Performance Monitoring and Evaluation of a UPC Implementation on a NUMA Architecture

François Cantonnet, Yiyi Yao, Smita Annareddy, Ahmed S. Mohamed*, Tarek A. El-Ghazawi
Department of Electrical and Computer Engineering, The George Washington University
Washington, DC 20052
{ fcantonn, yyy, asmita, sameh, tarek }@seas.gwu.edu

Abstract

UPC is an explicit parallel extension of ANSI C, which has been gaining rising attention from vendors and users. In this work, we consider the low-level monitoring and experimental performance evaluation of a new implementation of the UPC compiler on the SGI Origin family of NUMA architectures. These systems offer many opportunities for the high-performance implantation of UPC. They also offer, due to their many hardware monitoring counters, the opportunity for low-level performance measurements to guide compiler implementations. Early, UPC compilers have the challenge of meeting the syntax and semantics requirements of the language. As a result, such compilers tend to focus on correctness rather than on performance. In this work, we report on the performance of selected applications and kernels under this new compiler. The measurements were designed to help shed some light on the next steps that should be taken by UPC compiler developers to harness the full performance and usability potential of UPC under these architectures.

1. Introduction

UPC, also known as Unified Parallel C, is an explicit parallel programming language based on the ANSI C standard and the distributed shared memory programming model. Further, it capitalizes on the experience gained from its predecessor distributed shared memory C compilers such as Split-C[1], AC[2] and PCP[3]. The language keeps the C philosophy of making programs concise while giving the programmer the ability to get closer to the hardware to gain more performance. Therefore, UPC is becoming more and more widely accepted in the high-performance computing community. UPC is the effort of a consortium of government, industry and academia. This has translated into efforts at many of the vendors to offer UPC compilers. UPC compilers are available for SGI O2k and O3k, Cray T3E and X1, and HP AlphaServer SC. More implementations are underway. The UPC Specifications v1.0 were produced in February 2001[4], and a new release is currently under development by the UPC consortium.

UPC is developed based on the lessons learned from message passing and pure shared memory programming paradigms. The shared memory model brings ease of use. Under this style of programming remote memory accesses need not to be expressed explicitly. However, loss of performance can arise due to unintended remote accesses, as programmers have no control over how data gets placed in the address space. Message passing offers explicit distribution of data and work such that unnecessary messages can be avoided. On the other hand, message passing is characterized by its significant overhead for small messages and it is largely hard to use. The distributed shared memory programming paradigm, which is followed by UPC, brings the best of these two worlds. It allows the exploitation of data locality by distributing data and processing under the control of the programmer, while maintaining ease of use.

The SGI Origin family is characterized by its Non-Uniform Memory Access Architecture (NUMA). The architecture offers global address space, distributed among the nodes. Thus, this architecture is quite suitable for the distributed shared memory programming paradigms, such as UPC. In this study, it will be shown that there is still significant room for the current UPC compiler implementations to take more advantage of the capabilities offered by this hardware. In general, it will be shown that many of the current practices in developing distributed shared memory compilers can in fact mask out the benefits of such languages. Further, it can also ignore some of the powerful features of the underlying system.

* On leave from the American University in Cairo, P.O. Box 2511, Cairo Egypt

A number of work-around and hand-optimizations were already developed in [5]. The impact of such hand-coded optimizations in this architecture will be also explored in order to reveal the potential performance improvements for this compiler.

This paper is organized as follows. Section 2 gives a brief description of the UPC language, while section 3 introduces the experimental testbed and workloads used. Section 4 presents the performance measurements, followed by conclusions in section 5.

2. Unified Parallel C (UPC)

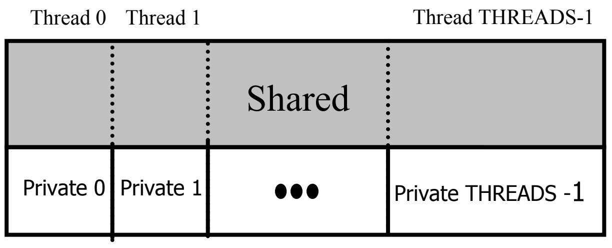

Figure 1. The UPC Memory and Execution Model

Figure 1 illustrates the memory and execution model seen by the UPC programmers. A number of threads are working independently and each can reference any address within the shared space as well as its own private space. UPC uses two special values, THREADS to indicate the total number of threads and MYTHREAD to identify each thread. UPC divides its memory space into two parts, one is shared and the other is for private memory spaces. The shared space is partitioned such that each thread has a unique association (affinity) with a shared partition. The underlying objective is to allow UPC programmers, using proper declarations, to keep the shared data that are dominantly processed by a given thread (and occasionally accessed by others) associated with that thread. Thus, a thread and the data that has affinity to it can easily be mapped by the hardware onto the same physical node. This way, data locality can simply be exploited inherently in applications.

Since UPC is an explicit parallel extension of ANSI C, all language features of C are already embodied in UPC. In addition, UPC declarations give the programmer control of the distribution of data across the threads. Among the interesting and complementary UPC features is a work-sharing iteration statement, known as *upc_forall()*. This statement helps to distribute independent loop iterations across the threads, such that iterations and data that are processed by them are assigned to the same thread. UPC also defines a number of rich private and shared pointer concepts. In addition, UPC supports dynamic shared memory allocations. There is generally no implicit synchronization in UPC. Therefore, the language offers a rich range of synchronization and memory consistency control constructs. Among the most interesting synchronization concepts is the non-blocking barrier, which allows overlapping local computations and inter-thread communications.

Figure 2 shows a short example of an UPC program, for performing a vector addition. In this example, the array v1, v2 and v1plusv2 are declared as shared integers. The default data distribution is used, placing one array element per thread, in a round-robin scheme. In UPC, an data can be also distributed across threads by arbitrary block sizes.

```
#include <upc_relaxed.h>

#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main(void)
{
  int i;
  upc_forall( i=0; i<N; i++; &v1[i] )
    v1plusv2[i] = v1[i] + v2[i];
}
```

Figure 2. An UPC Vector Addition

The UPC work-sharing construct, *upc_forall()*, has four fields. The first three are similar to those used for a regular C *for()* statement. The fourth field is called "affinity" and is used to indicate that the thread which has the element v1[i] will be executing the ith iteration. Thus, unnecessary remote accesses can be avoided. The fourth field can also be an integer type. In this case, the thread number (i mod THREADS) will be executing the ith iteration.

3. Testbed and Applications

3.1 TESTBED

As an experimental testbed, we have used the SGI Origin 3800 and the SGI Origin 2000. Both machines were running the GCC-UPC v3.2.0.4 compiler. The SGI Origin family is built on a NUMA architecture model. The Origin 3800 configuration we used has 32 MIPS R14000 processors, each running at 600 MHz, and organized as a network of SMP nodes, each with 4 processors. The total memory installed is 8GB with a bandwidth of 3.2GB/s. The Origin 2000 has 32 MIPS R10000 processors, each running at 195MHz, with 2 processors per node. The total memory available is 16GB with a bandwidth of 780MB/sec. Both systems are running under Irix 6.5.

The GCC-UPC compiler v3.2.0.4 [6] is based on the standard GCC compiler v3.2, but with added parsing modules for UPC.

The low-level monitoring is done using the SGI Speedshop software suite [7]. Perfex, one tool included in the Speedshop tool-kit, is capable of retrieving the value of selected CPU monitoring registers. The nature of several of these registers depends on the type of the CPU used, but a large number of such registers is common between the R10000 and R14000 processors. Altogether, there are up to 32 different monitoring registers, counting the number of particular CPU events, such as number of CPU cycles spent, number of loads or stores issued, number of instructions issued, number of TLB misses and many others.

3.2 APPLICATIONS

In order to analyze the performance of the UPC implementation, we examined the results of a UPC implementation of the STREAM benchmark. The Sobel Edge Detector, N-Queens Problem kernels and workloads from the NAS Parallel Benchmark (NPB)[8] are also considered. In addition, MPI[9] and SHMem[10] implementations are given for many of the workloads for reference.

The STREAM Benchmark, is a simple synthetic benchmark program which measures the sustainable memory bandwidth (in MB/sec) for single and block memory copy as well as computation rate for simple vector kernels. The UPC STREAM Benchmark has been extended from the original STREAM Benchmark[11] to focus on the UPC shared (local and remote) and private addresses. For these benchmarking experiments, the memory access rates are reported in MB/sec. The higher the bandwidth, the better the compiler deals with the data accesses.

GUPS[12], or giga updates per second, is a simple program performing updates to random locations in a large shared array. In the UPC GUPS implementation, each thread first computes its own set of random indices and store them into a private array called indices. After that, each thread updates (increments) the shared array at random locations as given by indices.

The Sobel Edge Detection, performs an edge-detection using the well-known Sobel operator. The computational process involves two 3x3 masks, or filters, for detecting the vertical and horizontal edges. These 3x3 masks are used at each pixel location for computing a pixel of the result image. This program is parallelized in UPC by partitioning the source image across the threads into equal contiguous chunks of rows. With such data distribution, each thread can work independently for all its block of lines except for the last line (ghost zone), in which remote accesses to the next thread are performed.

The N-Queens Problem, computes the number of different solutions to the problem of placing N queens on an N*N chessboard such that no queen can kill the other. In other words, no two queens can be present on the same row, column or diagonal cells of the board. The algorithm uses a depth-first searching and backtracking. The parallel algorithm first develops the first branches of the tree of available valid moves and then distributes a different initial set to each thread. Thus, each thread is computing the number of solutions of its own sub-tree sequentially. This problem is an embarrassingly parallel one.

The NAS Parallel Benchmark, is developed by the Numerical Aerodynamic Simulation (NAS) program at NASA Ames Research Center for the performance evaluation of parallel supercomputers. The NPB mimics the computation and data movement characteristics of large-scale computational fluid dynamics (CFD) applications.

NPB2 are MPI-based source-code implementations and distributed by NAS. They are intended to be run with little or no tuning.

The NPB suite consists of five kernels and three pseudo-application programs. However, we will be considering only three of them:

SP (Scalar Penta-diagonal), is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional compressible Navier-Stokes equations. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system has Scalar Penta-diagonal bands of linear equations that are solved sequentially along each dimension. SP uses coarse-grain communications.

CG (Conjugate Gradient), is a kernel that computes an approximation to the smallest eigenvalue of symmetric positive definite matrix. It features unstructured grid computations requiring irregular long-range communications.

EP (Embarrassingly Parallel), is a kernel that can run on any number of processors with little communication. It estimates the upper achievable limits for floating point performance of a parallel computer. It generates pairs of Gaussian random deviates according to a specific scheme and tabulates the number of pairs in successive annuli.

The performance analysis will be considering not only UPC but also MPI and SHMem implementations.

The Message Passing Interface (MPI) is the most widely used message passing paradigm. In this paradigm, parallel processing is derived from the use of a set of concurrent sequential processes, cooperating on the same task. Each process has its own private address space. Thus, two-sided communication in the form of send and receive operations is needed. This involves a significant overhead in case of a large number of small messages. The difficulty of programming is also one other problem with this paradigm.

The SHMem library provides a low-level high performance communication paradigm. It allows users to optimize performance at the cost of programmability. SHMem is an excellent reference point for UPC. The target here is to attempt to show how UPC can have a performance close to SHMem, while enjoying high-level programmability.

4. Experimental Measurements and Observations

4.1 STREAM Benchmark

Table 1 presents the results of the stream micro-benchmark. These measurements, made on the Origin 3800, show that UPC remote shared accesses can be up to 1.5 times slower than UPC local shared accesses. It illustrates that through affinity, UPC programmers have control that can lead to improving data locality and performance. However, UPC local shared accesses can be up to one order of magnitude slower than private accesses. This is mainly due to the fact that UPC compilers are not yet mature enough to avoid shared address resolution even in this case where the Origin architecture offers a global address space. In this case, it is expected to see the same bandwidth for local shared and private memory accesses. This implies that effective compiler implementations should be able to recognize automatically whether a shared variable is accessed locally or remotely. In case of a local shared access, the compiler should have the same cost as for private accesses.

MB/Sec	memcpy	array copy c[i] = a[i]	scale c[i] = K*a[i]
GCC	400	266	266
UPC Private	400	266	266
UPC Shared (Thread Local)	N/A	40	44
UPC Shared (SMP Local)	N/A	40	44
UPC Shared (Remote)	N/A	34	38
MB/Sec	sum c[i] += a[i]	upc_memget (shared to private)	upc_memput (private to shared)
GCC	800	N/A	N/A
UPC Private	800	N/A	N/A
UPC Shared (Thread Local)	100	400	400
UPC Shared (SMP Local)	88	266	266
UPC Shared (Remote)	72	200	200

Table 1. UPC Stream Measurements

In order to have a better understanding of the overhead caused by a local shared memory access, we monitored the STREAM benchmark *array_copy* test using perfex for two UPC single thread programs. The first program reads a local shared array and writes the result into a private table. The second program is performing a private array read and storing it into another private table. The results, shown in figure 3, estimate the overhead introduced by the compiler to complete a single local shared read. This experiment has been conducted on the Origin 2000.

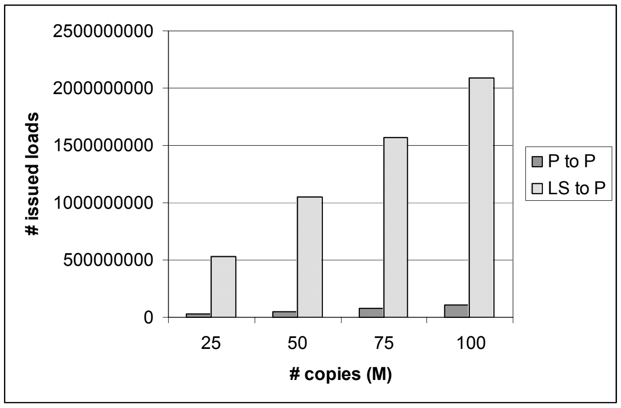

Figure 3a. Issued Loads

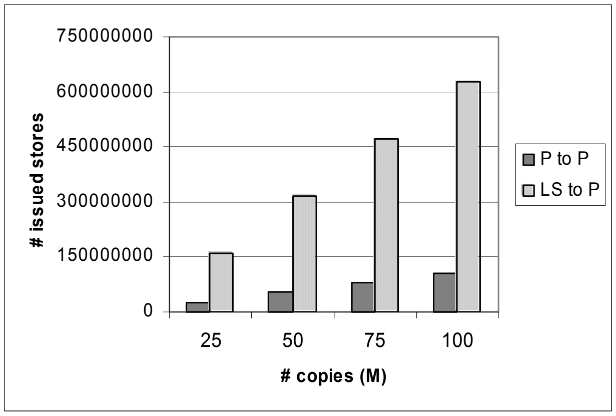

Figure 3b. Issued Stores

Figure 3. Issued memory access instructions for local shared versus private

In both cases, arrays for reads and writes are declared statically and their size is kept as a constant (100MB). Only the number of elements to be copied is variable. Since the low-level monitoring is particularly sensitive to any changes, the overhead associated with the size of static arrays is kept identical among the different runs. Each program has been run many times to decrease the amount of noise in the monitoring measurements. It is shown in Figure 3 that the ratio between local shared and private accesses is almost constant, for issued loads and stores parameters in this workload. The hardware monitoring register "decoded loads" is incremented when a load instruction is decoded in the previous cycle. It does not take in consideration any prefetch, cache operations and synchronization operations. The register "decoded stores" is in a similar manner, incremented when a store instruction is decoded in the previous cycle. These ratios express the amount of additional work associated with a single local shared memory get for this specific program. The ratio of issued decoded loads and stores from figure 3 are 20 and 6 respectively.

This shows that a local shared read of one data element leads the processor to perform 20 private reads and 6 private writes. This overhead is quite high, since the NUMA architecture has a global address space and the memory update can be done implicitly. This behavior as we look further is due to the fact that early UPC compilers that are emphasizing the UPC language issues but still have a significant room for improving their performance.

4.2 The GUPS microkernel

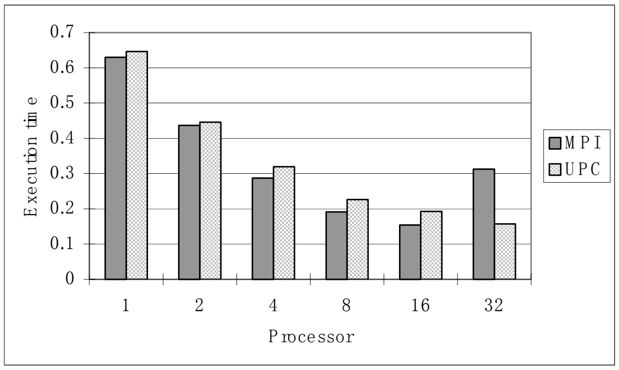

Figure 4. GUPS Performance (MPI and UPC)

Figure 4 considers the performance of the GUPS microkernel on the Origin 2000. This experiment confirms the fact that UPC has low overhead associated with its small accesses, as compared with MPI. As the number of processors grows, the short random accesses into remote memories increases. Therefore, while the UPC performance continues to improve beyond 16 processors, the performance of the MPI implementation starts to degrade.

4.3 The N-Queens and Sobel Edge Problems

Since UPC compilers are not mature enough, they are lacking automatic optimizations. These can be emulated to uncover their potential improvements. The most important one is privatization, which makes local shared space look as if it were private. By casting a local shared pointer to a private pointer, the entire overhead associated with the local shared memory accesses can be avoided. This is made possible by the fact the UPC, as C itself, allows casting, and the fact that a private pointer in UPC can access not only the private space of a thread, but also the part of the shared space that has affinity to that thread. One other optimization is to perform block moves, using *upc_memget* and *upc_memput*, to emulate prefetching and aggregating remote accesses, to hide the latency of remote data transfers and amortize their cost. Also, one can overlap local computations with remote accesses, using split-phase barriers, to hide synchronization cost.

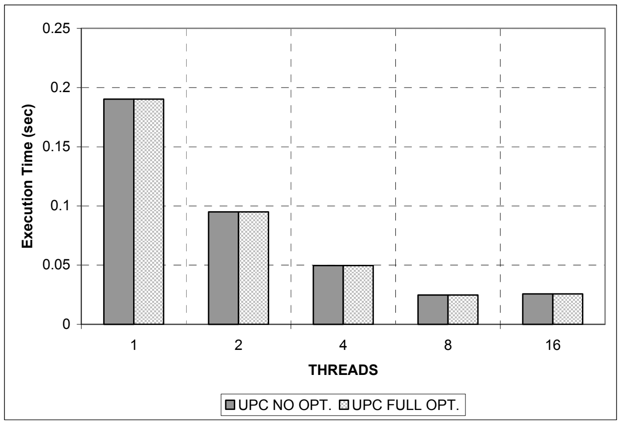

Figure 5. Impact of UPC Optimizations on Performance for N-Queens (N=14)

The impact of hand-optimization greatly depends on the application developed. As it is shown in Figure 5, the N-Queens problem does not get any benefit of such optimizations. This is due to the inherent nature of N-Queens, as N-Queens is embarrassingly parallel and does not require significant shared data. Thus, there is little shared memory use and all the computation is done using private variables.

On the other hand, the Sobel Edge Detection performance on the Origin 2000, presented in figure 6a, illustrates the importance of these optimizations. The image size is set to 2048x2048. The execution time is almost 9 times faster after having privatized all local shared memory accesses and prefetched the remote data lines being accessed.

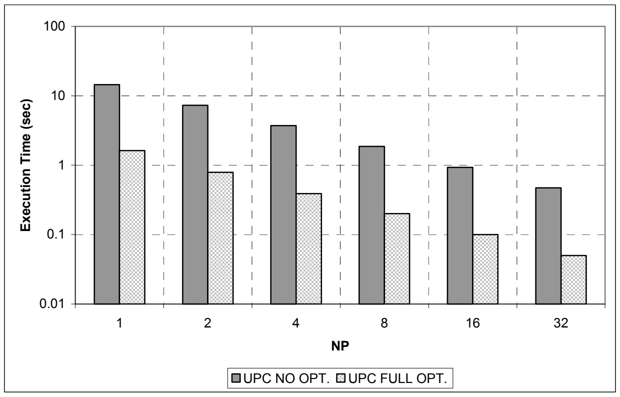

Figure 6a. Execution Time

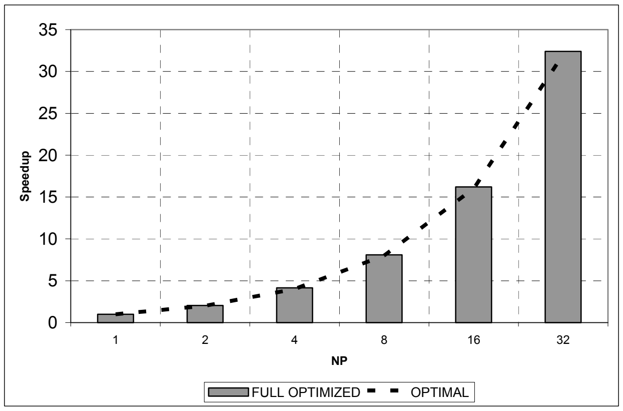

Figure 6b. Scalability

Figure 6. Impact of UPC Optimizations on Performance for Sobel Edge Detection

Figure 6b reports that the UPC optimized Sobel Edge implementation has nearly a linear speedup. Thus, the incorporation of these optimizations into the compiler can improve the execution time by one order of magnitude in some cases. However, this is clearly application dependent. Also, it is observed from N-Queens that embarrassingly parallel codes with little shared space activities are expected to perform well even without such optimizations.

4.4 The NPB Benchmark workloads

All the following experiments on the NPB workloads are using CLASS A on the Origin 3800. This class has moderate problem sizes, emphasizing both computational and communication aspects.

(sec)	F	CC	GCC	MPI (F)	SHMem (CC)	UPC (GCC)
CG	18.29	20.24	23.42	20.87	19.50	22.90
EP	38.82	80.19	97.99	38.94	79.14	98.69
SP	686.35	734.46	849.81	660.97	741.40	887.68

Table 2. Sequential Execution Times for NPB

Table 2 demonstrates how variable the performance is in uni-processor execution time [13]. All compiler optimizations were enabled for these measurements. For these three workloads, the Fortran 77 Compiler is the fastest. The C Compiler is following and GCC is worst. Except for EP, where the Fortran is 2 times faster than the C version, the average difference in speed is around 10%. GCC, however, is roughly 15% slower than the corresponding CC version. Thus, MPI is always starting with the best uni-processor time. SHMem is second, and UPC is third.

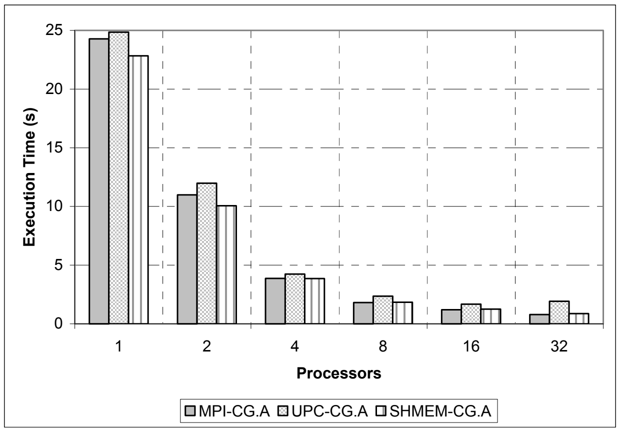

Figure 7. Parallel Performance of CG class A

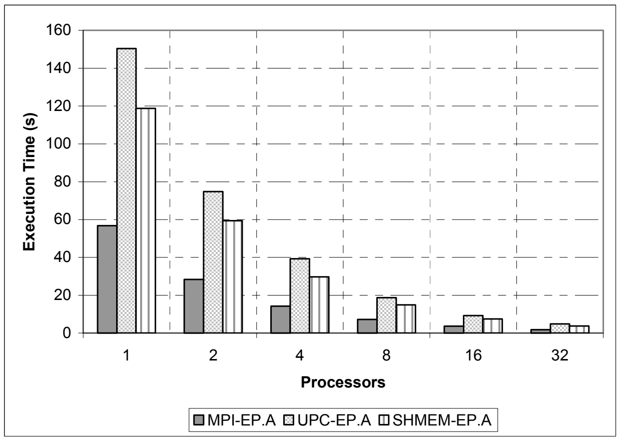

Figure 8. Parallel Performance of EP class A

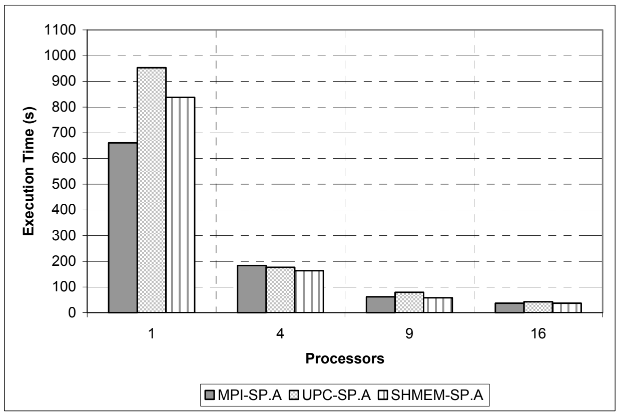

Figure 9. Parallel Performance of SP class A

Figures 7-9 report the performance measurements for the selected NPB workloads in MPI, SHMem and UPC. Each figure shows the execution time of those three programming paradigms for the respective workload. The UPC codes are hand-optimized, converting any local shared memory accesses to private accesses.

The performance measurements are showing that UPC can achieve similar levels of performance to MPI and SHMem. Note that this is the case in spite of the fact that UPC does not enjoy the same low-level optimization as the two other implementations. In addition to emulating the discussed UPC optimizations at the high level language, which has an associated cost, MPI and SHMem have collective operations libraries that are optimized at a very low level, which is not the case for UPC at present.

5. Summary and Conclusion

In this paper, several micro-benchmarks as well as application benchmarks were used to study the new UPC language implementation on the class of SGI NUMA architectures, known as O2k and O3k. The performance measurements have confirmed many of the known characteristics of UPC. In addition, the measurements have revealed many areas for potential improvements. When such improvements are incorporated into subsequent compiler developments, it is expected that the full performance and ease of use potential of UPC will materialize.

The STREAM benchmark has made it clear that under UPC, the programmer awareness of local versus remote shared data can help in improving performance by exploiting data locality. This benchmark has also demonstrated that the UPC compiler is unfortunately incurring inadequately high overhead in local shared accesses than in private accesses. This is an area for potential improvement as both local shared and private memories are available on the same node. By considering the low-level counters available under SpeedShop, it was shown that a local shared access could generate 20 loads and 6 stores under this benchmark. Such kind of costly overhead should be alleviated if the global address space capabilities of this architecture are fully exploited. In general, address resolution of shared objects at the software level must be avoided. Should the current global address machines found to be incapable of helping with that, additional hardware support should be incorporated. It is currently unclear to us whether current NUMA architectures should add more support, or the support is available but not exploited by compilers.

The GUPS micro-kernel has demonstrated that even with this overhead, UPC short random accesses are still incurring much less overhead than MPI, and for large number of processors, the scalability of MPI-GUPS starts to decrease.

The N-Queens application has attested to the efficiency of UPC, if overhead to shared accesses is removed. This is because N-Queens is embarrassingly parallel and has little shared memory activities. Thus, with no optimizations, N-Queens performance outperformed that of MPI. Sobel Edge has demonstrated a linear speedup when compiler optimizations were emulated.

Finally, the NAS Parallel Benchmark workloads have shown that while emulating compiler optimizations can make significant improvements in UPC performance, the front-end compiler, for example native CC or GCC, is also an important factor in the overall performance. In addition, low-level optimized collective libraries in the case of MPI and SHMem did help the performance.

In conclusion, by incorporating more automatic optimizations into UPC compilers and coming up with low-level optimized collective libraries, UPC will likely fulfill its promise of improved performance and better programmability. The impact of extra hardware support in current NUMA machines will be studied in our future work.

Acknowledgements

This work has been funded by DARPA through SGF under the contract #NBCHC020054 and by DoD LUCITE contract #MDA904-98-C-A081.

References

[1] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. Von Eicken, and Y. Yelick, "Introduction to Split-C", University of California, Berkeley, 1993.

[2] W. W.Carlson and J. M.Draper, "Distributed Data Access in AC," Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), Santa Barbara, CA, July 19-21, 1995, pp.39-47.

[3] Brooks, Eugene and Karen Warren. Development and Evaluation of an Efficient Parallel Programming Methodology, Spanning Uniprocessor, Symmetric Shared-memory Multi-processor, and Distributed-memory Massively Parallel Architectures, Poster SuperComputing'95, San Diego, CA., 1995

[4] T. A.El-Ghazawi, W.W.Carlson, J. M. Draper. UPC Language Specifications V1.0 (http://upc.gwu.edu). February, 2001.

[5] Tarek A.El-Ghazawi, Sébastien Chauvin, UPC Benchmarking Issues, Proceedings of the International Conference on Parallel Processing (ICPP'01). IEEE CS Press. Valencia, Spain, September 2001.

[6] Intrepid, The GCC UPC Compiler for SGI Origin Family (http:// www.intrepid.com/upc/)

[7] Silicon Graphics Inc., Speedshop User Guide

[8] D.Bailey,E., Barszcz, J.Barton. The NAS Parallel Benchmark RNR Technical Report RNR-94-007, March 1994.

[9] M. Snir, S. Otto, S. Huss, D. Walker, J. Dongarra, MPI: The Complete Reference, MIT Press, Boston, 1996.

[10] K. Hwang, C.L. Wang, and Z. Xu, "Resource Scaling Effects on MPP Performance: The STAP Benchmark Implications", IEEE Transactions on Parallel and Distributed Systems, 10(5), 1999.

[11] John D. McCalpin, Sustainable Memory Bandwidth in Current High Performance Computers, Silicon Graphics Inc.

[12] Brian R. Gaeke and K. Yelick, GUPS (Giga-Updates per Second) Benchmark, Berkeley

[13]Tarek A. El-Ghazawi, François Cantonnet, UPC Performance and Potential: A NPB Experimental Study, SuperComputing 2002 (SC2002). IEEE, Baltimore MD, USA, 2002.