Hybridizing S3D into an Exascale Application using OpenACC

An approach for moving to Multi-Petaflops and Beyond

John M. Levesque
CTO Office
Cray Inc
Knoxville, TN
levesque@cray.com

Dr. Ramanan Sankaran
Scientific Computing Group
Oak Ridge National Laboratory
Oak Ridge, TN
sankaranr@ornl.gov

Dr. Ray Grout
Computational Sciences Center
National Renewable Energy Lab
Golden, CO
Ray.Grout@nrel.gov

***Abstract*— Hybridization is the process of converting an application with a single level of parallelism to an application with multiple levels of parallelism. Over the past 15 years a majority of the applications that run on High Performance Computing systems have employed MPI for all of the parallelism within the application. In the Peta-Exascale computing regime, effective utilization of the hardware requires multiple levels of parallelism matched to the macro architecture of the system to achieve good performance. A hybridized code base is performance portable when sufficient parallelism is expressed in an architecture agnostic form to achieve good performance on a range of available systems. The hybridized S3D code is performance portable across today's leading many core and GPU accelerated systems. The OpenACC framework allows a unified code base to be deployed for either (Manycore CPU or Manycore CPU+GPU) while permitting architecture specific optimizations to expose new dimensions of parallelism to be utilized.**

Keywords-component; OpenMP; OpenACC; Accelerators; MPI; Directives; Multi-core; Communication overlap; Hybrid Architectures; Hybrid Programming

INTRODUCTION

High performance computing is entering an era where more than MPI needs to be used to express the parallelism within the application. Given recent node architectures that feature many processors on the node and the incorporation of accelerators on the node such as the Cray XK series, additional levels of parallelism must be identified on the node. In addition to adding a shared memory parallel paradigm on the node, the application must be written to assure vectorization at the low level. In this paper, a turbulent combustion solver called S3D is used to illustrate the porting of an MPI-only application to a hybrid MPI/OpenMP parallelism. Then OpenACC is employed to instruct the compiler how to generate code for a companion accelerator. A description of S3D software is provided in the next section followed by the approach taken to hybridize the parallelism in S3D. We have provided extensive examples of critical features of the approach to enable other application developers to assess if the approach, which we have found beneficial from a performance portability and maintainability standpoint, is tractable for their applications. Finally, we will discuss performance of the resulting code on nodes with and without accelerators in the recently deployed *Titan* system hosted at Oak Ridge National Laboratory.

S3D FORMULATION AND SOFTWARE

S3D is a massively parallel direct numerical simulation (DNS) solver for turbulent reacting flows developed at Sandia National Laboratories [2]. S3D solves the fully-compressible Navier–Stokes, total energy, species and mass continuity equations coupled with detailed chemistry. The governing equations are supplemented with additional constitutive relationships, such as the ideal gas equation of state, and models for reaction rates, molecular transport and thermodynamic properties. S3D performs a fully explicit time integration of the governing equations with high-order accurate, non-dissipative approximations for the spatial derivatives. The governing equations are solved on a structured 3D Cartesian mesh. The solution is advanced in time through a six-stage fourth-order explicit Runge–Kutta method [4]. The solution is spatially discretized using an eighth-order central differencing scheme and a tenth-order filter is used to remove any spurious high-frequency fluctuations in the solution [5]. The spatial difference and filter operators require nine and eleven point stencils, respectively. CHEMKIN and TRANSPORT software libraries [3] are used to evaluate thermodynamic and mixture-averaged transport properties while optimized routines for reaction rate evaluation are auto-generated from CHEMKIN format mechanism description files.

S3D is written in modern Fortran and employs a pure MPI approach for parallelization between on-node cores as well as between nodes. Serial per-core code is constructed to facilitate efficient compiler generation of SSE instructions for SIMD parallelism, a strategy that has historically produced an efficient, portable and maintainable codebase.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

The computational domain is decomposed in all three dimensions and each MPI process advances the solution in one piece of the 3D domain. The Cartesian mesh and the domain decomposition ensure that all MPI processes have the same number of grid points and the same computational load. The explicit flow solver requires inter-processor communication only between the nearest neighbors in the 3D MPI topology. All-to-all communications are required for occasional monitoring and synchronization ahead of I/O events.

CONVERTING AN ALL MPI APPLICATION TO A HYBRID MPI/OPENMP APPLICATION

There is ample evidence from early studies of MPI+OpenMP parallel implementations that when the communication is costly relative to the computation the MPI-everywhere approach becomes less efficient than a hybrid distributed/shared memory approach [2]. With the large stencils and spatial decomposition used by S3D, this is quickly becoming the case: a problem with 1024^3 gridpoints that scaled well to O(30k) cores only a few years ago has double the communication load relative to the local data size on a per rank basis on today's O(250k) core machines. Ultimately, there is a limit on strong scaling in the MPI-everywhere implementation as the local per core grid size is reduced to the stencil width. This motivates, even in the absence of heterogeneous hardware accelerators, moving to a hybrid distributed/shared memory parallelism.

Why OpenMP?

There are numerous options for employing shared memory parallelism within the node. OpenMP was chosen for this effort because of the desire to develop a performance portable application that would not only run well on many-core node MPP systems but also on many core nodes with accelerators. We have noted that previous efforts to rework codes for systems with accelerators have resulted in improvements with more far-reaching benefits than for the particular target architecture [7]. We wanted to ensure that algorithmic advances made in anticipation of accelerators would also benefit many core performance. Further, in the context of a production science code, which must run on a variety of architectures, it is desirable to not fork development to accommodate a new architecture. This diverts the resources of the code developers to implement and maintain new science capabilities in multiple branches. The intent of the rewrite of S3D was to develop a single code base that could either run on a homogeneous multi-core MPP or on a MPP system that consists of a multi-core chip with a companion accelerator. When the refactoring of S3D started, initial implementations of the proposed *OpenMP extensions for accelerators* directive based acceleration were used for major kernel development. Towards the end of 2011, we moved to the newly formulated *OpenACC* directives which have only cosmetic differences with the proposed *OpenMP* extensions for the subset (those employing data parallelism) needed for S3D. Consensus across a number of stakeholders for the *OpenACC* directives is emerging faster than the *OpenMP* process, allowing us to develop code using the Cray Compilation Enviroment (CCE) with confidence that the resulting *OpenACC* code will be compilable by both Portland Group Compiler and the CAPS compiler. [6] In addition, the code was written so that any *OpenMP* compliant compiler can generate shared memory code for any architecture even if *OpenACC* support is not available. Ideally, the same application that we develop for the current large XT5 and XE6 systems will also be the basis for the next generation of accelerator based systems. Aside from *OpenMP*, other shared memory programming paradigms have so far emphasized a single vendor from the range of future systems (*cuda*, *OpenCL*, and *Pthreads* focus on nVidia, AMD, and Intel architectures, respectively). *OpenMP* can address all of these systems and *OpenACC* can handle the accelerator extensions when accelerators are present.

Hybridization of S3D

On a first look, the hybrid parallelization of S3D using *OpenMP* appears straightforward. Almost all the computations in S3D are in loops traversing the three-dimensional grid. The loops are relatively simple without complex dependencies or critical regions. It is clear that most of these loops could be parallelized automatically by many of the current compilers. However the biggest issue that was foreseen in this approach was the granularity of the *OpenMP* regions generated through automatic parallelization. It is necessary that the hybrid parallelism be scalable to O(10) threads in the near future, and to O(100) threads in the long term. This requires that each of the parallel *OpenMP* regions have enough computation to offset the fixed cost in spawning an *OpenMP* region. This is even more important when a hybrid parallel region has to be launched on an accelerator. With this in mind, we sought to restructure S3D to express much larger granularity than is available at the loop levels.

The code structure of the original S3D code had the major computational loops at the lowest level of the tree. While there are a few exceptions (notably the evaluation of reaction rates and transport coefficients), the amount of computation in almost all of the individual loops was below 1% of the total cost. While the few loops consuming double digit percentages of the total cost could have been easily parallelized without any significant refactoring of the software, doing so would only address approximately 60% of the computation, and our ultimate objective was to cover close to 100% of the computation in S3D under the hybrid parallelism for very high scalability.

When the computational loops are at the lowest level of the call tree, it poses a formidable challenge to achieving efficient *OpenMP* parallelism at the loop level. First, when the loops cycle through the grid points one after the other on the entire sub-grid on the node, the cache utilization is poor. There is very poor data reuse from one computational loop to the other and several opportunities for cache reuse are hidden from the compilers. Secondly, temporary arrays are needed to carry forward the intermediate results between successive loops and *OpenMP* regions in the call chain. This increases the memory footprint on processors and will also increase the cost of launching successive *OpenMP* regions on an accelerator. The approach adopted here is to develop higher granularity *OpenMP* regions which will also become *OpenACC* regions for the accelerator.

The major logic of S3D resides within a '`rhsf`' routine which evaluates the 'right hand side' of the governing equations to be integrated in time by the outer-level `integrate` routine. All of the major computational routines are called from `rhsf`. When looking at the high level looping structures in the program, two major loops within the `integrate` routine call `rhsf`. The timestep loop is the outermost loop and the next inner loop is the Runge-Kutta loop that cycles through six Runge-Kutta sub-steps. A majority of the computational routines called by RHSF are pointwise; that is, they do not require information from adjacent grid points. In the original S3D, all of the communication is encapsulated within in three derivative routines, which are passed arrays to be differentiated in the x-y-z directions. When called, the derivative routines perform asynchronous communication, evaluate the derivative of the internal points, check on the receipt of the communication and then evaluate the derivatives on the boundaries of the sub-grid.

The primary strategy to modifying S3D to contain high-level *OpenMP* structures is to move the grid loops up into RHSF and combine the grid loops, so that all of the computation is separated from the derivative computation. Additionally the pre-posting of receives is moved outside of the routine that performs the derivative computation. This work is described in Section IV. This restructuring to separate the computation from communication now gives us *OpenMP* structures that have the maximum granularity available from spatial parallelism. An additional benefit of this restructuring is the ability to completely overlap computation and communication.

Hybridization of Point Wise operations

Several point-wise operations are required at the beginning of the RHSF routine to calculate quantities used to implement the physics from the primary solution variables. For example, while a transport equation is solved for density-weighted mass fractions (rho*Y), the chemical reaction rates require pure mass fractions Y. Similarly, temperature is required to specify the thermodynamic state and must be recovered from the energy and composition of the mixture. These operations are representative of several point-wise computational routines, and are restructured as shown in the following example:

```
!$omp parallel do private(i, ml, mu)
  DO i = 1, nx*ny*nz, ms
     ml = i
     mu =  MIN( i+ms-1, nx*ny*nz )
     CALL get_mass_frac_r(q, volum, &
                          yspecies, ml, mu )
     CALL get_velocity_vec_r( u, q, &
                          volum, ml, mu )
     CALL calc_inv_avg_mol_wt_r( &
        yspecies, avmolwt, mixMW, ml, mu)
     CALL calc_temp_r( temp, &
                      q(:,:,:,5)*volum, &
                           u, yspecies, cpmix, &
                           avmolwt, ml, mu )
     CALL calc_gamma_r( gamma, cpmix, &
                           avmolwt, ml, mu )
     CALL calc_press_r( pressure,&
                           q(:,:,:,4), &
                           temp, avmolwt, &
                           ml, mu )
  END DO
```

Three loops across the spatial directions are collapsed into a single loop that runs over all three dimensions of the grid. The variable "ms" is the chunksize that is passed down the call chain. This chunksize can be adjusted for the particular architecture to best utilize the cache, TLB and vector units. For the XT and XE systems, we chose "ms" to be 512, which when using 8 byte reals is exactly the size of a memory page. On the XT and XE systems, arrays are not allocated when the ALLOCATE statement is encountered. They are allocated when the array is initialized. In order to assure affinity of the data when using *OpenMP*, these arrays are all initialized with the exact same DO loop structure within *OpenMP* directives. By using "ms" equal to 512, all of the initializations will be done and allocated on page boundaries and since the initialization is also performed in an *OpenMP* loop, the data will be allocated to the processor (thread) that initializes it. This detail gives much better scaling across the cores of the node.

Once the re-write was complete for the *OpenMP* regions, there were 12 computational regions and 8 regions of communication. The major chemistry routine was broken into two separate computational sections. Some of the chemistry routines used with S3D utilize dynamic stiffness removal (DSR) [6] to reduce the computational burden for particularly complicated mechanisms. While without DSR the chemistry routine is purely point-wise, when the DSR algorithm is utilized the chemistry routine requires species diffusion rates that depend on spatial derivatives as an input. We separated the routine into the portion that did not require differentiation of the constitutive variables and that part that did require diffusion. This allows the major communication to be overlapped with this first portion of the chemistry routine. From this restructuring we were able to significantly improve the performance of the S3D application by

1. Reducing the number of MPI tasks, thus improving the overall scaling of the application,
2. Increasing the cache utilization, getting better re-use within the 512 word chunking,
3. Decreasing the amount of memory required by temporaries between computational regions,
4. Achieving good OpenMP scaling with high granularity looping structures,
5. Taking the first step to refactoring the application to run on accelerators.

Moving to Accelerators

Once the Hybrid version of S3D was finished and verified for correctness and performance, our effort turned to taking this version of S3D and porting it to the accelerator. When this effort was started the only defined directives for acceleration was the proposed *OpenMP extensions for accelerators*. Before the work was completed, as mentioned earlier, a new, more complete definition of directives (*OpenACC*) was proposed by a collaboration of Cray, nVidia, PGI and CAPS.[6] Given the approach that we had taken with the OpenMP extensions . changing to the new *OpenACC* required only a trivial change in syntax. Even early versions of the proposed *OpenACC* directives included all of the functionality required for our re-written codebase.

S3D target problems are typically not limited by available system memory, so a strategic decision was made early to target an accelerator that had enough memory to store an up-to-date version of the major computational arrays. With this approach, the only data that must be shipped back and forth from the host to the accelerator is the halo regions to communicate to the other nodes in the system. Given this strategy, we break up the code into code that runs only on the host, mostly the routines dealing with communication, boundary treatment and I/O and the code that runs on the accelerator, the 12 computational *OpenMP* structures. We augmented the reworked communication infrastructure (described in Section IV) to encapsulate accelerator/host communication, and focused on restructuring the *OpenMP*/*OpenACC* regions to facilitate good performance on the accelerator (described in the rest of this section).

Progress of running the application on the accelerator

We start with the time-step and Runge-Kutta loops. At this point all the major computational arrays are replicated on both the host and the accelerator. The first computational region we compute on the accelerator and at the end of this region, an update of halo regions on the host is performed. At this point the second computational region is computed on the accelerator while the host is communicating the halo region. When the communication is complete, the host updates the accelerator with the appropriate halo data. Once the accelerator finishes the second computational kernel, it waits for the updated halo prior to starting the third computational kernel. This continues on through the computation on the accelerator and the communication on the host. Initially we felt that he host should be used to perform the boundary computations to implement the Navier-Stokes Characteristic Boundary Conditions as formulated by Yoo et al. [8] for nodes adjacent to the domain boundary only. These computations contain extensive branching logic to select the appropriate boundary condition but are not compute intensive. Again data is transferred between the host and the accelerator as needed. At the end of each loop, the arrays on the host are brought up to date with the arrays on the accelerator. The size of the computational kernels can be adjusted once the communication timings are determined. Once this approach was coded we realized that much more data would have to be communicated between the accelerator and the host. Since the accelerator had the most recent copy of the variables to be differentiated, it must transfer 16 planes of data to the host, to have the host perform the 8 point stencil on the upstream and downstream in the three directions. This data transfer was excessive and we found having the accelerator do the derivatives for all grid points reduced the amount of data to be transferred and produced better timings. In order to have a compiler convert the application with the accelerator directives, several capabilities must be present. First, the compiler must be able to handle a call chain within an OpenMP region. In this effort we are using the Cray Compilation Environment (CCE). Additionally, there must be a mechanism to asynchronously run the accelerator, while the host is performing its work and the communication. And lastly the compiler must be able to update data on the accelerator from the host and on the host from the accelerator. All of these capabilities are called out in the directives and all compiler vendors involved in *OpenACC* have stated that they will support the proposal by the end of 2012.

Fine tuning the Accelerator Extensions

Part of the fine tuning of the application running on the accelerator requires that the low level DO loops vectorize very well. This will be covered in the next section. The other part of the fine tuning is to designate what arrays can utilize registers and/or shared memory on the accelerator. This second part is performed with *OpenACC*. For example, following is the description of the accelerator loop from the most recent proposed standard [1]:

In Fortran, the syntax of the **loop** directive is
!$acc loop *[clause [[,] clause]…] do loop*
where *clause* is one of the following:

> **collapse(** *n* **)**
> **gang** *[* **(** *scalar-integer-expression* **)** *]*
> **worker** *[* **(** *scalar-integer-expression* **)** *]*
> **vector** *[* **(** *scalar-integer-expression* **)** *]*
> **seq independent**
> **private(** *list* **)**
> **reduction(** *operator* : *list* **)**

Notice that there are several "knobs" that can be used to instruct the compiler to map the loop nest on the accelerator, blocking the iterations across the parallel units and within the SIMD units. Of course that tuning will be performed on the computational kernels that utilize a majority of the time and hopefully, the compiler itself will generate efficient code without needing the use of these directives.

Vectorizing the low level loops

While vectorization can give us only a modest speedup on current X86 architectures due to their short vector length, vectorization for the accelerators can deliver performance enhancements of 10x-20x. Fortunately most of the computational routines within S3D are vectorizable. There are a couple exceptions which we will examine in this section. Vectorization is a lost art and there is quite a bit of old documentation on how to vectorize certain, seemingly unvectorizable constructs. Going forward we anticipate that vectorization will play a much bigger role in performance than we have seen in the past ten years. Along with the vectorization, there is also a desire to increase the computational intensity of DO loops. The computational intensity is the number of floating point operations in the loop divided by the number of loads and stores. The higher the computational intensity, the better performance one can achieve with vectorizable loops. Of course we want to make sure we do not degrade cache performance in rewriting our loops to vectorize. On the X86 architectures, the inhibitors to vectorization can be simplified to the following four items:

1. Loop carried dependencies in the DO loop
2. IF statements in the DO loop
3. Subroutine Calls in the DO loop
4. Non-contiguous accessing of an array in the DO loop

On the accelerator, two of these can be handled, albeit by introducing overhead. For example, all accelerators have predicated execution. That is, a vector operation can be conducted under control of a bit vector which would control the saving of the result. The overhead of this type of operation is that overhead is introduced is not useful. For example, in the following example:

```
DO i = 1,n
     IF( A(i) .LT. 0.0 ) THEN
        B(i)= Complicated computation 1
     ELSE
        B(i)= Complicated computation 2
     ENDIF
ENDDO
```

With predicated execution, both paths of the computation would be performed for all values of I. On the X86 architecture that barely delivers a factor of two with vectorization, this would not be a clear win. On the accelerator, where the vectorized operation runs 20 times faster than scalar, it should be a win. Therefore on the accelerator more overhead can be expended to vectorize a looping construct. Another area where accelerators may perform better is when non-unit strides and/or indirect addressing are used, but this is not a blanket statement. When the accelerator is fetching operands from memory, it does prefer stride one fetches and larger strides will result in fetching more operands than are needed. But since vectorization is such a big win, the degradation of fetching and storing operands can be overcome with increased performance. Indirect addressing is not always bad. For example, if the indirect address is accessing a relatively small data region, one that would fit into the registers or shared memory of the accelerator, there may not be any degradation in the fetching and storing of the indirectly addressed arrays.

Increasing Computational Intensity

The following looping structure is found in a routine to evaluate a diffusive flux:

```
do i = 1, nx*ny*nz, ms
ml = i
mu =  min(i+ms-1, nx*ny*nz)
DIRECTION: do m=1,3
diffFlux(ml:mu,1,1,n_spec,m) = 0.0
grad_mixMW(ml:mu,1,1,m)=grad_mixMW(ml:mu,1,1,m)&
     *avmolwt(ml:mu,1,1)
SPECIES: do n=1,n_spec-1
diffFlux(ml:mu,1,1,n,m)=-Ds_mixavg(ml:mu,1,1,n)&
  *(grad_Ys(ml:mu,1,1,n,m)+Ys(ml:mu,1,1,n)*&
    grad_mixMW(ml:mu,1,1,m) )
diffFlux(ml:mu,1,1,n_spec,m)=&
         diffFlux(ml:mu,1,1,n_spec,m)-&
         diffFlux(ml:mu,1,1,n,m)

enddo SPECIES
enddo DIRECTION
enddo
```

In this example we have significant vector content. There are loops over the chunk-size that was discussed earlier, a loop over directions and a loop over the chemical species. Within the inner loop, six operands are being fetched and two values are being stored. Within the loop the vector operations consist of 3 vector adds and 2 vector multiplies. This represents a poor computational intensity of 5/8. Somehow we would like to increase the intensity of this DO loop. Notice the loop over the directions is only 3. Also notice that two of the arrays are independent of the direction loop. If we unroll the direction loop inside the SPECIES loop, we will get a more compute intensive vector operation. In addition, scalar temporaries are introduced to reduce the number of fetches and stores of `diffFlux`. Finally the looping structure is changed to obtain good vector lengths. These transformations yield the following code.

```
do k = 1,nz
  do j = 1,ny
   do i = 1,nx
     difftemp1 = 0.0
     difftemp2 = 0.0
     difftemp3 = 0.0
     grad_mixMW(i,j,k,1)= grad_mixMW(i,j,k,1)* &
                          avmolwt(i,j,k)
     grad_mixMW(i,j,k,2)= grad_mixMW(i,j,k,2)* &
                          avmolwt(i,j,k)
     grad_mixMW(i,j,k,3)= grad_mixMW(i,j,k,3)* &
                          avmolwt(i,j,k)
     do n=1,n_spec-1
       diffFlux(i,j,k,n,1)=- ds_mxvg(i,j,k,n)* &
       ( grad_Ys(i,j,k,n,1)  &
       + yspecies(i,j,k,n)*grad_mixMW(i,j,k,1) )
       diffFlux(i,j,k,n,2) =-ds_mxvg(i,j,k,n)* &
       ( grad_Ys(i,j,k,n,2)  &
       + yspecies(i,j,k,n)*grad_mixMW(i,j,k,2) )
       diffFlux(i,j,k,n,3) = - ds_mxvg(i,j,k,n)&
        *( grad_Ys(i,j,k,n,3) &
        +yspecies(i,j,k,n)*grad_mixMW(i,j,k,3) )
       difftemp1 = difftemp1-diffFlux(i,j,k,n,1)
       difftemp2 = difftemp2-diffFlux(i,j,k,n,2)
       difftemp3 = difftemp3-diffFlux(i,j,k,n,3)
     enddo  ! n
     diffFlux(i,j,k,n_spec,1) =  difftemp1
     diffFlux(i,j,k,n_spec,2) =  difftemp2
     diffFlux(i,j,k,n_spec,3) =  difftemp3
     grad_T(i,j,k,1)=-lambda(i,j,k)* &
                              grad_T(i,j,k,1)
     grad_T(i,j,k,2)=-lambda(i,j,k)* &
                              grad_T(i,j,k,2)
     grad_T(i,j,k,3)=-lambda(i,j,k)* &
                              grad_T(i,j,k,3)
     do n=1,n_spec
       grad_T(i,j,k,1)=grad_T(i,j,k,1)+ &
           h_spec(i,j,k,n)*diffFlux(i,j,k,n,1)
       grad_T(i,j,k,2)=grad_T(i,j,k,2)+ &
           h_spec(i,j,k,n)*diffFlux(i,j,k,n,2)
       grad_T(i,j,k,3)=grad_T(i,j,k,3)+ &
           h_spec(i,j,k,n)*diffFlux(i,j,k,n,3)
        enddo   ! n
      enddo     ! i
     enddo              ! j
   enddo                ! k
```

This restructuring, which results in a computational intensity of 15/15 helps the accelerator significantly and perhaps more importantly it helps the multi-core vectorization and cache utilization. In this example, the Cray Fortran compiler performs outer loop vectorization using the I loop as the principal vector loop. In the process, the I loop is interchanged with the inner N loop. This process significantly improves register utilization. On the Fermi + system this rewrite achieves an increase of 2-3 over the original and on the Opteron it achieves an increase in speed of 1.4. Many may think the compiler should perform the above transformations; however, at this stage no compiler can achieve the improvement achieved by the hand restructuring. Going forward to Exascale, these types of restructurings will be required to get the performance from cache based computation.

Vectorizing a loop that contains an iteration loop is one of the more difficult vectorizations that we had to employ in S3D. In the `calc_temp` routine, we find the following loop implementing a 1D Newton solve to calculate the temperature from the mixture enthalpy and heat capacity:

```
tmp1(ml:mu) = e0(ml:mu) - 0.5*tmp1(ml:mu)
LOOPM: DO m = ml, mu
   icount = 1
   r_gas = Ru*avmolwt(m)
   yspec(:) = ys(m, :)
   ITERATION: DO
    cpmix(m) = mixCp( yspec, temp(m) )
    enthmix  = mixEnth( yspec, temp(m) )
    deltat =    &
       ( tmp1(m) - (enthmix- &
         r_gas*temp(m)) )   &
          / ( cpmix(m) - r_gas )
    temp(m) = temp(m) + deltat
      IF( ABS(deltat) < atol ) THEN
         cpmix(m) = mixCp( yspec, &
     temp(m) )
     EXIT ITERATION
   ELSEIF( icount > icountmax ) THEN
     STOP
   ELSE
     icount = icount + 1
   ENDIF
  ENDDO ITERATION
ENDDO LOOPM
```

This loop contains several vectorization inhibitors. `mixCP` and `mixEnth` are function calls. Within these functions are additional IF tests, STOP statements and error prints. In this case we have to vectorize this loop: the other alternatives are not acceptable. We could run this on the host; however, that would require that we transfer a significant amount of data over to the host to perform the computation. Alternatively we could run it in scalar mode; however, scalar mode on the accelerator is much slower than the host, so we must restructure this code to achieve performance on the SIMT units (Single Instruction, Multiple Stream). Several restructurings applied here. First the ITERATION loop was moved outside the mesh loop and logic was introduced that computes the temp for those zones that did not converge. Also, checks are made on convergence and a simple integer array holds the converged (zero) and non-converged (non-zero) value. If the computation repeats more than icountmax times, we will STOP. Additionally the function calls perform the computation for all elements, even after some have converged and keep computing until all elements have converged. If the number of iterations is higher than the maximum allowed then we will print out an error message and halt. This will introduce additional computation. For example, if one of the 512 elements do not converge, then using predicated execution we will have to execute all 512 elements again until the one lone element converges. This particular restructuring improved the speed on the accelerator by a factor of 30. On the multi-core system it ran a little slower. Actually the number of iterations that were required to converge was typically 1, at most 2. Certainly some cases would be worse on the multi-core system. An improvement would be to pack the elements that need to be recomputed and only update those. This would probably run better on the multi-core since it would reduce the amount of operations when many iterations are required. This is an issue for some of the vectorization that is required for the accelerator: some rewrites may run slower on the multi-core.

REORGANIZATION OF THE MESSAGE PASSING TO ACHIEVE MORE OVERLAP

Provisioning the halo data required for the derivative stencil (4 points to either side of the point being evaluated in the direction of differentiation) requires communication with the neighboring MPI ranks. In the non-contiguous directions, this also involves packing data strided by nx or nx*ny.

The traditional derivative evaluation approach in S3D was designed to overlap communication and computation as efficiently as possible when considering each derivative to be evaluated in isolation for an MPI-everywhere formulation. S3D sequentially performed the tasks for each derivative (pack buffers, post non-blocking receives and sends, compute derivative for interior points, wait for communication, and compute derivative for points using halo data). The derivatives for the interior points were computed first, followed by the points requiring halo data after communication finished (see Figure 1). This approach was highly successful for previous systems, but has several drawbacks. Notably, separation of the communication from the derivative computation so that the communication can be overlapped with other (non-derivative) local computation is problematic.

Derivative Queue System

A derivative queuing system has been implemented in S3D. This system splits the derivative computation into separate tasks, allowing for freedom to interleave other tasks:

1. Memory allocation for send/receive buffers
2. Preposting of MPI receives

3. Buffer packing
4. Sending data
5. Blocking until communication is complete
6. Evaluating derivatives

At initialization, an array of structures is allocated to manage the data exchange in each (x,y,z) direction along with an array of buffers for packing data prior to send and receive in each direction. These two data structures are managed by four functions that replace the functionality of the historical `derivative_x`. These functions are, for derivatives in the x-direction:

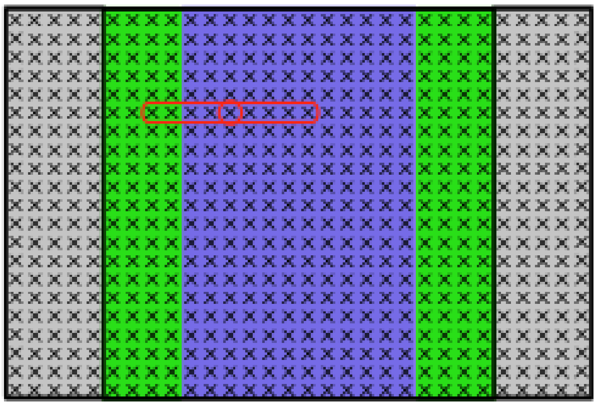

Figure 1: Slice of local data with halo data (grey) at periphery. Historical formulation computes interior derivatives (blue) first then blocks until communicaiton is complete before evaluating edges (green).

`derivative_x_post`

Marks entry as in use, optionally computes checksum of derivative field data, looks up send/receive rank ids, and pre-posts MPI_IRecv.

`derivative_x_send`

Packs data into contiguous buffers and initiatiates MPI_ISend.

`derivative_xyz_wait`

Blocks until all queued derivative communication is complete.

`derivative_x_calc_buffer`

Evaluates derivatives and marks buffers as unused.

For example, the derivative contributing to the advective and diffusive transport in the 'y' direction can be evaluated with the sequence of calls that are placed inside a loop over the number of species (index n):

```
CALL derivative_y_post( nx,ny,nz, &
                        tmp, n, &
                        'spec-y' )
tmp(:,:,:) = -q(:,:,:,5+n)*u(:,:,:,2) &
             - diffFlux(:,:,:,n,2)

CALL derivative_y_send( nx,ny,nz, &
                        tmp, n, &
                        'spec-y' )
! Opportunity to insert computational heavy
! code here
CALL derivative_xyz_wait(n)

CALL derivative_y_calc_buff(nx, ny, nz,&
                   tmp, tmp2, &
                   scale_1y, 1, SPECY )

rhs(:,:,:,5+n) = rhs(:,:,:,5+n)+tmp2
```

Similar loops exist for the x- and z- directions. Since there is no requirement that these calls be sequential – so long as the operand field (tmp above) is not modified between the `_send` and `_calc_buff` calls, other computation can be inserted to overlap with the communication. We take advantage of this capability to regroup the above; rather than three loops over the three directions where we pack, send, wait, then calculate for each species, we instead have two loops: in the first we pack and begin to send all of the species halos for differentiation in the three directions, then perform compute-heavy operations, wait for any remaining communication to finish, and finally evaluate the derivatives.

The communication between the accelerator/host is implemented by augmenting the `derivative_x_send` routine. We have arranged the restructured code with significant local computation positioned between the start of the data movement and the derivative evaluation to absorb the overhead of moving data.

One nice feature of OpenACC is that the host can asynchronously update the device as the messages arrive from other processors. This allow the PCI Express transfers to be overlapped as well as the communication between the hosts

Combining derivatives for to improve data reuse

In several instances, it is necessary to compute a species gradient, comprised traditionally of evaluating derivatives in the x-y-z directions sequentially. To improve data reuse and cache utilization, these are replaced by a single loop over the operand field where all three derivatives are computed.

RESULTS

In this section we will show the comparison of the hybrid version of S3D versus the all-MPI version on *jaguarpf* when it was a Cray XT5 containing over 20,000 nodes of two six core Opteron Istanbul sockets. Then we take the same version of the code and test it on a Cray XK6, which has a single 16 core Opteron Interlagos socket and a nVidia Fermi + accelerator. To be fair we will test the code running on the Cray XK6 compared to the code running on the Cray XE6, which has two 16 core Opteron Interlagos sockets. Both the XK6 and the XE6 take up the same amount of board space. At the time of paper submission, the XK6 had 960 nodes with Fermi+ accelerators, by Supercomputing 2012, the system will have a least 20PFlops of XK6 using the same Interlagos socket with the nVidia Kepler accelerator. At that time, we will give timings of S3D running across all available GPU nodes in the final configurations as the science case that will be run on Titan after exceptance will require the full machine.

Correctness

A particularly advantageous feature of common *OpenMP* / *OpenACC* regions from a developer productivity perspective is that significant correctness testing and debugging can be done on the *OpenMP* code, vastly reducing the effort to debug the implementation on the accelerator. The development process is much easier in the *OpenMP* framework because of a number of factors. The difference in build times needs attention. (an order of magnitude difference between a the time to build a debug version of the *OpenMP* code versus a few hours for an optimized accelerator build) alone gives a more rapid turnaround. Development tools are much more mature in a CPU-only environment, and executing tests in the *OpenMP* paradigm is a useful debugging tool in itself as the number of threads can be varied (down to a single thread per MPI rank), and errors associated with data movement between the host and accelerator can be isolated.

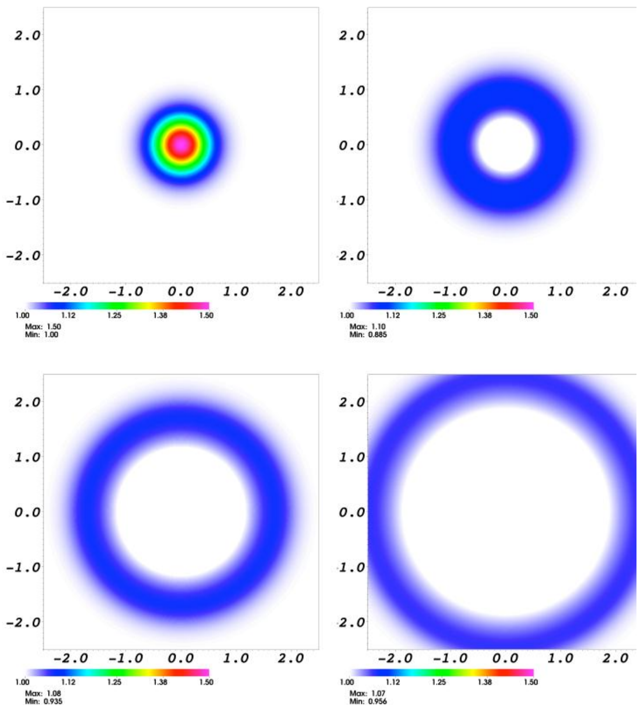

Figure 2: 2D Pressure wave test used for correctness testing.

Throughout the process, a hierarchy of tests was utilized to verify correctness. Many of these approached unit tests (e.g., applying the differentiation operators to known functions with analytic solutions), while others were physics-based comprehensive tests, including computing ignition delay times, acoustic wave propagation, and tests to ensure statistically homogenous turbulence problems were invariant under reflection or rotation. Such physics based tests are particularly useful as regression tests, because errors can often be identified based on understanding how various processes affect the solution. In the acoustic wave propagation solution shown in Figure 2 from the completed hybridized code, an initial pressure peak in the center of the domain spreads out symmetrically and passes through non-reflecting boundaries. An anomaly when the encountered a particular boundary, for example the left edge, could point us to a specific region of code.

Performance of the Hybrid version of S3D

S3D is routinely run on 120,000 cores using a single MPI task for each core on the socket. With the introduction of the Istanbul Opteron node containing two 6-core sockets, the scaling of S3D began degrading as the number of nodes employed grew. The principal reason for the degradation was the number of messages leaving the node destined for other nodes. By grouping a 3-D sub-domain of the grid on a node, S3D scaling was improved significantly. Even with this MPI task placement the scaling of S3D above 100,000 cores was not as good as we would like. Prior to developing the Hybrid version of S3D, limiting the number of MPI tasks by introducing threads on the node was viewed as the principal advantage of developing a hybrid implementation.

Figure 3 shows the wallclock time for various components during runs on 64 nodes of 12 threads, 720 nodes of 12 threads and 7200 nodes of 12 threads. The all-MPI runs used 768 cores, 8,640 cores and 86,400 respectively. The components plotted are User time; that is, the amount of time used computing, MPI time, the time used doing communication, MPI Sync time, the time cores were waiting for other cores and finally *OpenMP* Overhead time. The total runtime is also plotted.

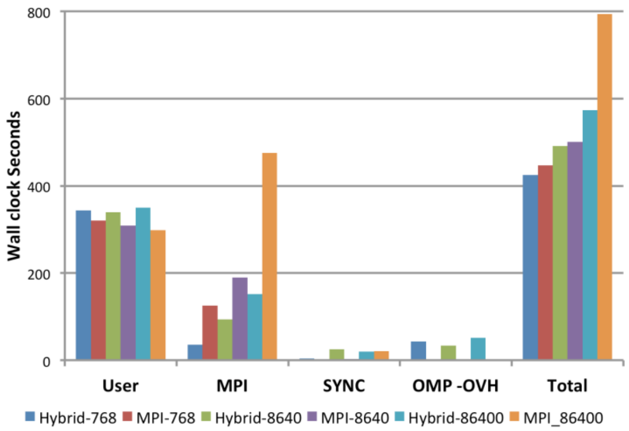

Figure 3: Comparison of 'MPI-everywhere' and Hybridized code

These times are averaged over all the cores used. The compute time comparisons are interesting. All MPI uses more wallclock time than the hybrid version of the code. The hybrid version of the code actually attains better cache utilization and produces a higher percentage of peak than the all MPI version of the code; however, the percentage of computation that is parallelized with *OpenMP* is not 100%.

The all MPI code does attain 100% parallelization of user time; however, the all MPI version of the code introduces overhead in the form of communication. The Hybrid version of S3D also has overhead of communication, but not as much because the number of MPI tasks is reduced by a factor of 12.

The real advantage of the hybrid version of the code is the reduction of the MPI time and MPI_SYNC time. The hybrid version does introduce overhead in the form of managing the *OpenMP* regions. The total of these four components of time illustrates an increasing advantage of the hybrid version of S3D as the number of utilized cores increases.

One final note on this comparison, these timings are collected on a problem important to the study of combustion in a engine cylinder – a very important problem to solve. Even without looking at the performance supplied by the companion accelerators – the ultimate goal of hybridizing S3D - the performance gain attained with the hybrid version of S3D is a significant improvement. Fortunately, the process of refactoring S3D to utilize an accelerated node, has produced a version of the code that runs much better on all Massively Parallel Systems with multi-core nodes.

Looking at XK6 performance

In the process of optimizing S3D for the accelerator, improving computational intensity and vectorization, a new OpenMP version was generated. In this comparison we give timings for two OpenMP versions. First the original hybrid version that was compared to the XT5 in the previous section and then the optimized version that is also the accelerated version. Fortunately in this case the optimizations applied to improve the performance of the accelerator improved the performance of the OpenMP version. Before discussing the results, it is appropriate to review the transformations that were applied to arrive at the final version of the Hybrid S3D.

Restructured the looping structure to raise the low level grid loops up into the RHS routine. In the process combined several point wise computations together within the same OpenMP structure.

Reordered the computation and communication to achieve as much overlap as possible

Restructured computation to reduce memory operations and vectorize all loops that would reside on the accelerator

Utilized OpenACC directives to move major computational arrays to the accelerator prior to the time step loop and control minimal data communication between the host and the accelerator with asynchronous updates

Fine tuned the OpenACC loop directives to achieve best blocking

The first three transformations represent the majority of the work in the project. This approach is highly recommended: generate the best hybrid version possible and then add the OpenACC directives. In this case, the acceleration of the hybrid code was only 30% of the total work.

The accelerator that was used in the timings is an early version of the nVidia Kepler with 14 parallel units (SM) and each SM containing 32 SIMT threads. Each node of the XK6 system contained a single 16 core Interlagos socket with a single nVidia Kepler accelerator. The connection between the host Opteron and the Fermi was a PCI Express Version 2 connection, rated at 8 GB/s.

Figure 4 gives the times of the original *OpenMP* version of S3D as well as final hybrid version of S3D on an XE6, containing 32 cores, the Opteron only part of the XK6, containing 16 cores and finally the full hybrid XK6 node of 16 cores and the nVidia accelerator. All runs were made with 768 MPI tasks where each node had a single MPI task. The number of *OpenMP* threads for the XE6 was 32, and for the XK6, 16. The accelerator code used 16 *OpenMP* threads for the host code and a single accelerator for the accelerated code.

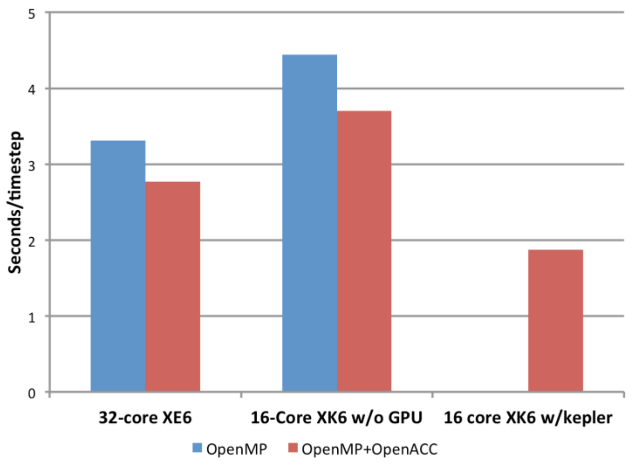

Figure 4
Performance of Hybrid S3D on Multi-core and Accelerated MPP systems

Future Improvements

Several improvements are slated for fall of 2012. The target accelerator for the *Titan* system at Oak Ridge National Laboratory will be a later version of the nVidia Kepler system. The details of the kepler based system cannot be currently disclosed in this paper. However the full performance details will be available for publication before September 2012.

In addition to the hardware improvement that arrives with Kepler, a new nVidia runtime is expected to allow for communication between the accelerators to be performed without going through host memory. The data will be able to be transferred directly from the memory on one accelerator, out to the XK6 interconnect into the memory of an accelerator on a remote node. While we don't have precise performance predictions for the S3D code on the kepler based Titan system, we are targeting 1.0 s/timestep, a factor of seven on a node to node comparison between the XT5 *Jaguarpf* system and the new *Titan* XK6 system.

CONCLUSION

In the paper we have shown an approach that can be used to rewrite an application for both performance and portability on the next generation of high performance computers. The *OpenACC* directives for accelerators provides the mechanism to stay in a current high level language, a characteristic important to moving innumerous legacy software to many-core and accelerator based nodes. While the approach relies upon the ability of compilers to perform the heavy lifting of generating efficient code for the target architecture, there is still a significant amount of work involved in developing high level, large granularity OpenMP regions. S3D was significantly refactored by application scientists who understand the application and computing specialists who understand the hardware and the art of vectorization working together. The work is comparable to that required twenty years ago to port an application from a shared memory parallel vector system to a distributed memory parallel system. When embarking upon the refactoring process, it is extremely important to target many core nodes with and without accelerators. For example, operations within the *OpenMP* regions that cannot be performed on the accelerator must not be used, as the case with our separation of the message passing from the computation.

S3D is one of six principal applications that ORNL has targeted for "day one" science when the *Titan* system is accepted later this year. To our knowledge, S3D is the first full production application that has been ported to *OpenACC*, and it is a demonstration of what can be achieved with the directive approach. The tremendous advantage of *OpenACC* is that it can incrementally be introduced to a MPI/*OpenMP* code. The compiler will give feedback on what it has done to move a particular *OpenMP* region to the accelerator, using nVidia's *cuda* profiler, the cost of the data transfers and kernel execution are given. The user can use this information to improve the performance of the kernel, eventually multiple regions can be combined and data kept on the accelerator. The user can even call a *cuda* routine from the *OpenACC* program, passing array locations on the accelerator. We believe that the final version of S3D is well positioned for any accelerator going forward. The code exhibits three levels of parallelism and has optimized data transfer to and from the host. If a particular accelerated node has a single address space shared by both the accelerator and the host, the data transfer may still be necessary to assure data locality. If the target accelerated node does not have a host, then the *OpenMP* option should run extremely well: this is a performance portable branch of S3D.

RELATED WORK

"Using a hybrid MPI/OpenMP strategy to implement a parallel algorithm has been the subject of much experimentation using both benchmarks (e.g., Drosinos and Koziris, 2004; Krawezik, 2003; Cappello and Richard, 1998), application kernels (e.g., Nakajima, 2005) and full applications (e..g, Djomehri et al., 2002; Jones et al., 2006; Su et al., 2004; Mininni et al., 2010). The common theme expressed in discussions of the hybrid approach by Rabenseifner et al., (2009) and Smith and Bull (2001) is that the hybrid approach excels where there is an opportunity to reduce communication or alleviate memory usage from replicated data by incorporating awareness of machine topology into the algorithm. MPI is very well suited to expressing locality, so an improvement by including OpenMP is closely dependent by how efficiently the local shared memory can be leveraged to reduce the overhead of intra-node communication (Giraud, 2002; Hu et al., 1999). Cappello and Etiemble (2000) indicated that this is likely to be the case when fast processors increase the relative cost of communication, but significant fine-grain parallelism must be expressed to realize an improvement over pure MPI."

REFERENCES

J.C. Beyer, E.J. Stotzer, A. Hart, and B.R. de Supinski, "OpenMP for Accelerators," *to appear in the proceedings of the 7th International Workshop on OpenMP*, Chicago, IL: 2011.

F. Cappello and D. Etiemble, "MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks," *Proceedings of the IEEE/ACM SC2000 Conference (SC'00)*, 2000.

J.H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E.R. Hawkes, S. Klasky, W.K. Liao, K.L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C.S. Yoo, "Terascale direct numerical simulations of turbulent combustion using S3D," *Comput. Sci. Disc.*, vol. 2, 2009, p. 15001.

R.J. Kee, F.M. Rupley, E. Meeks, and J.A. Miller, CHEMKIN-III: A fortran chemical kinetics package for the analysis of gas-phase chemical and plasma kinetics, 1996.

C.A. Kennedy, M.H. Carpenter, and R.M. Lewis, "Low-storage, explicit Runge-Kutta schemes for the compressible Navier-Stokes equations," *Applied Numerical Mathematics*, vol. 35, 2000, pp. 177-219.

NVIDIA,Cray,PGI,CAPS Unveil 'OpenACC'Programming Standard for Parallel Computing, Supercomputing 2011

T. Lu, C.K. Law, C.S. Yoo, and J.H. Chen, "Dynamic stiffness removal for direct numerical simulations," *Combustion and Flame*, vol. 156, Aug. 2009, pp. 1542-1551.

J. Mohd-Yusof, D. Livescu, and T. Kelley, "Adapting the CFDNS Compressible Navier-Stokes Solver to the Roadrunner Hybrid Supercomputer," *Parallel Computational Fluid Dynamics: Recent*

Advances and Future Directions, R. Biswas and N.A.S. NASA Ames Reserach Division, eds., DEStech Publications, Inc, 2010, p. 95.

C. Yoo, Y. Wang, A. Trouvé, and H. Im, “Characteristic boundary conditions for direct simulations of turbulent counterflow flames,” *Combustion Theory and Modelling*, vol. 9, Nov. 2005, pp. 617-646.