A Hybrid Approach of OpenMP for Clusters

Okwan Kwon Fahed Jubair Rudolf Eigenmann Samuel Midkiff

School of Electrical and Computer Engineering, Purdue University
West Lafayette, IN, 47907, USA
{kwon7, fjubair, eigenman, smidkiff}@purdue.edu

Abstract

We present the first fully automated compiler-runtime system that successfully translates and executes OpenMP shared-address-space programs on laboratory-size clusters, for the complete set of regular, repetitive applications in the NAS Parallel Benchmarks. We introduce a hybrid compiler-runtime translation scheme. Compared to previous work, this scheme features a new runtime data flow analysis and new compiler techniques for improving data affinity and reducing communication costs. We present and discuss the performance of our translated programs, and compare them with the performance of the MPI, HPF and UPC versions of the benchmarks. The results show that our translated programs achieve 75% of the hand-coded MPI programs, on average.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Optimization

General Terms Languages, Algorithms, Performance

Keywords OpenMP, MPI, Translator, Optimization, Hybrid, Runtime Data Flow Analysis, Runtime Environment

1. Introduction

The fundamental goal of the system described in this paper is to enable improved programming models for clusters, with a focus on laboratory-scale clusters of ten to one hundred nodes. At the same time, we aim to increase the portability of existing parallel codes written in OpenMP to cost-effective, larger machines.

The state of the art in programming cluster platforms is MPI – the "assembly language of parallel processing". Message passing programs reflect the underlying architectures' distributed address spaces. In MPI, programmers must explicitly partition the programs' data and insert messages to communicate between processors that own data and those that reference the data, which is a tedious and error prone process.

Several attempts have been made in the past to provide a higher-level programming abstraction for clusters. Shared, or global, address spaces promise higher productivity, as software engineers do not need to deal with data partitioning and communication. The challenge in providing such abstractions is two-fold. The underlying compilers or runtime systems need to translate the shared address space program to target the distributed hardware. This is difficult because the problem of finding the best distribution of computation and data for parallel processors/memories and orchestrating communication into a program that runs efficiently is extremely complex. Furthermore, any implementation scheme faces the classic difficulty of the compiler having overly conservative static knowledge of the full program execution (values of many variables are unknown until runtime) and runtime systems not knowing what will happen later in the program's execution. In addition, it has been pointed out that shared address space models, while promising higher productivity, also hide the complexity and costs of the underlying communication from the programmer and thus are vulnerable to "performance errors". We will comment on the difficulty of the translation from a shared to a distributed address space in Section 3, which leads to our OpenMP-to-MPI compiler-runtime scheme; we will discuss performance errors in Section 4 on performance evaluation.

Our paper is related to previous efforts that also proposed shared address space programming models for cluster architectures. One of the largest efforts was the development of High Performance Fortran (HPF) [11]. There are significant differences between the OpenMP-to-MPI translation approach presented here and that of HPF. Like OpenMP, HPF provides directives to specify parallel loops; however, HPF's focus is on data distribution directives as well as parallelization and scheduling of loop execution based on those directives. Data partitioning is explicitly given by these directives and computation partitioning is guided by them [10]. Data typically has a single owner and computation is performed by the owner of the written data, i.e., the *owner computes* rule. In contrast to HPF, OpenMP has no user-defined data distribution input. Our scheme does not derive computation partitioning from data distribution information. Instead, computation is distributed among processors based on OpenMP directives and data may move between different *dynamic owners* at different times during the program's execution. We compare the performance of our translator with an available HPF compiler.

PGAS (Partitioned Global Address Space) languages, such as UPC [19], Co-array Fortran [15], Titanium [21], and X10 [7], are programming paradigms that have been proposed to ease programming effort by providing a global address space that is logically partitioned between threads. For most cases, the programmer needs to specify the affinity between threads and data. In our work, the programmer writes a standard OpenMP shared memory program and the hybrid compiler/runtime translator converts this program into a message-passing executable. We will compare the performance of our translated OpenMP with available UPC programs.

Another approach to extending the ease of shared memory programming to clusters is the use of a Software Distributed Shared Memory (SDSM) system [8]. SDSM is a runtime system that provides a shared address space abstraction on distributed memory architectures. Researchers have proposed numerous optimization techniques to reduce remote memory access latency on SDSM.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.

Many of these optimization techniques aim to perform pro-active data movement by analyzing data access patterns either at compile-time or at runtime. The model has also been used in a commercial system (Intel's Cluster OpenMP [12]), but is no longer supported. Previous work has shown SDSMs to be substantially inferior to MPI programs, primarily due to the page granularity at which they operate. We will show that our translator comes close to MPI performance for the applications considered in this paper.

The hybrid compiler-runtime scheme applied in this paper also relates to many approaches that attempt to move optimization decisions into runtime [5, 6, 17]. A notable representative of hybrid analysis is Parasol [17]. Parasol's goal is runtime detection of parallelism; it is based on the LMAD array access representation [16], whose complex expressions are being evaluated at runtime. While this scheme and ours both use a hybrid approach, the aims and underlying algorithms are substantially different.

In this paper, we present a compiler-runtime scheme and a fully automated implementation that is able to translate the entire class of *regular and repetitive* applications of the NAS Parallel Benchmarks from their shared-address-space source (written in OpenMP) to message passing programs (using MPI), delivering performance close to hand-coded MPI programs. This paper builds on techniques introduced in [14], which presented a translator for a subset of the applications considered here. We also introduce a new compiler technique that improves data affinity by considering the dynamic owner of the data. Furthermore, this paper presents a novel *runtime data flow analysis technique* to optimize communication. The corresponding analysis is done at compile time in [14]. We will discuss and quantify the intrinsic limitations of the compile-time scheme as well as the overheads of the runtime scheme. In this paper, we call the previous approach [14] and our translation system OMPD-CT (Compile-time) and OMPD-RT (Runtime) respectively.

In this paper we make the following contributions:

- We present a fully automated translator and a runtime system that is able to successfully execute the OpenMP versions of all regular, repetitive applications of the NAS benchmarks on clusters.
- We introduce a hybrid compiler-runtime translation scheme. The scheme features a new runtime data flow analysis technique and a compiler technique for improving data affinity.
- We present and discuss the performance results of our scheme, as well as those obtained with HPF and UPC versions of our benchmarks, using available translators.
- We quantitatively compare compile-time and runtime communication generation schemes. We discuss intrinsic limitations of compile-time techniques as well as overheads of runtime techniques.

The remainder of the paper is organized as follows. Section 2 describes the system on which we build and identifies opportunities for improvement. Section 3 describes our hybrid compiler-runtime system. Performance of seven translated programs and other metrics of interest are evaluated in Section 4, followed by conclusions and future work in Section 5.

2. Foundation and Opportunities

Our translator builds on related work [14] that presented early results for a system that extended OpenMP beyond shared-memory architectures. Our novel contributions improve on this foundation in terms of both performance and application range.

The previous approach (OMPD-CT) translated OpenMP programs to message passing programs in two steps: (1) Translation

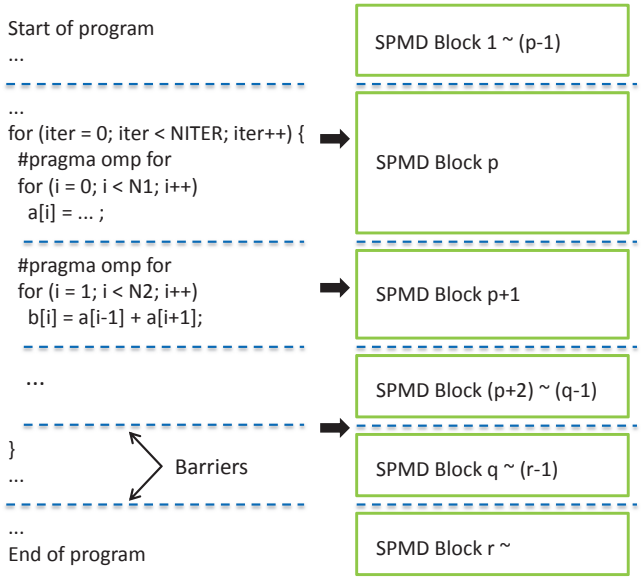


Figure 1: The translation system captures the structure of a regular-loop based OpenMP program as a series of SPMD blocks and barriers between them.

of OpenMP to SPMD form and (2) communication analysis and generation. A key feature of the first step is the partitioning of work onto the distributed MPI processes of the resulting SPMD program. In contrast to related approaches that are guided by data-distribution information, such as HPF, the *iterations of parallel loops* are partitioned and distributed. As a result, data may be written or read by different processes in different parts of a program. Data ownership thus is dynamic. The compiler analyzes the written and read data for each process; communication is generated from this access information, such that data written (defined) is sent to all future readers (uses).

Figure 1 shows the structure of a program for this approach. The unit of compiler analysis in the translated program is an *SPMD block*. Typically, programs contain a series of parallel and serial loops, enclosed by an outer serial loop (often a time-stepping loop). Parallel loops usually generate SPMD blocks that are work-shared, i.e., the iterations are partitioned, while a serial code section generates a replicated block [14]. The compiler analyzes local uses and definitions for each SPMD block. The sets of the local uses and local definitions of process i are denoted by LUSE_i and LDEF_i, respectively. LUSE represents a set of all LUSE_i, and LDEF the set of all LDEF_i, for $0 \leq i \leq nprocs - 1$.

LDEF and LUSE are the results of the array data flow analyses [14] based on the Cetus array section analysis. For each dimension of an array, the accessed region with lower bound (LB) and upper bound (UB) are summarized as [LB:UB]. LDEF and LUSE are analyzed by *Reaching-All Definitions* analysis and *Live-Any* analysis respectively.

In this paper, we consider applications that are *repetitive and regular*. This means that the communication pattern is the same in all iterations of the outer serial loop enclosing the SPMD blocks. The runtime system will evaluate this pattern in the first iteration and reuse it from then on. The programs are regular in that loop bound and array subscript expressions are affine. This requirement is not strict for the analysis of read accesses; the compiler overestimates array ranges (LUSEs) in the presence of non-affine expressions, at the cost of increased communication.

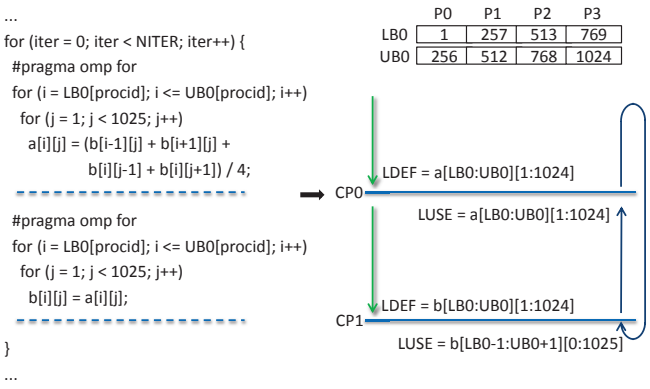


Figure 2: The compiler analyzes producer and consumer information from each SPMD block and puts the results in communication points around the SPMD block.

Communication is generated at the end of each SPMD block, i.e., at a *communication point* (CP). This is done by intersecting the LDEF with the *global use* (GUSE) which is all future LUSEs exposed to this CP. This algorithm is the centerpiece of the communication generation step; it was performed by the compiler in OMPD-CT [14]. The compiler passes the LDEF and GUSE information to the runtime system, which in turn computes their intersection and generates the communication.

Opportunities: The precision of the communication analysis is performance critical. Precise array section analysis allows the communication to be kept to a minimum, as illustrated in Figure 2. If the compiler has insufficient knowledge, it conservatively overestimates the GUSE sets. Even when program expressions do not depend on program input data, overestimation may result from operations on array sections during data flow analysis. In particular, subtractions of symbolic section representations tend to produce expressions of undesirable complexity, warranting approximation. Furthermore, cycles in the control flow graph may necessitate approximations in the computed GUSE sets. Such overestimation was one reason for the limited performance in the previous work [14] for large applications. The key opportunity being exploited in the present paper is to put more responsibility on the runtime, when symbolic expressions have become known values. In Section 4.9, we will show that the needed analysis can be done at runtime with low overhead.

A second opportunity being exploited is the data affinity resulting from the iteration distribution. The previous approach chose a fixed iteration-to-process assignment for all loops, resulting in data unnecessarily changing the dynamic owner, as for example in the FT benchmark. In the present paper, the compiler considers the dynamic data ownership and adjusts data affinity between the data producer and consumer, so as to reduce communication.

3. Hybrid Communication Analysis and Generation System

Our translation system includes a hybrid, compiler-runtime scheme for communication analysis that exploits the opportunities mentioned in the previous section. This section describes the novel elements:

- In addition to the basic OpenMP-to-SPMD translation described above, the compiler performs *Dynamic Owner Alignment*, which adjusts data affinity to reduce communication. This is described in Section 3.1.

Algorithm 1 Dynamic Owner Alignment Optimization

for each cp **in** {all communication points} **do**
  LUSESET ← collect INF LUSEs
  ensure:
    1) Each *LUSE* is from a single read access
    2) The last producer's *LDEF* defines its entire array
  for each *LUSE* **in** *LUSESET* **do**
    dim ← Get the partitioned dimension of the *LDEF*
    lb[], ub[] ← Get the partitioned bounds from the *LDEF*
    stmt ← statement containing the read access of interest
    expr ← array index expression of *stmt*
    Enclose *stmt* by if statement of the form
      if (*lb*[procid] ≤ *expr* ≤ *ub*[procid]) {
        stmt;
      }
    Update the *LUSE* to reflect the new bounds
  end for

  Restore the original loop bounds
end for

- The compiler interacts with the runtime system through a new interface, described in Section 3.2.
- The runtime system performs dynamic array section data flow analysis and generates communication, described in Section 3.3.

3.1 Compiler: Dynamic Owner Alignment

Dynamic Owner Alignment (DOA) is a compiler technique that improves data affinity. DOA adjusts the computation such that data uses happen on the process that previously produced them (i.e., by the dynamic owner). This optimization is able to transform a local USE set from its conservative form ([-INF:+INF]) into a narrow range([LB:UB]), with the effect that communication is eliminated.

Algorithm 1 describes the necessary steps for the transformation. For each LUSE candidate, it ensures the validity of the transformation by checking two conditions. If they are met then the transformation proceeds and the compiler finds the partitioned dimension of the last producer LDEF of the array by searching through the predecessor CPs. Next, it finds the bounds ([LB:UB]) of the partitioned dimension from the LDEF, and inserts an "if statement". The loop reverts to its original bounds and all processes execute all statements. The if-condition selects the execution of those statement instances that use data already present in the current process. Finally, the compiler updates the [-INF:+INF] LUSE by [LB:UB]. With this transformation, the communication between the LDEF and the transformed LUSE is completely removed.

A representative code example before and after the optimization is given in Figure 3. The code shows a reduction operation, where each process computes a partial reduction on a local private copy of the reduction variable *sum* and then performs the global update from the local values. The 1024 elements are summed up via indirection arrays. The initial SPMD transformation block-partitions the iterations. Because of the indirection, the compiler is unable to determine the accessed array ranges, which would result in extensive communication. The DOA optimization causes all processes to revert to their original iteration space, but each operates on the data it currently owns. As a result, no communication is needed. (Note that this transformation resembles but differs from the "owner computes" rules applied by HPF compilers. Owner-computes states that

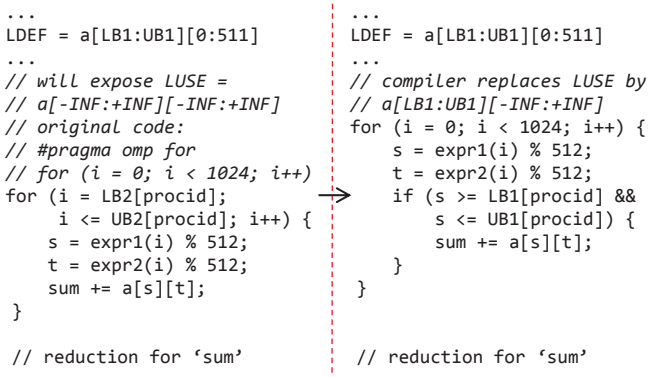


Figure 3: DOA transforms a conservative [-INF:+INF] LUSE expression into a form of [LB:UB] to improve data affinity.

a computation is performed by the process owning (as per the data distribution directives) the data *written* by the computation.)

Section 4.6 discusses the resulting performance improvements of the DOA optimization.

3.2 Compiler-Runtime Interface

The compiler-runtime interface facilitates the interoperation of the two system components. The compiler analyzes the data writes (LDEF) and reads (LUSE) by each SPMD block and informs the runtime system (RTS) thereof. Based on this information, the RTS computes the global uses (GUSEs) exposed at each communication point by performing a runtime data flow analysis. For this purpose, the compiler also passes information about the control flow graph to the RTS. Furthermore, at each communication point, the compiler calls the RTS to activate the requisite part of the data flow computation and schedule needed communication.

The interface is designed to compute the GUSEs efficiently at runtime. Note that LDEFs and LUSEs are already summarized to facilitate efficient GUSE computation at runtime. Since each LDEF/LUSE section is mapped to a communication point, the total number of sections passed to the RTS is proportional to the number of communication points. Furthermore, not all of the communication points are related to an array symbol. Thus, each array symbol maintains a reduced CFG by pruning out unrelated communication points to make the runtime evaluation fast.

The interface functions include `update_def(...)` and `update_use(...)` for informing the RTS of data accesses, `cfg_node(...)` for passing control flow graph information, and `rts_communicate(...)` for actions needed at each communication point. Each function has a communication point identifier as a parameter. Additional parameters include the data range descriptors, CFG connectivity information, and the data type.

The placement of these functions in the translated program is as follows. `cfg_node()` functions are called during program initialization, allowing the RTS to setup a basic CFG structure. The functions `update_def(...)` and `update_use(...)` are usually placed (hoisted) before the outermost serial loop, taking advantage of the fact that communication patterns are repetitive with respect to this loop and, thus, the access ranges are loop invariant. In the absence of such an outer loop, the functions are placed at the corresponding communication points. This information allows the RTS to annotate the control flow graph with local DEF and USE sets. The RTS initializes these annotations to the unknown range [-INF:+INF]. In effect, if the control flow algorithm needs a range before it has become known, it uses a conservative value. The `rts_communicate(...)` function is inserted at each communication point.

3.3 Runtime System: Dynamic Array Section Data Flow Analysis

The role of the RTS is to evaluate global use (GUSE) sections, schedule and execute communications. It has five stages:

1. Construction of Runtime Control Flow Graphs (RCFGs);
2. Update of nodes of an RCFG;
3. Evaluation of the GUSEs of a RCFG;
4. Scheduling communication;
5. Invoking scheduled communications.

3.3.1 Constructing and updating RCFG

A runtime control flow graph needs to be constructed for every array that is involved in communication. The compiler passes necessary information about CFGs collected at compile-time through the `cfg_node(...)` functions. The information includes various identifiers, the shape of the CFGs, and other necessary values. Each RCFG is identified by a unique number, RCFGID. The created RCFG records information about its symbol, such as the base address, the symbol's name string, the element type, dimension, and rank of each dimension. Each node in an RCFG also has its unique number (NODEID) and a corresponding CP number (CPID). With the combination of RCFGID and NODEID a node in a RCFG can be identified and used.

The `update_def()` and `update_use()` functions receive one or more sections and update them in the node of the corresponding RCFG. They check whether the sections are different from the old sections, if they exist, and update only when the new sections are different. They then mark the RCFG as dirty so that GUSEs will be newly evaluated later.

Algorithm 2 shows how `rts_communicate()` does the third, fourth, and fifth stages. When a program execution arrives at a CP, the `rts_communicate()` function examines if there is a RCFG that is associated with the CP and was marked as dirty by any update function call. If there is an RCFG and it is marked dirty, then the RTS performs the evaluation of GUSEs, scheduling and invoking communications. If it is not marked as dirty, then GUSEs of the RCFG are up-to-date and the RTS already has all information to invoke communication functions.

3.3.2 Evaluation of Global Use Sections (GUSE)

The GUSE evaluation uses an any-path backward data flow analysis. The analysis is performed by function `cfg_evaluate()` whenever the corresponding RCFG becomes dirty, as shown in Algorithm 2. The goal is to determine all future use sections seen by a given communication point. This information (GUSEOUT) is then intersected with the local definitions LDEF of the preceding SPMD block to generate communication messages.

Let p be the index of an SPMD block and CP(p) the communication point terminating block p. LDEF(p) represents the local definitions of block p and LUSE(p) are the local uses at CP($p-1$). For process i, GUSEIN$_i(p)$ is the set of sections that is live upon entrance to the SPMD block p. Similarly, GUSEOUT$_i(p)$ is the set of sections of process i that is live upon exiting the SPMD block p. Let Succ(p) be the set of all successors of the SPMD block p in the RCFG. GUSEOUT and GUSEIN for process i can be calculated as follows.

$$GUSEOUT_i(p) = \bigcup_{k \in Succ(p)} GUSEIN_i(k)$$

Algorithm 2 Pseudo code of rts_communicate() and cfg_evaluate().

```
rts_communicate(int cpid) {
  cfg_set ← control flow graphs of arrays involved in cpid
  for all cfg in cfg_set do
    if cfg is marked as dirty then
      call cfg_evaluate(cfg)
    endif
  end for

  schedule communication(cpid)
  invoke communication(cpid)
}

cfg_evaluate(cfg) {
  worklist ← build a work list from cfg in reverse topological order
  while work ← get_work(worklist) exists do
    node ← work's node
    calculate node's GUSEOUT from cfg
    if node's GUSEOUT is changed then
      mark node's GUSEOUT as dirty to reschedule communication
    endif
    calculate GUSEIN from cfg and node
    if node's GUSEIN is changed then
      put predecessors of node in worklist
    endif
    free work
  end while
  mark cfg as clean
}
```

$$GUSEIN_i(p) = \Big(GUSEOUT_i(p) - KILL(p)\Big) \bigcup GEN_i(p)$$

The upward exposed GUSEIN$_i(p)$ should not contain the sections that will be communicated in the CP after the SPMD block p, or redundant communication will happen; KILL(p) is defined as LDEF(p), which includes all LDEF$_i(p)$ for $0 \leq i \leq nprocs - 1$, while GEN$_i(p)$ is the same as LUSE$_i(p)$. Thus, even though the formulas resemble the conventional live variable analysis, our analysis is different in that GEN$_i(p)$, GUSEIN$_i(p)$, and GUSEOUT$_i(p)$ are localized for process i, while KILL spans all processes.

Algorithm 2 uses a work list; it starts from the terminal nodes in reverse topological order. The terminal nodes have empty GUSEOUTs because they do not have successors. The evaluation finishes when the work list becomes empty, which means none of the evaluated sections changed. GUSEOUTs are evaluated for and by all processes.

During the GUSE calculation, we use a precise subtraction method as shown in Fig. 4 to prevent loss of accuracy. It depicts a worst-case fragmentation scenario, where an n-dimensional array incurs the maximum number of fragmented sections, $2n$. The case of merging multiple sections into one is explained in the next section, and the performance of evaluating GUSEs will be discussed in Section 4.10.

3.3.3 Scheduling and invoking communication messages

Recall that the global use sets (GUSEs) express all future uses exposed at a communication point (CP), and local def sets (LDEF) express the data defined in the block terminated by the CP. The above algorithm has computed GUSEs at a given CP. To generate actual communication messages, we now intersects LDEF and GUSE for each pair of processes and for each array symbol. We have implemented efficient intersection functions for the given array range representations.

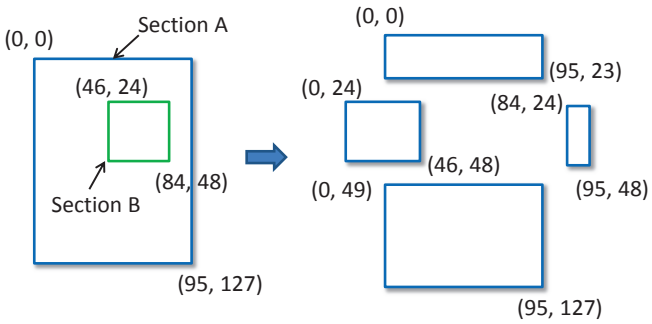


Figure 4: The RTS uses a precise subtraction method to maintain accurate intermediate USE sections. It slices Section A by excluding Section B from the highest to the lowest dimension.

GUSEs, even for a single array, may consist of a number of section fragments. To reduce the number of fragments after doing the intersections, at the cost of increased communication volume, the RTS makes a conservative approximation of them. This approximation extends at most to the boundaries of the intersecting LDEF range, as it would be incorrect to communicate data that has not been defined. Multiple LDEF ranges (e.g., stemming from multiple array access expressions) typically lead to multiple ranges that need to be communicated.

After intersecting, the RTS calls MPI messages to communicate the resulting data ranges. To deal with multiple, non-contiguous array ranges, we make use of the MPI library's ability to accept derived data types for subarrays. Using this rich feature, the MPI communication functions pack and unpack non-contiguous ranges into efficient messages.

The RTS uses different different communication operations. At each CP, the RTS creates a *communication schedule*, expressing the selected message type for each involved array symbol. For example, if the RTS determines that only neighbor communication happens, it chooses point-to-point communication. Determining the communication schedule is the role of the fourth stage of Section 3.3. Our RTS currently chooses between point-to-point with the left and right neighbors, allgather (or scatter), all-to-all, and reduction. The need for the communication type to agree on the sender and receiver side makes this selection non-trivial. We are currently using a simple, but effective heuristic.

`MPI_Alltoallw()` has proven to be an especially useful communication function. It supports generalized all-to-all communication that allows different data types, numbers of elements and displacement of sent and received data. Since it is able to control how many elements to send and receive, and displacement information for each process, it can be configured to be used as point-to-point, gather/scatter for one or all processes, all-to-all or any combination of them. However, its performance is dependent on the performance of the communication library. In our evaluation environment, it is slower than the explicit allgather communication and a point-to-point communication for a process (procid) with its direct left (procid-1) and right (procid+1) processes. Thus, the RTS detects these two types of communication patterns with the given intersection results to use explicit communication functions instead of `MPI_Alltoallw()`. Section 4.8 compares the use of `MPI_Alltoallw()` with `MPI_Allgather()` and `MPI_Isend/MPI_Irecv` followed by `MPI_Waitall()`. (Note, that only the synchronous version of `MPI_Alltoallw()` is available in the common MPICH2 and MVAPICH2 implementations. Improved support is planned in the coming MPICH3 release.)

Finally, the RTS uses the communication schedule just built to generate communication. It calls communication functions according to the communication type stored in the schedule data structure. For example, if the communication type is all-to-all, then the data structure will contain all information for the parameters of `MPI_Alltoallw()`. All communication schedules in a CP will be ordered and executed sequentially because the function is synchronous.

4. Evaluation

In this section, we evaluate the OMPD-RT system with seven publicly available benchmarks ranging from micro-kernels to applications. We provide short descriptions of each benchmark, the number of communication points, and the communication patterns. We compare the overall performance of OMPD-RT applications with their corresponding MPI program. Total execution time and speedup versus serial version is presented, followed by a discussion of the overhead of the runtime GUSE evaluation. In addition, performance comparisons among HPF, OMPD-RT, UPC and MPI are provided.

4.1 Experimental setup

We evaluate the performance on 32 nodes of a community cluster with two MPI processes per node, for a total of 64 MPI processes. The nodes are connected by an InfiniBand network, which provides 10 Gbps of bandwidth. Each node has two quad-core Intel Xeon E5410 processors running at 2.33Ghz with 6MB L2 caches per processor and 16GB of memory. The system is running a 64-bit Linux kernel, version 2.6.18, and MVAPICH2 version 1.5 MPI. We compiled all programs with gcc64 4.4.0 at optimization level 3. For the results in Section 4.11, we used MPICH2 version 1.4 MPI to use 1Gbps Ethernet and MVAPICH2 version 1.5 MPI to use the InfiniBand network. The PGI compiler suite version 11 is used for Fortran, C, and HPF. For the results in section 4.12, we used two available compilers to compile UPC codes: GCC UPC 4.5.1.2 compiler [2] and Berkeley UPC 2.12.2 compiler[1] [1]. UPC executables are always run with Berkeley UPC runtime 2.12.2 [1].

We timed each benchmark several times and recorded the minimum execution time, to eliminate network fluctuation due to foreign processes. We also ran the OMPD-RT and MPI programs ran in turn to further reduce these effects. Each graph has an axis labeled "Speedup" – this is the ratio of the execution time of the serial version of the benchmark running on one processor to the execution time of the translated benchmark running on the number of processors indicated on the x-axis.

We show results for seven benchmarks: JACOBI and SPMUL are micro-kernels available online and the other five are from the NAS Parallel Benchmark suites (NPB) [4], which provide serial, OpenMP and MPI versions in a package. We used the NPB2.3-omp-C suite [13] for the EP, CG, and FT benchmarks, and made C versions for BT and SP from the NPB3.3-OMP suite by translating their Fortran version to C version with the only significant non-syntactic change being to convert from a column-major to row-major array layout[2]. These programs include all NAS Parallel Benchmark that have regular, loop-based computations, and the communication pattern is repetitive, which means the producer and consumer relationship is static during their execution; the hoisting process in Section 3.2 plays an important role in reducing runtime overhead. The Class C input is used for all NPB programs.

The translation process from an OpenMP input to an OMPD-RT output code is fully automated. FT has manual modifications to the OpenMP code and their performance results after this modification are discussed in Section 4.6.

4.2 JACOBI Micro Kernel

JACOBI is a micro-kernel that solves Laplace equations using Jacobi iteration. It uses two 1,024 $\times$ 1,024 matrices with 100,000 iterations. It represents a basic point-to-point communication patterns because each process communicates only with its direct left and right neighbors[3]. Thus, JACOBI can be used to measure a system's point-to-point communication performance.

Figure 5a shows that the translated OMPD-RT program performs as fast as the MPI version because our RTS chooses the best communication type, i.e., point-to-point.

4.3 SPMUL Micro Kernel

SPMUL is a micro-kernel that performs matrix-vector multiplications. It uses a 100,000 $\times$ 100,000 sparse matrix as input. Both the OMPD-RT and the MPI programs partition the input matrix in one dimension[4]. Due to an indirection vector used for reading accesses, the communication pattern is all-gather. Thus, SPMUL can be used to measure a system's all-gather communication performance.

Figure 5b shows that the translated OMPD-RT program performs as fast as the MPI version. As the case of JACOBI, the RTS chooses `MPI_Allgatherv()` instead of `MPI_Alltoallw()`, and it makes OMPD-RT performance match with the hand-coded MPI version.

4.4 EP Benchmark

EP is a highly parallel kernel, often used to explore the upper limit of the floating point performance of a parallel system. The MPI version communicates by using only a reduction function, while the OpenMP version uses a critical section to perform the reduction operation on shared array variables. This critical section is detected at compile-time, above the abstraction interface, and is transformed into `ompd_allreduce()` function calls.

Figure 5d shows that the performance of both the OMPD-RT and the MPI versions is identical.

4.5 CG Benchmark

CG implements a conjugate gradient method, with a sparse matrix-vector multiplication taking most of the time. The benchmark has irregular read accesses on arrays via an indirection vector because of its unstructured grid computations.

The translated and the MPI programs have different data partitioning schemes. The MPI version partitions the sparse input matrix using a 2-D block distribution, which requires a smaller number of processes for reduction communication. By contrast, our translation scheme partitions the input matrix using a simple 1-D block distribution; this is because only the outer-most loop of the matrix-vector multiply computation is parallelized in the OpenMP version.

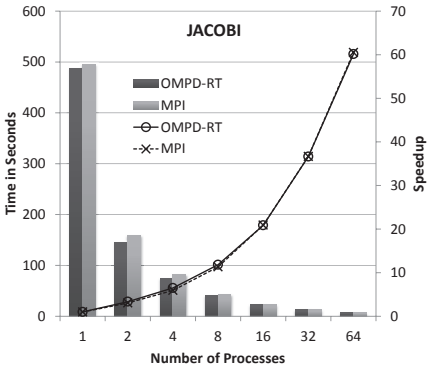
Our analysis of CG shows that performance could be improved further by developing compiler techniques that analyze array expressions in the presence of indirect accesses, hence reducing or eliminating the conservative [-INF:+INF] expression. Doing so is beyond the scope of this paper, however.

[1] Berkeley UPC-to-C translator translates UPC to C which then is compiled using the GCC compiler

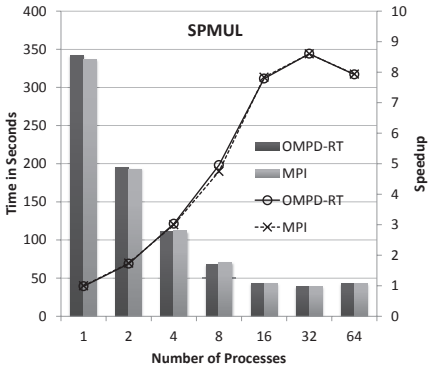
[2] The code of BT and SP in NPB2.3-omp-C suite is not as clean as the NPB3.3 BT and SP. SP in NPB2.3-omp-C was translated to C without considering the conversion of the column to row major mode, so we translated them from NPB3.3-OMP.

[3] The leftmost process communicates only with its right process and the rightmost process with its left process.

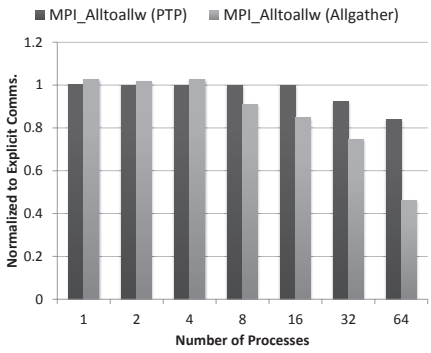
[4] The NPB CG benchmark partitions its input matrix along two dimensions.



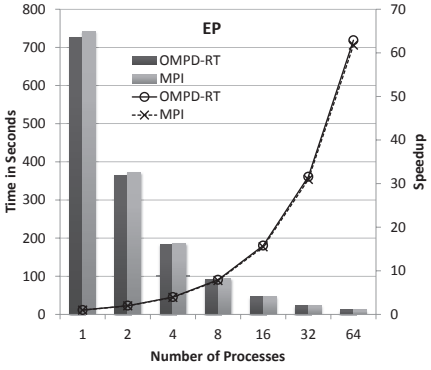
(a) The OMPD-RT JACOBI performance matches its MPI counterpart.



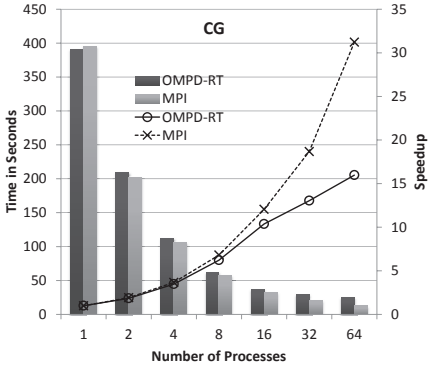
(b) The OMPD-RT SPMUL performance matches its MPI counterpart.



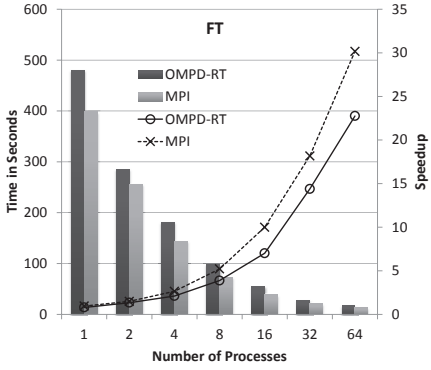
(c) The performance of MPI_Alltoallw() used for point-to-point and allgather communications is compared with its explicit communication functions. MPI_Alltoallw() performs 84% and 46% of explicit point-to-point explicit point-to-point and allgather communication schemes, respectively.



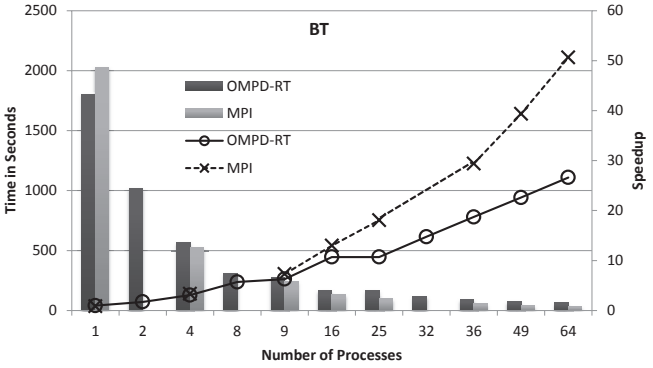
(d) The OMPD-RT EP performance matches its MPI counterpart. The transformation from a critical section to array reductions makes the communication pattern identical as well as their performance.



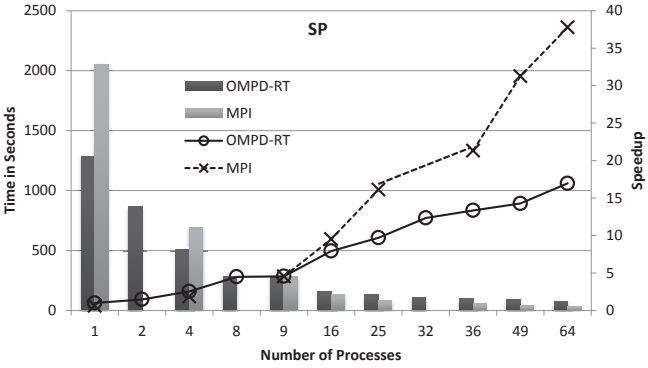
(e) The OMPD-RT CG performance scales according to the number of processes, but is behind the MPI performance. OMPD-RT is approximately 50% slower than MPI on 64 processes.



(f) The OMPD-RT FT performance scales according to the number of processes, achieving 75% of the performance of its MPI counterpart.



(g) The OMPD-RT BT achieves 52% of the performance of its MPI counterpart. Even though the OMPD-RT versions are slower than the MPI versions, their performance is scalable up to 64 processes.



(h) The OMPD-RT SP achieves 45% of the performance of its MPI counterpart. Even though the OMPD-RT versions are slower than the MPI versions, their performance is scalable up to 64 processes.

Figure 5: For the seven benchmarks, OMPD-RT achieves 75% of the hand-coded MPI programs, on average.

4.6 FT Benchmark

FT solves a 3-D partial differential equation using Fast Fourier Transforms. It can be used to test the communication performance of a system because it is very message intensive.

The OMPD-RT version has a manual change to the original OpenMP code that splits a `dcomplex` data structure into real and imaginary `double` parts. This is done because the current translator only supports native C data types, but not structured data types. The performance difference shown on one process in Fig. 5f is from the split, which affects cache performance. However, the effect diminishes as the number of processes increases resulting in smaller problem size per process and less impact from the CPU caches.

The major communication pattern of the MPI version is all-to-all, and the OpenMP version implements the computation by parallelizing loops along the z-axis and y-axis alternatively. This OpenMP computation pattern produces LDEF and LUSE sections partitioned in different dimensions and the RTS identifies their intersection as an all-to-all communication pattern. Thus, the translated version and the MPI version have identical communication patterns.

FT shows the most pronounced performance difference compared to related work [14] (other than BT and SP, which were not included). Figure 6 compares the results. The key techniques in our system are Dynamic Owner Alignment and Runtime GUSE Evaluation. The figure also shows the effect of manually applied advanced affinity analysis, showing the potential for improvements beyond the techniques presented here.

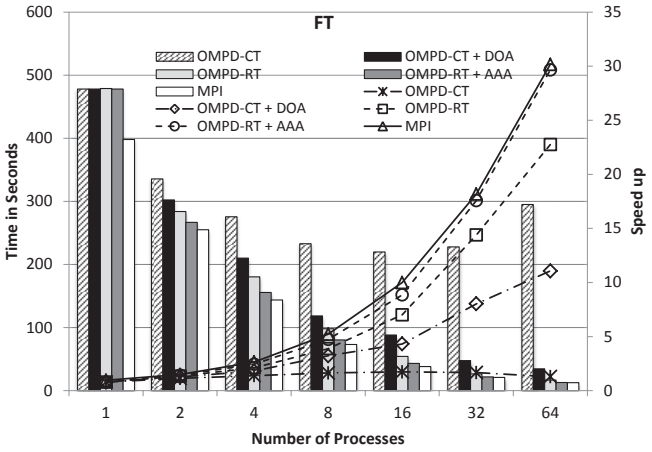


Figure 6: With the advanced affinity analysis (AAA), FT performs as fast as the MPI version on 64 processes.

OpenMP FT has a checksum function that does reduction operations with 1,024 selected elements in a shared array. Since the elements are picked from the entire data space of the shared array using the modulus operator (%), the USE section becomes [-INF:+INF][-INF:+INF][-INF:+INF]. The dynamic owner-computes compiler optimization in Section 3.1 transforms this infinite section into an affine [LB:UB][-INF:+INF][-INF:+INF] section and it completely removes the communication at the CP.

The fourth bar (OMPD-RT + AAA) is obtained by applying advanced affinity analysis (AAA) to the OpenMP source code manually. Sometimes it is beneficial for communication to partition a different dimension than the one specified by the users to increase affinity between LDEF and LUSE. The transformation removes communication from a CP in the `evolve()` function, gaining additional 30% performance. Implementing AAA would require new compiler techniques, such as detecting parallelizable indices from loops and deciding which loop to parallelize within a program's global scope.

In summary, with 64 processes, OMPD-RT's performance is 75% of MPI's, and with the help of AAA it could match the performance of the hand-coded MPI program.

4.7 BT and SP Benchmarks

BT solves a synthetic system of nonlinear partial differential equations using block tri-diagonal solver, while SP solves it using a penta-diagonal solver. Since the BT and SP algorithms have a similar structure and expose the same problems in OMPD-RT, we will discuss them together.

The algorithms of the OpenMP and MPI versions of the benchmarks are different; The OpenMP versions have parallel loops partitioned along the z-axis and y-axis alternatively as the OpenMP FT does, while the MPI versions solves the problem by using multi-partition scheme [20] which requires a square number of processes; a process in the multi-partition scheme communicates only with its neighbors in the 3-D space. Thus, unlike FT, the translated OMPD-RT versions and MPI versions have a different communication scheme. Because of this difference, the OMPD-RT version can use an arbitrary number of processes, including 2, 8, and 32, while the MPI version is limited to the square numbers of processes shown in Fig. 5g and 5h. The OMPD-RT BT and SP reach 52% and 45% of the performance of their MPI counterpart, respectively.

Even though we attribute the major performance difference to the different communication scheme, we identified some possible improvements that can be made to our system. For example, the compiler partitions parallel loops based on the iteration space, but for BT and SP it generates communications at the boundaries of processes' accessed data. Those boundary communications can be avoided when using a partitioning method based on the data space instead of the iteration space. These changes are beyond the scope of this paper, and are future work.

OMPD-RT BT and SP have mixed communication patterns. They have point-to-point, allreduce, all-to-all, and a combination of point-to-point and all-to-all communications. The combination of the point-to-point and all-to-all communication results from the union of USE sections just before the LDEF and GUSE intersection operation; A process sends and receives its full LDEF section with its direct neighbors, but with the other processes it sends and receives data in the all-to-all pattern. The communication scheme using `MPI_Alltoallw()` works well with this mixed pattern and provides reasonable performance.

4.8 MPI_Alltoallw() vs. Explicit communication calls.

Section 3.3.3 explained why the RTS uses `MPI_Alltoallw()` as well as two additional communication types. Here we compare the performance of `MPI_Alltoallw()` with explicit communication functions, such as allgather and point-to-point with a process' direct neighbor processes.

Using the JACOBI and SPMUL benchmarks which are the representative cases for the above patterns, Fig. 5c shows the performance of `MPI_Alltoallw()` normalized to the explicit point-to-point and the allgather performance. The efficiency of `MPI_Alltoallw()` is 84% and 46% of the explicit point-to-point and the allgather communication on 64 processes, respectively. Thus, it motivates the RTS to integrate the communication type detection step in the scheduling communication message step 3.3.3, and the RTS chooses the best scheme dynamically.

4.9 Performance Comparison of Compile-time and Runtime GUSE Evaluations

The runtime GUSE computation aims to improve accuracy of GUSE relative to the compile time method. Figure 7 shows how

much speedup is made for each benchmark on 64 processes. For JACOBI, SPMUL and EP, the runtime scheme does not improve the GUSE accuracy because of the small number of communication points. CG also shows little improvement, even though it has 52 communication points, because it has one dimensional data space and its data partition is simple and static. However, for large applications, such as FT, BT, and SP, the runtime evaluation improves performance substantially, making it an essential technique for realistic and large size applications.

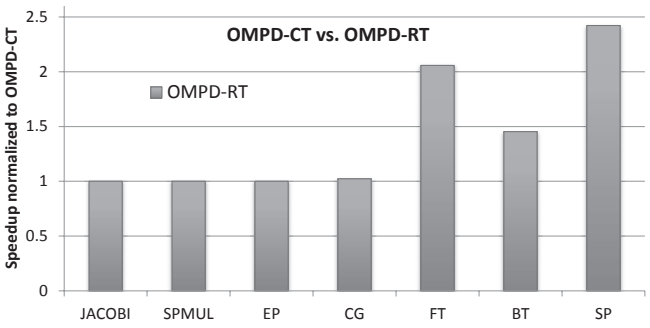


Figure 7: Comparison of speedup from the compile time and the runtime GUSE evaluation.

4.10 Runtime Overhead

The number of communication points affects the runtime overhead of evaluating GUSEs and is typically proportional to the size of the program. Table 1 shows the number of communication points in each benchmark.

Table 1: Numbers of communication points

Benchmark	JA	SPMUL	EP	CG	FT	BT	SP
# of CPs	2	2	2	52	22	71	79

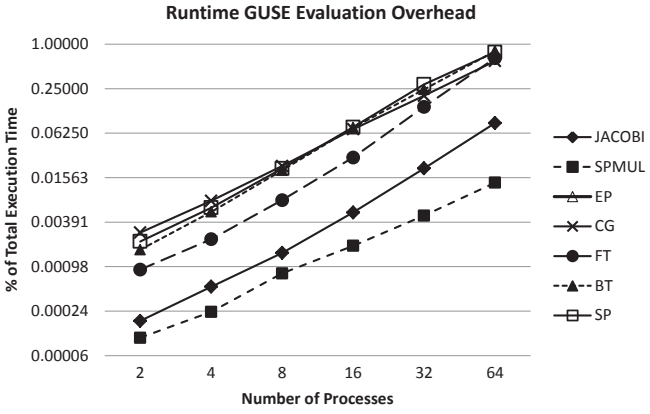


Figure 8: Evaluation of GUSEs incurs the largest overhead of our system. The runtime system shows acceptable overhead up to 64 processes. Additional optimizations will be needed for larger clusters.

Figure 8 shows the runtime GUSE evaluation overhead. All of the seven benchmarks have less than 1% overhead on up to 64 processes. EP does not incur runtime overhead because there are no LDEF and LUSE sections used to evaluate GUSEs. The overhead looks proportional to the number of communication points. The sum of the other runtime overheads, such as updating sections and creating RCFG and intersections, is not shown because it is much less than the GUSE evaluation.

Improving the scalability of our runtime techniques is part of our ongoing work. One opportunity is to parallelize the runtime algorithm. Each process has full information about the sections of the other processes, making it possible to perform localized evaluations with global exchange of the results.

4.11 Performance comparison with HPF

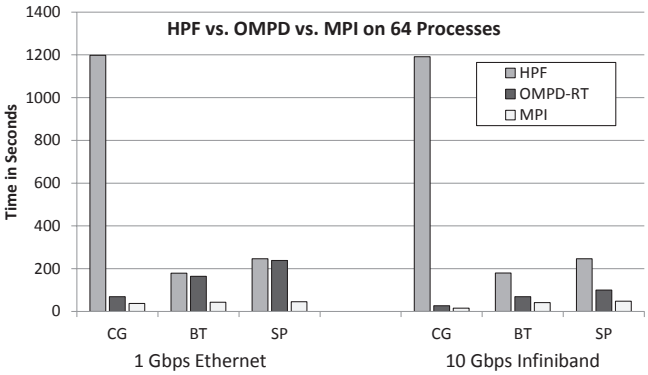


Figure 9: The translated OMPD-RT programs perform better than the hand-coded HPF programs on 64 processes.

Figure 9 shows performance comparisons among available HPF, OMPD-RT, and MPI programs on 64 processes. NPB3.0-HPF is used and FT is not shown because there are compilation errors from the FT HPF code. The HPF results show that the HPF programs used here do not take advantage of InfiniBand (IB), even though they are linked with the IB library, so we show 1Gbps Ethernet (1GE) results as well. The HPF CG suffers from the huge communication overhead of redistributing the p array, so it shows poor performance [9]. The OMPD-RT BT and SP with 1GE perform less than those with IB because the programs use a global data exchange using all-to-all communication, while the MPI BT and SP are optimized to communicate among neighbor processes in the 3-D space as discussed in Section 4.7. The translated OMPD-RT programs are still faster than the hand-coded HPF programs.

4.12 Performance comparison with UPC

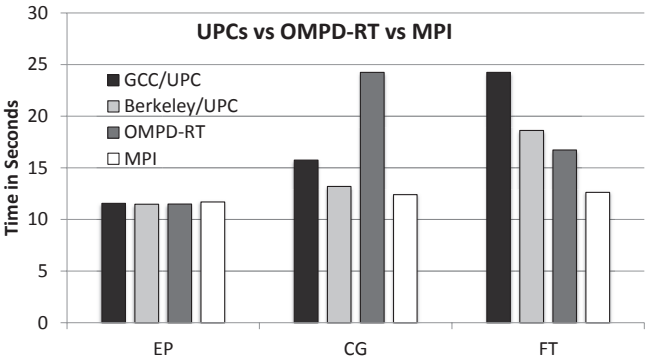


Figure 10: Execution times of UPC, OMPD-RT and MPI on 64 processes. UPC programs are compiled with two available compilers.

Unified Parallel C (UPC) has been proposed to ease programming effort by providing a global shared address space. It gives programmers more control than OpenMP; programmers can specify

the affinity between thread and data with additional efforts. There are publicly available NPB programs written in UPC.

We obtained UPC versions of the EP, CG and FT benchmarks from GWU NPB suite version 2.4 [3]. BT and SP benchmarks are not available. There are multiple versions of CG and FT with progressively advanced hand optimizations. Figure 10 shows the performance of the fastest ones, together with the OMPD-RT and MPI benchmark versions on 64 processes. In EP, all benchmarks achieved almost the same performance. In CG, both UPC versions outperform OMPD-RT, Berkeley UPC by 1.83 and GCC UPC by 1.54. In FT, OMPD-RT outperforms both UPC versions, Berkeley UPC by 1.11 and GCC UPC by 1.49. For more detailed study about Berkeley UPC performance on clusters, see [18].

5. Conclusions

We have presented a hybrid, compiler-runtime system for translating and executing OpenMP programs on clusters. The system features a novel runtime data flow analysis and communication generation scheme as well as a compiler technique for improving data affinity.

Our system shows that it is feasible to use OpenMP as a programming model for clusters of up to one hundred processes, which are important platforms for research laboratories and small enterprises. This holds for an important class of applications, which are regular and exhibit repetitive communication behavior. Our system derives the distribution of data automatically from the source program, unlike approaches such as HPF and UPC, which involve the user in this task. Our results show that it is possible to hide this task from the user.

The presented hybrid approach puts significant responsibility on the runtime system. We have shown that, for the considered class of applications, this approach is feasible, the system is able to overcome limitations of static compiler techniques and the runtime overhead can be kept small.

Our system achieves, on average, 75% of the performance of hand-tuned MPI applications. Hand-tuned applications will always achieve better performance; however, we have also identified several optimizations that may bring the performance even closer to MPI and improve the scalability beyond the platform size that this paper focused on. Developing these optimizations is a topic of our ongoing work.

Acknowledgments

This work was supported, in part, by the National Science Foundation under grants No. 0720471-CNS, 0707931-CNS, 0833115-CCF, and 0916817-CCF.

References

[1] Berkeley UPC - Unified Parallel C. Available at: `upc.lbl.gov`.

[2] GCC Unified Parallel C. Available at: `www.gccupc.org`.

[3] UPC NAS Parallel Benchmarks from The George Washington University High Performance Computing Laboratory. Available at: `threads.hpcl.gwu.edu/sites/npb-upc`.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. 1991.

[5] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 219–228, New York, NY, USA, 2009. ACM.

[6] D. Baxter, R. Mirchandaney, and J. H. Saltz. Run-time parallelization and scheduling of loops. In *Proceedings of the first annual ACM symposium on Parallel Algorithms and Architectures*, SPAA '89, pages 303–312, New York, NY, USA, 1989. ACM.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications. (OOPSLA '05)*, pages 519–538, New York, NY, USA, 2005. ACM.

[8] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proc. of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 186–197, 1996.

[9] M. Frumkin, H. Jin, and J. Yan. Implementation of NAS Parallel Benchmarks in High Performance Fortran. In *Symposium on Parallel and Distributed Processing*, 2000.

[10] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 71, New York, NY, USA, 1995. ACM.

[11] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex., 1993.

[12] J. P. Hoeflinger. Extending OpenMP to Clusters. White Paper, 2006.

[13] K. Kusano, M. Sato, T. Hosomi, and Y. Seo. The Omni OpenMP Compiler on the Distributed Shared Memory of Cenju-4. In *OpenMP Shared Memory Parallel Programming*, volume 2104 of *Lecture Notes in Computer Science*, pages 20–30. Springer Berlin / Heidelberg, 2001.

[14] O. Kwon, F. Jubair, S.-J. Min, H. Bae, R. Eigenmann, and S. Midkiff. Automatic Scaling of OpenMP Beyond Shared Memory. In *LCPC 2011: Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing*, Sept. 2011.

[15] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[16] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM Trans. Program. Lang. Syst.*, 24:65–109, January 2002.

[17] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 274–284, New York, NY, USA, 2002. ACM.

[18] H. Shan, F. Blagojević, S.-J. Min, P. Hargrove, H. Jin, K. Fuerlinger, A. Koniges, and N. J. Wright. A programming model performance study using the NAS parallel benchmarks. *Scientific Programming*, 18:153–167, August 2010.

[19] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Laboratory, 2005.

[20] R. F. V. D. Wijngaart. Efficient Implementation of a 3-Dimensional ADI Method on the iPSC/860. In *In Supercomputing '93*, pages 102–111, 1993.

[21] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency - Practice and Experience*, 10(11-13):825–836, 1998.