Producing Scalable Performance with OpenMP:

Experiments with Two CFD Applications

Jay Hoeflinger, hoefling@uiuc.edu,
Prasad Alavilli, alavilli@uiuc.edu,
Thomas Jackson, tlj@csar.uiuc.edu,
Center for Simulation of Advanced Rockets
University of Illinois at Urbana-Champaign;
Bob Kuhn, kuhn@kai.com, Kuck & Associates, Inc.

Abstract

OpenMP is a relatively new programming paradigm, which can easily deliver good parallel performance for small numbers (<16) of processors. Success with more processors is more difficult to produce. MPI is a relatively mature programming paradigm, and there have been many reports of highly-scalable MPI codes for large numbers (hundreds, even thousands) of processors. In this paper, we explore the causes of poor scalability with OpenMP from two points of view. First, we incrementally transform the loops in a combustion application until we achieve reasonably good parallel scalability, and chronicle the effect of each step. Then, we approach scalability from the other direction by transforming a highly scalable program simulating the core flow of a solid fuel rocket engine (originally written with MPI calls), directly to OpenMP, and report the barriers to scalability that were detected .

The list of incremental transformations includes well-known techniques such as loop interchange and loop fusion, plus new ones which make use of the unique features of OpenMP, such as barrier removal and the use of ordered serial loops. The list of barriers to scalability includes the use of the ALLOCATE statement within a parallel region, as well as the lack of a reduction clause for a PARALLEL region in OpenMP. We conclude with a list of key issues which need to be addressed to make OpenMP a more easily scalable paradigm. Some of these are OpenMP implementation issues; some are language issues.

1. Introduction

OpenMP [1] is a new parallelism model targeted at MIMD parallelism. OpenMP makes use of processors by employing them as threads of control in a shared address space. Directives express the parallelism in the program, which is then implemented by code generated by the compiler. The threads share data by default, but can also have private data. OpenMP is an industry standard on SMP systems and is usually available on distributed memory systems composed of clusters of SMP nodes.

On the other hand, MPI is well established as a high-performance parallel programming model. It usually is used in a Single Program Multiple Data (SPMD) mode. MPI is based on independent processes, which do not share any memory. Parallelism and data transfer in MPI are expressed through subroutine calls. MPI is widely regarded as a scalable parallel programming paradigm because the programming model causes the user to rewrite a serial application all at once into a domain decomposed program, which by its nature has high locality, and whose processors interact very little.

The use of OpenMP is growing. At the University of Illinois several applications are already using OpenMP. Table 1 illustrates some of these applications. Herein, we report on parallelization experiments we conducted with two of these codes: ROCFIRE and ROCFLO. The experiments were carried out on a Silicon Graphics Origin 2000 system, located at the National Computational Science Alliance (NCSA) in Urbana, Illinois. We measured the performance of the applications with several tools: perfex [7], guideview and assure [8,9].

Author	Application
Tom Jackson, John Buckmaster, Jay Hoeflinger	ROCFIRE, a combustion code implemented with incremental parallelism [2]
Prasad Alavilli , Jay Hoeflinger	ROCFLO, a solid rocket core flow code implemented with domain decomposition
Dinshaw Balsara	Adaptive Mesh Refinement [3]
Danesh Tafti	Flow across louver grids [4]
Jon Dantzig, Nik Provatas	Extrusion code [5]
Nahil Sobh, Jay Hoeflinger	Extrusion code [6]

Table 1. Some applications at the University of Illinois using OpenMP.

ROCFIRE solves a subset of the Navier-Stokes equations, including chemical kinetics (i.e. constant flow field plus equations for energy and species). It is the first of several combustion codes which will be used to model the combustion of the solid propellant in a solid rocket motor. ROCFLO solves the full compressible Navier-Stokes equations. Both of these codes are components of an integrated solid-fuel rocket simulation code being developed at the Center for Simulation of Advanced Rockets[10] (CSAR), a Department of Energy Accelerated Strategic Computing Initiative (ASCI) center, located at the University of Illinois at Urbana-Champaign.

1.1 Parallelization methods

We can divide the process used to parallelize ROCFIRE into two phases. There is no clear boundary between these phases but there is a major difference between the processes. The first we call **loop-level parallelization**. There the process is:

1. **Run a profile on the program.**
2. **Find the serial loop which takes the most time.**
3. **Parallelize it.**
4. **Repeat at step 1.**

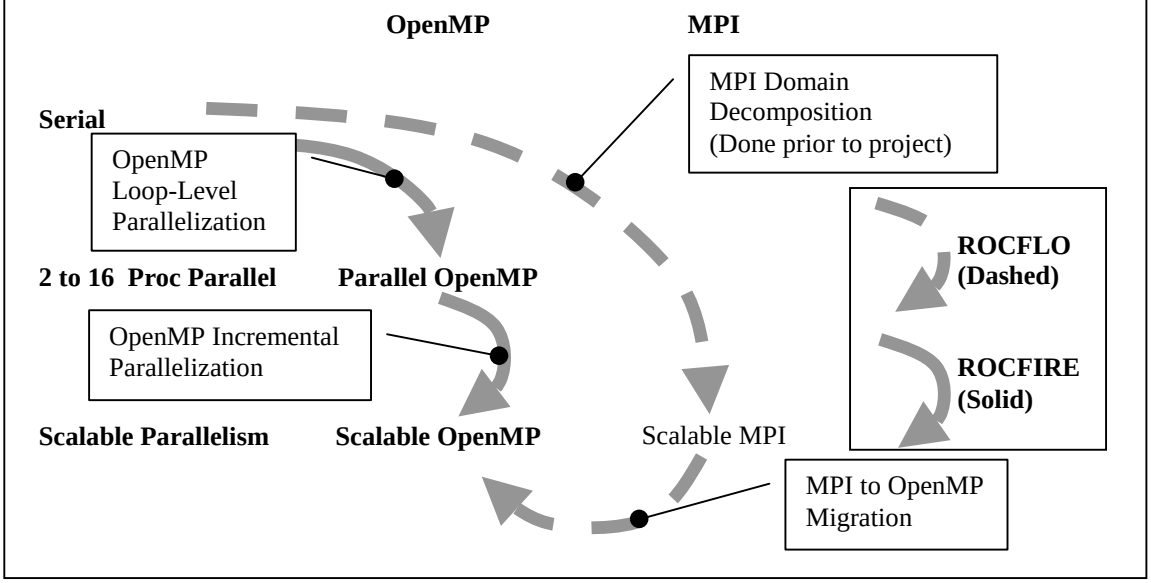
Many have experienced that this process leads to a very frustrating ***point of diminishing returns*** where further changes do not lead to significant further performance increases and the performance achieved does not increase linearly with the number of processors used. Another way of saying this is that the performance does not scale well.

This usually results when the programmer does not consider ***which*** technique could be used to increase performance further. So -- he continues to apply the ***same*** technique, namely parallelize the next most important serial loop. This process is motivated by Amdahl's Law, which leads one to believe that the performance problem is that the remaining serial code is limiting performance. What may be true instead is that already-parallelized parts of the code are hiding inefficient implementation of the parallelism.

We advocate a different approach. The act of selecting the next loop to work on in Step 2 above should be changed to selecting the ***least efficient loop***. Step 3 above must be changed from ***parallelize it*** to ***improve its efficiency***. We call the modified process **incremental parallelization**.

It should be pointed out that the transformation of a given loop may improve the efficiency of other loops in the program. This is possible because the processors in a shared-memory environment can easily affect each other by accessing shared data. For instance, careless scheduling of a single loop can "spoil" the cache, hurting performance of subsequent loops. Likewise, improving the schedule for that single loop can then improve the performance of those subsequent loops.

F igure 1. The two paths used to get scalable OpenMP performance.



In order to get a global improvement, especially on a NUMA machine, it is necessary to keep a global view of the program in mind. This allows the programmer to optimize data locality and minimize interconnect traffic.

Incremental parallelization techniques do not include a major technique that has been used by message passing programs -- domain decomposition. By domain decomposition, we mean *the separation of the total data space into fixed regions of data, i.e. domains, to be worked on separately by each processor*. Within a message passing program, domain decomposition is used as a matter of course, since processors cannot easily share data. This nearly forces the MPI programmer to use local data much more than non-local data, and performance due to that can be quite good. The problem with domain decomposition is the difficulty of coding it. The programmer must (all at once) change all array declarations and all loop bounds, and explicitly code the transfer of data between processors. This is a large and difficult step. We will show an incremental path, using OpenMP, to approximate domain decomposition with a series of simple steps.

In this paper we will describe the techniques used to do incremental parallelization of the serial version of ROCFIRE. We will also show the performance effects of the various techniques. The eventual code scaled fairly well to 64 processors. We will also describe our experiment of directly translating a highly scalable MPI program, ROCFLO, into OpenMP. This method of producing a domain-decomposed OpenMP program allowed us to directly identify constructs of OpenMP that are not scalable, since the MPI counterpart was scalable. The ROCFIRE and ROCFLO experiments are two alternative routes for arriving at a scalable OpenMP program, as shown in Figure 1.

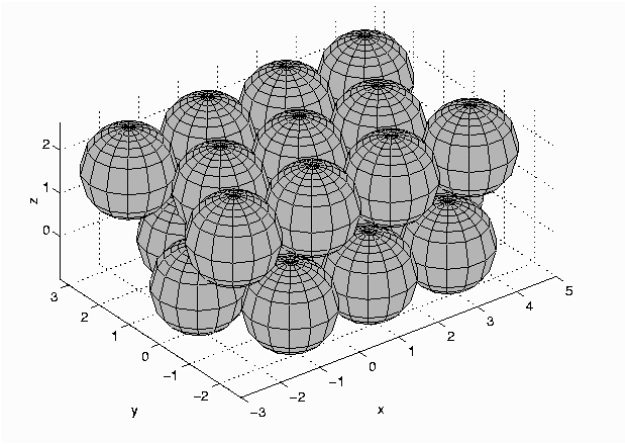
There are at least two ways of conducting scalability experiments: with a *fixed problem* and with *a scaled problem*. By a fixed problem we mean that the amount of data being used for the calculation remains fixed as we add processors. In other words, we try to solve a problem of a certain size by using more and more processors. This means that, as the number of processors increases, the amount of data used by each decreases. By a scaled problem, we mean that the amount of data being used for the calculation increases as we add processors, such that each processor always uses the same amount of data. The experiments reported in this paper are all with a fixed problem, unless a scaled problem is explicitly mentioned.

2. ROCFIRE

ROCFIRE solves a subset of the Navier-Stokes equations, including chemical kinetics, for a typical solid rocket motor (SRM) by the method of lines in the time direction to steady state. A fourth-order central difference scheme is used for spatial discretization, with a uniform mesh in the plane, tangent to the surface, and a non-uniform mesh in the normal direction. A typical grid consists of about one million grid points. The code is about 5000 lines of Fortran 77. It is the world's first 3-D solid propellant combustion simulation code, extending the 2-D work of Buckmaster, Jackson and Yao [13].

The propellant in an SRM consists of a high density packing of ammonium perchlorate (AP) particles imbedded in a fuel-binder, as shown in Figure 2. Typically, aluminum particles are also added, which burn in the gas-phase products of the AP-binder combustion, increasing the rocket chamber temperature. A mix of AP particle sizes is used in the 2-400 micron range, and the primary combustion field is located within tens of microns of the propellant surface. Some of the heat generated within the combustion field is conducted to the propellant surface, and is responsible for the surface regression, the necessary conversion of solid to combustible gases. The surface is not flat, as the instantaneous regression rates of AP and binder are not equal.

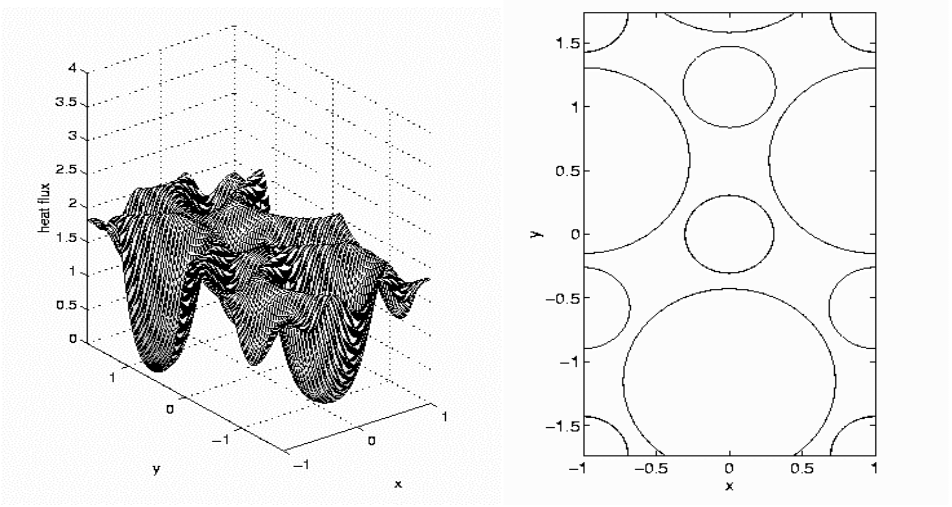
Figure 2. AP Particles in an binder.



As the propellant surface regresses, the surface structure changes. Pieces of AP buried in the binder are first exposed and then consumed. As a consequence, each portion of the combustion field near the surface experiences a chain of ignitions, and what effect these may have on the dynamics appears not to have been addressed in the previous literature. Figure 2 shows an array of AP particles imbedded in a modified binder, and Figure 3 shows the propellant surface (viewed as a slice through the z-plane) and the corresponding heat flux to the surface.

The ROCFIRE simulation is part of the integrated simulation code being built at CSAR. In order to describe the physics of combustion in an SRM

Figure 3. Slice through propellant surface and heat flux on slice.



more accurately, the regression of the surface must generate boundary conditions for the chamber flow. It is necessary to solve for the three-dimensional combustion field, coupling this with physical processes such as heat conduction in a thin surface layer in the propellant, and allowing for feedback from the chamber flow. Hence, the ultimate goal is an unsteady 3-D simulation of propellant burning.

The principles used to guide the incremental parallelization of ROCFIRE are simple:

1. Each processor should consistently access the same part of all shared data, as much as possible.
2. Maximize re-use of data in cache.
3. Fork , join, and barrier should be eliminated as much as possible.

Principle 1 amounts to an approximation of domain decomposition with OpenMP. It can be achieved with small, simple steps in OpenMP. Whatever data must be shared across the domain boundaries is communicated automatically *by the hardware*, making it transparent to the programmer and potentially very fast. Principle 1 can be achieved by:

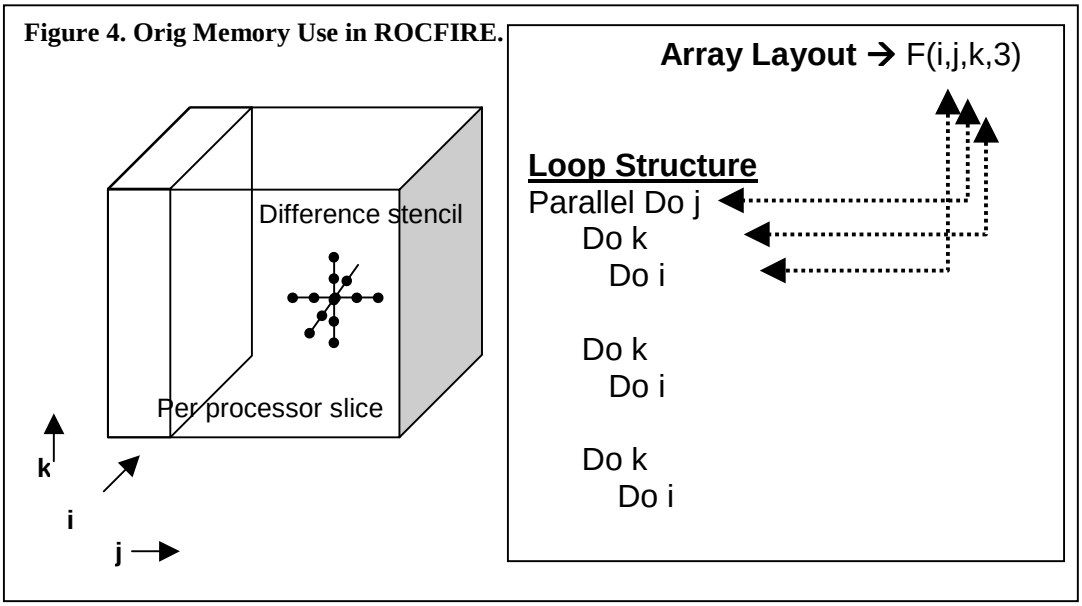
1. **Using first-touch data placement and parallelizing initialization loops.**
2. **Scheduling loops consistently (use loop transformations, e.g. interchange).**
3. **Making iteration spaces consistent.**

The following sections describe the incremental parallelization techniques used in detail**.**

2.1 Memory Usage Analysis

Before going too far in parallelizing a CFD application, the memory usage pattern should be analyzed. CFD applications tend to sweep through memory rapidly creating little opportunity for optimizing cache

Figure 4. Orig Memory Use in ROCFIRE.



and main memory utilization. These can easily become limiting factors in the performance of non-domain decomposed applications, and hence effective parallelization and efficient memory utilization are tightly coupled. On the other hand, finite difference computations can have reasonable memory reuse if the programmer keeps in mind the machine and program characteristics listed in Figure 5.

Principle Program Structures
Loop nest order
Flow field array dimension order
Finite Difference stencil
Dimensions of dataset

Principle Memory System Features
Cacheline size
Lookaside buffer
Cache capacity
Memory bandwidth

Figure 5. Program structures and machine characteristics to keep in mind when optimizing a program with OpenMP.

Our analysis of the ROCFIRE memory usage, illustrated in Figure 4, was as follows:

- The inner loop corresponded to access in *consecutive data elements* so that several memory references would stall until the *cache line* was loaded but then all of the cache line would be used. This is *good*.

- The only sharing of cache lines between processors was in the back/right edge of each processor slice, with the front/left edge of the slice to the right of it. Therefore, *false sharing* was *not* a major problem. This is *good*.

- The size of each processor's *cache* on the Origin 2000 was 4 megabytes, but vertical slabs executed by each processor (due to the *inner loops*), accessed a data volume much larger than that. This is ***not*** *good*. The finite difference stencil reads from adjacent planes offering the possibility of cache re-use, but due to the large data volume of a plane, it is likely that the data from the previous plane will flushed before it can be re-used.

- The size of the dimension in each direction was about 80 and the finite difference stencil requires six vertical planes for reading, and one for writing. When there are 16 processors or more, each processor will have slabs of 5 or fewer planes, so there will be a considerable sharing of pages. This is ***not*** *good*.

To address the problems mentioned, we applied the optimization techniques described in the following sections.

2.2 Consistent Scheduling

Maximum performance improvement proportional to cache latency.
CFD applications frequently have consecutive loop nests that access the large shared data arrays with similar loops and similar loop bounds. In a 3-D flow problem, there are usually three loops in a loop nest for each set of equations and it is best to consistently parallelize access to the same dimension of the shared data structures, so that cache locality can be preserved. There are two situations which can arise: the global flow array fits in the combined caches of all the processors, or it doesn't. This strategy works extremely well in the first case. If the data usage exceeds the collective cache, transformations which limit the data used, but maximize the data re-use in cache should be applied.

Given the goal of consistently parallelizing the same dimension, the question becomes which dimension in a 3-D flow problem to choose. Here are the main considerations:

- Because Fortran arrays are laid out in memory with the first dimension contiguous, poor performance results if that dimension is parallelized.
- When one dimension is less than a small multiple, say 5, of the number of processors you wish to use, choose a different dimension of the array. This rule is motivated by the need for good load balance to achieve an efficient parallel computation.
- Pick the loop dimension that has the most parallel loops. If sweeps in orthogonal directions are used in the application, it is probably the case that some loop will have dependences in each direction. In this case some memory motion is unavoidable.
- Frequently, 3-D flow field arrays have 4 dimensions where the fourth dimension is used to store various properties like velocity or pressure. This dimension is not a good candidate for parallelism. However, when equations couple these properties within a loop nest, placing this dimension first may allow memory bandwidth to be used more efficiently because each cache lines will contain properties that are going to be used together.

`Schedule(static)`, which is the default with OpenMP, will enable consecutive parallel loops to assign the same iterations to each processor, enhancing cache locality.

Serial loops between parallel loops can spoil the cache of one or more processors if left to the default of serial execution on the master thread. Instead, we used parallel loops with an `ORDERED` section around the whole loop body. Such loops get scheduled just like parallel loops, but one processor waits for the previous processor to finish its work, so the execution is actually serial. This tends to preserve the cache of each processor.

Figure 6. Making the iteration consistent with `if( )` statements.

```
!$OMP DO
       do j=-2,ny+2
!! Skip first and last iterations in this loop
       if ((j>-1) .and. (j<(ny+1))) then
        do k=1,nz
         do i=0,nx
          temp1 = ((oldsoln(i,j,k,1)-f(i,j,k,1))**2)
          err1a(id) = err1a(id) + temp1
         end do
        end do
       end if
       end do
```

In addition to having the same loop in the loop nest consistently parallel, the loop bounds should be consistent so that static scheduling assigns the same iterations to the same processor for each loop. An `if` statement containing the body of the loop can be used to extend shortened loops to use different loop bounds, as shown in Figure 6. Through this method, all loops can be made to use the same bounds. This type of transformation could easily be performed by the compiler, leaving the source code simpler. We propose a new clause for the OpenMP Parallel Do directive: `align([lowerbound],[upperbound])`.

2.3 First Touch, NUMA Memory Options and Scalable Runtime

Performance improvement proportional to NUMA latency.
On NUMA systems, it is important that pages of the shared arrays are owned by the processor that accesses them most. For systems which assign a page to the processor which first touches it (the first touch placement policy), the processor that will use the data for computation should execute code that initializes shared arrays. Although, extensions to OpenMP for placement are possible [3], the initialization loops can be either parallelized, or run serially with an ORDERED parallel loop, as mentioned in Section 2.2.

NUMA systems may have several system tuning options which should be investigated. Experimenting with these parameters may improve performance. On some NUMA systems there is an automatic page migration policy which may help or hinder performance. To achieve the most benefit of techniques like consistent scheduling above, the operating system must schedule a thread to the same processor consistently.

It was also crucially important to the success of the ROCFIRE parallelization effort that we use a scalable runtime system. The original OpenMP compiler that we used required a non-scalable startup time for parallel loops, which produced poor performance for the experiments with 16, 32 and 64 processors. When we switched to using the native SGI OpenMP compiler, this problem disappeared.

2.4 Loop Interchange and Fusion

Performance improvement proportional to *join barrier* cost plus parallel loop dispatch time.
Once an array dimension and its corresponding loop have been selected for parallelization, performance improvements can be obtained by moving this loop to the outermost position in a loop nest. This reduces the parallel overhead for entering and exiting a loop.

Once the parallel loop has been interchanged for consecutive loops, it may be possible to reduce overheads further by combining adjacent loops, with no serial code in between, into one loop. This is called loop

fusion. If a small amount of serial code intervenes, it is still often possible to fuse the loops by executing the intervening code redundantly on all processors.

2.5 Parallel Regions

Performance improvement proportional to region barrier cost.

OpenMP supports parallel regions. Inside a parallel region, all threads execute the code. `MASTER` or `SINGLE` thread regions can be used where it is necessary to execute serial code within a parallel region. Parallel loops inside the parallel region are executed in a work-sharing fashion. In ROCFIRE, performance improved when a series of parallel loops were enclosed within a parallel region. This replaced multiple fork/joins with a single fork/join.

2.6 Barrier Removal

Performance improvement proportional to Barrier Time.

Figure 7. Barrier removal by specifying NOWAIT.

```
!$OMP DO
      do j=-2,ny+2
      do i=-2,nx+2
         f(i,j,k,1) = alpha*xf
      end do
      end do
!$OMP END DO NOWAIT
!$OMP DO
      do j=-2,ny+2
      do i=-2,nx+2
         f(i,j,k,1) = f(i,j,k,1)*xf
      end do
      end do
```

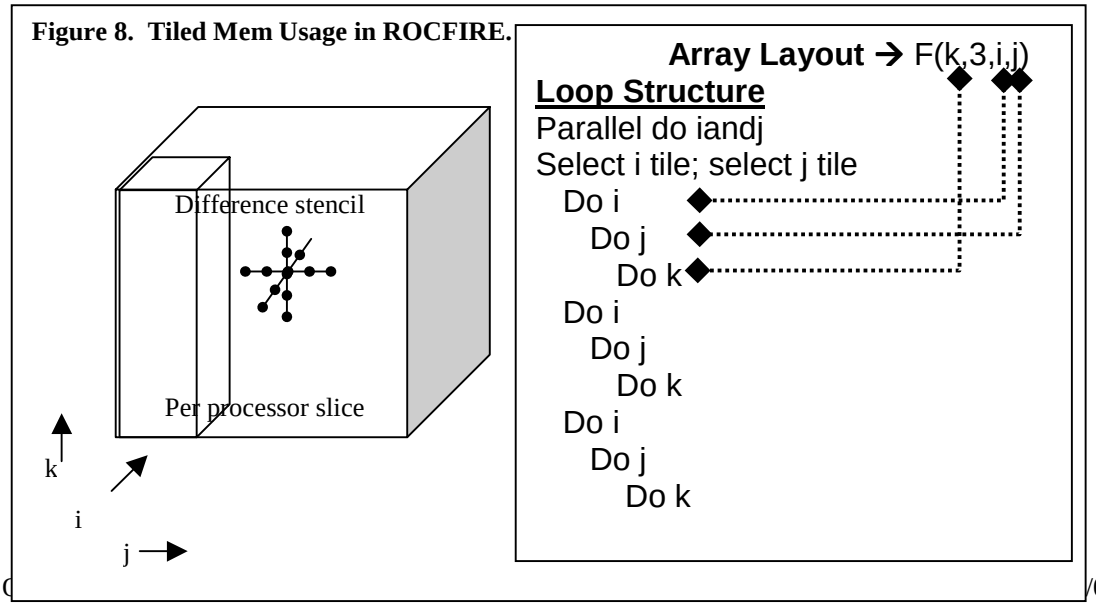
Barriers must be used in the parallel region between sections that compute a value on one thread that will be used on another thread. This avoids the second thread getting to the read before the first thread has written the data. OpenMP places a barrier by default at the end of each parallel loop. At the same time, barriers should be avoided to reduce overhead. A product such as KAI's Assure [8] can be used to determine when barriers can be removed. Sometimes the barrier can be removed by specifying NOWAIT on the ENDDO directive. See the example in Figure 7.

2.7 Loop Collapsing and Tiling

Performance proportional to cache latency.

Because the data volume of a slab in ROCFLO is larger than the cache size, data is driven out of the cache before it can be re-used for the next slab. Therefore, we decided to divide each slab into bricks to enable each processor to reuse data more efficiently. The bricks allowed the finite difference operators to reference data previously loaded from adjacent planes while it was still in cache. At the same time, it was necessary to reorder the dimensions of the flow field arrays for efficient access.

Figure 8. Tiled Mem Usage in ROCFIRE.



In ROCFIRE, loop collapsing and tiling was implemented as follows:

- The i and j loops were picked as the face of the brick because both dimensions could be parallelized.
- Loops were interchanged to bring i and j loops adjacent and to the outermost position.
- A loop was placed outside the i and j loops with the product of the iterations of the original i and j loops.
- The iterations of the new loop were grouped by a set of iterations for the i loop and for the j loop (the tile sizes for i and j). Together, these form a tile.
- The index for the outer loop is translated to starting iterations for i and j loops.

The flow field array dimensions were interchanged to optimize the new loop structure:

- The k index, the depth of the brick, was made the leading dimension to utilize the cache line best.
- The physical parameter dimension, was placed next.
- The i and j dimensions were placed last to complete contiguous bricks of data stored on each processor.

A good brick size was determined experimentally. A brick should fit entirely in cache. Figure 8 illustrates the memory usage after tiling.

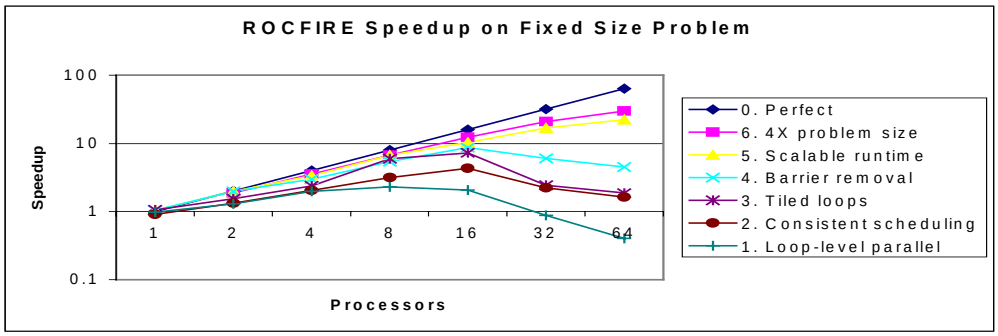
Tiling could be supported directly by OpenMP, relieving the programmer from the complicated tiling transformation. OpenMP allows nested loops to be described as parallel. A useful addition would be a clause to specify the tile size on the parallel loop directive. To our knowledge, there is only ***one*** implementation of OpenMP which truly implements nested parallel loops, the Nanos compiler[12] from the NANOS project in Barcelona. The NANOS group has proposed an OpenMP tiling clause, which we support.

2.8 Incremental OpenMP Parallelization Performance

We applied the incremental parallelization process to ROCFIRE. The results are shown in Figure 9.

1. Loop-level parallelization -- parallelized a loop in each loop nest in ROCFIRE.
2. Consistent scheduling – loop structure and iteration space made the same loop in each loop nest (using loop interchange and fusion). Performance improved and the peak performance shifted from 8 to 16 loops, an indication that scaling was being improved.
3. Loop tiling – Loop tiling and dimension re-ordering was performed. A tile size of 10x10 was used.
4. Barrier removal – Unnecessary barriers were removed from parallel loops and a better tile size was used (11x7).
5. Scalable runtime -- The OpenMP implementation was changed to one with a scalable runtime library. Scalable ROCFIRE performance had been achieved.
6. 4X problem size – the scalable runtime version of the code solving a four times larger problem.

Figure 9. Incremental steps in parallelizing ROCFIRE.



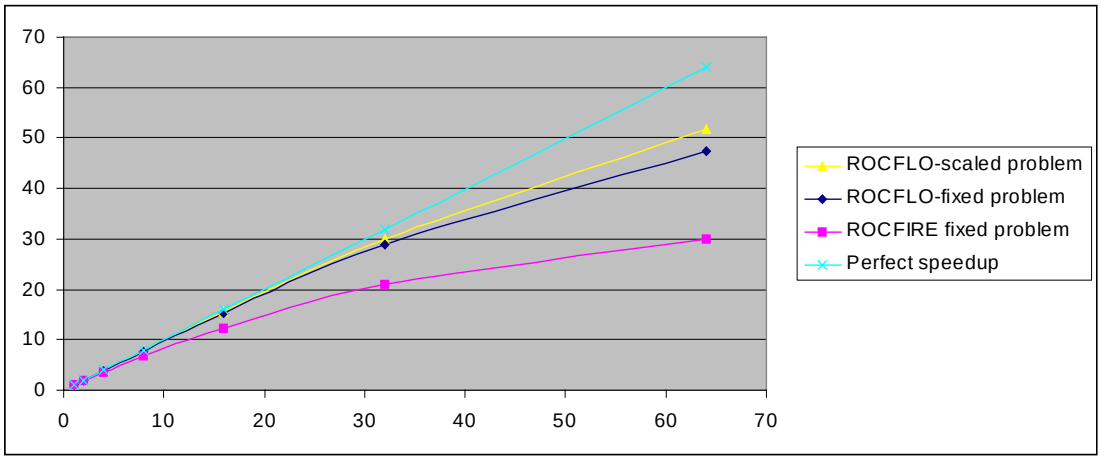


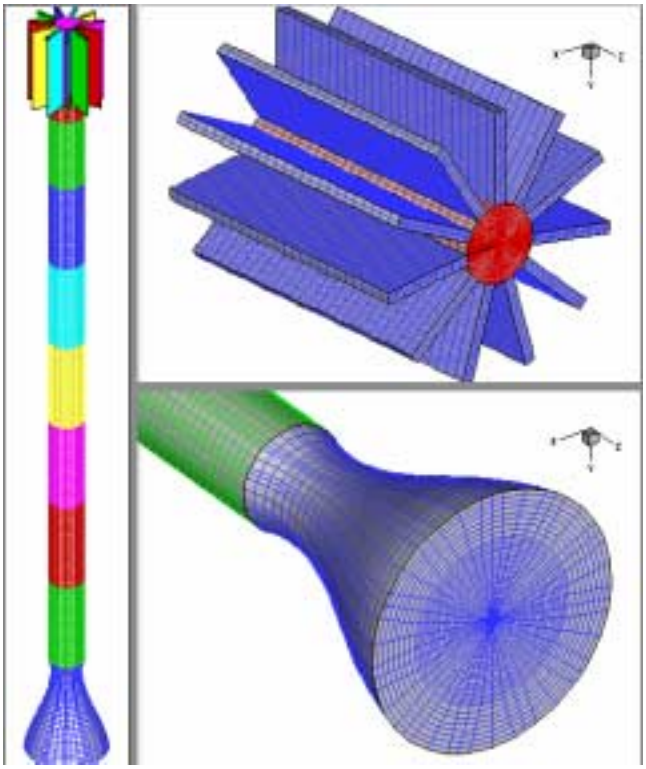
Figure 10. Speedup comparison between MPI ROCFLO and OpenMP ROCFIRE.

The MPI version of the ROCFLO application (discussed further in Section 3.) has been widely recognized as a highly-scaling program, both with a fixed problem and a scaled problem. But how does one recognize good scalability? We take the position, espoused in [14], that good scalability means that the parallel efficiency remains fixed, regardless of the number of processors. In other words, a plot of speedup versus processors should be a straight line. The slope of the line is unimportant. The important thing is consistent gains in speedup as the number of processors is increased. In Figure 10, we compare the speedups of the MPI version of ROCFLO (for scaled and fixed problems) with the speedup of the OpenMP version of ROCFIRE (for the 4X problem size). Even though the slope of the ROCFIRE line is less than the slope of the ROCFLO lines, they are still very similar, in terms of straightness (especially when comparing the fixed problems). From this we conclude that ROCFIRE has come close to matching the scalability of a highly-tuned MPI program.

3. ROCFLO

ROCFLO[11] is a CFD code currently being developed at CSAR, to simulate solid rocket booster core flow dynamics. Rocket flow problems exhibit singular features compared to other flow environments. Primary among them being the fast propagating acoustic pulses in the chamber that could potentially cause instabilities and lead to rocket malfunction.

Figure 11. Redesigned Solid Rocket Motor descretization for ROCFLO.



ROCFLO solves the unsteady, compressible, Navier-Stokes equations on dynamic meshes using a finite volume method. Dynamic meshes are handled through an Arbitrary Lagrangian-Eulerian (ALE) formulation. Second order accurate Central and Upwind TVD spatial discretization schemes have been implemented. An explicit multistage Runge-Kutta algorithm is used for time integration. Unsteady flows are computed using global time stepping, while steady state flows are more efficiently computed using local time stepping for convergence acceleration.

ROCFLO implements the above methods into a very general solver. A 3-D, multi-block structured approach is adopted to resolve the geometrical complexities of such flows. Domain decomposition is utilized for the parallel implementation.

Each computational block has an independent set of flow and simulation parameters allowing p-refinement.

The code is approximately 20,000 lines of Fortran90, using data encapsulation, pointers, and dynamic memory management. It runs on distributed memory parallel architectures using MPI for inter-process communication. Initial load balance in MPI is arranged statically by assigning one or more computational blocks to each of the processors. Each computational block is updated independently after which inter-block boundary conditions are set in a communications phase. Blocks residing on the same processor just copy boundary data while those on different processors communicate using MPI. This communication phase is made to overlap the computations involved in regenerating the interior mesh for the next step following the boundary motion, thus achieving latency hiding. *Persistent messaging* is used to reduce startup overhead in the send/receive calls.

In [11], the solver was applied to the study of the Space shuttle Redesigned Solid Rocket Motor (RSRM) including the 11 point star grain in the fore-end and the aft nozzle. Gross flow parameters agree very well with actual Shuttle flight data. A view of the computational meshes used in the Space Shuttle booster simulations is presented in Figure 11.

3.1 MPI Emulation with OpenMP

OpenMP contains nearly all the mechanisms it needs to emulate an MPI environment, so direct translation of an MPI program to an OpenMP program can be done in a straightforward way. Each MPI operation can be translated in place, macro expansion style, to OpenMP. The translations used for ROCFLO can be divided into 5 groups. The first describes how to set the OpenMP data and execution environment to emulate MPI. The second deals with data allocation not covered by the framework. The third includes the frequently used MPI collective operations. The fourth involves direct message passing. The last, at a higher level than MPI itself, is input/output operations.

3.2 MPI Environment Emulation

All MPI processes execute the same program in parallel. OpenMP programs begin execution by only the master thread, but have the PARALLEL region directive that causes all threads to begin execution. A PARALLEL region made to encompass the whole program effectively emulates the MPI execution behavior.

All data in each MPI process is thread-private. Within OpenMP, data can be either shared or private. All that an OpenMP program needs to emulate the MPI data environment is to specify the DEFAULT(PRIVATE) clause on the PARALLEL region directive enclosing the program.

These concepts were applied to ROCFLO by enclosing the main program with the two OpenMP directives

```
!$OMP PARALLEL DEFAULT(PRIVATE)

!$OMP END PARALLEL
```

The frequently used MPI idiom for master process execution –

```
      if (myid.eq.0) then
          . . .
      endif
```

was replaced with the OpenMP master directive block –

```
!$OMP MASTER
       . . .
!$OMP END MASTER
```

3.3 Data Allocation

Fortran has four forms of persistent data: module data, allocatable and pointer based data, data/save variables, and commons. In this subsection we consider how to make OpenMP treat the first three types in the same way as MPI. In OpenMP 1.0, module data is shared by default. In an MPI program, however, module data is private because all data is private. To emulate MPI, module data in OpenMP must be placed in a thread private common. In ROCFLO, for example, the module data –

```
      INTEGER, PUBLIC, ALLOCATABLE :: F_pndom(:)
      INTEGER, PUBLIC, ALLOCATABLE :: F_pdoms(:,:)
```

was replaced by –

```
      INTEGER, PUBLIC, POINTER :: F_pndom(:)
      INTEGER, PUBLIC, POINTER :: F_pdoms(:,:)
      COMMON /PRIV/ F_pndom, F_pdoms
!$OMP THREADPRIVATE (/PRIV/)
```

Fortran allocatables are similar to pointer-based data in persistence. In OpenMP when multiple threads execute an `allocate`, the threads receive pointers to different physical memory. This is semantically identical to MPI. However, because OpenMP is based on threads, sharing the virtual address space, when the allocation is performed mutual exclusion is used to guarantee that distinct sections of the address space are assigned. We found this to be a scalability limitation in the implementation of OpenMP that we tested.

Figure 13. Poor scaling of OpenMP F90 allocatable arrays.

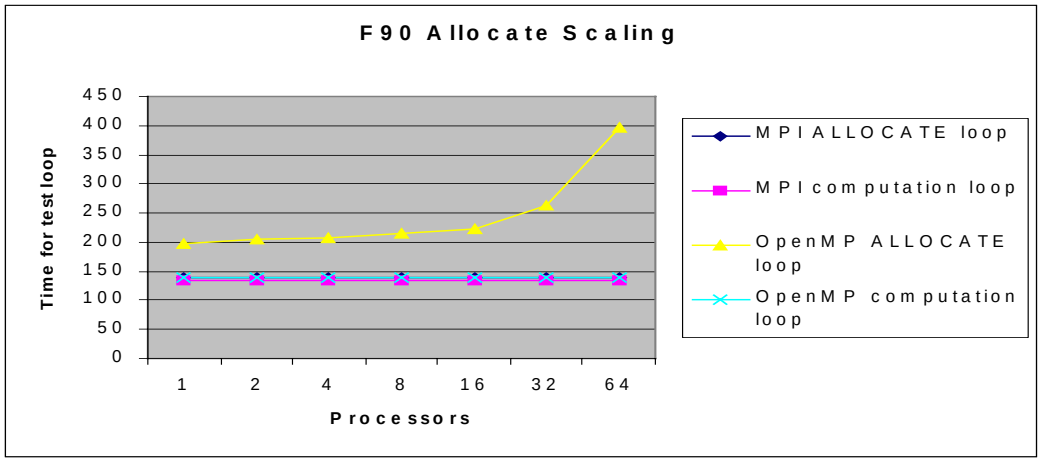


Figure 13 compares the performance of ALLOCATE used with OpenMP and MPI on the following simple test program –

```
DO i=1,1000000
     ALLOCATE( array1(N:M) )
     DO j=N,M
        array1(j) = fn(i,j,N,M)
           RESULT = RESULT + array1(j)
     END DO
     DEALLOCATE( array1 )
END DO
```

Both the OpenMP and the MPI versions were run once with an allocatable array and once with a static array to isolate the penalty for allocation. For this simple loop, as shown in the figure, OpenMP performance is much worse on an SGI O2000 system. Highly modular programs, such as ROCFLO, which allocate and free data frequently can suffer noticeably. The same performance occurred with both the native SGI and the KAI implementations of OpenMP. However, with a different approach, using pointer variables instead of allocatables, we could take advantage of a fast private `malloc()` available in the KAI implementation. Therefore, we consider this an implementation dependent OpenMP problem.

`Data` statements, which initialize variables, and `Save` statements, which make a value persistent across calls to the enclosing routine, define variables which are by default `SHARED` in OpenMP. This is different from the MPI semantics, where every variable is private. So these variables must be made thread private. For example,

```
      DATA variable2/0/
      SAVE variable3
```

would be replaced by –

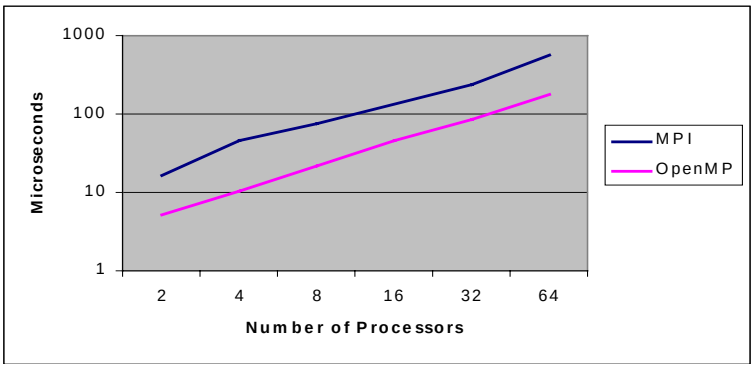
```
      COMMON /PRIV/ variable2, variable3
      DATA variable2/0/
!$OMP THREADPRIVATE(/PRIV/)
```

Attention should be paid to program "make" files which use a `-save` option, available in some compilers, to convert all locally scoped variables to saved variables.

3.4 Collective Operations

OpenMP barriers are semantically equivalent to MPI barriers. In this case, threads-based OpenMP enjoys a performance advantage over MPI, where messages must be sent between processes to implement barriers. On the SGI O2000 architecture, OpenMP barriers are consistently about a factor of 4 faster than MPI barriers. See Figure 14.

Figure 14. Performance of MPI barriers versus OpenMP barriers.



The concept of reductions in OpenMP is different than that in MPI. `MPI_Allreduce()` performs a reduction across processes. In OpenMP the `reduction()` clause applies across parallel loop iterations. When emulating MPI, one wants to "reduce" data across threads inside the parallel region, not inside a loop. Therefore, when the MPI code performed –

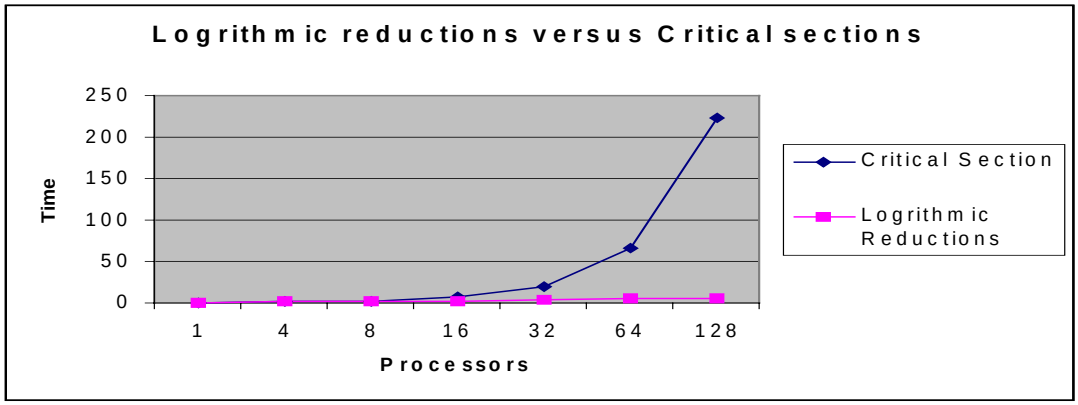
```
      CALL MPI_BARRIER (MPI_COMM_WORLD, F_mpierr)
      CALL MPI_ALLREDUCE (F_dtmin, dtmin, 1,
                          MPI_REAL, MPI_MIN,
                          MPI_COMM_WORLD, F_mpierr)
```

It was replaced by –

```
!$OMP SINGLE
      Global_min = 1.e28
!$OMP END SINGLE
! Implicit barrier
!$OMP CRITICAL
      Global_min = min(Global_min, F_dtmin)
!$OMP END CRITICAL
!$OMP BARRIER
      F_dtmin = Global_min
```

This implementation is straightforward but does not scale well. For scalability, high performance implementations use logarithmic or tree sum reductions. Figure 15 shows the performance of reduction kernels implemented with an OpenMP CRITICAL SECTION and with a REDUCTION clause, using a logarithmic algorithm. Because implementing a sophisticated reduction algorithm is complex and platform-dependent, we propose that OpenMP be extended to include a reduction across threads in a parallel region.

Figure 15. Scaling of Logarithmic reductions versus Critical sections.



In OpenMP, broadcasting data can be much more efficient than in MPI. Any shared data, is implicitly broadcast to all threads by the hardware. So, we need only make the broadcasted variable shared. In ROCFLO for example –

```
      IF (F_pid==0) THEN
            WRITE(*,*)'Enter test prefix:'
            READ(*,*)F_prefix
         ENDIF
      CALL MPI_BCAST (F_prefix, 20, MPI_CHARACTER,
                0, MPI_COMM_WORLD, F_mpierr)
```

was replaced by --

```
      IF (F_pid==0) THEN
            WRITE(*,*)'Enter test prefix:'
            READ(*,*)F_prefix
         ENDIF
!$OMP BARRIER
```

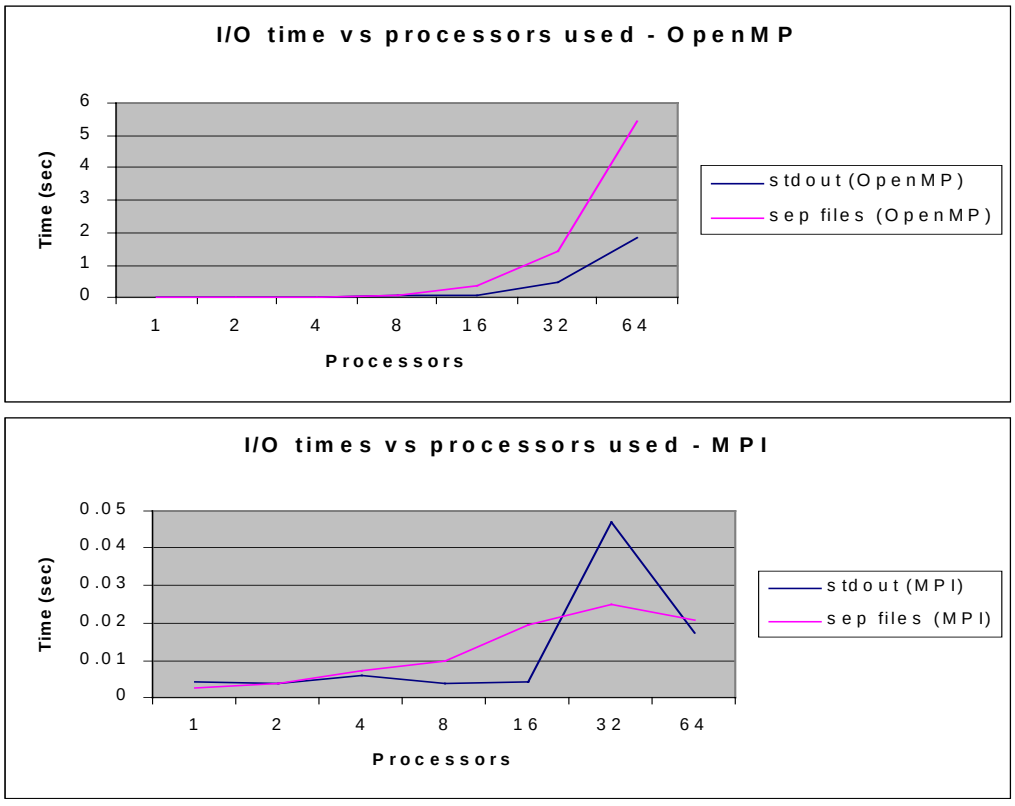
where `F_prefix` is made shared by placing it in a module. (Note that `IF(F_pid==0)` is an idiom for an OpenMP MASTER directive block.)

3.5 Message Passing

Aside from the collective operations described above, ROCFLO uses `MPI_Start()` and `MPI_Waitall()` for persistent sends and receives at the boundaries of adjacent domains. For message passing calls, we built a message passing infrastructure in OpenMP to emulate send and receive functions. Most of the code in the infrastructure deals with the data structures needed to handle persistent messages. The OpenMP part finally involves copying from a thread private data area to a shared message buffer and then the reverse for receives. Synchronization uses `OMP_SET_LOCK() and OMP_UNSET_LOCK()` on persistent message structures. `MPI_Waitall()` waits until all queued messages have been marked completed.

This implementation is not optimal, and was a performance bottleneck. It uses two copies for a send/receive pair. In optimal message passing, only one copy is requiredand OpenMP does not support the transfer of thread private data to thread private data. To make it optimal for message passing would require sharing all the data that is passed between processors. If one does that, one may be able to eliminate the one copy as well by placing logically adjacent data domains in adjacent shared memory.

Figure 16. Scalability of I/O to standard output and to separate files with OpenMP and MPI (notice the different time scales).

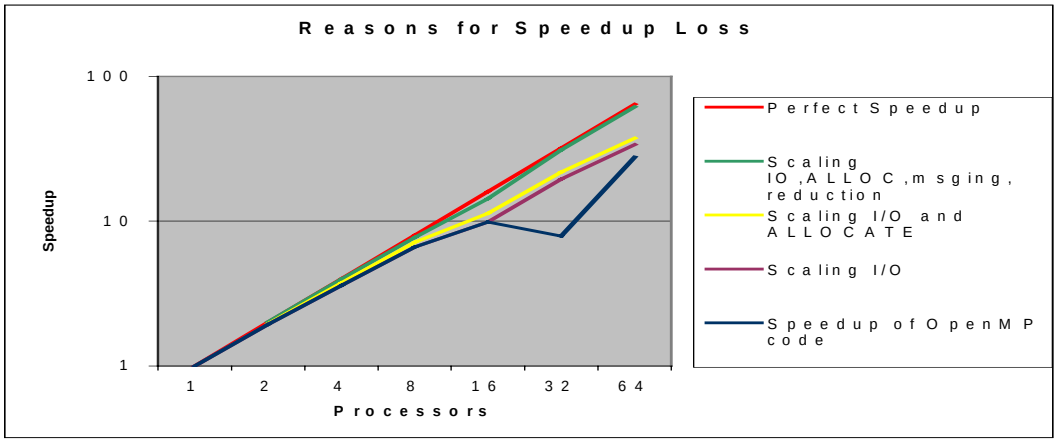


3.6 File I/O

In MPI, unless one uses MPI I/O, each process performs I/O on a separate file. This is inherently more scalable because contention exists only at the operating system and I/O system levels. In the OpenMP implementation we tested, all I/O operations are apparently locked, even those to separate unit numbers. This makes I/O a performance bottleneck for SPMD programs. Figure 16 shows the scalability difference for a simple I/O test kernel similar to the one shown above for allocation in Section 3.3. Note that even for a small number of processors, OpenMP I/O performed 100 times slower than MPI I/O.

3.7 ROCFLO OpenMP Version Performance

Figure 17. ROCFLO OpenMP performance breakdown.



When all of the components in the migration from MPI to OpenMP are complete, the OpenMP domain decomposed version of ROCFLO should scale very well. To isolate the effect on scalability of each of the components, we constructed an experiment whose results are shown in Figure 17. In the test, the size of the domain for each processor was held constant as processors were added. Instrumentation was added around each of the components of the MPI-to-OpenMP migration described above. The bottom curve shows the measured speedup of one run of the code. Each higher curve shows what the speedup would be if the indicated component(s) had scaled as well as with the MPI program. As can be seen, the listed components account for nearly all scaling problems because the final, topmost curve, is very close to the perfect speedup line. Here are our observations on each curve starting from the bottom:

- **I/O** has the greatest scaling problems. This is due, we believe, to two effects. First, the larger effect is that from one run to the next, I/O delays can vary dramatically. We believe that the anomalous speedup value at 32 processors represents run-to-run variations in I/O speed which we could not isolate or resolve. Other runs occasionally displayed similar regressions. Second, locking in OpenMP file descriptor access, as measured, above is a repeatable effect. We consider this to be an OpenMP implementation problem because we don't believe the language should include special I/O constructs. We can resolve this problem by doing less I/O, if saving fewer flow fields is acceptable for our application.

- **Allocates**. This degradation is described and isolated above. Again, we consider this an implementation problem. In this case, other OpenMP implementations have illustrated that the effect can be eliminated on other applications [9] (by using scalable memory allocators).

- **Messaging Infrastructure and Reductions**. As shown in the isolated data above, the majority of the non-scaling is due to non-scaling reductions as implemented in our library. In that sense it is implementation dependent and not a problem with OpenMP. However, we would also like to propose

that the OpenMP Architecture Review Board (ARB) consider supporting reductions of the type we used so that users don't have to re-implement scalable reductions.

4. Summary

To summarize our experiences with scalable parallelism in OpenMP on these two CFD codes, we summarize the major techniques we used and the OpenMP limitations we discovered in Table 2. Some limitations can be avoided by following the coding style we described. Using tools to identify and resolve problems can eliminate some limitations. Some problems are likely to be fixed in future releases of the compiler we used. Some will be aided by changes that are expected in OpenMP V2.0.

Table 2. Key OpenMP issues. (Requests for improvements in OpenMP denoted by "?".)

Point	Technique	Manual Resolution	OpenMP Features or Enhancements
	Incremental OpenMP Parallelization (ROCFIRE)		
Incr 1	Consistent scheduling of multiple fine grain loops.	Careful placement of directives. Hand reindex loops.	Static scheduling. Thread affinity to iteration directive?
Incr 2	Local page ownership on NUMA systems.	Parallelize init loop for first touch.	Data placement directive?
Incr 3	Data domains too big for cache	Interchange, collapse, and tile loops. Reorder array dimensions.	Cache analysis tools to find problems. True multiple level parallel implementation? (e.g.perfex)
Incr 4	Memory motion to master for serial loop between parallel ones.	Parallelize small loops or use ordered scheduling if serial.	OpenMP performance analysis tools, (e.g. guideview)
Incr 5	Minimize barriers.	Fuse loops or use nowait option.	Tool for determining barrier removal validity(e.g.assure)
Incr 6	Scalable barriers and reductions	Hand code logarithmic reduction operations.	Performance certification suite, scalable library code
	MPI to OpenMP Migration (ROCFLO)		
Mig 1	Partitioning data into domains for NUMA.	MPI-like privatization of all data but buffers.	Default private.
Mig 2	F90 allocation synchronization	Move data to local private or thread private common.	Scalable F90 allocation of privates?
Mig 3	DATA and SAVE made thread private.	Move to thread private common.	Private for DATA and SAVE, OpenMP 2.0.
Mig 4	Reductions, broadcast, … of private variables.	Shared reduction var updated with private in critical section.	Reduction on parallel region?
Mig 5	Emulate message passing.	Copy private data to shared buffers and back.	Tool to verify domain ownership of shared data. (e.g. assure)
Mig 6	File unit synchronization.	Use lower level I/O library?	Copy private, in OpenMP 2.0. Parallel I/O Library?

5. Conclusion

Through our ROCFIRE experiment, we believe that we have described an incremental parallelization methodology that can produce ***scalable*** performance, which comes close to that possible with a highly-tuned, domain-decomposed MPI program. In addition, through our ROCFLO experiment, we have illuminated several OpenMP performance problems.

We have suggested additions to OpenMP which would make the incremental path easier and which could fix performance problems. In the short-term, we have identified performance pitfalls which OpenMP programmers can avoid. In the long term, attention to these suggestions by the OpenMP ARB and compiler writers could go a long way toward making OpenMP a mature programming paradigm, in which high-performance is achievable through a series of incremental steps.

Acknowledgements

This work was supported by the U.S. Department of Energy through the University of California under Subcontract number B341494. The experiments were done using the facilities of the National Computational Science Alliance, located at the University of Illinois at Urbana-Champaign.

References

[1] OpenMP home page: "OpenMP: Simple, Portable, Scalable SMP Programming". http://www.openmp.org

[2] Jackson, T.L., J. Buckmaster. and J. Hoeflinger. "Three-dimensional flames supported by heterogeneous propellants", submitted, 1999.

[3] Balsara, D.S.and C.D.Norton, "Highly Parallel Structured Adaptive Mesh Refinement Using Parallel Language-Based Approaches", submitted to Journal of Parallel Computing, 1999.

[4] Tafti, D. K., and G. Wang. "Application of Embedded Parallelism to large Scale Computations of Complex Industrial Flows". International Mechanical Engineering Congress and Exposition, Anaheim, CA, November 1998.

[5] Provatas, N., N. Goldenfeld , and J. Dantzig. "Adaptive Mesh Refinement Computation of Solidification Microstructures using Dynamic Data Structures", Journal of Computational Physics, volume 148, 1998.

[6] Sobh, N., J. Huang, L. Yin, R. B. Haber, and D. A. Tortorelli, "A Discontinuous Galerkin Model for Precipitate Nucleation and Growth in Aluminum Alloy Quench Processes", submitted for publication in International Journal of Numerical Methods in Engineering, 1999.

[7] Zagha, M., B. Larson, S. Turner, M. Itzkowitz. "Performance Analysis Using the MIPS R10000 Performance Counters", Proceedings Supercomputing'96, November 1996, Pittsburgh, PA.

[8] Kuck & Associates, Inc. KAP/Pro Toolset Manuals, http://www.kai.com.

[9] Kuhn, B., P. Petersen, and E. O'Toole. "OpenMP versus Threading in C/C++", First European Workshop on OpenMP (EWOMP'99).

[10] Heath, M. T. and W. A. Dick. "Virtual Rocketry: Rocket Science Meets Computer Science", IEEE Computational Science and Engineering, volume 5:1, pages 16-26, 1998.

[11] Alavilli, P., D. Tafti. and F. Najjar. "The Development of an Advanced Solid-Rocket Flow Simulation Program ROCFLO", 38th AIAA Aerospace Sciences Meeting and Exhibit, January 10-13, 2000, Reno, Nevada.

[12] Ayguade, E., X. Martorell, J. Labarta, M. Gonzalez, and N. Navarro. "Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study". Proc. of the 1999 International Conference on Parallel Processing, Aizu, Japan, September 1999.

[13] Buckmaster, J, T.L. Jackson, and J. Yao. Combustion and Flame. Volume 117, 1999, pages 541-552.

[14] Grama, A., A. Gupta, and V. Kumar, "Isoefficiency Function: A Scalability Metric for Parallel Algorithms and Architectures". IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice, August 1993, Volume 1, Number 3, pp 12-21.