Reducing Synchronization Overhead for Compiler-Parallelized Codes on Software DSMs (extended abstract)

Hwansoo Han, Chau-Wen Tseng, Pete Keleher

Department of Computer Science, University of Maryland, College Park, MD 20742

1 Introduction

One method for increasing the programmability of message-passing machines is to combine powerful shared-memory parallelizing compilers with software distributed-shared-memory (DSM) systems that provide a coherent shared address space in software. Shared-memory parallelizing compilers are easy to use, flexible, and can accept a wide range of applications. Results from several recent studies [1, 5] indicate they can approach the performance of current message-passing compilers or explicitly-parallel message-passing programs on distributed-memory machines.

Unfortunately, load imbalance and synchronization overhead were identified as sources of inefficiency in compiler-parallelized programs. One study of programs parallelized on a 16 processor IBM SP-2 found little time spent explicitly executing barriers (1.5% of execution time), but sequential wait (idle time waiting for next parallel task) and load imbalance (idle time waiting for completion of current parallel task), comprise a large percentage of total execution time (14% and 27%, respectively). The compiler-generated code is balanced computationally, but the underlying DSM and OS effectively add random (and unequal) delays. The extra overhead caused by processing coherence traffic at synchronization points also contributes to idle time.

It is clear from these measurements that reducing idle time caused by synchronization is important for achieving good performance. In this paper we investigate a number of compiler techniques for reducing synchronization overhead and load imbalance. Our techniques are evaluated in a prototype compiler/run-time system [5] using the CVM [3] software DSM as a compilation target for the SUIF [2] shared-memory parallelizing compiler. We develop a number of new techniques, including 1) eliminating barriers by inexpensively detecting communication using local subscript analysis, 2) exploiting lazy release consistency to eliminate barriers guarding only anti-dependences, and 3) customizing nearest-neighbor synchronization for a software DSM. We also empirically evaluate the impact of compiler synchronization optimizations for software DSMs.

2 Background

CVM is a software DSM that supports coherent shared memory for multiple protocols and consistency models [3]. It is written entirely as a user-level library and runs on most UNIX-like systems. Its primary coherence protocol implements a multiple-writer version of *lazy release consistency* [4]. Release consistency allows a processor to delay making modifications to shared data visible to other processors until special *acquire* or *release* synchronization accesses occur. Lazy release consistency postpones the propagation of modifications further; updates to shared data are not made visible to a processor until the next time that processor acquires a released synchronization variable.

The SUIF shared-memory parallelizing compiler [2] was retargeted to generate code for CVM by providing a run-time interface based on CVM thread creation and synchronization primitives. Performance was improved by adding customized support for reductions, as well as a *flush update* protocol that at barriers automatically sends updates to processors possessing copies of recently modified shared data [5].

The fork-join model used by shared-memory parallelizing compilers requires a *broadcast barrier* before the parallel loop body to wake up available worker threads and a *barrier* after the loop body to ensure all worker threads have completed. In previous work, we developed compiler algorithms for eliminating barriers based on the observation that synchronization between a pair of processors is only necessary if they communicate shared data [6]. If *data dependence analysis* shows two adjacent loop nests access disjoint memory locations, then the barrier separating them may be eliminated. If two loop nests access the same data, but *communication analysis* based on data and computation decomposition information proves no remote data is accessed, the barrier may also be eliminated. Finally, if communication analysis identifies data is shared only between neighboring processors, the barrier may be replaced by nearest-neighbor synchronization.

3 Novel Synchronization Optimizations

We describe three enhancements to compile-time barrier elimination.

Communication Analysis Using Local Subscripts Previous communication analysis relied on compile-time information on the data and computation decomposition selected for a program to precisely determine whether interprocessor communication takes place [6]. We find that an alternative communication analysis technique based on *local subscript analysis* can handle simple common cases found in practice, but with less complex analysis.

We assume dependence analysis has already eliminated all array reference pairs between loop nests proven to access disjoint memory locations. Local subscript analysis relies on the compiler selecting a consistent chunk partitioning of parallel loop iterations. Consistent iteration partitioning of loops with identical loop bounds will then always assign the same loop iterations to each processor. If any subscript pair containing parallel loop index is identical, each processor will only access local data. If the subscripts differ by only a constant, then each processor will access remote data a constant number of processors away. Since constant differences in subscripts are usually small, processors will end up accessing remote data on neighboring processors, allowing barriers to be replaced by nearest-neighbor synchronization.

Local subscript analysis is designed to complement full communication analysis. It is less precise, but can be computed efficiently because it only relies on local symbolic information. Local subscript analysis can thus quickly eliminate simple array reference pairs that do not require synchronization, reducing the number of dependences that must be analyzed with communication analysis. Local subscript analysis can also be applied in some cases where standard communication analysis fail, since it relies only on local program information.

Exploiting Lazy Release Consistency An enhancement is made possible by the underlying semantics of lazy-release-consistency software DSMs. In a traditional shared-memory model, synchronization is needed between two loop nests if two processors access shared data, with at least one of the processors performing a write to the shared data. However, in a lazy-release-consistency software DSM, if the shared reference is a read in the first loop and a write in the second loop, no synchronization is needed because writes from the second loop will not become visible to the read in the first loop until synchronization is encountered. In other words, anti-dependences (write-after-read) do not need to be synchronized.

The one exception is when the processor performing a read does not yet possess a copy of the shared data, since it may retrieve a copy of the data with the new values. For scientific computations where iterative computations are the rule, this is rarely the case. In fact, for regular computations, the compiler can prove this situation may only occur the first time the barrier is encountered. We can thus improve performance by replacing a barrier protecting only anti-dependences with a special execute-once barrier which synchronizes processors only the first time it is invoked at each point in the program.

Since anti-dependences may be ignored, the algorithm for inserting barrier synchronization becomes similar to the algorithm for *message vectorization*. The level of the deepest true/flow cross-processor dependence becomes the point where synchronization must be inserted to prevent data races. Synchronization at lower loop levels can be replaced by execute-once barriers.

Customized Nearest-Neighbor Synchronization The third enhancement is to develop special nearest-neighbor synchronization primitives for software DSMs. At some barriers, the compiler can detect communication which only takes place between neighboring processors [6]. To take advantage of this information, we implemented a customized routine for nearest-neighbor synchronization (where each processor has either zero, one, or two neighbors) directly in CVM. The routine sends a single message to each neighboring processor upon arrival, and continues as soon as messages are received from all neighboring processors. In comparison, for normal global barriers all processors send a single message to the barrier master, which broadcasts a reply once all processors have checked in.

The customized nearest-neighbor synchronization has several advantages over standard global barriers. The most important is that nearest-neighbor synchronization allows at least some of the induced load imbalance to smooth out before it delays all processors. However, this benefit usually occurs only if there are multiple nearest-neighbor synchronization events invoked in sequence. If nearest-neighbor synchronization and global barriers are executed in alternating sequence, opportunities to smooth out load imbalance are lessened. Second, the serial bottleneck of the barrier master is avoided. This is not a large advantage for the size of the system that we are currently evaluating, but should be significant for larger systems. Finally, synchronization and data may be aggregated onto common messages, because both flow only between neighboring processors.

4 Experimental Results

We evaluated the performance of our compiler/software DSM interface with five Fortran programs. EXPL and REDBLACK are dense stencil kernels typically found in iterative PDE solvers. JACOBI is a stencil kernel combined with a convergence test that checks the residual value using a `max` reduction. SWM and TOMCATV are programs from the SPEC benchmark suite containing a mixture of stencils and reductions. We evaluated our optimizations on an IBM SP-2 with 66MHz RS/6000 Power2 processors operating AIX 4.1. Programs were automatically parallelized by SUIF, with the resulting C output code compiled by `g++`, then linked with the SUIF run-time system and the CVM libraries to produce executable code.

Measurements show the compiler is effective both at eliminating parallel loops and barriers encountered during compilation, and at reducing the number of parallel loops and barriers actually executed by the application at run time. Table 1 displays the percentage of doalls and barriers eliminated with different levels of optimization. Four versions of each program are created: "merge" merges adjacent parallel regions to reduce parallelism startup overhead, "depend analysis" eliminates barriers based on data dependences, "comm analysis" uses local subscript analysis to eliminate barriers or replace them with nearest-neighbor synchronization, "lazy analysis" eliminates barriers guarding only anti-dependences. Optimizations are cumulative. On average 53% of all barrier executions are eliminated in these five programs, with 6% of barriers replaced by nearest-neighbor synchronization.

Figure 1 displays the impact of synchronization optimizations on application performance on the SP-2, for both small and large data sets. For each graph, the Y-axis measures improvement over unoptimized programs; the X-axis presents three optimized versions of each program for both 8 and 16 processor runs. Except for REDBLACK and SWM, performance for "merge" is the same as that for "depend" and is omitted.

Program	Doalls executed		Barriers executed by program						
	original number	% eliminated	original number	% eliminated				% replaced	
				merge	depend	comm	lazy	comm	lazy
expl	60	67	120	33	33	33	50	33	17
jacobi	40	50	80	25	25	25	50	25	–
redblack	80	75	160	38	63	63	63	13	13
swm	265	33	530	17	33	33	33	–	–
tomcatv	140	71	280	36	36	71	71	–	–
Average	117	59	234	30	38	45	53	14	6

Table 1. Impact of Synchronization Optimizations on Doalls/Barriers Executed

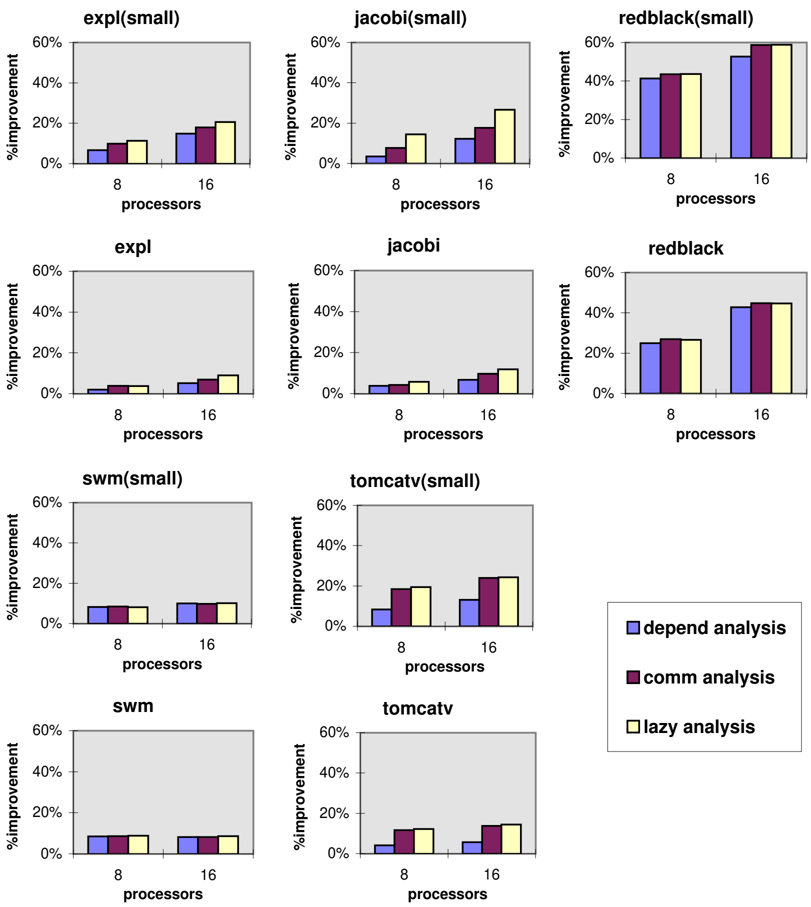

Fig. 1. Impact of Synchronization Optimizations on Speedups (8 & 16 Proc SP-2)

Performance for applications cover a broad range. For 16 processor runs with large data sets, average improvements from synchronization optimizations are 14% for dependence analysis, 17% for communication analysis, and 18% when also eliminating barriers using lazy release consistency. As expected, optimizations have greater impact for smaller data sets and more processors, since synchronization overhead is more significant. For small data set runs on 16 processors, average improvements in speedups due to optimizations increase to 28%, but benefits of optimizations are reduced for 8 processor runs. These results show that synchronization optimizations can improve parallel speedups, particularly as the number of processors increases.

5 Conclusion

In this paper we investigate ways to improve the performance of shared-memory parallelizing compilers targeting software DSMs. We present techniques for reducing synchronization overhead based on compile-time elimination of barriers. Our algorithm extends previous techniques by reducing the synchronization overhead of compiler-parallelized programs on software DSMs. We believe that we are contributing to our long-term goal: effectively running applications that are too complex to be compiled directly to message-passing code.

References

1. A. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel. Evaluating the performance of software distributed shared memory as a target for parallelizing compilers. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997.
2. M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.
3. P. Keleher. The relative importance of concurrent writers and weak consistency models. In *16th International Conference on Distributed Computing Systems*, Hong Kong, May 1996.
4. P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.
5. P. Keleher and C.-W. Tseng. Enhancing software DSM for compiler-parallelized applications. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997.
6. C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.