Evaluating the Impact of Programming Language Features on the Performance of Parallel Applications on Cluster Architectures

Konstantin Berlin[1], Jun Huan[2], Mary Jacob[3], Garima Kochhar[3], Jan Prins[2], Bill Pugh[1], P. Sadayappan[3], Jaime Spacco[1], and Chau-Wen Tseng[1]

[1] Department of Computer Science,
University of Maryland, College Park, MD 20742

[2] Department of Computer Science,
University of North Carolina, Chapel Hill, NC 27599

[3] Department of Computer and Information Science,
Ohio State University, Columbus, OH 43210

Abstract. We evaluate the impact of programming language features on the performance of parallel applications on modern parallel architectures, particularly for the demanding case of sparse integer codes. We compare a number of programming languages (Pthreads, OpenMP, MPI, UPC) on both shared and distributed-memory architectures. We find that language features can make parallel programs easier to write, but cannot hide the underlying communication costs for the target parallel architecture. Powerful compiler analysis and optimization can help reduce software overhead, but features such as fine-grain remote accesses are inherently expensive on clusters. To avoid large reductions in performance, language features must avoid degrading the performance of local computations.

1 Introduction

Parallel computing can potentially provide huge amounts of computation power for solving important problems in science and engineering. However, the difficulty of writing parallel programs poses a major barrier to exploiting the power of parallel architectures. Programming is especially difficult for applications with irregular, fine-grain memory access patterns, since current parallel programming languages, tools, and architectures are evolving in directions less suited for these codes. Three vital goals are in conflict when choosing a parallel programming paradigm for clusters of shared-memory multiprocessors:

- Exploitation of maximum machine performance on a particular platform
- Portability of code and performance across various high performance computing platforms
- Programmability: easy creation of correct, reliable and efficient programs

Parallel programming languages are designed by making different tradeoffs, depending on assumptions of the underlying compiler, runtime system, hardware support, target application characteristics, and acceptable user effort.

For embarrassingly parallel applications with coarse-grain communication, the choice of a parallel programming language is less important since almost all languages can achieve good performance with low programmer effort. Unfortunately, no current parallel programming paradigm is satisfactory for more complex applications with fine-grain parallelism and irregular remote accesses. MPI is the most portable and achieves the best performance on distributed-memory machines for most codes, but is difficult to program and is inefficient for applications with many irregular fine-grained accesses. OpenMP and Pthreads are simple and efficient on shared-memory nodes, but do not work well (if at all) on clusters. HPF is portable but limited in its flexibility and applicability. Java is popular but does not yet have widely adopted libraries/APIs for efficient parallel execution on clusters.

A promising approach for easing the task of writing codes with fine-grain parallel accesses is to use programming languages that provide flexible remote accesses and support for a shared address space, such as UPC and Co-Array Fortran. These hybrid languages simplify code development because programmers can rely on language support for fine-grain remote accesses to get a working version quickly, before selectively putting effort into modifying a small subset of the code for enhanced performance. In comparison, programming paradigms such as MPI require explicit communications to be inserted throughout the code for correctness.

A problem with this hybrid approach is the architectural trend towards building high-end supercomputers from clusters of PCs or shared-memory multiprocessors (SMPs) using commodity parts, since this approach yields systems with expensive, high latency inter-processor communication. As a result users are gravitating towards parallel programming paradigms such as MPI that can efficiently support coarse-grain bulk communications. Parallel programming paradigms such as UPC that rely on fine-grained remote accesses may find it difficult to achieve good performance on clusters, because the underlying architecture does not efficiently support such operations.

Our goal in this paper is to evaluate and quantify the performance of parallel language features based on experimental evaluations of a number of challenging parallel applications, particularly those requiring fine-grain remote accesses. We identify programming language features that can reduce programmer effort and quantify the overhead encountered when using such features. We attempt to determine the feasibility of using a hybrid fine and coarse-grain parallel programming model on cluster architectures. We pay special attention to the performance of UPC because it is the first widely available commercially supported high-level parallel programming language that provides flexible non-local accesses for both shared and distributed memory paradigms. We also attempt to place our evaluation in the context of ongoing trends in parallel architectures and applications. More specifically, the contributions of this paper include:

1. Experimental evaluation of language features for challenging irregular parallel applications
2. Observations on programmability and performance for Pthreads, OpenMP, MPI, and UPC
3. Suggestions for achieving both programmability and good performance in the future
4. Predictions on impact of architectural developments on performance of parallel language features

While our findings that fine-grained parallel applications perform poorly on cluster architectures is not surprising, our study quantifies the performance penalty for interesting programming languages using several challenging irregular benchmarks.

In the remainder of the paper, we explain our choice of evaluation parameters (applications, parallel languages) and present our experimental results. We present our observations on programming language features and their impact on performance, followed by a number of suggestions for their usage in developing parallel applications. We conclude with a discussion of the impact of architecture trends and comparison with related work.

2 Applications

Many scientific applications have very regular memory access patterns and can be easily parallelized and implemented efficiently for a large number of parallel architectures. We chose for our evaluation three application classes that are more complex and represent challenging test cases for parallel programming paradigms. The three types of parallel applications are:

Irregular table update Many parallel database operations can be viewed as making irregular parallel accesses to a large distributed table of values. If the accesses perform associative reduction operations (e.g., summation), the application is similar to a large histogram and may be implemented using a coarse-grain bucket algorithm. Accesses may also perform arbitrary read-modify-write operations, in which case fine-grain algorithms are necessary. The amount of computation in table updates is static and may be distributed evenly at compile time. Table update has potentially very high communication requirements.

Irregular dynamic accesses A second class of challenging parallel applications perform irregular parallel accesses to sparse data structures. The application may allow a limited amount of coarse-grained accesses. The amount of computation is static and may be distributed evenly at compile time, and has very high communication requirements.

Integer sort Large in-memory sorting is a third parallel application class that is surprisingly difficult to perform efficiently on distributed-memory parallel architectures. Many parallel implementations are possible, including both coarse and fine-grained algorithms. Sorting has high communication requirements.

All three types of benchmarks are characterized by irregular memory access to large data structures. Depending on the benchmark, both coarse and fine-grained remote accesses may be necessary.

3 Programming Paradigms

Broadly speaking, parallel paradigms can be classified as shared-memory with explicit threads (Pthreads, Java threads), shared-memory with task/data parallelism (OpenMP, HPF), distributed memory with explicit communication (MPI, SHMEM, Global Arrays), or distributed-memory with special global accesses (Co-Array Fortran, UPC). We describe paradigms used in our study in more detail.

Pthreads (POSIX threads) is a shared-memory programming model where parallelism takes the form of parallel function invocations [LB98]. A parallel function body is executed in parallel by many threads, which can all access shared global data. Pthreads is the underlying implementation of parallelism for many programming paradigms. Java is a general purpose programming language that supports parallelism in the form of threads [OW97]. Parallel Java programs on SMPs resemble Pthreads programs. Pthreads and Java are available only on SMPs.

OpenMP is a shared-memory programming model where parallelism takes the form of parallel directives for loops and functions [CMD00]. OpenMP directives specify loops whose iterations should be executed in parallel, as well as functions that may be invoked in parallel. Additional directives specify data that should be shared or private to each thread. Compilers translate OpenMP programs into code that resembles Pthreads programs, where parallel loop bodies are made into parallel functions. OpenMP is an industry standard and is supported in many languages and platforms. OpenMP is currently available only on SMPs.

MPI (Message Passing Interface) is a distributed-memory programming model where threads explicitly communicate using functions in the MPI run-time library to send and receive messages [GLS94]. It also includes a large selection of efficient collective communication routines. MPI is widely available (virtually every parallel platform) and well tuned for performance. Despite the programming effort required, MPI is the current programming paradigm of choice for its portability and performance.

UPC (Unified Parallel C) is a shared-memory programming model based on a version of C extended with global pointers and data distribution declarations for shared data [CDC99]. Accesses via global pointers are translated into inter-processor communication by the UPC compiler. A distinguishing feature of UPC is that global pointers may be cast into local pointers for efficient local access. Explicit one-way communication similar to SHMEM [**?**] is also supported in the UPC run-time library via routines such as upc_memput() and upc_memget(). It is the compiler's responsibility to translate memory addresses and insert inter-processor communication. UPC is the first commercially supported parallel paradigm that supports flexible remote accesses to a shared memory abstraction.

4 Performance Evaluation

We believe performance is a key factor (if not the key factor) determining the success of parallel programming paradigms. To gain insight into the factors underlying performance, we performed an experimental performance evaluation of a number of programming paradigms on the following parallel platforms.

Compaq AlphaServer SC. A 64-node cluster located at ORNL. Each node is an SMP with 2GB of memory, four ES-40 processors, and a single Quadrics network adapter. The nodes run AlphaServer 2.0 OS, the MPI implementation is built on the native Quadrics libraries.

Sun SunFire 6800. A 24-processor Sun shared-memory multiprocessor located at the University of Maryland, with UltraSparc III processors, 24GB memory, and crossbar interconnect running SunOS 5.8.

4.1 Table Update

TableUpdate performs irregular updates on a large distributed hash table. Updates are commutative and may be reordered. Several different versions of TableUpdate are used:

– *MPI.* Coarse-grain algorithm uses buckets to store updates to data on other processors. All buckets are synchronously exchanged between processors once buckets are filled. Upon receiving buckets, updates in bucket are applied to the local portion of the table.
– *UPC.* Fine-grained algorithm uses global pointers to update non-local table elements.
– *UPC (bucket).* Coarse-grain algorithm also uses bucketized algorithm as in MPI code. One-way explicit communication used to transfer buckets between processors.
– *C with Pthreads.* Shared-memory code uses parallel function calls to update table elements. All threads directly access table as shared array.
– *C with OpenMP.* Shared-memory code parallelizes loops computing table elements using OpenMP annotations.

– *Java.* Shared-memory code uses Java threads to update shared global table.

Figure 1 presents the performance of TableUpdate for a table of size 2^{22} on a Compaq AlphaServer SC for MPI, UPC, and UPC (bucket). Performance is measured in number of table updates per millisecond per processor, and is presented using log scale. Results show that MPI greatly outperforms UPC, though UPC using a coarse-grain bucket algorithm can approach the performance of MPI. UPC suffers significant performance degradations when using fine-grain access patterns because of software and hardware overhead in making point-wise remote accesses.

We next examine TableUpdate performance on a Sun SunFire SMP. Results in Figure 2 show that Java, C with Pthreads, and C with OpenMP implementations of TableUpdate achieve comparable performance, though Java performance is slightly higher (possibly because it is better tuned for performance by the vendor). The SUN UPC compiler has significantly poorer performance because of software overhead in translating point-wise accesses to shared data.

4.2 Conjugate Gradient

The conjugate gradient benchmark (NAS CG benchmark) finds the principal eigenvalue of a sparse $n \times n$ real matrix A with random pattern of kn nonzeros using the inverse power method [BBB94]. This involves solving a linear system of the form $Ap = z$ for different vectors z. The solver uses the conjugate gradient method and repeatedly calculates the sparse matrix-vector product $w = Av$, where v, w are dense vectors of length n. This benchmark is widely used and stresses memory and communication performance. We evaluated the following versions of CG:

– *MPI.* This Fortran 77 version was taken from the NAS 2.3 suite, and uses explicit MPI communication operations. The implementation uses a (block, block) distribution of A, and replicates the appropriate section of v for the dot product with the corresponding section of A. The total size of the implementation is 1800 lines.
– *OpenMP.* This is a shared-memory implementation in C with OpenMP directives, derived from the NAS 2.3 serial code by the RWC in Japan, and has total size of 900 lines. This implementation uses a static partition across processors of the row-loop of the matrix-vector product. A long-lived parallel region is used to reduce overheads between successive sparse-matrix vector products. OpenMP work distribution directives are inserted for initializations, sparse matrix-vector product, and dot products in the algorithm.

Fig. 1: Table Update (AlphaServer, 2^22 table)

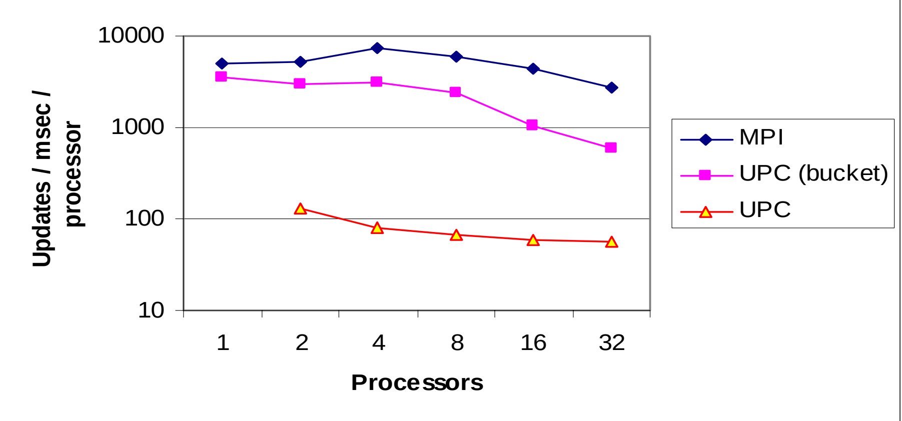


Fig. 2: Table Update (Sun SMP, 2^25 table)

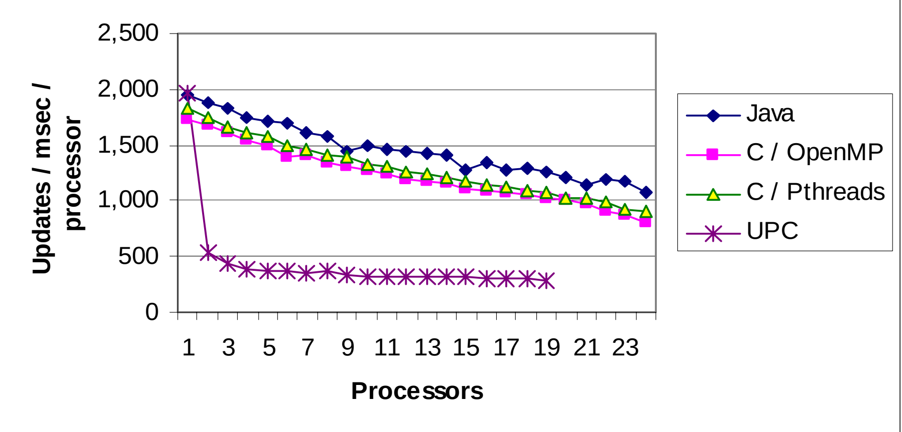
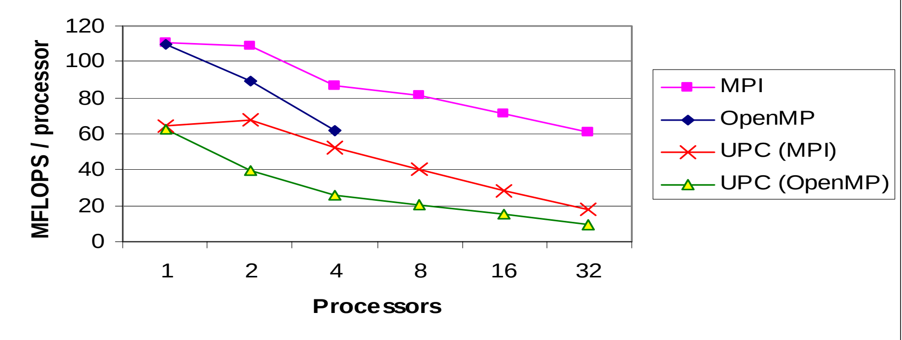


Fig 3: Conjugate Gradient (AlphaServer, Class B)



- *UPC (OpenMP).* This UPC implementation was derived from the OpenMP shared-memory version. About 1/3 was rewritten from OpenMP, and 1/4 was added new. The total size of this version is 1300 lines. It distributes the matrix A using a block-cyclic distribution with a large block size. This is the best distribution for this problem that can be expressed directly in UPC without explicitly partitioning the matrix A. Work is partitioned between processors in the sparse vector-matrix product according to the portions of A held by each processor. The vector v is replicated to reduce communication; the default strategy of distributing the shared vector leads to run times that are two orders of magnitude larger due to the repeated fine-grain random accesses to v in the sparse matrix-vector product.
- *UPC (MPI).* This UPC implementation more closely follows the MPI algorithm. It uses an explicit (blocked,*) distribution of A and replicates the vector v. Coarse-grain data movement (e.g., upc_memget(), upc_memput()) is used to replicate the result w. The total size of this version is 1600 lines.

Figure 3 presents our results for a class B problem size for CG on the AlphaServer SC. Results are reported in MFLOPS per processor. The total number of FLOPS required is defined by the problem size. OpenMP results are only available up to the 4 processors on each node and scale relatively poorly due to the replication of v into the processor caches through misses on v in random order. MPI outperforms both versions of UPC, though the UPC (MPI) implementation is closer in performance.

The sequential performance of the UPC implementations is 50-60% of the single processor MPI and OpenMP performance. The MPI implementation achieves a speedup of 10.4 with 16 processors, and a speedup of 17.6 with 32 processors. The UPC (OpenMP) speedup is 4.0 with 16 processors, and 5.0 with 32 processors, hence performs at only 28% of the MPI implementation at 32 processors. The UPC (MPI) speedup is better at 7.0 with 16 processors, and 9.1 with 32 processors, hence performs at 52% of the MPI implementation at 32 processors.

The performance of CG is heavily dependent on memory system performance. For comparison, a vectorized implementation of the CG benchmark achieves about 1,500 MFLOPS on a single processor of an NEC SX-6, and about 1,100 MFLOPS per processor using all eight processors of an SX-6 node.

4.3 Integer Sort

Integer sort performs a parallel radix sort of a large collection of integer data. We timed MPI and UPC implementations on an AlphaServer SC. Both implementations used coarse-grain parallel algorithms employing bulk explicit messages, since a fine-grain UPC implementation was found to be intolerably inefficient. A 128K key input data size is used. Performance is reported as efficiency. Results in Figure 4 show that MPI outperforms UPC slightly, with the difference increasing for larger numbers of processors.

4.4 UPC Microbenchmark

Our experimental results for entire applications showed that fine-grain algorithms were exceedingly inefficient for cluster architectures. We repeated our experiments using the Berkeley UPC compiler [CB+03] on the AMD Athlon PC cluster at Ohio, and UPC performance was only slightly improved relative to MPI.

The problem, we believe, is caused by the overhead of fine-grained accesses in UPC. UPC provides global shared pointers that can easily access non-local data, providing a convenient shared-memory abstraction for parallel programming. Though a shared data element can be accessed in a completely transparent fashion by any process executing on any processor, the overhead of direct point-wise access can be quite significant. To quantify both the hardware and software overheads in greater detail, we used UPC microbenchmarks to evaluate performance on a wide range of parallel architectures.

- Compaq AlphaServer SC system (Falcon) at Oak Ridge National Laboratory, running Version 1.7 of the Compaq UPC compiler.
- Single node AlphaServer Marvel at University of Florida, running Version 2.1 of the Compaq UPC compiler.[4].
- AMD Athlon Cluster (64 dual-processor nodes) with Myrinet interconnect at the Ohio Supercomputer Center, running the Berkeley UPC compiler.
- SUN SunFire 6800 system (24-nodes) at the University of Maryland, running the Sun UPC compiler.
- Cray T3E system at Michigan Tech University, running the original UPC compiler.
- SGI Origin 2000 at University of North Carolina, running the Intrepid UPC compiler.

We measured the cost of direct point-wise shared data access costs, using both private and shared pointers. Figure 5 shows the per-word access cost using a read-modify-write (increment-by-one) operation on floating-point doubles, for various modes of access:

- Private: local shared data that is accessed as private data via casting UPC pointer to private.
- Shared-local: local shared data that accessed directly as using a UPC shared pointer
- Shared-same-node: non-local shared data that is local to another process on the same SMP node.
- Shared-remote: non-local shared data that is on a different node.

[4] The authors would like to thank Dr. Alan George at the University of Florida-Gainesville for providing access to this machine

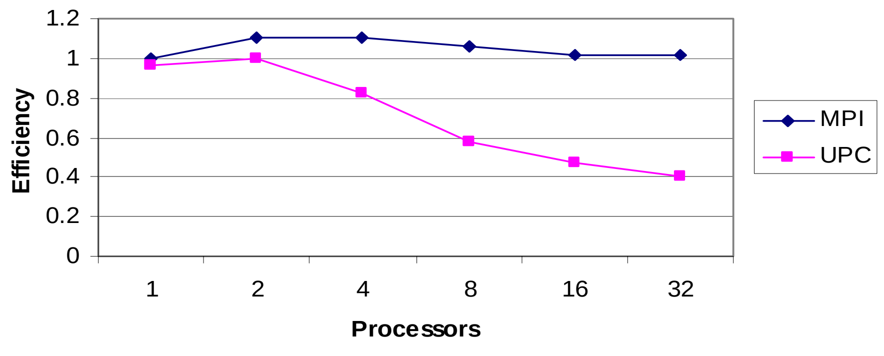


Fig. 4: Integer Sort (AlphaServer, 128K keys)

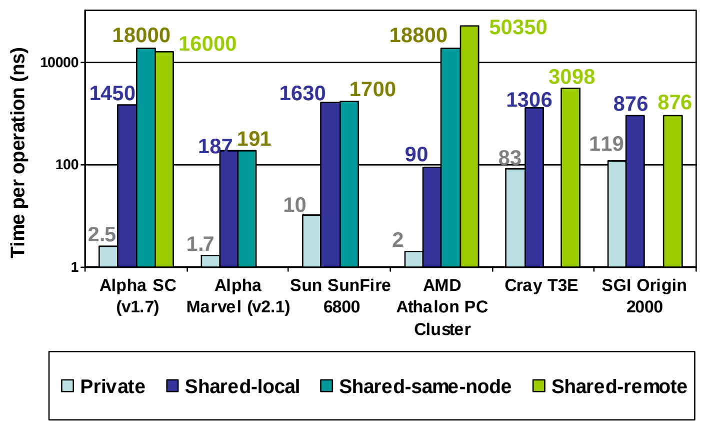


Fig. 5: UPC Point-wise Data Access Costs (read-modify-write double)

It can be observed that on all systems, there is a significant difference in the access time for private data and shared-local data, even though there is no data movement involved with the latter. The difference represents the overhead of translating a shared UPC reference into a node-address pair. This overhead was over 500 times a local memory access cost on the Compaq AlphaServer with the earlier version (v1.7) of the Compaq UPC compiler. Compiler enhancements have reduced the overhead in later versions (v2.1) of the compiler to around 100 times the private data access cost.

Another area where compiler optimization can reduce software overhead for memory access costs was in accessing non-local data located on the same node (belonging to another thread on the same node). More powerful compiler optimizations can use more efficient local memory accesses in this situation, as demonstrated by the newer Compaq UPC compiler (v2.1). Nonetheless, even with both optimizations (for local shared data and same-node shared data), memory access costs are still two orders of magnitude higher than access to private memory for UPC on the AlphaServer Marvel system. Fine-grain non-local accesses must therefore be used sparingly if at all in performance critical sections of a parallel UPC program.

4.5 Evaluation Summary

Summarizing our results, we find on SMPs threads-based paradigms are closest to the underlying hardware and provide the best performance. On clusters, paradigms with explicit communication have the lowest overhead and achieve the best performance. UPC programs can achieve good performance when written in a similar coarse-grain style using bulk communication routines, otherwise performance can be extremely poor.

5 Language Features

Based on our experimental evaluation, we present some observations and suggestions with high-level language features. A number of parallel programming languages provide language features for providing the illusion of shared memory. The UPC programming model provides access to cyclically distributed shared arrays through global pointers, though when accessing only local portions of a shared array, global pointers may be cast back into local pointers for greater efficiency. In addition, the UPC run-time library also provides one-way, coarse- grained explicit communication primitives through functions such as upc_memget() and upc_memput(). We make the following observations about these language features:

A global shared memory programming model is easy to use. At the core of the UPC programming model is the ability to easily access non-local data in a parallel program simply through global pointers. Programmers need only specify data that is to be distributed across processors, and reference them through special global pointers. The fine-grained UPC programming model is very simple and easy to use. The resulting code is cleaner and more maintainable than paradigms such as MPI that require explicit communication in the program.

User level shared memory is not a good reflection of clusters. While the programming model may allow easy fine-grain access to non-local data, this is not supported by the underlying hardware architecture. The interconnect between nodes of a cluster typically provides high bandwidth but also long latencies, making aggregate coarse-grained communication much more efficient than many fine-grained remote accesses. This problem will only worsen as future parallel architectures continue to evolve towards clusters of SMPs. In comparison, the coarse-grain one-way communication primitives in many languages more accurately reflect the actual communication mechanisms supported by the hardware.

A shared-memory programming model can encourage poor performance on clusters. Because the fine-grained shared-memory programming model is so seductive, one can argue that it actually leads to poor performance by encouraging programmers to write fine-grain codes that execute poorly on clusters. Programmers can code around this problem, but usually only at the cost of complicating the programming model or changing their coarse-grain algorithm.

We are dubious that compiler techniques will solve this problem. Given the lack of hardware support for efficient fine-grain communication on clusters, we believe programmers will need to develop parallel algorithms with coarse-grain block data movement to achieve good performance. Compilers can remove some of the inefficiencies of fine-grain communication, but cannot robustly transform fine-grain parallel algorithms into efficient block parallel codes for clusters.

The (hybrid) programming model can combine fine-grain and coarse-grain accesses. One advantage of the UPC programming model is that it allows integration of fine-grain remote accesses with global pointers and coarse-grain explicit communication using library routines such as upc_memput() and upc_memget(). As we stated previously, a hybrid programming paradigm such as UPC can ease the development and maintenance of parallel codes. Most of the program may be written cleanly using global pointers, inserting explicit coarse-grain communication only for performance critical sections. Our experimental evaluation shows that when done well, the resulting codes can achieve performance close to MPI on clusters. However, programmers must be extremely careful because the cost of using global pointers for remote accesses is so high. Developing coarse-grain parallel algorithms for performance-critical sections of the program may also require extensive modifications to the algorithms and data structures used in the code.

Programming language features must avoid degrading local computations. Many computations in parallel programs can be performed on purely local or previously prefetched remote data. Parallel programming languages should be designed so that these local computations can be compiled (and optimized) by the native sequential compiler. Otherwise performance can degrade, sometimes significantly. A great deal of the success of MPI can be attributed to following this rule, since all computations depend only on local data after calls to MPI communications functions return. In comparison, UPC require user-inserted explicit copies of remote global data to local buffers (or casting global pointers to local if shared data is alrady local) to avoid excessive overhead. Simply accessing global shared data is too expensive, even though the global data may be completely located locally. For instance, accessing local data using a global pointer in UPC can result in over 100 times slowdown. is actually local.

5.1 Advice on choosing parallel paradigms

We summarize our observations on the parallel language features as follows. Even though a language like UPC may support a fine-grain programming model, it can achieve respectable performance on clusters only if fine-grain remote accesses are used sparingly. Coarse-grain parallel algorithms and bulk communication are still essential for achieving good performance. For fine-grain parallel algorithms, even though language and compiler support can improve performance compared to naive implementations, absolute performance on clusters is likely so poor that differences will be insignificant.

Based on our experiences, we believe that the prime factor in choosing parallel paradigms is the nature of the algorithm. For coarse-grain parallel algorithms on clusters, many choices are possible. For peak performance, explicit message passing paradigms such as MPI and SHMEM will likely provide the best performance. If program development time is an issue, choosing a hybrid UPC implementation and selectively using bulk and collective communication such as upc_memget() and upc_memput() routines in computationally intensive portions of the program can be useful. Programming effort can also be reduced by exploiting existing libraries where possible. For fine-grain parallel algorithms, there are fewer options. Implementations on clusters using only fine-grain language features are likely to be extremely slow. If the data size is small, these codes may be executed on SMPs. Otherwise coarse-grain alternatives should be developed if possible.

On the Cray T3E (the original platform for UPC), UPC appears to be an unqualified success and one of the best possible choices for a programming language/paradigm. However, the suitability of fine-grain programming languages for cluster environments, with higher latencies and message overheads, is unclear. Obtaining good performance from a shared memory in a cluster environment requires programming in specific and sometimes convoluted styles, discarding many of the easy of use features of the language. Advancing compiler technology can help in some cases, but still results in an environment with a complicated and opaque performance model. The ability of a programmer to write a complicated fine-grain parallel program and have confidence that it will achieve good performance across a range of platforms still seems a distant dream.

6 Impact of Trends in Parallel Architectures

We also wish to evaluate parallel language features in the context of ongoing architectural developments. Here we examine developments and trends in parallel computer architectures and their impact on parallel programming paradigms.

Faster interconnects. High-speed cluster interconnects continue to improve in bandwidth and latency. Both proprietary interconnects (e.g., Quadrics Elan used in Compaq AlphaServer) and systems for connecting commodity processors (e.g., SCI, Dolphin, Myrinet, VIA, InfiniBand) are improving in performance. Such interconnects also offer better support for shared memory, small messages, and one-sided communication, and thus may improve fine-grain communication performance. On the other hand, while the absolute performance of inter-processor communication is steadily improving, the cost of communication relative to computation continues to increase due to ever faster nodes and processors. We see no technological developments that will reduce or even slow this gap in the near future.

Larger memories. Although memory latency is increasing relative to processor speeds, memory size is increasing due to greater chip densities. As memory prices continue to drop, it is becoming possible to construct parallel systems with much larger amounts of memory than in the past. Cluster and MPP systems can now be built with several Terabytes of memory, and even SMPs can be purchased with 256 Gigabytes or more of memory. Continuing increases in SMP memory size may allow them to run (commercial) applications previously limited to MPPs and clusters, reducing the demand and vendor support for more complicated programming models.

Processor/memory integration. Processor-in-memory (PIM) designs can potentially offer enormous improvements for specific problems by providing efficient parallel operations on data. However, they do not obviate the need for interprocessor communication. Hence the general utility of such designs will still depend on communication performance. Specific aspects of PIM designs may start to appear in memory controllers for conventional systems, but are probably still a few years away. In general, PIM- like systems will likely increase the cost of non-local memory accesses relative to computation, increasing rather than reducing the difficulty of efficient parallel programming.

Multithreading. Microprocessor design seems to be heading towards greater support for multithreading to tolerate increasing memory latencies. Increasing levels of task-level multithreading will start to make even single processor nodes on MPP systems resemble SMPs, and likely accelerate the shift into hybrid programming models suitable for cluster architectures.

The good news is that as parallel architectures improve, programs will be able to process larger irregular problems more quickly. The bad news is that the efficiency of parallel programs will continue to decrease.

7 Related Work

Obviously there is a tremendous amount of research on parallel language design and benchmarking. The most relevant to this paper is the recent work analyzing the performance of UPC. El-Ghazawi et al. have been developing and benchmarking UPC codes [CY+03,EC01,EC02] and have discovered performance can be respectable, if a coarse-grain programming style is adapted. Yelick et al. have actually developed their own UPC translator/compiler [CB+03]. Their experiments show similar results, that fine-grain accesses are significantly more expensive, and performance improves if the compiler can aggregate remote accesses to reduce costs. In comparison, we study a wider range of parallel languages on a slightly different set of applications. Pugh and Spacco use similar benchmarks to evaluate MPJava, a method for developing high-performance parallel computations in Java [PS03].

8 Conclusions

In this paper, we evaluated features from a number of parallel programming languages (MPI, UPC, OpenMP, Java, C/Pthreads) for their performance and ease of use. We find that languages such as UPC that support a shared memory and flexible non-local accesses can reduce the difficulty of parallel programming. Unfortunately, parallel applications requiring fine-grain accesses still achieve poor performance on clusters because the amount of inherent software and hardware overhead, regardless of the programming paradigm or language feature used. Language support for fine-grain non-local accesses can still prove useful, by reducing the difficulty of parallel programming. Decent performance is achievable by using coarse-grain bulk communication in performance-critical sections of the code.

References

[BBB94] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks," Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[CB+03] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler, Proceedings of the 17th Annual International Conference on Supercomputing (ICS'03), June 2003.

[CDC99] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and Language Specification," Center for Computing Sciences Technical Report CCS-TR-99-157, May 1999.

[CMD00] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, "Parallel Programming in OpenMP," Morgan Kaufmann Publishers, 2000.

[CY+03] F. Cantonnet, Y. Yao, S. Annareddy, A. Mohamed, T. El-Ghazawi. Performance Monitoring and Evaluation of a UPC Implementation on a NUMA Architecture, Proceedings of the International Conference on Parallel and Distributed Parallel Systems (IPDPS'03), April 2003.

[EC01] T. El-Ghazawi and S. Chauvin, UPC Benchmarking Issues, Proceedings of the International Conference on Parallel Processing (ICPP'01), September 2001.

[EC02] T. El-Ghazawi and F. Cantonnet. UPC Performance and Potential: A NPB Experimental Study, Proceedings of SC2002, Baltimore, November 2002.

[GLS94] W. Gropp E. Lusk, and A. Skjellum, "Using MPI: Portable Parallel Programming with the Message-Passing Interface," MIT Press, Cambridge, MA, 1994.

[LB98] B. Lewis and D. J. Berg, "Multithreaded Programming with Pthreads," Prentice Hall, 1998.

[OW97] S. Oaks and H. Wong, "Java Threads. Nutshell Handbook," O'Reilly & Associates, Inc., 1997.

[PS03] B. Pugh and J. Spacco, "MPJava: High-Performance Message Passing in Java using Java.nio," Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC'03), College Station, TX, October 2003.