Productivity and Performance Using Partitioned Global Address Space Languages

Katherine Yelick[1,2], Dan Bonachea[1,2], Wei-Yu Chen[1,2], Phillip Colella[2], Kaushik Datta[1,2], Jason Duell[1,2], Susan L. Graham[1], Paul Hargrove[1,2], Paul Hilfinger[1], Parry Husbands[1,2], Costin Iancu[2], Amir Kamil[1], Rajesh Nishtala[1], Jimmy Su[1], Michael Welcome[2], and Tong Wen[2]

University of California at Berkeley[1]
Lawrence Berkeley National Laboratory[2]

titanium-group@cs.berkeley.edu, upc@lbl.gov

ABSTRACT

Partitioned Global Address Space (PGAS) languages combine the programming convenience of shared memory with the locality and performance control of message passing. One such language, Unified Parallel C (UPC) is an extension of ISO C defined by a consortium that boasts multiple proprietary and open source compilers. Another PGAS language, Titanium, is a dialect of Java^{TM} designed for high performance scientific computation. In this paper we describe some of the highlights of two related projects, the Titanium project centered at U.C. Berkeley and the UPC project centered at Lawrence Berkeley National Laboratory. Both compilers use a source-to-source strategy that translates the parallel languages to C with calls to a communication layer called GASNet. The result is portable high-performance compilers that run on a large variety of shared and distributed memory multiprocessors. Both projects combine compiler, runtime, and application efforts to demonstrate some of the performance and productivity advantages to these languages.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; E.1 [**Data Structures**]: Distributed data structures

General Terms

Languages, Performance, Human Factors

Keywords

UPC, Titanium, PGAS, Partitioned Global Address Space, GASNet, One-Sided Communication, NAS Parallel Benchmarks

1. INTRODUCTION

PASCO'07, July 27–28, 2007, London, Ontario, Canada.

Partitioned global address space (PGAS) languages offer programming abstractions similar to shared memory, but with control over data layout that is critical to high performance and scalability. The most common PGAS languages include Unified Parallel C (UPC) [34], Titanium (a scientific computing dialect of Java) [17], and Co-Array Fortran (CAF) [27]. In this paper we describe our experience with the first two languages, UPC and Titanium. Compared to approaches in which programmers use two-sided message passing, these languages offer significant advantages in productivity and, perhaps more surprisingly, real performance advantages due to their use of faster one-sided communication. We describe some of these benefits by studying microbenchmarks and applications written in both languages.

Prerequisites for any successful parallel programming language include ubiquity across parallel machines and interoperability with other languages and libraries. We have developed portable compilers for both UPC and Titanium based on source-to-source translation and the use of a lightweight communication layer called GASNet. Both languages run on most high end machines as well as laptops, desktops, and generic clusters; they can interoperate with Fortran, C, C++, and MPI, so that programmers interested in experimenting with these languages can write part of their code in a PGAS language without rewriting the entire application.

There are several commercial and open-source compilers available for UPC [5, 11, 15, 19, 26]. In this paper we used the Berkeley UPC compiler [5], which translates UPC to ISO-compliant C using a compiler based on the Open64 infrastructure [28]. The Titanium compiler [33] uses a similar source-to-source model, translating the extended Java language to C. There is no Java Virtual Machine in the execution of Titanium, and therefore some Java features such as dynamic class loading are not supported. In both cases the translators perform serial and parallel optimizations that we have described previously [9, 18, 32, 35], although the languages also have sufficient support to allow for many kinds of hand-tuning. On a shared memory machine, accesses to the global address space translate into conventional load/store instructions, while on distributed memory machines, they translate into calls to the GASNet layer [6].

This paper gives an overview of the PGAS model and the two languages (section 2), the performance implications of

one-sided communication (section 3), hand-optimized benchmarks (section 4), some of our application experience (section 5), and optimization techniques developed in the two compilers (section 6). We end with a summary of the ongoing and future plans for PGAS languages.

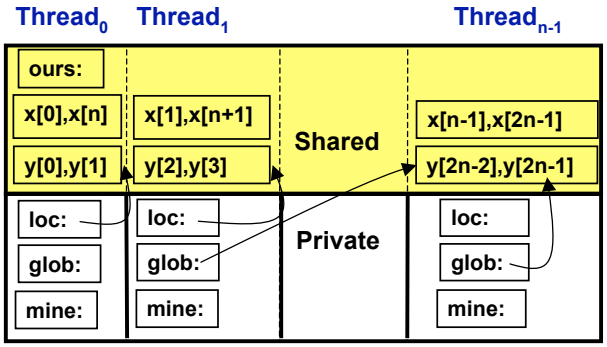
2. UPC AND TITANIUM

In a partitioned global address space model, a thread running on one processor can directly read or write the memory associated with another. This supports the construction of large shared data structures, such as sets, trees, matrices and other array-based and pointer-based data structures. The term *global address space* is used instead of *shared memory* to distinguish the *semantic* notion that the collective memory of a set of threads is addressable by each (global address space) from the *hardware implementation technique* in which several processors can address a shared memory and locally cache its contents. Implementations of a global address space language *may* run on shared-memory hardware or software that caches remote values, but the languages are designed to encourage programmers to associate parts of a distributed data structure with each thread and assume that accesses to other parts are more expensive.

The upper half of Figure 1 shows a picture of the global address space. Each thread has a space for private local memory and some partition of the shared space to which it has *affinity*. Variables in the shared space are accessible to other threads through pointers and distributed arrays. Both UPC and Titanium distinguish local and global pointers: local pointers can only reference data within the same memory partition, whereas global pointers can refer to data in another partition. Local pointers are generally representable as a simple memory address. On machines without hardware support for a global address space, global pointers require additional information to encode which thread identifier has affinity to the data and an extra test for locality when the pointer is dereferenced. This test is negligible when the data is remote, but can be significant for truly local data.

The lower half of Figure 1 shows some example UPC variable declarations that are also represented in the picture. A private object may only be accessed by its corresponding thread, whereas all threads can read or write any object in the shared address space. The UPC code contains two scalar integers, one private (`mine`) and one shared (`ours`), as well as two shared arrays, one cyclically mapped (`x`) and one blocked (`y`). The *local pointer*, `loc`, may only refer to variables in the local partition, but they can be in either private or shared space. The *global pointer*, `glob`, may refer to remote or local variables. Titanium's type system has the same local vs. global distinction on pointers (i.e., references), although it lacks the distributed array declarations provided by UPC. The two languages also differ on the default reference type, which is local in UPC and global in Titanium. In both languages, the partitioning of the shared space into regions with logical affinity to threads allows programmers to explicitly control data layout, which is then used by the runtime system to map threads and their associated data to processors: on a distributed memory machine, the local memory of a processor holds both the thread's private data and the shared data with affinity to that thread.

Each PGAS language design attempts to respect the language philosophy of their underlying sequential language.




```
#define n THREADS
shared int ours;
int mine;
shared double x[2*n];
shared [2] double y[2*n];
double *loc;
double shared *glob;
```

Figure 1: Partitioned Global Address Space and a UPC code fragment showing variable declarations.

UPC is the lower-level language and retains from C the ability to use pointers to access arrays, with full address arithmetic and the exposed memory layout that this implies. Titanium maintains Java's type-safety features, and extends then with numerous high-level features designed to improve productivity and performance for scientific parallel applications. What distinguishes the class of PGAS languages from others is the global address space and its partitioning into chunks that can be mapped to local memory on clusters or distributed memory supercomputers.

UPC and Titanium both use a Single Program Multiple Data (SPMD) programming model of parallelism in which the number of threads is fixed at program startup time and is equal to the number of logical memory partitions. Not all PGAS languages use this static parallelism model—the DARPA HPCS languages, for example, combine dynamic multithreading with the partitioned global address space [1, 7, 38], and data-parallel languages use a global address space with array layout statements that provide a flavor of memory partitioning [16, 31].

3. PGAS AND ONE-SIDED COMMUNICATION

PGAS languages rely on one-sided communication: a thread directly accesses remote memory without involving the application program on the remote node. On many systems, it is natural to implement these semantics with Remote Direct Memory Accesses (RDMA): the remote CPU is not involved in the transfer, but instead the network interface directly responds to remote requests for data. One-sided communication avoids the overhead of message and tag matching, decouples data transfer from interprocess synchronization, and also allows transfers to be reordered, since a data transfer encodes information about where the data should be placed in memory, rather than relying on the order of receive operations in a remote program. The picture on the left-hand side of Figure 2 shows the basic difference

in one-sided vs. two-sided messages. A two-sided message contains a message identifier that must be matched with a receive operation on the remote side to find the location in memory where the data should go; a one-sided message directly encodes the remote address.

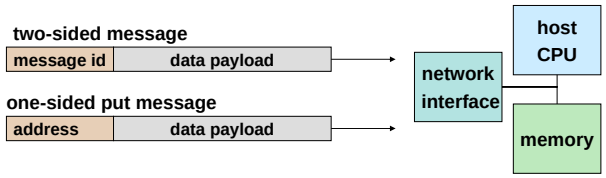


Figure 2: Picture of one-sided vs. two-sided transfer.

The Titanium and Berkeley UPC compilers translate to C with calls to a runtime layer that implements the parallelism features of the languages. Communication in both cases goes through GASNet [6, 14], which provides portable, high-performance, one-sided communication in the form of put and get operations for small and large messages, strided and indexed data, and blocking and nonblocking semantics. GASNet is built on an extensible core based on the concept of Active Messages, which is used to support remote locking and various runtime operations such as data packing. Basic put and get primitives can be implemented as simple Active Message calls, but typically use special hardware support for shared memory or RDMA operations. GASNet has optimized network-specific implementations for the Cray XT networks (Portals), IBM SP network (LAPI), Myrinet (GM), InfiniBand, Quadrics (elan3/4), SGI Altix (shmem), and Cray X1 (shmem). To ensure portability (although not recommended for performance), there are also implementations of GASNet for Ethernet and MPI.

Our research using GASNet has shown that the one-sided model has basic performance advantages for latency and bandwidth over two-sided communication. Figure 3 compares the latency and bandwidth of GASNet and MPI. The top graph shows that GASNet has consistently lower latency across several networks. In addition, the bottom graphs display bandwidth that is at least as high as MPI's, with a significant advantage noticeable at the 4KB message size. This shows that many networked cluster systems are better suited to the kind of one-sided communication found in UPC and Titanium than to MPI's two-sided message-passing model. The performance results demonstrate that the GASNet communication layer matches or exceeds the performance of MPI message-passing in all cases, notably providing a significant improvement in small message round-trip latencies and medium-sized message bandwidths. The primary explanation for the performance gap is fundamental to the communication semantics: GASNet's put/get primitives were specifically designed to map very closely to the RDMA and distributed shared-memory capabilities of modern interconnects.

4. HAND-OPTIMIZED BENCHMARKS

The performance benefits of one-sided communication are not limited to microbenchmarks. In a case study of a 3D FFT computation [4] (one of the NAS benchmarks [2]), which is notoriously limited by bisection-bandwidth due to a global transpose, we found that the ability to overlap communication with computation allows for a more effective use of the network and memory systems. Comparing multiple implementations of the benchmark in UPC and MPI (both using the same serial FFT code for normalization), the best UPC implementation was able to profitably send smaller messages than the best MPI version and thereby achieve more effective, finer-grained overlap, thanks to the superior small message bandwidth. The three implementations were: 1) a *chunk* algorithm that packs data from multiple planes to send only a single message in each direction per processor pair; 2) a *slab* algorithm, which sends contiguous data in a given plane as soon as the local FFTs on a set of rows destined for the same remote thread are complete; 3) a *pencil* algorithm that sends each row individually after the FFT on that row is complete.

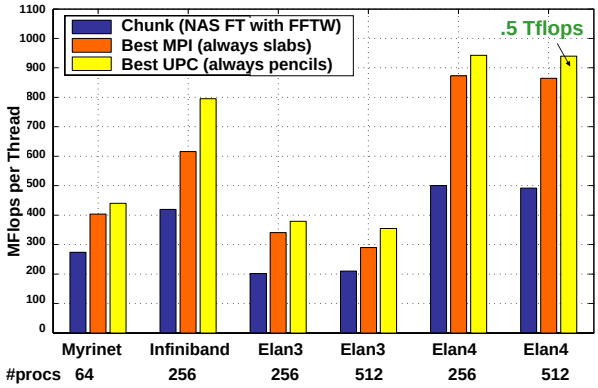


Figure 4: Comparison of 3D FFT performance across several machines using a bulk-synchronous MPI implementation that minimizes message counts but precludes overlap, an MPI code that uses overlap, and a UPC code that uses finer-grained overlap and smaller messages.

Figure 4 shows the results of the FFT study on several machines with networks that support RDMA. The FFT computation involves local FFTs, followed by a transpose, followed by more local FFTs. The first bar is the traditional bulk-synchronous algorithm in which the transpose is performed in a separate phase. It has no overlap of communication with computation, but minimizes the number of messages. The second bar is the best MPI performance we obtained with all three algorithms, which is the slab algorithm in all cases. The third is the best UPC performance we obtain, again over all three algorithms, which in this case is the pencil algorithm for this set of machines and problem sizes. The UPC implementation outperforms the MPI implementation on all of these machines; MPI could not run the small-message *pencil* algorithm as effectively as the slab implementation or the UPC pencil version because the per-message cost of the small MPI messages is too high. The UPC code obtains some of its advantages from overlap, and some from sending smaller messages, which are a more natural fit to the data structures used in the computations.

The performance advantage from one-sided communication is not evident in all applications, although performance is generally comparable to that of MPI code. Figure 5 shows the performance for two other NAS Parallel Benchmarks,

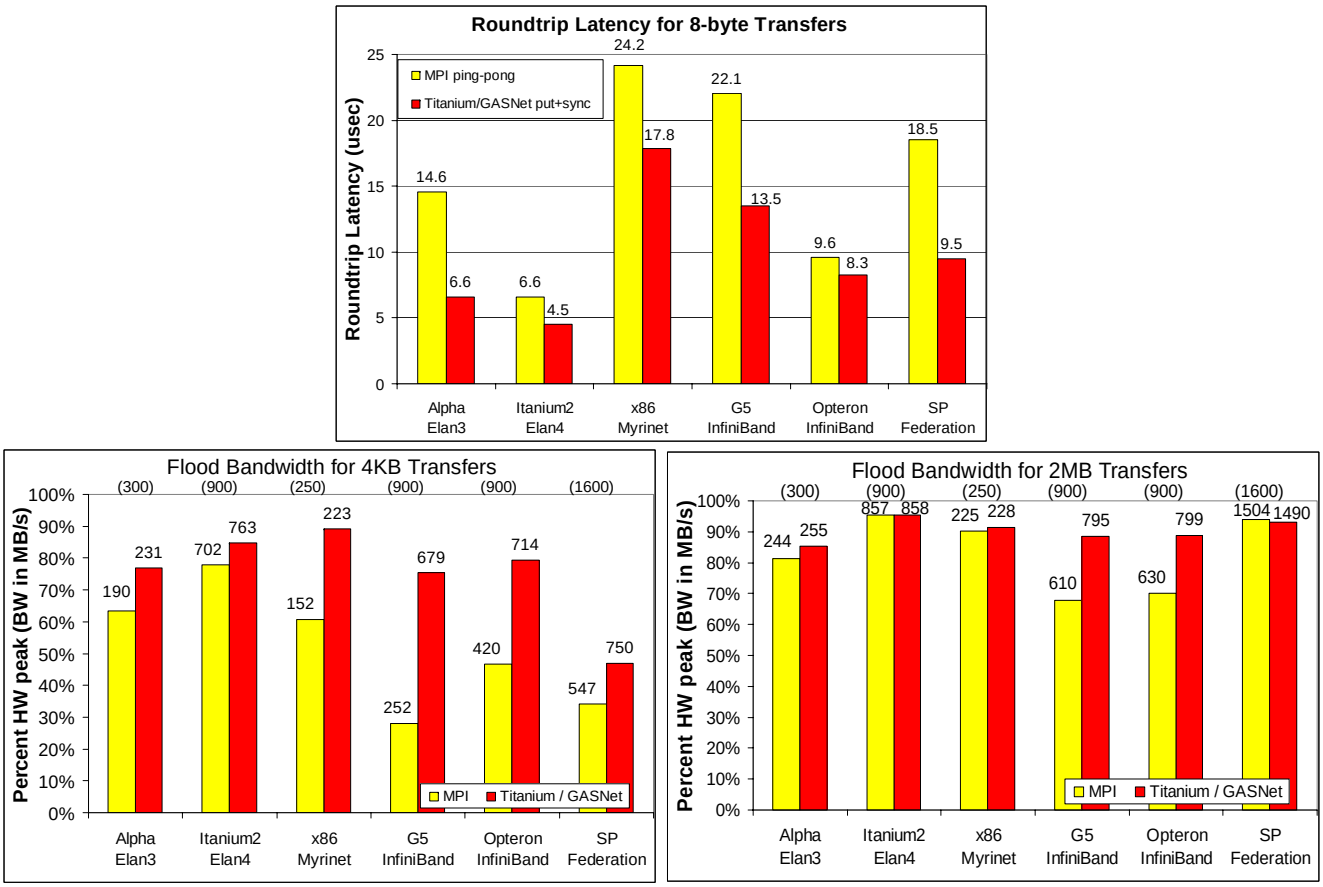


Figure 3: Performance of GASNet and MPI. The top graph shows *round-trip* latency for an 8-byte raw MPI message-passing ping-pong (best case over MPI_Send/MPI_Recv or MPI_ISend/MPI_IRecv) against a GASNet blocking put operation (which blocks for the round-trip acknowledgment). The bottom graphs show the bandwidth for 4KB messages (left) and 2MB message (right) as a percentage of hardware peak for a unidirectional message flood of the given size, and no unexpected MPI messages. Hardware peaks are computed as the minimum of the I/O bus bandwidth and link speed, and are shown in parentheses, while the bar labels show the absolute value in MB/s ($MB = 2^{20}$ $bytes$).

Conjugate Gradient (CG) and Multigrid (MG). The plots show speedup, normalized to the best performance at the smallest processor count. (The problem sizes shown here are for class D, the largest class defined by NAS, and are too large to run on smaller processor counts.) The NAS MG benchmark shows very close performance between Titanium and Fortran with MPI; there is a noticeable gap in the CG performance, due in part to the lack of collective communication support (e.g., reductions) for thread subsets. More details are available in [12].

5. APPLICATIONS

In addition to the benchmarks described in section 4, the Titanium and UPC teams have both benefited from application development in their languages, sometimes decoupled from the team, but in most cases as an integrated part of the compiler and language development effort. In Titanium, the application experience includes five of the NAS Benchmarks: Conjugate Gradient (CG), Multigrid (MG), 3D FFT (FT), Integer Sort (IS) and Embarrassingly Parallel (EP) kernels [2]. In addition, Yau developed a distributed matrix library that supports blocked-cyclic layouts and implemented Cannon's Matrix Multiplication algorithm and Cholesky and LU factorization (without pivoting). Balls and Colella built a 2D version of their Method of Local Corrections algorithm for solving the Poisson equation for constant coefficients over an infinite domain (PPS) [3]. Bonachea, Chapman and Putnam built a Microarray Optimal Oligo Selection Engine for selecting optimal oligonucleotide sequences from an entire genome of simple organisms, to be used in microarray design. Our most ambitious efforts have been application frameworks for Adaptive Mesh Refinement (AMR) algorithms [36, 37] and Immersed Boundary (IB) method simulations. Specific application of these frameworks are an AMR elliptic solver (AMR-Poisson) and a heart simulation using the IB method. In both cases, these application efforts have taken a few years and were preceded by implementations of Titanium codes for specific problem instances, e.g., older AMR Poisson [29], AMR gas dynamics [24], and IB for 1D immersed structures [25, 39]. In addition, several smaller benchmarks were used in evaluating the compiler, such as dense and sparse matrix-vector multiplication (demv and spmv), a simple Monte Carlo cal-

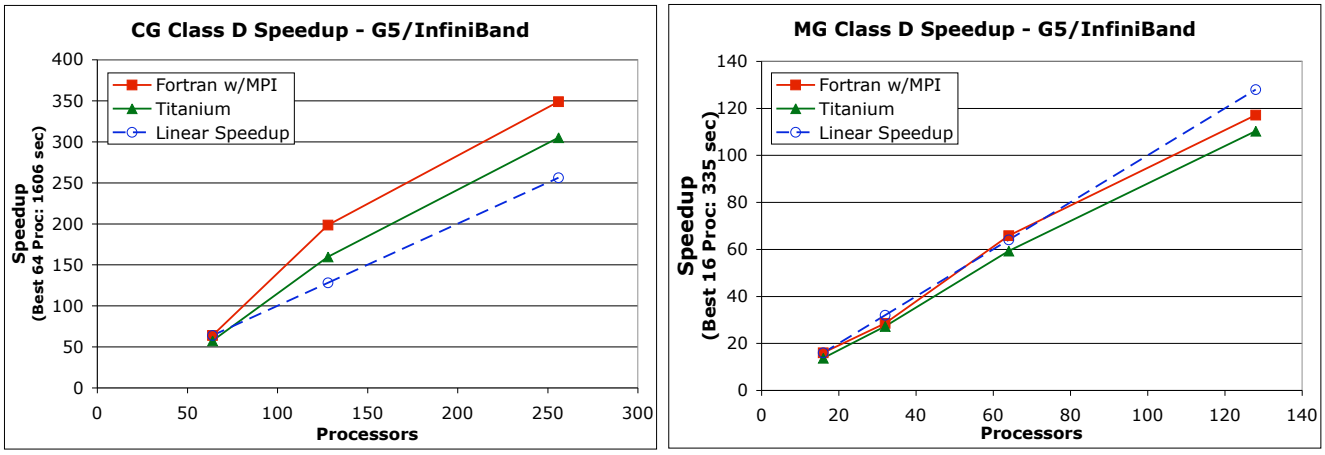


Figure 5: Comparison of Titanium and Fortran/MPI performance on a G5/Infiniband cluster for the NAS CG (left) and MG (right) benchmarks.

culation of pi (pi), and a Red-Black Gauss-Seidel relaxation code on a structured mesh (gsrb).

The Titanium implementation of the framework for finite difference discretization on block-structured adaptive meshes followed the design of Chombo, an AMR library written in C++ and Fortran with MPI [10]. Adaptive Mesh Refinement algorithms are challenging to implement due to the irregularity introduced by local mesh refinement. In addition to the regular operations one may find in ordinary finite difference calculations, this class of applications typically involves irregular (hierarchical, pointer-based) data structures, input-dependent computational load which easily requires domain-specific and dynamic load balancing, as well as fine-grained communications and irregular operations for updating grid boundaries in the adaptive mesh hierarchy. Chombo was designed to ameliorate the programming difficulties posed by AMR algorithms particularly for traditional Fortran programmers to whom AMR data structures are unfamiliar. Compared with Chombo, the Titanium implementation is much more compact [40]. Although no hand optimizations have been done to optimize the communications which are expressed at the logical level and therefore may result in fine-grained messages, we have shown that matching performance and scalability can be achieved by Titanium through automatic optimizations at the compiler and runtime level.

As part of the Berkeley UPC effort there have also been several in-house application efforts. The group developed versions of the NAS MG, CG, and FT benchmarks. Husbands developed a Delaunay Triangulation code in UPC, which uses a divide-and-conquer algorithm and a form of object-caching to improve locality. Welcome developed a multi-block code for computational fluid dynamics (CFD), specifically a gas dynamics code that partitions the space into variable sized blocks. Husbands also developed dense LU and sparse Cholesky factorization codes, both using a lightweight multithreading layer on top of UPC's threads. Iancu implemented a version of the Barnes-Hut algorithm (Barnes), which uses a tree-structure algorithm to solve the n-body problem. The UPC group also uses several benchmarks developed by others: Gups performs random read/modify/write accesses to a large distributed array; Mcop solves the matrix chain multiplication problem; Sobel performs edge detection with Sobel operators (3x3 filters) [13]; and Psearch performs parallel unbalanced tree search [30].

Because latency hiding through non-blocking communication is an important component of many of our optimized codes, we have been exploring alternative methods of managing outstanding communications and their dependent operations. One such method is multithreading [20, 22]. Here, processors are oversubscribed with threads and they context switch on long latency operations. In addition, the threads are free to perform any computation as soon as the data is ready.

We used this idea to implement a new dense LU factorization code, similar in functionality to the well-known High Performance Linpack (HPL) code. The standard Gaussian Elimination algorithm is decomposed into its major operations and these became the threads. In our work, we identified a number of challenges to performance and correctness. Chief among them were memory use and scheduling. Because multiple threads are running on a processor, their memory footprint must be controlled in order to prevent resource exhaustion. In addition, the execution schedule must be tuned so that parallel progress is made.

The code performed comparably to the MPI HPL implementation, obtaining over 2 TFlop/s on 512 Itanium 2 processors with a Quadrics network. It also was implemented in about 1/5th the size of the HPL code while only needing a small number of tuning parameters.

6. COMPILATION TECHNOLOGY

The NAS benchmark results shown above and by others in both UPC [13] and Titanium [12] have shown that the performance is comparable to MPI when the applications are hand-optimized to perform communication aggregation, pipelined communication, and overlap of communication with computation. When there are differences, the PGAS languages generally have faster communication, while there are still cases where computational kernels written in Fortran outperform C [40]. To address the serial performance issue, both languages allow calls to highly optimized serial code written in other languages. In these hand-optimized applications, no sophisticated compiler transfor-

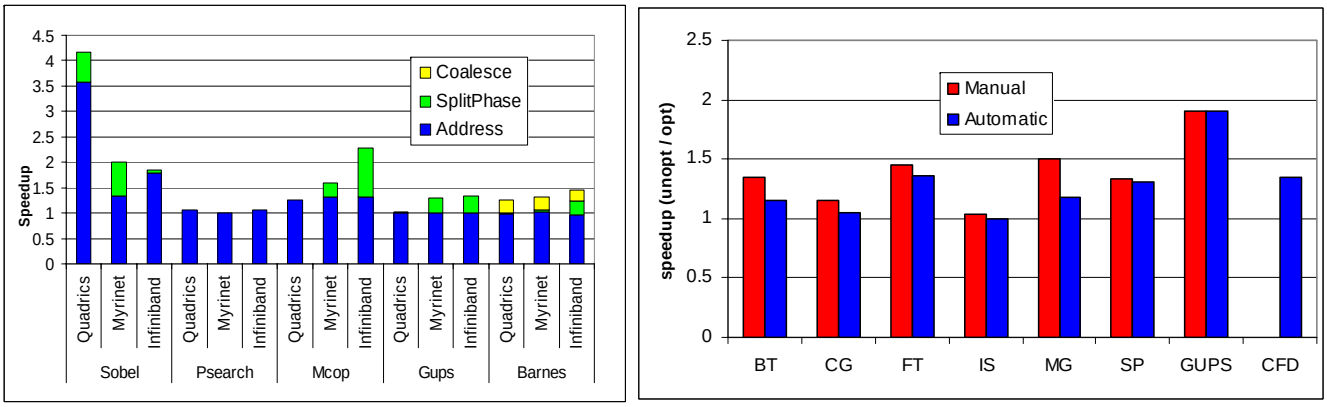


Figure 6: Speedups from optimizing UPC programs with fine-grained accesses (left, using 4 processors) and automatic overlap of bulk put/get operations (right, using 64 processors).

mations are necessary. The translation to C is relatively straightforward in UPC; it is somewhat more complicated in Titanium due to the lowering of Java style object-oriented code to C, but still quite tractable. The implementation challenges are mainly in the runtime libraries and include how to best use the communication primitives, how to map each of the variable types to an appropriate memory space on the machine, how to represent global pointers, and how to create and initialize the necessary threads. Our experience is therefore quite different from that of implementors of data-parallel languages like HPF [16] or automatic parallelizing compilers, where sophisticated transformations are a prerequisite to good performance.

The role of compiler analyses and optimizations in PGAS languages is to improve productivity by allowing programmers to write simpler code with fewer hand optimizations. Both the Titanium and Berkeley UPC teams have done extensive work in this area, demonstrating that the use of automatic optimizations can approach the performance of hand-optimized programs for some applications. The Berkeley UPC compiler has support for optimizing programs with fine-grained remote accesses, e.g., individual word reads and writes as well as automatic overlap of bulk operations. The left-hand side of Figure 6 shows the speedups obtained from optimizations on several UPC codes with fine-grained accesses. The optimizations include overlapping communication (split-phase), combining contiguous accesses (coalesce) and redundancy elimination in pointer manipulation (address). The right-hand side of Figure 6 shows the results of automatic overlap on several UPC applications involving bulk put and get operations, comparing the optimizations of hand-optimized code to automatic optimizations performed by the compiler and runtime. These optimizations rely on the UPC relaxed memory model, which roughly says that while each thread will always observe its own memory operations executing in order (local dependencies are preserved), it may observe another thread's memory operations happening out of order, unless the programmer explicitly labels the memory operations as *strict*. UPC has constructs for fine-grained control over the strict/relaxed semantics, although most UPC programs use only relaxed access combined with the UPC synchronization primitives (barriers and locks), which also act as strict consistency points. We have described the optimizations, required analyses, and details about the performance results elsewhere [8, 35].

Titanium inherits the strong typing of Java, which provides a wealth of static information useful in program analysis. In addition, the Titanium optimizer benefits from two properties of the parallel control flow of the program:

1. The barrier synchronizations in Titanium are required to be *textually aligned*, meaning that all threads must reach the same textual instance of a barrier before others can proceed (a requirement which is conservatively enforced by the compiler through static checking). This property divides programs into independent phases. Each textual instance of a barrier defines a phase, which includes all the expressions that can run after the barrier but before any other barrier. All threads must be in the same phase, which implies that no two phases can run concurrently.

2. To allow static enforcement of textual barrier alignment, Titanium introduces the concept of *single-qualification*—the `single` type qualifier guarantees that the qualified value is coherently replicated across all SPMD threads in the program. Since a single-valued expression must take the same sequence of observed values on all processes, all processes must take the same sequence of branches of a conditional guarded by such an expression. If such a conditional is only executed at most once in each phase, then the different branches cannot run concurrently.

The Titanium compiler performs analysis to determine what sections of code may operate concurrently and a novel multi-level pointer analysis to determine whether pointers marked as global are actually referring only to local data or to data within a set of processors that physically share memory. These analyses can be used to detect races, convert more expensive global pointers into local ones, and overlap communication with computation while preserving the illusion that statements execute in order, i.e., providing *sequential consistency*, which is equivalent to making all of the variable accesses strict in the UPC sense [21].

Pointers in Titanium can be declared as local or global, but are global by default. The default is chosen to make the shared accesses in Titanium closer to those of Java and

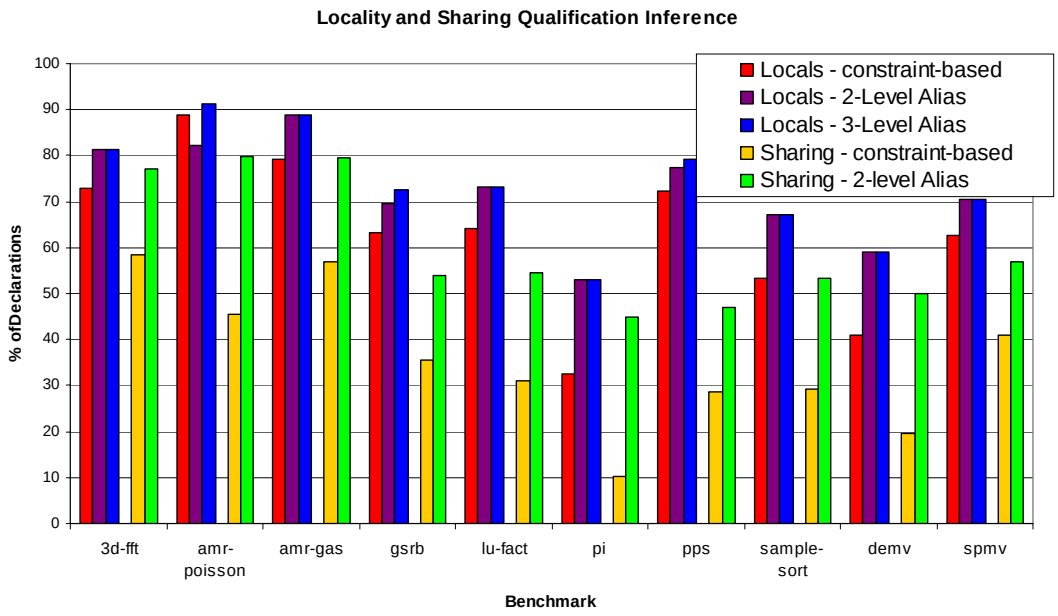


Figure 7: References in several Titanium benchmarks that are labeled as local (in the first three bars) or private (in the last two).

simplify porting of Java code into Titanium. This generality comes at a cost, however, since global pointers are less space- and time-efficient than local pointers. Manually inserting local qualifiers into user code can be tedious and error-prone. Furthermore, since we wanted to use Java's standard library implementations directly, we could not perform local qualification there.

The Titanium optimizer includes a *local qualification inference* (LQI) that automatically determines a conservative set of pointers that can be safely converted to local. Using a constraint-based inference, it automatically propagates locality information gleaned from allocation statements and programmer annotations through the application code [23]. Local qualification enables several important optimizations in the implementation of pointer representation, dereferencing, and array accesses. These optimizations reduce serial overheads associated with global pointers and enable more effective optimization and code-generation by the backend C compiler. For example, speedups of over 2x for NAS CG and 4x for NAS MG were obtained through LQI on an 8-processor cluster [40].

A more recent multi-level pointer analysis also supports conversion of global to local references and identification of private references, which are useful in detecting data races and in preserving sequential consistency while performing communication optimizations. Figure 7 shows the percentage of references in several applications that were automatically labeled as local or private (non-shared). The analysis is hierarchical, and can restrict pointers to being private within a shared memory node or within a thread. The graph compares the hierarchical pointer analysis to one based on constraints.

The Titanium compiler has support for the *inspector/ executor* framework [32] to optimize irregular remote accesses of the form `a[b[i]]` that appear in a loop. The array access pattern is computed in an initial *inspector* loop. All the required elements are prefetched into a local buffer. The *executor* loop then uses the prefetched elements for the actual computation. Different methods of communication can be used for prefetching data into a local buffer, including:

1. Pack method: only communicates the needed elements without duplicates. The needed elements are packed into a buffer before sending them to the processor that needs the data.
2. Bound method: a bounding box that contains the needed elements is retrieved.
3. Bulk method: the entire array is retrieved.

Figure 8 shows the performance of a Titanium implementation of sparse matrix-vector multiplication compared to that of a popular MPI library, Aztec. The choice of communication method depends on both the matrix structures and the machine characteristics, so the compiler uses a performance model of the machine parameters and determines which communication method is best based on the matrix structure that affects each thread pair. The Titanium code outperforms the Aztec code because of the lightweight communication layer and because the Titanium compiler will sometimes select different communication strategies between different processor pairs, a technique that would be tedious in application code.

7. CONCLUSIONS AND FUTURE WORK

The PGAS languages provide a single parallel programming abstraction that is useful for both shared-memory multiprocessors and for clusters. The languages give control over data layout (critical on clusters) while providing a global address space in which to build large shared structures. The languages do not require new compiler technology to allow for high performance; even without sophisticated optimizations, the performance of compiled UPC and Titanium is

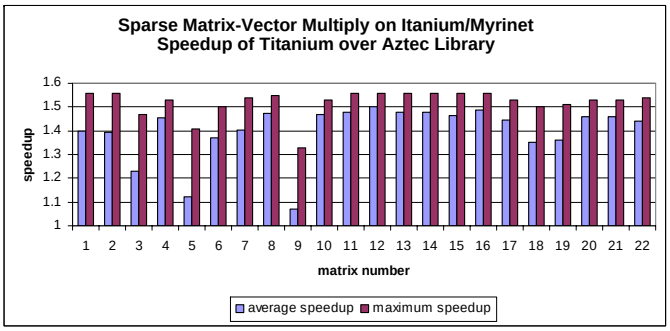


Figure 8: Performance of sparse matrix-vector multiplication in Titanium using dynamic optimizations compared to the Aztec library written in C. The Titanium data was collected over different processor configurations for 1-16 processors. The average and maximum speedups shown are drawn from this data.

competitive with that of MPI. In addition, there are some kernels (notably the FFT) on which the one-sided communication model of the PGAS languages yields a significant performance advantage. The one-sided model decouples synchronization (message receipt) from data transfer, and allows for lower latency for small (single-word) messages and higher bandwidth for mid-range (4KB) messages. We continue to explore these issues with language extensions and compiler optimizations, to aid users in maximizing the characteristics of their networks and only paying for features like per-message synchronization when it is needed.

Although sophisticated compilers are not a prerequisite to language success, both teams have invested in compiler analyses and optimizations as a way of getting better performance out of code that is written without significant hand-optimization. Automating overlap of communication with computation, aggregating small transfers into larger ones, converting global pointers to local ones, and use of dynamic optimizations for irregular codes are all optimizations that have been used in one or both of the compilers and have shown tremendous benefits for some of the benchmarks.

The SPMD model used in both languages results in efficient use of processors, since the application programmer is directly responsible for mapping the application-level parallelism to the physical machine resources. Furthermore, the one-to-one mapping between memory domains and threads makes the language runtime straightforward. If an application demands dynamic load balancing or relocation of data (as in the Delaunay triangulation), the application programmer is responsible for implementing these features on top of the fixed SPMD threads. Our experience with the matrix factorization codes in UPC shows that the PGAS model can also be used in an event-driven style that avoids the cost of global synchronization points and allows for better overlap of communication and computation. However, it also reveals subtle resource management issues since unconstrained use of dynamic threads can easily swamp a fixed set of physical resources. We are continuing to explore ways of mixing dynamic threads into the PGAS model while retaining the kind of control over layout and scheduling that has proven useful to obtaining high performance for many current PGAS applications.

8. REFERENCES

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. The Fortress language specification. Available from `http://research.sun.com/projects/plrg/`.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[3] G. T. Balls and P. Colella. A finite difference domain decomposition method using local corrections for the solution of poisson's equation. In *Journal of Computational Physics, Volume 180, Issue 1, pp. 25-53*, July 2002.

[4] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[5] The Berkeley UPC Compiler, 2002. `http://upc.lbl.gov`.

[6] D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.

[7] Chapel: The Cascade high productivity language. `http://chapel.cs.washington.edu/`.

[8] W. Chen, C. Iancu, and K. Yelick. Automatic nonblocking communication for partitioned global address space programs. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2007.

[9] W. Chen, A. Krishnamurthy, and K. Yelick. Polynomial-time algorithms for enforcing sequential consistency in SPMD programs with arrays. In *16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2003.

[10] The Chombo website. `http://seesar.lbl.gov/ANAG/software.html`.

[11] Cray C/C++ reference manual. `http://www.cray.com/craydoc/manuals/004-2179-003/html-004-2179-003/`.

[12] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study. In *The 18th International Workshop on Languages and Compilers for Parallel Computing*, October 2005.

[13] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: A NPB experimental study. In *Supercomputing2002 (SC2002)*, November 2002.

[14] GASNet home page. `http://gasnet.cs.berkeley.edu`.

[15] Hewlett-Packard Company. *HP UPC Version 2.0 for Tru64 UNIX*. `http://h30097.www3.hp.com/upc/`.

[16] High Performance Fortran Forum. High Performance Fortran Language Specification. http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20, Jan. 1997.

[17] P. Hilfinger, D. Bonachea, D. Gay, S. Graham,

B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. Tech Report UCB/CSD-01-1163, U.C. Berkeley, November 2001.

[18] C. Iancu, P. Husbands, and W. Chen. Message strip mining heuristics for high speed networks. In *Proc. 6th International Meeting on High Performance Computing for Computational Science (VECPAR)*, 2004.

[19] Intrepid Technology, Inc. *GCC/UPC Compiler*. `http://www.intrepid.com/upc/`.

[20] L. V. Kale and S. Krishnan. CHARM++ : A portable concurrent object oriented system based on C++. *ACM SIGPLAN Notes*, 28(10):91–108, 1993.

[21] A. Kamil, J. Su, and K. Yelick. Making sequential consistency practical in Titanium. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, November 2005.

[22] C. Leiserson and R. Blumofe. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[23] B. Liblit and A. Aiken. Type systems for distributed data structures. In *the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2000.

[24] P. McCorquodale and P. Colella. Implementation of a multilevel algorithm for gas dynamics in a high-performance Java dialect. In *International Parallel Computational Fluid Dynamics Conference (CFD'99)*, 1999.

[25] S. Merchant. Analysis of a contractile torus simulation in Titanium. Masters Report, Computer Science Division, University of California Berkeley, August 2003.

[26] MuPC portable UPC runtime system. `http://www.upc.mtu.edu/`.

[27] R. Numrich and J. Reid. Co-array fortran for parallel programming. In *ACM Fortran Forum 17, 2, 1-31.*, 1998.

[28] Open64 compiler tools. `http://open64.sourceforge.net`.

[29] G. Pike, L. Semenzato, P. Colella, and P. N. Hilfinger. Parallel 3D adaptive mesh refinement in Titanium. In *9th SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, Texas*, March 1999.

[30] J. Prins, J. Huan, W. Pugh, et al. UPC implementation of an unbalanced tree search benchmark. Technical Report 03-034, Department of Computer Science, University of North Carolina, 2003.

[31] L. Snyder. *The ZPL Programmer's Guide*. MIT Press, 1999.

[32] J. Su and K. Yelick. Automatic support for irregular computations in a high-level language. In *19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[33] Titanium home page. `http://titanium.cs.berkeley.edu`.

[34] UPC language specifications, v1.2. Technical Report LBNL-59208, Berkeley National Lab, 2005.

[35] W.Chen, C. Iancu, and K. Yelick. Communication Optimizations for Fine-Grained UPC Applications. In *14th International Conference on Parallel Architectures and Compilation TechniquesPACT*, 2005.

[36] T. Wen and P. Colella. Adaptive Mesh Refinement in Titanium. In *19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[37] T. Wen, P. Colella, J. Su, and K. Yelick. An adaptive mesh refinement benchmark for modern parallel programming languages. Submitted to Supercomputing 2007.

[38] The X10 programming language. `http://www.research.ibm.com/x10`.

[39] S. M. Yau. Experiences in using Titanium for simulation of immersed boundary biological systems. Masters Report, Computer Science Division, University of California Berkeley, May 2002.

[40] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, and T. Wen. Parallel languages and compilers: Perspective from the Titanium experience. *The International Journal of High Performance Computing Applications*, 21(2), 2007.