Porting and Performance Evaluation of Irregular Codes using OpenMP [1]

Dixie Hisley*, Gagan Agrawal†, Punyam Satya-narayana* and Lori Pollock†

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

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

Abstract

In the last two years, OpenMP has been gaining popularity as a standard for developing portable shared memory parallel programs. With the improvements in centralized shared memory technologies and the emergence of distributed shared memory (DSM) architectures, several medium-to-large physical and logical shared memory configurations are now available. Thus, OpenMP stands to be a promising medium for developing scalable and portable parallel programs.

In this paper, we focus on evaluating the suitability of OpenMP for developing scalable and portable irregular applications. We examine the programming paradigms supported by OpenMP that are suitable for this important class of applications, the performance and scalability achieved with these applications, the achieved locality and uniprocessor cache performance and the factors behind imperfect scalability. We have used two irregular applications and one NAS irregular code as the benchmarks for our study. Our experiments have been conducted on a 64 processor SGI Origin 2000.

Our experiments show that reasonably good scalability is possible using OpenMP if careful attention is paid to locality and load balancing issues. Particularly, using the Single Program Multiple Data (SPMD) paradigm for programming is a significant win over just using loop parallelization directives. As expected, the cost of remote accesses is the major factor behind imperfect speedups of SPMD OpenMP programs.

1 Introduction

Several portable mechanisms for developing parallel programs have been standardized over the last couple of years. 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.

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.

Within the last 4-5 years, two major technological developments have facilitated shared memory programming on medium and large parallel configurations. First, there have been significant advances in the bus and coherence technologies for centralized shared memory architectures. As a result, reasonably large centralized shared memory configurations are now available; systems with up to 32 processors are very common. Second, hardware distributed shared memory architectures have started emerging. These systems implement a decentralized coherence protocol in hardware, and allow a logical shared memory view to the programmers.

[1] Author Agrawal was supported in part by NSF CAREER award 9733520. Author Pollock was partially supported by NSF under grant EIA-9870370.

Distributed shared memory architectures are scalable to very large configurations, e.g., Origin 2000 from SGI is available with up to 1024 processors.

With these developments, OpenMP stands to be a promising medium for developing scalable and portable parallel programs. As compared to MPI, the main benefit associated with OpenMP is that the programmers do not need to explicitly insert communication while parallelizing their programs. However, the principal drawback associated with shared memory programming on large configurations is the lack of efficiency for remote accesses. Though there is a logical view of shared memory, not all the portions of the memory can be accessed with the same costs.

Both the loop parallelization directives and the SPMD style programming directives of OpenMP allow natural expression of regular computations. However, it is not very clear how irregular computations should be expressed using OpenMP. In this paper, we present an evaluation of the suitability of OpenMP for developing scalable and portable irregular applications. In particular, we focus on:

- Which programming paradigms within OpenMP are suitable for expressing irregular computations ?
- What level of performance and scalability can be achieved with the use of OpenMP for porting irregular applications ?
- What is the locality and uniprocessor cache performance achieved ?
- What are the factors behind imperfect scalability ? What are the factors that the programmers should pay particular attention to in trying to achieve good scalability ?

We have used three irregular benchmark programs for our study:

- IRREG, which is abstracted from a computational fluid dynamics (CFD) application that uses unstructured meshes to model a physical problem.
- LES, which is a CFD code used to compute the large eddy simulation of turbulent flow.
- CG, which is a NAS kernel benchmark that solves an unstructured sparse linear system, typical of unstructured mesh applications, by the conjugate gradient method.

Our first development efforts involved using automatic parallelization technology from SGI's compilers. However, the compiler does not recognize and process the irregular reductions prevalent in these codes. We next hand-inserted OpenMP directives to parallelize the loops. OpenMP loop parallelization directives leave the work and data distribution to the compiler and/or the runtime system. While this is a simple approach from a programming point of view, we found that poor locality and load imbalance result for irregular codes using the compiler and runtime system's support for work and data distribution. The performance achieved using loop parallelization directives ranged from no speedups for IRREG to modest speedups for CG.

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 subdomains. 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 reasonably good scalability. For example, speedup by a factor of 30 was achieved on 32 processors for IRREG.

We have also carefully studied uniprocessor and parallel performance of the different applications used in our study. As expected, the cost of remote memory accesses is the major factor behind imperfect scalability of SPMD programs. If careful domain decomposition is not performed, load imbalance can also be a significant factor. False sharing and synchronization turn out to be insignificant for the shared memory programs in our benchmark set.

The rest of the paper is organized as follows. In Section 2, we explain the programming environment 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. We compare our work with related work in Section 5 and conclude in Section 6.

2 Programming Environment and Applications

Our experiments were performed on a 64-processor Origin 2000. We first describe the Origin 2000 architecture and the programming tools we utilized on the Origin, and then give an overview of the benchmarks.

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. 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. Each processor has 1024 MB of local memory.

We believe that the Origin 2000 is an important and interesting architecture to target for this study, because it supports shared memory programming on a large number of processors. Moreover, because it is a distributed shared memory machine, the effect of locality is more significant on the performance, as compared to centralized shared memory machines.

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 *perfex*, 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.

2.2 Parallel Programming Environments

On the Origin 2000, OpenMP can be used through three different options:

- Using the compiler directive `-pfa` and relying on the ability of the parallelizing compiler to automatically extract parallelism from loops and insert loop parallelization directives of OpenMP.
- 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.

We performed some initial experiments using the compiler's automatic parallelization technology, but the results were generally 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. All reported results are from using the MipsPro 7.2.1 compiler and f90 with -O2 optimizations.

2.3 Benchmarks and Problem Sizes

We have chosen three irregular codes, two applications called IRREG and LES, and CG, which is an irregular kernel available in the NAS Parallel Benchmarks (NPB).

Our first benchmark is IRREG[DMS+94], which 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. We ran IRREG on a realistic submarine mesh, in which the number of nodes, edges, and faces were 92,564, 623,003, and 504,947, respectively.

The second benchmark we have studied is an application code which performs a 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

```
do icyc=1, numTimeSteps
  call Sweep()
enddo


! code segment from Sweep
!$OMP PARALLEL DO (i, n1,n2,r)
do i = 1, num_edges
  n1 = edge(1,i)
  n2 = edge(2,i)
  r = konst * (velocity(1,n1)- velocity(1,n2))
  buf(n1,proc_id) = buf(n1,proc_id) - r
  buf(n2,proc_id) = buf(n2,proc_id) - r
enddo
!$OMP END PARALLEL DO
```

```
!$OMP PARALLEL PRIVATE(node_begin, node_end,
!$OMP edge_begin, edge_end, tstart,tstop)
!$OMP MASTER
  nprocs = omp_get_num_threads()
  iprocs = omp_get_num_procs()
!$OMP END MASTER
!$OMP BARRIER
  if (proc_id .eq. 0) then
    call coordinate_bisection(...)
  endif
!$OMP BARRIER
  do icyc=1, numTimeSteps
    call Sweep()
  enddo
!$OMP END PARALLEL

! code segment from Sweep
do i = edge_begin, edge_end
  n1 = edge(1,i)
  n2 = edge(2,i)
  r = konst * (velocity(1,n1)- velocity(1,n2))
  buf(n1,proc_id) = buf(n1,proc_id) - r
  buf(n2,proc_id) = buf(n2,proc_id) - r
enddo
```

Figure 1: Code Segments from Loop Level (left) and SPMD Versions (right) of IRREG

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.

Our third application is CG, which is taken from the NAS Parallel Benchmark set. 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 set mimics 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.

3 OpenMP Implementation of Irregular CFD codes

In this section, we describe our approaches to parallelizing the two non-NAS, irregular CFD applications, LES and IRREG, using first the loop-level style and then the SPMD style of parallelism supported by OpenMP on the Origin 2000.

A loop-level versions of IRREG was developed by inserting parallel directives around work-intensive loops. Figure 1 shows an example of parallelization of a loop in the IRREG application. In this example, the `velocity` and `buf` arrays are indexed indirectly through the values in the `edge` array. `velocity` is only referenced in the loop and not defined anywhere in the loop. The `buf` array is replicated on all processors and is initialized to zero. Since the loops involved only perform increments on the `buf` array, the local increments are stored on all processors. A later loop (not shown here) performs element by element addition of the arrays `buf` on all

processors to obtain the correct final array.

For the parallel loops, no transformations were performed to reorder the data accesses for better locality. The data distribution is performed at run-time based on the first-touch data distribution policy that is the default on the Origin 2000; the first-touch policy may not distribute the data in a satisfactory fashion for the entire program.

The SPMD style of OpenMP parallel programming relies heavily on domain decomposition, similar to message passing programming. Message passing is replaced by shared data that can be read by any thread. Synchronization of writes to shared data is required. While domain decomposition can result in a coarse grain program that exhibits good scalability, the decomposition is now the programmer's responsibility, making the programming task more difficult. Once the problem domain is decomposed, the same sequential algorithm is executed by the multiple processes, but is modified to handle the multiple subdomains. The program is replicated on each thread, but has different extents for the subdomains. Also, data that is local to a subdomain (no need to share globally) is specified as private or thread-private. Thread-private is used for subdomain data that need file scope or are used in common blocks.

In Figure 1, the major components of the OpenMP SPMD version of IRREG are shown, with irrelevant code details elided. The Recursive Coordinate Bisection (RCB) partitioner is used in the code; it attempts to reduce the amount of communication and to load balance the computational work. Additionally, once the data has been assigned to the processors, it is regrouped for better spatial data locality. The SPMD program was created by surrounding the entire program by a parallel region. Inside the parallel region, code segments to be executed by the manager process are surrounded by the `MASTER` directives, or embedded within special conditionals of the form `if (proc_id .eq. 0) then`. The manager process calls the partitioner and then rearranges the data as described above. Then, all processes execute the computational segments of the code on the portions of the data they own. The code segment from the procedure `Sweep` demonstrates the kinds of loop structures executed by each process in order to perform computations over their specific subdomain.

For an SPMD version of LES, the computational kernel, which is embedded in a sequential time-step loop, is placed in a parallel region. Within this parallel region, the computational domain is divided into blocks in one of the three dimensions. This simple block partitioning scheme allows easy SPMD parallelization of LES. The performance improvements possible from more advanced partitioning schemes need to be examined in the future.

4 Experimental Results

We now present the detailed experimental results from our OpenMP implementations of the three irregular codes on an Origin 2000. As we described in the introduction, we had the following goals while performing our experiments:

- Study which programming paradigms supported by OpenMP lead to good scalable performance for irregular codes.
- Study the best performance and scalability achieved for each of the codes.
- Examine the locality and uniprocessor cache performance achieved for each code.
- Study the factors behind less than perfect speedup.

We have focussed on both loop-level and SPMD versions of IRREG, the SPMD version of LES and loop-level version of CG for reporting results in this section. The loop-level version of IRREG produced no speedups, therefore, we did not develop a loop-level version of LES, which is a more complicated code. On the other hand, modest speedups were achieved from the loop-level version of CG, and therefore, we did not develop an SPMD version.

On a single processor, the memory requirements of IRREG, LES and CG were 1,413 MB, 371 MB and 483 MB respectively. Recall that the amount of local memory available on each processor is 1024 MB. Therefore, for any number of processors, the data for LES and CG fit into the main memory. For IRREG, data did not fit into memory on a single processor, but did fit into memory on two or more processors.

We report on the scalability of the four programs in the first subsection. Locality and uniprocessor cache performance is examined in the next subsection. Detailed examination of the factors behind imperfect speedups is the topic of the last subsection.

4.1 Scalability of Irregular Codes

The scalability of the OpenMP versions of IRREG, LES and CG is shown in Figure 2. A linear speedup curve is also included for comparison.

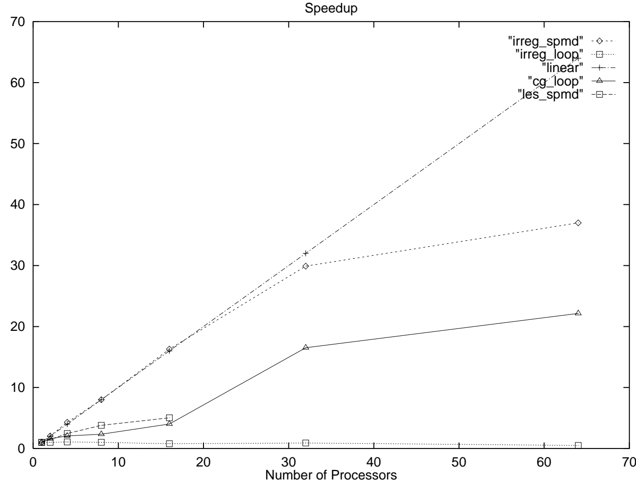


Figure 2: Speedup of OpenMP version of IRREG, LES and CG

We see that the SPMD version of IRREG, up to 32 processors, scales very 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. LES did not scale beyond 8 processors, therefore, no data points are included for LES above 16 processors. A speedup of 5 is obtained on 16 processors. In LES, the matrix solver, the most expensive module, is optimized for locality. Still, two more modules remain poorly optimized due to inherent data dependencies; this factor contributes to the imperfect scaling observed for 16 processors. The loop-level version of CG achieves a speedup of 22 on 64 processors.

4.2 Locality and Uniprocessor Cache Performance

One important factor in performance of any computationally intensive code is the locality. Locality manifests itself on a uniprocessor in terms of cache miss rates. On multiprocessors, besides cache performance on each processor, the frequency of remote references plays a very critical role.

In this subsection, we report on the uniprocessor cache performance and the overall time spent on memory references. Uniprocessor cache performance is measured as the ratios of cache misses for both the level one (L1) and level 2 (L2) cache. The L1 and L2 cache miss rates for the four programs are presented in Figure 3.

In general, a reasonably good level of uniprocessor cache performance is seen for all programs at four processors and above. Although the L1 cache miss rate is as high as .33 for CG for four processors, the L2 miss rate at four processors and above for CG is less than .07. Since L2 cache miss penalty is significantly higher than L1 cache miss penalty, the overall cache performance for CG is quite reasonable. The L1 and L2 miss rates for IRREG and LES are consistently lower than .11 for more than four processors. In all cases, the cache miss rates decrease with an increasing number of processors. This is because each processor is performing computations on a smaller data set, as the same problem size gets divided between all available processors.

Perfex also reports on the average time spent accessing memory, which is computed as a fraction of the time spent on floating point operations. This metric is an important indicator of the level of locality achieved by a program. The time spent on accessing memory includes the time spent on accessing local memory after L1 and L2 cache misses, as well as the time spent on accessing remote data after a miss from the Table Look-ahead Buffer (TLB).

The average time spent accessing memory as a fraction of the time spent on floating point operations is shown in Figure 4. In the 1 and 2 processor cases, the fraction is quite high for all the four programs. This, we believe, is because the data sets are quite large for the 1 and 2 processor cases, and therefore, poor cache

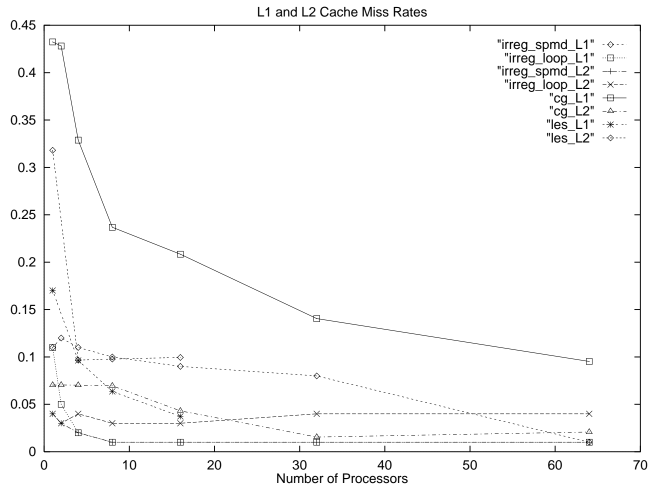


Figure 3: Cache Miss Rates of IRREG, LES and CG

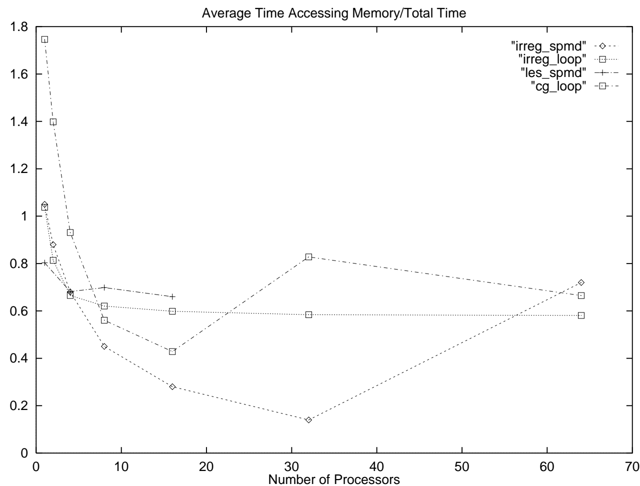


Figure 4: Memory Access Time of IRREG, LES and CG

locality is observed. On the 4 processor case, the fraction is approximately 0.9 for CG and close to 0.7 for all the other three programs. Interesting trends are observed beyond 4 processors for these 4 programs. For the SPMD version of IRREG, where careful domain decomposition has been performed, the fraction reduces to 0.42, 0.24 and 0.16 for 8, 16 and 32 processors, respectively. This shows that the domain decomposition and the renumbering strategy optimizes this code for good cache locality and a very high fraction of local memory references. The fraction increases significantly, to 0.75, for the 64 processor case. This is because each processor's data set has become quite small, and therefore, a larger fraction of non-local references is found. This is consistent with the observation that the code does not scale beyond 32 processors.

A very different trend is shown by the loop-level version of IRREG. The fraction remains constant at around 0.6 between 8 and 64 processors. We believe that this is caused by two factors. First, because the domain decomposition and renumbering has not been performed, uniprocessor locality is lower as compared to the SPMD version. Secondly, because of poor load balance, the amount of data set processed by certain processors is still quite high even when a large number of processors are used. Interestingly, at 64 processors, the SPMD version has a higher fraction of time spent on memory accesses than the loop-level version, while there is a factor 40 difference in their absolute performance. This is because the load balance is very poor on the loop-level version. Some of the processors perform most of the work, while the others stall, but locality is better on the processors which perform most of the work, because of larger data sets.

CG shows the same trends as the SPMD version of IRREG, but the overall fraction is significantly higher.

This is consistent with the higher L1 and L2 cache miss rates. Also, the fraction starts increasing after 16 processors.

LES is similar to the loop-level version of IRREG, in the sense that the fraction remains between 0.8 and 0.6 for all configurations. Considering the fact that it had good uniprocessor cache performance, we can conclude that the fraction of remote references is quite high even on small configurations. This also explains why this code does not scale beyond 8 processors. In our future work, we plan to examine alternative domain decomposition strategies for LES to try to reduce the number of remote references.

4.3 Factors Behind Imperfect Speedups

We have carried out detailed experiments with the goal of understanding the factors behind limited scalability for each of the four programs we have studied. Before presenting the results from this study, we give some background information on the factors behind imperfect speedup for any distributed shared memory program.

In a distributed shared memory architecture, the following factors usually contribute to lower than ideal speedup:

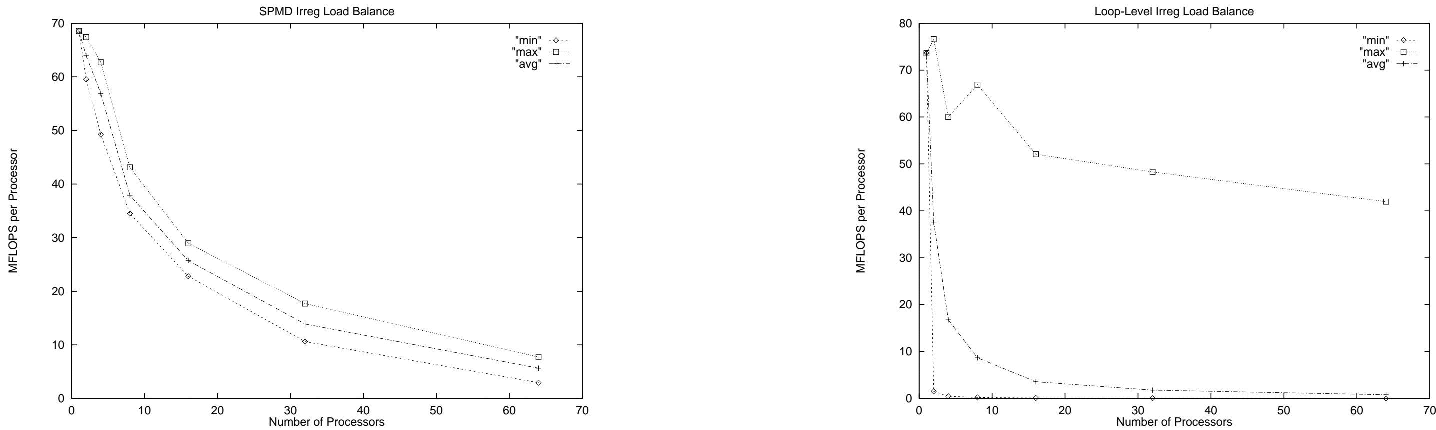
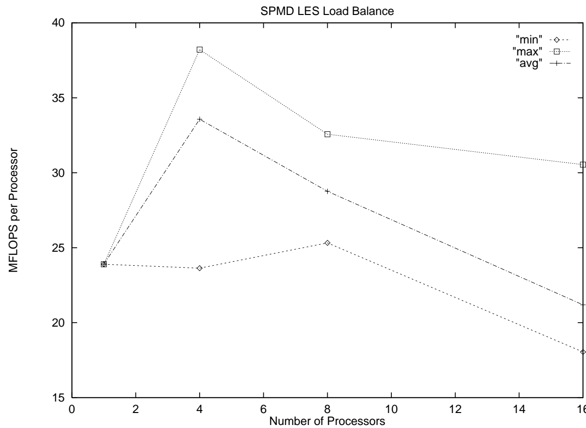


Figure 5: Load Balance of SPMD IRREG (left) and Loop-Level IRREG (right)

- *Load Imbalance*, which implies that the work in parallelized loops is not evenly distributed among the processors.
- *Remote Accesses*, which implies that the memory references are frequently to locations on other processors.
- *Synchronization Costs*, which denotes the time spent by processors in coordinating the progress of the computation among themselves.
- *False Sharing*, which means that two or more processors access different variables that happen to be co-located on the same cache block, with at least one of the accesses being a write. Once the write occurs, the entire cache line is invalidated to other processors. Thus, any attempt by the other processors to use another data item in the cache line will require the entire cache line to be updated first. False sharing is a potential problem in any cache-based, shared memory multiprocessor system.

As we described earlier, the R12000 design in the Origin 2000 provides hardware support for counting the number of cycles devoted to various types of hardware events. These counters can be used to determine which of the factors described above are responsible for the limited speedup. Load imbalance can be measured by determining whether all threads issue a similar number of graduated floating point operations. For applications where there are no references to the disks, the memory access time (as a ratio to the computation time) is an indicative of number of remote accesses. Excessive synchronization costs are determined by examining whether the number of cycles spent on store conditionals is high. If false sharing is the major problem, then the number of cycles spent on stores exclusive to shared blocks will be very high.



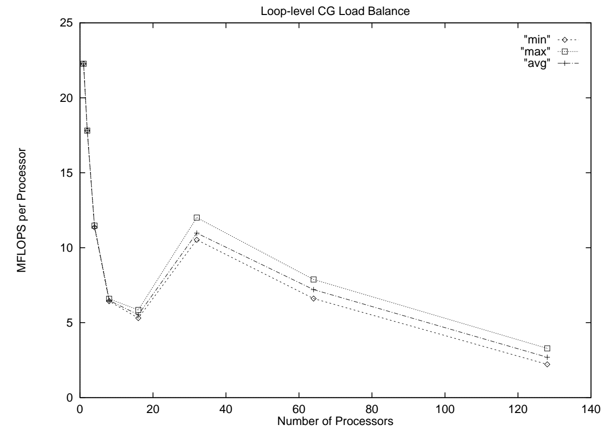
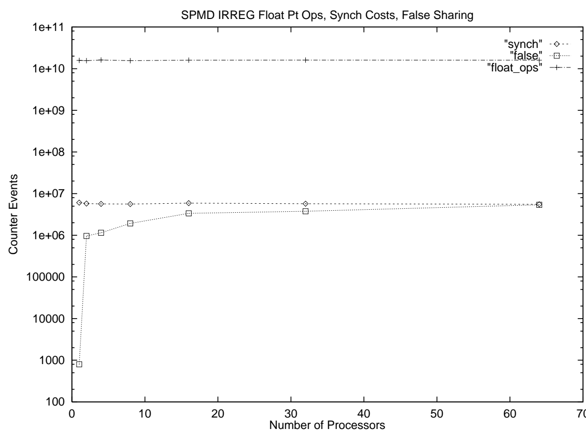


Figure 6: Load Balance of SPMD LES (left) and Loop-Level CG (right)

We now examine to what degree each of the above four factors contributed to imperfect speedups in our benchmarks. In Figures 5 and 6, 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.

From these figures, we see that none of the codes are perfectly load balanced for all runs. Loop-level CG appears to have excellent load balance upto to 16 processors, however the load becomes imbalanced after that. SPMD IRREG also shows good load balance. The difference between the maximum and minimum floating point operations among all processors remains roughly the same beyond 1 processor. The loop-level version of IRREG is severely imbalanced, which is because the compiler and runtime systems are unable to distribute the work evenly among processors. The SPMD version of LES is also load imbalanced. This indicates that more effort is required for performing good domain decomposition for this code.



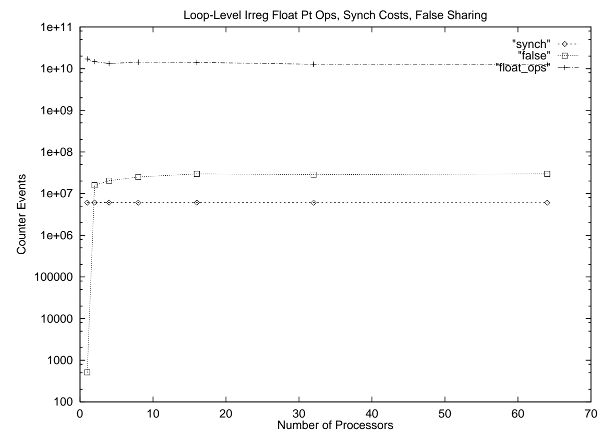


Figure 7: Aggregate Event Counts for SPMD IRREG (left) and Loop-Level IRREG (right)

An interesting trend to note is the absolute MFLOPs per processor for each of the 4 programs, at each configuration. For LES, the MFLOPs per processor initially increases as we go from 1 to 4 processors. This is because of a decreased amount of time spent performing memory operations, which in turn is because computations are performed on a smaller data set at each processor. After 4 processors, this average MFLOPs per processor starts decreasing again. This is due to an increasing number of remote operations. A similar trend is noted for CG.

A comparison of the aggregate counts over all processors of the floating point operations, synchronization costs, and false sharing versus number of processors is shown for each of our programs in Figures 7 and 8. These aggregate numbers represent a sum of each event count for all the processors involved, as reported by perfex.

From these figures, we can observe the trends in relative magnitudes of the event counts. Above four

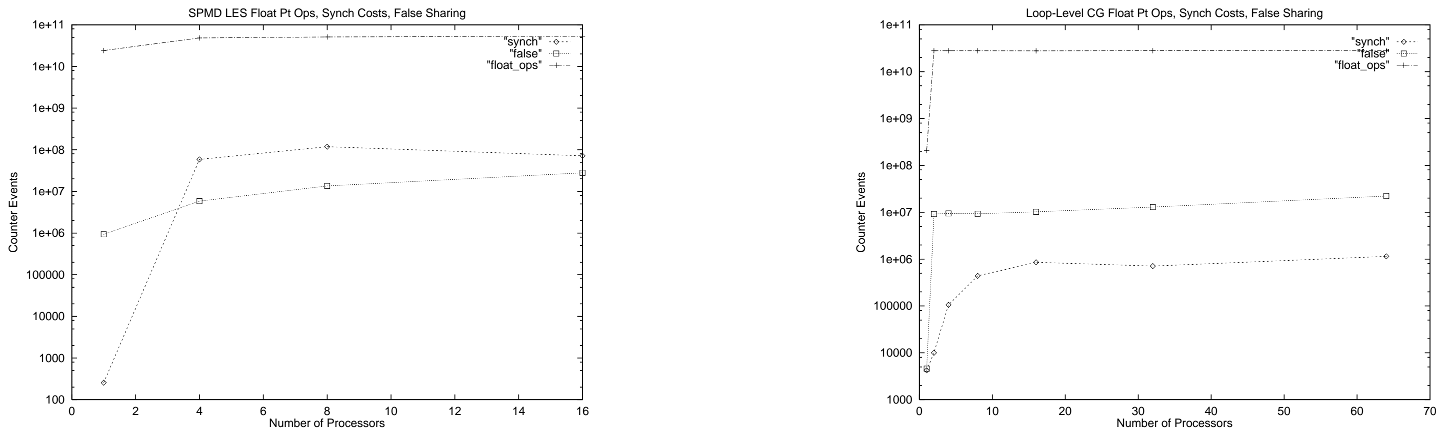


Figure 8: Aggregate Event Counts for SPMD LES (left) and Loop-Level CG (rigth)

processors, the total number of graduated floating point instructions performed remains constant for both paradigms, as expected. However, Figures 5 and 6 for load balance must be examined to determine how the floating point operations are distributed among the processors. Although the total amount of work being performed remains constant, it is not always being distributed uniformly to all the processors, as discussed earlier.

False sharing and synchronization can be two important factors in performance of distributed shared memory programs. We now examine the impact of these two factors on the scalability of our set of benchmarks, from Figures 7 and 8. Besides looking at the counts of synchronization and false sharing operations, we also look at the typical times spent on these factors, as reported by perfex. The count of these operations is at least three orders of magnitude lower than the count of floating point operations.

On 64 processor version of SPMD IRREG, the maximum time any processor spent on stores or prefetches exclusive to a shared block was 5 milli-seconds, while the total execution time was 5.4 seconds. Similarly, the maximum time any processor spent on store conditionals was 5 micro-seconds. This clearly establishes that false sharing or synchronization were not significant factors behind limited scalability of the 64 processor version of SPMD IRREG. On the other hand, the average time spent on accessing memory increased sharply between 32 and 64 processor versions of SPMD IRREG (Figure 4). Thus, the limited scalability of 64 processor version of SPMD IRREG can be attributed to increased number of remote accesses, as each processor operates on smaller data-chunks on larger configurations.

Let us now consider the 16 processor version of LES. Again, the fraction of time spent on memory accesses is quite high. The maximum time spent on stores or prefetches exclusive to a shared block was 7 milli-seconds and the maximum time spent on store conditionals was 34 milli-seconds, while the total execution time was 203 seconds. Again, this establishes that false sharing and synchronization were not significant factors behind limited scalability. Unlike SPMD IRREG, the load imbalance between processors is quite severe for LES, and is obviously a major factor behind limited scalability.

We next consider the 64 processor version of CG. Like SPMD IRREG, the load balance is quite good, though not perfect. The maximum time spent on stores or prefetches exclusive to a shared block was 2 milli-seconds, and the maximum time spent on store conditional was .9 milli-seconds, while the total execution time was 55 seconds. So, the time spent on false sharing and synchronization is not significant.

Overall, we can note the following trends from this detailed profiling data:

- False sharing and synchronization are not significant factors behind limited scalability for these applications.
- If careful domain decomposition is performed, especially using the same techniques that are used in developing message passing programs, load imbalance is again not a significant factor behind limited scalability.
- The cost of remote accesses is uniformly an important factor, especially at large configurations.

5 Related Work

A number of other projects have focussed on developing irregular applications on scalable parallel machines. Most of these efforts have used either a message passing library like the Message Passing Interface (MPI) [For94], a runtime library built on top of the message passing layer or have used a research distributed shared memory machine where the cache coherence protocol can be modified to suit the requirements of the application. We are not aware of any previous study focusing on the development of irregular applications using OpenMP on a commercially available scalable shared memory machine.

Saltz *et al.* developed the notion of *inspector/executor* for efficiently parallelizing irregular applications on message passing machines [SCMB90]. The key idea is to perform runtime analysis for predetermining the communication between processors and aggregating the messages. The notion of inspector/executor was implemented in the PARTI/CHAOS library [BSS91, HMS+95] which was used to implement efficient and scalable message passing versions of several irregular applications like the EULER solver [MDVS92] from which IRREG is derived, and the molecular dynamics code CHARMM [HDS+95].

In some other efforts, compiler techniques were developed for parallelizing irregular applications written in High Performance Fortran (HPF) extensions [AS95, Han93, KM91, PSC93, WDS+95]. However, these techniques have not yet been fully implemented in a production level compiler.

The architecture researchers working on software and hardware distributed shared memory machines have also paid significant attention to irregular applications [ISL96, LCD+97, MSH+95]. The main difference between their efforts and ours is that they could modify the coherence protocols to suit the applications, whereas, this is not a possibility on a commercial hardware distributed shared memory machine like the Origin 2000. Also, they have used custom interfaces for their machines, and not OpenMP. The Treadmarks group at Rice University has recently ported OpenMP onto a cluster of workstations and have performed several application studies on this configuration [CHLZ99, LHZ98]. In comparison, we have focused on a tightly coupled configuration, which we believe, is the main intended target of OpenMP.

Several groups have focussed on porting applications using OpenMP and comparing OpenMP with other mechanisms like MPI and HPF [FHJ+98, NP99]. Our main contributions have been to particularly focus on irregular applications using OpenMP, and also to perform a very detailed analysis of the factors behind imperfect speedups.

6 Conclusions

In this paper, we have focused on evaluating the suitability of using OpenMP for developing scalable shared memory irregular programs. We have used 3 irregular applications and conducted our experiments on a 64 processor Origin 2000.

Our overall conclusion is that reasonably good speedups for irregular codes are possible using OpenMP on Origin 2000 if sufficient attention is paid to load balancing and locality issues. The easiest paradigm for shared memory programs, loop-level parallelism, did not lead to good performance, except for the benchmark code CG. In loop-level parallelism, the programmers just specify parallel loops and shared and private arrays. The data and work distribution is performed by the compiler and/or runtime system. These default mechanisms for data and work distribution did not lead to good load balance and locality for irregular codes.

OpenMP also allows Single Program Multiple Data (SPMD) style parallelism in which the programmers explicitly partition data and work. SPMD style programming lead to excellent scalability for IRREG up to 32 processors. Reasonably good speedups up to 8 processors were also found for LES. We believe that load balancing and locality can be improved for the current implementation of LES to improve the scalability.

Note that unlike loop-level parallelism, significantly higher programming effort is required for developing SPMD OpenMP programs. However, we believe that this effort is still significantly lower than the effort required for developing a message passing version of the same program. In developing message passing programs, the programmers need to correctly insert communication between processors, besides performing data and work partitioning.

In analyzing detailed profiling data, we found that the cost of non-local references and load imbalance are the major factors inhibiting perfect speedups. It is well known that both these factors are crucial in the performance of message passing programs also. False sharing and synchronization costs, which do not arise in message passing programs, turned out to be insignificant for our OpenMP implementations. We believe that this is important

preliminary evidence that OpenMP with SPMD style programming can provide comparable performance with highly optimized message passing versions, but at significantly lower programming effort.

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

[AS95] Gagan Agrawal and Joel Saltz. Interprocedural compilation of irregular applications for distributed memory machines. In *Proceedings Supercomputing '95*. IEEE Computer Society Press, December 1995.

[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.

[BSS91] Harry Berryman, Joel Saltz, and Jeffrey Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.

[CHLZ99] A. L. Cox, Y. C. Hu, H. Lu, and W. Zwaenepoel. OpenMP on Networks of SMPs. *International Parallel Processing Symposium*, April 1999.

[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.

[FHJ+98] Michael Frumkin, Michelle Hribar, Haoqiang Jin, Abdul Waheed, and Jerry Yan. A comparison of automatic parallelization tools/compilers on the SGI Origin2000. *Supercomputing '98*, 1998.

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3-4), 1994.

[Han93] Reinhard v. Hanxleden. Handling irregular problems with Fortran D - a preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993. Also available as CRPC Technical Report CRPC-TR93339-S.

[HDS+95] Yuan-Shin Hwang, Raja Das, Joel H. Saltz, Milan Hodoscek, and Bernard R. Brooks. Parallelizing molecular dynamics programs for distributed memory machines. *IEEE Computational Science & Engineering*, 2(2):18–29, Summer 1995. Also available as University of Maryland Technical Report CS-TR-3374 and UMIACS-TR-94-125.

[HMS+95] Yuan-Shin Hwang, Bongki Moon, Shamik D. Sharma, Ravi Ponnusamy, Raja Das, and Joel H. Saltz. Runtime and language support for compiling adaptive irregular programs. *Software–Practice and Experience*, 25(6):597–621, June 1995.

[ISL96] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Understanding application performance on shared virtual memory systems. In *Proceedings of the 23nd Annual International Symposium on Computer Architecture*, pages 122–133. ACM Press, May 1996. ACM Computer Architecture News, Vol. 24, No. 2.

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[KM91] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[LCD+97] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and Software Distributed Shared Memory Support for Irregular Applications. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 48–56, June 1997.

[LHZ98] H. Lu, Y.C. Hu, and W. Zwaenepoel. OpenMP on Networks of Workstations. *Supercomputing '98*, October 1998.

[MDVS92] D.J. Mavriplis, R. Das, R.E. Vermeland, and J. Saltz. Implementation of a parallel unstructured Euler solver on shared and distributed memory architectures. In *Proceedings Supercomputing '92*, pages 132–141. IEEE Computer Society Press, November 1992.

[MSH+95] S.S. Mukherjee, S.D. Sharma, M.D. Hill, J.R. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 68–79. ACM Press, July 1995. ACM SIGPLAN Notices, Vol. 30, No. 8.

[NP99] Dimitrios S. Nikolopoulos and Theodore S. Papatheodorou. A Comparison of MPI, SHMEM and Cache-coherent Shared Address Space programming Models on the SGI Origin2000. *International Conference on Supercomputing*, June 1999.

[PSC93] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Proceedings Supercomputing '93*, pages 361–370. IEEE Computer Society Press, November 1993. Also available as University of Maryland Technical Report CS-TR-3055 and UMIACS-TR-93-32.

[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.

[SCMB90] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.

[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 alrge eddy simulation of turbulent flows with and without heat transfer*. PhD thesis, Iowa State University, 1995.

[WDS+95] Janet Wu, Raja Das, Joel Saltz, Harry Berryman, and Seema Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(6):737–753, June 1995.

[WY10] Abdul Waheed and Jerry Yan. Parallelization of NAS Benchmarks for Shared Memory Multiprocessors. Technical report, Nasa Ames Research Center, NAS-98-010.