Int J Parallel Prog (2013) 41:825–854
DOI 10.1007/s10766-012-0230-9

Experiences Developing the OpenUH Compiler and Runtime Infrastructure

Barbara Chapman · Deepak Eachempati · Oscar Hernandez

Received: 24 January 2012 / Accepted: 25 October 2012 / Published online: 21 November 2012

Abstract The OpenUH compiler is a branch of the open source Open64 compiler suite for C, C++, and Fortran 95/2003, with support for a variety of targets including x86_64, IA-64, and IA-32. For the past several years, we have used OpenUH to conduct research in parallel programming models and their implementation, static and dynamic analysis of parallel applications, and compiler integration with external tools. In this paper, we describe the evolution of the OpenUH infrastructure and how we've used it to carry out our research and teaching efforts.

Keywords Compilers · OpenMP · PGAS · Parallelization

1 Introduction

At the University of Houston, we are pursuing a pragmatic agenda of research into parallel programming models and their implementation. Our research interests span language support for application development on high-end systems through embedded systems. Our practical work considers both the need to implement these languages efficiently on current and emerging platforms as well as support for the application developer during the process of creating or porting a code. These activities are

B. Chapman · D. Eachempati (✉)
Department of Computer Science, University of Houston, Houston, TX, USA
e-mail: dreachem@cs.uh.edu

B. Chapman
e-mail: chapman@cs.uh.edu

O. Hernandez
Oak Ridge National Laboratory, Oak Ridge, TN, USA
e-mail: oscar@ornl.gov

complemented by coursework, primarily at the graduate level, that explores the use of programming languages for parallel computing as well as their design and implementation.

Starting roughly 10 years ago, we began a program of research into language enhancements and novel implementation strategies for OpenMP [58], a set of compiler directives, runtime library routines and environment variables, which is the de facto programming standard for parallel programming in C/C++ and Fortran on shared memory and distributed shared memory systems. We also were interested in learning how to exploit compiler technology to facilitate the process of OpenMP application development, with the goals of reducing the human labor involved and helping avoid the introduction of coding errors. Since that time, our research interests have broadened to encompass a range of parallel programming models and their implementations, as well as strategies for more extensive support for parallel application creation and tuning.

In order to enable experimentation, to ensure that we understand the implementation challenges fully, and to demonstrate success on real-world applications, we strove to implement our ideas in a robust compiler framework. Moreover, we decided to realize a hybrid approach, where portability is achieved via a source-to-source translation, but where we also have a complete compiler that is able to generate object code for the most widely used ABIs. This permits us to evaluate our results in a setting that is typical of industrial compilers. Within the context of OpenMP, for instance, our ability to generate object code helps us experiment to determine the impact of moving the relative position of the OpenMP lowering within the overall translation, and allows us to experiment with a variety of strategies for handling loop nests and dealing with resource contention. It is of great value in our research into feedback optimizations. Given the high cost of designing this kind of compiler from scratch, we searched for an existing open-source compiler framework that met our requirements. We chose to base our efforts on the Open64 [1] compiler suite, which we judged to be more suitable for our purposes than, in particular, the GNU Compiler Collection [25] in their respective states of development.

In this paper, we describe the experiences of our research group in building and using open source compiler based on the Open64 compiler infrastructure. OpenUH has a unique hybrid design that combines a state-of-the-art optimizing infrastructure with the option of a source-to-source approach. OpenUH is open source, supports C, C++, Fortran 95/2003, includes numerous analysis and optimization components, and offers support for OpenMP 3.0 and Coarray Fortran. OpenUH includes a PTX backend from NVIDIA for implementing CUDA, and supports automated instrumentation as well as providing additional features for deriving dynamic performance information and carrying out feedback optimizations. It is also the basis for a tool called Dragon that supplies program information to the application developer and is designed, in particular, to meet the needs of program maintenance and porting. We hope that this compiler (which is available at [59]) will complement other existing compiler frameworks and offer a further attractive choice to parallel application developers, language and compiler researchers and other users.

The remainder of this paper is organized as follows. Section 2 provides background on Open64, the basis of our compiler, and the parallel programming models that we

are working with in our research. In Sect. 3, we describe several of our research projects which entailed the development and use of OpenUH. Section 4 describes how OpenUH has been used to support classroom learning of compiler and parallel language concepts. Section 5 concludes the paper with a discussion of future work in OpenUH.

2 Background

2.1 Overview of Open64

Open64 is a robust and modern compiler infrastructure which supports C, C++, and Fortran 95/2003 and includes state-of-the-art analyses and optimizations. The major modules of Open64 are the multiple language front-ends, the inter-procedural analyzer (IPA) and the middle-end/back-end, which is further subdivided into the loop nest optimizer (LNO), global optimizer (WOPT), and code generator (CG). Five levels of a tree-based intermediate representations (IR) called WHIRL exist to support the implementation of different analysis and optimization phases. They are classified as being Very High, High, Mid, Low, and Very Low levels, respectively. Open64 also includes two IR-to-source translators named *whirl2c* and *whirl2f* which can be useful for debugging and also, potentially, leveraged for source-to-source compiler translation.

Open64 originated from the SGI MIPSPro compiler for the MIPSR10000 processor, and was open-sourced as Pro64 in 2000 under the GNU public license. The University of Delaware became the official host for the compiler, now called Open64, in 2001 and continues to host the project today. Over the past 10 years, Open64 has matured into a robust, optimizing compiler infrastructure with wide contributions from industry and research institutions. Intel and the Chinese Academy of Sciences partnered early on to develop the Open Research Compiler (ORC) which implemented a number of code generator optimizations and improved support for the Itanium target. A number of enhancements and features from the QLogic PathScale compiler were also merged in, including support for an x86 back-end.

Open64 has an active developer community including participants from industry and academic institutions. For example, NVIDIA used Open64 as a code optimizer in their CUDA toolchain. AMD is active in enhancing the loop nest optimizer, global optimizer, and code generator. HP has long been active in maintaining the compiler and supporting related research projects using Open64.

Universities currently working on Open64 projects include, but are not limited to, University of Houston, Tsinghua University, the Chinese Academy of Sciences, National Tsing-Hua University, and University of California, Berkeley. For the past several years, an annual Open64 workshop has been held to provide a forum for developers and users to share their experiences and on-going research efforts and projects. As a member of the Open64 Steering Group (OSG), we engage other lead Open64 developers in the community to help make important decisions for the Open64 project including event organization, source check-in and review policies, and release management.

2.2 Parallel Programming Models

2.2.1 OpenMP

OpenMP is a fork-join parallel programming model with bindings for C/C++ and Fortran to provide additional shared memory parallel semantics. The OpenMP extensions consist primarily of compiler directives (structured comments that are understood by an OpenMP compiler) for the creation of parallel programs; these are augmented by user-level runtime routines and environment variables. The major OpenMP directives enable the program to create a team of threads to execute a specified region of code in parallel (`omp parallel`), the sharing out of work in a loop or in a set of code segments (`omp do` (or `omp for`) and `sections`), data environment management (`private` and `shared`), and thread synchronization (`barrier`, `critical` and `atomic`). An explicit asynchronous tasking model is also available to support unstructured parallelism (`task` and `taskwait`), which allows blocks of work (tasks) to be defined and scheduled for execution on the active thread team. Runtime routines allow users to detect the parallel context (*omp_in_parallel*()), check and adjust the number of executing threads (*omp_get_num_threads*() and *omp_set_num_threads*()) and use locks (*omp_set_lock*()). Environment variables may also be used to adjust runtime behavior of OpenMP applications particularly by setting defaults for the current run. For example, it is possible to set the default thread team size (`OMP_NUM_THREADS`) and the default iteration scheduling policy for parallel loops (`OMP_SCHEDULE`).

Its popularity stems from its ease of use, incremental parallelism, performance portability and wide availability. Recent research at language and compiler levels, including our own, has considered how to expand the set of target architectures to include recent system configurations, such as SMPs based on Chip Multithreading processors [48], as well as clusters of SMPs [32]. However, in order to carry out such work, a suitable compiler infrastructure must be available. In order for application developers to be able to explore OpenMP on the system of their choice, a freely available, portable implementation was considered to be desirable.

Many compilers support OpenMP today, including proprietary products such as the Intel compilers, Sun Studio compilers, and SGI MIPSpro compilers. However, their source code is mostly inaccessible to researchers and they cannot be used to gain an understanding of OpenMP compiler technology or to explore possible improvements to it. Several open source research compilers (including Omni [64], Mercurium [6], Cetus [45], and Rose [49]) have been developed. But none of them translate all of the source languages that OpenMP supports, and most of them are source-to-source translators with reduced scope for analysis and/or optimization.

2.2.2 Coarray Fortran

Compiler technology needs to evolve to support parallel programming models for large-scale distributed systems. Global address space models are attractive because they provide a familiar programming model. However, there are severe challenges in getting them to run well at scale. The Partitioned Global Address Space (PGAS) models enhance GAS by exposing processor-memory affinity, which is critical to efficiently

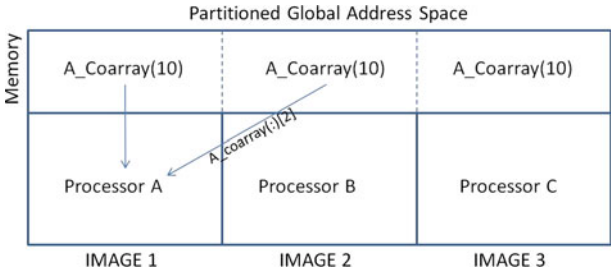

Fig. 1 Logical view of memory in CAF

implement GAS models on distributed architectures. Coarray Fortran (CAF) is a PGAS Fortran extension which has been incorporated into the Fortran 2008 standard. CAF adds new features to the Fortran language to make Fortran programs execute in parallel asynchronously. It follows the SPMD (Single Program Multiple Data) model, where copies of the same program are executed on multiple processing elements with local memories (referred to as *images*), which may or may not reside on the same physical node.

Figure 1 shows the logically shared but partitioned memory view that characterizes PGAS programming models. Specifying *A_coarray(n)* without cosubscripts (square brackets) accesses only the local coarray. This differentiates PGAS from the shared memory model which does not distinguish between local and remote data. Since the remote memory access is explicit, it provides a clearly visible marker for potentially expensive communication operations in the code.

Only objects declared as coarrays can be accessed remotely. Coarrays can be global/ static or dynamically allocated, but in any case they must exist on all images (hence, allocation of an allocatable coarray is a collective operation). Coarrays may be declared with multiple codimensions, in which case the number of images are logically organized into a multi-dimensional grid. For example, the declaration $real::c(2,3)[2,3:4,*]$ logically arranges the images into a $2 \times 2 \times n$ grid for all cosubscripted references to the coarray c. A cosubscripted coarray reference generally indicates a remote memory access. For example, the statement $b(5:6)[2] = a(3:4)$ writes to the 5th and 6th element of coarray b on image 2. Similarly, the statement $a(1:2) = b(5:6)[2]$ reads from the 5th and 6th element of coarray b on image 2.

CAF provides both a global barrier synchronization statement (`sync all`) and a partial barrier synchronization statement (`sync images`) which may be used to to synchronize with a specified list of images. Critical sections, locks, and atomic operations are also part of the language. Additionally, CAF includes several intrinsic functions for image inquiry such as returning the image index of the executing process (*this_image*), the total number of running images (*num_images*), and the image index holding a coarray with specified cosubscripts (*image_index*). Additional features such as *notify*/*wait*, team-based collectives, reductions, and parallel I/O support are being discussed for inclusion into the Fortran standard.

Having an open-source compiler is important for an emerging language as it promotes sharing of ideas and encourages people to freely experiment with it. There have

been few public Coarray Fortran implementations to date. Dotsenko et al. developed CAFC [20], a source-to-source implementation based on Open64 with runtime support based on ARMCI [56] and GASNet [9]. They used Open64 as a front-end and implemented enhancements in the IR-to-source translator to generate Fortran source code to be compiled using GNU compilers. G95 [8] provides a coarray implementation (with closed-source runtime support). G95 allows coarray programs to run on a single machine with multiple cores, or on multiple images across homogeneous networks. In this second mode, images are launched and managed via a *G95 Coarray Console*. There has been a recent effort to implement coarrays in GFortran [53], and an updated design document for this implementation is maintained online. As of this writing, the gfortran implementation does not yet support for multi-image execution, and coarray intrinsics are not supported for coarrays with bounds that are determined at runtime. Rice, more recently, has developed an open source compiler for CAF 2.0 [51] which uses the ROSE compiler infrastructure.

2.2.3 Performance Tools

In our research, we work with a variety of performance tools for parallel applications. By combining static analysis from our compiler with dynamic information reported by these tools, users can more efficiently identify performance bottlenecks in their codes due to suboptimal data layout, poor memory utilization, communication and synchronization overheads, etc. Additionally, feedback from tools can be used by the compiler to direct its optimizations. Many tools are available that can be potentially used for HPC application development, and most, but not all, focus on detecting problems in the structure of MPI programs. Existing tools to support some parts of this process include ParaWise [42], Intel Thread Checker [60], PAPI [10], DynInst [12], SvPablo [63], INTONE [57], Paraver [61], Vampir [11] and TAU [65] all address some aspect of performance analysis and tuning. Modeling and prediction tools include MetaSim [52], PROPHET [23,24], POEMS [5] and DIMEMAS [26]. DIMEMAS provides postmortem performance prediction for MPI applications.

3 OpenUH Development and Results

The OpenUH [47] compiler is a branch of the open source Open64 compiler suite for C, C++, Fortran 95/2003, supporting the IA-64, IA-32, Opteron Linux ABI, and PTX generation for NVIDIA GPUs. Figure 2 depicts an overview of the design of OpenUH based on Open64. It consists of the front-ends with support for OpenMP 3.0 and Coarray Fortran (CAF), optimization modules, back-end lowering phases for OpenMP and coarrays, portable OpenMP and CAF runtimes, a code generator and IR-to-source tools. Most of these modules are derived from the corresponding original Open64 modules. OpenUH may be used as a source-to-source compiler for other machines using the IR-to-source tools. We have undertaken a broad range of infrastructure development in OpenUH to support our research in parallel languages, static analysis of parallel programs, performance collection and analysis, and parallel runtime

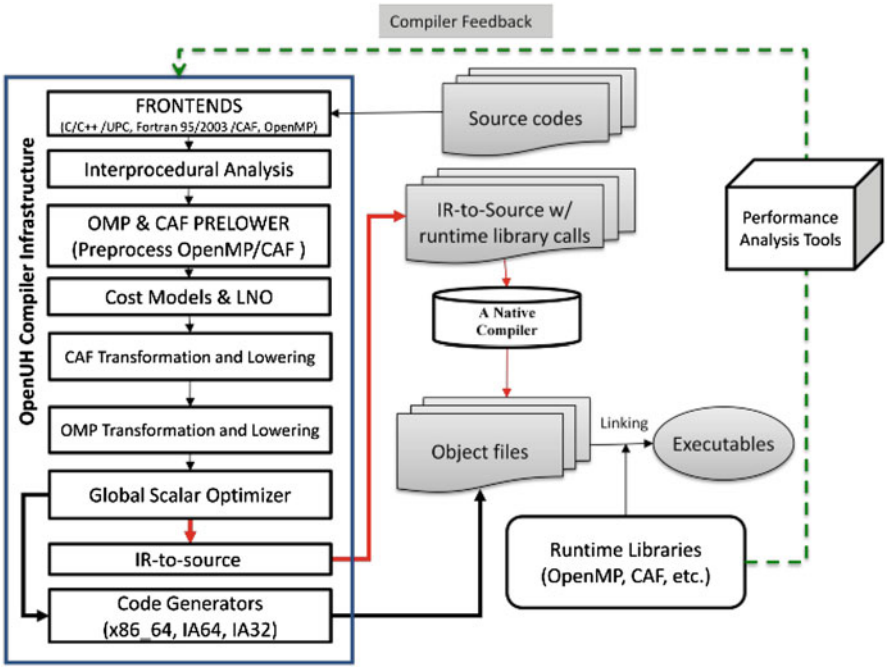

Fig. 2 The OpenUH Compiler/Runtime Infrastructure

systems [2,37,38,28,31]. In the following sections we describe how we used OpenUH to support our research work.

3.1 Support for Scalable OpenMP

A major goal in developing OpenUH was to provide a robust OpenMP compiler for C/C++/Fortran which can generate high-level source code or optimized binaries. We improved the underlying Open64 infrastructure for OpenMP, including adding better support for IR-to-source output, making the data flow analysis "OpenMP-aware", enabling automatic scoping of variables in parallel loops, handling nested parallelism, and adding performance collection support into the runtime system. We have also implemented compiler translation strategies for retargeting existing OpenMP applications to distributed memory systems [34,21].

Anticipating the hardware trends of rapidly increasing degrees of parallelism, a major thrust of our research has also been exploring strategies for making OpenMP more scalable. The major bottleneck for OpenMP in this regard is its shared memory model, for which there is no notion of affinity between shared data and individual threads. To address this issue, we have used OpenUH to implement support for thread subteams [33], as well as locality control via data distribution and co-locating tasks with their associated data [36]. Global barriers can also hinder scalability for large thread team sizes, so we have also implemented a variety of scalable barrier

algorithms [55] in our runtime system which may be seleced via a control variable by the user.

3.1.1 Thread Subteams

From our experience of parallelizing different applications with OpenMP [15], we found tha the inability to control the assignment of work to subsets of threads in the current thread team and to orchestrate the work of different threads artificially limited the performance that we could achieve on large-scale systems. In order to overcome the first difficulty, we proposed a new clause, `onthreads`, for work-sharing constructs that assigns the work to a subteam of the existing threads. The clause supports flexible worksharing among the logical thread subteams. Additionally, the proposed extension added new library routines to the OpenMP API for creating and querying subteam information. The code enclosed within a work-sharing region will be executed by the threads specified in the `onthreads` threadset, which consists of some or all of the threads that encounter the construct. Other threads may proceed past this construct, much as they would if they encountered an *omp single* with a *nowait*. We would consider all threads to have "encountered" any implicit barrier associated with the construct, but only the threads specified in the threadset wait on the barrier. If no *onthreads* clause is present, the threadset is equal to all threads in the current team (in other words, there is no change).

We extended the translation by processing the new `onthreads` clause. The compiler generates a new function call, *ompc_subteam_create*, whenever the translation encounters the clause. This routine creates a new global data structure for subteam and return its address to a pointer. Note that the subteam structure needs to be shared by the team of threads in order to synchronize them. We then pass the pointer into all related OpenMP runtime functions, such as scheduling and barrier. To preserve the backward compatibility, we pass a NULL pointer to these functions if there is no subteam specified on an OpenMP worksharing directive. Therefore, the runtime functions are able to distinguish if only a subteam of threads or the whole team of threads need to participate in the worksharing. Figure 3 shows an `OMP DO` construct with the subteam clause specified and its corresponding compiler translation by OpenUH. The source code is generated using the IR-to-source capability of OpenUH. A subteam structure *mp_subteam*_7937 is created and passed into scheduling and barrier functions. The scheduling function *ompc_static_init*_4 returns an empty workload to threads that are not belonging to the current subteam. These threads are not waiting in the barrier function *ompc_barrier* either.

Our experiments demonstrated that the subteam concept is easy to use and can greatly enhance the scalability of code. Together with colleagues at NASA Ames Research Center, we evaluated the performance of four versions of the NAS BT Multi-zone benchmarks using OpenMP nested parallelism (2 versions), OpenMP with the subteam implementation, and hybrid MPI+OpenMP [41] on an SGI Altix system with 512 Itanium 2 processors. Figure 4 presents the results. The experiments were conducted on an SGI Altix 3700BX2 system with 512

```
!$OMP DO ONTHREADS(1:NUMTHREADS-1:1)
      do i=1,N
         A(i) = i
      enddo
!$OMP END DO
```

(a)

```
tmp0 = ompc_subteam_create(mp_subteam_7937,
 %val(mplocal_t$5), %val(mplocal_t$6), %val(mplocal_t$7))
tmp0 = ompc_static_init_4(%val(ompv_temp_gtid),
 %val(1), ompv_temp_do_lower, ompv_temp_do_upper,
 mpv_temp_do_stride, %val(1), %val(mplocal_t$8),
 mp_subteam_7937)
DO WHILE(ompv_temp_do_lower .LE. temp_limit)
 IF(ompv_temp_do_upper .GT. temp_limit) THEN
  ompv_temp_do_upper = temp_limit
 ENDIF
 DO mplocal_I = ompv_temp_do_lower, ompv_temp_do_upper, 1
  A(mplocal_I) = mplocal_I
 END DO
 ompv_temp_do_lower = (ompv_temp_do_lower
   + ompv_temp_do_stride)
 ompv_temp_do_upper = (ompv_temp_do_upper
   + ompv_temp_do_stride)
END DO
tmp0 = ompc_barrier(mp_subteam_7937)
```

(b)

Fig. 3 A compiler translated OpenMP code with the subteam clause. **a** The original OpenMP program with subteam. **b** The corresponding compiler translated code

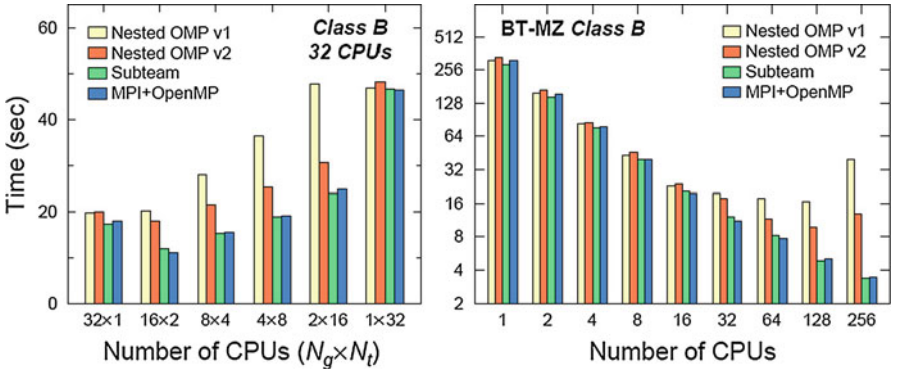

Fig. 4 Comparison of subteams with equivalent versions of BT multizone benchmark

Itanium 2 processors, which is one of the 20 nodes comprising the Columbia supercomputer installed at the NASA Ames Research Center. Our experiments demonstrated that the subteam and hybrid versions are close in performance since both enable a similar data layout and reuse the data efficiently. Moreover, the code for the subteam version turned out to be much simpler than the other versions.

3.1.2 Locality Language Extension

A major limitation for OpenMP relating to scalability is that its memory model assumes a flat shared memory space with uniform access time. Given the fact that memory systems for large-scale parallel systems are generally hierarchical and exhibit non-uniform access time, we believe that it is important to introduce new features to OpenMP to manage the data layout and co-locate tasks with data to exploit locality. We have developed extensions to allow the the programmer to 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 specify data layout with respect to them. A location is a logical construction for grouping a set of co-located tasks and their associated data. The proposed extension entails a new environment variable, `OMP_NUM_LOCS`, which defines the number of locations used in the application. The execution of parallel work and the layout of shared data is done with respect to a specified set of locations, using the `onLoc` clause and the `distribute` directive. We refer the reader to [37] for a more detailed description.

We have developed support for the proposed extensions in the OpenUH compiler. Runtime support for thread binding and locality management has been incorporated into the OpenMP runtime using libnuma, a library in Linux systems for supporting NUMA systems. 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 analyzed how support for data layout management (with the `distribute` directive) and affinity specification in parallel loops (with the `onLoc` clause) can impact performance, tests we refer to as data-transposition and loop-affinity respectively. The OpenUH compiler was installed on both the SGI Altix and the 48-core AMD system. Figures 5 and 6

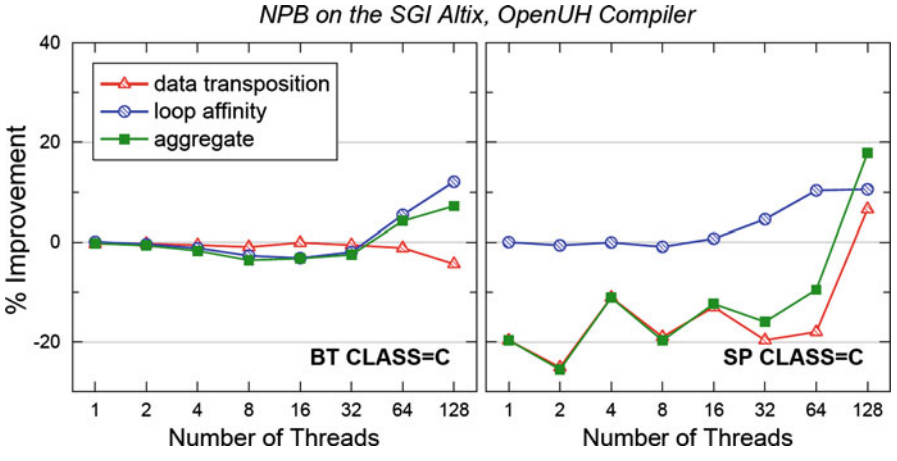

Fig. 5 Performance comparison on the SGI Altix using the OpenUH compiler

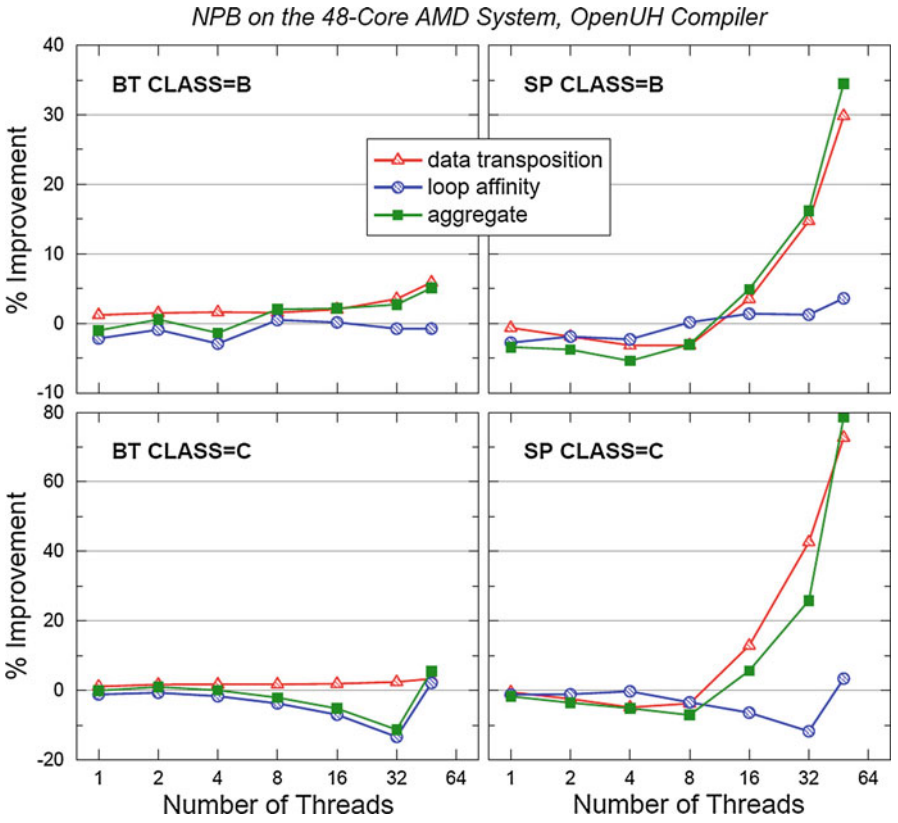

Fig. 6 Performance comparison on the 48-core AMD system using the OpenUH compiler

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 show the combined effects of expressing data layout and affinity with our extensions. Negative values indicate performance degradation. For the SGI Altix system (see Fig. 5), the results showed performance improvement at large thread counts (32) from expressing loop affinity with our directives. However, we observed substantial performance degradation (20 %) from data transposition for SP and no improvement for BT. On the 48-core AMD system (Fig. 6), there is no performance gain from applying loop affinity; in fact, negative effects are observed for the Class C problem. 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.

The notion of data layouts via distribution and affinity with loop iterations with `onLoc` allows a user to carefully optimize data layout with the data access pattern and, thus, achieve performance gain on large NUMA systems From the experiments,

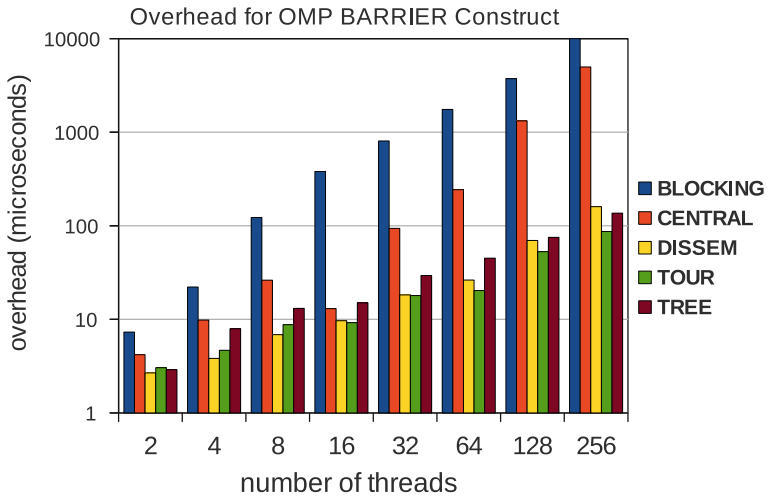

Fig. 7 Overhead measurements for the different barrier algorithms in OpenUH runtime

we observe significant performance impact from different data layouts on the NUMA system, especially for larger data sets. However, there is still considerable discrepency in the performance impact for multi-core systems versus distributed shared memory systems. We are working on improving the implementation to achieve better performance portability across different parallel architectures.

3.1.3 Barrier Enhancements

Another feature we have added to enhance the overall performance of OpenMP code is to make available several different implementations of barrier operations in the runtime library [55]. We have extended our runtime to accept a user-specified barrier algorithm best suited for the application's needs on a specific architecture and given number of threads. The algorithms supported are blocking, central, dissemination, tour and tree. Figure 7 shows the time (in microseconds) of the different barrier algorithms in a SGI 3600 Altix system up to 256 threads. After evaluating the different barrier algorithms with two fluid dynamic applications: GenIDLEST and ASPCG, we found out that the best barrier algorithm depends on the application characteristics (i.e. memory intensity) and the total number of threads. Table 1 shows these results.

3.2 Support for OpenMP Tasking Model

OpenUH also includes support for OpenMP 3.0 tasks. This consists of front-end support ported from the GNU C/C++ compiler, back-end translation we implemented jointly with Tsinghua University, and an efficient task scheduling infrastructure we have developed in our runtime library. We have implemented a configurable task pool framework that allows the user to choose at runtime an appropriate task queue organization to use. This framework also allows for fast prototyping of new task pool designs.

Table 1 Best barrier algorithms for ASPCG and GenIDLEST

Number of threads	ASPCG	GenIDLEST
2	Tournament	Blocking
4	Dissemination	Blocking
16	Tournament/tree	Dissemination
32	Tournament	Tournament
64	Tournament	–
128	Dissemination	–

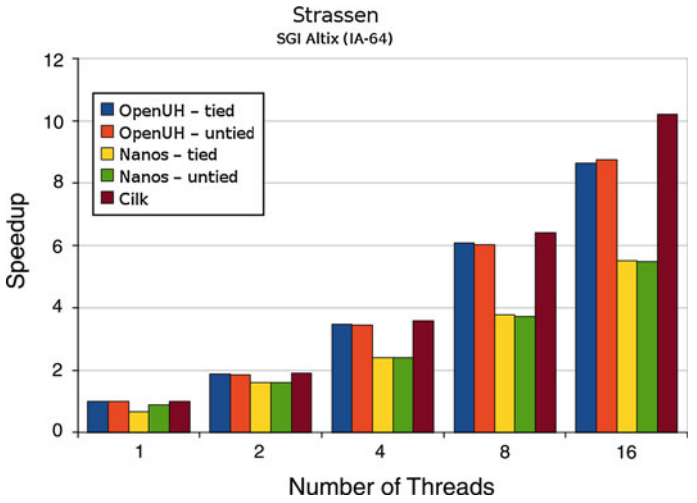

Fig. 8 Speedup with tasking runtime on SGI Altix 350 system

Furthermore, the user may control the order in which tasks are removed from a task queue for greater control over task scheduling. We have merged these recent improvements in our runtime (including improved nested parallelism and tasking support) into the official OpenMP 3.0 branch in the Open64.net source repository.

Figure 8 shows the results from the Strassen benchmark comparing our tasking runtime, based on the Portable Coroutines Library, 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 16 GB 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

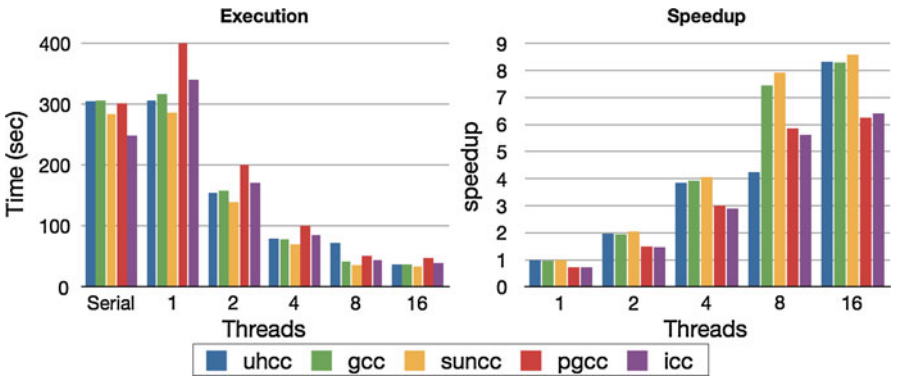

Fig. 9 Performance of NPB BT-MZ with tasking runtime on dual Nehalem E5520 machine

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 implementation are shown in Fig. 9 using 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 32 KB L1 and 256 KB 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: (OpenUH) uhcc compiler,—O2—LNO; (GNU C compiler) gcc 4.6.1,—O3—fargument-noalias-global; (Oracle) suncc 5.11, -xO3; (PGI) pgcc 11.7, -fast; and (Intel) 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). For more details on the runtime implementation for tasks in OpenUH, the reader may refer to [43].

3.3 Coarray Fortran Compilation

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, our approach is to use a single, unified compiler infrastructure to translate, optimize and generate binaries from CAF codes. CAF support in OpenUH [22] comprises three areas: (1) an extended front-end that accepts the coarray syntax and related intrinsic functions, (2) back-end optimization and translation, and (3) a portable runtime library.

3.3.1 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 [20], 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.

3.3.2 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:

$$A(i, j, 1:n)[q] = B(1, j, 1:n)[p] + C(1, j, 1:n)[p] + D[p] \tag{1}$$

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:

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

For each expression of the form $t \leftarrow R(\ldots)[\ldots]$, the compiler generates an allocation for a *local communication buffer* (LCB) t of the same size as the array section

$R(\ldots)$. 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 $L(\ldots)[\ldots] \leftarrow t$ follows a similar pattern, except the compiler generates a `PUT` runtime call.

```
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 in the initial lowering phase for the statement representation given in (2). A subsequent optimization will convert `GET` and `PUT` calls to non-blocking optimization and use data flow analysis to overlap communication with computation and potentially aggregate messages, similar to work described in [17] which was also done in an Open64-based compiler.

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 it is 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, message vectorization where the Fortran 90 array section syntax is not specified by the programmer, and generating non-blocking communication calls where it is feasible and profitable.

3.3.3 Runtime

The implementation of our supporting runtime system relies on an underlying communication subsystem provided by ARMCI [56] or GASNet [9]. 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. Because CAF has a relaxed consistency memory model, we get perform optimizations to cache remote coarray data and prefetch data. 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.

3.3.4 Evaluation Using Seismic Code

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 (2,640 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 24 GB 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 40 Gbps. 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 (the OpenUH implementation of CAF) and MPI. In order to isolate performance over the communication network, we ran these experiments with only one process per SMP node.

The *Titled Transverse Isotropic (TTI) Wave Equation* code models an-isotropic media and 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 Open MPI version uses traditional assumed shape array declarations instead of dynamic allocation (to prevent performance impact). The program is executed twice

Table 2 Total execution time (seconds) for 16 GB domain size

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

with Intel MPI, using the *xhost* flag, which tells the Intel compiler to optimize for the specific hardware.

Table 2 compares the total execution time of the TTI program for the OpenUH CAF implementation and various MPI implementations. The matrix dimensions are $1,024 \times 2,048 \times 2,048$ with 4 ghost points. The buffer size in the table is the sum of all the communication buffer of all processes. 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. The CAF performance outperforms the Open MPI implementation by a significant margin, and yields similar performance on this code to the tuned Intel MPI implementation.

3.4 Compiler Analysis for Parallel Programs

Traditionally, compiler analysis has dealt with sequential programs. The non-determinism introduced by parallel programming models and the sheer complexity of parallel architectures has made effective static analysis for parallel programs a significant challenge. We have developed new analyses in OpenUH to enable the compiler the reason about the parallelism expressed in applications and expose these results to users and tools. In this section, we describe extensions to the OpenUH cost model that we developed for OpenMP parallel loops as well as extensions to the data flow analysis for shared memory parallelism.

3.4.1 Cost Models for Parallel Programs

OpenUH includes a set of cost models inherited from Open64 loop nest optimizer (LNO) [68] that can be used to estimate, in CPU cycles, the cost of executing singly nested loop (SNL) nests. SNL loop nests comprise perfectly nested loop nests, and imperfect ones that are eligible to be transformed into perfect ones. The compiler uses its cost models to choose a combination of different loop level optimizations, including transformations such as arbitrary loop interchange, tiling and outer loop unrolling. The cost model may also guide automatic parallelization. There are three major models: the *processor model*, the *cache model*, and the *parallel model*. The *processor model* is used to estimate the CPU cycles needed to execute one iteration of an SNL loop, without considering latencies from the memory hierarchy, while taking into account register spilling and dependencies between memory operations. The *cache model* helps predict cache misses the associated penalty cycles required to execute inner loops. The *parallel model* is used to predict the costs of parallelizing a given loop, taking into account overheads from the fork/join operations.

$$
\begin{aligned}
Parallel\ region_c &= Fork_c + \sum_{j=1}^{n}[maximum(Thread_0\ exe\ j_c, ..., Thread_{n-1}\ exe\ j_c)] + Join_c \\
Thread_i\ exe\ j_c &= Work\ sharing_c + Synchronization_c \\
Work\ sharing_c &= Omp\ for_c/Omp\ sections_c/Omp\ single_c \\
Synchronization_c &= Master_c/Critical_c/Barrier_c/Atomic_c/Flush_c/Lock_c
\end{aligned}
$$

Fig. 10 Equations of cost model for OpenMP

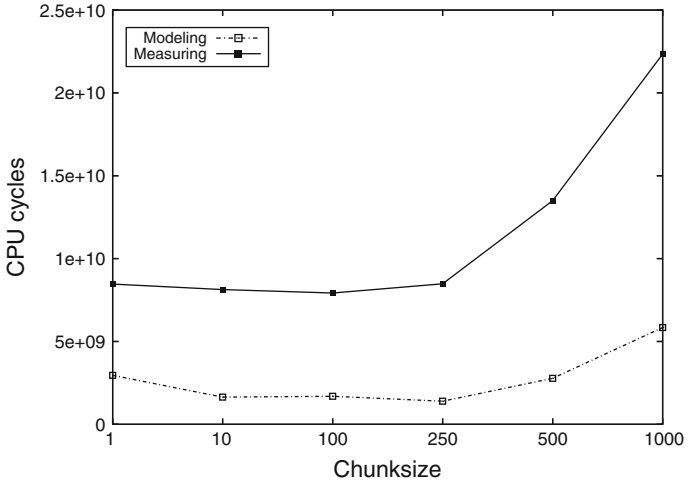

Fig. 11 Modeling schedule(static,n) for matrix-matrix multiply on 4 threads

We extended the existing parallel model to model explicit parallel constructs in OpenMP programs, which we called an OpenMP cost model. OpenUH models the cost for each encountered parallel regions, which may in turn contain multiple work-sharing regions and synchronization constructs. The formulas used for this are shown in Fig. 10, and a more detailed explanation of the models can be found in [46]. We evaluated our cost model using a classic parallel matrix-matrix multiplication (MMM) kernel, which has also been widely used in previous research [67,69] due to its importance in scientific numerical computation. The results of modeling OpenMP with `schedule` clause is given in Fig. 11 using array size $1{,}000 \times 1{,}000$ with 4-thread execution. Only `static` scheduling results are shown because `dynamic` and `guided` scheduling have very similar results. While our model has clear room for improvement in terms of absolute accuracy, its ability to capture relative performance using varying iteration chunk sizes for the loop schedule is sufficient to help guide OpenMP compilation or provide hints back to the user.

We have also worked to create a framework for performance modeling of hybrid OpenMP and MPI applications [4,3]. We designed our model to capture the communication and computational overheads introduced by the OpenMP and MPI programming models. Our methodology is to combine static information about the application and a system profile to model expected application performance. We used the OpenUH compiler to create an application signature via static analysis of the source code. Then, we

used the Sphinx benchmark and Perfsuite to generate profiles that detail performance information for various communication operations and overheads due to parallelization on the system. The static information is saved into XML format to allow easy inspection by developers and to also allow other performance and analysis tools to exploit it.

3.4.2 Parallel Data Flow Analysis

In past work [35,38], we have designed and implemented an extension to the data flow analysis framework (PDFA) in OpenUH to describe data flow between concurrently executing threads in an shared memory parallel regions. For this work, we implemented a Parallel Control Flow Graph (PCFG) for representing OpenMP programs in order to enable aggressive optimizations, while guaranteeing correctness. The PCFG is not unlike the Program Execution Graph and the Synchronized Control Flow Graph proposed by other researchers [7,14]. The distinction between our PCFG and their flow-graph is that ours is based upon the relaxed memory consistency model of OpenMP, and its barrier and flush synchronizations instead of event-based synchronizations (such as signal-wait). We have also added support for Parallel SSA (PSSA) form, an extension of SSA that represents reaching definitions for shared variables within parallel regions. We incorporate ψ- and π-functions into our representation, based in part on work by Lee et al. [44].

Data flow analysis for UPC was performed earlier in Open64 in the Berkeley UPC compiler [16], but this was concerned with intra-thread data flow. We are currently expanding on our OpenMP PDFA framework so that it may be used for data flow analysis of PGAS implementations such as Coarray Fortran. In this context, we are interested in dependencies that exist for statements executing on different images, and exploring how static analysis can be used to reduce communication and synchronization costs.

3.5 Instrumentation and Performance Analysis

OpenUH provides a complete compile-time instrumentation module covering different compilation phases and different program scopes. We have designed a compiler instrumentation API that can be used to instrument a program. It is language independent to enable it to interact with performance tools such as TAU [50], VampirTrace and KOJAK [54] and support the instrumentation of Fortran, C and C++. The instrumentation module in OpenUH can be invoked at six different phases during compilation, which come before and after three major stages in the translation: inter-procedural analysis, loop nest optimizations, and SSA optimizations. For each phase, the following kinds of user regions can be instrumented: functions, conditional branches, switch statements, loops, call sites, and individual statements. Each user-region type is further divided into subcategories when possible. For instance, a loop may be of type do-loop, while-loop. Conditional branches may be of type if-then, if-then-else, true-branch, false-branch, or select. MPI operations are instrumented via PMPI so that the compiler does not instrument these call sites. OpenMP constructs are handled via

runtime library instrumentation, where it captures the fork and joint events, implicit and explicit barriers. Procedure and control flow instrumentation is essential to relate the MPI and OpenMP-related output to the execution path of the application, or to understand how constructs behave inside these regions.

3.5.1 Selective Instrumentation

We take advantage of the interprocedural analysis within the compiler to reduce the number of instrumentation points. We adapted the inlining methodology to enable selective instrumentation, for which we have defined a cost model in the form of procedures scores. The instrumentation algorithm locates procedures that are significant and are infrequently called and have large bodies. We call a procedure significant if it contains many callsites and is well connected in the callgraph. Our cost model consists of three metrics in the form of instrumentation scores. The first metric computes the weight of the procedure using the compilers control flowgraph, which is defined as PUweight = (5 $\times$ total basic blocks) + total statements + total callsites. As can be seen, this metric puts emphasis on procedures with multiple basic blocks. If run- time information is known, the PUweight formula will use the number of times or effective number of basic blocks, statements and callsites invoked at runtime. The other metric we use is the frequency with which a procedure is invoked, taking their position within loop nests into account. The formula used is: PUloop_score = (100 $-$ loopnest level) $\times$ 2,048. This formula gives higher scores to procedures invoked with fewer nesting levels. The third metric is a score that quantifies how many calls exist within a procedure. PUcallsite_score = (callsites in callee) $\times$ 20,482. This formula gives a small score to procedures invoked as leaf nodes in the callgraph or that have few calling edges. The constants of the formulas were determined empirically based on the inlining algorithm of the compiler which was tuned to avoid under or over inlining. Our assumption here is that important procedures are connected with others, and thus are associated with several edges in the callgraph. It is important to note that we will not count callsites to procedures that are not going to be instrumented. The overall score used to decide whether we will instrument a procedure is as follows: $InstrumentationScore = PUweight + PUloop\_score + PUcallsite\_score$.

Our strategy for computing this score means that we will favour procedures with large bodies, invoked few times and with multiple edges connecting them to other procedures in the callgraph. We avoid the instrumentation of small procedures invoked at high loopnest levels and that are leaf nodes in the callgraph. With this score we then define a threshold that can be changed depending on the size of the application, in order to avoid over or under-instrumentation. Also, we generalize our approach to take into consideration the lowest score that a procedure has from its different callsites. If a score for a procedure is below a pre-defined threshold, the procedure will not be instrumented. *Instrument Procedure* $<$ *Threshold* $<$ *Do not Instrument* When this method is applied to the NAS benchmarks, we were able to reduced significantly the overhead for profiling. Figure 12 shows the overhead of using selective instrumentation versus full procedure instrumentation using TAU in a Altix 3600 system using four

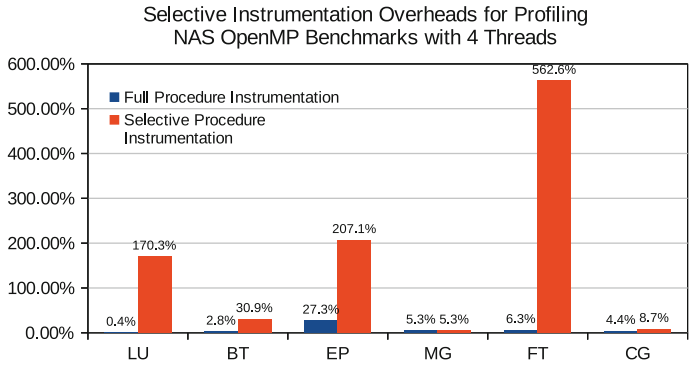

Fig. 12 The overhead of selective instrumentation versus full instrumentation in the NAS OpenMP parallel benchmarks

threads. We can see that our methodology reduces the overall profiling overhead in the code.

3.5.2 Compile-Time Instrumentation: A Case Study

In this section, we present some performance numbers gathered in the tuning environment. The OpenUH compiler selectively instruments the GenIDLEST with APIs connected with performance tools TAU and KAJAK. The functionality provides a great convenient way to gather and analyze performance data with little overhead. By using the environment, the purpose of the work is to understand why the performance the OpenMP version of GenIDLEST is slower than the corresponding MPI program on a distributed shared memory system. The OpenMP version is slower by a factor of 2.5 times when we used 8 and 16 threads. The procedure diff_coeff and pc_implicit form part of the main computational phase and they consume most of the computational time in our profiling. We then further apply a do loop level instrumentation automatically using OpenUH, and we found that there are two main computational loops taking significant amount of time in the procedure diff_coeff up with eight OpenMP threads.

Using the performance algebra from CUBE (which is integrated with OpenUH), we can see that the metric that most varies among the different loops are the exception and flush counters. Exception and flush counters indicate that the memory access exceptions happens due to page faults and requires system handling. We believe that the reason is that there are much more remote memory accesses in the OpenMP version that over saturation of the NUMA link to fetch data. Delays in the interconnect access are probably the cause of exceptions and make the processor to flush data or context switch. To optimize the code we applied the data privatization and data placement strategy and the performance of the procedure is greatly improved. Figure 13 shows the performance after privatization and data placement strategy is applied, which is similar to the performance show in for the MPI version. By conducting the optimization, the procedure gains 10 times speedup, and it improves the overall application performance by 20 %.

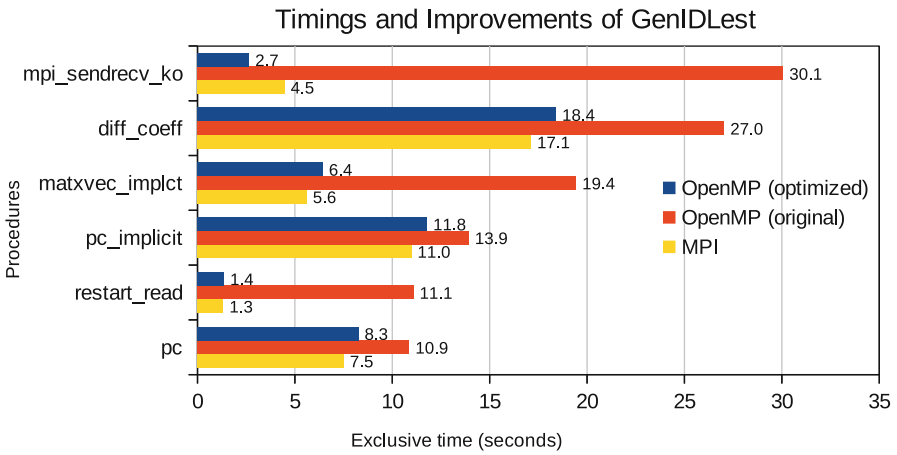

Fig. 13 Timings for GenIDLEST procedures before and after privatization. It also contains the timings for the MPI version of the code

3.5.3 OpenMP Collector API

The OpenMP Collector API was proposed as a standard means of enabling performance tools to interact with OpenMP implementations [40]. The collector's event-based interface provides bi-directional communication between the OpenMP runtime library and performance tools, thereby overcoming the lack of standard interfaces in the runtime layer. A performance tool that utilizes the collector interface may gather information about a program's execution from the runtime system by providing callback handlers for specific OpenMP collector events. The runtime library will notify the collector tool when the execution reaches a specific point that corresponds to the registered OpenMP collector event.

We have implemented the collector tool API for OpenMP applications within our runtime [13,30]. Performance tools may issue requests to the runtime library as the application is running to query important state information. To satisfy these requests, we have added support for (1) initiate/pause/resume/stop event generation, (2) responding to queries for the ID of the current/parent parallel region, and (3) responding to queries for the current state of the calling thread. By running an application with a performance tool that uses this API, the user can uncover important information about their program, for example synchronization costs. The goal of this work is to provide a mechanism to collect the OpenMP events with minimal overhead. Figure 14 shows the overhead of our current implementation of the collector API with the NAS parallel benchmarks with different number of threads. The overhead is minimal is below 2 % in most of the benchmarks.

3.5.4 Optimization Framework Based on Collector API

It is often straightforward to develop a shared memory parallel program using OpenMP, difficult to get good performance. be difficult. Deep understanding about a program's

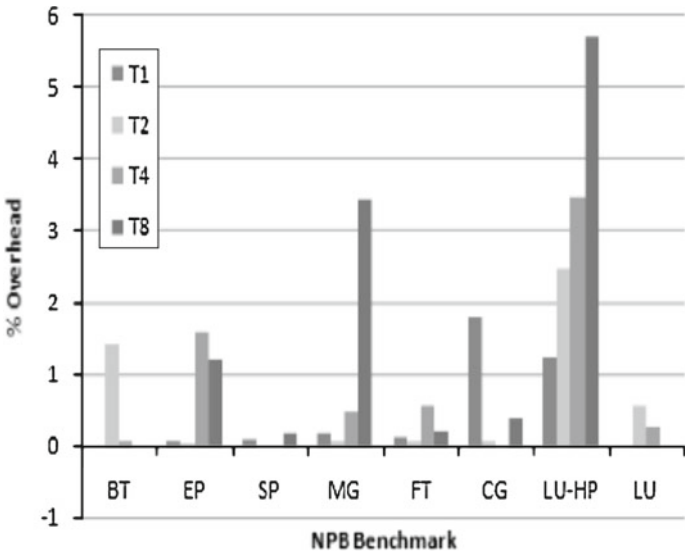

Fig. 14 The overhead of the OpenMP collector API for the NAS parallel benchmarks with different number of threads

dynamic behavior, particularly with respect to its data accesses, is needed in order to improve its behavior. Information on data accesses may enable the compiler, or the application developer, to improve data locality on a NUMA system and avoid more subtle performance problems such as those caused by false sharing. False sharing occurs when two or more threads access data on the same cache line nearly simultaneously and one of the accesses writes data. The cache line will be invalidated when it is written to, and must be refetched from main memory before other threads may use data on the same line. Thus false sharing can lead to substantial performance degradation. Yet it can be very hard for the application developer to detect. In [29], we showed that the OpenMP collector interface introduced above is useful for directing performance data collection by starting and stopping hardware performance counter at specific points. As a result, it can be the starting point for a variety of strategies to collect and exploit data pertaining to dynamic OpenMP program behavior in order to improve the code.

We have designed a collector-based dynamic optimization framework, shown in Fig. 15, that uses the collected performance data as feedback to affect the runtime behavior of the program, and have utilized it to help optimize data accesses in an OpenMP code. The framework utilizes the collector interface to communicate with the OpenUH OpenMP runtime and direct the performance monitoring and program optimization. It uses various open source libraries to collect performance data and apply optimization strategies.

The collector tool coordinates the optimization activity and is thus at the heart of the framework. This component utilizes the collector interface to communicate with the OpenMP runtime and gain insight about a program's execution. The performance monitoring component utilizes the processor's hardware counters to investigate the

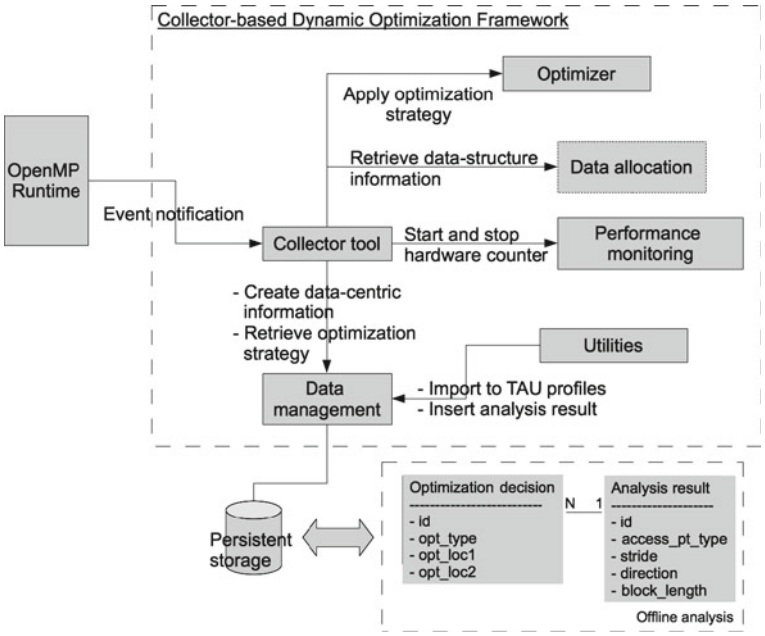

Fig. 15 Dynamic optimization framework using the collector interface

performance characteristics of an application. It uses the *libpfm* library to access a processor's specific features, such as the DEAR [39] that is available on the Itanium2 processor, in order to pinpoint the specific location in the program that is causing performance problems.

We have demonstrated its use by creating the means to track the data layout on a NUMA platform and identify arrays that involve many non-local data accesses; we have also used it to accurately detect memory accesses that give rise to false sharing, which often incurs a high penalty. Our solution requires two phases, one to gather the required performance data and other information that enables the system to map this data to program constructs, and a subsequent phase that uses this information to apply the desired optimizations. In [66], we described how used this framework to help identify false sharing and alleviate its impact to improve scalability and the reader can refer to this paper for a more detailed discussion. In Fig. 16, we see speedup results for several codes in the Phoenix [62] benchmark suite. Our framework was able to identify false sharing problem existed for *histogram*, *linear_regression*, *reverse_side*, *string_match*, *work_count*. Significant improvements were obtained for *histogram* and *linear_regression* using our framework, while modest improvements were obtained for *string_match* and *work_count*.

4 OpenUH in Teaching and Learning

We have used our compiler infrastructure to support our instructional efforts in a graduate course offered to Computer Science students. In that context, the richness of this

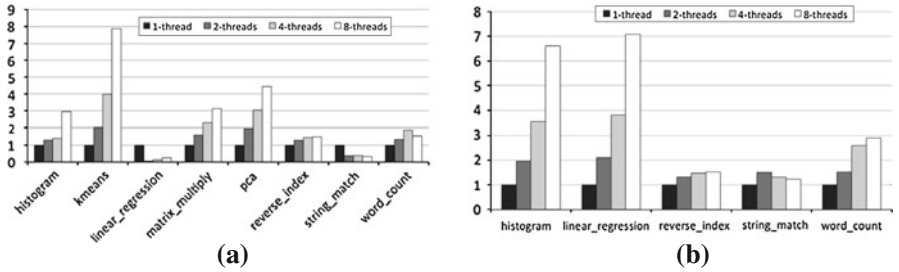

Fig. 16 Alleviating false sharing problem using a dynamic optimization framework. **a** The speedup of the original program. **b** The speedup after adjusting the memory alignment

infrastructure has made it a valuable resource. For example, we have illustrated our discussion of the roles and purposes of various levels of intermediate representation by showing how OpenUH represents selected constructs and simple executable statements. We are able to get students to apply certain features and then output source code. Even though some help is needed to explain the structure of this output code, it offers insight into the manner in which the compiler applies transformations. Moreover, students have routinely successfully carried out minor adaptations to the compiler, or retrieved specific information from its internal structures. Advanced topics such as Hashed SSA representation for indirect memory operations [19], and SSA partial redundancy elimination [18] can be understood more concretely with the help of OpenUH's trace functionality.

Support for parallel computing via efficient implementations of parallel programming models and parallelization strategies is a common theme in our compiler courses. For this purpose, we have used OpenUH to illustrate how parallelism can be represented and analyzed with a modern compiler infrastructure. Students have explored compilation techniques for shared memory models and PGAS models using OpenUH. Because OpenUH shares a common code base with other available compilers based on Open64, familiarity with OpenUH has also helped students study other Open64-based programming model implementations including Berkeley UPC compiler[17] and the HICUDA compiler [27].

5 Future Work

In this paper, we have described our development of the OpenUH compiler and how we have used it to support our research in supporting programming models for high performance computing. Active development in OpenUH is on-going. We are currently developing support for the latest OpenMP 3.1 standard, and exploring new implementations for mapping parallel work to hierarchical and potentially heterogeneous parallel systems. We are also working on new IRs for representing task dependence graphs, applying transformations on them, and executing them efficiently with our runtime task scheduler.

Compiler techniques for PGAS languages is an important area of research that we are heavily involved in. In addition to CAF, a UPC implementation is underway in

OpenUH and should be available by the end of 2012. We will be exploring analyses and transformations for PGAS languages that can be used for both CAF and UPC programs compiled with OpenUH. We are also studying strategies for analyzing and optimizing hybrid codes (e.g. programmed with PGAS languages and OpenMP).

A third major thrust of our future work will be continued development of infrastructure to interface the OpenUH compiler with program analysis and performance tools. We envision a "glass box" paradigm, in which infrastructure is in place for easily sharing information between the compiler and tools in a cohesive development environment.

Acknowledgments We would like to thanks our funding agencies for their support. The work described in this paper was funded by the following grants: National Science Foundation under contracts CCF-0444468, CCF-0702775, CCF-0833201; Department of Energy under contracts DE-FC03-01ER25502, DE-FC02-06ER25759. Support for our CAF implementation was partially sponsored by Total.

References

1. The Open64 compiler. http://www.open64.net (2011)
2. Addison, C., LaGrone, J., Huang, L., Chapman, B.: OpenMP 3.0 tasking implementation in OpenUH. In: Open64 Workshop in Conjunction with the International Symposium on Code Generation and, Optimization (2009)
3. Adhianto, L., Chapman, B.: Performance modeling and analysis of hybrid MPI and OpenMP applications. University of Houston Department of Computer Science, technical report (2006)
4. Adhianto, L., Chapman, B.: Performance modeling of communication and computation in hybrid MPI and OpenMP applications. In: ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems, pp. 3–8. IEEE Computer Society, Washington, DC, USA (2006). doi:10.1109/ICPADS.2006.81
5. Adve, V.S., Bagrodia, R., Browne, J.C., Deelman, E., Dube, A., Houstis, E.N., Rice, J.R., Sakellariou, R., Sundaram-Stukel, D.J., Teller, P.J., Vernon, M.K.: Poems: end-to-end performance design of large parallel adaptive computational systems. IEEE Trans. Softw. Eng. **26**(11), 1027–1048 (2000). doi:10.1109/32.881716
6. Balart, J., Duran, A., Gonzalez, M., Martorell, X., Ayguade, E., Labarta, J.: Nanos Mercurium: a research compiler for OpenMP. In: The 6th European Workshop on OpenMP (EWOMP '04). Stockholm, Sweden (2004)
7. Balasundaram, V., Kennedy, K.: Compile-time detection of race conditions in a parallel program. In: ICS '89: Proceedings of the 3rd International Conference on Supercomputing, pp. 175–185. ACM Press, Crete, Greece (1989). doi:10.1145/318789.318809
8. Beddall, A.: The g95 project. http://www.g95.org/coarray.shtml
9. Bonachea, D.: Gasnet specification, v1.1. Technical report, Berkeley, CA, USA (2002)
10. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14**(3), 189–204 (2000). doi:10.1177/109434200001400303
11. Brunst, H., Kranzlm ller, D., Nagel, W.E.: Tools for scalable parallel program analysis Vampir VNG and DeWiz. In: DAPSYS, pp. 93–102 (2004)
12. Buck, B., Hollingsworth, J.K.: An API for runtime code patching. Int. J. High Perform. Comput. Appl. **14**(4), 317–329 (2000). citeseer.nj.nec.com/buck00api.html
13. Bui, V., Hernandez, O., Chapman, B., Kufrin, R., Tafti, D., Gopalkrishnan, P.: Towards an implementation of the OpenMP collector api. In: PARCO (2007)
14. Callahan, D., Kennedy, K., Subhlok, J.: Analysis of event synchronization in a parallel programming tool. In: PPOPP '90: Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 21–30. ACM Press, Seattle, Washington, USA (1990). doi:10.1145/99163.99167
15. Chapman, B.M., Huang, L., Jin, H., Jost, G., de Supinski, B.R.: Toward enhancing OpenMP's worksharing directives. In: Europar 2006, pp. 645–654 (2006)

16. Chen, W.Y., Bonachea, D., Iancu, C., Yelick, K.: Automatic nonblocking communication for partitioned global address space programs. In: Proceedings of the 21st Annual International Conference on Supercomputing, ICS '07, pp. 158–167. ACM, New York, NY, USA, 2007, 10(1145/1274971), pp. 1274995 (2007). doi:10.1145/1274971.1274995
17. Chen, W.Y., Iancu, C., Yelick, K.: Communication optimizations for fine-grained upc applications. In: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, PACT '05, pp. 267–278. IEEE Computer Society, Washington, DC, USA (2005). doi:10.1109/PACT.2005.13
18. Chow, F., Chan, S., Kennedy, R., Liu, S.M., Lo, R., Tu, P.: A new algorithm for partial redundancy elimination based on ssa form. In: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, PLDI '97, pp. 273–286. ACM, New York, NY, USA (1997). doi:10.1145/258915.258940
19. Chow, F.C., Chan, S., Liu, S.M., Lo, R., Streich, M.: Effective representation of aliases and indirect memory operations in SSA form. In: Computational Complexity '96: Proceedings of the 6th International Conference on Compiler Construction, pp. 253–267. Springer, London, UK (1996)
20. Dotsenko, Y., Coarfa, C., Mellor-Crummey, J.: A multi-platform co-array fortran compiler. In: PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pp. 29–40. IEEE Computer Society, Washington, DC, USA (2004). doi:10.1109/PACT.2004.3
21. Eachempati, D., Huang, L., Chapman, B.M.: Strategies and implementation for translating OpenMP code for clusters. In: Perrott, R.H., Chapman, B.M., Subhlok, J., de Mello, R.F., Yang, L.T. (eds.) HPCC, Lecture Notes in Computer Science, vol. 4782, pp. 420–431. Springer (2007)
22. Eachempati, D., Jun, H.J., Chapman, B.: An open-source compiler and runtime implementation for coarray Fortran. In: PGAS '10. ACM Press, New York, NY, USA (2010)
23. Fahringer, T., Clovis Seragiotto, J.: Aksum: a performance analysis tool for parallel and distributed applications, pp. 189–208 (2004)
24. Fahringer, T., J nior, C.S.: Automatic search for performance problems in parallel and distributed programs by using multi-experiment analysis. In: Proceedings of the 9th International Conference On High Performance Computing (HiPC 2002), pp. 151–162. Springer, Bangalore, India (2002)
25. the GNU compiler collection. http://gcc.gnu.org (2005)
26. Girona, S., Labarta, J., Badia, R.M.: Validation of dimemas communication model for mpi collective operations. In: Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 39–46. Springer, London, UK (2000)
27. Han, T.D., Abdelrahman, T.S.: /hi/cuda: a high-level directive-based language for gpu programming. In: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, pp. 52–61. ACM, New York, NY, USA (2009). doi:10.1145/1513895.1513902
28. Hernandez, O., Chapman, B.: Compiler support for efficient profiling and tracing. In: Parallel Computing (ParCo 2007) (2007)
29. Hernandez, O., Chapman, B., et al.: Open source software support for the openmp runtime api for profiling. In: The 2nd International Workshop on Parallel Programming Models and Systems Software for High-End, Computing (P2S2) (2009)
30. Hernandez, O., Nanjegowda, R.C., Chapman, B.M., Bui, V., Kufrin, R.: Open source software support for the openmp runtime api for profiling. In: ICPP Workshops, pp. 130–137 (2009)
31. Hernandez, O.R.: Efficient performance tuning methodology with compiler feedback. Ph.D. thesis, Houston, TX, USA (2008). AAI3313493.
32. Huang, L., Chapman, B., Kendall, R.: OpenMP on distributed memory via global arrays. In: Parallel Computing (PARCO 2003). DRESDEN, Germany (2003)
33. Huang, L., Chapman, B., Liao, C.: An implementation and evaluation of thread subteam for openmp extensions. In: Programming Models for Ubiquitous Parallelism (PMUP 06). Seattle, WA (2006)
34. Huang, L., Chapman, B., Liu, Z.: Towards a more efficient implementation of OpenMP for clusters via translation to global arrays. Parallel Comput. **31**(10–12) (2005)
35. Huang, L., Eachempati, D., Hervey, M.W., Chapman, B.: Extending global optimizations in the openUH compiler for openMP. In: Open64 Workshop at CGO 2008, In Conjunction with the International Symposium on Code Generation and Optimization (CGO). Boston, MA (2008)
36. Huang, L., Jin, H., Chapman, B.: Introducing locality-awareness computation into openmp. In: IWOMP '10 (2010, submitted)

37. Huang, L., Jin, H., Yi, L., Chapman, B.: Enabling locality-aware computations in OpenMP. Sci. Program. **18**(3), 169–181 (2010)
38. Huang, L., Sethuraman, G., Chapman, B.: Parallel data flow analysis for openmp programs. In: Proceedings of IWOMP (2007)
39. Intel: Intel itanium2 Processor Reference Manual for Software Development and Optimization, vol. 1 (2004)
40. Itzkowitz, M., Mazurov, O., Copty, N., Lin, Y.: White paper: an openmp runtime api for profiling. Technical report, Sun Microsystems, Inc. http://www.compunity.org/futures/omp-api.html. (2007)
41. Jin, H., Chapman, B., Huang, L.: Performance evaluation of a multi-zone application in different openmp approaches. In: Proceedings of IWOMP (2007)
42. Johnson, S.P., Evans, E., Jin, H., Ierotheou, C.S.: The parawise expert assistant widening accessibility to efficient and scalable tool generated OpenMP code. In: WOMPAT, pp. 67–82 (2004)
43. LaGrone, J., Aribuki, A., Addison, C., Chapman, B.M.: A runtime implementation of openmp tasks. In: 7th International Workshop on OpenMP, IWOMP2011, pp. 165–178 (2011)
44. Lee, J., Padua, D.A., Midkiff, S.P.: Basic compiler algorithms for parallel programs. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '99), pp. 1–12. ACM SIGPLAN, Atlanta, Georgia, USA (1999)
45. Lee, S.I., Johnson, T.A., Eigenmann, R.: Cetus an extensible compiler infrastructure for source-to-source transformation. In: LCPC, pp. 539–553 (2003)
46. Liao, C., Chapman, B.: Invited paper: a compile-time cost model for OpenMP. In: 12th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS) (March 2007)
47. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: an optimizing, portable OpenMP compiler. In: 12th Workshop on Compilers for Parallel Computers (2006)
48. Liao, C., Liu, Z., Huang, L., Chapman, B.: Evaluating OpenMP on chip multithreading platforms. In: 1st International Workshop on OpenMP. Eugene, Oregon, USA (2005)
49. Liao, C., Quinlan, D.J., Panas, T., de Supinski, B.R.: A rose-based openmp 3.0 research compiler supporting multiple runtime libraries. In: Sato, M., Hanawa, T., M ller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP, Lecture Notes in Computer Science, vol. 6132, pp. 15–28. Springer (2010)
50. Malony, A.D., Shende, S., Bell, R., Li, K., Li, L., Trebon, N.: Advances in the tau performance system. Performance Analysis and Grid, Computing, pp. 129–144 (2004)
51. Mellor-Crummey, J., Adhianto, L., Scherer, W.: A new vision for Coarray Fortran. In: PGAS '09. Rice University (2009)
52. MetaSim: www.sdsc.edu/pmac/metasim/metasim.html
53. Moene, T.: Towards an implementation of Coarrays in GNU Fortran. http://ols.fedoraproject.org/GCC/Reprints-2008/moene.reprint.pdf
54. Mohr, B., Wolf, F.: KOJAK a tool set for automatic performance analysis of parallel applications. In: Proceedings of the European Conference on Parallel Computing (EuroPar), pp. 1301–1304 (2003)
55. Nanjegowda, R.C., Hernandez, O., Chapman, B.M., Jin, H.: Scalability evaluation of barrier algorithms for openmp. In: IWOMP, pp. 42–52 (2009)
56. Nieplocha, J., Carpenter, B.: 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, pp. 533–546. Springer (1999)
57. for Non-Experts, I.O.T.: www.cepba.upc.es/intone
58. OpenMP: simple, portable, scalable SMP programming. http://www.openmp.org (2006)
59. The OpenUH compiler project. http://www.cs.uh.edu/ openuh (2005)
60. Petersen, P., Shah, S.: OpenMP support in the Intel Thread Checker. In: Proceedings of the Workshop on OpenMP Applications and Tools (WOMPAT). Toronto, Ontario, Canada (2003)
61. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: a tool to visualize and analyze parallel code. In: Nixon, P. (ed.) Proceedings of WoTUG-18: Transputer and Occam Developments, pp. 17–31 (1995)
62. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for multi-core and multiprocessor systems. In: In HPCA O07: Proceedings of the 13th International Symposium on High-Performance Computer, Architecture (2007)

63. de Rose, L.A., Reed, D.A.: SvPablo: a multi-language architecture-independent performance analysis system. In: ICPP '99: Proceedings of the 1999 International Conference on Parallel Processing, p. 311. IEEE Computer Society, Washington, DC, USA (1999)
64. Sato, M., Satoh, S., Kusano, K., Tanaka, Y.: Design of openmp compiler for an smp cluster. In: In EWOMP '99, pp. 32–39 (1999)
65. TAU Tuning and Analysis Utilites. http://tau.uoregon.edu (2008)
66. Wicaksono, B., Tolubaeva, M., Chapman, B.M.: Detecting false sharing in openmp applications using the darwin framework. In: In Proceedings of 24th International Workshop on Languages and Compilers for Parallel Computing (2011)
67. Wolf, M.E., Maydan, D.E., Chen, D.K.: Combining loop transformations considering caches and scheduling. In: MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, pp. 274–286. IEEE Computer Society, Washington, DC, USA (1996)
68. Wolf, M.E., Maydan, D.E., Chen, D.K.: Combining loop transformations considering caches and scheduling. Int. J. Parallel Program. **26**(4), 479–503 (1998). doi:10.1023/A:1018754616274
69. Yotov, K., Li, X., Ren, G., Cibulskis, M., DeJong, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P., Wu, P.: A comparison of empirical and model-driven optimization. In: PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pp. 63–76. ACM Press, New York, NY, USA (2003). doi:10.1145/781131.781140