operations that control access to individual molecules in the simulation (UPDATE_FORCES_MolLock_comp_acq). The explanations produced by waiting time analysis (shown in the Characterization Map of Figure 1**c**) show that waiting time at a lock acquire operation is not caused by the actions of another processor (since the left-hand, or negative, side of the explanation is empty); it can be attributed almost entirely to the cost of the lock acquire operation itself (which appears on the right-hand, or positive, side of the explanation).
2. Roughly one quarter of the total waiting time occurs at a barrier (MD-MAIN_barrier). The explanation for this waiting time (not shown in the figure) is the code associated with initialization, which is serialized and therefore produces waiting time on every other processor. Furthermore, the serialized code (which, for simplicity, exploits the same loops used by the parallel code, and therefore includes unnecessary lock operations) is dominated by the cost of lock operations.
3. About 14% of the waiting time occurs at a barrier at the end of the routine `INTERF`, where processors wait until all the forces are updated. Although the explanation suggests some load imbalance among the processors in the function `UPDATE_FORCES`, the majority of the waiting time is again explained by the cost of acquiring and releasing locks.

Our analysis shows that lock acquire operations are the dominant source of overhead for **Water** on Treadmarks. Each acquire operation is expensive and therefore results in overhead. What is surprising, and is only discovered by waiting time analysis, is the extent to which expensive lock operations on one processor indirectly affect other processors, which must wait at barriers or other synchronization points while waiting for a lock acquire to complete elsewhere.

To reduce both the direct and indirect effects of locks, we examined the execution profiles and the waiting time explanations to identify the source code that is causing the overhead. Most of the overhead is associated with two lock acquire calls, which are used to update molecule accelerations within an iteration. Since the modifications associated with the lock are only used in a subsequent iteration, we can modify the program to accumulate the changes locally within an iteration, and then update the global array of molecules. This modification,

which reduces significantly the number of lock acquire operations and was already incorporated into Water in the Splash2 suite [16], improves the execution time by a factor of 17 on four processors.

The original version of **Water** was written for a shared-memory machine, where lock operations are relatively cheap and excessive synchronization is a small price to pay for simplicity. In DSM systems like Treadmarks the tradeoffs are very different, and locks should be avoided wherever possible. This example shows that **Carnival** is particularly helpful in analyzing parallel programs that are being ported to a DSM system from another architecture, since it identifies both direct and indirect consequences of tradeoffs made in one environment, and identifies the source code that must be modified to reflect different tradeoffs in the new environment.

4.2 Scheduling and Data Layout in Ocean

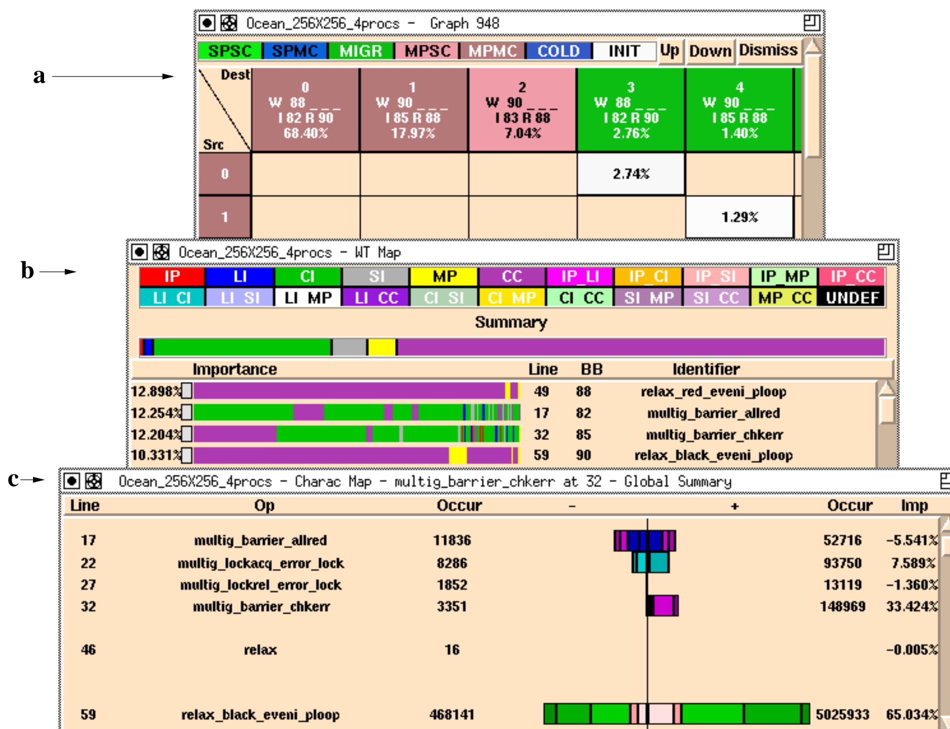

Fig. 2. Carnival visualization of Ocean

Our second example examines **Ocean**, an application in the Splash2 suite [16] that models large-scale ocean movements based on eddy and boundary currents.

The original Splash2 code was ported to Treadmarks by colleagues at the Federal University in Rio de Janeiro without any changes to the data layout scheme. We executed the program on four processors with a grid size of 258 x 258, a grid resolution of 20000, and a time between relaxations of 28800. This execution took 151 seconds.

The global execution-time profiles show that, for **Ocean**, processors are idle for 60% of the overall execution time, while another 31% of the execution time is spent in the Treadmarks protocol (including garbage collection). Of the overall waiting time, about half is spent by processors waiting for page requests to be satisfied, with the other half spent by processors waiting at a synchronization point. Waiting time analysis identifies two parallel loops (`relax_red_eveni_ploop` – basic block 88 and `relax_black_eveni_ploop` – basic block 90) as the source of most of the page requests, and two barriers as the source of most synchronization overhead (Figure 2**b**). Furthermore, the analysis shows (Figure 2**c**) that most waiting time spent at the barriers is caused by the communication in the loops. From this analysis we conclude that communication is responsible (directly or indirectly) for approximately 75% of the overall execution time.

The communication profiles show that the two parallel loops account for 62% of the overall communication in the program. The profiles also show that the variable `multi`, a shared data structure containing the various grids used in the red-black Gauss-Seidel multigrid equation solver, is the only shared variable accessed in those portions of the code. In fact, accesses to `multi` are responsible for 75% of the overall communication cost of the application.

At this point in the analysis, we know that the communication costs of two parallel loops are a major cause of performance degradation and the only variable involved in this communication is `multi`. We use communication analysis to examine the access patterns for `multi` and discover that each page in this data structure has multiple producers and multiple consumers (MPMC). In the graph presented in Figure 2**a**, we can see that 86% of the communication costs can be attributed to a MPMC access pattern (the sum of percentages in columns 0 and 1), and the two loops are always writing on each page (i.e., the data written in basic block 88 is requested by basic block 90 and vice-versa). Furthermore, each page is always accessed by the same set of processors. An examination of the two loops reveals that the boundary conditions do not overlap among processors, and therefore we attribute the MPMC behavior to false sharing.

The Splash2 implementation of **Ocean** adopts a tiling allocation policy to improve the communication-to-computation ratio [16]. Under this allocation, less than two percent of all accesses are to boundary entries shared with another processor. However, using a tiling allocation of sub-matrices of 500K each, coupled with the 8K page size in Treadmarks, means that **every** access to multi under Treadmarks is a shared access. Since the boundaries of `multi` sub-matrices are not aligned on page boundaries, every write access to a page in this data structure generates an invalidation. Adopting the blocked allocation policy of the original Splash version of **Ocean**, and padding sub-arrays to align on 8K page boundaries, alleviates this problem, and improves the running time on four

processors by a factor of 8.

It is not surprising that a program written for a shared-memory machine with relatively small units of coherency exhibits false sharing on a DSM system with large units of coherency, nor is it surprising that padding of data structures in such a program improves performance on a DSM system. The point of this example is to illustrate how waiting time analysis and communication analysis can be used to find the sources of excess communication in the source code, and suggest changes. In this particular example, our analysis lead us to focus on communication in the two loops, even though a global profile would have suggested a focus on synchronization at two barriers elsewhere in the program.

5 Conclusions

In this paper we presented two automated techniques for analyzing the performance of DSM applications: waiting time analysis (which determines the causes of idle processor cycles) and communication analysis (which determines the causes of page requests). We described how these techniques are implemented within **Carnival**, a performance visualization tool, and Treadmarks, a DSM system. We used the **Carnival** interface and our techniques to analyze the performance of two Splash applications, Water and Ocean, on a DEC Alpha implementation of Treadmarks. Our experience demonstrates that these techniques can be used effectively to understand the causes of poor performance, and to identify specific improvements in the source code.

We are continuing to analyze applications using these techniques, to better understand the limits of our techniques, and to improve the way in which performance information is presented to the user by **Carnival**. Furthermore, we plan to compare the performance of applications under Treadmarks and Cashmere [9] (a DSM system under development at Rochester), and consider how best to apply our techniques to understanding tradeoffs in the protocols.

Acknowledgements

We would like to thank Sandhya Dwarkadas for her comments on this work. We also would like to thank Cristiana Seidel and Lauro Whately of the Federal University in Rio de Janeiro for the Treadmarks version of Ocean.

References

1. C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: shared memory computing on networks of workstations. *IEEE Computer*, February 1996.
2. R. Bianchini, L. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. Amorim. Hiding communication latency and coherence overhead in software DSMs. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston,MA, October 1996.

3. M. Blumrich, C. Dubnicki, E. Felten, K. Li, and M. Mesarina. Virtual-memory-mapped network interfaces. *IEEE Micro*, 15(2):21–28, February 1995.
4. T. Chilimbi, T. Ball, S. Eick, and J. Larus. Stormwatch: A tool for visualizing memory system protocols. In *Proceedings of Supercomputing'95*, San Diego, CA, December 1995. IEEE.
5. R. Gillett. Memory channel network for PCI. *IEEE Micro*, pages 12–18, February 1996.
6. A. Goldberg and J. Hennessy. MTool:an integrated system for performance debugging shared memory multiprocessor applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, January 1993.
7. L. Iftode, C. Dubnicki, E. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computing Architecture*. IEEE, February 1996.
8. P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 1992. ACM.
9. L. Kontothanassis and M. Scott. High performance software coherence for current and future architectures. *Journal of Parallel and Distributed Computing*, 29:179–195, November 1995.
10. M. Martonosi, A. Gupta, and T. Anderson. Memspy: Analyzing memory system bottlenecks in programs. *Performance Evaluation Review*, 20(1):1 – 12, June 1992. Reprint of a paper presented in Sigmetrics' 92.
11. W. Meira Jr., T. LeBlanc, and A. Poulos. Waiting time analysis and performance visualization in Carnival. In *Proceedings of SPDT96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 1–10, Philadelphia, PA, May 1996. ACM.
12. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.
13. John K. Ousterhout. *Tcl and Tk Toolkit*. Addison Wesley, 1994.
14. R. Rajamony and A. Cox. A performance debugger for eliminating excess synchronization in shared-memory parallel programs. In *Proceedings of the 4th International Workshop on Modeling, Analysis, and Simulation of COmputer and Telecommunication Systems (MASCOTS)*, February 1996.
15. J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.
16. S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995. ACM.

This article was processed using the LaTeX macro package with LLNCS style