DEVELOPING AN OPTIMIZED UPC COMPILER FOR FUTURE ARCHITECTURES

Tarek El-Ghazawi, François Cantonnet, Yiyi Yao
Department of Electrical and Computer Engineering
The George Washington University
tarek@gwu.edu

Ram Rajamony
IBM Austin Research Lab
rajamony@us.ibm.com

Abstract

UPC, or Unified Parallel C, has been gaining rising attention as a promising productive parallel programming language that can lead to shorter time-to-solution. UPC enables application developers to exploit both parallelism and data locality, while enjoying a simple and easy to use syntax. This paper examines some of the early compilers that have been developed for UPC on many different platforms. Using these developments and previous experiments, the paper abstracts a number of considerations that must be observed in new UPC compiler developments, such as for the future IBM PERCS[1] architecture, in order to achieve high-performance without affecting the ease-of-use. In addition to those general considerations, the paper also examines some of the interesting features of the future architectures that can be exploited for running UPC more efficiently.

1. INTRODUCTION

A Quick UPC Overview

Figure 1 illustrates the memory and execution model as viewed by UPC applications and programmers [ELGH03]. Under UPC, application memory consists of two separate spaces: a shared memory space and a private memory space. A number of independent threads work in a single-program-multiple-data (SPMD) fashion, where 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. In order to express locality in the shared space, the shared space is logically partitioned and each partition has affinity to a given thread. Programmers can then, with proper declarations, keep the shared data that is to be mainly manipulated by a given thread allocated to the shared partition of that thread. Thus, a thread and its pertinent data can likely be mapped by into the same physical node.

[1] PERCS is supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

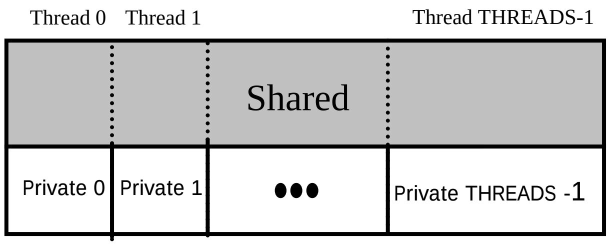

Figure 1. The UPC Memory and Execution Model

Since UPC is an explicit parallel extension of ISO C, all features of C are a subset of those of 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 concepts for pointers to both private and shared memory. There is generally no implicit synchronization in UPC. Therefore, the language offers a broad range of synchronization and memory consistency control constructs. Among the most interesting synchronization concepts is the non-blocking barrier, which allows overlapping local computations with synchronization. A collective operations library is now integrated into the language specifications. Parallel I/O library specifications have been recently developed and will be soon integrated into the formal UPC language specifications.

UPC language has many features that make it easy to use for programmers [ELGH03]. It is a simple extension to the well known C specification and keeps, and extensions are fairly small and straightforward. Due to its shared space model, a remote write can amount to assigning a value to a remote shared object, while a remote read can be caused by evaluating an expression that contains a remote shared object. UPC is not implemented on top libraries, which eliminates many function calls and argument passing.

Thus, UPC inherits the conciseness of C, and Table 1 shows the lines of codes needed for converting from the sequential code to the parallel code, using both UPC and MPI.

		SEQ	**UPC**	**SEQ**	**MPI**	**UPC Effort (%)**	**MPI Effort (%)**
NPB-CG	#line	665	710	506	1046	6.77	106.72
	#char	16145	17200	16485	37501	6.53	127.49
NPB-EP	#line	127	183	130	181	44.09	36.23
	#char	2868	4117	4741	6567	43.55	38.52
NPB-FT	#line	575	1018	665	1278	77.04	92.18
	#char	13090	21672	22188	44348	65.56	99.87
NPB-IS	#line	353	528	353	627	49.58	77.62
	#char	7273	13114	7273	13324	80.31	83.20

NPB-MG	#line	610	866	885	1613	41.97	82.26
	#char	14830	21990	27129	50497	48.28	86.14

Table 1: Manual Effort of parallelization

The last two columns of the table show the effort in converting from the sequential code to the parallel code. It is calculated as (LOC(Parallel) - LOC(seq)) / (LOC(seq)).

As it is shown in the table, it takes much less effort to parallelize a piece of code using UPC than MPI.

Background on PERCS and UPC

The DARPA HPCS program was launched in mid 2002 and is aimed at engendering a new class of supercomputers by 2010 that is characterized by high-productivity. Thus, instead of focusing only on speed of execution (e.g. Time or FLOPS), this program is concerned with the development time of applications as well, and aims at significantly reducing the overall time-to-solution. IBM is researching a future system architecture called PERCS under the auspices of the DARPA HPCS program. PERCS is using innovative techniques to address lengthening memory and communication latencies.

A number of UPC compiler implementations and have become available over the past few years. Many of these early systems focused on complying with the language specifications, with less emphasis on the need for incorporating automatic compiler optimizations. In this paper cite possible optimizations and their potential benefit, based on our studies on various architectures like Distributed Memory systems (e.g. clusters), Distributed Shared Memory systems (NUMA machines) and advanced Vector systems. Section 2 presents some of our experimental investigations of UPC compiler implementations on distributed memory systems, and abstracts the lessons learned. Section 3 presents similar investigations conducted and results on distributed shared memory machines and vector systems. Section 4 uses the lessons learned from the studied compilers to highlight the most important issues for new developments of UPC compilers for future architectures. Section 5 provides conclusions and directions for future works.

2. UPC COMPILERS ON DISTRIBUTE MEMORY SYSTEMS

Low-Level Library Implementations

UPC initially did not include any standard libraries. Recently a collective library has been incorporated and a parallel I/O library has been drafted. In the absence of such libraries or low level implementations of them, the programmer may resort to user level implementations of such functionalities. An early study has shown that such user level implementations can result in significant performance degradations [ELG02] especially for those architectures such as clusters. Vendor optimized libraries, however, can take advantage of hardware features and optimize, e.g., the collective communications library at the interconnection fabric level, as it is the case in good MPI implementations.

Address Translation and Local Space Privatization

MB/s	**Put**	**Get**	**Scale**	**Sum**
CC	N/A	N/A	1565.039	5409.299
UPC Private	N/A	N/A	1687.627	1776.81
UPC Local	1196.513	1082.881	54.2158	82.6978
UPC Remote	241.4293	237.5119	0.0882	0.1572
MB/s	**Copy (arr)**	**Copy (ptr)**	**Memcpy**	**Memset**
CC	1340.989	1488.017	1223.864	2401.264
UPC Private	1383.574	433.4484	1252.465	2352.707
UPC Local	47.2	90.6728	1202.795	2398.904
UPC Remote	0.088	0.2003	1197.219	2360.586

Table 2. UPC Stream Bandwidth Measurements (in MB/sec)

UPC has a rich memory model designed to enhance locality exploitation while providing the ability to use different data structures. Accesses under the UPC memory model need to be translated into the underlying virtual address space. In some compiler implementations this can be costly to the point that makes an access into a thread's private space orders of magnitude faster than accesses into the thread's local shared space, even though both spaces can be mapped into the same physical memory. 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, gathered from a cluster machine, 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 demonstrates the value of the affinity concept in UPC which helps localizing the accesses. However, these local shared 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 then treated as private, thereby avoiding unnecessary processing for general shared memory operations, as well as interconnect interaction. Table 2 shows that in this compiler implementation, sequential C is faster than private UPC access. This needs not to be the case and is seen only in some of the compilers. It is also observed when the back end of the UPC compiler is not using the same sequential compiler available on a given machine. Table 2 is quite dramatic and more recent developments in UPC compilers have improved upon the relative behaviors depicted by this table. The table is therefore showing some of the pitfalls for UPC compilers, many of which are somewhat avoided in recent releases.

An important emulation of effective access to local shared data is privatization, which makes local shared objects appear as if they were private. This emulation is implemented by casting a local shared pointer to a private pointer and using the private pointer for accessing the local shared objects as if they were private. 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 its local shared space. Such optimization can be incorporated automatically through a source to source translator that makes such conversion based on a simple affinity analysis to discover what the local shared data are.

Aggregation and Overlapping of Remote Shared Memory Accesses

UPC is a locality conscious programming language. Affinity analysis at compile time may reveal what remote accesses will be needed and when. A typical scenario is ghost zones, where affinity analysis can tell that a contiguous chunk of data with affinity to another thread will be needed. Therefore, one other optimization is aggregating and prefetching remote shared accesses, which as shown in Table 2 can provide substantial performance benefits. This can be established through block moves, using upc_memget and upc_memput. Also, one can overlap local computations with remote accesses, using split-phase barriers, to hide synchronization cost. All these optimizations can conveniently be done at the application level. Most of them are in the interim used as a way to hand tune UPC applications. However, they can be captured as source to source automatic compiler optimizations.

The impact of these optimizations is greatly application dependent. To illustrate this, we consider the N-Queens and the Sobel Edge Detection Algorithm [ELG01], on a Distributed Shared Memory model, NUMA architecture.

The N-Queens problem does not get any benefit from such optimizations. This is due to the inherent nature of N-Queens as an embarrassingly parallel application which does not require significant shared data. However, it is important to note that N-Queens and embarrassingly parallel applications that do not require extensive use of shared data will not be adversely affected by such optimizations.

The Sobel Edge Detection performance on the NUMA machine is presented in Figure 3a. The image size is set to 2048x2048. This case illustrates the importance of these optimizations. The execution time is almost 9 times faster after having privatized all local shared memory accesses and prefetched the remote data lines to be accessed.

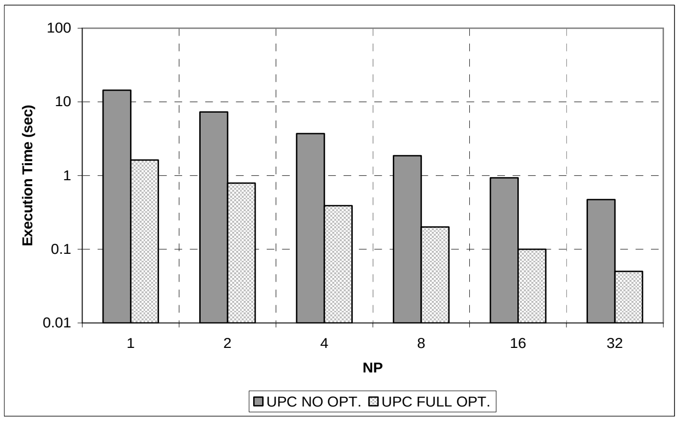

Figure 3a. Execution Time

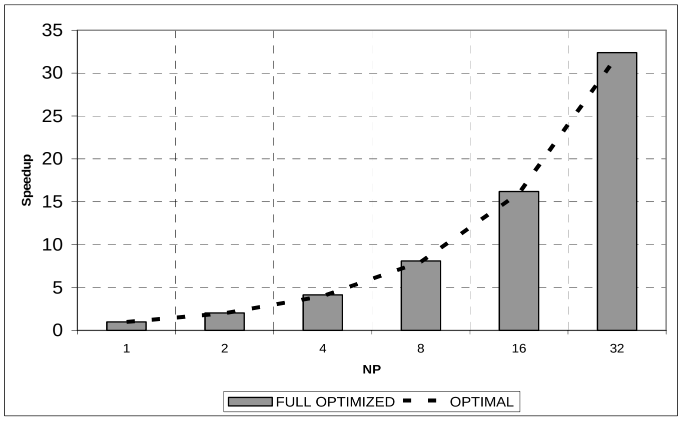

Figure 3b. Scalability

Figure 3. Impact of UPC Optimizations on Performance for Sobel Edge Detection

Figure 3b 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 almost one order of magnitude in some cases.

Low level communication optimizations
For distributed memory machines, an efficient communication layer will be important in delivering a good performance for global addressable language like UPC. Low level communication optimizations like Communication and Computation Overlap and Message Coalescing and Aggregation have been shown in the GASNET interface [CHEN03]. Such optimizations were shown to be helpful in reducing communication overhead of UPC.

3. UPC OPTIMIZATIONS AND MEASUREMENTS ON DISTRIBUTED SHARED MEMORY SYSTEMS

Shared Address Translation
In order to confirm the belief that the memory model translation to virtual space can be quite costly, even on platforms that support shared memory model, an investigation was conducted to quantify this overhead experimentally on a distributed shared memory model NUMA machine. Figure 4 illustrates the different overhead sources for any shared access, namely function calls to get and put, that reside in the UPC Runtime system, as well as the virtual address translation. Studies have shown that a single local shared read causes about 20 memory reads and 6 memory writes [CANT03], as detected by a low-level profiling facility on the architecture.

To quantify the amount of time spent in each step X, Y and Z as per Figure 4, a small program which performs an element by element array set, in which either a shared or private array is initialized, is used. Using this program, the time to set a complete private array was measured and the average memory access time Z was computed. The time X was estimated by introducing a dummy function in the UPC runtime system and calling it. The address calculation overhead Y was deduced from the overall execution time X+Y+Z while accessing a local shared array. The corresponding measurements are given in Figure 5.

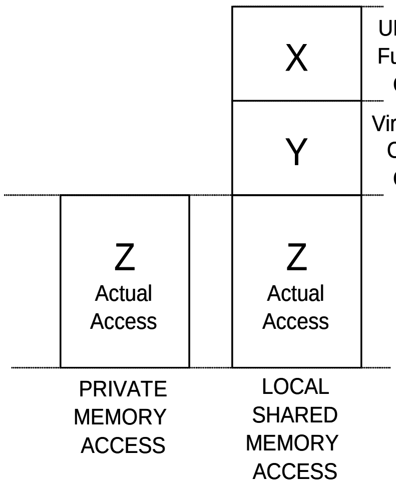
X
UPC Put/Get Function Call Overhead
Y
Virtual Address Calculation Overhead
Address Translation Overhead
Z Actual Access
Z Actual Access
PRIVATE MEMORY ACCESS
LOCAL SHARED MEMORY ACCESS

Figure 4. Overheads Present in Local-Shared Memory Accesses

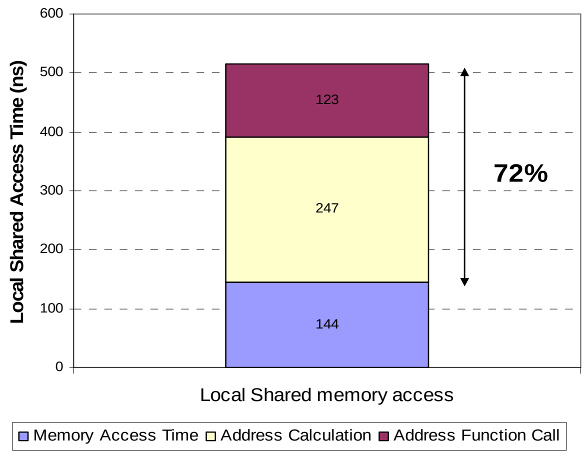

Figure 5. Quantification of the Address Translation Overheads

From this figure, it is clear that the address translation overhead is quite significant as it added more than 70% of overhead work in this case. This also demonstrates the real need for optimization.

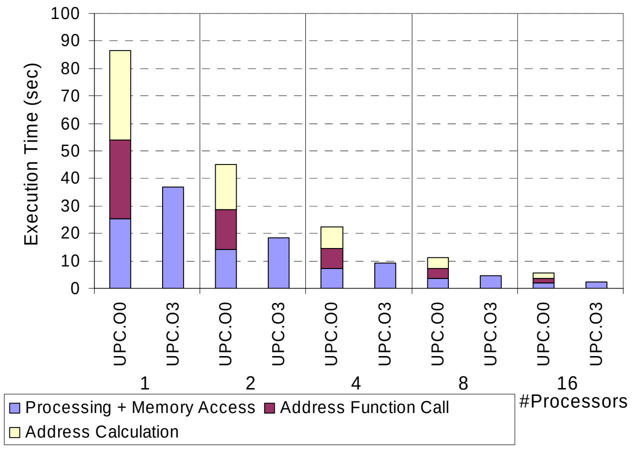

Figure 6. Performance of the Regular UPC and Optimized UPC for Sobel Edge Detection

Figure 6 illustrates the amount of time spent in computation as well as in the address translation for the Sobel Edge Detection kernel. Two UPC codes are compared, the unoptimized UPC code, referred as UPC.O0, and a hand-optimized version of the same code, called UPC.O3. The Ox notations accounts for the amount of optimizations emulated manually in the code [ELG01]. O0 denotes a 'plain' UPC code, in which no hand-optimization have been done, whereas O3 denotes a hand-optimized UPC code, in which privatization, aggregation of remote accesses as well as prefetching have been manually implemented. It can be seen in the case of Sobel that space-privatization lowers significantly the execution time, by roughly a factor of 3 when comparing the execution time of UPC.O0 to UPC.O3. One important remark is that in Sobel Edge, the address translation overhead is taking 2/3 of the total execution time. Other applications have shown even bigger overhead.

UPC Work-sharing Construct Optimizations

UPC features a work-sharing construct, *upc_forall*, that can be easily used to distribute independent loop body across threads, figure 7. It has been shown that *upc_forall* loops may have a considerable overhead over the regular *for* loops if not translated carefully [CANT03].

By thread/index number
```
upc_forall(i=0; i<N; i++; i)
     loop body;
```
By the address of a shared variable
```
upc_forall(i=0; i<N; i++; &shared_var[i])
     loop body;
```

Figure 7. upc_forall constructs

A naïve translation of *upc_forall* constructs, presented in Figure 7, is illustrated in Figure 8.

By thread/index number
```
for(i=0; i<N; i++)
{
   if(MYTHREAD == i%THREADS)
        loop body;
}
```

By the address of a shared variable
```
for(i=0; i<N; i++)
{
   if(upc_threadof(&shared_var[i]) == MYTHREAD)
        loop body;
}
```

Figure 8. Equivalent *for* loops for *upc_forall* constructs

All the threads will go through the loop header N times, although each thread just executes N/THREADS iterations. High quality compiler implementations should be able to avoid these extra evaluations. Evaluating the affinity field will introduce additional overhead. The performance will be improved if the compiler can recognize the *upc_forall* constructs and transform them accordingly into "overhead-free" *for* loops. Figure 9 illustrates the measured performance of the all five variants: two *upc_forall* constructs with their equivalent *for* loops and the "overhead-free" *for* loop. All variants seem to scale, however, the optimized *for* loop delivers the best performance and scalability among all.

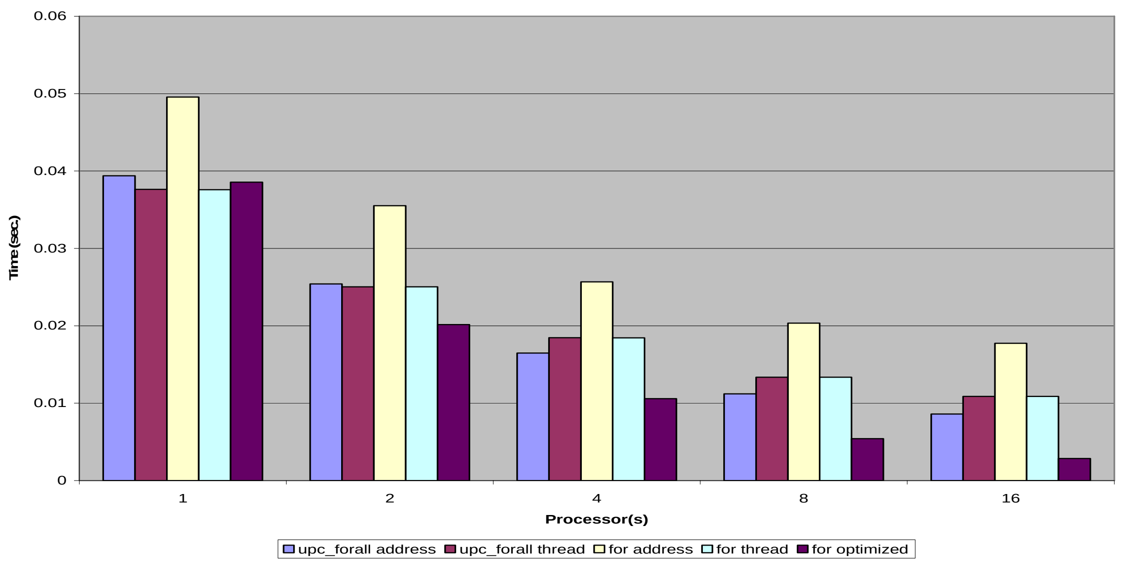

Figure 9. Performance of equivalent upc_forall and for loops

To illustrate the possible impact of optimized *upc_forall* implementations on applications, a sparse-matrix by dense vector multiplication problem is considered. Figure 10 illustrates the performance measured using a regular *upc_forall* (using a shared address as affinity argument) compared to the performance of an equivalent optimized *for* loop. The optimized *for* loop implementation is constantly delivering better performance.

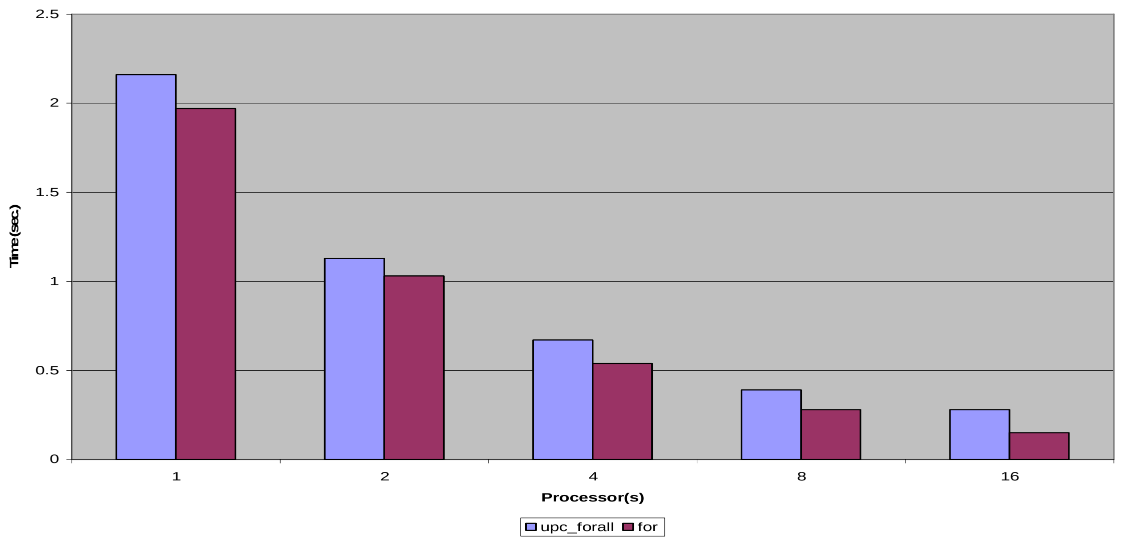

Figure 10. Sparse matrix by dense vector multiplication (N=1M 32-bit integer elements)

Performance Limitations Imposed by Sequential C Compilers
As UPC is simply an extension of ISO C, UPC compilers typically use a sequential C compiler backend. Thus, the basic performance of UPC is tightly coupled to the performance delivered by the sequential C compiler. Recent studies [ElGH05] over the distributed shared memory NUMA machines, clusters, and Scalar/vector architectures have shown that the sequential performance between C and Fortran can greatly differ. In general, the vector supercomputing era has resulted in a wealth of work in optimizing Fortran compilers, specially for scientific codes, where pipelining vecotrization can work very well. Table 3 summarizes the performance of the STREAM benchmark in both C and Fortran. The Fortran version, as well as the C, are based on the original STREAM. Every effort has been exerted to make them both very similar in order to study the differences in C and Fortran compilation. The measurements show that sequential C is generally performing roughly two times slower than Fortran.

NUMA	BULK			Element-by-Element						
	memcpy	memset	Struct cp	Copy (arr)	Copy (ptr)	Set	Sum	Scale	Add	Triad
F	291.21	163.90	N/A	291.59	N/A	159.68	135.37	246.3	235.1	303.82
C	231.20	214.62	158.86	120.57	152.77	147.70	298.38	133.4	13.86	20.71

Vector	BULK			Element-by-Element						
	memcpy	memset	Struct cp	Copy (arr)	Copy (ptr)	Set	Sum	Scale	Add	Triad
F	14423	11051	N/A	14407	N/A	11015	17837	14423	10715	16053
C	18850	5307	7882	7972	7969	10576	18260	7865	3874	5824

Table 3 Fortran VS C STREAM Bandwidths Performances (expressed in MB/sec)

In trying to understand the reason why these discrepancies in the STREAM micro-benchmark, a closer look into the difference of how the Fortran and C compiler are dealing with certain constructs become necessary. Especially for the scalar/vector architecture in which the vectorized codes usually perform much faster than those executed only in the scalar units.

Thus, using the code analysis utility provided on the target scalar/vector architecture, loopmark listing can be used to reveal how loops were optimized. This has in general revealed to us that Fortran was able to achieve more vectorization and parallelism (multistreaming) as compared to C. However, with the use of proper *pragma*(s), similar levels of performance were obtained. In general, more work on C optimizations including vectorizing C compilers is needed. In the interim, more analysis might be required by the UPC front-ends to make up for that.

4. LESSONS LEARNED AND PERCS UPC

Many UPC optimizations are applicable to different types of architectures. UPC implementation on future architectures such as PERCS must take advantage of the performance studies on other architectures. In general there are four types of optimizations that are needed: 1. Optimizations to Exploit the Locality Consciousness and other Unique Features of UPC, 2. Optimizations to Keep the Overhead of UPC low, 3. Optimizations to Exploit Architectural Features, and 4. Standard Optimizations that are Applicable to all Systems Compilers. Specifics of these optimizations have been already covered in the previous sections. These optimizations can be implemented at three levels. Improvement to the C backend compilers to compete with Fortran, incorporating most UPC specific optimizations into a source to source compilation phase, creating a strong run-time system that can work effectively with the K42 Operating System, and ensuring that K42 is able to provide the language run-time system with the control it needs. The run-time system will be extremely important in future architectures due to its control over the system resources.

5. SUMMARY AND CONCLUSIONS

UPC is a locality-aware parallel programming language. It was shown that with proper optimizations, in previous studies, that UPC can outperform MPI in random short accesses and can otherwise perform as good as MPI. UPC is very productive and UPC applications result in much smaller and more readable code than MPI. UPC compiler optimizations are still lagging, in spite of the fact that substantial progress has been made on other non-IBM machines. For future architectures such as PERCS, UPC has the unique opportunity of have very efficient UPC implementations as most of the pitfalls and obstacles are now revealed along with adequate solutions. More work on how to match the specific features of PERCS with those of UPC is needed, specially when more information about PERCS become available.

References

[CANT03] F. Cantonnet, Y. Yao, S. Annareddy, A.S.Mohamed and T. El-Ghazawi, "Performance Monitoring and Evaluation of a UPC Implementation on a NUMA Architecture", IPDPS 2003 PMEO workshop, Nice, April 2003.

[CANT04] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi**. "**Productivity Analysis of the UPC Language", IPDPS 2004 PMEO workshop, Santa Fe, April 2004.

[CANT05] François Cantonnet, Tarek A. El-Ghazawi, Pascal Lorenz, Jaafer Gaber Fast Address Translation Techniques for Distributed Shared Memory Compilers, IPDPS 05, Denver, May 2005.

[ELG01] T. El-Ghazawi, S. Chauvin, "UPC Benchmarking Issues", Proceedings of the 2001 International Conference on Parallel Processing, Valencia, September 2001.

[ELG02] T. El-Ghazawi and F. Cantonnet. "UPC performance and potential: A NPB experimental study". Supercomputing 2002 (SC2002), Baltimore, November 2002.

[ELGH03] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. "UPC language specifications" v1.1.1, October 2003. http://www.gwu.edu/upc/documentation.html.

[ELGH05] T. El-Ghazawi, F. Cantonnet, and Y. Yao. Unpublished Work, January 2005.

[CHEN03] W. Chen, D. Bonachea, J. Duell, P. Husbands, C Iancu, K. Yelick, "Performance Analysis of the Berkeley UPC Compiler", ICS 2003, June 2003.