UPC Performance and Potential: A NPB Experimental Study

Tarek El-Ghazawi and Francois Cantonnet

Department of Electrical and Computer Engineering
The George Washington University
Washington,DC 20052
{tarek, fcantonn }@seas. gwu.edu

Abstract

UPC, or Unified Parallel C, is a parallel extension of ANSI C. UPC follows a distributed shared memory programming model aimed at leveraging the ease of programming of the shared memory paradigm, while enabling the exploitation of data locality. UPC incorporates constructs that allow placing data near the threads that manipulate them to minimize remote accesses.

This paper gives an overview of the concepts and features of UPC and establishes, through extensive performance measurements of NPB workloads, the viability of the UPC programming language compared to the other popular paradigms. Further, through performance measurements we identify the challenges, the remaining steps and the priorities for UPC.

It will be shown that with proper hand tuning and optimized collective operations libraries, UPC performance will be comparable to that of MPI. Furthermore, by incorporating such improvements into automatic compiler optimizations, UPC will compare quite favorably to message passing in ease of programming.

1. INTRODUCTION

A good programming language should be founded on a good programming model. Programming models present the programmer with an abstract machine which allows application developers to express the best way in which their application should be executed and its data should be handled.

0-7695-1524-X/02 $17.00 (c) 2002 IEEE

Programming models should be sufficiently independent of the underlying architecture for portability, yet they should expose common architecture features to enable efficient mapping of the programs onto architectures. Programming models should remain, however, simple for ease of use.

Two popular parallel programming models are the Message Passing Model and the Shared Memory Model. In addition, we advocate the Distributed Shared Memory Model, which is followed by UPC.

In the message passing model, parallel processing is derived from the use of a set of concurrent sequential processes cooperating on the same task. As each process has its own private space, two-sided communication in the form of a sends and receives would be needed. This results in substantial overhead in interprocessor communications, especially in the case of small messages. With separate spaces, ease of use becomes another concern. The most popular example of message passing is MPI , or the Message Passing Interface.

Another popular programming model is the shared memory model. The view provided by this model is one in which multiple, independent threads operate in a shared space. The most popular implementation of this model is OpenMP. This model is characterized by the ease of use as remote memory accesses need not be treated any differently from local accesses by the programmers. As threads become unaware of whether their data is local or remote, however, excessive remote memory accesses might be generated.

The distributed shared memory programming model can potentially achieve the desired balance between ease of use and exploiting data locality. UPC is, therefore, designed as an instance of this model. Under this model, independent threads are operating in a shared space. However the shared space is logically partitioned among the threads. This enables the mapping of each thread and the space that has affinity to it to the same physical node. Programmers can thus declare data that is to be processed by a given thread, in the space which has affinity to that thread. This can be easily achieved in UPC.

Through experimental measurements, this paper will establish the important compiler implementations that should take place as the next step of the UPC effort. In particular it will be shown that the availability of an optimized collective communication library is crucial to the performance of UPC. Therefore, the UPC consortium is currently engaged in defining and implementing such a library, which can be optimized at low levels. Other optimizations and potential benefits will be also discussed based on the results.

This paper is organized as follows. Section 2 gives a brief overview of UPC, while section 3 introduces the experimental study strategy. Section 4 presents the experimental measurements, followed by conclusions in section 5.

2. AN OVERVIEW OF UPC

History and Status

UPC, or Unified Parallel C, builds on the experience gained from its predecessor distributed shared memory C languages such as Split-C[Cul93], AC[Car99], and PCP[Bro95]. UPC maintains the C philosophy by keeping the language concise, expressive, and by giving the programmer the power of getting closer to the hardware. These UPC features have gained a great deal of interest from the community. Therefore, support for UPC is consistently mounting from high-performance computing users and vendors. UPC is the effort of a consortium of government, industry and academia. Through the work of this community, the UPC specifications V1.0 were produced in February 2001 [ElG01a]. Consortium participants include GWU, IDA, DoD, ARSC, Compaq, CSC, Cray Inc., Etnus, HP, IBM, Intrepid Technologies, LBNL, LLNL, MTU, SGI, Sun Microsystems, UC Berkeley, and US DoE. This has translated into efforts at many of the vendors to develop and commercialize UPC compilers. One example is the Compaq UPC effort, which has produced a relatively mature product. Other commercial implementations that have been just released or underway include compilers for SGI O2000, Sun Servers and Cray SV-2. An open-source implementation for the Cray T3E also exists. There are many other implementations underway. A total view debugger is now available from Etnus, for some of the implementations.

While UPC specifications are the responsibility of the whole UPC consortium, the consortium carries out its work through three working groups: low level communication libraries, collective operations, and benchmarking and I/O.

UPC Model and Basic Concepts

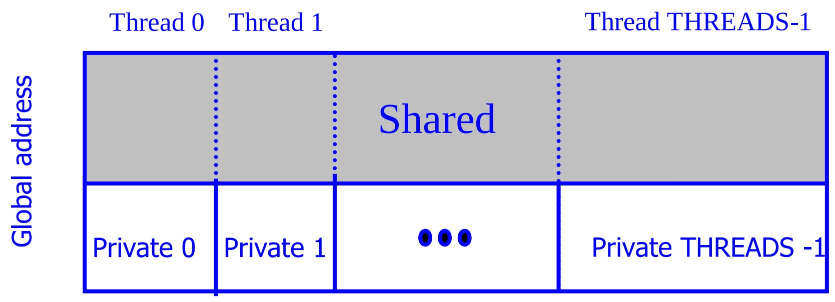


Figure 1. The UPC Memory and Execution Model

Figure 1 illustrates the memory and execution model as viewed by UPC codes and programmers. Under UPC, memory is composed of a shared memory space and a private memory space. A number of threads work independently and each of them can reference any address in the shared space, but only its own private space. The total number of threads is “THREADS” and each thread can identify itself using “MYTHREAD,” where

"THREADS" and "MYTHREAD" can be seen as special constants. The shared space, however, is logically divided into portions each with a special association (affinity) to a given thread. The idea is to make UPC enable the programmers, with proper declarations, to keep the shared data that will be 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 likely be mapped by the system into the same physical node. This can clearly exploit inherent data locality in applications.

UPC is an explicit parallel extension of ANSI C. Thus, all language features of C are already embodied in UPC. In addition, UPC declarations give the programmer control over how the data structures are distributed across the threads, by arbitrary sized blocks, in the shared space. Among the interesting and complementary UPC features is a work-sharing iteration statement known as upc_forall. This statement can spread independent loop iterations across threads based on the affinity of the data to be processed. This is also complemented by the presence of rich private and shared UPC pointers and pointer casting ability that offer the sophisticated programmer substantial power. In addition, UPC supports dynamic memory allocation in the shared space.

On the synchronization and memory consistency control side, UPC offers many powerful options. A fact worth mentioning here is that the memory consistency model can be set at the level of a single object, a statement, a block of statements, or the whole program.

Many other rich synchronization concepts are also introduced including non-blocking barriers that can overlap synchronization with local processing for hiding synchronization latencies.

A First UPC Example

```
#include <upc_relaxed.h>
#define N 100 * THREADS
shared int v1[N], v2[N], v1plusv2[N];

void main () {

      int i;
      upc_forall (i = 0; i < N; i ++; &v1[i])
            v1plusv2[i] = v1[i] + v2[i];

}
```

Figure 2. A UPC Vector Addition

Figure 2 shows a short example of a UPC program, a vector addition. In this example, arrays v1, v2 and v1plusv2 are declared as shared integers. By default they are distributed across the threads by blocks of one element each, in a round robin fashion. Under UPC, this default distribution can be altered and arbitrary block sizes can be utilized. The UPC work sharing construct “upc_forall” explicitly distributes the iterations among the threads. “upc_forall” has four fields, the first three are similar to those found in the “for” loop of the C programming language. The fourth field is called “affinity” and in this case it indicates that the thread which has the v1[i] element should execute the ith iteration. This guarantees that iterations are executed without causing any unnecessary remote accesses. The fourth field of a upc_forall can also be an integer expression, e.g. i. In such a case, thread (i mod THREADS) executes the ith iteration.

Data and Distribution

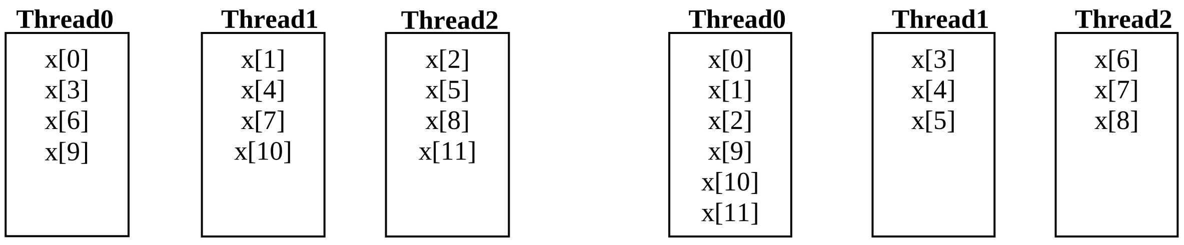
In UPC, an object can be declared as shared or private. A private object would have several instances, one at each thread. A scalar shared object declaration would result in one instance of such object with affinity to thread 0. The following example shows scalar private and shared declarations.

```
int x;              // x is private, one x in the private space of each thread
shared int y;       // y is shared, only one y at thread 0 in the shared space
```

UPC distributes shared arrays by blocks across the different threads. The general declaration is shown below:

```
    shared [block-size] array [number-of-elements]
```

When the programmer does not specify a blocking factor, UPC automatically assumes a blocking factor of one. For example if the user declares an array `x` as `shared int x[12]`, UPC would use a blocking factor of 1 and distribute the `x` array in a round robin fashion as shown in figure 3a. Alternatively, if this declaration were `shared[3] int x[12]`, UPC would distribute the array with a blocking factor of 3. In this case, `x` will be distributed across the shared memory space in three-element blocks across the threads in a round robin fashion, see figure 3b.



`shared int x[12]`
a. Default blocking factor

`shared [3] int x[12]`
b. User defined blocking factor

Figure 3. Data layout in UPC

Pointers in UPC

Pointer declarations in UPC are very similar to those of C. In UPC there are four distinct possibilities: private pointers pointing to the private space, private pointers pointing to the shared space, shared pointers pointing to the shared space, and lastly shared pointers pointing into the private space, see figure 4.

Where does the pointer point to?

		Private	**Shared**
Where do the pointers reside?	**Private**	**PP**	**PS**
	Shared	**SP**	**SS**

Legend :

PP – private to private
PS – private to shared
SP – shared to private
SS – shared to shared

Figure 4. Pointer Table

Consider the following pointer declarations :

```
int *p1;                        // private to private
shared int *p2;                 // private to shared
int *shared p3;                 // shared to private
shared int *shared p4;          // shared to shared
```

The first statement declares a private pointer p1, which points to the private space and resides in the private space. We notice that p1 is clearly a typical C pointer. p2 is a private pointer that points to the shared space. Therefore, each thread has an instant of p2. On the other hand, p4 is a shared pointer pointing to the shared space. Thus, it has one instance with affinity to thread 0. p3, however, is a shared pointer to the private space and therefore should be avoided.

Figure 5 demonstrates where the pointers of the previous example declarations are located and where they are pointing.

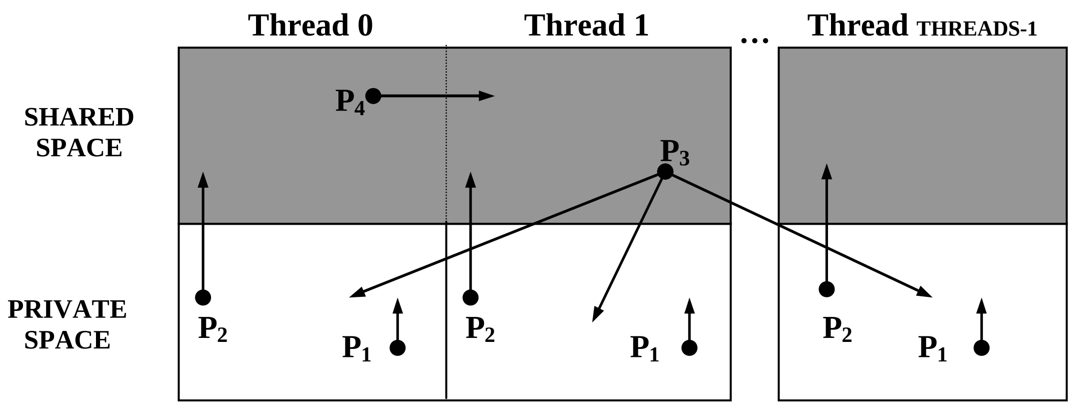


Figure 5. UPC Pointer Scenarios

In order to keep track of shared data, a UPC shared pointer (pointer to a shared object) is composed of three fields : thread, phase, and virtual address (see figure 6). The thread information clearly indicates the thread affinity of the data item, i.e. to which thread the datum has its affinity. On the other hand, V_{add} indicates the block address, and Phase indicates the location of the data item within that block. UPC allows the casting of one pointer type to the another. Casting a shared pointer to a private pointer results in the loss of the thread information. On the other hand, casting a private pointer to a shared pointer is not advised and would produce unknown results. Shared pointers can also have a blocking factor to traverse blocked arrays as needed.

Thread	V_{add}	Phase

Figure 6. A Shared Pointer Format

Dynamic Memory Allocation

Being an ANSI C extension, allocating private memory is already supported in UPC under the C syntax. Shared memory allocation is, however, supported using the UPC extensions.

UPC provides three ways to dynamically allocate shared memory. They are **upc_global_alloc()**, **upc_all_alloc()**, and **upc_local_alloc()**.

The **upc_global_alloc()** routine allocates memory across all the threads and returns one pointer to the calling thread. The syntax for **upc_global_alloc()** is as follows:

```
shared void *upc_global_alloc(size_t nblocks, size_t nbytes)
```

upc_global_alloc() is not a collective call. Be aware that if **upc_global_alloc()** is called by multiple threads, each thread will allocate the amount of memory requested. If the intent is to give all the threads access to the same block of memory, **upc_all_alloc()** should be used instead.

The **upc_all_alloc()** routine is called by all the threads collectively and each thread is returned with the same pointer to memory. The syntax for **upc_all_alloc()** is:

shared void *upc_all_alloc(size_t nblocks, size_t nbytes)

This routine allows the threads easy access to a common block of memory, which could be very useful when performing a collective operation.

Finally the **upc_local_alloc()** routine is provided as a way to allocate local memory to each thread. **upc_local_alloc()** is not a collective function. The syntax for **upc_all_alloc()** is:

shared void upc_local_alloc(size_t nblocks, size_t nbytes)

upc_local_alloc() returns a pointer to the local shared memory.

The **upc_free()** function is used to free dynamically allocated shared memory space. This call is not a collective routine. In the case of a global shared buffer created by **upc_all_alloc()**, the freeing is only effective when all the threads have completed the **upc_free()** call.

Memory Consistency

The adopted memory consistency model determines the order of data accesses in the shared space. Under a strict memory consistency, accesses take place in the same order that would have resulted from sequential execution. Under a relaxed policy, however, the compiler can order such accesses in any way that can optimize the execution time.

UPC provides memory consistency control at the single object level, statement block level, and full program level. Either relaxed or strict mode can be selected. In the relaxed mode, the compiler makes the decisions on the order of operations. In the strict case, sequential consistency is enforced. Statements including implicit or explicit barriers and fences force all prior accesses to complete. The programmer defines the memory consistency behavior, as follows.

At the global level:

```
#define <upc_relaxed.h>   /* for relaxed consistency */

#define <upc_strict.h>   /* for strict consistency */
```

At the statements level:

```
#pragma upc_strict       /* following statements have strict consistency */

#pragma upc_relaxed      /*  following statements have strict consistency */
```

At the object level:

The keyword `strict` or `relaxed` can be used when a variable needs to be treated in a strict or relaxed manner. For example, a possible declaration could be:

```
strict shared int x;
```

Thread Synchronization

Thread synchronization in UPC is facilitated through locks and barriers. Critical sections of code can be executed in a mutually exclusive fashion using locks. Example lock primitives are:

```
void upc_lock(shared upc_lock *l)          /* grabs the lock */
void upc_unlock(shared upc_lock *l)        /* releases the lock */
int upc_lock_attempt(shared upc_lock *l)   /* returns 1 on successful lock and 0
                                              otherwise */
```

In addition to the previous primitives, locks can be also dynamically allocated and initialized.

Another commonly used synchronization mechanism is barrier synchronization. A barrier is a point in the code at which all threads must arrive, before any of them can proceed any further.

Two versions of barrier synchronization are provided. In addition to the commonly used barriers, that simply ensure that all threads have reached a given point before they can proceed, UPC provides a split-phase (non-blocking) barrier. The split-phase barrier is designed to overlap synchronization with computation. The syntax for the regular barrier follows.

```
upc_barrier;             // regular barrier
```

For the split-phase barrier, the following pair of statements is used:

Example:

```
upc_notify;              //  notify others and start local processing
…                        //  local processing
upc_wait;                //   do not proceed until others notify
```

3. EXPERIMENTAL STRATEGY

Testbed

As an experimental testbed, we have used the Compaq AlphaServer SC and the Compaq UPC 1.7 Compiler. The AlphaServer SC has a NUMA architecture. It is based on the AlphaServer ES40, which is a SMP machine with four Alpha EV67 processors. The machine is essentially a cluster of these SMP nodes in which each node has four processors and an interconnect. The interconnect is a Quadrics switch with 440Mb/s throughput (206Mb/s under MPI) and 6ns latency. The nodes are connected together through the Quadrics switch into a fat tree topology. The communication hardware allows one sided communication with limited software support. The Compaq implementation of UPC takes advantage of this particular communication layer when performing remote accesses.[Compaq99]

Applications

First, to illustrate how to optimize UPC codes as well as the potential for the UPC language, we examined micro-benchmarks based on the stream benchmark as well as the Sobel edge detection problem in UPC. Then, in order to establish the current standing of UPC and the remaining steps that need to be taken, we have implemented and presented performance measurements for workloads from the NAS Parallel Benchmark (NPB) [Bai94]. The UPC implementations were compared against an NPB 2.3 MPI implementation, which is written in Fortran+MPI, except for IS which is written in C+MPI.

UPC Stream Benchmarks

The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. The micro-benchmarks used for testing the Compaq UPC 1.7 compiler was extracted from the STREAM benchmark(e.g. Put, Get and Scale) and extended for UPC. For these synthetic benchmarking experiments, the memory access rates are measured and presented in MB/s (1000000 bytes transferred per second). The higher the bandwidth, the better and more complete are the complier optimizations.

Sobel Edge Detection

Edge detection has many applications in computer vision, including image registration and image compression. One popular way of performing edge detection is using the Sobel operators. The process involves the use of two masks for detecting horizontal and vertical edges. The west mask is placed on the top of the image with its center on top of the currently considered image pixel. Each of the underlying image pixels is multiplied by the corresponding mask pixel and the results are added up, and the sum is squared. The same process is applied to the north mask and the square root for the two squared sums becomes the pixel value in the edge image. In parallelizing this application, the image is portioned into equal contiguous slices of rows that are distributed across the threads, as blocks of a shared array. With such contiguous horizontal distribution, remote accesses into the next thread will be needed only when the mask is shifted over the last row of a thread data to access the elements of the next row.

The NAS Parallel Benchmark Suite

The NAS parallel benchmarks (NPB) 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 computation fluid dynamics (CFD) applications.

The NPB comes in two flavors **NPB 1** and **NPB 2**. The **NPB 1** are the original "pencil and paper" benchmarks. Vendors and others implement the detailed specifications in the NPB 1 report, using algorithms and programming models appropriate to their different machines. On the other hand **NPB 2** are MPI-based source-code implementations written and distributed by NAS. They are intended to be run with little or no tuning. Another implementation of NPB 2 is the **NPB 2-serial**; these are single processor (serial) source-code implementations derived from the NPB 2 by removing all parallelism [NPB]. We have therefore used NPB 2 in our MPI execution time measurements. NPB 2-serial was used to provide the uniprocessor performance when reporting on the scalability of MPI.

The NPB suite consists of five kernels (EP, MG, FT, CG, IS) and three pseudo-applications (LU, SP, BT) programs. The bulk of the computations is integer arithmetic in IS. The other benchmarks are floating-point computation intensive. A brief description of each workload is presented in this section.

BT (Block Tri-diagonal) is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions. The resulting systems are Block-Tridiagonal of 5x5 blocks and are solved sequentially along each dimension. BT uses coarse-grain communications.

SP (Scalar Penta-diagonal) is a simulated CFD application that has a similar structure to BT. 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 Pentadiagonal bands of linear equations that are solved sequentially along each dimension. SP uses coarse-grain communications.

LU (Block Lower Triangular) : is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block **L**ower and **U**pper triangular systems. LU performs a large number of small communications (five words) each.

FT (Fast Fourier Transform): This benchmark solves a 3D partial differential equation using an FFT-based spectral method, also requiring long range communication. FT performs three one-dimensional (1-D) FFT's, one for each dimension.

MG (MultiGrid): The MG benchmark uses a V-cycle multigrid method to compute the solution of the 3-D scalar Poisson equation. It performs both short and long range communications that are highly structured.

CG (Conjugate Gradient): This benchmark computes an approximation to the smallest eigenvalue of symmetric positive definite matrix. This kernel features unstructured grid computations requiring irregular long-range communications.

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

IS (Integer sorting): This benchmark is a parallel sorting program based on bucket sort. It requires a lot of total exchange communication.

There are different versions/classes of the NPB like Sample, Class A, Class B and Class C. These classes differ mainly in the size of the problem. Figure 7 gives the problem sizes and performance rates (measured in Mflop/s) for each of the eight benchmarks, for Class A and Class B problem sets on a single processor Cray YMP.

Targeted Measurements

In this study, a number of issues were taken into account. First, as UPC itself is an extension of ANSI C, it is perhaps a fair assumption that in most cases UPC compiler writers start from an already written sequential C compiler. Given a particular machine, the available sequential C compiler (cc) may be very different from the sequential compiler that was extended to become UPC. Another observation that should also be taken into account is that the single node performance of a parallel code may be very different from that of the sequential code. In order to account for these differences and their implications, we study the performance of the applications under different compilation/execution scenarios such as: 1) sequential code produced by the "CC" compiler running serially, 2) Single process execution of code produced by CC+MPI and 3) UPC code compiled with UPC compiler and running with a single thread. In addition, measurements that can demonstrate the potential improvement from compiler optimizations and hand tuning are shown for the Sobel edge workload. Two optimizations will be demonstrated: space privatization and block prefetching. Finally, hand tuned versions of NPB in UPC are considered along with their MPI counterparts, and the respective cost of collection operations. These are used to show that if such an optimized library is made available for UPC, then UPC would have similar performance to MPI.

Table 1. NPB Problem Sizes

a) Class A workloads (Smaller Version)

Benchmark	Size	Operations(x10^3)	MFLOPS(YMP/1)
EP	2^{28}	26.68	211
MG	256^3	3.905	176
CG	14,000	1.508	127
FT	256^2 x 128	5.631	196
IS	2^{23} x 2^{19}	0.7812	68
LU	64^3	64.57	194
SP	64^3	102.0	216
BT	64^3	181.3	229

b) Class B workloads (Bigger Version)

Benchmark	Size	Operations(x10^3)	MFLOPS(YMP/1)
EP	2^{30}	100.9	
MG	256^3	18.81	498
CG	75,000	54.89	447
FT	512 x 256^2	71.37	560
IS	2^{25} x 2^{21}	3.150	244
LU	102^3	319.6	493
SP	102^3	447.1	627
BT	102^3	721.5	572

4. EXPERlMENTAL RESULTS

Performance measurements

Table 2 . UPC Stream Measurements

MB/s	**Put**	**Get**	**Scale**
CC	400.0	640.0	564.0
UPC Private	565.0	686.0	738.0
UPC Local	44.0	7.0	12.0
UPC Remote	0.2	0.2	0.2
MB/s	**Block Put**	**Block Get**	**Block Scale**
CC	384.0	384.0	256.0
UPC Private	369.0	369.0	253.0
UPC Local	150.0	300.0	145.0
UPC Remote	146.0	344.0	155.0

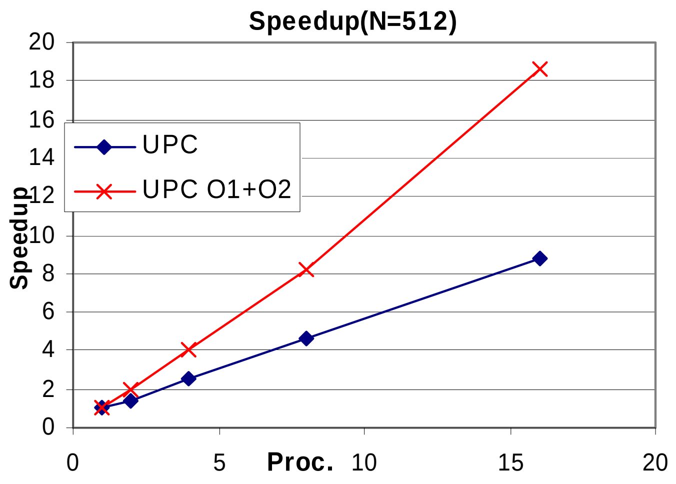
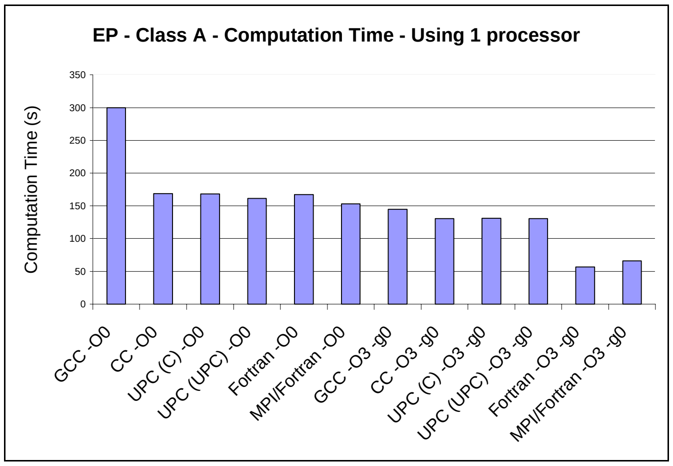


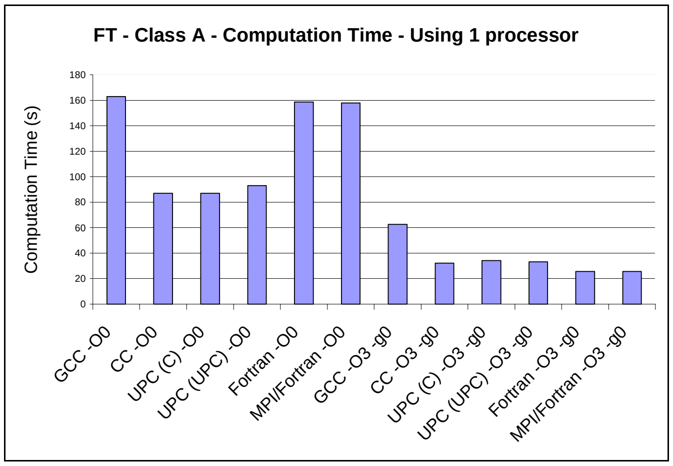
Figure 7. Sobel Edge Detection

Table 2 is based on a synthetic micro-benchmark, modeled after the stream benchmark, which measures the memory bandwidth of different primitive operations (put and get) and the bandwidth generated by a simple computation, scale. These measurements clearly demonstrate that UPC local accesses are 1 or 2 orders of magnitude faster than UPC remote accesses. This shows that UPC achieves its objective of much better performance on local accesses, and the value of the affinity concept in UPC. However, these accesses are also 1 to 2 orders of magnitude slower than private accesses. This is mainly due to the fact that UPC compilers are still maturing and may not automatically realize that local accesses can avoid the overhead of shared address calculations. This implies that one effective compiler optimization would be the ability to automatically recognize whether a shared data item is local or remote. If local, it can be treated as private, thereby avoiding unnecessary shared address translation. The powerful pointer features of UPC, allow this to be directly implemented also by programmers as a hand tuning, by casting the shared pointer to a private one. Performance of remote accesses are also shown in table 2. They suggest that prefetching, specially block prefetching as seen from the measurements is another critical automatic complier optimization. The UPC language features block puts and gets. These can be alternatively used to accomplish the same benefit through direct hand tuning.

Figure 7 considers the performance of edge detection on the Compaq AlphaServer SC. In this figure, O1 indicates the use of private pointers instead of shared pointers, while O2 indicates the use of block get to mimic the effect of prefetching. The figure shows substantial scalability increase when these hand tunings are introduced. In particular, it can be concluded that treating local accesses as private and block prefetching are two important compiler optimizations and must be incorporated in UPC compiler implementations. Considering the case and measurements of figure 7, such optimizations improved scalability by a factor of two and achieved linear scaling.



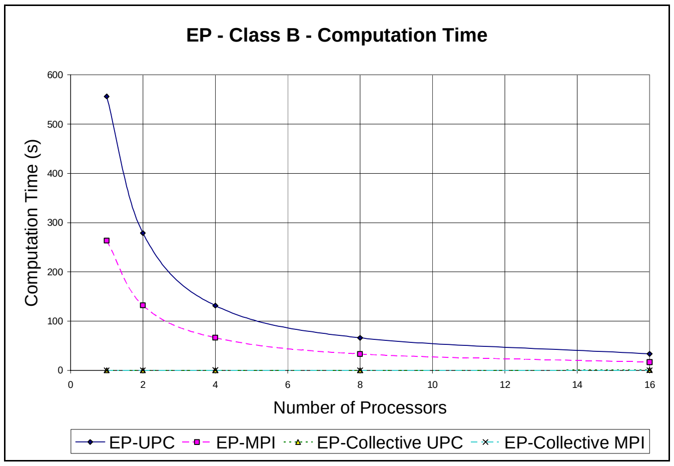
a)

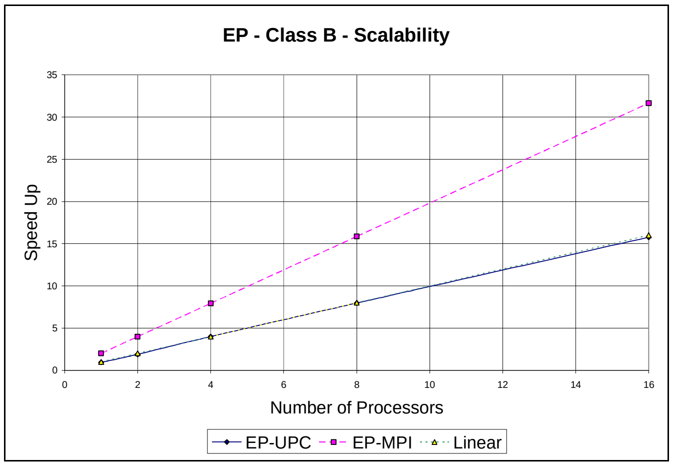


b)

Figure 8. Sequential execution time for NAS parallel benchmark suite workloads with different levels of compiler optimizations

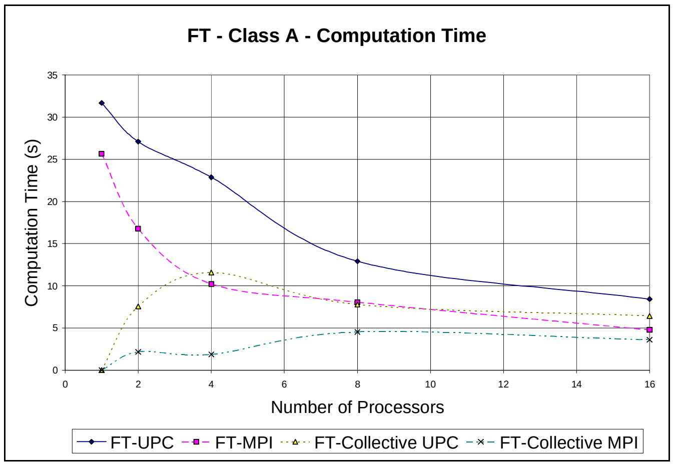
Figure 8 demonstrates how variable the performance in uniprocessor execution time is. This variability depends on the specific compiler front-end used, the specific optimization level and the associated overhead with a given machine implementation. In some of the cases, it should be also noted that the performance of C and Fortran was quite different. In case of EP, figure 8a, with all optimizations turned on, Fortran was slightly more than twice as fast as C, while in the case of FT, figure 8b, Fortran was faster by roughly 20%. Similar variability was noticed in the case of the other workloads as well. The scalability was, therefore, reported in the next set of figures based on the sequential behavior of the serial code compiled with the options that can produce the best performing sequential execution, cc –O3 –g0 in case of UPC and f77 –O3 –g0 in case of Fortran.

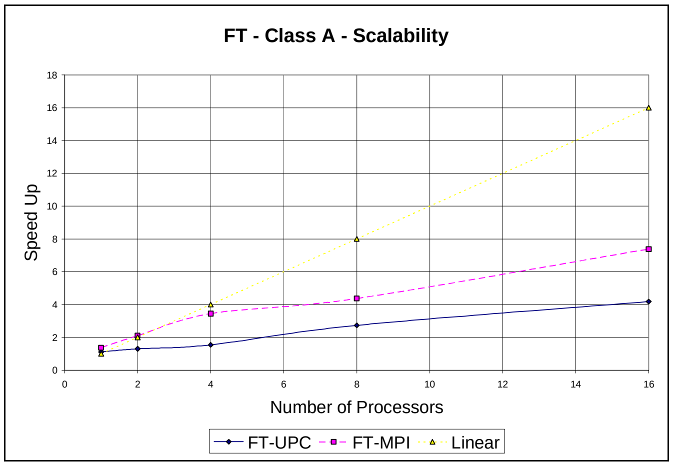




a) b)

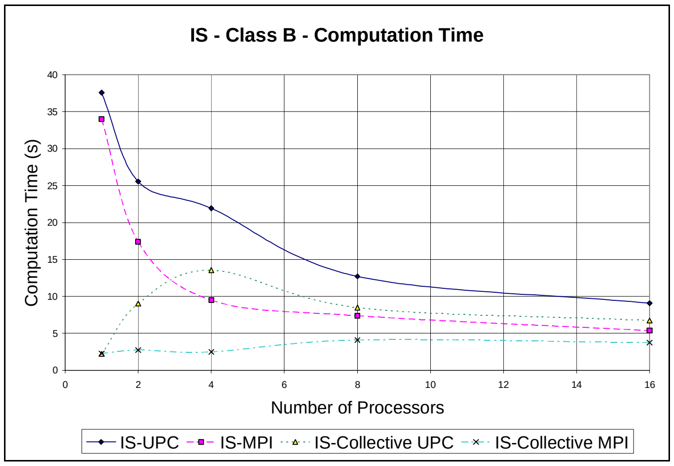
Figure 9. Parallel Performance of EP in UPC and MPI

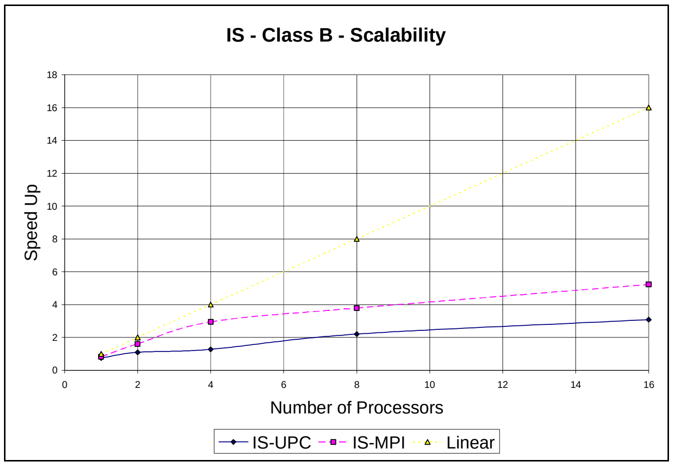




a) b)

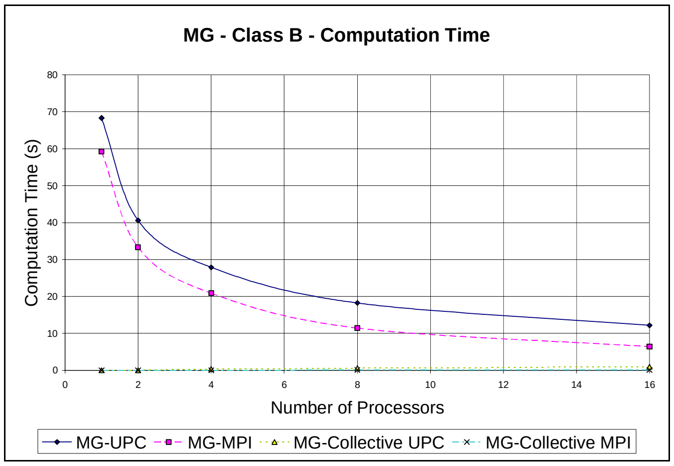
Figure 10. Parallel Performance of FT in UPC and MPI



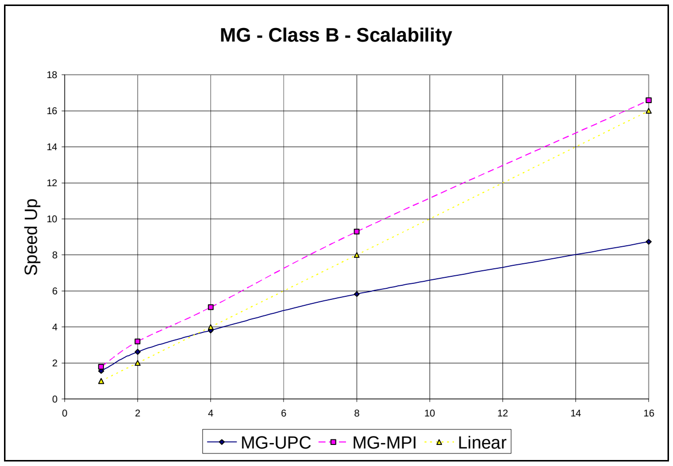


a) b)

Figure 11. Parallel Performance of IS in UPC and MPI

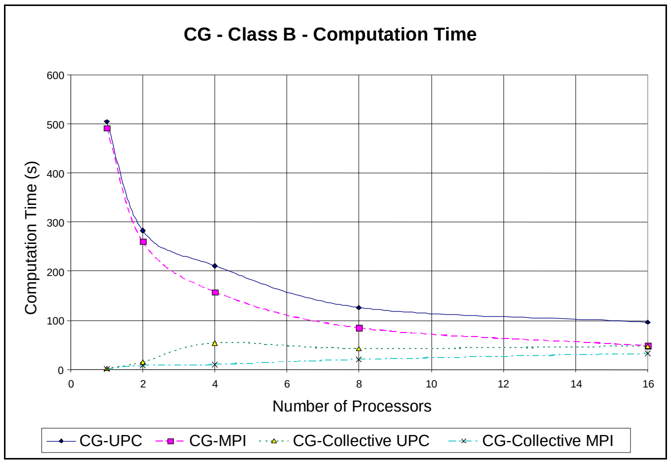


a)

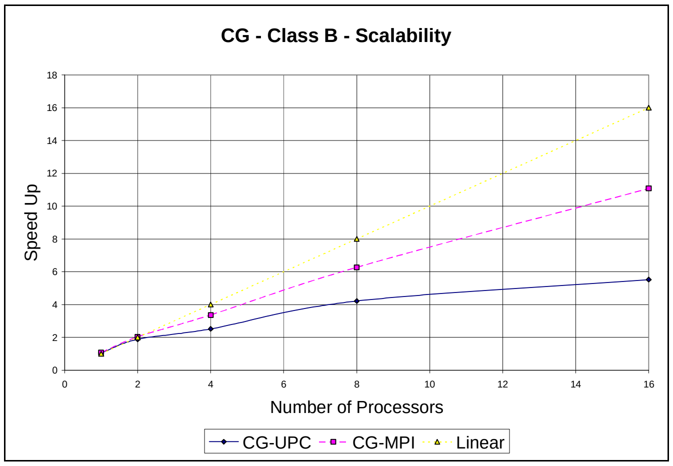


b)

Figure 12. Parallel Performance of MG in UPC and MPI

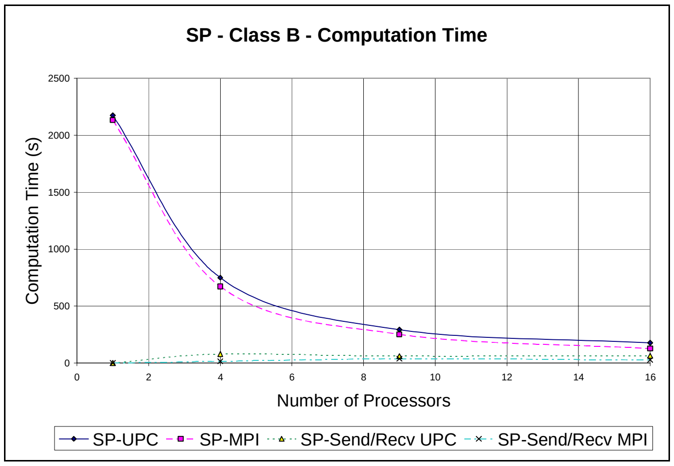


a)

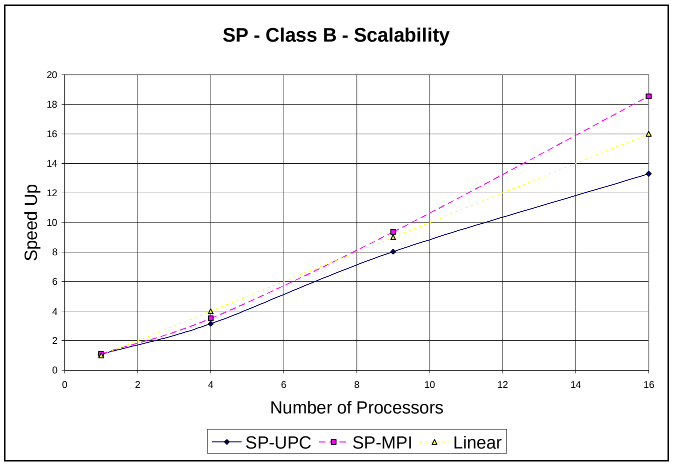


b)

Figure 13. Parallel Performance of CG in UPC and MPI



a)



b)

Figure 14. Parallel Performance of SP in UPC and MPI

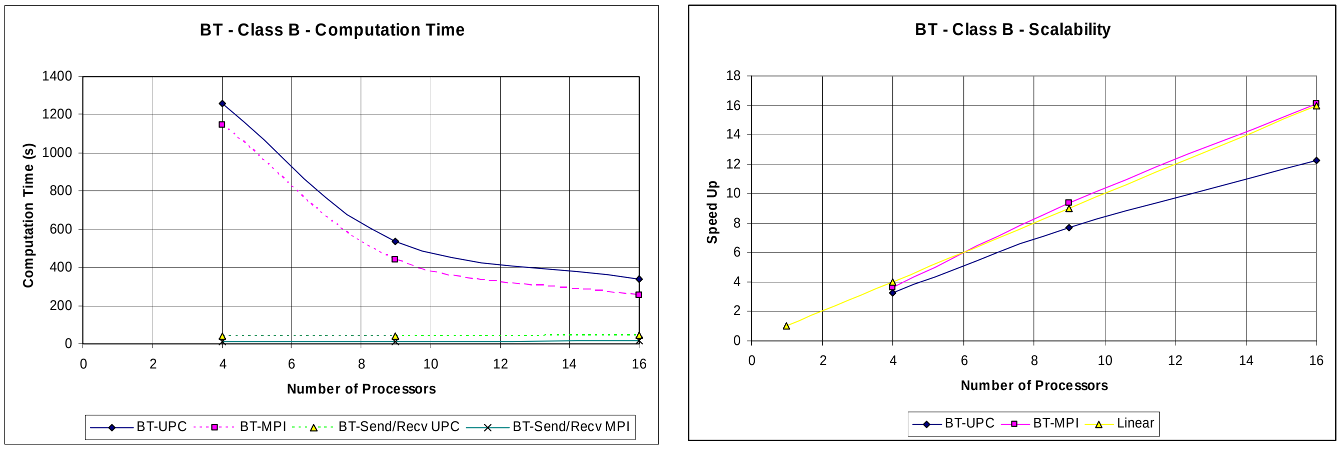


Figure 15. Parallel Performance of BT in UPC and MPI

Figures 9–15 report the performance measurements for NPB workloads in both UPC and MPI. Part a in each figure shows the timing of the respective workload under both UPC and MPI, as well as the time spent in collective operations. Part b in each of these figures show the corresponding scalability for both UPC and MPI cases, with respect to the pure C or pure Fortran code. The UPC codes are hand optimized by converting all local shared accesses to private accesses. One unaccounted-for optimization is that of implementing and optimizing collective operations at the communication fabric level. This is because, the collective operations specifications of UPC are still under development and their implementations are expected soon after. This is clearly the case for the used architecture where collective operations in UPC are implemented at the high level language while for MPI collective operations are optimized and implemented at the switch API level. Therefore, the time for collective communication is measured separately for both UPC and MPI.

Figures 9–15 provide adequate support to the proposition that UPC will perform similar to MPI if implementation qualities were similar. First, in workloads that are demanding in collective operations, such as FT, IS and CG (figures 10, 11, and 13), the cost of collective operations in part a of these figures is quite higher in UPC. This is because in UPC, collective operations are expressed at the UPC language syntax level. In case of MPI, however, collective operations are separate calls that are optimized by vendors, as previously mentioned. Close examination of the cost of such operations as shown in part a of figures 10, 11, and 13, shows that such cost is roughly equal to the time difference between executing the UPC and the MPI codes.

In SP and BT, figures 14 and 15 respectively, those differences between UPC and MPI are clearly smaller. This is due to the lower need of collective calls in these workloads.

EP and MG, figures 9 and 12, have low collective operation requirements, yet they are showing differences in performance. This only indicates that other performance factors do exist. Some of such factors are language and implementation dependent. Recall for example the performance differences between the sequential Fortran and C implementations, figure 8.

The performance measurements have simply shown that UPC can perform at similar levels to those of MPI when problems in implementations are accounted for. Thus, if a collective operation library for UPC is specified and implemented at the same low level, UPC can have a performance similar to that of MPI. Potentially, with the small overhead of UPC, for example in the case of short remote transactions, some UPC implementations may outperform current paradigms such as MPI on given architectures. Other automatic compiler optimizations such as space privatization and block prefetching will avoid programmers the effort of hand tuning, thus making programming easier.

One extension of the thread-based space privatization optimization, which makes a thread treats its local shared space as private, is the SMP space privatization. In this case multiple threads that are mapped into the same node can treat each other's local shared space as private. This is an interesting feature, but is of less importance than thread based space privatization. The reason for this is that through the concept of affinity, the programming model enables thread based locality exploitation, rather than SMP based locality exploitation. In addition, as thread based privatization is more tied to the language, it can be efficiently exploited at compile time. Exploiting SMP based locality is more tied however to the system and may require run-time support that could increase overhead.

5. CONCLUSIONS

While UPC compilers are maturing, a combination of hand tuning and optimized collective operations library can enable UPC codes to perform at levels similar to those of the current paradigms. In fact, given the added programmer's control over data placements and executions, it is conceivable to believe that for many applications and architectures UPC can outperform current paradigms.

As compilers mature and provide more automatic optimizations, UPC would become substantially easier to use as compared to existing parallel languages and tools, while maintaining a high-level of performance.

These observations define the next steps that need to be taken by the UPC community. In specific there is a need to approach the next stage of the UPC development with two parallel efforts. One that emphasizes more optimized libraries and hand tuning methods, while the other explores and exploits more aggressive automatic compiler optimizations.

In general, UPC compiler optimizations should focus on space privatization for local shared accesses, block prefetching of remote data, and low level optimized implementation of collective operation libraries. When all these optimizations are implemented efficiently, UPC may outperform current parallel programming paradigms in performance and ease of use.

ACKNOWLEDGEMENTS

This work has been funded by the DoD under the LUCITE contract #MDA904-98-C-A081. UPC is the work of an extended consortium. The authors however would like to thank Bill Carlson of IDA CC for being the source of inspiration for this work. His work with Jesse Draper have been the key to the development of the UPC language. The authors would like also to thank Lauren Smith, US Government, for her important contributions to this research direction. Finally we would like to acknowledge the work of Sébastien Chauvin, Frédéric Vroman and Ludovic Courtes. Their hard work helped us advance the UPC benchmarking effort. Input from Brian Wibecan of Compaq has been always instrumental. The authors are also grateful for the help of Hossam Abdallah, Smita Annareddy, and Sinthop Kaewpijit.

REFERENCES

[Bai94] D.Bailey,E., Barszcz, J.Barton. The NAS Parallel Benchmark RNR Technical Report RNR-94-007, March 1994.

[Bir00] John Bircsak, Kevin Harris, Robert Morgan, Brian Wibecan. Efficient Implementation of UPC, unpublished manuscript.

[Bro95] 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'9*5, San Diego, CA., December 3-8, 1995.

[Car01] William Carlson, Tarek El-Ghazawi, Bob Numeric, and Kathy Yelick. Full day tutorial in IEEE/ACM SC01, Denver, CO., November 12, 2001. (http://www.upc.gwu.edu/~upc/doc/upcsc01.pdf)

[Car99] William W.Carlson, Jesse M.Draper, David Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification CCS-TR-99-157.

[Car95] Carlson, William W. and Jesse 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.

[Compaq99] Compaq AlphaServer SC announcement whitepaper, Compaq. (http://www.compaq.com/alphaserver/download/scseriesv1.pdf).

[Cul93] Culler, David E., Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick, "Parallel Programming in Split-C," in *Proceedings of Supercomputing'*93, Portland, OR, November 15-19, 1993, pp. 262-273.

[ElG01a] Tarek A.El-Ghazawi, William W.Carlson, Jesse M. Draper. UPC Language Specifications V1.0 (http://upc.gwu.edu). February, 2001.

[ElG01b] Tarek A.El-Ghazawi, Programming in UPC (http://projects.seas.gwu.edu/~hpcl/upcdev/tut/sld001.htm), March 2001.

[ElG01c] Tarek A.El-Ghazawi, Sebastien Chauvin, UPC Benchmarking Issues, Proceedings of the International Conference on Parallel Processing (ICPP'01). IEEE CS Press. Valencia, Spain, September 2001.

[McC95] John D.McCalpin, A Survey of Memory Bandwidth and Machine Balance in Current High Performance Computers, December 1995 (http://home.austin. rr.com/mccalpin/papers/balance/index.html).

[NPB] http://www.nas.nasa.gov/Software/NPB

APPENDIX A: UPC QUICK REFERENCE GUIDE

UPC Keywords

THREADS: Total number of threads.

MYTHREAD: Identification number of the current thread (between 0 and THREADS-1).

UPC_MAX_BLOCK_SIZE: Maximum block size allowed by the compilation environment.

Shared variable declaration

Shared objects

The shared variables are declared using the type qualifier "shared". The shared objects have to be declared statically (that is either as global variables or with the keyword static).

Example of shared object declaration:

```
shared int i;
shared int b[100*THREADS];
```

The following will not compile if you do not specify the number of threads:

```
shared int a[100];
```

All the elements of `a` in `thread 0`:

```
shared [] int a[100];
```

Distribute the elements in a round robin fashion by chunks of 2 elements: all the elements of a in `thread 0`; `a[2]` and `a[3]` in `thread 1` ...:

```
shared [2] int a[100];
```

Shared pointers

Private pointer to shared data:

```
shared int *p;
```

Shared pointer to shared data:

```
shared int* shared sp;
```

Work sharing

Distribute the iterations in a round-robin fashion with wrapping from the last thread to the first thread:
```
upc_forall(i=0; i<N; i++; i)
```

Distribute the iterations by consecutive chunks:
```
upc_forall(i=0; i<N; i++; i*THREADS/N)
```

The iteration distribution follows the distribution layout of `a`:
```
upc_forall(i=0; i<N; i++; &a[i])
```

Synchronization

Memory consistency

Define strict or relaxed consistency model for the whole program.
```
#include “upc_strict.h” or “upc_relaxed.h”
```

Set strict memory consistency for the rest of the file:
```
#pragma upc strict
```

Set relaxed memory consistency for the rest of the file:
```
#pragma upc relaxed
```

All accesses to `i` will be done with the relaxed consistency model:
```
shared relaxed int i;
```

All accesses to `i` will be done with the relaxed consistency model:
```
relaxed shared int i;
```

All accesses to `i` will be done with the strict consistency model:
```
strict shared int i;
```

Synchronize locally the shared memory accesses; it is equivalent to a null strict reference.
```
upc_fence;
```

Barriers

Synchronize globally the program:

```
upc_barrier [value];
```
The value is an integer.
// Before the barrier
```
     upc_notify [value];
```
// Non-synchronized statements
  relative to this on-going barrier
```
     upc_wait [value];
```
// After the barrier

UPC operators

`upc_threadof(p)`	: thread having affinity to the location pointed by p
`upc_phaseof(p)`	: phase associated with the location pointed by p
`upc_addrfield(p)`	: address field associated with the location pointed by p

Dynamic memory allocation

3 different memory allocation methods are provided by UPC:

`upc_local_alloc(n, b)`: allocates nxb bytes of shared data in the calling thread only. It needs to be called by one thread only.

`upc_global_alloc(n, b)`: globally allocates nxb bytes of shared data distributed across the threads with a block size of b bytes. It needs to be called by one thread only.

`upc_all_alloc(n, b)`: collective allocates nxb bytes of shared data distributed across the threads with a block size of b bytes. It needs to be called by all the threads.

`upc_free(p)`: Frees shared memory pointed to by p from the heap.

String functions in UPC

Equivalent of memcpy **:**

`upc_memcpy(dst, src, size)`	: copy from shared to shared
`upc_memput(dst, src, size)`	: copy from private to shared
`upc_memget(dst, src, size)`	: copy from shared to private

Equivalent of memset**:**

`upc_memset(dst, char, size)` : initialize shared memory with a character

Locks

```
// static locks
{
      static shared upc_lock_t l;

      upc_lock_init(&l);

      //…
      upc_lock(&l);
      // protected section
      upc_unlock(&l);
}

// Dynamic lock globally allocated
{
      shared upc_lock_t* l;

      if (MYTHREAD==3)
      l = upc_global_lock_alloc ();
}

// Dynamic lock collectively allocated
{
      shared upc_lock_t* l;

      l = upc_all_lock_alloc();
}
```

General utilities

Terminate the UPC program with exist status:

```
upc_global_exit(status);
```

APPENDIX B: COMPILING AND RUNNING UPC

Compiling and Running on Cray

To compile with a fixed number (4) of threads:

```
upc –O2 –fthreads-4 –o myprog myprog.c
```

To run the program:

```
./myprog
```

Compiling and Running on Compaq

To compile with a fixed number of threads and run:

```
upc –O2 –fthreads 4 –o myprog myprog.c

prun ./myprog
```

To compile without specifying a number of threads and run:

```
upc –O2 –o myprog myprog.c

prun –n 4 ./myprog
```

Compiling and Running on SGI

To compile without specifying a number of threads and run:

```
upc –o2 myprog myprog.c

./myprog
```

Note: GNU-UPC version 1.8, automatically allocates resources to its processors, specifying the number of threads may not result in optimal allocation.