Evaluation of UPC on the Cray X1

Tarek A. El-Ghazawi, François Cantonnet, Yiyi Yao,
Department of Electrical and Computer Engineering, The George Washington University and **Jeffrey Vetter**,
Computer Science and Mathematics Division, Oak Ridge National Laboratory

ABSTRACT: *UPC is parallel programming language which enables programmers to expose parallelism and data locality in applications with an efficient syntax. Recently, UPC has been gaining attention from vendors and users as an alternative programming model for distributed memory applications. Therefore, it is important to understand how such a potentially powerful language interacts with one of today's most powerful, contemporary architectures: the Cray X1. In this paper, we evaluate UPC on the Cray X1 and examine how the compiler exploits the important features of this architecture including the use of the vector processors and multi-streaming. Our experimental results on several benchmarks, such as STREAM, RandomAccess, and selected workloads from the NAS Parallel Benchmark suite, show that UPC can provide a high-performance, scalable programming model, and we show users how to leverage the power of X1 for their applications. However, we have also identified areas where compiler analysis can be more aggressive and potential performance caveats.*

KEYWORDS: Cray X1, Unified Parallel C, Distributed Shared Memory paradigm, programming model, STREAM, RandomAccess, NAS Parallel Benchmark, vector, multistreaming

1. Introduction

Unified Parallel C (UPC) is an explicit parallel programming language extension of ISO C based on the partitioned global address space (PGAS) programming model[ELG03]. UPC leverages the work done on may predecessor efforts, such as Split-C[CUL93], AC[CAR99], and PCP[BRO95], as well as the direct input of a consortium of vendors, researchers, and practitioners. Like C itself, UPC has an efficient syntax and provides the programmer low-level access to the underlying system and architecture from the perspective of an abstract high-level language. UPC allows programmers to manage data distributions explicitly. Thus, like MPI and message passing paradigms, in general, UPC allows application developers to co-locate processing potentially in the same node and avoid unnecessary overhead. UPC provides a global address space view, therefore, like shared memory paradigms such as OpenMP[OPE99], UPC can hide much of the complexity of private and shared memory. In UPC, a simple assignment statement can cause a remote memory read and a remote memory write, which hides much of the underlying data movement from the application developer.

The Cray X1 combines the globally-addressable, distributed shared memory architecture with vector and traditional processing capabilities. In this study, we examine Cray X1 UPC on several important benchmarks including STREAM, RandomAccess, and selected workloads from the NAS Parallel Benchmark suite. We study the behavior of these workloads in response to automatic compiler optimizations as well as to our devised emulations that can mimic the effects of automatic optimizations [ELG02]. We also examine compiler output to determine how well the compiler takes advantage of the architectural features of the Cray X1 and characterize where improvements may still possible.

This paper is organized as follows. Section 2 gives a brief description of the UPC language, while section 3 introduces the Cray X1 architecture. Next, Section 4 discusses the experimental testbed and workloads used. Section 5 presents the performance measurements of UPC, while Sections 6 and 7 highlight further improvement capabilities, followed by conclusions in section 8

2. Overview of UPC

Application memory consists of two separate spaces in UPC: a shared memory space and a private memory space. Figure 1 illustrates the memory and execution model as viewed by UPC applications and programmers. A number of independent threads work in a single-program-multiple-data (SPMD) fashion, where each of them can reference any address in both the shared space and its own private space, but not in other thread's private spaces. 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. Programmers can then, with proper declarations, keep the shared data that is to be mainly manipulated by a given thread (and less frequently accessed by others) associated with that thread. Thus, a thread and its pertinent data can likely be mapped by the system into the same physical node. This can clearly exploit inherent data locality in applications.

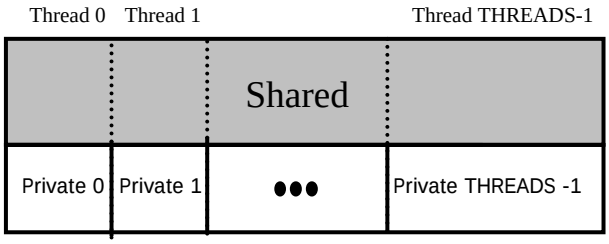

Figure 1. The UPC Memory and Execution Model

Since UPC is an explicit parallel extension of ISO C, all language features of C are already embodied in UPC. In addition, UPC declarations give the programmer control of the distribution of data across the threads. Among the interesting and complementary UPC features is a work-sharing iteration statement, known as `upc_forall`. This statement helps to distribute independent loop iterations across the threads, such that iterations and data that are processed by them are assigned to the same thread. UPC also defines a number of rich concepts for pointers to both private and shared memory. Additionally, UPC supports dynamic shared memory allocations. 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 and inter-thread communications. Parallel I/O and collective operation libraries specifications have been developed and are to be integrated into the next UPC language specifications.

3. Cray X1 Overview

The Cray X1 is an attempt to incorporate the best aspects of previous Cray vector systems and massively parallel processing (MPP) systems into one design. The design of the X1 is hierarchical in processor, memory, and interconnect. The basic component in this hierarchy is the multi-streaming processor (MSP), which is capable of 12.8 GF/s for 64-bit operations. Each MSP, shown in Figure 2, is comprised of four single-streaming processors (SSPs), each with two 32-stage 64-bit floating-point vector units and one 2-way super-scalar unit[CRA04]. The SSP uses one clock frequency of 800 MHz for the vector units and another clock frequency of 400 MHz for the scalar unit. Each SSP is capable of 3.2 GF/s for 64-bit operations. These four SSPs share a 2 MB "Ecache"

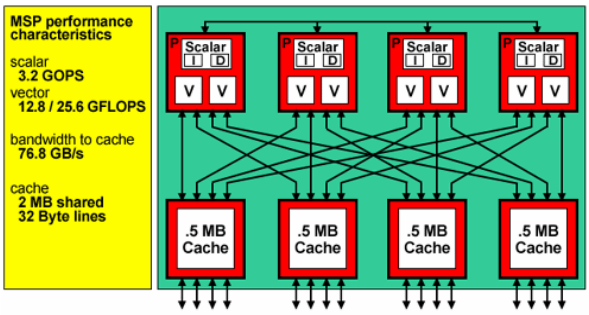

Figure 2. Cray MSP Module

Four MSPs and a flat, shared memory of 16 GB form a Cray X1 node. The memory banks of a node provide 200 GB/s of bandwidth, enough to saturate the paths to the local MSPs and service requests from remote MSPs. Each bank of shared memory is connected to a number of banks on remote nodes, with an aggregate bandwidth of roughly 50 GByte/sec between nodes. This represents one byte per flop of interconnect bandwidth per computation rate. The collected nodes of an X1 have a single system image.

The Cray X1 nodes are connected using X1 routing modules. Each node has 32 1.6 GBs full duplex links. Each memory module has an even and odd 64-bit (data) link forming a plane with the corresponding memory modules on neighbouring nodes. The local memory bandwidth is 200 GB/s, enough to service both local and remote memory requests. A 4-node X1 can be connected directly via the memory modules links. With 8 or fewer

cabinets (up to 128 nodes or 512 MSPs), the interconnect topology is a 4-D hypercube. However, larger configurations use a modified 2D torus.

A single four-MSP X1 node behaves like a traditional SMP. Like the T3E, each processor has the additional capability of directly addressing memory on any other node. Different, however, is the fact that these remote memory accesses are issued directly from the processors as load and store instructions, going transparently over the X1 interconnect to the target processor, bypassing the local cache. This mechanism is more scalable than traditional shared memory, but it is not appropriate for shared-memory programming models, like OpenMP, outside of a given four-MSP node. In contrast to the T3E's E-registers, both scalar and vector loads are blocking primitives, which limits the ability of the system to overlap communication and computation. This remote memory access mechanism is a natural match for distributed-memory programming models, particularly those using one-sided put/get operations, such as UPC and Co-Array Fortran.

4. Applications and Testbed

The performance of the UPC language over the X1 is studied over benchmarking workloads. These workloads included the STREAM benchmark, Random Memory Accesses benchmarks, and several selected benchmarks from the NAS Parallel Benchmark (NPB) suite. In addition, measurements of MPI [SNI98] implementations are given as performance references. These workloads can be downloaded from the UPC website [UPC] (http://upc.gwu.edu), except the NPB MPI suite, which is available from the official NAS website [NPB02]. All the performance measurements have been collected under the Cray X1 Programming Environment 5.2.

4.1 UPC Stream Benchmarks

The STREAM benchmark [MCC95] [ELG01] 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 were extracted from the STREAM benchmark (e.g. Bulk as well as element-by-element operations as shown in Table 1) and extended for UPC, focusing on the UPC local shared and remote shared as well as private memory accesses. For these synthetic benchmarking experiments, the memory access rates are measured and presented in MB/s. The higher the bandwidth, the better and more complete are the compiler optimizations.

	Operation	Detail
BULK Operations	Memcpy	
	Memset	
	Structure-copy	
	Get	
	Put	
Element-by-Element Operations	Copy (array)	c[j] = a[j]
	Copy (pointer)	*(ptr_a++) = *(ptr_c++)
	Set	b[j] = (set++)
	Sum	sum += a[j]
	Scale	c[j] = scalar * a[j]
	Add	a[j] = b[j] + c[j]
	Triad	b[j] = a[j] + scalar * c[j]

Table 1. Operations of STREAM benchmark

4.2 RandomAccess Microbenchmark

Unlike the regular accesses of Stream, the Random Access micro-benchmark is a program performing updates to random locations in a large shared array. It is used to evaluate the machine's capability of performing CPU-to-memory transactions. It is included as part of HPC Challenge Benchmark suite [HPC04]. In the UPC version of Random Access, the main table is located in the global accessible shared memory space which provides a similar programming view as the sequential algorithm. Random numbers are generated on-the-fly by each thread and are used to access and update the according elements of the main table. These random numbers are use to compute indices of the main table to be updated. This is similar to the GigaUpdates per Second, or GUPS, micro-benchmark [CAN03]. The performance of UPC will be compared to available MPI-1 codes as well as one implementation in MPI-2. Three different MPI-1 implementations are considered: using bulk transfer, partially using asynchronous operations and fully using asynchronous operations. The bulk transfer offers typically the best performance but it is not in accordance with the fine granularity nature of the problem statement. The two other implementations are based on element-by-element transfers. The MPI-2 implementation has been developed using one-sided communications.

4.3 NAS Parallel Benchmark Suite

The NAS Parallel Benchmarks (NPB) are developed by the Numerical Aerodynamic simulation (NAS) program at NASA Ames Research Center for the performance evaluation of parallel supercomputers [NPB02]. The NPB mimics the computation and data movement characteristics of large–scale computation fluid dynamics (CFD) applications. 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.

We will be considering the following workloads:

- 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.

- CG (Conjugate Gradient) 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) 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.

- FT (Fast Fourier Transform) 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) benchmark uses a V-cycle multi-grid method to compute the solution of the 3-D scalar Poisson equation. It performs both short and long-range communications that are highly structured.

There are different sizes/classes of the NPB including Sample, Class A, Class B, Class C and even Class D, introduced in the NPB 2.4 release. These classes differ mainly in the size of the problem. The performance analysis presented in this paper considers both NPB UPC as well as the official MPI implementations, distributed by NAS, in their NPB 2.4 release.

5. Performance of UPC

5.1 STREAM Benchmark Performance

Table 2 presents the results of the stream micro-benchmark, using tables of 64M 'double'. The measurements were collected through 8 runs, and the maximum bandwidth is reported for each operation in MB/sec. In this table, C means the sequential code, UPC-C means the sequential C code compiled and run under the UPC environment. UPC private means that all data is in the private memory space of the executing thread. Likewise, in the local and remote shared cases, the data is in the shared space, but in one case it is in the part of the space that has affinity to the executing thread, and in the other it has affinity with a different thread. These measurements, collected from the Cray X1 machine, show a consistent behavior with the exception of one operation that has to do with the use of pointers in the case of copy. In this case, the measured bandwidth was much lower than the other cases. This indicates that for private pointer implementations under UPC are not efficient. Besides, UPC performs at similar levels to those of C, regardless of the type of shared data, local or remote. This indicates two important facts. First, the hardware architecture global address space support on the X1, such as the Remote Translation Table, must be quite efficient. Secondly, the UPC compiler must be doing a good job in exploiting such hardware features.

MB/sec	Bulk Operations					
	Memcpy	**Memset**	**Struct Cpy**	**Get**	**Put**	**Copy (ptr)**
C	18897	5330	N/A	N/A	N/A	N/A
UPC-C	18839	5281	N/A	N/A	N/A	N/A
UPC private	19609	5307	2429	N/A	N/A	7859
UPC local shared	21155	5342	1904	19011	19751	6818
UPC remote shared	21764	5353	340	19842	19102	1297
MB/sec	**Element-by-Element Operations**					
	Copy (arr)	**Set**	**Sum**	**Scale**	**Add**	**Triad**
C	7869	10412	18133	7872	5749	5731
UPC-C	7867	10260	18280	7666	5684	5752
UPC private	7841	10564	18488	7888	10150	5746
UPC local shared	6814	10132	18263	7814	13110	9150
UPC remote shared	1297	12368	12367	5461	5794	7237

Table 2. UPC STREAM Measurements (in MB/sec)

5.2 UPC-NPB Performance

Figure 3 to Figure 7 show the performance of the selected UPC NPB workloads (BT, CG, EP, FT and MG). Part a in each figure shows the execution time of the respective workload. All the variants of the UPC kernels, integrating different levels of hand-tuning as described in

[ELG01] that emulate expected compiler optimizations are shown. Part b in each of these figures shows the corresponding scalability of UPC along with the linear speedup. The O0 variant indicates a code without any hand tuning, O1 indicates a code in which accesses into local shared data are handled with the same low overhead of private data, and O3 indicates that in addition to O1, we also anticipate and prefetch remote data. The reason for these particular variants is that it was discovered in our previous work[ELG01] that they are easy to do under the UPC locality conscious model, and they can result in significant performance improvements.

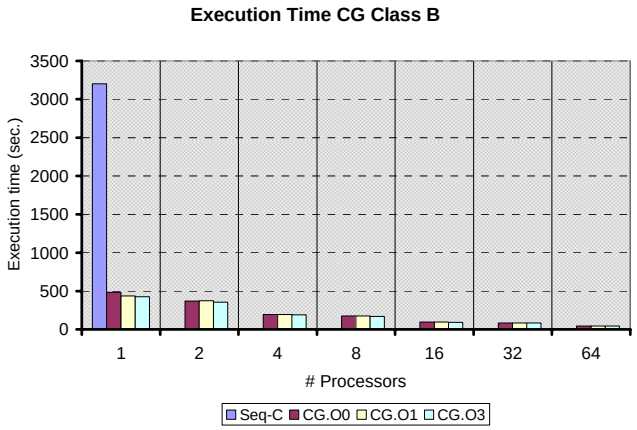

Figure 3. Performance of CG Class B

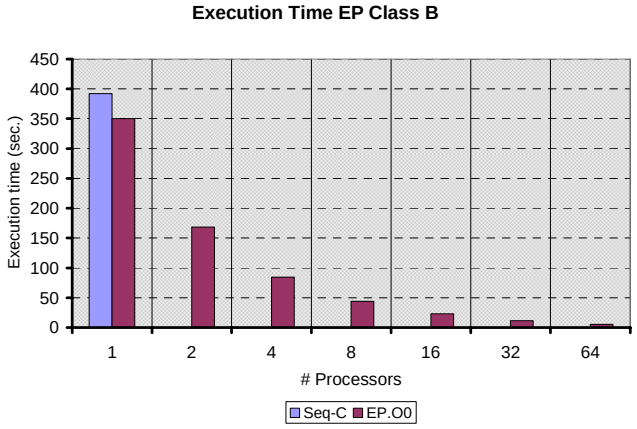

Figure 4. Performance of EP Class B

All UPC variants available for each given workload are providing similar level of performance, illustrating that the compiler is using efficiently the underlying architecture features, especially for fast address translation. Thus, the un-optimized UPC code (noted O0), is able to provide a level of performance similar to the fully hand-tuned UPC code (noted O3), leading to faster code development [CAN04].

Figure 5. Performance of FT Class B

Figure 6. Performance of MG Class B

Figure 7. Performance of BT Class B

For most of the tested workloads, UPC shows good scalability up to the 64 processors. As vector processors have shown great performance Fortran applications, and since UPC is mainly based on C, we consider in the next section the relative C to Fortran performance on the X1 and its implications for the execution times of UPC.

6. C and Fortran Performance

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 however has been exerted to make them both very similar on almost a line by line basis in order to study the differences in C and Fortran compilation. The measurements show that mostly with operations that require element-by-element access, C is performing at a lower level than Fortran, down to 2 times slower.

To explain such differences in performance, the Cray Performance Analysis Tools (PAT) [CRA04] have been used to highlight fundamental differences in the behavior of STREAM, in C and Fortran. The PAT tool set uses the Performance Counters located in the MSP, Memory and Cache levels to provide statistical measurements about the usage of each hardware feature of the X1. Table 4 shows the output gathered for a run with the Fortran and the C implementations of STREAM. The major differences are underlined in bold. The C version has issued more a lot more operations than the Fortran code. The C code has also more than 87 Millions synchronization instructions, which implies a large overhead on synchronization between the vector and the scalar units. This is generally due to a concurrent usage of scalar and vector resources, which is the case in execution scenarios such as conditionally vectorized loop.

MB / sec	**Memcpy**	**Memset**	**Copy (array)**	**Set**
F	14859	11038	14849	11558
C	19718	5330	6855	10671
MB / sec	**Sum**	**Scale**	**Add**	**Triad**
F	18568	14850	11477	16545
C	18571	7895	7015	9278

Table 3. Fortran and C STREAM Performances

To explain in further details the discrepancy in performance in some operations, loopmarking for three selected operations, memset, set and add, has been used. These operations have been chosen for their different behavior. The loopmark listing is a text-file output created by the compiler exposing how each loop is optimized, if any, during compilation. Different levels of optimizations are available, as shown in Table 5. The loopmarks of the three STREAM benchmark workloads are shown in Figure 8.

Fortran	
CPU Seconds	5.558187 sec
Vector instructions	267,437,036 instr
Scalar instructions	982,612,043 instr
Vector ops	**7,115,749,285 ops**
Total FP ops	3,825,686,927 ops
Scalar integer ops	**92,457,548 ops**
Scalar memory refs	**2,741,604 refs**
Total TLB misses	2,036 misses
Dcache references	**2,738,735 refs**
Dcache bypass refs	**2,869 refs**
Dcache misses	**1,125,446 misses**
Vector integer adds	**2,013,284,993 ops**
Vector memory refs	10,605,633,707 refs
Scalar memory refs	2,741,604 refs
Average vector length	63.999
Syncs Instr	**11,644 instr**
Stall VLSU	8,303,447,454 clks
Stall VU	8,500,732,702 clks
C	
CPU Seconds	11.224599 sec
Vector instructions	275,791,321 instr
Scalar instructions	945,536,277 instr
Vector ops	17,650,368,988 ops
Total FP ops	5,368,736,843 ops
Scalar integer ops	76,032,146 ops
Scalar memory refs	203,338 refs
Total TLB misses	1,769 misses
Dcache references	201,822 refs
Dcache bypass refs	1,516 refs
Dcache misses	62,539 misses
Vector integer adds	1,342,184,547 ops
Vector memory refs	10,939,401,644 refs
Scalar memory refs	203,338 refs
Average vector length	63.999
Syncs Instr	87,038,205 instr
Stall VLSU	5,795,559,120 clks
Stall VU	11,657,918,576 clks

Table 4. Part of the output of the PAT tools over STREAM C and STREAM Fortran

```
Primary Loop Type        Modifiers
------- ---- ----        ---------
A  - Pattern matched     b - blocked
C  - Collapsed           f - fused
D  - Deleted             i - interchanged
E  - Cloned              m - streamed but
                           not partitioned
I  - Inlined             p - conditional,
                           partial and/or
                           computed
M  - Multistreamed       r - unrolled
P  - Parallel/Tasked     s - shortloop
V  - Vectorized          t - array syntax
                           temp used
W  - Unwound             w - unwound
(1  - no optimization performed to loop)
```

Table 5. Loopmark Legend, showing how loops are optimized during compilation

First, let us consider the `memset` workload of the STREAM benchmark. The `memset` operation is a library call in C, which is basically a black box that cannot be altered. On the other hand, the equivalent Fortran statement, can be Multistreamed and fully-vectorized. A fully vectorized loop is a loop that runs entirely on the vector processor since all of its iterations are independent.

Fortran

MEMSET (bulk set)

```
146. 1      t = mysecond(tflag)
147. 1 V M--<><> a(1:n) = 1.0d0
148. 1      t = mysecond(tflag)-t
149. 1      times(2,k) = t


                  SET
158. 1      arrsum = 2.0d0;
159. 1      t = mysecond(tflag)
160. 1 MV------< DO i = 1,n
161. 1 MV      c(i) = arrsum
162. 1 MV      arrsum = arrsum + 1
163. 1 MV------> END DO
164. 1      t = mysecond(tflag)-t
165. 1      times(4,k) = t

                  ADD
180. 1      t = mysecond(tflag)
181. 1 V M--<><> c(1:n) = a(1:n) +
          b(1:n)
182. 1      t = mysecond(tflag)-t
183. 1      times(7,k) = t
```

C

MEMSET (bulk set)

```
163. 1    times[1][k] = mysecond_();
164. 1    memset(a, 1,
        NDIM*sizeof(elem_t));;
165. 1    times[1][k] = mysecond_()
        - times[1][k];

                  SET
217. 1    set = 2;
220. 1    times[5][k] = mysecond_();
222. 1 MV--< for (i=0; i<NDIM; i++)
223. 1 MV   {
224. 1 MV    c[i] = (set++);
225. 1 MV-->  }
227. 1    times[5][k] = mysecond_()
        - times[5][k];

                  ADD
283. 1    times[10][k]= mysecond_();
285. 1 Vp--< for (j=0; j<NDIM; j++)
286. 1 Vp   {
287. 1 Vp    c[j] = a[j] + b[j];
288. 1 Vp-->  }
290. 1    times[10][k] = mysecond_()
        - times[10][k];
```

Figure 8. Loopmark of MEMSET, SET and ADD operations in Fortran and C

Due to the pipelining of vector operations, this provides a much higher performance than the equivalent execution on a scalar processor. Multistreaming, like multithreading, is a mechanism which distributes a loop iteration over the four SSPs. In this case, the lower performance of C shows that the `memset` implementation in the C library is not efficient and cannot take advantage well of the underlying hardware, as compared to Fortran.

Figure 8, however, shows that the set operation is compiled with multistreamed and fully-vectorizable loop optimizations, offering great performance in both Fortran and C.

In the add operation, however, the Fortran loop is found to be multistreamed and fully vectorized (MV), whereas in the C case, it is compiled as a conditionally vectorizable loop (Vp). A conditionally vectorized loop results in two loops, duplicated in both the scalar and vector units. During run-time, the conditional expression chooses the scalar loop when needed to avoid recurrence; otherwise the vector loop is chosen. This induces synchronization between the two units, as well as decision overhead at runtime, which leads to lower performance by comparison to multistreamed and fully vectorized loops.

7. Exploiting Vectorization and Multistreaming for UPC Applications

The previous section demonstrated that in the case of C, possible optimizations are not exploited. In this section, exploiting vectorization and multistreaming opportunities for UPC will be investigated. These optimizations are considered to be critical for applications to run efficiently on the Cray X1 architecture. Three approaches to do so are studied in the following.

7.1 At the pragma level

MultiStreamed – C

ADD

```
283. 1         times[10][k] = mysecond_();
284. 1    #pragma csd parallel
285. 1         {
286. 1    #pragma _CRI ivdep
287. 1    #pragma csd for
288. 1 MV--<       for (j=0; j<NDIM; j++)
289. 1 MV         {
290. 1 MV          c[j] = a[j] + b[j];
291. 1 MV-->        }
292. 1         }
293. 1
294. 1         times[10][k] = mysecond_() -
                     times[10][k];
```

Figure 9. Hand-tuning for the C ADD operations, forcing it as a MV loop

Cray provides directives to enforce Vectorization and Multistreaming at the code level, through the use of pragmas. However, this requires an in-depth understanding of the code and implies additional effort by the developer. As far as the add operation of the STREAM benchmark is concerned, the C compiler seems not to be aggressive enough to consider it as multistreamed and vectorizable loop, as it is the case with Fortran. To remedy this, the add operation has been first forced to be fully-vectorizable then transformed into a multistreamed fully vectorized loop.

Figure 9 shows the transformation done to the code and illustrates that use of pragmas to improve performance. Table 6 summarizes the results and show that multistreaming leads to the best performance boost, as it splits the workload across SSPs.

MB / sec	F	C	C Fully Vectorized	C Multi-Streamed	C (MultiStreamed and Fully Vectorized)
Scale	11477	7015	7090	13045	13079

Table 6. Performance of the ADD operation with hand-tuning (in MB/sec)

7.2 At the construct level

For each algorithm to be implemented there is numerous ways to code it. However, these different ways of coding can lead to significant performance drop or boost.

By considering the RandomAccess micro-kernel and introducing such hand-tuning, we have observed an execution time reduced by half, as illustrated in Figure 10 and Figure 11. This was done by rephrasing a loop initializing the local shared area of the main table, replacing a `upc_forall` statement by its equivalent for statement. This is due to the change at the loopmark level: the `upc_forall` was marked as conditionally-vectorizable (Vp) whereas the equivalent for statement is marked as multistreamed and vectorized (MV). Such MV loopmark should have been automatically enforced for 'flat' `upc_forall` loops, since these loops have independent iterations by definition.

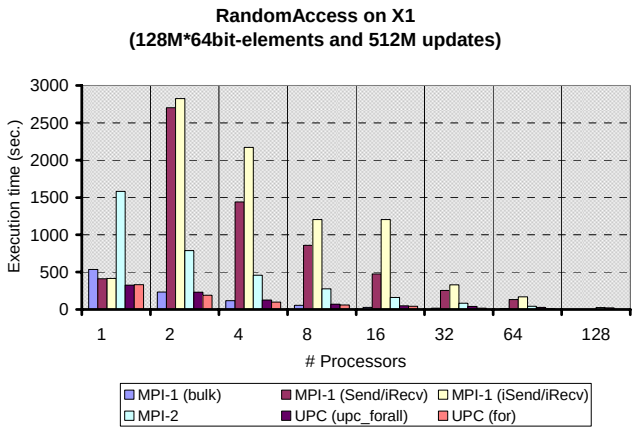

Figure 10. Execution Time of RandomAccess

As it is shown in Figure 10, the optimized UPC implementation is outperforming the MPI-1 bulk implementation, which highlights the high potential of UPC when proper mapping to the architecture features is done.

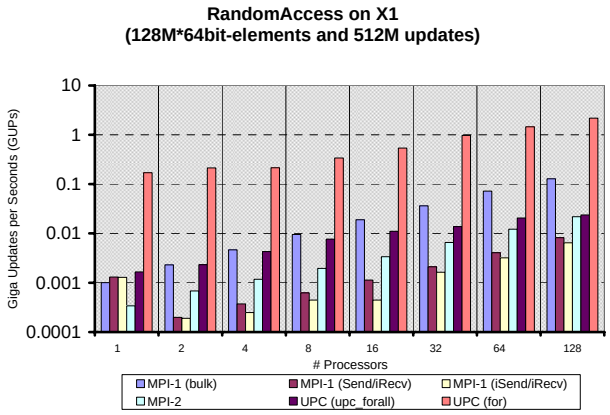

Figure 11. GUPS Index of RandomAccess

7.3 At the execution level

Considering that each MSP on Cray X1 machine contains 4 SSPs, the way the compiler distributes the workload across the available SSPs (multi-streaming) is quite critical to the performance. Thus, the ability of the compiler to support multi-streaming is very important. The following results focus on this point and uses Fortran+MPI as a reference point.

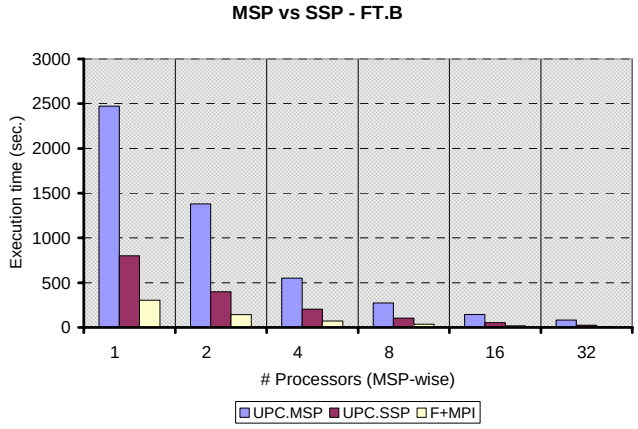

Figure 12. FT in MPI+Fortran versus UPC-MSP and UPC-SSP

As we have just shown, one way to guarantee multi-streaming for UPC codes that can advantage of it, is by adding Cray CSD directives [CRA03]. This, however, requires programmer's efforts to refine the codes. Another possible way is taking advantage of UPC's work-sharing constructs to distribute workload across threads from one end, and compile and run with SSP mode on Cray X1 machine from the other end. Thus UPC will consider the SSP as the basic processor units, instead of MSP, and distributes the workload among those SSPs.

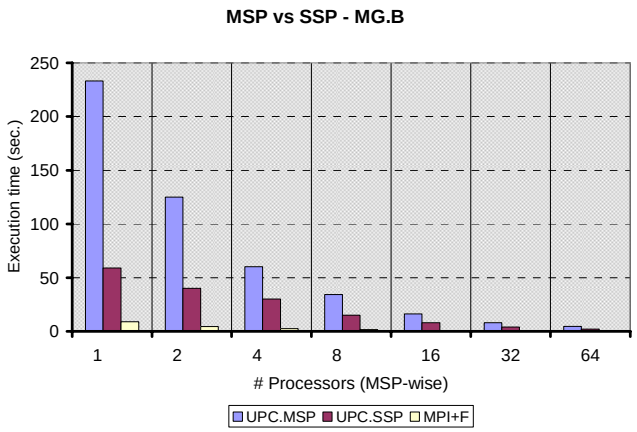

Figure 13. MG in MPI+Fortran versus UPC-MSP and UPC-SSP

By taking the same amount of MSP processors, FT and MG kernels perform much better in SSP mode than MSP mode, as shown in Figure 12 and Figure 13. The MPI FT and MG implementations, using the Fortran compiler, are performing extremely well, making good utilization of the vector resources.

8. Concluding Remarks

This paper illustrates the current standing of UPC on the Cray X1 machine, by using selected workloads from the NPB Suite. The analysis shows that Cray UPC compiler is quite efficient in accessing the global address space when compared with other cases [CAN05][CAN03].

The vectorization and multi-streaming capabilities of the compiler are critical to map applications effectively to the Cray X1 architecture. One clear observation is that the C compiler is not as aggressive as the Fortran compiler. Thus, UPC as an extension of ISO-C, suffers from the poor performance of its C compiler infrastructure.

While new versions of the UPC compiler are making their way into existence, programmers can remedy the problem using loopmark analysis to discover areas for improvements. Should the dependency relations of the code permit, three distinct approaches can be used. The first is to add compiler directives to force vectorization and multistreaming. The second is to write code which can be seen as friendly for automatic vectorization and multistreaming from the compiler. The third is to specify the SSP execution mode and distribute independent iterations using the UPC workload sharing constructs. Both methods can improve the overall UPC performance and can bring it closer to those of Fortran and MPI. However, even with that it is clear that the C compilers for X1 need dramatic improvements. Should this happen, it is expected that the UPC compiler can perform similar to parallel Fortran paradigms, such as Fortran+MPI, but with a lot less programming effort.

References

[BRO95] Brooks, Eugene and Warren Karen, Development and Evaluation of an Efficient Parallel Programming Methodology, Spanning Uniprocessor, Symmetric Shared-memory Multi-processor and Distributed-memory massively Parallel Architectures, Poster SuperComputing 1995, San Diego, CA, December 3-8, 1995.

[CAN05] Cantonnet François, El-Ghazawi Tarek, Lorenz Pascal and Gaber Jaafar, Fast Address Translation Techniques for Distributed Shared Memory Compilers, International Parallel & Distributed Processing Symposium (IPDPS), IEEE, Denver CO, April 3-8 2005

[CAN04] Cantonnet François, Yao Yiyi, Zahran Mohamed, El-Ghazawi Tarek, On the Productivity of UPC, International Parallel and Distributed Processing Symposium (IPDPS), Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO) Workshop, 2004, New Mexico

[CAN03] Cantonnet François, Yao Yiyi, Annareddy Smita, Mohamed Ahmed, El-Ghazawi Tarek, Performance Monitoring and Evaluation of a UPC Implementation on a NUMA architecture, International Parallel and Distributed Processing Symposium (IPDPS), Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO) workshop, 2003, Nice France

[CAR99] Carlson William and Draper Jesse, 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

[CRA03] Cray C and C++ Reference Manual, http://www.cray.com S-2179-50, 2003

[CRA04] Cray Inc., Online Cray Documentation, http://www.cray.com/craydoc/, 2004

[CUL93] Culler, Dusseau Andrea, Goldstein Seth Copen, Krishnamurthy Arvind, Lumetta Steven, Von Eicken Thorsten and Yelick Katherine, Parallel Programming in Split-C, Proceedings of SuperComputing 1993, Portland, OR, November 15-19, 1993

[ELG03] El-Ghazawi Tarek, Carlson William and Draper Jesse, UPC Language Specifications v1.1 (http://upc.gwu.edu), October 2003

[ELG02] El-Ghazawi Tarek and Cantonnet François, UPC Performance and Potential: A NPB Experimental Study, SuperComputing 2002, IEEE, Baltimore MD, November 2002

[ELG01] El-Ghazawi Tarek and Chauvin Sébastien, UPC Benchmarking Issues, 30th Annual Conference IEEE International Conference on Parallel Processing,2001 (ICPP01) Pages: 365-372

[MCC95] McCalpin John, Sustainable Memory Bandwidth in Current High Performance Computers, Technical Report, Advanced Systems Division, SGI. October 12, 1995

[NPB02] NAS Parallel Benchmark Suite, NASA Advanced Supercomputing, 2002, http://www.nas.nasa.gov/Software/NPB

[OPE99] OpenMP, OpenMP Reference, http://www.openmp.org, 1999.

[HPC04] High Performance Computing Challenge Benchmarks, University of Tennessee, Knoxville http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/, 2004

[SNI98] Snir Marc, Gropp William, MPI: The Complete Reference, Published by MIT Press, 1998

[UPC05] The UPC Website. http://upc.gwu.edu, May 2005.

About the Authors

Tarek El-Ghazawi is professor of the Electrical and Computer Engineering department at The George Washington University, and director of the High Performance Computing Laboratory. François Cantonnet is a Senior Research Scholar at the High Performance Computing Laboratory. Yiyi Yao is a Research Assistant in the High Performance Computing Laboratory. They can be reached at (202) 994-3769 or at gwu-upc@hermes.gwu.edu. Jeffrey Vetter is a computer scientist in the Computer Science and Mathematics Division (CSM) of Oak Ridge National Laboratory (ORNL), where he leads the Future Technologies Group. He can be reached at (865) 576-7115 or at vetterjs@ornl.gov.