# UPC Benchmarking Issues

**Tarek El-Ghazawi**
**Sébastien Chauvin**

**School of Computational Sciences**
**George Mason University**
**4400 University Drive**
**Fairfax, VA 22030-4444**
**{tarek, schauvin}@gmu.edu**

## Abstract

*UPC, or Unified Parallel C, is a parallel extension of ANSI C. UPC is developed around the distributed shared-memory programming model with constructs that can allow programmers to exploit memory locality, by placing data close to the threads that manipulate them in order to minimize remote accesses. Under the UPC memory sharing model, each thread owns a private memory and has a logical association (affininty) with a partition of the shared memory. This paper discusses an early release of UPC_Bench, a benchmark designed to reveal UPC compilers performance weaknesses to uncover opportunities for compiler optimizations.*

*The experimental results from UPC_Bench over the Compaq AlphaServer SC will show that UPC_Bench is capable of discovering such compiler performance problems. Further, it will show that if such performance pitfalls are avoided through compiler optimizations, distributed shared memory programming paradigms can result in high-performance, while the ease of programming is enjoyed*[*].

## 1. Introduction

UPC, or unified parallel C, builds on the experience gained from its predecessor distributed shared memory C compilers such as Split-C[Cul93], AC[Car99], and PCP[Bro95]. UPC maintains the C philosophy of keeping the language concise, expressive, and by giving the programmer the power of getting closer to the hardware. UPC's simplicity and reliance on distributed shared memory contributed to its low overhead. 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. Consortium participants include NSA, IDA, GMU, ARSC, Compaq, CSC, Cray Inc., Etnus, HP, IBM, Intrepid Technologies, LBNL, LLNL, MTU, NSA, SGI, Sun Microsystems, UCB, 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. Furthermore, there are other implementations underway by vendors. The UPC specifications v1.0 were released in February 2001. UPC material is available on the UPC national web site (http://hpc.gmu.edu/~upc). It includes sample programs, compiler testing suite, tutorial [ElG00b], publications, and other. This paper is organized into 5 sections. Section 2 gives a brief overview of UPC, while section 3 presents UPC_Bench and its structure. The performance measurements from UPC_Bench are presented in section 4 for the synthetic part of UPC and in section 5 for the applications part of the suite. Section 6 closes with conclusions from this study.

## 2. An overview of UPC

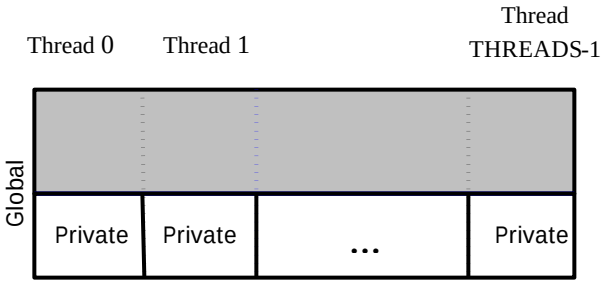



**Figure 1. The UPC Memory Sharing Model**

Figure 1 illustrates the underlying programming model of UPC. One or more threads work

[*] This work is sponsored by the Department of Defense under the LUCITE contract #MDA904-98-C-A081

independently. Memory under UPC is logically divided into a shared memory space and a private memory space. Each thread has a private memory space that can be accessed only by that thread. The shared memory space is logically partitioned into portions each of which has a logical association, or affinity to a given thread. The entire shared memory space, regardless of affinity, can be accessed by all threads.

Figure 2 shows an example of a short UPC program, vector addition. Using the "shared" type qualifier, arrays v1, v2 and v1plusv2 are declared as part of the shared memory space. The array elements of each of these arrays are distributed in a round robin fashion across the threads by default. The array data can be also partitioned by larger blocks across the threads. The UPC construct "upc_forall" explicitly distributes the iterations among the threads. "upc_forall" is similar to the "for" in C, except it has a fourth field. The fourth field in this case indicates that each thread will perform the computations associated with the array data that has affinity to that thread.

```
#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 Example**

## 3. UPC_Bench

While the DSM programming paradigm is quite effective in revealing memory locality in a distributed memory system, it is up to the compiler implementation whether to take full advantage of the underlying optimization opportunities. These optimizations can be classified into the following categories: 1) exploiting data locality through caching and prefetching of remote accesses, 2) aggregating remote accesses for overhead amortization, 3) recognizing thread-local shared data and accessing them with the same low overhead methods for dealing with private data, and 4) recognizing node-local shared data and accessing them with the same low overhead methods for dealing with private data. UPC_Bench is designed to highlight compiler deficiencies when such optimizations are not provided.

In order to reveal implementations performance problems, as outlined above, UPC_Bench contains a synthetic benchmark, UPC_Synthetic, as well as an application suite, UPC_Applications. UPC_Synthetic is focused on isolating which DSM-compiler-specific optimizations are or are not implemented. The application suite, UPC_Applications, currently has three applications selected to represent codes that have default remote access requests. These includes very small remote accessing requirements, or embarrassingly parallel applications, that have others moderate requirements, and applications that have high remote accessing demands.

### 3.1 The Synthetic Benchmark UPC_Synthetic

| | Private pointer | Shared pointer | Private pointer with indexes | Shared pointer with indexes |
|---|---|---|---|---|
| **Private memory** | ? | N/A | ? | N/A |
| **Local shared memory** | ? | ? | N/A | ? |
| **Remote shared memory** | N/A | ? | N/A | ? |

**Table 1. Combinations of Data Object Attributes and Ways of Accessing Them**

The synthetic benchmark tests all combinations of the ways of accessing data objects and the place of the data objects. Places of data considered are 1) private memory space, 2) local-shared space, and 3) remote-shared space. The ways of accessing data objects considered are 1) private-pointers, 2) shared-pointers, and 3) indexes. Table 1 shows these combinations. Invalid combinations are labeled N/A in the table entries. In addition, equivalence of table entries is also examined to remove redundant testing. This includes the entry in the second row and first column, private-pointer accessing shared-local, and the very first entry, private-pointer accessing private-memory. These two entries use local addresses, private pointers, and are essentially the same.

UPC_Synthetic uses these test cases along with some of the concepts from the stream benchmark [Cal95]. Thus, UPC_Synthetic measures the available memory bandwidth under selected combinations of Table 1, while performing different operations. These are copy, get, put and scale, as described later. The corresponding performance measurements are then used to understand how well a compiler recognizes and takes advantage of the potential low overhead in accessing local shared

objects. In order to reveal the ability of compilers to aggregate remote accesses, to amortize the associated overhead, another version of these tests using UPC block transfer functions to aggregate data objects is used to compare with single accesses. Since sequencing through large arrays tests single accesses, the results of such a benchmark depend also on prefetching, and performance will be higher if compilers are able to recognize such opportunities for prefetching and access aggregation and exploit them. Furthermore, the raw memory bandwidth obtained using memcpy is used as a performance reference point, as it represents the maximum sustainable memory read/write bandwidth available to applications. Thus, the benchmarking provided by the UPC_Synthetic assesses the effects and/or the cumulative effects of the first three of the aforementioned four optimizations, as well as their expected performance benefits for a given machine. In a UPC sense, the benchmark recognizes the presence of two types of shared memory, as follows.

**Local shared memory** indicates that the object of interest is in the shared space, but resides in the portion of the shared memory that has affinity to the local thread.

**Remote shared memory** indicates that the object in question is in a portion of the shared memory space that has affinity to another thread.

For measurement purposes, the remote locations can be determined based on affinity to threads and knowledge of how many CPUs or threads are there in each SMP node. The measurements collected here consider only thread local, but not node local cases.

## 3.2 The Benchmarking Application Suite UPC_Applications

The application benchmarking suite contains three applications, selected to represent a wide spectrum of remote memory access requirements. The applications include: (1) the Nqueens problem, which is an embarrassingly parallel problem requiring minimal remote memory accesses, (2) the Sobel edge detection, which requires remote memory accessing only for the guard zones, and (3) matrix multiplications, which is characterized by heavy remote memory access requirements, as compared to local accesses and processing.

**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, the west mask and the north mask, for detecting, respectively, horizontal and vertical edges. The masks extend on 3x3 pixels and associate a weight to each neighbor. 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 summed squares becomes the pixel value in the edge image. The west and the north masks are shifted along the rows and the process is repeated at each pixel.

In parallelizing this application, the image is partitioned into equal contiguous slices of rows that are distributed across the threads, as blocks of a shared array. A "upc_forall" is then used to get each thread to work mainly on the data that it has. 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, and limited to the elements of the next row.

**NQueens:** In the N Queens problem we seek to find all solutions to the problem of placing N queens on an NxN chessboard such that no queen can kill the other. This means that no two queens may be placed on the same row, column, or diagonal. The algorithm uses a depth-first searching and backtracking. For each row, we try to add a queen by checking the occupied columns and diagonals. When a queen is placed, we do the same for the next row. If no queen can be placed in a row, we go back and move the preceding queen. If N queens have been placed, we have obtained a solution. We save this solution and continue the algorithm. If we try to go back in the first row, it means we have reached the end.

The parallel solution to this problem is very straightforward. This is because the branches can be distributed across the threads with no thread interaction being needed. A job is described as searching along the subtree which corresponds to a given row position combination for the first L rows. All threads proceed to perform the sequential search along their own subtrees. No attempt to dynamically balance the workload has been made. The remote accesses associated with this algorithm are only the initial broadcast of the parallel job description parameters (3 integers) and the reduction of the total number of solutions found.

**Matrix multiplication:** Matrix multiplication is a very well known problem, which involves the multiplication of a matrix A by some matrix B to produce a result matrix, C. The computation of a C matrix element of the ith row and the jth column is performed by multiplying each element of the ith row of A by the corresponding element in the jth column of B, and accumulating the products. One interesting parallel solution is to distribute A by slices of contiguous rows and B by slices of contiguous columns. The C matrix

would also have the same distribution as matrix A. Each thread will then perform the computations associated with producing its share of the C elements. The problem with this scenario is the excessive remote memory transactions. Therefore, making local copies of matrix B in each of the threads can work better and it has been used in obtaining our benchmarking results.

Table 2 summarizes the applications selected, the size of the problems considered, and the remote accessing demands of these applications. Future releases of this benchmark will likely contain more applications and problem sizes. Intensity of remote accesses indicates whether the ratio of remote accesses to local accesses and processing is high, medium, or small.

| Application | Size of the problem | Intensity of Remote Accesses |
|---|---|---|
| N-Queens | 16 | small |
| Edge detection | 512x512 | medium |
| Matrix multiplication | (512x512) * (512x512) | high |

**Table 2. The UPC_Applications Suite**

### 3.3 Hardware And Compiler Testbed

All benchmarking in this paper has been performed on the AlphaServer SC. The Alphaserver SC has a NUMA architecture. It is based on the AlphaServer ES40, which is a SMP node with 4 Alpha 21264A processors. The nodes are interconnected through the quadrics switch (in a fat tree topology) from QSW in the U.K.. The hardware allows one-sided communication with limited software support. The Compaq implementation of UPC (Compaq UPC compiler v1.5) takes advantage of this particular communication layer when performing remote accesses. [Compaq99]

## 4. Synthetic benchmark measurements

This section uses UPC_synthetic, the synthetic part of UPC_Bench to study the behavior of the Compaq UPC v1.5 compiler, running on the Compaq AlphaServer SC. For the synthetic benchmarking experiments, the memory access rates are measured and presented in MB/s (1000000 bytes transferred per second). The higher the bandwidth, as it compares to memcpy in the private space, the better and more complete the compiler optimizations are. The number of bytes transferred per second is computed by multiplying the number of operations per second, times the number of bytes transferred by each operation. As described before, the effect of aggregating memory requests to amortize the associated overhead is also studied. The memory bandwidth is measured in these cases of aggregated and separate accesses in the private and shared space. In the case of the shared space, shared pointers are also cast to private when the addressed data is local in order to determine whether the compiler was able to recognize and exploit the fact that shared data might be local and does not require the same address processing needed for the remote shared data.

Table 3 shows the published relevant performance figures for the Compaq. They can help establishing the performance bounds of the observations obtained from the benchmark.

| | Local memory | Remote memory (using shmem) | |
|---|---|---|---|
| | Bandwidth | Bandwidth | latency |
| **Compaq AlphaServer SC** | 5.2 GB/sec | 200MB/sec | 3 us |

**Table 3. Published Memory Performance Rates**

While obtaining and studying such benchmarking results, one should remember that a sequential CC compiler and a UPC compiler could have different optimizations for single node compiling. For instance, the set operation on the Compaq was observed to be 28% slower with the CC compiler as compared with the UPC compiler, see Table 4.

Table 4 reports the memory copying performance rates using UPC_Synthetic for the Compaq AlphaServer SC. The memory bandwidth is compared among the cases of one sequential process (generated by the sequential C compiler) copying from one buffer to another, one UPC thread making memory copies within its private space, one UPC thread copying between the local and the remote shared space, and one UPC thread making copies from one buffer to another within the local shared memory space. Where it is applicable, the memory bandwidth is measured for the memcpy call as a reference point to establish the relative performance of copying aggregated cells, copying individual cells using array indexes and finally using pointers.

| MB/s | Memcpy | Struct cp | Copy (arr) | Copy (ptr) |
|---|---|---|---|---|
| **CC** | 711.0 | 768.0 | 564.0 | 564.0 |
| **UPC Private** | 711.0 | 800.0 | 711.0 | 564.0 |
| **UPC local** | | 331.0 | 56.0 | 66.0 |

| | | | | |
|---|---|---|---|---|
| **UPC remote** | | 342.0 | 1.3 | 1.3 |

**Table 4. Bandwidth of Different Memory Copying Operations**

From table 4, it is clear that in the case of the AlphaServer copying within the private space of a UPC thread has the same performance as copying in the space of a sequential cc process, as expected. Some differences are noted, but they are mainly due to the fact that the basic one node translation engine in the UPC version is different in nature from the CC compiler. Aggregating the accesses is producing a little over 10% improvement. Shared accesses that are local, however, have dramatically lower bandwidth, as compared to the bandwidth of copying in the private space. It is almost one order of magnitude in the case of single accesses, and more than 50% worse in the aggregated accesses.

| **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 |

**Table 5. Memory Bandwidth under Different Operations**

Table 5 compares the measured bandwidth, in a way similar to that of Table 4, but under different operations. In the case of the Compaq, it is noticed in this table that while copy and scale result in almost the same bandwidth in the case of private memory, this is not the case when shared memory is involved. This is because on the Compaq the copy from shared to shared is optimized by calling a one particular function instead of two separate calls to get and put. It should be also noted that aggregating transfers in case of private memory has a negative impact on the bandwidth since it only adds more work and unlike the case of remote accesses it does not amortize any overhead. Aggregating data still helps in the case of local shared, however, when compiler is not recognizing the difference between remote and local shared. Also, it should be noted that when shared accesses, local or remote, are aggregated they give the same performance. Compiler, or run-time, optimizations, however, can make the bandwidth of the local accesses comparable to that of the private ones.

## 5. Application measurements

The relative performance of UPC programs to sequential codes, compiled with cc for example, depend upon a number of factors. Many of these factors are due to compiler implementations and not the true potential of the UPC language or the underlying architecture. One of the chief objectives of UPC_Bench is to reveal such shortcomings and serve as a tool for compiler writers to demonstrate the opportunities for performance enhancements and assess their potential benefit.

### 5.1 Single Processor Measurements

As UPC itself is an extension of ANSI C, 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 when studying performance 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 use UPC_Bench to study the performance of the applications under 4 different compilation/execution scenarios as follows: 1) sequential code produced by the "CC"compiler running serially, 2) Sequential C program compiled with UPC compiler and running sequentially, and 3) UPC code compiled with UPC compiler and running with a single thread.

| **Time (sec.)** | **N=16** | **N=14** | **N=12** |
|---|---|---|---|
| **CC** | 182.25 | 4.17 | 0.12 |
| **C Code UPC Compiler** | 168.30 | 3.84 | 0.11 |
| **UPC /1 Thread** | 168.90 | 3.85 | 0.11 |

**Table 6. Sequential Nqueens Under Different Compiling/Execution Scenarios for the Compaq AlphaServer SC**

| **Time (sec.)** | **[512x512] * [512x512]** | **[1024x 512]*[512 x512]** | **[2048x 512]* [512x512]** |
|---|---|---|---|
| **CC** | 6.4 | 12.1 | 24.4 |
| **C Code UPC Compile r** | 7.4 | 13.1 | 27.2 |

| | | | |
|---|---|---|---|
| **UPC /1 Thread** | 289.1 | 568.3 | 1128.5 |

**Table 7: Sequential Matrix Multiplications Under Different Compiling/Execution Scenarios for the Compaq AlphaServer SC**

Table 6 shows that only small differences exist. The single node UPC compilation is clearly more efficient than the CC compiler.

Table 7 shows that for some unclear reason, at this point, matrix multiplications when written and compiled in UPC and ran with a single thread has a dramatically worse performance than any of the other sequential execution scenarios. Later on, it will be shown that this is because compilers may not realize that some of the data declared as shared are actually local to the thread and can be accessed with much less overhead. we will also apply some hand optimizations that can remedy this problem and demonstrate the potential performance benefits.

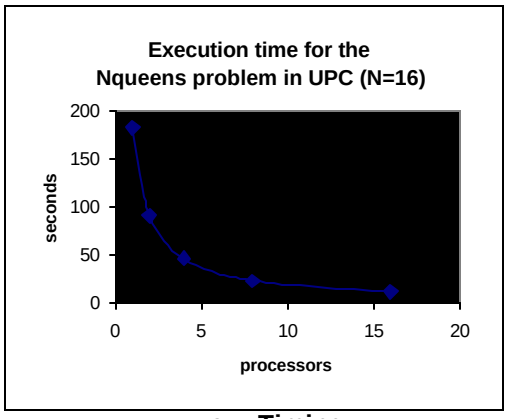
| **Time (sec.)** | **Configuration 1 256x256** | **Configuration 2 512x512** |
|---|---|---|
| **CC** | 0.011 | 0.046 |
| **UPC – C** | 0.009 | 0.037 |
| **UPC –1 thread** | 0.88 | 3.55 |

**Table 8. Sequential Edge Detection Under Different Compiling/Execution Scenarios for the Compaq AlphaServer SC**

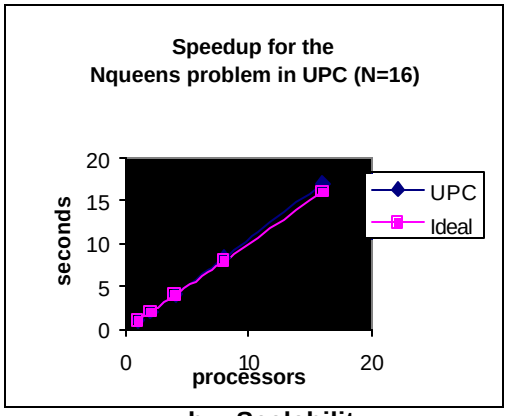
Table 8 shows the same kind of behavior for the edge detection problem. Less shared accesses, than in the case of the matrix multiplication, are needed in this case and therefore the penalty in the last case is much smaller.

For such shared local accesses, if the compiler discovers them they can be easily optimized and treated in such efficient way as private accesses are treated. Such optimizations can be also done at run time. However, in this case some tests to determine if the shared location being accessed is local to the processor will be needed. Then "move" is executed if the reference is local. The run-time testing, however would add some overhead.

## 5.2 Parallel Performance Measurements




a. Timing




b. Scalability

**Figure 3. Performance of the Nqueens on the Compaq AlphaServer**

The Nqueens problem as discussed earlier is an embarrassingly parallel problem. It uses very minimal shared data and has almost no remote access requirements. Therefore, it runs very efficiently on the Compaq and does in fact have superlinear speedup, as seen in figure 3.

Tables 7 and 8 indicate, however, that even if the UPC codes for matrix multiplication and for edge detection scale, they would not at all be able to compare well with their sequential code counterparts which defeats the whole purpose for parallel processing. However, as we already have seen in section 4.1, the reason for the poor performance is attributed to the fact that current UPC compilers are maturing and have not yet incorporated appropriate optimizations in an effective way. Therefore the following two sections will show the parallel performance when some of these optimizations that can be done automatically by compilers are introduced through hand tuning.

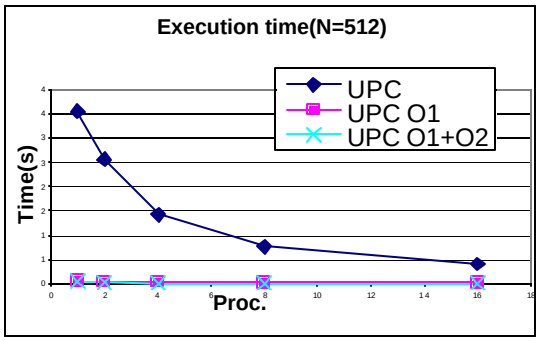
**5.2.1 Converting Local Shared Accesses To Private Accesses.** As noted earlier, shared memory accesses have significant overhead even if the sought

after locations are local to the accessing thread. The benefit from an optimization that treats such accesses as private can be determined if explicit casting of shared pointers to private pointers is used. For simplicity, we will refer to this optimization as optimization 1, or O1, in the rest of this paper.

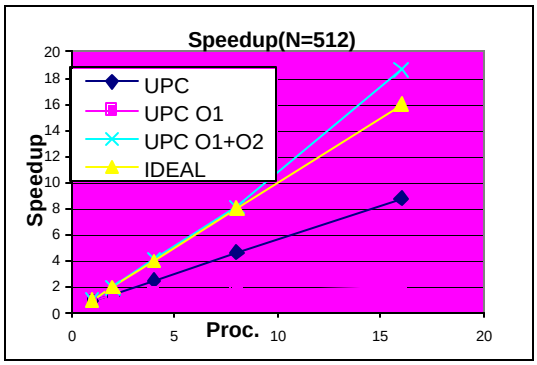
Some implementations may be using a runtime system, rather than compile time methods, to check whether the elements accessed are in the local memory and thus optimize at run time. This testing comes at the cost of adding processing overhead. Casting of shared pointers to private will be able to determine whether such overhead is insignificant or is rendering the run-time optimization useless.

**5.2.2 Prefetching And Aggregation Of Data Movements.** In a distributed shared memory programming it is possible for a program accessing a block of data through a loop, for example, to generate a large number of small requests each accessing one of the data elements. A good compiler can spot such opportunity of optimization by transferring such array as one buffer to the accessing thread. Thus, the transfer overhead of such a block of data can be amortized over all elements and most accesses to the elements will be available locally when a request is made. In order to uncover the potential benefit from such optimization, we aggregate data, transfer it using UPC block moves, to the thread that needs it, and access local shared data as private. The effect of this hand tuning shows the benefit from all optimizations including such aggregation and prefetching, as well as bypassing the shared memory overhead for local accesses as described earlier. For simplicity we will refer to aggregation and prefetching as optimizations 2, or O2.

## 5.3 Parallel Results For The Different Optimizations



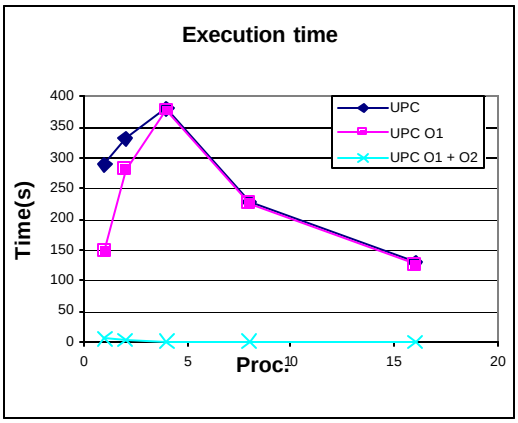

a. Execution time



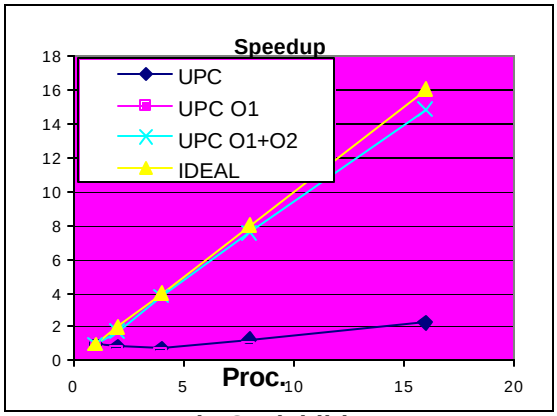

b. Scalability

**Figure 4. Performance of Edge detection on the Compaq AlphaServer SC**

Figure 4 shows the performance improvements obtained due to optimizations O1 and O2. It is clear from this figure the performance gain from these optimization, that the benefit from O1 is constant while the benefit from O2 grows as the number of threads grow, which as we pointed earlier magnifies the penalty for remote accesses.




a. Execution Time




b. Scalabiltiy

**Figure 5. Performance of UPC for Matrix Multiplication**

Figure 4, however, makes a very important point, that is when all pointed optimizations are incorporated into UPC compilers, the UPC application codes used here, can potentially show a superlinear speedup.

Figure 5 demonstrates the benefit from the optimizations and shows the performance of optimized UPC for matrix multiplication. This figure simply affirms the observations of figure 4. Once again, O1 has a constant benefit while, the benefit from O2 increases with the increase of the number of threads.

### 6. Conclusions

UPC_Bench, in spite of its simple synthetic codes and applications, can reveal very critical facts for the development of the UPC language. One very important fact is that UPC is a powerful programming language and can execute very efficiently, and thus compare favorably against other paradigms for many applications. UPC_Bench, however, is showing that in order to maintain programming simplicity while performing well, UPC compilers must embody a number of optimizations. The opportunities for optimizations arise from four scenarios in distributed shared memory programming paradigms such as UPC. These four optimizations are 1) exploiting data locality through caching and prefetching remote accesses, 2) aggregating remote accesses for overhead amortization, 3) recognizing thread-local shared data and accessing it with the same low overhead methods for dealing with private data, and 4) recognizing node-local shared data and accessing it with the same low overhead methods for dealing with private data.

Recognizing local shared data and dealing with it with the same low overhead as private has a constant improvement on performance. The amount of improvement will clearly depend on the ratio between remote and local accesses. Aggregating and prefetching of block remote accesses, however, has a much larger positive impact on performance and the benefit from it grows as the number of threads grow.

Providing such optimizations as early as possible in UPC compiler releases, more support for UPC is likely to mount quickly and the user community can grow very rapidly. The lack or delay of such optimizations will on the other hand will likely mask out the great intrinsic power that the UPC model has.

## 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 session at *Supercomputing '9*5, San Diego, CA, December 3-8, 1995.

[Car99] William W. Carlson, Jesse M. Draper. 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'9*3, Portland, OR, November 15-19, 1993, pp. 262-273.

[ElG00a] Tarek A. El-Ghazawi, William W. Carlson, Jesse M. Draper. UPC Language Specifications V1.0 (http://hpc.gmu.edu/~upc). December, 2000.

[ElG00b] Tarek A. El-Ghazawi, Programming in UPC (http://hpc.gmu.edu/~upc/tutorial). December, 2000.

[ElG00c] Tarek A. El-Ghazawi, Sebastien Chauvin UPC Benchmarking and Performance Issues , unpublished manuscript.

[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.htm) .