An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C*

Cristian Coarfă Yuri Dotsenko John Mellor-Crummey
{ccristi,dotsenko,johnmc}@cs.rice.edu
Rice University

François Cantonnet Tarek El-Ghazawi Ashrujit Mohanty Yiyi Yao
{fcantonn,tarek,ashrujit,yyy}@gwu.edu
George Washington University

Daniel Chavarría-Miranda
daniel.chavarria@pnl.gov
Pacific Northwest National Laboratory (PNNL)

ABSTRACT

Co-array Fortran (CAF) and Unified Parallel C (UPC) are two emerging languages for single-program, multiple-data global address space programming. These languages boost programmer productivity by providing shared variables for communication instead of message passing. However, the performance of these emerging languages still has room for improvement. In this paper, we study the performance of variants of the NAS MG, CG, SP, and BT benchmarks on several modern architectures to identify challenges that must be met to deliver top performance. We compare CAF and UPC variants of these programs with the original Fortran+MPI code. Today, CAF and UPC programs deliver scalable performance on clusters only when written to use bulk communication. However, our experiments uncovered some significant performance bottlenecks of UPC codes on all platforms. We account for the root causes limiting UPC performance such as synchronization model, communication efficiency of strided data, and source-to-source translation issues. We show that they can be remedied with language extensions, new synchronization constructs, and, finally, adequate optimizations by the back-end C compilers.

*This work was supported in part by the Department of Energy under Grant DE-FC03-01ER25504/A000, the Los Alamos Computer Science Institute (LACSI) through LANL contract number 03891-99-23 as part of the prime contract (W-7405-ENG-36) between the DOE and the Regents of the University of California, Texas Advanced Technology Program under Grant 003604-0059-2001, and Compaq Computer Corporation under a cooperative research agreement. The computations were performed in part on an Itanium cluster purchased with support from the NSF under Grant EIA-0216467, Intel and Hewlett Packard, and on the National Science Foundation Terascale Computing System at the Pittsburgh Supercomputing Center. This research was performed in part using the Molecular Science Computing Facility (MSCF) in the William R. Wiley Environmental Molecular Sciences Laboratory, a national scientific user facility sponsored by the U.S. Department of Energy's Office of Biological and Environmental Research and located at the Pacific Northwest National Laboratory. Pacific Northwest is operated for the Department of Energy by Battelle.

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

1. INTRODUCTION

Writing shared-memory parallel programs is widely viewed as simpler than writing message-passing programs. However, today message passing is the undisputed leader for achieving performance, scalability and portability on parallel systems in general, and commodity clusters in particular. The principal obstacle for scalability and performance of shared-memory parallel programming models has been their inability to exploit locality effectively.

For this reason, there has been considerable interest in developing locality-aware paradigms for shared-memory parallel programming. The Partitioned Global Address Space (PGAS) model for Single Program Multiple Data (SPMD) parallel programming was developed to address this issue. In the PGAS model, multiple SPMD threads (or processes) share a part of their address space. However, the shared space is partitioned and a portion of it is local to each thread or process. Programs using the PGAS model can exploit locality by having each thread or process principally compute on data that is local to it.

Unified Parallel C [11] and Co-Array Fortran [16] are two programming languages based on the PGAS model. UPC and CAF both aim to support locality-aware shared mem-

ory parallel programming, but they differ in important design choices. For example, UPC provides the abstraction of a "flat" address space in which any element in a distributed data structure can be accessed uniformly (with some overhead). In contrast, processor boundaries of distributed arrays are explicit in CAF and a special syntactic construct is used to access remote memory.

Compilers for CAF and UPC generally use a source-to-source compilation strategy, in which they translate programs into Fortran 90 and C programs augmented with communication code. This approach leverages the capabilities of sequential language compilers to efficiently map the transformed code into highly-optimized machine code. For PGAS languages to be efficient, high-level parallel language constructs, such as pointers to remote memories and remote array sections, must be carefully compiled into constructs in the target sequential language with an understanding of the translation's implications for performance.

This paper presents a comparative study of the relative performance of UPC and CAF. To assess the ability of these languages to deliver performance, we study a subset of the NAS Parallel Benchmarks [1] implemented in each language. For reference, we also compare the performance of UPC and CAF variants of these benchmarks to their standard implementations written in Fortran+MPI. The goal of this study is to determine which features are necessary to enable PGAS parallel programming languages to deliver high performance on a range of target platforms.

Section 2 briefly describes the CAF and UPC languages along with the compilers that we used in our study. Section 3 presents our experimental evaluation of the CAF and UPC programming models and compilers using the NAS parallel benchmarks. Section 4 summarizes the conclusions we draw from our study.

2. BACKGROUND

2.1 Co-array Fortran

Co-Array Fortran extends Fortran 95 with a few language constructs to supports SPMD parallel programming. An executing CAF program consists of a static collection of asynchronous process images. As in MPI programs, CAF programs explicitly manage data locality and partitioned computation.

CAF provides a partitioned global address space. Distributed data is declared using a natural extension to Fortran 90 syntax. For example, the declaration `integer :: a(n,m)[*]` declares a shared co-array `a` with $\mathtt{n} \times \mathtt{m}$ integers local to each process image. Instead of explicitly coding message exchanges to access data belonging to other processes, a CAF program can directly reference non-local values using an extension to Fortran 90 syntax for subscripted references. For instance, process `p` can read the first column of co-array `a` from process `p+1` with the right-hand side reference to `a(:,1)[p+1]`. CAF has a primitive for a synchronous barrier among all or a subset of process images. A more complete description of the CAF language can be found elsewhere [16].

In previous studies [6, 7], we identified a few weaknesses of the original CAF language specification that can reduce the performance of CAF codes. In particular, CAF's original team-based synchronization requires collective operations that are often not necessary; stronger synchronization typically reduces performance. Also, the original CAF specification prescribes the use of memory fences before and after each procedure call; this can inhibit overlapping communication with computation. In response, we proposed refinements to CAF that enable these sources of performance degradation to be avoided. In [6], we proposed extending the CAF model with unidirectional point-to-point synchronization primitives: `sync_notify` and `sync_wait`. These primitives offer a high-performance alternative to collective team-based synchronization. In [7] we proposed a small set of directives to help exploit non-blocking communication in CAF programs; these directives help programmers overlap communication with computation.

2.2 The Rice CAF compiler

The Rice CAF compiler, `cafc` [20], was designed with the major goals of being portable and delivering high-performance on a multitude of platforms. Ideally, a programmer will write a CAF program once in a natural style and `cafc` will adapt it for high performance on the target platform of choice.

To achieve this goal, `cafc` performs source-to-source transformation of CAF code into Fortran 90 code augmented with communication operations. For communication, `cafc` typically generates calls to ARMCI's [15] one-sided communication primitives; however, for shared memory systems `cafc` can generate code that uses load and store operations for communication. `cafc` is based on Open64/SL [18], a version of the Open64 [17] compiler infrastructure that was modified to support source-to-source transformation of Fortran 90 and CAF.

2.3 Unified Parallel C

UPC is an explicitly parallel extension of ISO C that supports a global address space programming model for writing SPMD parallel programs. In the UPC model, SPMD threads share a part of their address space. The shared space is logically partitioned into fragments, each with a special association (affinity) to a given thread. UPC declarations give programmers control over the distribution of data across the threads; they enable a programmer to associate data with the thread primarily manipulating it. A thread and its associated data are typically mapped by the system into the same physical node. Being able to associate shared data with a thread makes it possible to exploit locality. In addition to shared data, UPC threads can have private data as well; private data is always co-located with its thread.

UPC's support for parallel programming consists of a few key constructs. UPC provides the `upc_forall` work-sharing construct. At run time, `upc_forall` is responsible for assigning independent loop iterations to threads so that iterations and the data they manipulate are assigned to the same thread. UPC adds several keywords to C that enable it to express a rich set of private and shared pointer concepts. UPC supports dynamic shared memory allocation. The language offers a range of synchronization and memory consistency control constructs. Among the most interesting synchronization concepts in UPC is the non-blocking barrier, which allows overlapping local computation and inter-thread synchronization. Parallel I/O [9] and collective operation library specifications [21] have been recently designed

and will be soon integrated into the formal UPC language specifications. Also, [3] proposes a set of UPC extensions that enables efficient strided data transfers and overlap of computation and communication.

2.4 Unified Parallel C Compilers

The Berkeley UPC (BUPC) compiler [5] performs source-to-source translation. It first converts UPC programs into platform-independent ANSI-C compliant code, tailors the generated code to the the target architecture (cluster or shared memory), and augments it with calls to the Berkeley UPC Runtime system, which in turn, invokes a lower level one-sided communication library called GASNet [2]. The GASNet library is optimized for a variety of target architectures and delivers high performance communication by applying communication optimizations such as message coalescing and aggregation as well as optimizing accesses to local shared data. We used both the 2.0.1 and 2.1.0 versions of the Berkeley UPC compiler in our study.

The Intrepid UPC compiler [12] is based on the GCC compiler infrastructure and supports compilation to shared memory systems including the SGI Origin, Cray T3E and Linux SMPs. The GCC-UPC compiler used in our study is version 3.3.2.9, with the 64-bit extensions enabled. This version incorporates inlining optimizations and utilizes the GASNet communication library for distributed memory systems.

3. EXPERIMENTAL EVALUATION

To assess the ability of PGAS language implementations to deliver performance, we compare the performance of CAF, UPC and Fortran+MPI implementations of the NAS Parallel Benchmarks (NPB) MG, CG, SP and BT. The NPB codes are widely used for evaluating the performance of parallel compilers and parallel systems. For our study, we used MPI codes from the NPB 2.3 release. Sequential performance measurements used as a baseline were performed using the Fortran-based NPB 2.3-serial release. The CAF and UPC benchmarks were derived from the corresponding NPB-2.3 MPI implementations; they use essentially the same algorithms as the corresponding MPI versions. To achieve higher perfomance, we experimented with UPC extensions for strided and asynchronous communication [3] and with CAF extensions for point-to-point synchronization and non-blocking communication [7]. We evaluated the benefits of point-to-point synchronization in UPC programs for large number of processors; to the best of our knowledge this is the first study of point-to-point synchronization for UPC codes.

3.1 NAS Parallel Benchmarks

NAS MG. The MG multigrid kernel calculates an approximate solution to the discrete Poisson problem using four iterations of the V-cycle multigrid algorithm on a $\mathtt{n} \times \mathtt{n} \times \mathtt{n}$ grid with periodic boundary conditions [1]. MG's communication is highly structured and repeats a fixed sequence of regular patterns.

NAS CG. The CG benchmark uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix [1]. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication and employs sparse matrix vector multiplication. The irregular communication employed by this benchmark is a challenge for clusters.

NAS SP and BT. The NAS BT and SP benchmarks are two simulated CFD applications that solve systems of equations resulting from an approximately factored implicit finite difference discretization of three-dimensional Navier-Stokes equations [1]. The principal difference between the codes is that BT solves block-tridiagonal systems of 5x5 blocks, whereas SP solves scalar penta-diagonal systems resulting from full diagonalization of the approximately factored scheme [1]. SP and BT use a skewed-cyclic block distribution known as multipartitioning [1, 14].

3.2 Experimental platforms

Our experiments studied the performance of the MG, CG, BT and SP benchmarks on four architectures.

The first platform is a cluster of 92 HP zx6000 workstations interconnected with Myrinet 2000. Each workstation node contains two 900MHz Intel Itanium 2 processors with 32KB/256KB/1.5MB of L1/L2/L3 cache, 4-8GB of RAM, and the HP zx1 chipset. Each node is running the Linux operating system (kernel version 2.4.18-e plus patches). We used the Intel compilers V8.0 as our back-end compiler and the Berkeley UPC compiler V2.1.0[1] with the `gm` conduit.

The second platform was the Lemieux Alpha cluster at the Pittsburgh Supercomputing Center. Each node is an SMP with four 1GHz processors and 4GB of memory. The operating system is OSF1 Tru64 v5.1A. The cluster nodes are connected with a Quadrics interconnect (Elan3). We used the Compaq Fortran 90 compiler V5.5 and Compaq C/C++ compiler V6.5 as well as the Berkeley UPC compiler V2.0.1[2] using the `elan` conduit.

The other two platforms are non-uniform memory access (NUMA) architectures: an SGI Altix 3000 and an SGI Origin 2000. The Altix 3000 has 128 Itanium2 1.5GHz processors with 6MB L3 cache, and 128GB RAM, running the Linux64 OS with the 2.4.21 kernel, Intel compilers V8.0, and the Berkeley UPC compiler V2.1.0[3] using the `shmem` conduit. The Origin 2000 has 32 MIPS R10000 processors with 4MB L2 cache and 16 GB RAM, running IRIX64 V6.5, the MIPSpro Compilers V7.4 and the Berkeley UPC compiler V2.0.1[4] using the `smp` conduit.

3.3 Experimental Methodology

For each application and platform, we selected the largest problem size for which all the MPI, CAF, and UPC versions ran and verified within the architecture constraints (mainly memory). We do not report performance results for NAS MG on the Alpha+Quadrics platform because we were not able to collect reliable data for UPC.

For each benchmark, we compare the parallel efficiencies of the CAF, UPC and MPI versions. We compute parallel efficiency as follows. For each parallel version ρ, the efficiency metric is computed as $\frac{\mathtt{t_s}}{\mathtt{P} \times \mathtt{t_p}(\mathtt{P},\rho)}$. In this equation, $\mathtt{t_s}$ is the execution time of the original Fortran sequential version implemented by the NAS group at the NASA Ames Research Laboratory; $\mathtt{P}$ is the number of processors; $\mathtt{t_p}(\mathtt{P},\rho)$ is the time for the parallel execution on $\mathtt{P}$ processors using

[1]back-end compiler options: -override_limits -O3 -g -tpp2
[2]back-end compiler options: -fast -O5 -tune host -intrinsics
[3]back-end compiler options: -override_limits -O3 -g -tpp2
[4]back-end compiler options: -64 -mips4 -DMPI -O3

parallelization ρ. Using this metric, perfect speedup would yield efficiency 1.0. We use efficiency rather than speedup or execution time as our comparison metric because it enables us to accurately gauge the relative performance of multiple benchmark implementations across the *entire* range of processor counts[5].

3.4 NAS MG

Figures 1 (a) and (b) present the performance of classes A (problem size 256^3) and C (problem size 512^3) on an Itanium2 cluster with a Myrinet 2000 interconnect. The *MPI* curve is the baseline for comparison as it represents the performance of the NPB-2.3 official benchmark. The *CAF* curve represents the efficiency of the fastest code variant written in Co-Array Fortran and compiled with `cafc`. To achieve high performance, the *CAF* code uses communication vectorization, synchronization strength reduction, procedure splitting and non-blocking communication, as described elsewhere [6, 7]. The *CAF-barrier* version is similar to *CAF*, but uses barriers for synchronization.

In Figure 1, the *BUPC*, *BUPC-restrict*, *BUPC-strided*, and *BUPC-p2p* curves display the efficiency of NAS MG coded in UPC and compiled with the BUPC compiler. The UPC implementation uses a program structure similar to that of the *MPI* version. All UPC versions declare local pointers for each level of the grid for more efficient access to local portions of shared arrays. The *BUPC-restrict*, *BUPC-p2p* and *BUPC-strided* differ from *BUPC* by declaring these local pointers as restricted, using the C99 `restrict` keyword to improve alias analysis in the back-end C compiler. *BUPC* and *BUPC-restrict* use barriers for interprocessor synchronization; *BUPC-p2p* and *BUPC-strided* use point-to-point synchronization implemented at the UPC language level. *BUPC*, *BUPC-restrict* and *BUPC-p2p* use `upc_memput` for bulk data transfers; *BUPC-strided* uses UPC extensions to perform bulk transfers of strided data.

The results show that *CAF* has an efficiency comparable to that of *MPI*; the *CAF-barrier* performance is similar to that of *MPI* for small numbers of CPUs, but the performance degrades for larger numbers of processors. The original *BUPC* version is as much as seven times slower than *MPI* and *CAF*. We identified three major causes for this performance difference. The principal cause is lower scalar performance due to source-to-source translation issues, such as failing to convey aliasing information to the back-end compiler and inefficient code generated for linearized indexing of multidimensional data in UPC. Second, using barrier synchronization when point-to-point synchronization suffices degrades performance and scalability. Third, communicating non-contiguous data in UPC is currently expensive.

Source-to-source translation challenges. The following code fragment is for the residual calculation, `resid`, which is computationally intensive. *MPI*, *CAF*, and *CAF-barrier* use multidimensional arrays to access private data.

```
subroutine resid(u,v,r,n1,n2,n3,...)
  integer n1,n2,n3
  double precision u(n1,n2,n3),v(n1,n2,n3),r(n1,n2,n3),a(0:3)
! loop nest accounting for 33% of total walltime
  r(i1,i2,i3) = v(i1,i2,i3) - a(0) * u(i1,i2,i3) - ...
  ...
end subroutine resid
```

The corresponding routine in the UPC versions uses C pointers to access the local parts of shared arrays as shown below.

```
typedef struct sh_arr_s sh_arr_t;
struct sh_arr_s {
  shared [] double *arr;
};

void resid( shared sh_arr_t *u, shared sh_arr_t *v,
  shared sh_arr_t *r, int n1, int n2, int n3, ...) {
#define u(iz,iy,ix) u_ptr[(iz)*n2*n1 + (iy)*n1 + ix]
#define v(iz,iy,ix) v_ptr[(iz)*n2*n1 + (iy)*n1 + ix]
#define r(iz,iy,ix) r_ptr[(iz)*n2*n1 + (iy)*n1 + ix]
  double *restrict u_ptr, *restrict v_ptr, *restrict r_ptr;
  u_ptr = & u[MYTHREAD].arr[0];
  v_ptr = & v[MYTHREAD].arr[0];
  r_ptr = & r[MYTHREAD].arr[0];
  // loop nest accounting for 60% of total walltime
  r(i3, i2, i1) = v(i3, i2, i1) - a[0] * u(i3, i2, i1) - ...
  ...
}
```

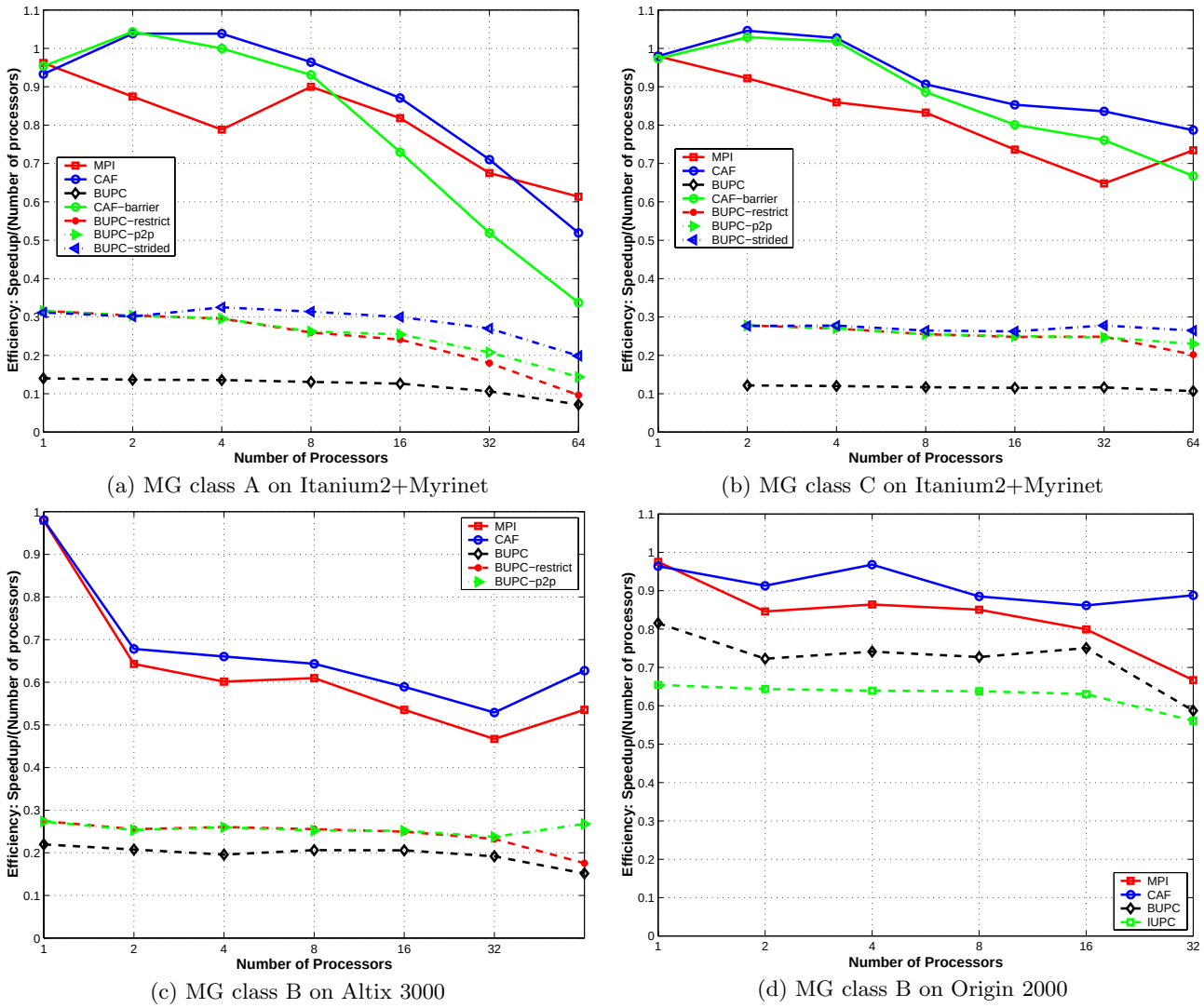
If `u` were used to access shared local data via UPC's runtime address resolution for shared pointers [4, 5], the performance would suffer from executing a branch per data access. The use of `u_ptr`, a regular C pointer, enables the local portion of the shared array `u` to be accessed without the need for runtime address resolution.

In Fortran, `u` is a subroutine argument and cannot alias other variables, while in C, `u_ptr` is a pointer. Hence, lacking sophisticated alias analysis, a C compiler conservatively assumes that `u_ptr` can alias other variables. In turn, this prevents the C compiler from doing some high-level loop nest optimizations. Using Rice's HPCToolkit [19, 13] (a suite of tools for profile-based performance analysis using statistical sampling of hardware performance counters) we analyzed one-processor versions of MG class B. We discovered that the *BUPC* version of `resid` had 2.08 times more retired instructions and executed ten times slower than its Fortran counterparts. For the entire benchmark, the performance of the *BUPC* version was seven times lower (144 vs. 21 seconds).

To inform the back-end C compiler that `u_ptr` does not alias other variables, we annotated the declaration of `u_ptr` with the C99 `restrict` keyword. Restricting all relevant pointers in `resid` resulted in a 20% reduction in the number of retired instructions and yielded a factor of two speedup for this routine. Using `restrict` for *BUPC-restrict* where it was safe to do so resulted in 2.3 times performance improvement reducing the execution time to 63 seconds, only three times slower than MPI instead of the original factor of seven.

With CAF, we had previously encountered a similar difficulty with overly conservative assumptions about aliasing in back-end Fortran compilers when computing on the local parts of COMMON/SAVE co-arrays. In CAF, global co-arrays do not alias, but their pointer-based representation does not convey this information to back-end Fortran compilers. To address this problem, we previously developed a source-to-source transformation known as procedure splitting [7]. This transformation eliminates overly conservative assumptions about aliasing by transforming a pointer-

[5]There are also sequential C implementations of the NAS MG, CG, SP, and BT benchmarks that employ the same algorithms as the original Fortran versions. The performance of the C versions of SP and CG is similar to that of the original Fortran versions. The C version of BT is up to two times slower than its Fortran variant. The C version of MG allocates each row in the multigrid data structure as a separate object in memory; this degrades performance by as much as a factor of seven compared to its Fortran counterpart.



(a) MG class A on Itanium2+Myrinet

(b) MG class C on Itanium2+Myrinet

(c) MG class B on Altix 3000

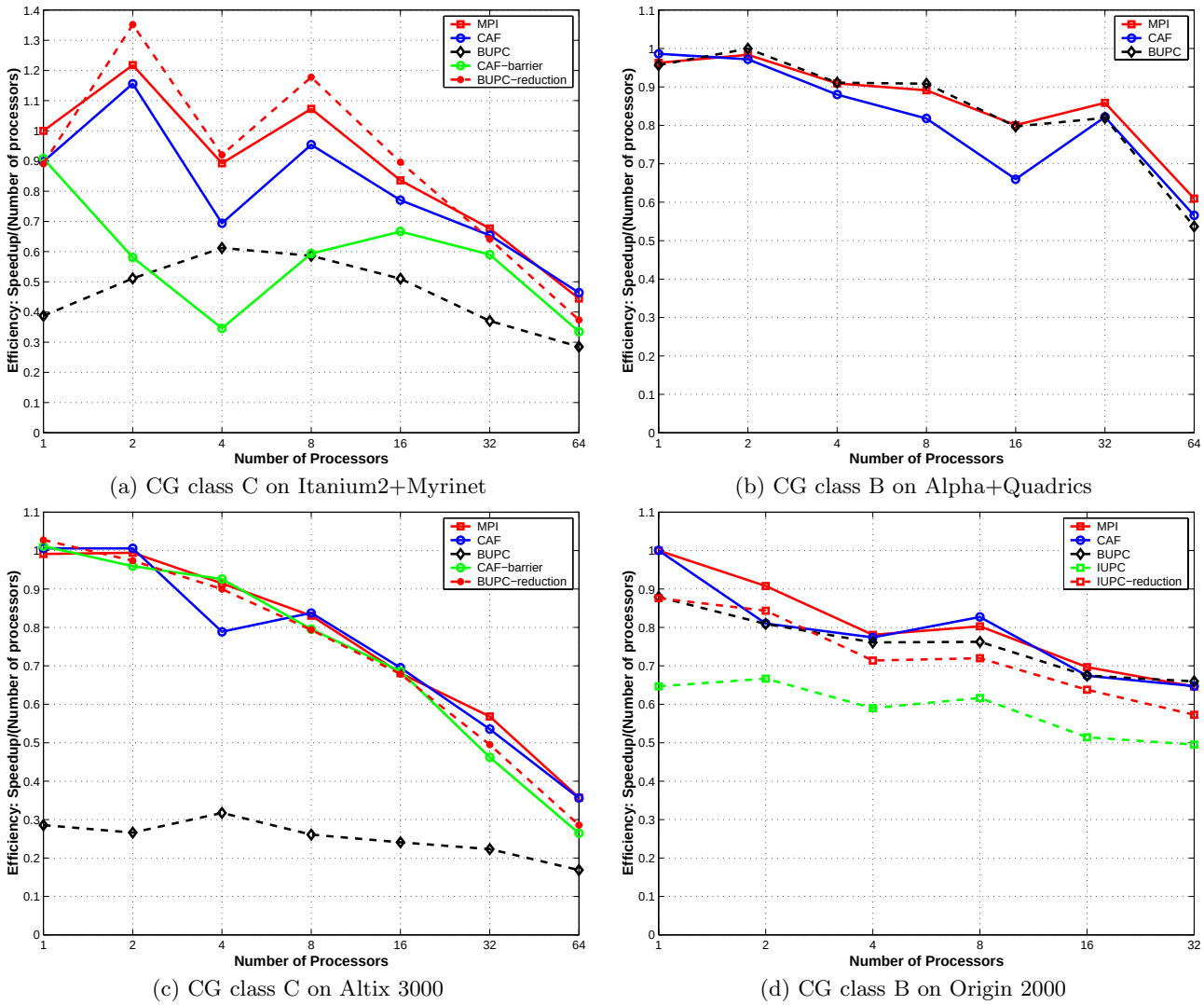
(d) MG class B on Origin 2000

Figure 1: Comparison of MPI, CAF and UPC parallel efficiency for NAS MG.

based representation for co-array data into one based on dummy arguments, which are correctly understood to be free of aliases.

While alias analysis of UPC programs can be improved by having programmers or (in some cases) UPC compilers add a `restrict` keyword, there is another fundamental issue preventing efficient optimization of scientific C codes. The Fortran code snippet above uses multidimensional arrays with symbolic bounds, expressed as specification expressions by parameters passed to the `resid` subroutine. In UPC MG, the macro `u` creates the syntactic illusion of a multidimensional array, but in fact, this macro linearizes the subscript computation. C does not have the ability to index `u` using a vector of subscripts. Thus, to safely reorder such references during optimization, C compilers must perform dependence analysis of linearized subscripts, which is harder than analyzing a vector of subscripts. This tends to degrade the precision of dependence analysis, which limits the ability of C compilers to exploit some high-level optimizations, and thus yields slower code. To estimate the performance degradation due to linearized subscripts in C, we linearized subscripts in a Fortran version of `resid`. This change doubled the execution time of the Fortran version of `resid` and degraded the overall performance of MG class B on one processor by 30% on the Itanium2+Myrinet cluster.

Point-to-point synchronization. In MG, each SPMD thread needs to synchronize only with a small number of neighbors. While a collective barrier can be used to provide sufficient synchronization, it provides more synchronization than necessary. Our experiments show that unnecessary collective synchronization degrades performance on loosely-coupled architectures. This effect can be seen in Figures 1 (a) and (b) by comparing the efficiency of the *BUPC-restrict* and *BUPC-p2p* versions. We derived *BUPC-p2p* from *BUPC-restrict* by using a reference language-level implementation of point-to-point synchronization. The performance boost is evident for the larger number of processors and amounts to 49% for class A and 14% for class C in 64 processor executions. The class A executions benefit more from using point-to-point synchronization because they are



(a) CG class C on Itanium2+Myrinet

(b) CG class B on Alpha+Quadrics

(c) CG class C on Altix 3000

(d) CG class B on Origin 2000

Figure 2: Comparison of MPI, CAF and UPC parallel efficiency for NAS CG.

more communication bound. A similar effect can be seen for CAF: for 64 processors, *CAF-barrier* shows a 54% slowdown for class A and 18% slowdown for class C.

Non-contiguous data transfers. For certain programs, efficient communication of non-contiguous data can be essential for high efficiency. For MG, the y-direction transfers of *BUPC-restrict* are performed using several communication events, each transferring a contiguous chunk of memory equal to one row of a 3D volume. The *BUPC-strided* version is derived from *BUPC-p2p*. It moves data in the y-direction by invoking a library primitive to perform a strided data transfer; this primitive is a member of a set of proposed UPC language extensions for strided data transfers [3]. Even using Berkeley's reference implementation of the strided communication operation (as opposed to a carefully-optimized implementation) yielded a 28% performance improvement of *BUPC-strided* over *BUPC-p2p* for class A on 64 processors and a 13% efficiency improvements for class C on 64 processors. The most efficient communication can be achieved by packing data into a contiguous communication buffer and sending it as one contiguous chunk. A version that uses packing is marginally more efficient than *BUPC-strided*, thus, we do not show it on the plot.

While in most cases using the UPC strided communication extensions is more convenient than packing and unpacking data on the source and destination, we found it more difficult to use such library primitives than simply reading or writing multi-dimensional co-array sections in CAF using Fortran 90 triplet notation, which `cafc` automatically transforms into equivalent strided communication. For CAF programs, a compiler can automatically infer the parameters of a strided transfer, such as memory strides, chunk sizes, and stride counts; whereas in UPC, these paramenters must be explicitly managed by the user.

Figure 1(c) presents performance results of NAS MG class B (problem size 256^3) for the Altix 3000 architecture. The *MPI*, *CAF*, *BUPC*, *BUPC-restrict*, and *BUPC-p2p* curves are similar to the ones presented for the Itanium2+Myrinet2000 cluster. We used the same versions of the Intel Fortran and C compilers. Therefore, we expected similar trends for the scalar performance of *MPI*, *CAF* and *BUPC*. Indeed, *MPI* and *CAF* versions show comparable performance,

while *BUPC* is up to 4.5 times slower and *BUPC-restrict* is 3.6 times slower than *CAF*. The efficiency of all programs is lower on this architecture compared to that on the Itanium2+Myrinet2000 cluster because in our experiments on the Altix architecture we ran two processes per dual node, sharing the same memory bus.

For CAF, using barrier-based instead of point-to-point synchronization does not cause a significant loss of performance on this architecture for 32 or fewer processors. However, for 64 processors, we observed a performance degradation of 29% when CAF MG used barriers for synchronization. For UPC, *BUPC-p2p* outperforms *BUPC-restrict* by 52% for NAS MG on 64 processors.

Figure 1(d) presents the performance results on the Origin 2000 machine for NAS MG class B (problem size 256^3). The *MPI* curve corresponds to the original MPI version implemented in Fortran. The *CAF* curve gives the performance of the optimized CAF version with the same optimizations as described previously except that non-blocking communication is not used because the architecture supports only synchronous interprocessor memory transfers. The *BUPC* and *IUPC* curves describe the performance of the UPC version of MG compiled with the Berkeley UPC and the Intrepid UPC compilers respectively.

The *CAF* version slightly outperforms the *MPI* version due to more efficient one-sided communication [8]. The *MPI* version slightly outperforms *BUPC* which, in turn, slightly outperforms *IUPC*. The MIPSPro C compiler, which is used as a back-end compiler for *BUPC*, performs more aggressive optimizations compared to the Intel C compiler. In fact, using the `restrict` keyword does not yield additional improvement bacause the alias analysis done by the MIPSPro C compiler is more precise. Nonetheless, it is our belief that the lack of multidimensional arrays in the C language prevents the MIPSPro C compiler from applying high-level loop transformations such as unroll & jam and software pipelining resulting in an 18% slowdown of *BUPC* MG class B on one processor relative to the one-processor *MPI* version. The *IUPC* version was compiled with the Intrepid compiler based on GCC [12], which performs less aggressive optimization than the MIPSPro compiler. Lower scalar performance of the *IUPC* version results in a similar 48% slowdown.

The one-processor *BUPC* versions of MG class A execute approximately 17% slower than the corresponding *CAF* version (65 seconds vs. 55 seconds). To determine the cause of this performance difference, we used SGI's `perfex` hardware counter-based analysis tool to obtain a global picture of the application's behavior with regards to the machine resources. A more detailed analysis using SGI's `ssrun` and Rice's HPCToolkit, led us to conclude that the *BUPC* version completes 51% more loads than the *CAF* version. The cause of this was the failure of the MIPSPro C compiler to apply loop fusion and alignment to the most computationally intensive loop nest in the application (in the `resid()` routine). The MIPSPro Fortran compiler performed loop fusion and alignment. This reduced the memory traffic by reusing results produced in registers, which in turn improved the software-pipelined schedule for the loop. We expect similar issues to inhibit the performance of other less computationally intensive loops in the BUPC-compiled application.

3.5 NAS CG

Figure 2(a) shows the parallel efficiency of NAS CG class C (problem size 150000) on an Itanium2+Myrinet 2000 cluster. In the figure, *MPI* represents the NPB-2.3 MPI version, *CAF* represents the fastest CAF version, *BUPC* represents a UPC implementation of CG compiled with the Berkeley UPC compiler, *CAF-barrier* represents a CAF version using barrier synchronization, and *BUPC-reduction* represents an optimized UPC version.

The *CAF* version of CG was derived from the *MPI* version by converting two-sided MPI communication into equivalent calls to notify/wait and vectorized one-sided communication [6]. The *BUPC* version is also based on the *MPI* version; it uses UPC shared arrays for communication and split-phase barriers and employs thread-privatization [10] (using regular pointers to access shared data available locally) for improved scalar performance.

The performance of the *MPI* and *CAF* versions is comparable for class C, consistent with our previous studies [6, 7]. The performance of *BUPC* was up to a factor of 2.5 slower than that of *MPI*. By using HPCToolkit, we determined that for one CPU, both the *MPI* and the *BUPC* versions spend most of their time in a loop that performs a sparse vector-matrix product; however, the *BUPC* version spent over twice as many cycles in the loop as the Fortran version. The UPC and the Fortran versions of the loop are shown in Figure 3. By inspecting the Intel C and Fortran compilers optimization report, we determined that the Fortran compiler recognizes that the loop performs a sum reduction and unrolls it, while the C compiler does not unroll it. We manually modified the UPC version of the loop to compute the sum using two partial sums, as shown in Figure 3(c); we denote this version *BUPC-reduction*. On Itanium processors, this leads to a more efficient instruction schedule.

```
sum = 0.0;
for (k = rowstr[j];
     k < rowstr[j+1];
     k++) {
  sum +=
    a[k-1]*p[colidx[k-1]-1];
}
```

(a) UPC

```
sum = 0.d0
do k=rowstr(j),rowstr(j+1)-1
  sum = sum + a(k)*p(colidx(k))
end do
```

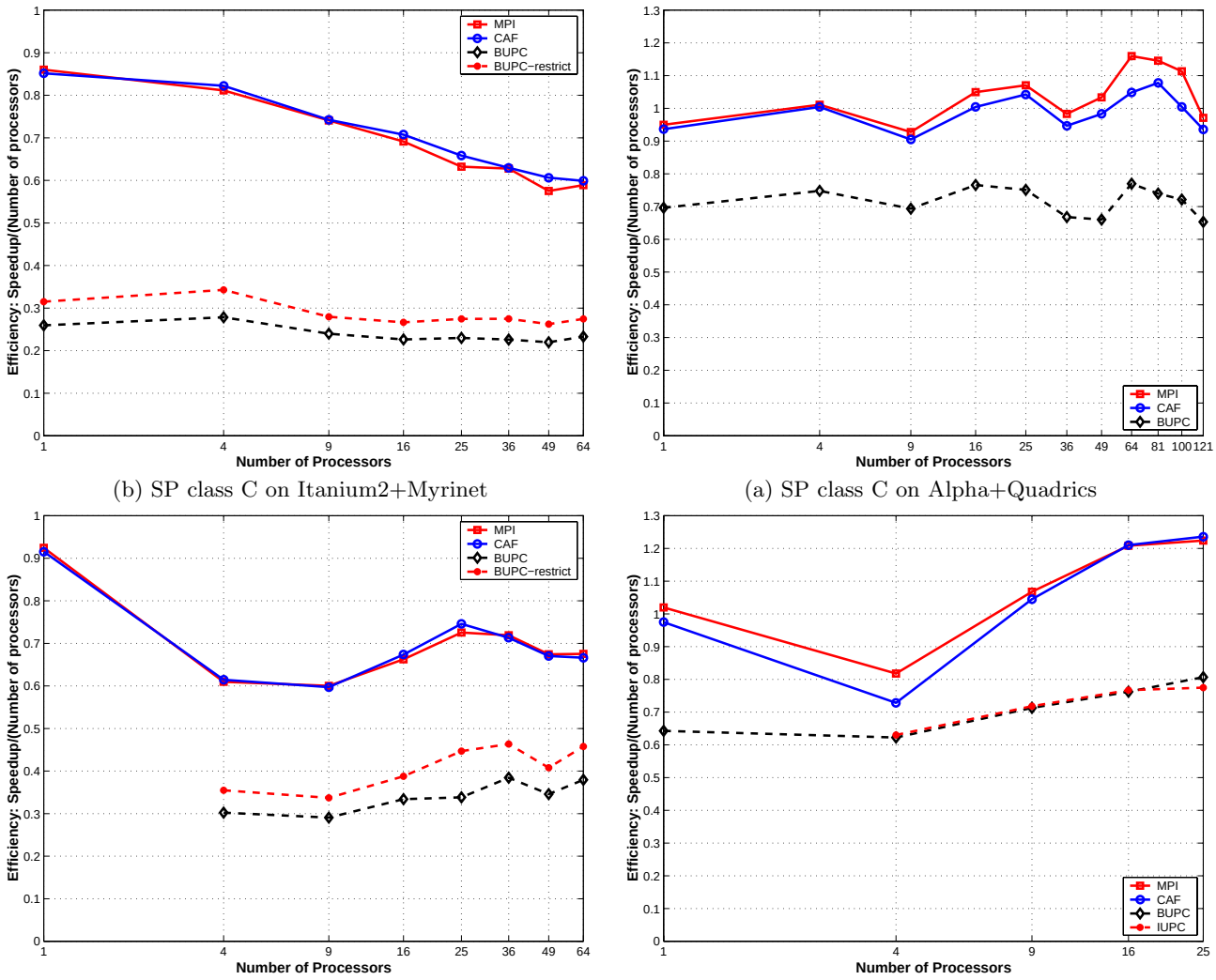
(b) Fortran

```
t1 = t2 = 0
for (...; k+=2 ) {
  t1 += a[k-1] * p[colindex[k-1]-1]
  t2 += a[k] * p[colindex[k]-1]
}
// + fixup code if the range of k isn't even
sum = t1 + t2
```

(c) UPC with sum reduction

Figure 3: UPC and Fortran versions of a sparse matrix-vector product

For one CPU, *BUPC-reduction* achieved the same performance as *MPI*. The graph in Figure 2(a) shows that *BUPC-reduction* is up to 2.6 times faster then *BUPC*. On up to 32 CPUs, *BUPC-reduction* is comparable in performance to *MPI*. On 64 CPUs, *BUPC-reduction* is slower by 20% than the *MPI* version. To explore the remaining differences, we investigated the impact of synchronization. We implemented a CAF version that uses barriers for synchronization to mimic the synchronization present in *BUPC-reduction*. As shown in Figure 2(a), the performance of *CAF-barrier* closely matches that of *BUPC-reduction* for large numbers



(b) SP class C on Itanium2+Myrinet

(a) SP class C on Alpha+Quadrics

(c) SP class C on Altix 3000

(d) SP class B on Origin 2000

Figure 4: Comparison of MPI, CAF and UPC parallel efficiency for NAS SP.

of CPUs; it also experiences a 38% slowdown compared to the *CAF* version.

Figure 2(b) shows the parallel efficiency of NAS CG class B (problem size 75000) on an Alpha+Quadrics cluster. This study evaluated the same versions of the *MPI*, *CAF* and *BUPC* codes as on the Itanium2+Myrinet 2000 cluster. On this platform, the three versions of NAS CG achieve comparable performance. The Compaq compiler was able to optimize the non-unrolled C version of the sparse matrix-vector product loop; for this reason *BUPC* and *BUPC-reduction* yield similar performance.

Figure 2(c) shows the parallel efficiency of NAS CG class C (problem size 150000) on an SGI Altix 3000. This study evaluates the same versions of NAS CG as those used on the Itanium2+Myrinet 2000 cluster. The *CAF* and *MPI* versions have similar performance. *BUPC* is up to a factor of 3.4 slower than *MPI*. *BUPC-reduction* performs comparably to *MPI* on up to 32 CPUs and it is 14% slower on 64 CPUs. The *CAF-barrier* version experiences a slowdown of 19% relative to *CAF*. Notice also that while the performance degradation due to the use of barrier-only synchronization is smaller on the SGI Altix 3000 than on the Itanium2+Myrinet 2000 cluster, it prevents achieving high-performance on large number of CPUs on both architectures.

The parallel efficiency of NAS CG class B (problem size 75000) on the SGI Origin 2000 is shown in Figure 2(d). We used the same *MPI* and *CAF* versions as for the previous three platforms. We used the Berkeley UPC and the Intrepid UPC compilers to build the UPC codes; the corresponding versions are *BUPC* and *IUPC*. On this platform, *MPI*, *CAF* and *BUPC* have comparable performance across the range of CPUs. In each case, the MIPSPro compilers were able to optimize the sparse matrix-vector product loop automatically and effectively; consequently, using the partial sums version didn't boost performance. We also didn't notice a performance difference between *CAF* and *CAF-barrier*. The *IUPC* version is up to 50% slower than the other three versions. The principal loss of performance stems from ineffective optimization of the sparse-matrix vector product computation. *IUPC-reduction* represents an IUPC-compiled version of UPC CG with the sparse matrix-

vector product loop unrolled; this version is only 12% slower than *MPI*.

3.6 NAS SP

Figure 4(a) shows the parallel efficiency curves for NAS SP class C (problem size 162^3) on the Itanium2+Myrinet2000 cluster. The *MPI* curve serves as the baseline for comparison and represents the performance of the original NPB-2.3 SP benchmark. The *CAF* curve shows the performance of the fastest CAF variant. It uses point-to-point synchronization and employs non-blocking communication to better overlap communication with computation. The *BUPC* and *BUPC-restrict* curves show the performance of two versions of SP compiled with the Berkeley UPC compiler.

The performance of the *CAF* version is roughly equal to that of *MPI*. Similar to the other UPC NAS benchmarks compiled using the back-end Intel C compiler, the scalar performance suffers from poor alias analysis: the one-processor version of *BUPC* class C is 3.3 times slower than the one-processor *MPI* version. Using the `restrict` keyword to improve alias analysis yielded a scalar performance boost: the one-processor version of *BUPC-restrict* class C is 18% faster than *BUPC*. The trend is similar for larger number of CPUs.

There is a conceptual difference in the communication implementation of the dimensional sweeps in *CAF* and *BUPC*. The CAF implementation uses point-to-point synchronization, while the UPC implementation uses split-phase barrier synchronization. In general, it is simpler to use split-phase barrier synchronization, however, for NAS SP, point-to-point and split-phase barrier synchronization are equally complex. Since barrier synchronization is stronger than necessary in this context, it could potentially degrade performance.

Figure 4(b) reports the parallel efficiency of the *MPI*, *CAF*, and *BUPC* versions of NAS SP class C (problem size 162^3) on the Alpha+Quadrics platform. It can be observed that the performance of *CAF* is slightly worse than that of *MPI*. We attribute this to the lack of non-blocking notification support in the CAF runtime layer. The performance of the *BUPC* version is lower than that of *MPI* due to worse scalar performance of the C code: it is 1.4 times slower for one processor and increases to 1.5 times slower for 121 processors. The use of the `restrict` keyword does not have any effect on performance because of the high quality dependence analysis of the Alpha C compiler.

Figure 4(c) shows the efficiency of *MPI*, *CAF*, *BUPC*, and *BUPC-restrict* versions of NAS SP class C (problem size 162^3) on the Altix 3000 system. The performance of *CAF* and *MPI* is virtually identical, while *BUPC* is a factor of two slower on four processors (we were not able to run one-processor version due to memory constraints). Using the `restrict` keyword improves performance on average by 17%.

Figure 4(d) shows parallel efficiency of *MPI*, *CAF*, *BUPC*, and *IUPC* versions of NAS SP for class B (problem size 102^3) on the Origin 2000 machine. The performance of *CAF* is very close to that of *MPI*. Both UPC versions have similar performance and are slower than *MPI* or *CAF*. Again, the difference is attributable to lower scalar performance. For the one-processor SP class B, *BUPC* was 57% slower than *MPI*.

We observed that the one-processor BUPC-compiled version of SP class A executes approximately 43% slower than the corresponding *MPI* version. Using hardware performance counters, we found that the BUPC-compiled version executed twice as many instructions as the CAF version.

A detailed analysis of this difference using HPCToolkit and SGI's `ssrun` helped us identify that a computation intensive single-statement loop nest present in both versions was getting compiled ineffectively for UPC. In Fortran, the loop, which accesses multidimensional array parameters, was unrolled & jammed to improve outer loop reuse and software pipelined by the MIPSPro Fortran compiler. The corresponding UPC loop, which accesses 1D linearized C arrays through pointers, was not unrolled or software pipelined by the MIPSPro C compiler, leading to less efficient code. This more than doubled the number of graduated instructions for the loop in the *BUPC* version compared to the *MPI* version.

The generated code for the Fortran loop was able to reuse not only floating point data, but integer address arithmetic as well. We believe that this specific observation applies to many of the single-statement computationally intensive loops present in SP's routines. For multi-statement loops, the difference in the number of executed instructions between the UPC and MPI versions was not as significant. Such loops already have opportunities for reusing address arithmetic and floating point values even without applying transformations such as unroll & jam.

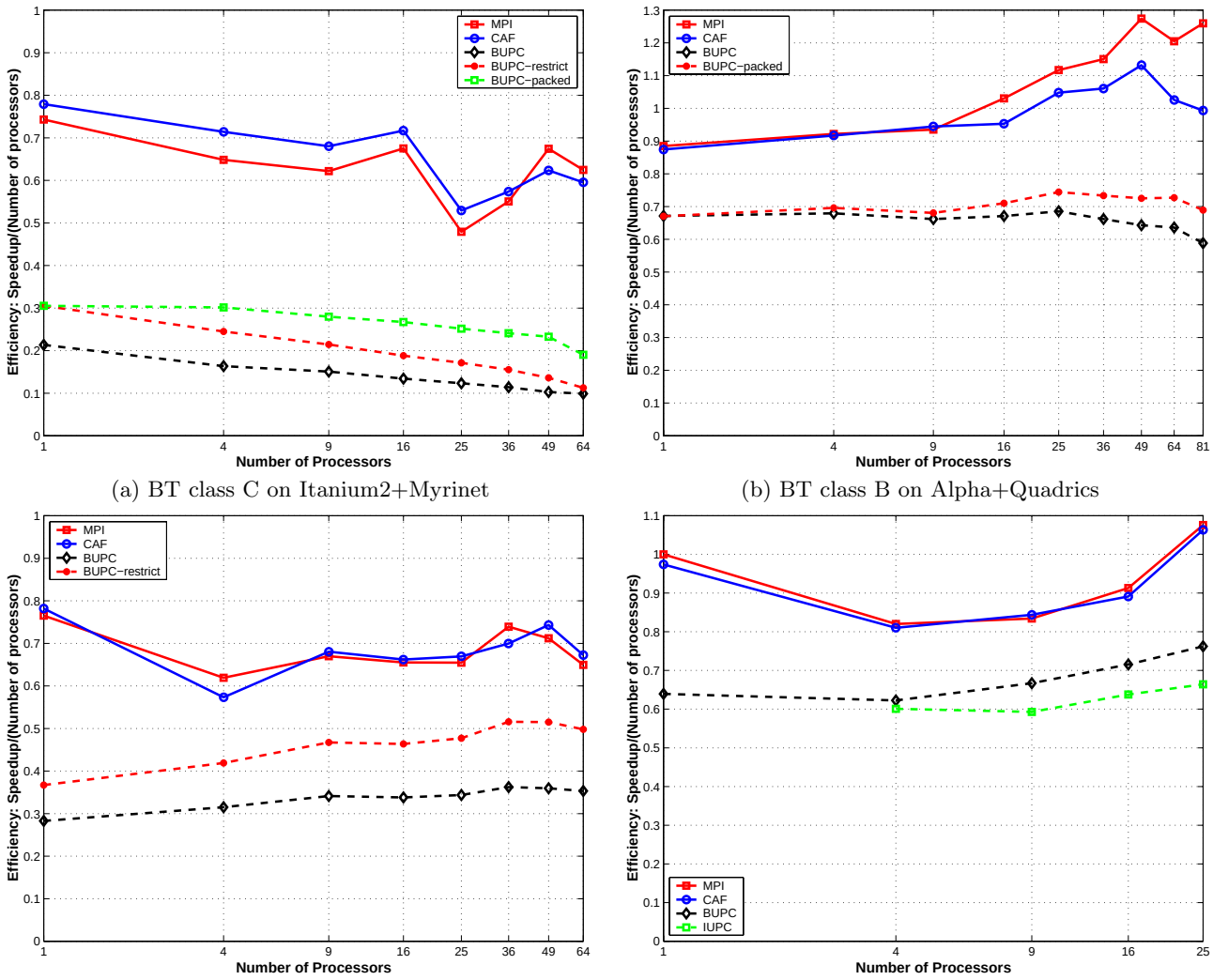
3.7 NAS BT

In Figure 5(a), we present parallel efficiency results of NAS BT class C (problem size 162^3) on an Itanium2+Myrinet 2000 cluster. We used the NPB-2.3 MPI version, *MPI*, the most efficient CAF version, *CAF*, a UPC implementation similar to *MPI* and compiled with the Berkeley UPC compiler, *BUPC*, and two optimized UPC versions, *BUPC-restrict* and *BUPC-packed*. Due to memory constraints, we couldn't run the sequential Fortran version of BT for class C; to compute parallel efficiency we assume that the efficiency of MPI on four CPUs is one, and compute the rest of the efficiencies relative to that baseline performance.

The CAF implementation of BT is described in more detail elsewhere [6, 7]. It uses communication vectorization, a trade-off between communication buffer space and amount of necessary synchronization, procedure splitting and non-blocking communication. It also uses the packing of strided `PUTs`, due to inefficient multi-platform support of strided `PUTs` by the CAF runtime; we are actively working to address this issue.

The performance of the *CAF* version is better than or equal to that of *MPI*. The performance of the initial UPC version, *BUPC*, was up to a factor of five slower than that of the MPI version. By using HPCToolkit, we determined that several routines that perform computation on the local part of shared data, namely `matmul_sub`, `matmul_vec`, `binvrhs`, `binvcrhs` and `compute_rhs`, are considerably slower in *BUPC* compared to the *MPI* version. To reduce overly conservative assumption about aliasing, we added the `restrict` keyword to the declarations of all the pointer arguments of the subroutines `matmul_sub`, `matmul_vec`, `binvrhs`, and `binvcrhs`. The modified UPC version of NAS BT is *BUPC-restrict*; it is up to 42% faster than *BUPC*.

To investigate the impact of communication performance on parallel efficiency, we instrumented all NAS BT versions to record the times spent in communication and synchronization. We found that *BUPC-restrict* spent about 50-



(a) BT class C on Itanium2+Myrinet

(b) BT class B on Alpha+Quadrics

(c) BT class B on Altix 3000

(d) BT class A on Origin 2000

Figure 5: Comparison of MPI, CAF and UPC parallel efficiency for NAS BT.

100 times more in communication on the Itanium2+Myrinet 2000 cluster because the communication in the sweeps was not fully vectorized; it transfers a large number of messages of 25 double precision numbers. In [7] we show that, in the absence of efficient runtime support for strided communication, packing for the *CAF* version of BT can improve performance by as much as 30% on cluster platforms.

We transformed the *BUPC-restrict* version to perform packing and unpacking and used the UPC `upc_memget` primitive to communicate the packed data; the resulting version with packed communication is denoted *BUPC-packed*. This version is up to 32% faster than *BUPC-restrict*. Overall, *BUPC-packed* yields a factor of 2.44 improvement over *BUPC*.

In Figure 5(b) we present the results for NAS BT class B[6](problem size 102^3) on an Alpha+Quadrics cluster. The *MPI* version yields the best performance; *CAF* is up to 26% slower than *MPI*, and *BUPC* is up to two times slower than *MPI*. As noticed in section 3.6 for SP class C on the Alpha+Quadrics cluster, using the `restrict` keyword didn't have an effect; consequently, *BUPC* and *BUPC-restrict* have similar performance. This shows that even though the back-end C compiler can optimize routines such as `matmul_sub`, `matmul_vec`, `binvrhs`, and `binvcrhs`, which contain at most one loop or just straight-line code, it has difficulties optimizing `compute_rhs`. This subroutine contains several complex loop nests and performs references to the local parts of multiple shared arrays using private pointers; this poses a challenge to the back-end C compiler. In the *CAF* version, `compute_rhs` performs the same computations on local parts of co-arrays; to convey the lack of aliasing to the back-end Fortran compiler we use procedure splitting. Packing of communication led to a performance gain: *BUPC-packed* is up to 14% faster than *BUPC*, although it is still up to 82% faster than *MPI*.

[6]We used class B due to limitations encountered for class C for the *CAF* and *BUPC* versions. *CAF* could not allocate the large data size required for BT class C on small number of processors, while *BUPC* could not allocate memory for a number of threads larger than 100.

In Figure 5(c) we present the results for NAS BT class B (problem size 102^3) on an SGI Altix 3000 platform. We studied class B, due to memory and time constraints on the machine. The *MPI* and *CAF* versions have similar performance, while *BUPC* is up to two times slower than *MPI*. *BUPC-restrict* is up to 30% faster than *BUPC* and up to 43% slower than *MPI*. *BUPC-packed* has the same performance as *BUPC-restrict*. Packing didn't improve the performance because fine-grain data transfers are efficiently supported in hardware.

Finally, in Figure 5(d) we present results on the SGI Origin 2000 machine. We studied class A (problem size 64^3) of NAS BT due to memory and time constraints. The *CAF* and *MPI* versions perform comparably, while *BUPC* performs 40% slower than the *MPI* version. Similar to our experiences with the other benchmarks, using `restrict` didn't improve the performance of *BUPC-restrict*, and similar to the SGI Altix 3000, communication packing didn't improve the performance of *BUPC-packed*.

4. CONCLUSIONS

Our study showed that both UPC and CAF versions of the NAS benchmarks can yield scalable performance on cluster platforms when using bulk communication. Currently, neither the Rice `cafc` compiler for CAF nor the Berkeley and Intrepid UPC compilers automatically synthesize bulk communication for multi-dimensional arrays from element-wise references to remote data. Thus, the responsibility for achieving bulk communication currently falls on programmers. For CAF, programming bulk communication requires accessing remote co-array data using Fortran 90 array triplet notation, which is not a significant hardship. For UPC, a programmer must code bulk communication explicitly using UPC communication primitives such as `upc_memput`, which is no easier than using MPI.

Our experiments showed that support for communicating strided sections efficiently is necessary for achieving top performance with regular codes. For instance, using strided communication primitives instead of element-wise communication boosts performance 13-28% for BUPC MG on 64 processors on the Itanium2+Myrinet platform. While users can manually pack and unpack data in CAF and UPC programs to boost communication performance, this is tedious. CAF and UPC compilers and runtime systems should provide efficient support for strided transfers. For CAF, the Fortran 90 strided array sections syntax provides a convenient way to express strided transfers, while in UPC, the programmer uses a low-level UPC extensions such as `upc_memput_fstrided`, which requires manually computing memory strides, chunk sizes, and stride counts for each strided transfer.

Our experiments also showed that using only barriers for synchronization can significantly hurt performance. There are three alternatives. One can do nothing and accept performance losses that we observed as high as 30–50% for CAF MG on the Itanium2+Myrinet cluster; one can develop synchronization strength reduction algorithms and rely on CAF and UPC compilers to replace barriers with point-to-point synchronization; or one can add point-to-point primitives to UPC as we have previously proposed adding to CAF. We believe that developing effective compiler algorithms for synchronization strength reduction is appropriate. However, we also believe that having point-to-point synchronization available explicitly within the languages is important to avoid performance loss when compiler optimization is inadequate. Working with the BUPC developers, we created a language-level implementation of point-to-point synchronization primitives in UPC, and used it in the MG benchmarks. We observed perfomance improvements of 14-49% improvement on an Itanium2+Myrinet cluster and 52% improvement on an SGI Altix 3000 for 64 processor runs. It is worth mentioning that a language-level implementation of notify/wait will not allow overlapping of notifications with asynchronous communication events on all interconnects. We believe that a specialized, lower-level implementation of the point-to-point primitives is necessary to provide such an overlap.

The only feasible way to build multi-platform CAF or UPC compilers is to employ source-to-source translation. This enables leveraging existing compilers for C and Fortran on the target architectures. However, employing this strategy successfully requires generating code that minimizes the impact of back-end compiler limitations, such as conservative alias and dependence analysis. Analysis shortcomings in back-end compilers inhibit important high-level loop transformations, such as unroll-and-jam and software pipelining, which are critical to performance on modern microprocessor-based systems.

Our experiments showed that compiler loop transformations including alignment, fusion, software pipelining, unroll and jam, and optimization of reductions were responsible for dramatically improving the efficiency of CAF and F77+MPI codes. However, in many cases, C compilers compiling BUPC-generated code or the Intrepid UPC compiler failed to apply these key optimizations; this was a major impediment to achieving high-performance with UPC.

Unlike CAF, UPC does not provide special syntax for accessing the local part of shared arrays. If programmers reference the local part implicitly through pointers to shared objects, the resulting code will be ineficient due to unnecessary UPC runtime address translation. To work around this, UPC programmers can use private C pointers to access local part of shared objects. This is cumbersome and also introduces extra aliasing in the code. Another impediment to analysis and optimization of UPC codes is the use of linearized subscripts to access multi-dimensional data through pointers. Currently, multi-dimensional shared arrays are not a primitive in UPC because of the difficulty of managing multi-dimensional data distributions at run-time. In some cases, by marking pointers with `restrict` in the UPC codes, we were able to assist the vendor C compilers in avoiding worst-case assumptions about linearized subscripts so compilers could generate better code through software pipelining. In other cases, even after restricting pointers, we never observed any high-level transformations such as fusion or unroll-and-jam being applied on the UPC codes.

Extending the UPC model to incorporate multidimensional shared arrays with symbolic bounds would enable a UPC compiler to perform more accurate alias and dependence analysis and consequently to apply high-level loop transformations. Having an explicit syntax for local accesses to shared arrays would permit simple programming and the use of restricted pointers in the code generated by a UPC compiler. Improving the quality of back-end C compilers is critical if UPC performance is to match that of MPI on scientific codes.

Acknowledgments

We thank J. Nieplocha and V. Tipparaju for collaborating on the refinement and tuning of ARMCI. We thank F. Zhao and N. Tallent for their work on the Open64/SL Fortran front-end. We thank D. Bonachea, C. Bell, J. Duell, W. Chen, C. Iancu, and K. Yelick for their discussions and assistance regarding the Berkeley UPC compiler, GASNet library, UPC extensions, and UPC language-level point-to-point synchronization.

5. REFERENCES

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[2] D. Bonachea. Gasnet specification, v1.1. Technical Report CSD-02-1207, U.C. Berkeley, October 2002.

[3] D. Bonachea. Proposal for extending the upc memory copy library functions and supporting extensions to gasnet, v1.0. Technical Report LBNL-56495, Lawrence Berkeley National, October 2004.

[4] F. Cantonnet, Y. Yao, S. Annareddy, A. Mohamed, and T. El-Ghazawi. Performance monitoring and evaluation of a UPC implementation on a NUMA architecture. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Nice, France, Apr. 2003.

[5] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. In *Proceedings of the 17th ACM International Conference on Supercomputing*, San Francisco, California, June 2003.

[6] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-array Fortran Performance and Potential: An NPB Experimental Study. In *Proc. of the 16th Intl. Workshop on Languages and Compilers for Parallel Computing*, number 2958 in LNCS. Springer-Verlag, October 2-4, 2003.

[7] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A Multiplatform Co-Array Fortran Compiler. In *Proceedings of the 13th Intl. Conference of Parallel Architectures and Compilation Techniques*, Antibes Juan-les-Pins, France, September 29 - October 3 2004.

[8] Y. Dotsenko, C. Coarfa, J. Mellor-Crummey, and D. Chavarría-Miranda. Experiences with Co-Array Fortran on Hardware Shared Memory Platforms. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, September 2004.

[9] T. El-Ghazawi, F. Cantonne, P. Saha, R. Thakur, R. Ross, and D. Bonachea. *UPC-IO: A Parallel I/O API for UPC v1.0*, July 2004. Available at `http://upc.gwu.edu/docs/UPC-IOv1.0.pdf`.

[10] T. A. El-Ghazawi and F. Cantonnet. UPC performance and potential: A NPB experimental study. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (CDROM)*, Baltimore, MD, Nov. 2002. IEEE Computer Society.

[11] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. *UPC Language Specifications v1.1.1*, October 2003.

[12] Intrepid Technology Inc. GCC Unified Parallel C. `http://www.intrepid.com/upc`.

[13] J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, 2002. *Special Issue with selected papers from the Los Alamos Computer Science Institute Symposium.*

[14] V. Naik. A scalable implementation of the NAS parallel benchmark BT on distributed memory systems. *IBM Systems Journal*, 34(2), 1995.

[15] J. Nieplocha and B. Carpenter. *ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems*, volume 1586 of *Lecture Notes in Computer Science*, pages 533–546. Springer-Verlag, 1999.

[16] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, August 1998.

[17] Open64 Developers. Open64 compiler and tools. `http://sourceforge.net/projects/open64`, Sept. 2001.

[18] Open64/SL Developers. Open64/SL compiler and tools. `http://hipersoft.cs.rice.edu/open64`, July 2002.

[19] Rice University. HPCToolkit performance analysis tools. `http://www.hipersoft.rice.edu/hpctoolkit`.

[20] Rice University. `cafc` - A Multiplatform, Open Source Co-Array Fortran Compiler. `http://www.hipersoft.rice.edu/caf`, Apr. 2005.

[21] E. Wiebel, D. Greenberg, and S. Seidel. *UPC Collective Operations Specifications v1.0*, December 2003. Available at `http://upc.gwu.edu/docs/UPC_Coll_Spec_V1.0.pdf`.