Final Report: Center for Programming Models for Scalable Parallel Computing Towards Enhancing OpenMP for Manycore and Heterogeneous Nodes

Barbara Chapman
chapman@cs.uh.edu

February 2012

Project Period: September 2010 - September 2011
University of Houston,
Computer Science Department
Philip G. Hoffman Hall 501
4800 Calhoun Rd
Houston, TX 77204-3010

1 Overview of Research at University of Houston

OpenMP was not well recognized at the beginning of the project, around year 2003, because of its limited use in DoE production applications and the inmature hardware support for an efficient implementation. Yet in the recent years, it has been graduately adopted both in HPC applications, mostly in the form of MPI+OpenMP hybrid code, and in mid-scale desktop applications for scientific and experimental studies. There are at least two reasons behind it, the shift to spatial parallelism in the form of multiple and many homogeneous and heterogeneous power-efficient cores, and the urgency to have a programming model and support toolchains to enable productive development of applications untilizing the large-degree of parallelism. We have observed this trend and worked deligiently to improve our OpenMP compiler and runtimes, as well as to work with the OpenMP standard organization to make sure OpenMP are evolved in the direction close to DoE missions. In the Center for Programming Models for Scalable Parallel Computing project, the HPCTools team at the University of Houston (UH), directed by Dr. Barbara Chapman, has been working with project partners, external collaborators and hardware vendors to increase the scalability and applicability of OpenMP for multi-core (and future manycore) platforms and for distributed memory systems by exploring different programming models, language extensions, compiler optimizations, as well as runtime library support.

We are active participants in the activities of the OpenMP Architecture Review Board (ARB), the standards organization that maintains and further develops the OpenMP programming interface. Since joining the Multicore Association (MCA) in the last project period, we have been actively involved in the development of low-level standards for programming heterogeneous systems that may serve as an implementation vehicle for future OpenMP enhancements for nodes that have heterogeneous cores. The technical exploration during this final project period consists of language extensions, compiler and runtime enhancement, as well as application experiments with other programming systems for the current and future computer platforms. These research and development activities address multiple challenges in the context of OpenMP, such as locality, compiler and runtime optimizations for memory accesses of parallel programs, productivity and robustness of compilers etc. Our approaches are applicable to general node-level programming model. We are also collaborating with our partners in the PModels Project to explore OpenMP interoperability with other programming models and ensure that it can be deployed with each of the message passing libraries and PGAS models. Our practical work is implemented within the robust OpenUH compiler infrastructure which serves as an indispensable test bed for validating the results of our research on real applications.

We have continued to explore language and compiler solutions to locality challenges of OpenMP programs by extending the "*location*" concepts introduced in the previsous project year. Using *locations*, programmers will be able to control where data is located as well as to manage tasks to be executed close to their associated data [11]. With the concept, OpenMP user can specify a parallel region mapping with a collection of *locations*, determine an OpenMP work-sharing construct to be executed by a set of *locations*, and allocate an OpenMP task on a specific *location*. User can use *locations* to further optimize their OpenMP code by specifying data and task affinity within a *location*. Additionally, we introduced a mechanism to express data layout into OpenMP, to allow OpenMP programmers to control and manage the data layout.

In our efforts to create an OpenMP 3.x compiler and runtime, we have implemented a configurable task pool framework that allows the user to choose at runtime which task pool organizations to employ [13]. We currently have four different task pools implemented that utilize distributed, hierarchical, and hybrid queue organizations. We have added to the OpenUH runtime a new synchronization mechanism similar to the clock in X10 and the phaser in the X10 variant Habanero-Java. This would allow point-to-point synchronizations and aid in the implementation of reduction operations on task groups. This provides the ability to form implicit groupings of task as well as more flexible synchronization.

OpenMP is under active growth thus a compiler should adapt itself to the rapid change of OpenMP specifications. Based on PI's previous work together with colleagues at the University of Stuttgart, Germany [15] [18], we developed an OpenMP validation suite designed to validate the correctness of an OpenMP implementation. It covers all feature tests of the latest OpenMP 3.1 directives and clauses as well as stress tests to evaluate the robustness of OpenMP compiler implementations. Furthermore, we also developed an execution environment infrastructure that could manage and automate validation tests easily as well as analyze and show the final results in a user-friendly manner.

We also continued our work of creating a production-quality Co-Array Fortran compiler, including improving our compiler to conform to the Fortran 2008 standard for Co-Array handling, and enhancing both the compiler and runtime to perform optimizations for operations that access Co-Arrays. In a joint project between UH and Total, we have investigated CAF as a viable programming model for production Oil and Gas applications [7]. We also investigated optimizations to alleviate communication and synchronization costs during compilation and at runtime. In the course of this work, we demonstrated the viability of CAF as a programming model that allows for more intuitive and cleaner algorithms while delivering competitive performance compared to MPI.

Writing a parallel shared memory application that achieves good performance and scales well as the number of threads increases can be challenging. One of the reasons is that as threads proliferate, the contention among shared resources increases and this may cause performance degradation. The work in the runtime optimization research focused on detecting performance bottlenecks caused by false sharing in OpenMP applications. We have introduced a dynamic framework [23] to help application developers detect instances of false sharing as well as identify the data objects in an OpenMP code that cause the problem. The framework that we have developed leverages features of the OpenMP collector API to interact with the OpenMP compiler's runtime library and utilizes the information from hardware counters. We have demonstrated the usefulness of this framework on actual applications that exhibit poor scaling because of false sharing.

Compiler approach to addressing the false sharing problems in an program has also been studied [22]. In this approach, we used a compile-time cost model to estimate the performance impact of false sharing on parallel loops. With the help of cost models, the compiler is able to estimate whether the specific transformation is profitable in terms of execution time and determine the optimal level of the transformation, if applied. We validated our model by comparing the false sharing overhead percentages obtained by measuring from the execution time against the ones computed by our model. The modeling results are comparable to the real execution behavior from 2 to 48 threads tested, showing the model can accurately quantify the false sharing impact at compile-time. The false sharing cost model will be used by compilers to guide the parallel loop transformations by providing more accurate timing estimation for parallel loops. These modeling and estimation results could also be useful for programmers for performance tuning and locality optimizations.

HPC systems now exploit GPUs within their compute nodes to accelerate program performance. As a result, high-end application development has become extremely complex at the node level. In addition to restructuring the node code to exploit the cores and specialized devices, the programmer may need to choose a programming model such as OpenMP or CPU threads in conjunction with an accelerator programming model to share and manage the difference node resources. This comes at a time when programmer productivity and the ability to produce portable code has been recognized as a major concern. In this work, we have evaluated the state of the art accelerator directives to program several applications kernels, explore transformations to achieve good performance, and examine the expressivity and performance penalty of using high-level directives versus CUDA. We also compare our results to OpenMP implementations to understand the benefits of running the kernels in the accelerator versus CPU cores.

2 Technical Accomplishments

In this section, we describe our accomplishments in the PModels 2 project during this final project period. We have further enhanced our OpenMP implementation with locality support, described in Section 2.1, have created the OpenMP 3.x compiler and runtime implementation in our OpenUH compiler, see Section 2.2, and also developed a validation suite for the latest OpenMP specification, illustrated in Section 2.3. We have made significant progress in the work of creating a Co-Array Fortran compiler and runtime, which will be reported in Section 2.6. Our work in runtime and compile to help optimize memory access and locality in OpenMP parallel codes are described in Section 2.7 and 2.8. We have also reported our evaluations of directive based GPGPU programming model and compared it with other approaches, as presented in Section 2.9.

2.1 Locality-Aware OpenMP

We have continued to explore language and compiler solutions to locality challenges of OpenMP programs by extending the "location" [12] concepts introduced in the previsous project year. Using *locations*, programmers will be able to control where data is located as well as to manage tasks to be executed close to their associated data. Our goal is to enhance OpenMP with explicit-locality programming constructs that will scale to the demands of emerging petascale and future exascale architectures. *Loca-*

tions provides an additional logical layer between the current OpenMP programming model and underlying hardware.

With the concept, OpenMP user can specify a parallel region mapping with a collection of *locations*, determine an OpenMP work-sharing construct to be executed by a set of *locations*, and allocate an OpenMP task on a specific *location*. By specifying the *locations* in a program, user can control where a task is executed, bind threads with hardware, and set data layout. User can use *locations* to further optimize their OpenMP code by specifying data and task affinity within a *location*. The fundamental assumption of the concept is that a task affinity will be able to access data at the same *location* faster than data at other *locations*.

We introduce a new runtime environment variable OMP_NUM_LOCS that defines the number of locations, similar to the number of threads in the current OpenMP specification. If the environment variable has not been set, the program assumes one location by default. We introduce a parameter NLOCS as a pre-defined variable for the number of locations in an OpenMP program. It is similar to the THREADS parameter defined in the UPC language [8]. The parameter remains constant during an execution. The parameter is necessary in the definition of data layout, as well as in compiler and runtime implementation. Each location is uniquely identified with a number MYLOC in the range of [0:NLOCS-1].

2.1.1 Syntax of Location

In our design, we limit the location usage as a clause associated with OpenMP parallel, OpenMP worksharing, and task directives. The syntax of location clause is $location(m[:n])$ where m and n are two integer numbers ranging from 0 to NLOCS-1. A single number "m" represents the id of the location where the associated OpenMP construct will be executed. Two numbers separated with a colon, such as "m:n", represent a range of locations where the associated OpenMP construct will be executed on. m and n are the lower bound and upper bound of the range. We consider the location clause as a hint, instead of a command for compiler implementation. It means that a compiler may ignore the clause without introducing correctness issue. Error conditions should be gracefully handled when the parameters of the clause are not in the range. For example, if a location number specified by programmer does not exist during runtime, e.g. location 4 is specified but there are only 2 locations for the execution, then these tasks specified running on the location will still be executed on a location in a round-robin fashion.

2.1.2 Threads and Locations Mapping

The default mechanism to map threads to locations is by block distribution. For example, if we have 16 threads and 4 locations, then the first location holds threads 0-3, the second location has threads 4-7, and so on. The block fashion maps threads with locations compactly to increase the data access locality. We also define the cyclic mechanism to map threads with locations, which distributes threads in a scatter fashion that can be used in the case of increasing memory access bandwidth. A user can modify the mapping rule by calling the omp_location_policy ([BLOCK, CYCLIC]) runtime routine. If CYCLIC is specified, the threads will be mapped with locations in a cyclic fashion, i.e. threads 0, 4, 8 and 12 are placed on location 0, thread 1, 5, 9 and 13 are placed on location 1, and so on in the above example. The location inheritance rule for parallel regions and tasks without the ”location” clause is hierarchical, that is, it is inherited from the parent in term of nested parallelism. In the beginning of a program execution, the default location association is to the entire collection of locations. Thus, when there is no location associated with a top-level parallel region, the parallel region will be executed across all locations in a block distribution fashion for all threads if possible. For nested parallelism, the inner parallel region will start by default at the same location where its parent thread is associated. If a task has been assigned to a particular location, all of its child tasks will be running on the same location if no other location is specified. On the contrary, if a location is specified to one of its child tasks, the task will be executed on the specified location.

2.1.3 Defining Data Layout

With the concept of location, we can further introduce a mechanism to express data layout into OpenMP. The goal of this feature is to allow OpenMP programmers to control and manage the data layout, and to map it with hierarchical memory systems. We borrow the data distribution syntax from SGI to express data layout as a directive in OpenMP as follows.

#pragma omp distribute (dist-type-list: variable-list) [location(m:n)]

”dist-type-list” is a comma-separated list of distribution types for the corresponding array dimensions in the variable-list. Variables in the variable-list should have the same dimension that matches with the number of distribution types listed. Possible distribution types include ”BLOCK” for a block distribution across a list of locations given in the location clause, and ”*” for non-distributed dimension. The symbol ”*” means that the indicated dimension remains, while ”BLOCK” indicates that the indicated dimension will be distributed across a set of locations. The location clause indicates a set of locations for the data to be distributed. If location is not present, it means for the entire set of locations. We only introduce the block data distribution at this time, and will consider other types of data distribution and impacts in the future. The distributed data still keeps its global address and is accessed in the same way as to other shared data. If no data distribution is specified for a shared variable, it is allocated in the shared memory space and it follows the current OpenMP implementation, mostly likely following the first-touch policy for data locality. The only difference between distributed data and non-distributed shared data is that user controls the physical locations of the distributed data so as to improve data locality in OpenMP programs.

2.1.4 Mapping tasks with Locations

To achieve greater control of task-data affinity, we can map OpenMP implicit tasks (from parallel region) and explicit tasks to locations based on either the location number or the association with distributed data. In this section, we introduce the syntax of mapping OpenMP work sharing and tasks with locations. To map a work sharing construct or a task with a location, one can simply specify the location number using the ”location” clause. However, it is much more intuitive to use a distributed data element location to determine where to run a task, instead of using the location number directly. For this purpose, we define the ”OnLoc” clause that maps a task with specified data, i.e. it assigns a task to a location where the specified data is located. Only distributed variables are allowed to be in the ”OnLoc” clause. The variable can be either an entire array for the parallel construct or an array element for the task construct. The following example illustrates how to use a distributed array A as an indication to schedule a parallel loop by considering the data layout over the location. The implicit tasks generated in the parallel loop will be executed on a list of locations where the variable A is distributed to, and be scheduled according to where data is located. For example, if A[0] is located on location #1, then the iteration i=0 will be executed on location #1 too.

```
#pragma omp parallel for OnLoc(A[i])
    for(i=0;i<N;i++){
        foo(A[i])
    }
```

Figure 1: Example: A code snippet using the proposed location feature

The ”OnLoc” clause changes the original OpenMP scheduling by introducing additional factor. The ”OnLoc” clause determines how iterations are distributed to different locations, while inside each location, the original OpenMP scheduling applies. For example, in the above example, the iteration space is determined first for each location, and these iterations will be further distributed to multiple threads (if any) bound to a location using static scheduling. The following example illustrates how to map a task to a location where A[i] is located. In this case, it is the programmer’s responsibility to define where the task will be executed by specifying the location where A[i] is stored.

```
for(i=0;i<N;i++){
    #pragma omp task OnLoc(A[i])
    foo(A[i])
}
```

Figure 2: Example: A code snippet using the proposed location feature

Compared to the location clause, ”OnLoc” maps a task or tasks to a location or a set of locations based upon where a variable is located or distributed. The location clause uses explicit location number(s), while the ”OnLoc” clause derives location information from the distribution of a variable, which is more closely related to the task-data affinity.

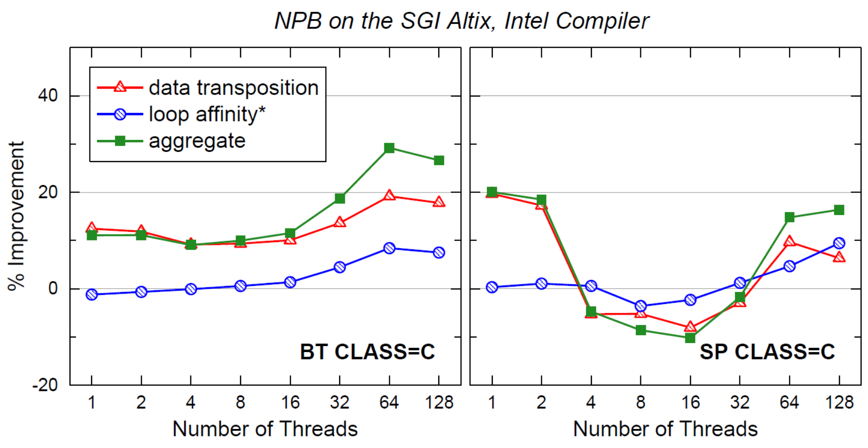


Figure 3: Performance comparison on the SGI Altix using the Intel compiler

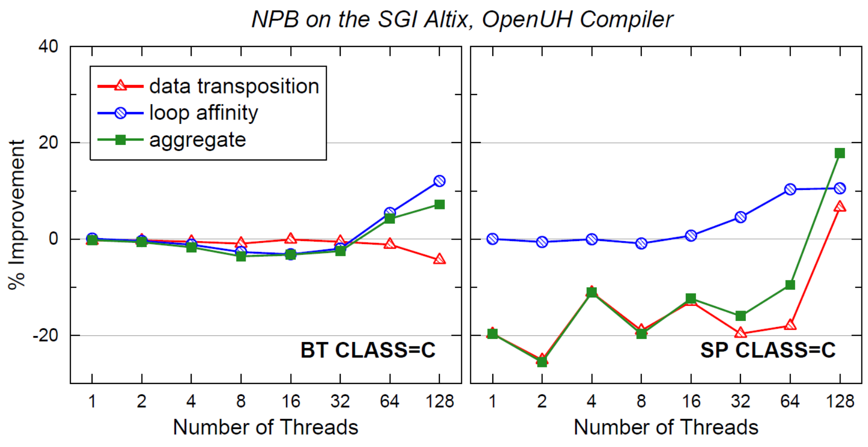


Figure 4: Performance comparison on the SGI Altix using the OpenUH compiler

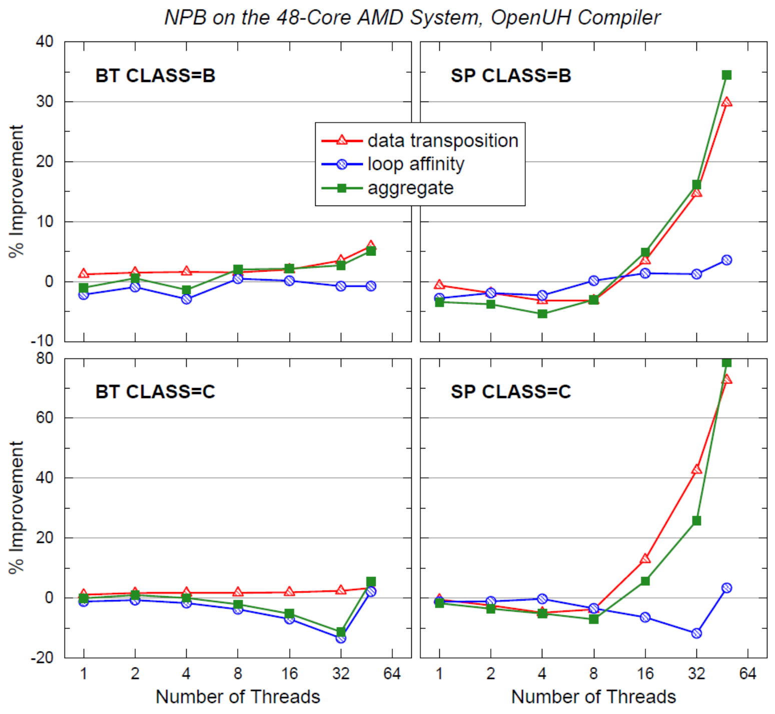


Figure 5: Performance comparison on the 48-core AMD system using the OpenUH compiler

2.1.5 Performance Study

We have tested the implementation on an SGI Altix NUMA system (part of the NASA Columbia supercomputer) and a 48-core AMD workstation for two selected NAS Parallel Benchmarks (NPB) (BT and SP). The OpenMP versions of NPB3.3 is used as a baseline for performance comparison. We created two versions: the first one applied the data transposing without the OnLoc clause (data-transposition), and the second one applied the distribute + OnLoc as described (loop-affinity). The OpenUH compiler was installed on both the SGI Altix and the 48-core AMD system. For additional comparison we also manually created a loop-affinity version to mimic the scheduling of loop iterations to threads based on where the data resides. Then use the Intel compiler to compile the translated code.

Figures 3, 4, 5 show the percentage performance improvement of the new versions over the baseline version for the Class B and C problems at various thread counts. The "aggregate" values in the figures are those accumulated from the two components. Negative values indicate performance degradation. On the SGI Altix using the Intel compiler, we observe as much as 10% performance improvements at large thread counts from loop affinity. Data transposition improves the BT performance by 10-20%, but has variable effects on SP. This seems to be related to the balance between the cost of extra data copies and the improvement from better data alignment and cache access in the computation. The results using the OpenUH compiler (see Figure 4) show performance improvement at large thread counts (32) from loop affinity. However, we observe substantial performance degradation (20%) from data transposition for SP and no improvement for BT.

On the 48-core AMD system (Figure 5), there is no performance gain from applying loop affinity; in fact, negative effects are observed for the Class C problem. Such results are somewhat counter-intuitive. One possible explanation is that the current OpenUH runtime is experimental and may not handle data and thread binding optimally. On the other hand, we do observe performance improvement from applying data transposition for both BT and SP. The improvement for BT is less than 5%, but for SP it increases substantially when the number of threads is larger than 8. The larger problem (Class C) exhibits close to 80% performance improvement over the baseline version at 48 threads.

From the experiments, we observe significant performance impact from different data layouts on the NUMA system, especially for larger data sets. The notion of data layouts via distribution and affinity with loop iterations via OnLoc allows a user to carefully optimize data layout with the data access pattern and, thus, achieve performance gain on large NUMA systems. We intend to carefully evaluate our language enhancements and the end-to-end implementation of these features in the final project year, in which we will also contribute our findings to the standardization effort that was recently initiated.

2.2 OpenMP 3.0 Compiler Implementation

In order to fully and accurately support the OpenMP 3.1 specification, we have been actively working on both the compiler support and runtime implementation. Specif-

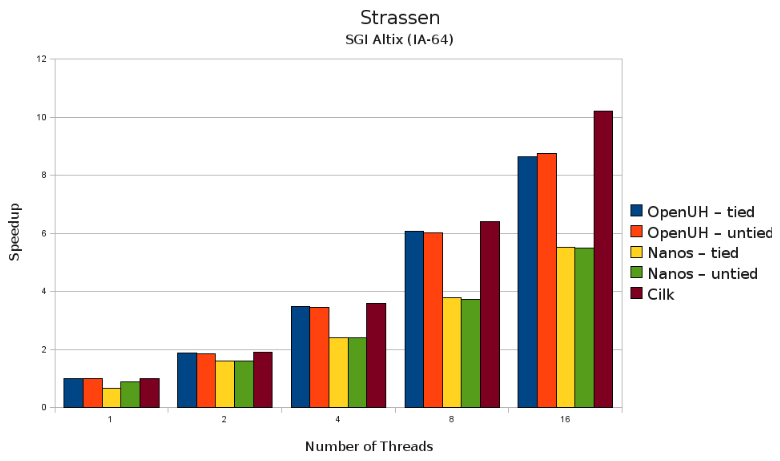


Figure 6: Speedup with tasking runtime on SGI Altix 350 system

ically, we have completed our translation of OpenMP tasks in the OpenUH compiler and have focused on improvements to the task scheduler. Additionally, we have implemented a configurable task pool framework that allows the user to choose at runtime which various task pool organizations to employ. We developed a preliminary design for an extension to OpenMP for grouping explicit tasks. We have added to the OpenUH runtime a new synchronization mechanism similar to the clock in X10 and the ”phaser” in the X10 variant Habanero-Java.

We completed our translation of OpenMP tasks in the OpenUH compiler and have focused on improvements to the task scheduler. The initial speedup results from our tasking runtime were favorable. Figure 6 shows the results from the Strassen benchmark comparing our tasking runtime with the Nanos runtime and Cilk on an SGI Altix 350 consisting of eight nodes. Each node is an SMP with two Itanium2 processors running at 1.6 GHz with 16GB of main memory (128 GB total). All implementations were compiled with GCC 4.2.3 using -O2 optimization levels. In all of our tests, our runtime performs as well as or better than Nanos, and in some cases it performs better than Cilk. Recently, we added full support for nested parallel regions and also revamped the task implementation after determining that the use of the Portable Coroutines Library incurred more overhead than what is necessary. The use of coroutines provided more scheduling flexibility since tasks can easily be switched from one thread to another. Thus it provides a useful mechanism for supporting untied task migration, a feature which to our knowledge is not well supported in the major vendor implementations. The downside is that creating a coroutine with its own stack (64K by default) for every task was very expensive, and this would more often than not offset its benefits. Removing this overhead resulted in significant (often an order of magnitude) improvements in execution times. We are currently investigating more efficient mechanisms for migrating untied tasks.

Recent performance results for our tasking runtime are shown in Figure 7 using

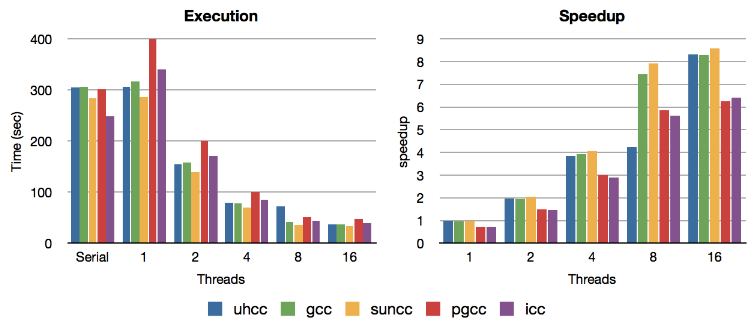


Figure 7: Performance of NPB BT-MZ with tasking runtime on dual Nehalem E5520 machine

a version of the NAS Parallel Benchmarks BT-MZ implemented with OpenMP tasks. Results are taken from a system with dual 2.27 GHz Nehalem E5520 and 32 GB memory capable of 16 threads. Each core has 32KB L1 and 256KB L2 caches with each processor sharing 8MB L3 cache. The benchmark was compiled with both commercial and open source compilers. The following optimization flags were used: for our Open64-based uhcc compiler, -O2 -LNO; GNU C compiler gcc 4.6.1, -O3 -fargument-noalias-global; Oracle's suncc 5.11, -xO3; PGI's pgcc 11.7, -fast; and Intel's icc 12.0.0, -O3 -fno-alias.

Additionally, we have implemented a configurable task pool framework that allows the user to choose at runtime which specific task pool organization to employ. We currently have four different task pools implemented that utilize distributed, hierarchical, and hybrid queue organizations. Each of these may impact task creation, task scheduling, or both. This has provided a lighter weight tasking implementation and easy experimentation of the impacts of using various the task pool organizations with a given application. This framework also allows a quick implementation of new task pool designs. Furthermore, the user may control the order in which tasks are removed from a task queue for greater control over task scheduling. For most implementations we reviewed, tasks are generally removed from queues in LIFO order (though when "work-stealing" it occur in FIFO order). This results in what is effectively a depth-first scheduler, and it appears to be a good default option as it works well for codes exhibiting data locality. However, we found that for some codes (e.g. the Fibonacci, Floorplan, and NQueens kernels) where data locality isn't as much a concern, it is best to employ a breadth-first scheduler (i.e. tasks are always removed in FIFO order). Some of this work was presented and published in the proceedings of the 2011 International Workshop on OpenMP (IWOMP) as "A Runtime Implementation of OpenMP Tasks".

We developed a preliminary design for an extension to OpenMP for grouping explicit tasks. In the current report period, we changed the initial design to provide task-to-task synchronization as a way of communicating among tasks rather than explicitly grouping them. We have added to the OpenUH runtime a new synchronization mechanism similar to the clock in X10 and the ”phaser” in the X10 variant Habanero-Java. This would allow point-to-point synchronizations and aid in the implementation of reduction operations on task groups. This provides the ability to form implicit groupings of task as well as more flexible synchronization. We will complete this work by using this extension with NASA applications, including NASA’s NPB MT-LU, with pipelining and wavefront execution to test whether it is flexible and not error-prone.

We are collaborating with researchers at Tsinghua University to provide a complete implementation of OpenMP 3.0 in the standard version of the Open64 compiler. Based upon an evaluation of available runtime software as part of this effort, the OpenUH support for tasking was determined to have the highest quality of those evaluated and it is therefore being used as the basis for this compiler suite, which will be supplied to a broad range of vendors (e.g. AMD, HP, Google, Absoft, Qualcomm) in addition to research groups worldwide to form the basis for their OpenMP 3.0 implementations and experimentation. We have merged recent improvements in our runtime (viz. nested parallelism and improved tasking implementation) into the official OpenMP3.0 branch in the Open64.net subversion repository. The OpenMP Architecture Review Board updated the OpenMP specification in July 2011 to version 3.1 and we intend to begin its implementation in the OpenUH compiler and runtime in the near future.

2.3 OpenMP Validation Suite

In this work, the goal is to build an efficient framework, i.e. a testing environment, that will be used to validate the OpenMP implementations in OpenMP compilers. With the introduction of new versions of OpenMP, there is an absolute need to check for completeness and correctness of the OpenMP implementation. We need to create an effective testing environment in order to achieve this goal. In prior work, we collaborated with colleagues at University of Stuttgart, to create validation methodologies for OpenMP versions 2.0 and 2.5 reported in [16, 18] respectively. We have built our current framework on top of the older one. We have improved the testing environment and now the OpenMP validation testsuite covers all tests for the directives and clauses in OpenMP 3.1. This testing interface is portable, flexible and offers an user-friendly framework that can be tailored to accommodate specific testing requirements. Tests could be easily added/removed adhering to the changes in the OpenMP specification in the future. In our current work we have ensured that the bugs in the previous validation testsuite have been fixed.

2.4 Design of OpenMP Validation Suite

The basic idea to design the OpenMP validation suite is to provide short unit tests wherever possible and check if the directive being tested has been implemented correctly. For instance, the `parallel` construct and its corresponding clauses such as `shared` are tested for correctness. A test will fail if the corresponding feature has not been implemented correctly. We defined such typical tests to be *normal tests*.

In a given code base, there might be more than a few directives being used at a given time. However, it is a challenge to check for correctness for a particular directive

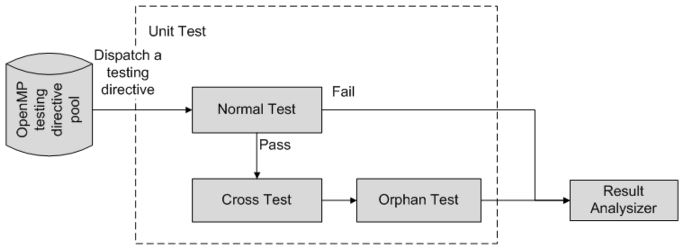


Figure 8: The OpenMP Validation Suite Framework.

of interest, for instance `loop`, among several others. To solve this issue, we perform another test methodology called *cross test*, to validate only the directive under consideration. If this directive is removed from the code base, the output of the code will be incorrect.

Besides, we also need to ensure that the directive is serving its purpose. For instance lets consider a variable declared as `shared`. We also know that the variable is `shared` by default irrespective of explicitly declaring it as `shared`. Let us replace the `shared` with a `private` clause or any other clause which does not contain the functionality of the directive being tested, which in this case is `shared`. As a result, the *cross test* will check for the output result, which has to be incorrect because the variable is no longer being `shared`.

Moreover, in order to determine if the directive being tested is capable of correct execution when ”orphaned” from the main function, we created a new test methodology named as *orphan test*. In the *orphan test*, the directive being tested is placed into a children procedure which is called by the main function.

All test results will be statistically analyzed. Each test will be repeated multiple times. The purpose of this is to ensure that the test fails if the directive being tested does not function as required. In order to estimate the probability that a test is passed accidentally we take the following approach: if n_f is the number of failed cross tests and M the total number of iterations, the probability of that test will fail is $p = \frac{n_f}{M}$. Thus the probability that an incorrect implementation passes the test is $p_a = (1-p)^M$, and the certainty of test is $p_c = 1 - p_a$, which means the probability that a directive is validated.

Currently the validation suite contains more than 70 unit tests covering all of the clauses in the OpenMP version 3.1 release. Each of the unit tests has three types of tests: *normal, cross*, and *orphan test*. One challenge is, however, if we implement each of them separately, the whole suite will be ad-hoc and error-prone. It would be also challenging to manage and analyze the results generated out of so many tests. So we created an execution environment that will manage these several tests methodically.

Figure 8 shows the proposed framework i.e. the execution environment of the OpenMP validation suite. In this framework, we create a testing directive pool that will consist of templates for the unit tests for each of the OpenMP directive that is being tested. This framework has been developed mainly using the Perl scripting language. We use this framework to parse through the several templates that have been written for each of the OpenMP directive. Executing this framework will deliver the source code for the three types of tests, namely *normal*, *cross* and *orphan* tests. The *normal* tests will be the first test to be performed in this process. If this particular test fails then there is no need to perform the *cross* and *orphan* tests. As a result, the corresponding source codes for *cross* and *orphan* tests will not be generated. This has been carefully crafted into our framework. If the *normal* test passes successfully, the framework will automatically generate source codes for the other two tests. Note that we had to create only one template in order to generate source codes for all the three types of tests. As a result we emphasize that the framework adopts an automatic approach while creating the different kinds of tests necessary to check the correctness of the directives. There is very little manual labor involved. Once these different tests have been created, our framework will compile and execute them as and when necessary. There is also a result analyzer component as part of the framework that will collect the results from each of the unit tests once all of them have completed execution. These results will be in the form of log files and the analyzer component will help in generating a complete report in a user-friendly manner.

The advantages that the execution environment offers are as follows:

- Creates one template for each test that suffices to automatically generate source codes for the three types of tests, i.e normal, cross and orphan tests.
- Creates bug reports that consist of adequate information about the compiler being used for testing purposes. The report will consist of version number of the compiler, build and configuration options, optimization flags used, and so on.
- Launches all the tests automatically, although individual tests will be performed only for those directives that are being tested.
- Generates reports that are easy to read and understand. These user-friendly reports will contain information about the bugs identified. The details of the compilation and execution are also provided.

The framework is easy to use and maintain. It is quite flexible enough to accommodate changes as and when OpenMP specification gets updated with newer features.

2.5 Implementations

In this section we discuss the basic idea of each unit test OpenMP directive and clause. The previous publications [15, 18] presented the unit tests for the constructs in OpenMP version 2.5. Hence we restrict the discussions to the unit tests for the newer features in OpenMP version 3.1. We have inserted code snippets for only few of the directive and clauses due to space constraints.

2.5.1 Directives and Clauses

`Task` is a new construct in OpenMP 3.0. It provides a mechanism to create explicit tasks. Tasks could be executed immediately or delayed by any assigned thread. Figure 9 shows the test for OpenMP `task` construct. The basic idea is to generate a set of tasks by a single thread and execute them in a parallel region. The tasks should be executed on more than one threads. In the cross test, the `task` *pragma* is removed. As a result, every task is executed only by one thread since the tasks are in the `single` region hence giving incorrect output.

```
int test_omp_task(){
 int tids[NUM_TASKS];
 int i, result=0;

 #pragma omp parallel
 {
  #pragma omp single
  {
   for (i = 0; i < NUM_TASKS; i++){
    int myi = i;
    #pragma omp task
    {
     sleep (SLEEPTIME);
     tids[myi]=omp_get_thread_num();
    } /* end of omp task */
   } /* end of for */
  } /* end of single */
 } /*end of parallel */

 /*now check for results*/
 for (i = 0; i < NUM_TASKS; i++){
  if (tids[0] != tids[i])
   result = 1;
 }
 return result;
} /* end of test */
```

Figure 9: Test for `task` construct.

```
int test_omp_taskwait(){
 int i, result = 0;
 int array[NUM_TASKS];/* omit init*/

 #pragma omp parallel
 {
  #pragma omp single
  {
   for (i = 0; i < NUM_TASKS; i++){
    int myi = i;
    #pragma omp task
    array[myi] = 1;
   } /* end of for */
#pragma omp taskwait

   /*check for all tasks finish*/
   for (i = 0; i < NUM_TASKS; i++){
    if (array[i] != 1)
     result++;
   } /*end of for*/
  } /* end of single */
 } /*end of parallel */
 /*check result is correct*/
 return (result == 0);
}/*end of test*/
```

Figure 10: Test for `taskwait` construct.

The `taskwait` construct specifies a synchronization point where the current task is suspended until all children tasks have completed. Figure 10 shows the code listing for testing the `taskwait` construct. A flag is set to each element of an array when a set of tasks are generated. If `taskwait` executes successfully, all elements in the array should be 1; otherwise, the elements should have some other values. In the cross test, we remove the `taskwait` construct and check the value of elements in the array. Obviously, the value will be arbitrary if there is no ”barrier” at the completion of tasks. Consequently, it is able to validate the `taskwait`construct.

The `shared` clause defines a set of variables that could be shared by threads in `parallel` construct or shared by tasks in `task` construct. The basic idea to test it is to update a shared variable i.e. *i* by a set of tasks and check whether it could be shared by all tasks. If this is the case, the value of the shared variable should be equal to number of tasks. In the cross test, we check if the result is wrong without the `shared` clause. `Shared` is replaced by the `firstprivate` clause, i.e., the attribute of *i* is changed to firstprivate. As a result, the value of *i* should be incorrect.

As opposed to `shared` clause, the `private` clause defines that variables are private to each task or thread. The idea of testing for the `private` clause is first to generate a set of tasks as before and each task update a private variable, e.g., *local_sum*. We compare the value with the *known_sum* which is calculated in prior. In the cross test, we remove the `private` clause from `task` construct. Thus the private variable now becomes `shared` by default. As a result, we see that the value of `local_sum` should be incorrect.

The `firstprivate` clause is similar with `private` except that the new item list has been initialized prior to the `task` construct encountered. As a result, in contrast to `private` clause, we do not need to initialize variables declared as `firstprivate` attribute. The initialized value follows immediately prior to the `task` construct. Consequently, test for the `firstprivate` is similar as the test for `private` clause except that variable `local_sum` do not need to be initialized to zero in the `task` region. In the cross test, the `firstprivate` is removed thus the variable `local_sum` becomes `shared` again.

The `default` clause determines the data-sharing attributes of variables implicitly. In C language, the variables declared as `default` is `shared`, while in Fortran from OpenMP 3.0, it allows variables declared as `private` or `firstprivate` by default. In addition, OpenMP 3.0 also allows variables does not have any predetermined data-sharing attribute declared as `none`. As a result, the idea of testing for `default` clause is actually the same as the test for `shared` clause in C and `firstprivate`, `private` in Fortran. The `if` clause controls the `task` implementation as shown in

```
int test_omp_task_if(){
 int count, result=0;
 int cond_false=0;

 #pragma omp parallel
 {
  #pragma omp single
  {
   #pragma omp task if (cond_false)
   {
    sleep (SLEEPTIME_LONG);
    result = (0 == count);
   } /* end of omp task */

   count = 1;
   #pragma omp flush (count)
  } /* end of single */
 } /*end of parallel */
 return result;
} /*end of test*/
```

Figure 11: Test for `if` clause.

```
int omp_for_collapse(){
 int is_larger = 1;
 #pragma omp parallel
 {
  int i,j,my_islarger = 1;
  #pragma omp for schedule(static,1)
  collapse(2) ordered
   for (i = 1; i < 100; i++)
    for (j =1; j <100; j++)
    {
     #pragma omp ordered
       my_islarger = my_islarger &&
         check_i_islarger(i);
    } /* end of for */
  #pragma omp critical
   is_larger=is_larger &&
       my_islarger;
 } /*end of parallel*/
 return (is_larger);
} /*end of test*/
```

Figure 12: Test for `collapse` clause.

Figure 11. If the `if` is evaluated as false then the encountering task will be suspended and a new task is executed immediately. The suspended task will be resumed until the generated task is finished. The idea of testing the `if` clause is to generate a set tasks by a single thread and pause it immediately. The parent thread shall set a counter variable that the paused task will consider when the thread wakes up. If the `if` clause is evaluated to false, the `task` region will be suspended and the counter variable *count* will

be assigned to 1. When the `task` region resumes, we evaluate the value of the counter variable *count*. In the cross test, we removed `if` clause from the `task` construct, since `if` is evaluated to true by default, the `task` region will be executed immediately and the counter variable `count` will still 0.

In OpenMP 3.0, task is executed by a thread of the team generated it and is tied by default,i.e., tied tasks are executed by the same thread after the suspension. If the `untied` clause is presented, any thread could resume the task after the suspension. Thus, the idea of testing the `untied` clause is shown as in figure 13. First we create a set of tasks in parallel region and save the thread id executed each task. Then we suspend all the tasks by `taskwait`. We send half of the threads into a busy loop so that at least half of the other idle threads could be rescheduled to the suspended tasks. We compare the thread number before and after the suspension. Since task is untied, tasks could be rescheduled by different threads after the suspension. In the cross test, the `untied` clause is removed so that tasks are tied with the execution thread by default. As a result, the thread number before and after the task suspension should be the same leading to incorrect result.

Besides the `tasking` model, OpenMP 3.0 defines a new `collapse` clause for the `loop` construct that handles perfectly nested multi-dimensional loops. This clause collapses the loops, it is associated with, into one single loop. and controls the number of loops associated with one larger loop. The order of iterations in the collapsed loop is determined by the order of iterations in all loops before the collapse. If no `collapse` clause specified, the only loop that is immediately followed by the `loop` construct is associated.

Figure 12 shows the basic idea of testing for the `collapse` clause which binds the two loops together. With the `ordered` clause, both *i* and *j* loops should be executed in order, thus the variable *my_islarger* should be TRUE. In the cross test, since the `collapse` clause is removed, the only loop that is associated with the `loop` construct is the *i* loop, the one that immediately follows the construct which should be executed in parallel and the only *j* loop will be executed in order. Consequently, the result will be incorrect.

2.5.2 Support for OpenMP 3.1

OpenMP version 3.1 was released in July 2011, a refined and extended version of OpenMP 3.0. The `taskyield` construct defines an explicit scheduling point that the current task is suspended and switched to a different task in the team. The test for the `taskyield` construct is similar to the test for `untied` clause, except for the `taskwait` begin replaced by `taskyield`.

The OpenMP 3.1 also provides a new feature to reduce the task generation overhead by `final` and `mergeable` clause. If the expression in `final` clause is evaluated to true, the task that is generated will be the final task and no further tasks will be generated. Consequently, it reduces the overheads of generating new tasks, especially in recursive computations such as in *Fibonacci* series when the *Fibonacci* numbers are too small. Test for the `final` clause is showed in Figure 14. The idea is to set a threshold that if task number is larger to the threshold, the task will be the final task. We save the *task id* to check whether the task larger than the threshold is executed by

```
int omp_task_untied(){
 int init_tid[NUM_TASKS];
 int curr_tid[NUM_TASKS];
 int i, count=0;
 #pragma omp parallel
 {
  #pragma omp single
  {
   for (i = 0; i < NUM_TASKS; i++){
    int myi = i;
    #pragma omp task untied
    {
     init_tid[myi]=omp_get_thread_num();
     #pragma omp taskwait
     if ((init_tid[myi]%2) == 0){
      sleep (SLEEPTIME);
      curr_tid[myi]=omp_get_thread_num();
     } /*end of if*/
    } /* end of omp task */
   } /* end of for */
  } /* end of single */
 } /* end of parallel */
 for(i=0;i<NUM_TASKS;i++){
  if(curr_tid[i]!=init_tid[i])
   count++;
 } /*end of for*/
 return count;
} /*end of test*/
```

Figure 13: Test for `untied` clause.

```
int test_omp_task(){
 int tids[NUM_TASKS];
 int i, error=0;

 #pragma omp parallel
 {
  #pragma omp single
  {
   for (i = 0; i < NUM_TASKS; i++){
    int myi = i;
    #pragma omp task final(myi>=THRESH)
    {
     sleep (SLEEPTIME);
     tids[myi]=
        omp_get_thread_num();
    } /* end of omp task */
   } /* end of for */
  } /* end of single */
 } /*end of parallel */

 /*check tid beyond thresh*/
 for (i =THRESH;i < NUM_TASKS;i++)
 {
  if (tids[THRESH] != tids[i])
   error++;
 }

 return (error==0);
} /* end of code */
```

Figure 14: Test for `final` clause.

the same task. Test for the `final` clause is showed in The idea is to set a threshold that if task number is larger to the threshold, the task will be the final task. We save the *task id* to check whether task larger than the threshold is executed by the same task.

In OpenMP 3.1, the `atomic` is refined to include the `read, write, update,` and `capture` clauses. The `read` with the construct `atomic` guarantee an atomic read operation in the region. For instance `x` is read atomically if `v=x`. Similarly, the `write` forces an atomic write operation. It is much more lightweight using `read` or `write` separately than just using `critical`. The `update` clause forces an atomic update of an variable, such as *i++, i–*. If no clause is presented at the `atomic` construct, the semantics are equivalent to atomic update. The `capture` clause ensures an atomic update of an variable that also captures the intermediate or final value of the variable. For example, if `capture` clause is present then in `v = x++`, `x` is atomically updated while the value is captured by `v`.

OpenMP 3.1 also extends the `reduction` clause to add two more operators: `max` and `min`, which is to find the largest and smallest values in the reduction list respectively. Since the algorithms to find maximum or minimum number is quite straightforward, we do not discuss it here.

2.5.3 Evaluations

In this section, we will evaluate the OpenMP validation suite on several open source and vendor compilers, including OpenUH, GNU C and Intel C/C++.

	OpenUH				GNU				Intel			
Directive	N	C	O	OC	N	C	O	OC	N	C	O	OC
task	100	100	100	100	100	100	100	100	100	100	100	100
taskwait	100	100	100	100	100	100	100	100	100	100	100	100
task_shared	100	100	100	100	100	100	100	100	100	100	100	100
task_private	100	100	100	100	100	100	100	100	100	100	100	100
task_firstprivate	100	100	100	100	100	100	100	100	100	100	100	100
task_if	100	100	100	100	100	100	100	100	100	100	100	100
task_untied	100	100	100	100	100	100	100	100	100	100	100	100
task_default	100	100	100	100	100	100	100	100	100	100	100	100
for_collapse	100	100	100	100	0	-	0	-	100	100	100	100

Table 1: Experimental results fragment on various compilers.

The first experiment is to evaluate the OpenMP validation suite using our in-house OpenUH compiler [20, 14], testing the implementation correctness on OpenMP. As of now OpenUH supports OpenMP version 3.0, we did not evaluate the tests for OpenMP 3.1. OpenUH compiler is a branch of the open-source Open64 compiler suite for C, C++, Fortran 95/2003, with support for a variety of targets including x86 64, IA-64, and IA-32. It is able to translate OpenMP 3.0, Co-array Fortran, UPC, and also translates CUDA into PTX format. An OpenMP implementation translates OpenMP directives into corresponding POSIX thread code with the support of runtime libraries.

The experiments were performed on a Quad dual-core Opteron-880 machine and we used eight threads during the evaluations. We also disabled all the optimizations flags in order to avoid any potential uncertainties, e.g.code reconstruction while compiling the unit tests. The version of GNU compiler is 4.6.2, Intel C/C++ is 12.0.

Table 1 shows the experimental results on several compilers. For each sub-column, N is normal test, C is cross test, O is orphan test while OC is orphan cross test (the cross test within orphan test). Each row is the tested directive which is namely straightforward. For instance, the `para_shared` is to test the `shared` clause in the `parallel` construct. The certainty of each directive passed the test is calculated according to the statistics explained earlier. 100% means that directive is verified with 100% certainty.

From the experiment results it is not a surprise to see that most of tests passed with 100% certainty, which is because compilers need to be under strict tests before release. However, we could still see that the GNU C compiler fails the `collapse` test.

2.6 Co-Array Fortran Implementation and Experiments

Coarray Fortran (CAF) is a PGAS Fortran extension which has been incorporated into the Fortran 2008 standard. It enables parallel programming in Fortran with minimal change to the language syntax. In a joint project between UH and Total, we have investigated CAF as a viable programming model for production Oil and Gas applications. In contrast to other open-source implementation efforts [6, 3, 17], our approach is to use a single, unified compiler infrastructure to translate, optimize and generate binaries from CAF codes. CAF support in OpenUH [7] comprises three areas: (1) an extended front-end accept the coarray syntax and related intrinsic functions, (2) back-end optimization and translation, and (3) a portable runtime library (Figure 15).

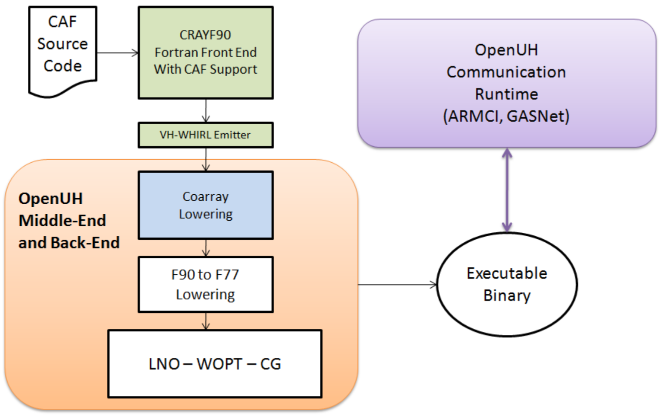

CAF Source Code
CRAYF90 Fortran Front End With CAF Support
VH-WHIRL Emitter
OpenUH Middle-End and Back-End
Coarray Lowering
F90 to F77 Lowering
LNO – WOPT – CG
OpenUH Communication Runtime (ARMCI, GASNet)
Executable Binary

Figure 15: Framework of OpenUH CAF Compiler/Runtime System

2.6.1 Co-Array Fortran Compiler

Front-end We modified the Cray Fortran 95 front-end that comes with OpenUH to support our coarray implementation. Cray had provided some support for CAF syntax, but its approach was to perform the translation to the underlying runtime library in the front-end. It accepted the `[]` syntax in the parser, recognized certain CAF intrinsics, and it targeted a SHMEM-based runtime with a global address space. In order to take advantage of the analysis and optimizing capabilities in the OpenUH back-end, we needed to preserve the coarray semantics into the back-end. To accomplish this, we adopted a similar approach to that used in Open64/SL Fortran front-end from [6], where co-subscripts are preserved in the IR as extra array subscripts. We also added support for CAF intrinsic functions such as `this_image`, `num_images`, `image_index`, and more as defined in the Fortran 2008 standard.

Back-end We have in place a basic implementation for coarray lowering in our back-end and are in the midst of adding an analysis/optimization phase. The current implementation will generate communication based on remote coarray references. Suppose the Coarray Lowering phase encounters the following statement:

$$\begin{aligned} A(i,j,1:n)[q] =& B(1,j,1:n)[p] \\ &+ C(1,j,1:n)[p] + D[p] \end{aligned} \quad (1)$$

This means that array sections from coarrays B and C and the coarray scalar D are brought in from image p. They are added together, following the normal rules for array addition under Fortran 90. Then, the resulting array is written to an array section of coarray A on process q. To store all the intermediate values used for communication, temporary buffers must be made available. Our translation creates 4 buffers t1, t2, t3, and t4 for the above statement. We can represent this statement in the following way:

$$\begin{aligned} A(i,j,1:n)[q] \leftarrow t1 = & t2 \leftarrow B(1,j,1:n)[p] \\ & +t3 \leftarrow C(i,j,1:n)[p] \\ & +t4 \leftarrow D[p] \end{aligned} \tag{2}$$

For each expression of the form $t \leftarrow R(...)[...]$, the compiler generates an allocation for a *local communicatio buffer* (LCB) t of the same size as the array section $R(...)$. The compiler then generates a `GET` runtime call. This call will retrieve the data into the buffer t using an underlying communication subsystem (either ARMCI or GASNet, as specified by the user). The final step is for the compiler to generate a deallocation for buffer t. An expression of the form $L(...)[...] \leftarrow t$ follows a similar pattern, except the compiler generates a `PUT` runtime call.

```
! omitted creation and
! initialization of dope vectors
GET( t2, B(1, j, 1:n), [p] )
GET( t3, C(i, j, 1:n), [p] )
GET( t4, D, [p] )
t1 = t2 + t3 + t4
PUT( t1, A(i, j, 1:n), [q] )
```

The above pseudo-code depicts the communication pattern generated for the statement representation given in (2). Currently, all generated communication is blocking, and we have not yet implemented optimizations for the buffering.

Fairly early in the back-end processing, a F90 lowering phase is carried out in which F90-supported elemental array operations are translated into loops. We make use of the higher-level F90 array operations, supported by the *very high WHIRL* IR in our compiler, for generating block communication in our translation. The implemented translation strategy is as follows:

1. **Lower CAF Intrinsics:** Calls to `this_image` and `num_images` are replaced with loads of external symbols representing the runtime-initialized variables `_this_image` and `_num_images`, respectively.

2. **Lower Co-indexed References:** A co-indexed coarray variable signifies a remote access. ARRAY and ARRAYSECTION nodes in the compiler IR are processed to determine if they represent a co-indexed array reference. A temporary *local communication buffer* (LCB) is allocated for either sending (if its a write) or receiving (if its read) the accessed elements.

3. **Symbol Table Cleanup:** After coarrays are lowered, their corresponding *type* in the WHIRL symbol tables are adjusted so that they only contain the local array dimensions.

One of the key benefits of the CAF programming model is that programs are amenable to aggressive compiler optimizations. The back-end also consists of a prelowering phase which normalizes the IR emitted from the front-end to facilitate dependence analysis. This will enable many optimizations, including hoisting potentially expensive coarray accesses out of loops and generating non-blocking communication calls where it is feasible and profitable.

2.6.2 Co-Array Fortran Runtime

The implementation of our supporting runtime system relies on an underlying communication subsystems provided by ARMCI [19] or GASNet [4]. We have adopted both the ARMCI and GASNet libraries for most communication and synchronization operations required by the CAF execution model. This work entails memory management for coarray data, communication facilities provided by the runtime, and support for synchronizations specified in the CAF language. We have also added preliminary implementation of reductions in the runtime.

CAF lacks many of the features provided by MPI such as non-blocking communication. Since remote communication is a major performance bottleneck on distributed memory systems, the implementation is responsible for hiding latency by reducing communication or overlapping it with computation. We have implemented optimizations in the CAF runtime to address this. A get-cache is used to reduce the number of remote reads, and non-blocking prefetching is used to increase communication-computation overlap. To improve remote write performance, we make all remote writes automatically non-blocking.

2.6.3 Experiments and Preliminary Results

TOTAL performs seismic exploration to find oil both on land and beneath the sea. Sound energy waves are created on the surface using dynamites. Sound waves travel at different velocity in different kind of materials. The timings of the reflected waves are recorded using geophones and hydrophones. The timings are processed to create seismic profiles using different mathematical models. The programs that are used to evaluate our implementation's performance are part of this process.

The experiments are performed on a cluster of 330 compute nodes (2640 cores) which have a peak performance of 29.5 TFLOPS. Each node has 2 Intel Nehalem quad-core CPUs, with each core operating at a frequency of 2.8 GHz. The nodes are diskless and have 24GB memory. The interconnect is QDR Infiniband on 8X PCIe 2.0 in a fat tree topology. The upload and download bandwidth of the interconnect is 40Gbps. It uses a shared parallel file system.

The MPI version of the program are executed using Intel MPI version 12. MPI uses 2-sided non-blocking send and receive calls, MPI_ISEND and MPI_IRECV. The compiler flag '-fp-model precise' is used to ensure that floating point operations conform to IEEE standard. Compiler optimization level -O3 is used for both UHCAF and

MPI. The experiments do not evaluate performance on SMP, as only 1 process is run on 1 node. Evaluating SMP performance is out of the scope of these experiments as several other factors (like location of data in memory) have to be considered. Time is measured using the C function gettimeofday. All measurements are an average of 100 iterations. Only the GASNet version of UHCAF is used because ARMCI has some limitations (about 2GB) on the amount of memory that can be registered when using the Infiniband native API. Since the domain size of the programs are mostly greater than 2GB, ARMCI cannot be used.

Isotropic Forward Wave Equation This is the simplest case of the wave equation as the velocity of sound does not change in the isotropic medium. In this program, two 3-D matrices are used to calculate 2-way wave equation using scalar acoustic wave equation.

Table 2 shows the time taken (seconds) to execute the isotropic forward wave equation solver on 8 nodes. The computation timings are only for the kernel code, which does not include file I/O. The Intel MPI version is 12 and UHCAF uses GASNET. Both use optimization level O3. The buffer size is the total size of the data that is being communicated by each process. The domain size is the total size of the 3-D matrix. The 'compute' column contains the average of the computation time among all processes. The 'comm' column contains the average of the communication time among all processes. The 'Total' column contains the total time, which is same on all processes due to synchronization.

Table 2: Isotropic forward wave equation solver timings (sec) with 8 processes

Buffer	**Domain**	**Intel MPI**			**UHCAF**		
Size	**Size**	**Compute**	**Comm**	**Total**	**Compute**	**Comm**	**Total**
0.75M	64M	1.55	0.24	1.84	1.81	0.14	2.09
1.25M	128M	3.09	0.52	3.65	3.45	0.28	4.03
2M	256M	6.18	1.12	7.34	6.91	0.48	7.78
3M	512M	11.21	1.40	12.87	13.70	0.68	16.2
5M	1G	22.14	1.75	24.29	30.23	1.15	34.23
8M	2G	49.66	3.33	53.49	64.56	1.85	72.33
12M	4G	96.78	6.25	105.95	135.07	3.26	145.02
20M	8G	184.46	7.37	201.71	254.27	2.87	285.7
32M	16G	410.53	33.64	452.40	492.63	10.41	544.66

Even though the total execution time of the UHCAF version is more than the Intel MPI version (Figure 16), the communication time is significantly less as shown in the Figure 17.

Figure 18 shows the increase in performance more clearly. For 32M buffer UHCAF is three times faster than Intel MPI. This shows that UHCAF makes better utilization of one-sided RDMA capabilities provided by the Infiniband interconnect. Another interesting point is that MPI uses non-blocking communication while CAF uses blocking.

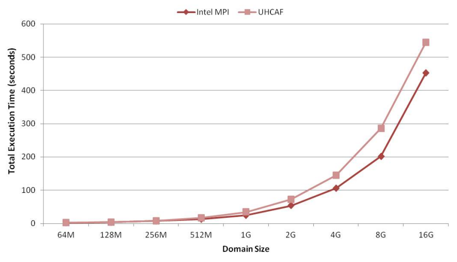


Figure 16: Kernel execution time (includes communication)

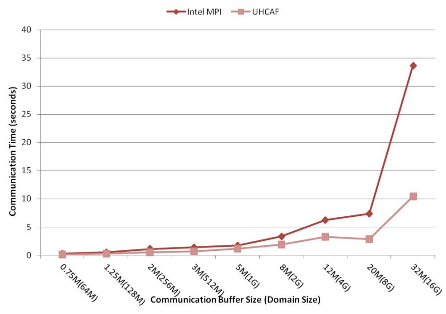


Figure 17: Communication time (seconds) with 8 processes

As discussed before, Infiniband buffers large messages and into smaller blocks. This reduces the performance benefit of using non-blocking for larger message sizes.

Table 3 shows the timing (seconds) when the program is executed using a fixed problem size of 16GB. This is used to measure the speed-up. Four processes are used for the first run and increased to 128 processes. The buffer size is not uniformly distributed among all processes for some cases. The computation and communication times are the average of all processes. Figure 19 shows the speed-up when the number of process gets doubled. The speed-up is more than 2 is some cases.

Table 3: Timing (seconds) for domain size 16GB

Buffer	**Num**	**Intel MPI**			**UHCAF**		
Size	**Process**	**Compute**	**Comm**	**Total**	**Compute**	**Comm**	**Total**
32M	4	892.35	21.44	923.47	1356.79	2.12	1358.92
32M	8	410.53	33.64	452.40	492.63	10.41	544.66
20M-28M	16	194.19	11-12	208.05	261.23	5.4-7.3	274.91
12M-20M	32	100.07	6-10	110.66	141.26	3-5.7	148.08
8-16M	64	45.68	6-7	55.72	61.45	3-7.4	70.93
5-10M	128	21-23	3-3.6	27.34	27-30	1.5-3.6	34.79

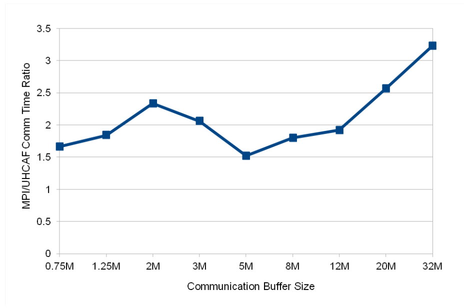


Figure 18: Communication time ratio (MPI/UHCAF) with 8 processes

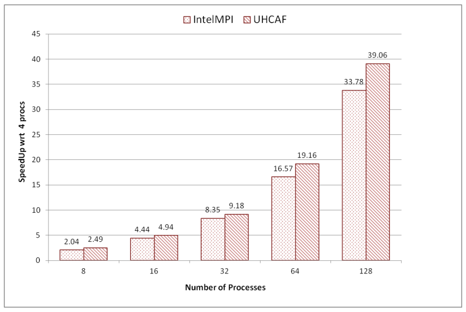


Figure 19: Speedup over 4 processes using fixed domain size of 16GB

Tilted Transverse Isotropic (TTI) Wave Equation This program is much more complex than the Isotropic program as it models an-isotropic media, which has a lot more parameters to consider. This program requires six 3-D matrices to store the timing data, which is subdivided to be processed by each image. After each iteration the ghost cells is exchanged. Due to huge memory requirement, the program cannot be executed with less than 16 images. The OpenMPI version uses traditional assumed shape array declarations instead of dynamic allocation (to prevent performance impact). The program is executed twice with Intel MPI, with and without the xhost flag. The xhost flag tells the compiler to optimize for the specific hardware. Table 4 and Figure 20 compares the communication time between UHCAF and MPI using 16GB domain size. The matrix dimensions are 1024x2048x2048 with 4 ghost points. The buffer size in the table is the sum of all the communication buffer of all processes.

Table 4: Communication time (seconds) for TTI

Buffer (GB)	#Processes	UHCAF	OpenMPI	Intel MPI	Intel MPI xhost
75.54	16	23.17	34.67	34.13	27.28
101.03	32	13.22	17.45	19.48	19.23
152.20	64	17.87	20.60	18.84	14.14
203.96	128	8.84	9.18	9.41	9.12

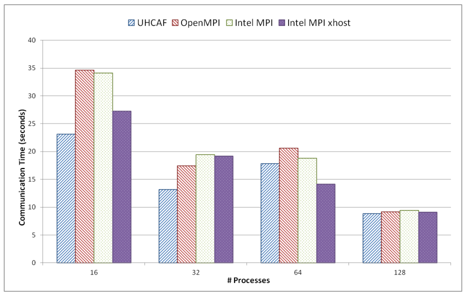


Figure 20: Communication time (seconds) for TTI

Table 5 and Figure 21 compares the total execution time of the TTI program. Note that it does not include file IO. The buffer size in the table is the sum of all the communication buffer of all processes. Figure 22 shows the speedup with respect to 16 processes.

2.7 Runtime Optimization For Memory Access

Multi-threaded applications, including OpenMP, are rather sensitive to the memory accesses of the executing threads, both at the cache level as well as at the page level. At the cache level, a performance problem called false sharing may occur on multi-core platforms because blocks of data are fetched into cache on per-line basis. When

Table 5: Total execution time (seconds) for 16 GB domain size

Buffer (GB)	#Processes	UHCAF	OpenMPI	Intel MPI	Intel MPI xhost
2.08	16	2084.81	3149.93	2248.07	2128.65
1.15	32	1094.02	1559.49	1247.73	1172.55
0.61	64	519.54	866.08	622.67	528.76
0.26	128	276.01	449.17	328.40	271.15

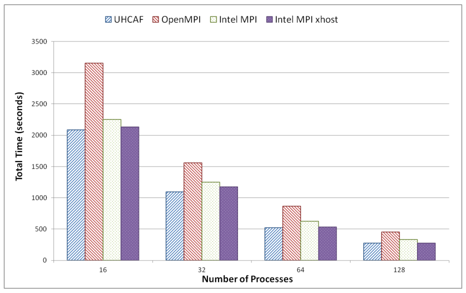


Figure 21: Total time (seconds) for TTI

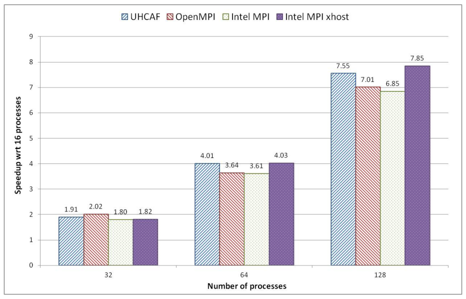


Figure 22: Speedup over 16 processes

one thread accesses data that happens to be on the same line as the data simultaneously accessed by another thread, both need up-to-date copies of the same cache line. In order to maintain the consistency of the shared data, the processor may then generate additional cache misses that degrade performance.

At page level, data locality can be a significant performance factor since data is allocated to physical memory bank on per-page basis. This problem can occur particularly on a cache coherent Non-Uniform Memory Access (ccNUMA) system, where different memory banks are connected to different multi-core processors. In such system, a processor has to use the system interconnect to access the memory banks connected to the other processors. This remote memory access has a longer latency and can be a major bottleneck if it happens frequently.

Many novice OpenMP developers are not aware of memory bottlenecks caused by the false sharing effect and the ccNUMA behavior. Even with this awareness, It can be very hard for the developers to correctly identify the source of such performance problems, as it requires some amount of understanding of the underlying system. Furthermore, they may not know how to adapt an application in order to fix the problem.

To address these issues, we have developed a dynamic optimization framework called DARWIN, that is based on the open-source OpenUH [20] compiler. The core feature of this framework is its usage of the OpenMP collector API [9] to interact with a running OpenMP program. The collector API can track various OpenMP states on a per-thread basis, such as whether a thread is in a parallel region. DARWIN also utilizes hardware counters to obtain detailed information about the programs execution. When combined with its other capabilities, such as relating the performance data to the data structures in the source code, DARWIN is able to help the application developer to gain insights about dynamic code behavior, particularly in the hot-spots of a parallel program.

2.7.1 Components of DARWIN

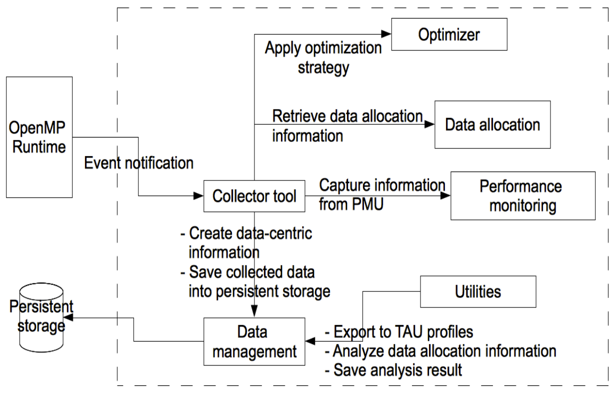

OpenMP Runtime
Event notification
Optimizer
Apply optimization strategy
Retrieve data allocation information
Data allocation
Collector tool
Capture information from PMU
Performance monitoring
- Create data-centric information
- Save collected data into persistent storage
Utilities
Persistent storage
Data management
- Export to TAU profiles
- Analyze data allocation information
- Save analysis result

Figure 23: The DARWIN Framework

Figure 23 illustrates the main modules of the DARWIN framework. These modules are responsible for catching OpenMP collector event notifications, performance monitoring, capturing data allocations, data management, optimization, and utilities for supporting performance data analysis.

The collector tool is the central part of the DARWIN framework since it coordinates the profiling and optimization activity. This component utilizes the OpenMP Collector API to communicate with the OpenMP runtime and thus gain insight about a program's execution. In a common scenario, we configure the collector tool to catch the events associated with the beginning and the end of an OpenMP parallel region. The parallel region, most of the time, is the basis for finding the hot spots in an OpenMP program.

The performance monitoring module abstracts access to the processor-specific hardware counter(s). In the work we have performed here, an appropriate counter is used to pinpoint the data structure that is causing the performance problem under investigation. The DARWIN framework has been implemented on the Itanium 2 platform that provides the Data Event Address Register (DEAR) suitable for this purpose. The DEAR can track load instructions and capture instruction and data addresses, as well as the latency of data cache misses. We can use this information to help programmers optimize the memory behavior of the program, as described further in Section 2.7.2 and Section 2.7.3. We use the *libpfm* library to implement this module.

The data allocation module is used to capture information on the placement of global, static, and dynamically allocated data. The information recorded includes the starting virtual memory address, allocation size, and an identifier. For global and static data, it uses the variable name as the identifier. For dynamic data, it uses the function name that calls *malloc* and the line number in the source code as the identifier. We use *libelf*, *libpsx*, and intercept the calls to memory allocation routines (malloc, calloc, etc) with an interposition library to implement its functionality. Information about the allocated data is required for relating the captured performance data with the corresponding data structure in the source code.

The data management module is responsible for relating the captured performance data to the appropriate data structure in the source code and thus for producing data-centric information that is comprehensible by the application developer. It is also used to store all of the captured information and analysis results into persistent storage. Currently the *SQLite* portable database is used to implement this persistent data storage. It has an in-memory database feature to reduce disk access for improved performance. Its support for the SQL language offers a convenient way to access the data.

The optimizer module provides an implementation of several optimization strategies that use input from the analysis result. Two kinds of optimization strategies, high level and low level, are distinguished. The high level optimizations essentially utilize OS routines, such as those for setting thread affinity, memory page migration, or making calls to other library routines; e.g. for modifying the number of threads, adjusting core frequency, or accessing a specialized malloc library. The low level optimizations are applied by transforming the source code or modifying instructions in the binary. We have implemented a high level optimization strategy for distributing data on a cc-NUMA platform.

The **utilities module** provides support for offline data analysis. One tool is used to read the collected performance data from the persistent data storage, aggregate them, and write the result into text files that follow the Tuning and Analysis Utilities (TAU) profile format. A second tool can be used to insert analysis results into the framework's persistent data storage.

The DARWIN framework is used as a feedback-based dynamic optimization system that has two execution phases, the monitoring phase and the subsequent optimization phase. The monitoring phase collects the performance data required for analysis, the results of which will be used during the optimization phase in a subsequent run.

2.7.2 Optimizing data distribution on ccNUMA platforms

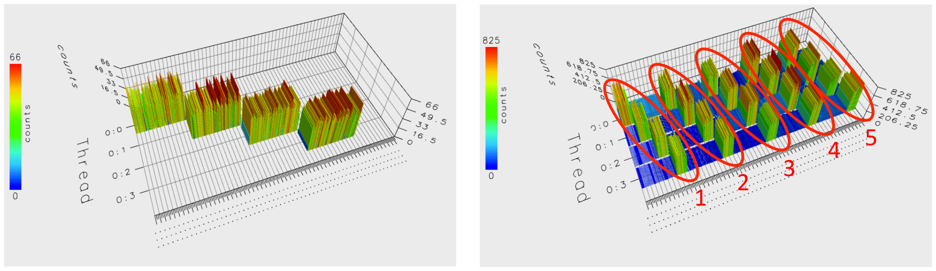
The placement in memory of the pages holding a program data can have a major impact on the performance of an application program. The effects of data placement are more evident on ccNUMA systems than those with symmetric memory access. On such platforms, each processor can directly access the local main memory, but has to use the system interconnect to access the memory banks of the other processors (remote memory). Remote memory accesses become a major bottleneck if the data is not carefully placed. The traditional "first-touch" policy implemented within the operating system is sometimes very effective, but can also lead to an inefficient page placement, especially if the programmer is not aware of this policy. With knowledge of the memory access patterns, one can devise an efficient memory placement strategy and reduce the number of remote memory accesses.

We have used the DARWIN framework to help the application developer find variables that have a data distribution problem, i.e. whose placement leads to many remote memory accesses, and also to perform the necessary optimization to reduce the number of remote memory accesses. In the monitoring phase of DARWIN, we collect the memory reference information in every parallel region. The collector tool module is responsible for starting and stopping the performance monitoring module when the thread reaches the beginning and the end of a parallel region respectively. By using the data allocation information, we create data-centric information by associating the memory address on each of the references with its corresponding variable name. The generated data-centric information is delivered to *TAU ParaProf* for offline analysis. *TAU ParaProf* can provide a visualization that helps the application developer to classify the data access pattern type of each variable and to identify the variables that have unoptimized data placement.

Figure 24(a) and 24(b) show the memory reference visualization supplied by *TAU ParaProf*. The vertical axis gives the total amount of references on each page. The horizontal axis contains the page numbers of the variable. The depth axis provides the thread ID that accesses the variable. The application developer can identify the access pattern type of a variable by inspecting this visualization.

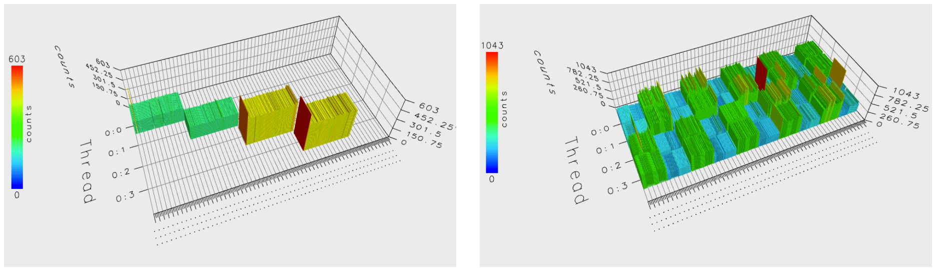
Figure 25(a) and 25(b) present examples of the latency visualization. The vertical axis provides the average latency of the accesses to a page. The horizontal axis contains the page number starting from the left. The depth axis shows the thread id. With this visualization, the application developer can determine whether a variable has a latency imbalance and if it is worthwhile optimizing the accesses.

During the optimization phase, the optimizer module calculates the page numbers of a variable that need to be redistributed and maps them to the destination memory



(a) (b)

Figure 24: Memory reference visualization: (a) block access pattern for *colidx* on CG. (b) cyclic access pattern for *rhs* on SP.



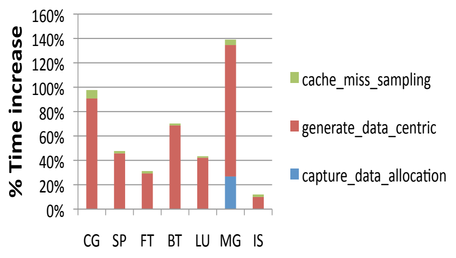
(a) (b)

Figure 25: Memory access latency visualization: (a) *colidx* in CG has a latency imbalance. (b) *rhs* in SP is already optimized for ccNUMA platform.

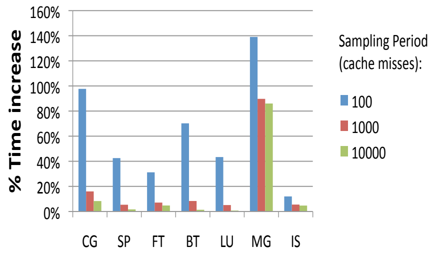
node based on the variable's access pattern type. After the mapping is obtained, this module performs the optimization by using the first-touch method for initial placement and next-touch method on dynamic data or data that has multiple access pattern types across different parallel regions. Optimization based on the data access pattern is flexible in the sense that it is not sensitive to the runtime configuration, such as the data size and the number of threads. We do not need to repeat the monitoring phase even if the runtime configuration is changed. This is very useful for monitoring the applicaiton on a reduced data set and subsequently optimizing the application in the production environment, which typically will require a larger data size and more threads.

We tested our method on seven programs from the OpenMP C version of the NPB-2.3 benchmark: CG, BT, MT, FT, IS, LU, and MG. All programs were compiled with the OpenUH compiler with optimization level O2 and class A data size. The experiments were performed on an SGI Altix 3700 consisting of 32 nodes with dual 1.3 GHz Intel Itanium2 processors per node running the SUSE 10 operating system.

Figure 26(a) gives the increase in time of the monitoring phase with a sampling period of 100 cache misses. The sampling period determines how many cache misses will be skipped by the hardware counter before it captures a cache miss. The monitoring overhead consists of the time taken to capture the data allocation, create data-centric information, and collect performance data. Based on the experimental results, most of the overhead came from the time required to create the data-centric information. This overhead is related to the amount of captured information. Figure 26(b) shows that a higher sampling period, which produces less performance data, can reduce the time required to generate data-centric information.



(a)

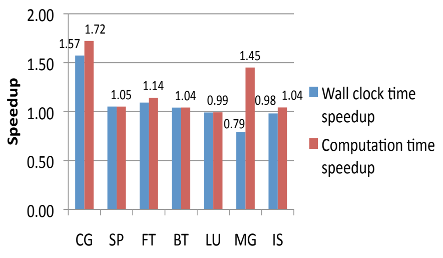


(b)

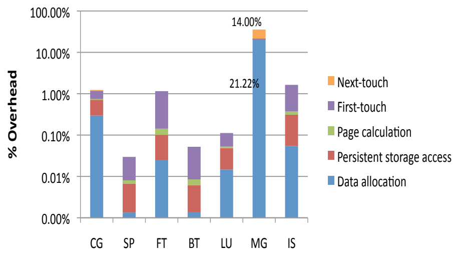
Figure 26: Monitoring phase overhead: (a) overhead breakdown with 100 cache misses for the sampling period. (b) overhead with higher sampling period

Figure 27(a) shows the speedup on the overall wall clock execution time and the exclusive computation time of the optimization phase in the subsequent run. The wall clock execution time includes the overhead, initialization, and the computation time. The optimization attempt had a positive impact on the wall clock and computation time of 4 of the programs, with up to 1.72x speedup. However, with this small problem size, that was not the case for LU, MG, and IS. The result for LU was expected because we did not find any variables that need to be optimized. MG did not experience an improvement of its wall clock time because the overhead of performing the optimization with the next-touch method and capturing the dynamic data allocation exceeded the optimization benefit, as shown by the overhead breakdown in Figure 27(b). We can reduce the overhead significantly by removing the requirement of capturing the data allocation information if the data size is consistent, e.g. in a production environment. IS did not experience much improvement of its wall clock time because 80% of its execution time is spent in the initialization stage, and that is carried out in serial mode. Adjusting the data distribution actually increased the number of remote memory accesses during the initialization of IS, which also reduced the optimization benefit.

To show the flexibility of the optimization method, i.e. that the monitoring results can be applied to different runtime configurations, we performed another optimization attempt on these benchmark programs with a larger data size and with more threads. In this case, MG and IS also benefit from our optimization. The optimization experiment was performed on class B data size with four compute nodes, eight physical CPUs, and eight OpenMP threads. We reused the same optimization strategy as on the class A programs. Figure 28 shows the speedup of this run. It clearly shows that all programs, except for LU, gained a higher speedup if compared to the experiment on the class A data size.

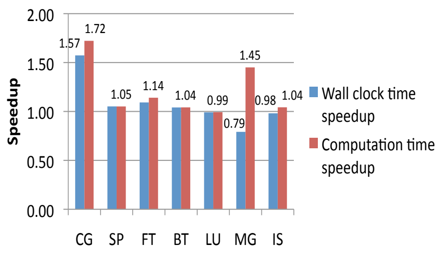


(a)

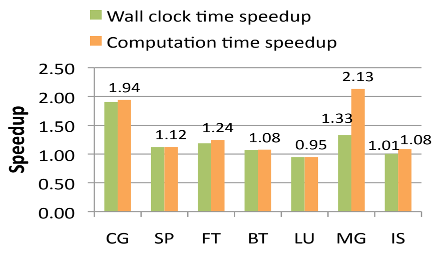


(b)

Figure 27: Optimization result on class A with 4 threads : (a) performance speedup. (b) overhead.



(a)



(b)

Figure 28: Performance speedup : (a) class A with 4 threads. (b) class B with 8 threads.

2.7.3 Detecting false sharing

False sharing is an unnecessary condition that may arise as a consequence of the cache coherence mechanism working at cache line granularity. It does not imply that there is any error in the code. This condition may occur when multiple processor cores access different data elements that reside in the same cache line. A write operation to a data element in the cache line will invalidate all the data in all copies of the cache line stored in other cores. A successive read by another core will incur a cache miss, and it will need to fetch the entire cache line from either the main memory or the updating core's private cache to make sure that it has the up-to-date version of the cache line. Poor scalability of multi-threaded programs can occur if the invalidation and subsequent read to the same cache line happen very frequently.

Our approach for determining the data that exhibits false sharing consists of two stages. The first stage checks whether cache coherence misses contribute to a major bottleneck in the program. If they do, then the second stage isolates the data structures that cause the false sharing.

In the first stage, we use DARWIN to count the number of cache line invalidations. Modern processors provide Performance Monitoring Unit (PMU) support for this purpose. For example, the Intel Core I7 family supports an event called *MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM* that indicates the number of retired memory load instructions that hit dirty data in sibling cores. Intel's Itanium 2 processor, the platform where DARWIN is currently implemented, can count the number of cache line invalidation transactions represented by the *BUS_MEM_READ_BRIL_SELF* event. If a large number of these events is detected during a program's execution, it indicates that a serious cache coherence problem can occur when the program is executed with multiple threads.

We move on to the second stage after we determine that an application exhibits a significant cache coherence problem. This part is traditionally done by tracing the memory operations, and analyzing the sequence of the read and write operation. But this method can produce very high overheads, significantly reducing its applicability. To alleviate the overhead, DARWIN captures the memory reference information that is used to identify those data structures that have a significant false sharing problem. Only the high level symptoms are observed, namely the high memory access latency and the large number of references. In contrast to the method discussed in Section 2.7.2 that works at page level, here we analyze the memory reference information at cache line granularity.

We examine the data allocation information to determine if the identified data structures suffer from false sharing. If some or all parts of a data structure share the same cache line with another data structure, we conclude that the problem is due to false sharing. We also look at the data access pattern type when the data structures are shared among threads. False sharing can exist on an array that is accessed by multiple threads, where each thread only accesses a portion of the array. It is possible that some elements near the boundary of two disjoint data portions are in the same cache line. If the array size is small, e.g. it takes up about as many bytes as there are in a cache line, most of the array might be contained in the same cache line, causing an increased risk of false sharing.

The results of our false sharing detection are validated by performing manual optimization to the source code that adjust this, without making other changes. We use the *aligned* variable attribute and *posix_memalign* function to allocate data on different cache lines. The result of the detection is considered to be valid when the performance of the optimized code is substantially better.

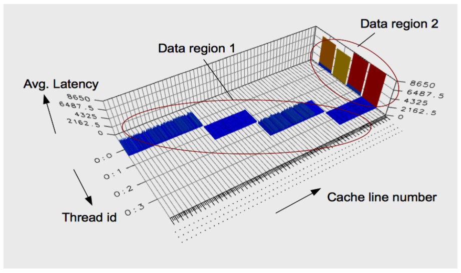
We tested our method on the Phoenix suite that implements MapReduce for shared memory systems. All programs were compiled with the OpenUH compiler with optimization level O2. The experiments were performed on an SGI Altix 3700 consisting of 32 nodes with dual 1.3 GHz Intel Itanium2 processors per node running the SUSE 10 operating system.

Table 6 depicts the results of the cache invalidation event measurement. It shows that *histogram*, *linear_regression*, *reverse_index*, *string_match*, and *word_count* all had a very large number of cache invalidation events when using higher numbers of threads. This is an indication that these programs suffer from a cache coherence problem. We then focused on these programs in order to identify the variables that give rise to false sharing.

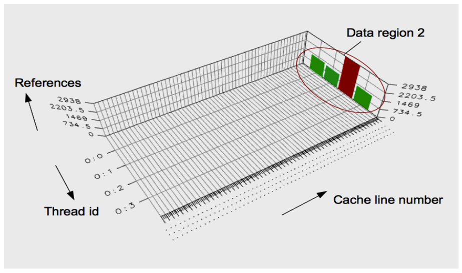
Figures 29(a) and 29(b) show the *TAU Paraprof* visualization of the average latency and the number of references to each cache line for the *string_match* program. We identified two distinct data regions with different access patterns. Both figures clearly show that the accesses to data structures in data region 2 were causing the major bottleneck

Program Name	Total Cache Invalidation Count			
	1-thread	2-threads	4-threads	8-threads
histogram	13	**7,820,000**	**16,532,800**	**5,959,190**
kmeans	383	28,590	47,541	54,345
linear_regression	9	**417,225,000**	**254,442,000**	**154,970,000**
matrix_multiply	31,139	31,152	84,227	101,094
pca	44,517	46,757	80,373	122,288
reverse_index	4,284	89,466	217,884	**590,013**
string_match	82	**82,503,000**	**73,178,800**	**221,882,000**
word_count	4,877	**6,531,793**	**18,071,086**	**68,801,742**

Table 6: Cache invalidation event measurement result.



(a) Average memory latency



(b) Memory reference count

Figure 29: *String_match* memory access visualization

of this program. Data region 2 contained the memory accesses to variable *key1_final*, *key2_final*, *key3_final*, *key4_final*, and *string_match_map_253_3*. The first four variables were global variables allocated in the same cache line. *string_match_map_253_3* is a dynamic data structure that has one of its cache lines shared with another dynamic data object allocated by other threads. Table 7 presents information on several data structures captured by DARWIN.

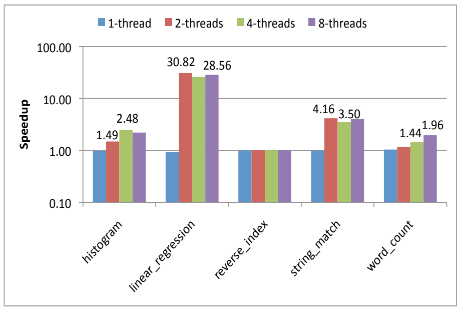
Parallel region id	Thread id	Variable name	Starting cache line	Last cache line	Size (bytes)
0	0	key_1_final	**0x4c00**	**0x4c00**	8
0	0	key_2_final	**0x4c00**	**0x4c00**	8
0	0	key_3_final	**0x4c00**	**0x4c00**	8
0	0	key_4_final	**0x4c00**	**0x4c00**	8
1	0	string_match_map_253_0	0x14049d80	0x1404a180	1024
1	1	string_match_map_253_1	0x45400	0x45800	1024
1	3	string_match_map_253_2	0x18000880	0x18000c80	1024
1	2	string_match_map_253_3	**0x1404a580**	0x1404a980	1024
1	0	string_match_map_254_0	0x1404a180	**0x1404a580**	1024

Table 7: Data allocation information for several variables in *string_match*.

Figure 30(a) presents our attempt to adjust the alignment of the variables that were suspected to be the major cause of false sharing. Figure 30 (b) shows the speedup after we performed adjustments to the source code. The speedup is defined as the execution time of the original program divided by the execution time of the modified one. The number of cache invalidation events in *reverse_index* was successfully reduced by 50%, even though it did not experience a significant improvement.

```
...
char *key1_final;
__attribute__((aligned (128)))
char *key2_final;
__attribute__((aligned (128)))
char *key3_final;
__attribute__((aligned (128)))
char *key4_final;
__attribute__((aligned (128)))
...
posix_memalign(&cur_word,256,MAX_REC_LEN);
...
```

(a)



(b)

Figure 30: (a) Adjusting data alignment in *figures/string_match*. (b) The speedup after adjusting the memory alignment.

Figure 31 shows the slowdown of the monitoring phase, defined as the monitoring execution time divided by the original program execution time. The monitoring phase generated a slowdown ranging from 1.15x to 1.70x, with the average around 1.36x. This overhead is actually pretty low compared to the traditional methods, which can produce a slowdown from 5x to 10x or more.

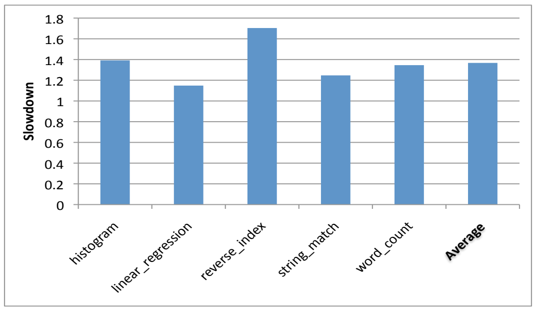


Figure 31: The slowdown of the monitoring phase relative to the original program.

2.8 Cost Model and Compile-time Optimization

Existing Open64's cache model analyzes the spatial and temporal locality of single thread execution, while Open64's parallel model focuses on the worksharing benefits from concurrent execution of threads. However, neither of them, nor their combination takes into account the performance impact from the interference or contention for resources between parallel threads such as the false sharing effects, the competition to use shared cache or memory bus. With increasing number of cores and the decrease of average memory and bus bandwidth per cores, such interference and contention will have significant performance impacts for applications.

We have studied one of these interferences, false sharing, and enhanced the Open64's cost model to include false sharing effects for it to output more accurate performance estimations. Our false sharing cost model estimates the number of false sharing cases in a parallel loop at compile-time, and computes the overhead cost incurred by the problem to the whole execution of the loop. For wide applicability, we use OpenMP parallel loops in this model. Given an OpenMP parallel loop, there are four steps to analyze the cost incurred by false sharing:

1. Obtain array references made in the innermost loop of a loop nest
2. Generate a cache line ownership list for each thread
3. Apply a stack distance analysis to each cache line ownership list
4. Detect false sharing

The false sharing is only revealed at runtime and is sensitive to lots of details about how the program is being executed, e.g. the alignment of allocated memory, the number of threads working on the victim data, and other background applications that may compete for the cache resources. It is necessary to supply enough runtime information to the compiler when estimating the false sharing effects. In this model, the compiler needs information about the number of threads executing the loop, loop boundaries, step sizes, index variables and the *chunk size* if specified for the OpenMP parallel loop. The chunk size is the number of iterations of a loop that are distributed to each thread. In this work, we assume that chunks of a loop are distributed to threads in a round-robin fashion. If the loop boundaries are not known at compile-time, the model only outputs the false sharing rate estimated per full cycle of iterations executed by all of the threads. One full cycle of iterations executed by the thread team is the sum of iterations executed by each thread in one chunk size.

We have implemented our false sharing cost model within the Open64 compilers LNO phase. We have used OpenMP versions of loop kernels in *heat diffusion* [2] and *discrete fourier transform (DFT)* [1] programs for our experiments. To evaluate the accuracy of our false sharing cost model, we compare the percentages of measured and computed false sharing overhead costs. We expect that the measured percentage of false sharing overhead should be close to the percentage of false sharing overhead computed by our false sharing cost model as follows:

$$\frac{T_{fs\_measured} - T_{nfs\_measured}}{T_{fs\_measured}} \approx \frac{N_{fs\_pred} - N_{nfs\_pred}}{N^{*}_{fs\_pred}} \tag{3}$$

where $T_{fs\_measured}$ is the measured time needed to execute a loop incurring false sharing; $T_{nfs\_measured}$ is the measured time needed to execute the same, but optimized, loop that does not incur any false sharing; N_{fs_pred} is the number of false sharing cases estimated by our model on a loop incurring false sharing; N_{nfs_pred} is the number of false sharing cases estimated by our model on an optimized loop; $N^{*}_{fs\_pred}$ is a normalized value for false sharing cases estimated by our model on a loop incurring false sharing.

Results for our experiments given in Tables 8 and 9 show that the computed FS overhead percentages estimated by our cost model are close to the measured FS overheads, indicating that the results of our cost model are promising, and that by modeling false sharing we can accurately estimate the false sharing overhead cost at compile-time.

Table 8: Comparison of % of false sharing overheads incurred in Heat Diffusion kernel

# of threads	Measured Time with chunk size=1 FS case (sec)	Measured Time with chunk size=64 non-FS case (sec)	FS effect on execution time (%)	Computed FS cases effect (%)
2	0.3593	0.2901	19.2%	6.9%
4	0.2263	0.1646	27.2%	6.9%
8	0.1639	0.156	4.8%	6.9%
16	0.6586	0.6205	5.7%	7.0%
24	1.0049	0.9564	4.8%	7.1%
32	1.4671	1.3608	7.2%	7.2%
40	1.8455	1.6130	12.5%	7.2%
48	2.247	2.1501	4.3%	7.2%

Table 9: Comparison of % of false sharing overheads incurred in DFT kernel

# of threads	Measured Time with chunk size=1 FS case (sec)	Measured Time with chunk size=16 non-FS case (sec)	FS effect on execution time (%)	Computed FS cases effect (%)
2	2.0978	1.7624	15.9%	32.0%
4	1.762	0.9618	45.4%	31.6%
8	0.8976	0.6033	32.7%	31.5%
16	0.599	0.3688	38.4%	33.2%
24	0.5041	0.3163	37.2%	32.8%
32	0.4727	0.2827	40.1%	35.6%
40	0.4792	0.2669	44.3%	36.7%
48	0.4664	0.279	40.1%	35.8%

Possible optimizations that can be applied by the compiler based on our false sharing cost model results are as follows:

- Determine the optimal chunk size value for OpenMP loops and the optimal number of threads to execute the loop.

- Guide traditional loop transformations to decide parameters suitable for executing parallel loops on multicore architecture.
- Decide whether software prefetching is profitable, and determine the best prefetching instruction.

2.9 Evaluations of HMPP for Programming GPGPUs

Directive-based interfaces already exist that can be used to program GPUs, although they are proprietary. In order to understand the potential of this approach, to learn what information must be supplied by such directives, and to evaluate their potential to provide both performance and productivity, we have used them to create versions of two application codes that exploit both CPUs and GPUs [21].

Last year we studied the DoE High-Order Multi-scale Modeling Environment application (HOMME) that is one of the highly promising frameworks for integrating the atmospheric primitive equations in spherical geometry. In the study phase, we analyzed the code in terms of its suitability for GPUs. We also studied the various directive-based approaches and their execution and memory models, as well as compiler-based optimization techniques to port the application onto GPU and CPU cores. HOMME applies a spectral element method to conserve both mass and energy using an isotropic hyper-viscosity term. To discretize the horizontal dimension, it uses a cubed-sphere grid and in the radial direction a vertical dimension. The HOMME application consists of several hundred Fortran 90 subroutines with the computations spread evenly across them and whose relevance depends on the input problem. For each of the spherical elements in the grid, HOMME maintains a global data structure that stores the state of the element, including velocity, temperature, pressure, divergence and geo-potential.

This year we have performed experiments on the HOMME kernel using the directive-based approaches from PGI and CAPS (HMPP). We have also used OpenMP and CUDA to provide additional experimental data. We considered several optimization strategies in order to best port this application on the GPU. While using the PGI-based approach, we did a careful analysis and implemented a suitable data initialization i.e. we initialized the computational arrays on the GPUs directly as far as possible to avoid the expensive operation of transferring data from CPUs. For the HMPP-based approach, we used a loop collapsing technique i.e. fusing multiple loops into one loop. Another technique we employed for the HMPP code version was to use their advanced-load clause to reduce unnecessary transfer of data between the GPU and the CPU.

In 32, we list the HMPP and PGI implementation code snippets discussed below. The ie loop iterates over the spherical elements, the q loop over the advected physics, the k loop iterates over the vertical radial grid points and the j and i loops iterate over the horizontal plane grid points.

With the PGI accelerator directives, it was necessary to do some code restructuring to achieve good performance. We inlined the procedure divergence sphere and inserted an acc region directive to accelerate the compute intensive ie loop. This was necessary for the PGI directives since their approach cannot handle function calls inside an accelerated region. The next step was to specify how to parallelize the loop nest iterations

```
!$acc region
!$acc do parallel(nete)
  do ie=nets, nete
!$acc do parallel(qsize)
   do q=1,qsize
!$acc do vector(32)
   do k=1,nlev
!$acc do vector(nv)
    do j=1,nv
!$acc do vector(nv) private(dudx00,dvdy00i)
     do l=1,nv
      dudx00=0.0d0
      dvdy00i=0.0d0
      do i=1,nv
       dudx00 = dudx00 + Dvv(i,l  ) * &
       (metdet(i,j,ie)*(Dinv(1,1,i,j,ie)* &
        gradQ5da(i,j,k,q,1,ie) + &
        Dinv(1,2,i,j,ie)*gradQ5da(i,j,k,q,2,ie)))

        dvdy00i = dvdy00i + Dvv(i,j  ) * &
        (metdet(l,i,ie)*(Dinv(2,1,l,i,ie)* &
        gradQ5da(l,i,k,q,1,ie) + &
        Dinv(2,2,l,i,ie)*gradQ5da(l,i,k,q,2,ie)))
      end do
      divdp4da(l,j,k,q,ie)= &
               rmetdetp(l,j,ie) * &
              (rdx(ie))*dudx00+(rdy(ie))*dvdy00i
       end do
      end do
     end do
    end do
   enddo
!$acc end region
```

(a) PGI Accelerator Directives

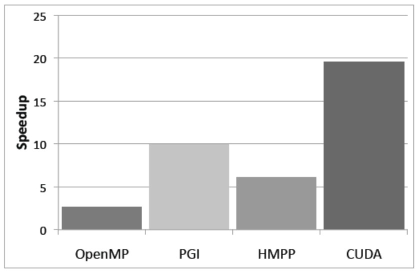
```
!$hmppcg grid blocksize 4x4
!$hmppcg parallel
  do i2=nets, nete*nlev ! ie, q
!$hmppcg parallel
   do i1=1, qsize*nv  ! q, j
!$hmppcg set b2 = BlockId(i2)
!$hmppcg set t2 = RankInBlock(i2)
!$hmppcg set b1 = BlockId(i1)
!$hmppcg set t1 = RankInBlock(i1)
   ie=b2+1
   q = b1+1
   k =t2 +1
   j =t1+1
     do l=1, nv
       dudx00=0.0d0
       dvdy00i=0.0d0
       do i=1,nv
          dudx00 = dudx00 + Dvv(i,l  ) * &
          (metdet(i,j,ie)*(Dinv(1,1,i,j,ie)* &
          gradQ5da(i,j,k,q,1,ie) + &
          Dinv(1,2,i,j,ie)*gradQ5da(i,j,k,q,2,ie)))

          dvdy00i = dvdy00i + Dvv(i,j  ) * &
          (metdet(l,i,ie)*(Dinv(2,1,l,i,ie)* &
          gradQ5da(l,i,k,q,1,ie) + &
          Dinv(2,2,l,i,ie)*gradQ5da(l,i,k,q,2,ie)))
        end do
      divdp4da(l,j,k,q,ie)= rmetdetp(l,j,ie)
             * (rdx(ie)*dudx00+rdy(ie)*dvdy00i)
     enddo
   enddo
  enddo
end subroutine
```

(b) HMPP Implementation

Figure 32: The inlined and accelerated divergence sphere code snippet.

HOMME/SE Timings (Miliseconds)	
SERIAL	1366.37
HMPP Kernel	224.73
PGI Kernel	137.43
CUDA	70.00
OpenMP 4 Threads (best)	510.62

(a)HOMME/SE Timing Table



(b)HOMME/SE Divergence Sphere Speedup

Figure 33: HOMME/SE Kernel Experiments.

across the GPU Symmetric Multi-processors (SM) and within the SMs efficiently. We used the parallel and vector clauses to specify the grid size and vector size: in this case we specified a block size of nv*nv*nv. To avoid non-coalesced memory accesses we eliminated a temporary array gv and fused the inner loops. We also allocated and initialized the data inside the GPU by using the PGI data region directive and copyout directive. To obtain best results for the kernel, we allocated and initialized the twelve spectral elements state on the GPU.

We used a similar code transformation to implement the kernel with HMPP directives. We used HMPP to outline the ie loop to a separate procedure creating a codelet. In contrast to the strategy for the PGI directives, here we had to transform the loops by collapsing the ie and the q loop with the i and the e loop, respectively, to provide enough work for a two-dimensional thread block and get the desired performance. Our experiments utilized HMPP 2.5.0 that only supported two dimensional thread blocks. 33(a) shows the execution time for HOMME using the directives and OpenMP; 33(b) shows the speedup of the kernel. The OpenMP version (on 4 threads) achieves a speedup of 2.67 over the serial version. Without counting the data transfer time, the GPU implementations achieve a speedup of 9.9x for the (ACC) PGI directives, 6.08 for HMPP and 19.51 for the CUDA implementation.

2.1.2 S3D Parallel Combustion Application on Heterogeneous Platform

We also performed experiments on the S3D parallel combustion application using the GPU-based directives. The S3D parallel combustion application is a flow solver for the direct numerical simulation of turbulent combustion. S3D solves fully compressible Navier-Stokes, total energy, species and mass conservation equations coupled with detailed chemistry. The governing equations are supplemented with additional constitutive relations, such as the ideal gas equation of state, models for chemical reaction rates, molecular transport and thermodynamic properties. These relations and detailed chemical properties are implemented as kernels or community-standard libraries that are amenable to acceleration through GPU computing. For this work, we chose the thermodynamics kernel that evaluates the mixture-specific heat, enthalpy and Gibbs functions as a temperature polynomial. The coefficients of the thermodynamic polynomials and their relevant temperature ranges are obtained from thermodynamic databases following the conventions used in the NASA Chemical Equilibrium code.

```
!$acc data region copyin(temp,...),&
   copyout(enth)
do j = 1, MR
!$acc region
!$acc do parallel(np)
  do i = 1, np
    enth(i) = 0.0
    do m = 1, nslvs
      if(temp(i)<midtemp(m)) then
        enth(i)=enth(i)+yspec(i,m)*&
          Rsp(m)*(&
          ...
        coefflow(5, m)*rp05)))))
      else
        enth(i)=enth(i)+yspec(i,m)*&
          Rsp(m)*(&
          ...
        coeffhig(5, m)*rp05))))))
      end if
    end do
  end do
!$acc end region
!$acc region
!$acc do parallel(np)
  do i = 1, np
    cp(i) = 0.0
    do m = 1, nslvs
      ...
    end do
  end do
!$acc end region
end do
!$acc end data region
```

(a) PGI

```
!$hmpp <cudagroup> group, target=CUDA
!$hmpp <cudagroup> resident, args[Rsp].io=in
real,parameter::Rsp(1:nstts)=Ru/molwgt(1:nstts)
!$hmpp <cudagroup> resident, args[midtemp].io=in
real,parameter::midtemp(68)=(/ ... /)
!$hmpp <cudagroup> resident,args[coeffhig].io=in
real,parameter::coeffhig(7,68)=reshape(/.../)

subroutine calc_mixenth(np, ... ,cp)
implicit none
. . .
!$hmpp <cudagroup> allocate
!$hmpp <cudagroup> s3d_mixenth advancedload,&
       args[::Rsp; ...; ::coeffhig]
!$hmpp <cudagroup> s3d_mixenth callsite
call hmpp_kernel1(np, ... , coeffhig)
!$hmpp <cudagroup> s3d_mixcp callsite, &
       arg[::Rsp; ...].advancedload=true
call hmpp_kernel2(np, temp, ... , coeffhig)
!$hmpp <cudagroup> release
end subroutine calc_mixenth

!$hmpp <cudagroup> s3d_mixenth codelet, &
    args[np;...;yspec].io=in,args[enth].io=out
subroutine hmpp_kernel1(np,temp,...,coeffhig)
    ...
end subroutine hmpp_kernel1
!$hmpp <cudagroup> s3d_mixcp codelet, &
      args[np;...;yspec].io=in,args[cp].io=out
subroutine hmpp_kernel2(np,...,coeffhig)
    ...
end subroutine hmpp_kernel2
```

(b) HMPP

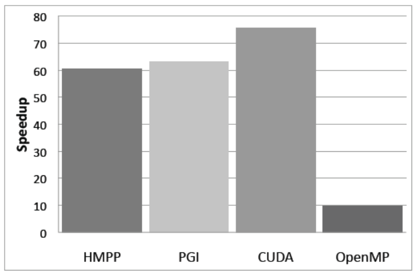
Figure 34: S3D Thermodynamics Kernel Code snippet.

The thermodynamic kernel with small variations is applicable across a wide range of reacting flow applications.

Our first attempt to accelerate the code with PGI and HMPP directives, by inserting an acc region directive and creating a HMPP codelet, respectively, for the main computational loopnest yielded very little performance gain (2x speedup for PGI and 1.2 speedup for HMPP). By using the CUDA profiler from NVIDIA, we observed that most of the time spent in the accelerated kernels was on data transfer between the CPU and GPU. So we had to inline the main computational kernel loop (loop i) to the procedure that was invoking it within its loop j. This transformation is shown in 34(a), which gives the code corresponding to the PGI implementation. It employs a data region directive to define the data that resides in the GPU. For HMPP, we used the group and resident directives to allocate data in the GPU and share data among the codelets of the same group. In 34(b) we show the HMPP implementation, where codelets s3d_mixenth and s3d_mixcp belong the same group named cudagroup. Arrays Rsp, midtemp, coeffhig and coefflow are declared as resident variables, which makes them accessible by the two codelets defined in the HMPP group. In order to optimize the data transfers, we used the advancedload directive to transfer the data required to initialize the read only GPU variables one time before the first codelet calc_mixenth is executed. We also used the advancedload clause of the HMPP callsite directive to notify HMPP that the read only data is available in the GPU for the second codelet.

S3D Thermodynamics Timings (Seconds)	
SERIAL	21.926
HMPP	0.363
HMPP Kernel	0.3192948
HMPP Data Transfer	0.042834
PGI	0.346305
PGI Kernel	0.320225
PGI Data Transfer	0.02608
CUDA	0.29
CUDA Kernel	0.269265
CUDA Data Transfer	0.019952
OpenMP 12 Threads (best)	2.274

(a) S3D Thermodynamics Timing Table



(b) S3D Thermodynamics Speedup

Figure 35: S3D Thermodynamics Kernel Code snippet.

We compared the results of the different parallelization and acceleration methods and we found that the HMPP and PGI implementations produced a 60 and 63 times speedup, respectively. The native CUDA implementation produced a speedup of 76 times, while the OpenMP version using twelve threads produced a speedup of 10, as shown in 35(b). The timings of our experiments are shown in 35(a). Our results show that by managing the data correctly we were able to obtain good performance with the PGI and HMPP accelerator directives, within 80% of the native CUDA performance.

While porting these two applications to GPU as discussed above, we noted that some of data structures used in the CPU had to be manually changed to meet certain specific programming requirements, such as removing pointers that were defined inside a data structure. The HMPP and PGI implementations performed a variety of code transformations during the transformation process: this is necessary for performance; however the user is largely oblivious to what is happening under the hood. Moreover, since even a powerful compiler cannot always perform the necessary restructuring, many other code changes may be needed in order to be able to successfully employ directives. We began to explore how we might be able to support the user by helping them to re-apply successful code modifications, and to decide where to focus their attention. It was challenging to identify a good order of adapting subroutines (or procedures) in a large scientific code and to decide on which of the available subroutines to port first and what has to follow next.

In order to help solve this porting issue, we have proposed a methodology based on compiler technology to address an important aspect of software porting that receives little attention, namely planning support. We also explored how to redefine the notion of similarity of code regions in a manner that helps re-use code modification strategies. Unlike previous similarity-based approaches, our work adopts a bio-inspired view of the program. We participated in the development of Klonos software to implement our ideas (Klonos is under development at Oak Ridge National Laboratory and this work was performed while UH student Wei Ding was a summer intern at this lab) and to conduct experiments on the OpenMP porting of the NASA parallel benchmark suite. We showed that the methodology is effective in providing planning support for the porting of these scientific applications. In fact, we were able to use the methodology to identify a possible optimization that the programmer missed for one of the codes, which is beyond the scope of planning.

3 Future Work

This final project period has seen active discussion and development towards exascale computing. Both hardware and software challenges posed from exascale computer systems requires rethinking of the current evolutionary approach to addressing performance, power and resilience bottleneck. The tasks accomplished in this final project period have equipped us valuable experiences for building the next generation high performance software development toolchains [5]. This is exceptionally valuable as we are entering into the exascale computing era which poses much challenging problems and requires much more innovative solutions. To align the DoE vision for the research and development of the programming model and tool chains to support exascale computing, we intend to continue the following work:

1. We will continue working on the implementation of the PGAS languages UPC and Co-Array Fortran. Our long term goal is to provide an industry-quality optimizing compiler to support CAF, including desired collectives and parallel I/O support. We will also explore CAF implementations on alternative HPC systems, and develop and evaluate hybrid programming models incorporating the use of CAF with other languages/APIs (e.g. OpenMP). We will also work with our partners and the larger community, in particular researchers at national laboratories, to develop standard, benchmarks and validation suites. Another efforts that we would like to involve is to foster community involvement through promoting the use of CAF in the Oil/Gas industry and in the broader HPC arena.

2. We intend to continue our work to define, implement and evaluate novel features to express code with high data and computational locality. Our work will explore how such features may enable the utilization of large numbers of cores, clusters of multicore nodes, and heterogeneous nodes. We will continue working closely with the OpenMP ARB to help define new features for the next release of OpenMP (4.0). We will continue to evaluate OpenMP 3.1 and later compilers whenever it is available. We will collaborate with the community to evaluate OpenMP on heterogeneous embedded systems. We will extend the OpenMP validation suite to validate more programming models and platforms.

3. We will continue to implement compile-time cost models for modeling shared cache contention and memory bus bandwidth, both of which are not considered by existing Open64 cost models. Moreover, we will investigate to integrate use of cost models for efficient task reordering and scheduling purposes. We will further study runtime libraries for support of hybrid programming models, so that we are able to use a single runtime layer to help in the execution of programs that may span multiple nodes and be based on multiple programming interfaces.

4. We will participate in the standards activities of the OpenMP Architecture Review Board and the Multicore Association. We will continue our research in extending OpenMP to heterogeneous systems [10].

4 Additional Information

During this project period, the PI, Professor Barbara Chapman was appointed to the Advanced Scientific Computing Advisory Committee (ASCAC) at the Department of Energy (DOE). This committee advises the DOE on a variety of complex scientific and technical issues related to its Advanced Scientific Computing Research program.

4.1 Education

Three members, Besar Wicaksono, Debjyoti Majumder, and Amrita Banerjee completed their M.S. thesis during this project period. They contributed to the work of OpenMP DARWIN dynamic optimization for memory accesses, and CAF runtime research, and our OpenUH compiler tool support, respectively.

4.2 Employment

Table 10 lists the number of senior researchers and graduate students who have been supported by this grant. The M.Sc. student supported our implementation efforts. In addition to these, two Ph.D. students and one M.Sc. student received funding for two months each to enable them to support the considerable implementation activities that are part of this project.

Research Staff	Number
Senior researcher	1
Ph.D. students	4
M.S. students	1

Table 10: research staff and students

4.3 Publications

During this project period, we have several publications in the proceedings of conference and workshops, and journals, which are included in the reference section.

References

[1] Discrete fourier transform. `http://mathworld.wolfram.com/DiscreteFourierTransform.html`. Last accessed 29-February-2012.

[2] Heat diffusion equation. `http://ccl.northwestern.edu/papers/ABMVisualizationGuidelines/palette/examples/Heat%20Difussion/`. Last accessed 29-February-2012.

[3] Andrew Beddall. The g95 project. url:http://www.g95.org/coarray.shtml.

[4] Dan Bonachea. GASNet specification, v1.1. Technical report, Computer Science Department, University of California, Berkeley, 2002.

[5] Barbara M. Chapman, William D. Gropp, Kalyan Kumaran, and Matthias S. Müller, editors. *OpenMP in the Petascale Era - 7th International Workshop on OpenMP, IWOMP2011*, Lecture Notes in Computer Science. Springer, 2011.

[6] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A multi-platform co-array fortran compiler. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40, Washington, DC, USA, 2004. IEEE Computer Society.

[7] Deepak Eachempati, Hyoung Joon Jun, and Barbara Chapman. An Open-Source Compiler and Runtime Implementation for Coarray Fortran. In *PGAS'10*, New York, NY, USA, Oct 12-15 2010. ACM Press.

[8] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, May 2005.

[9] Oscar Hernandez, Van Bui, Richard Kufrin, Ramachandra C. Nanjegowda, and Barbara Chapman. Open source software support for the openmp runtime api for profiling. In *The Second International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, pages 130–137, 2009.

[10] Lei Huang, Oscar Hernandez, Wei Ding, Barbara Chapman, and Richard Graham. Towards a High-Level GPU Programming Model (Submitted). *Parallel Computing Journal*, 2010.

[11] Lei Huang, Haoqiang Jin, Liqi Yi, and Barbara M. Chapman. Enabling locality-aware computations in openmp. *Scientific Programming*, 18(3-4):169–181, 2010.

[12] Haoqiang Jin, Rupak Biswas, Dennis Jespersen, Piyush Mehrotra, Lei Huang, and Barbara Chapman. High Performance Computing Using MPI and OpenMP on Multi-core Parallel Systems (Submitted). *Parallel Computing Journal*, 2010.

[13] James LaGrone, Ayodunni Aribuki, Cody Addison, and Barbara M. Chapman. A runtime implementation of openmp tasks. In *7th International Workshop on OpenMP, IWOMP2011*, pages 165–178, 2011.

[14] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. Openuh: An optimizing, portable openmp compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2007.

[15] Pavel Neytchev Matthias S. Mller. An openmp validation suite. In *in Fifth European Workshop on OpenMP, Aachen University, Germany*, 2003.

[16] Pavel Neytchev Matthias S. Mller. An openmp validation suite. In *in Fifth European Workshop on OpenMP, Aachen University, Germany*, 2003.

[17] Toone Moene. Towards an implementation of Coarrays in GNU Fortran. http://ols.fedoraproject.org/GCC/Reprints-2008/moene.reprint.pdf.

[18] Matthias Mller, Christoph Niethammer, Barbara Chapman, Yi Wen, and Zhenying Liu. Validating openmp 2.5 for fortran and c/c. In *in Sixth European Workshop on OpenMP, KTH Royal Institute of Technology*, 2004.

[19] Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 533–546. Springer-Verlag, 1999.

[20] The OpenUH compiler project. `http://www.cs.uh.edu/~openuh`, 2005.

[21] Barbara M. Chapman Ramanan Sankaran Richard Graham Oscar Hernandez, Wei Ding and Christos Kartsaklis. Experiences with high-level programming directives for porting applications to gpus. In *In Proceedings of Facing the Multicore-Challenge II, Karlsruhe Institute of Technology, Germany*, 2011.

[22] Munara Tolubaeva, Yonghong Yan, and Barbara M. Chapman. Compile-time detection of false sharing via loop cost modeling. In *Proceedings of 17th International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2011*, 2012.

[23] Besar Wicaksono, Munara Tolubaeva, and Barbara M. Chapman. Detecting false sharing in openmp applications using the darwin framework. In *Proceedings of 24th International Workshop on Languages and Compilers for Parallel Computing, 2011*, 2011.