

Porting and Performance Evaluation of Irregular Codes using OpenMP

Dixie Hisley, Punyam Satya-narayana
HPC Division
U.S. Army Research Lab
APG, MD 21005
hisley,psatya@arl.mil
voice: (410) 278-9156

Gagan Agrawal, Lori Pollock
CIS Department
University of Delaware
Newark, DE 19716
agrawal, pollock@cis.udel.edu
(302) 831-2783

Abstract

Over the last few years, OpenMP has been gaining popularity as a standard for developing portable shared memory parallel programs. At the same time, hardware and software Distributed Shared Memory (DSM) machines have emerged, allowing the programmers to choose between shared memory based and message passing based parallelization mechanisms. As a result, application programmers now have more options available for developing portable parallel programs.

In particular, OpenMP provides directives that allow the programmer to implement a single program multiple data (SPMD) paradigm that is very similar in spirit to message passing, or a loop-level style of parallel programming. In this paper, we present detailed experiments to investigate the porting and performance evaluation of irregular codes using these two styles of developing OpenMP parallel codes. We use two applications and one NAS benchmark, which were parallelized using OpenMP on a 128 processor SGI Origin 2000. Detailed profile data was collected to understand the factors causing imperfect scalability. Our results show that load imbalance and cost of remote accesses are the main factors causing limited speed-up of the OpenMP versions. Also, the coarser granularity and improved data locality of the SPMD versions were the primary reasons for the improved performance of this technique over the loop-level technique.

1 Introduction

Over the last several years, portable mechanisms for developing parallel programs have been standardized. This set includes relatively low-level libraries like the Message Passing Interface (MPI) [For94], higher level languages like High Performance Fortran (HPF) [KLS⁺94] and parallelization directives like OpenMP [DM98]. Unlike the use of vendor-specific libraries and compiler directives, these libraries and language extensions are supported on a large number of systems.

At the same time, Distributed Shared Memory (DSM) systems are emerging as an important class of parallel machines. This includes both the hardware distributed shared memory systems like the SGI Origin 2000 and software distributed shared memory systems like Treadmarks [ACD⁺96]. The main advantage of such systems is that the programmers have the option of programming them using either a shared memory or message passing paradigm. For example, the Origin 2000 supports both MPI and OpenMP. In this paper, we focus on developing irregular codes using the OpenMP directives implemented on the SGI Origin 2000.

In October 1997 and October 1998, the Fortran API and the C/C++ API for OpenMP were released respectively. Since then, OpenMP has been gaining popularity as a standard for developing portable shared memory parallel programs. Prior to OpenMP, vendors that had a directive based API for shared memory programming (SGI, Cray, Kuck and Associates,) only supported loop level parallelism. OpenMP also provides directives that allow the programmer to implement a single program multiple data (SPMD) paradigm;

however, experiments comparing the programmability and performance of these two styles are lacking.

We compare the loop-level and SPMD styles of developing OpenMP parallel code for three irregular codes; IRREG, CG, LES, and evaluate the resulting performance and scalability. IRREG is abstracted from a computational fluid dynamics (CFD) application that uses unstructured meshes to model a physical problem. CG is a NAS kernel benchmark that solves an unstructured sparse linear system, typical of unstructured mesh applications, by the conjugate gradient method. LES is a CFD code used to compute the large eddy simulation of turbulent flow.

Our first development efforts involved using automatic and loop-level OpenMP directives. However, the indirect addressing that is prevalent in these irregular codes makes it difficult to do dependence analysis, thus preventing the compiler from parallelizing the code automatically. Therefore, we hand-inserted OpenMP directives to force the parallel treatment of loops. While this is a simple approach from a programming point of view, we found it difficult to do this such that a large fraction of the code is performing parallel work. Moreover, the first-touch data distribution policy that is the default on the Origin 2000 does not necessarily distribute the data in an optimal fashion for the entire program. It is difficult to reorder the data for better locality with a loop-by-loop approach because of the indirect addressing that is not resolved until run-time.

Next, SPMD versions of IRREG and LES were developed. The SPMD model of parallel programming relies heavily on domain decomposition. While domain decomposition can result in a coarse grain program that exhibits good scalability, it does transfer the responsibility of decomposition from the compiler to the programmer. Once the problem domain is decomposed, the sequential algorithm is followed, but is modified to handle the multiple sub-domains. Also, the data associated with each subdomain can be reordered at run-time for improved locality.

Our experiments show that the SPMD style of programming results in the best scalability. Besides comparing the scalability of these versions, we use hardware counter based performance data to understand the performance of different versions and the reasons for imperfect scalability. Our experimental results show that load imbalance is a frequent problem with shared memory versions. False sharing and synchronization turn out to be insignificant for the shared memory programs in our benchmark set. Besides load imbalance, the cost of remote accesses is another significant factor hindering the performance of the OpenMP versions.

To summarize, our focus in this paper is the development of irregular applications using OpenMP and understanding the programming techniques within OpenMP which can lead to good performance. In particular, we study the SPMD versus loop-level techniques. We also study the factors behind less than perfect scalability using run-time performance data. The rest of the paper is organized as follows. In Section 2, we explain the programming environments and benchmarks used for our experimental study. The OpenMP implementations are described in Section 3. The results from our experiments are presented and analyzed in Section 4 and conclusions are presented in Section 5.

2 Programming Environment and Applications

In this section, we describe the programming environment and applications we used for our experimental study.

2.1 Origin 2000

The Origin 2000 is a distributed shared memory (DSM) architecture. The Origin 2000 utilized for this study is part of the Army Research Laboratory's (ARL) Major Shared Resource Center (MSRC) supercomputing assets. The largest configuration available is comprised of 128 nodes. Each node has 1024 MB of local memory. Each processor is a MIPS R12000 64 bit CPU running at 300 MHz with two 32-KB primary caches and one 8-MB secondary cache.

Our goal is to compare the SPMD and loop-level paradigms for parallel programming using OpenMP as implemented on the Origin 2000. A good comparison of the advantages and disadvantages of these different paradigms for programming parallel machines is thus possible, using the same underlying hardware for computation and communication. We believe that the Origin 2000 is an important and interesting architecture to target for this study, because it supports the shared memory parallel programming paradigm using OpenMP and is a popular commercially available CC-NUMA architecture.

Performance Problem	Event Count
Load imbalance	number of floating point operations issued per process
Excessive synchronization	number of store conditionals
False sharing	number of store exclusives to a shared block

Table 1: Using Event Counts for Performance Problem Identification

Another interesting aspect of the Origin 2000 system is its capability for reporting detailed profile information to the application programmers. The MIPS R12000 is one of the very few systems in which the hardware counters are made visible to the end-users of the machine. The intent of these performance counters is to provide system designers, compiler writers, and application developers with detailed information on the behavior of the system. A small set of events are monitored by the hardware counters, including cache misses, memory coherence operations, floating point operations and branch mispredictions. Because this monitoring is done in hardware, rather than software, it is possible to extract detailed information about the state of the system without affecting the behavior of the program being monitored.

In our study, profiling data is collected by running the codes with *perfer*, a profiling tool which reports a count for the 32 countable event types, with no modifications to the targeted program, and with only a minimal effect on its execution time.

We focused on the subset of event counts that are indicative of specific performance inhibitors to scalability. Table 1 shows some of the performance inhibitors that we examined, and the corresponding event counts that were used to evaluate those potential problems. These possible sources of poor scaling and the event counts which distinguish them were identified in an SGI Education manual distributed with ARL's Origin 2000 system. [need better reference here!].

Load imbalance occurs when each thread is not doing the same amount of work. This can be checked by examining whether all threads issue a comparable number of floating point operations. In the shared memory model, threads read and write shared variables. Therefore, there is no need for explicit message passing. Instead, synchronization is used to protect against race conditions. Excessive synchronization occurs when the number of store conditionals are high. By default, the storage attribute of a variable is shared. Other storage attributes (private, firstprivate, threadprivate) should be selected if possible to minimize synchronization. False sharing occurs when a cache line is invalidated because some word in the line has been written into, and a subsequent reference to another word (that is perfectly valid) in the cache line causes a miss. False sharing may be indicated as a problem when the counts of store exclusive to a shared block are high. Paying attention to granularity, and padding shared arrays can help alleviate this problem. The event counts are returned as both a counter value and an associated typical time. Although the reported typical times are only statistical averages, they do allow one to assess their relative significance in terms of their order of magnitude compared to other events and to the overall time.

In addition to monitoring the events identified by SGI manuals as possible sources of poor scaling, we also monitored cache miss rates, the average time accessing memory/total time and TLB misses.

2.2 Parallel Programming Environments

On the Origin 2000, shared memory parallel programs can be created through three different options:

- Using the compiler directive `-pfa` and relying on the ability of the parallelizing compiler to automatically extract parallelism from loops.
- Using OpenMP to explicitly indicate loop-level parallelism, synchronization, and shared versus local variables.
- Using OpenMP to explicitly indicate SPMD-style parallelism, synchronization, and shared versus local variables.

OpenMP consists of a set of compiler directives and runtime library calls that can be used within Fortran, C or C++ programs. We performed some initial experiments using the automatic compiler directive; however, the results were generally quite poor on our set of applications. Therefore, for this study, we concentrated on

using explicit OpenMP directives as the mechanism for implementing shared memory programming. Also, we used the MipsPro 7.2.1 compiler and compiled the applications with f90 using -O2 optimizations.

2.3 Benchmarks and Problem Sizes

We have focused our study of OpenMP on CG, an irregular kernel available in the NAS Parallel Benchmarks (NPB) and two additional applications which we call IRREG and LES. The NPB set was developed by the Numerical Aerodynamic Simulation (NAS) program at NASA Ames Research Center for the performance evaluation of parallel computing systems [BHS⁺20, SB18, SWY10, WY10]. The NPB mimic the computation and data movement characteristics of large-scale computational fluid dynamics (CFD) applications.

We obtained a beta release of the “Programming Baseline for NPB” (PBN) NAS benchmarks for CG. This version was provided by the PBN working team (pbn@nas.nasa.gov). The rationale behind the PBN version, given by the working team, is to provide the community with a sample OpenMP implementation. This paper focuses on the Class B problem size (75000 nodes) for CG, as it is the the closest in size to realistic problem sizes, as defined by the applications commonly run at the Army Research Laboratory.

An application code we have been focusing on is the Large Eddy Simulation (LES)[Wan95]. LES can be used to characterize turbulent flow, where large length scales signify the domain size and small length scales represent dissipative eddies. Complex turbulent flows usually consist of distinct flow regimes, such as boundary layers, mixing layers, flow separation, re-attachment and recovery flows in the presence of strong adverse pressure gradients. Although small scales are modeled due to their isotropic nature, high performance computing resources are required to capture the large energy carrying length scales. In this paper, a vectorized simulation code is optimized and parallelized for Origin 2000 performance. A realistic simulation of flow past a backward-facing step with problem size of $64 \times 64 \times 64$ is used to study scaling behavior. Periodic boundary conditions are applied in the stream-wise and span-wise directions.

The second application code we have been examining is IRREG[DMS⁺94]. IRREG is abstracted from a computational fluid dynamics application that uses unstructured meshes to model a physical problem. The mesh is represented by nodes, edges that connect two nodes, and faces that connect three or four nodes. For the realistic submarine mesh used in our benchmark runs, the number of nodes, edges, and faces were 92564, 623003, and 504947, respectively.

3 OpenMP Implementation of Irregular CFD codes

In this section, we describe our experience in parallelizing two irregular CFD applications, LES and IRREG using OpenMP on Origin 2000. We had earlier described the computational characteristics of these codes in Section 2.3.

Both of the non-NAS benchmarks, LES and IRREG, were parallelized using the SPMD style of OpenMP parallel programming (as opposed to loop level parallelism) that relies heavily on domain decomposition. While domain decomposition can result in a coarse grain program that exhibits good scalability, it does transfer the responsibility of decomposition from the compiler to the programmer. Once the problem domain is decomposed, the same sequential algorithm is followed, but is modified to handle the multiple sub-domains. The program is replicated on each thread, but has different extents for the sub-domains. Also, data that is local to a sub-domain (no need to share globally) is specified as private or thread-private. Thread-private is used for sub-domain data that need file scope or are used in common blocks. This style of programming is similar in spirit to message passing in that it relies on domain decomposition. Also, message passing is replaced by shared data that can be read by any thread. Synchronization of writes to shared data is required.

For LES, initialization of the data is parallelized using one parallel region for better data locality among active processors. The main computational kernel is embedded in the time advancing loop. The time loop is treated sequentially and the kernel itself is parallelized using another parallel region. In this parallel region, the $64 \times 64 \times 64$ mesh is blocked in the z-direction and each block is tasked to a different processor.

The IRREG code contains a series of loops that iterate over nodes, edges, and faces. The loops over edges and faces involve indirect accesses to memory locations, which are difficult to analyze and parallelize in a loop-level sense. However, a parallel version of the code can be accomplished by partitioning the nodes among the processors. The edges and faces are assigned to the processor which owns a majority of the

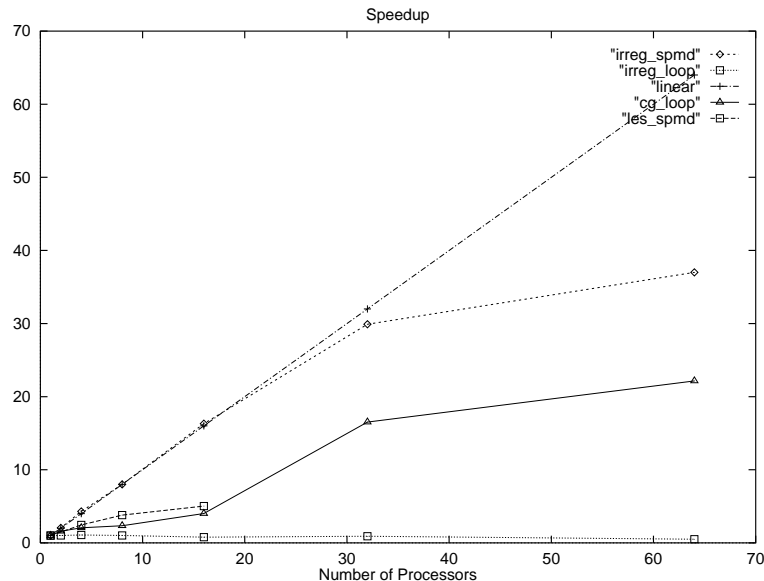


Figure 1: Speedup of OpenMP versions of IRREG, LES and CG

corresponding nodes. The RCB partitioner used in the code does not optimally minimize the number of cut edges (communication effort), but does attempt to reduce the amount of communication and load balance the computational work.

4 Experimental Results

In this section, we present our comparison of the performance of SPMD and loop-level OpenMP parallelism for IRREG, LES, and CG. Performance will be described in terms of scalability, cache friendliness, load balance, the average time accessing memory/total time and a comparison of the relative magnitudes of floating point operations, synchronization costs, false sharing and TLB misses.

The scalability of the OpenMP versions of IRREG, LES and CG is shown in Figure 1, as well as a linear speedup curve for comparison. We see that the SPMD version of IRREG, up to 32 processors, scales quite well, getting a factor of 30.0 on 32 processors. However, scalability falls off above 32 processors with IRREG achieving a speedup of only 37 on 64 processors. The loop-level parallelization of IRREG resulted in almost no speedups. Data and work distribution using specialized partitioners is extremely important for the parallel performance of this code, which could not be achieved through directives for loop-level parallelism. Because of some difficulties in obtaining dedicated time from the Origin 2000, we currently have results for only up to 16 processors for LES. A speedup of 5.1 is obtained on 16 processors. In LES, the matrix solver, the most expensive module, is made cache-friendly by optimizing it for single CPU efficiency. Still, two more modules remain poorly optimized due to inherent data-dependencies and this factor contributes to the imperfect scaling observed for sixteen processors. The loop-level version of CG achieves a speedup of 22 on 64 processors.

In general, a good level of cache friendliness, Figure 2, was seen for all programs at four processors and above. Cache friendliness was examined by looking at the average L1 and L2 miss rates returned by perfex. Although the L1 cache miss rate is as high as .33 for CG at four processors, the L2 miss rate at four processors and above for CG is less than .07. The L1 and L2 miss rates for IRREG and LES are consistently lower than .11 above four processors. Although these numbers indicate good memory performance can be expected, the time spent accessing memory/total time must also be examined to see if the memory access costs are nevertheless overtaking the computational costs.

The average time spent accessing memory/total time for each of our programs is shown in Figure 3. Initially, all the programs show trends of decreasing amounts of overall time spent accessing memory compared to the sequential runs. However, the SPMD version of IRREG and the loop-level version of CG are showing

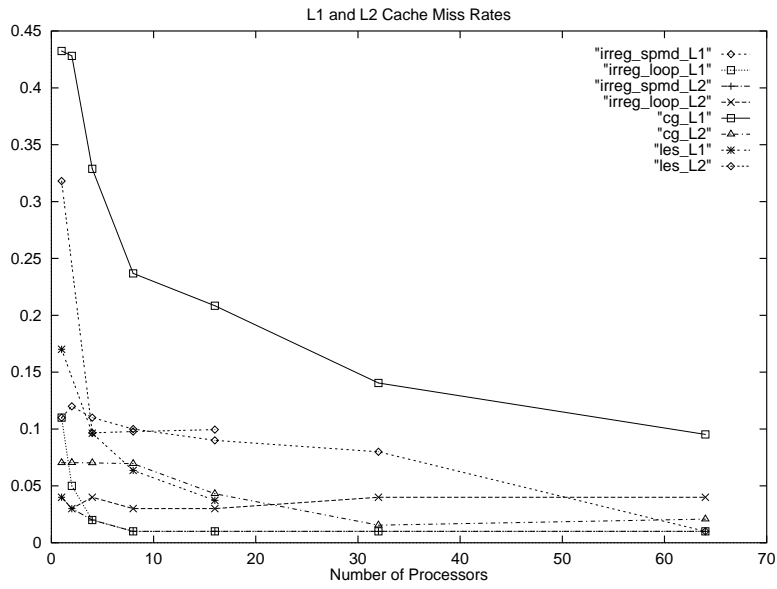


Figure 2: Cache Miss Rates of IRREG, LES and CG

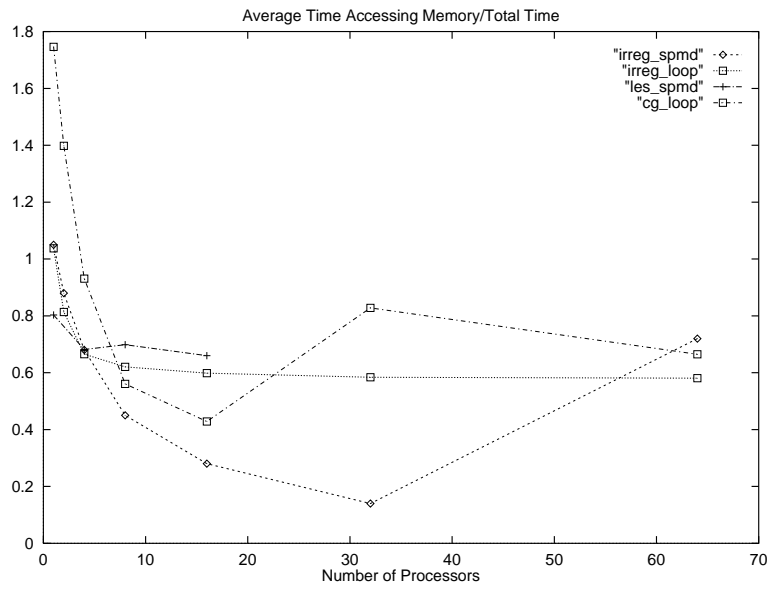


Figure 3: Memory Access Time of IRREG, LES and CG

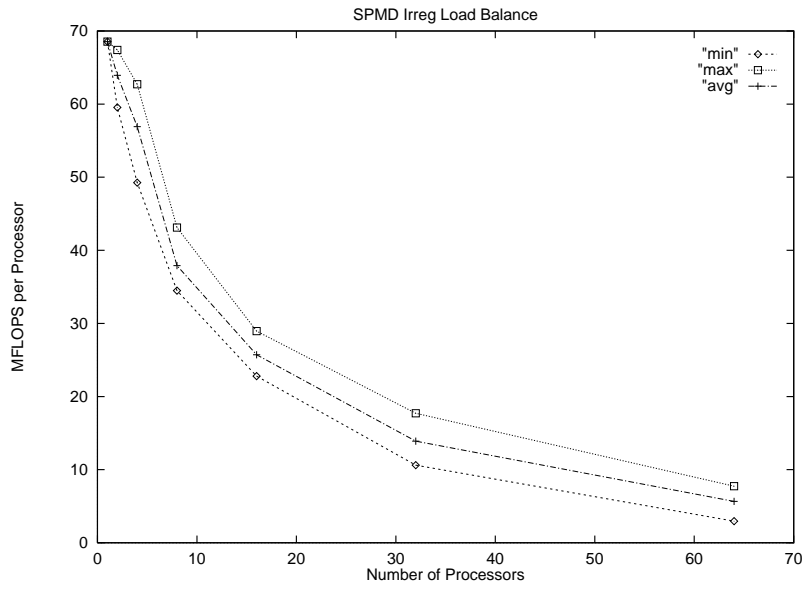


Figure 4: Load Balance of SPMD IRREG

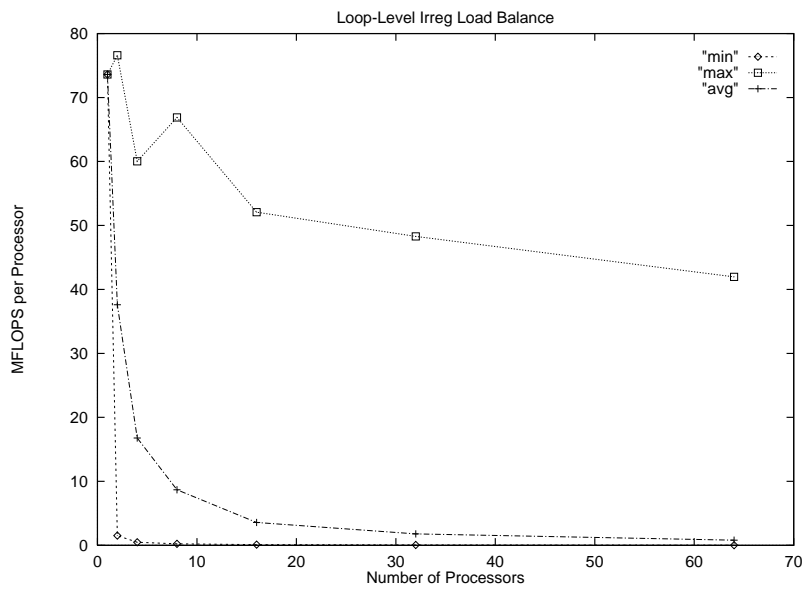


Figure 5: Load Balance of Loop-Level IRREG

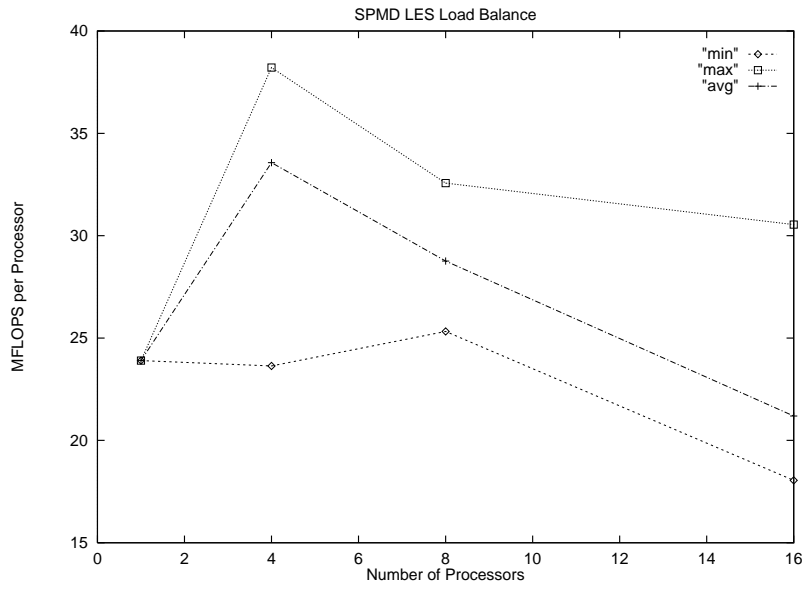


Figure 6: Load Balance of SPMD LES

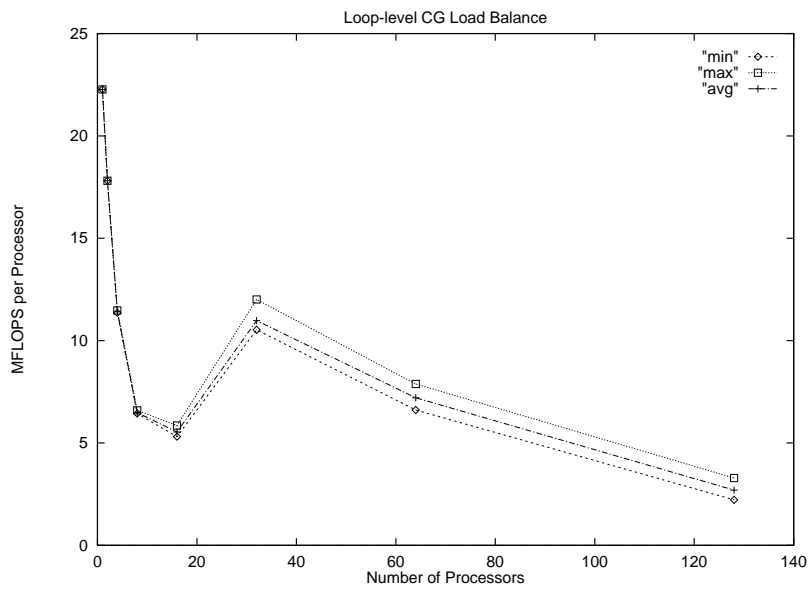


Figure 7: Load Balance of Loop-Level CG

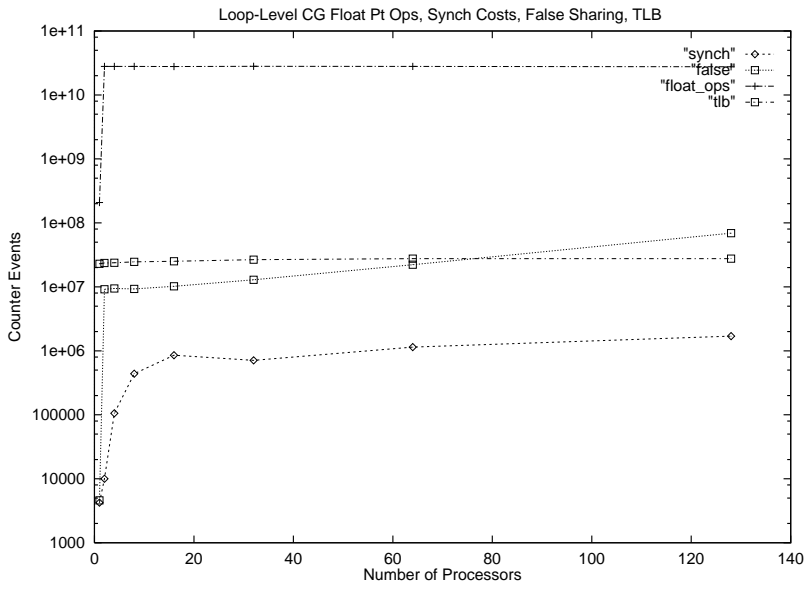


Figure 8: Relative Event Counts for Loop-Level CG

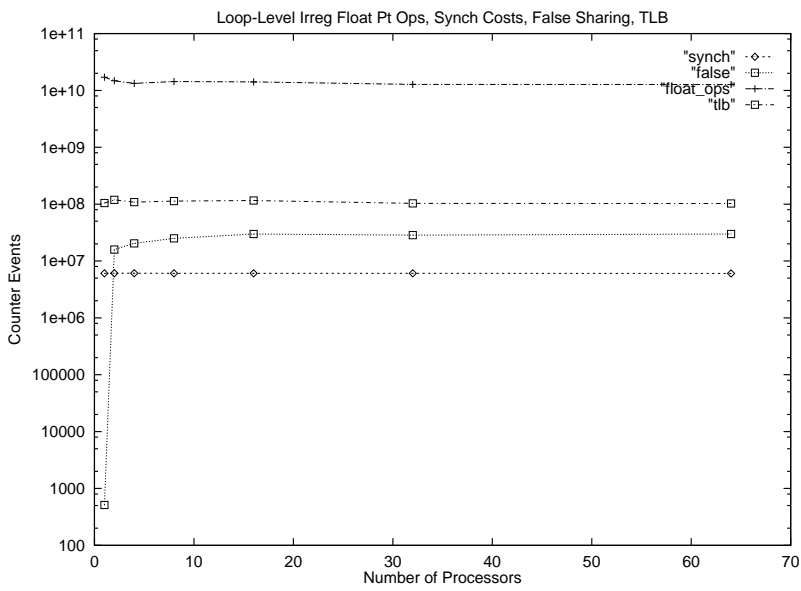


Figure 9: Relative Event Counts for Loop-Level IRREG

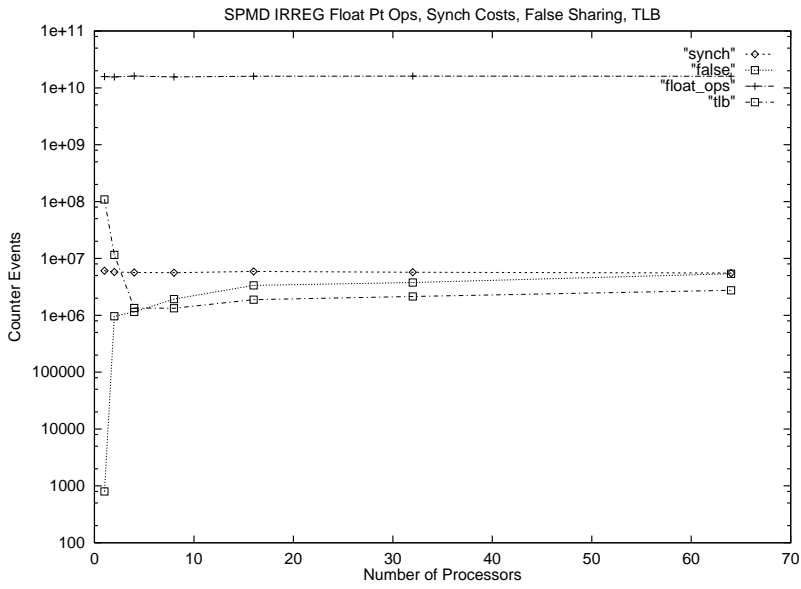


Figure 10: Relative Event Counts for SPMD IRREG

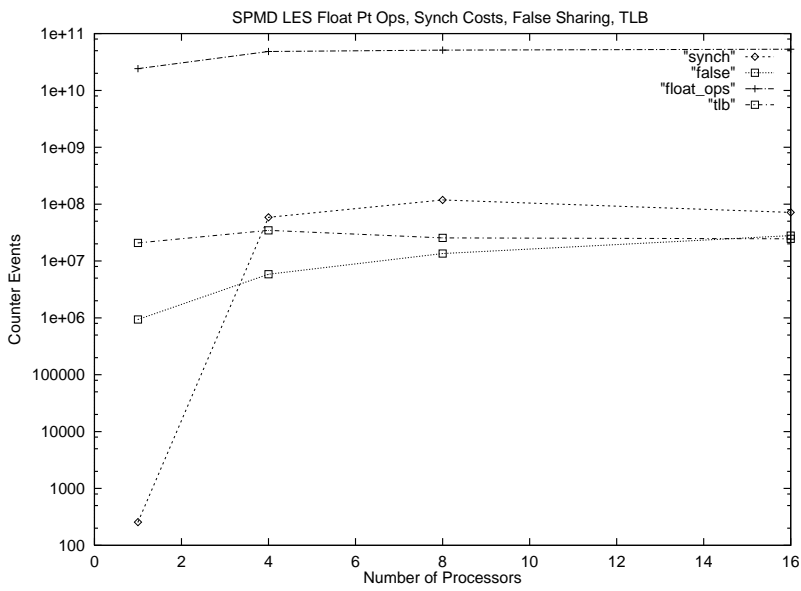


Figure 11: Relative Event Counts for SPMD LES

trends of an increase in time accessing memory above 32 processors and 16 processors, respectively. Since we know the codes are cache-friendly at these number of processors and above, we can deduce that the off-processor remote memory accesses are nevertheless starting to represent a larger portion of the overall time compared to the time spent on computational work.

The load imbalance issue was examined by looking at the number of floating point operations performed over different processors in each run. A load balanced program will have very similar numbers for the number of floating point operations performed across all processors. In Figures 4, 5, 6 and 7, we show the minimum, maximum, and average number of cycles spent on floating point operations across all processors for the OpenMP versions of IRREG, LES, and CG.

A comparison of the relative magnitudes of floating point operations, synchronization costs, false sharing and TLB misses is shown for each of our programs in Figures 8, 9, 10 and 11.

5 Conclusions

In this paper, we have conducted experiments to study the performance achieved through SPMD and loop-level shared memory OpenMP implementations for 3 irregular benchmark programs with realistic problem sizes, run on an Origin 2000. Moreover, we analyzed hardware profiling data to understand the reasons for imperfect speedups of these codes.

Our experiments lead to several interesting observations. First and foremost, the paradigm which leads to good performance for irregular codes is the SPMD style of OpenMP programming. The loop-level technique produces poorly parallelized code. Only the outer loops were parallelized, and on large configurations, not all processors could be kept busy. Particularly for irregular codes, data dependences often prohibited the parallelization of work-intensive loops.

The main factor behind the limited scalability of the SPMD OpenMP versions was load imbalance. False sharing and synchronization costs were insignificant for the programs in our benchmark set because the data decomposition is very good. Another potentially important performance obstacle for OpenMP versions is the cost of non-local references.

In our experience in developing parallel versions of two irregular CFD codes, we found that the SPMD style parallelization facility of OpenMP enables efficient parallelization of these applications with a level of programming effort that is less than or comparable to that required with MPI. One possibility that requires future exploration for obtaining good OpenMP speedup is to take a good MPI parallel code and replace the message passing with shared data structures and OpenMP synchronization.

From this study, we can conclude that programmers need to concentrate on achieving good work distribution while optimizing the performance of OpenMP versions.

Acknowledgments: We would like to acknowledge Haoqiang Jin and Jerry Yan, who generously shared the beta version of the updated CG NAS benchmark with us.

References

- [ACD⁺96] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.
- [BHS⁺20] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. NAS Parallel Benchmarks 2.0. Technical report, Nasa Ames Research Center, NAS-95-020.
- [DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computational Science and Engineering*, 5(1), 1998.
- [DMS⁺94] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives. *AIAA Journal*, 32(3):489–496, March 1994.

- [For94] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3-4), 1994.
- [KLS⁺94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.
- [SB18] Subhash Saini and David H. Bailey. NAS Parallel Benchmark (Version 1.0) Results 11-96. Technical report, Nasa Ames Research Center, NAS-96-018.
- [SWY10] William Saphir, Alex Woo, and Maurice Yarrow. NAS Parallel Benchmarks 2.1 Results. Technical report, Nasa Ames Research Center, NAS-96-010.
- [Wan95] W. P. Wang. *Coupled compressible and incompressible finite volume formulations of the large eddy simulation of turbulent flows with and without heat transfer*. PhD thesis, Iowa State University, 1995.
- [WY10] Abdul Waheed and Jerry Yan. Parallelization of NAS Benchmarks for Shared Memory Multiprocessors. Technical report, Nasa Ames Research Center, NAS-98-010.