RC25360 (WAT1303-028) March 13, 2013
Computer Science

IBM Research Report

Performance Analysis of the IBM XL UPC on the PERCS Architecture

Gabriel Tanase[1], Gheorghe Almási[1], Ettore Tiotto[2], Michail Alvanos[3], Anny Ly[2], Barnaby Dalton[2]

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 208
Yorktown Heights, NY 10598
USA

[2]IBM Software Group
Toronto, Canada

[3]IBM Canada CAS Research
Toronto, Canada

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Performance Analysis of the IBM XL UPC on the PERCS Architecture

Gabriel Tanase
IBM TJ Watson Research Center
Yorktown Heights, NY, US
igtanase@us.ibm.com

Gheorghe Almási
IBM TJ Watson Research Center
Yorktown Heights, NY, US
gheorghe@us.ibm.com

Ettore Tiotto
IBM Software Group
Toronto, Canada
etiotto@ca.ibm.com

Michail Alvanos†
IBM Canada CAS Research
Toronto, Canada
malvanos@ca.ibm.com

Anny Ly
IBM Software Group
Toronto, Canada
annyly@ca.ibm.com

Barnaby Dalton
IBM Software Group
Toronto, Canada
bdalton@ca.ibm.com

March 6, 2013

Abstract

Unified Parallel C (UPC) has been proposed as a parallel programming language for improving user productivity. Recently IBM released a prototype UPC compiler for the PERCS (Power 775 [35]) architecture. In this paper we analyze the performance of the compiler and the platform using various UPC applications. The Power 775 is one of IBM's latest generation of supercomputers. It has a hierarchical organization consisting of simultaneous multithreading (SMT) within a core, multiple cores per processor, multiple processors per node (SMP), and multiple SMPs per cluster. A low latency/high bandwidth

†Also, with the Barcelona Supercomputing Center, Cr. Jordi Girona 29, 08034 Barcelona, Spain
This research was supported in part by the Office of Science of the U.S. Department of Energy under contracts DE-FC03-01ER25509 and DE-AC02-05CH11231, and by the DARPA HPCS Contract HR0011-07-9-0002.

network with specialized accelerators is used to interconnect the SMP nodes (also called octants).

In this paper we discuss how XL UPC takes advantage of the hardware features available on this machine to provide scalable performance when using up to 32k cores. We analyze several benchmarks discussing the performance, describe limitations of some of the features of the language and computation paterns and discuss software and runtime solutions designed to address these limitations.

1 Introduction

The IBM XL Unified Parallel C compiler is an implementation of the Unified Parallel C language, Version 1.2, supporting IBM Power Systems (POWER7) servers running Linux. The XL Unified Parallel C compiler builds on the strength of the IBM XL compiler family and contains a run-time system designed to provide scalability on large clusters of Power Systems, and exploit the capabilities of the IBM Parallel Active Message Interface (PAMI) communication library.

In this paper we describe the main optimizations employed by XL UPC to provide scalable parallel application performance on the P775 supercomputer architecture. We discuss optimizations we performed in the compiler (High-Level Optimizer), the runtime, and user applications, and evaluate the performance of various benchmarks that benefit from all these optimizations. The experiments we consider cover multiple patterns that users may already employ in their applications and for most of them we show expected performance using the language and how to avoid some common pitfalls related to the PGAS programming model. The data gathered in the experiments included in this paper will help XL UPC users better reason about the performance of their own individual applications.

The rest of the paper is organized as follows. In Section 2 we give an overview of the P775 architecture including some of the key network and processor parameters such as bandwidth and FLOPS. Section 3 contains an overview of the XL UPC compiler and the main UPC specific high level optimizations. Section 4 describe the XL UPC runtime system and novel optimizations we employed to better exploit the P775 architecture. Starting with Section 5 we evaluate the performance of various benchmarks and applications and describe scalability challenges and possible solutions.

2 IBM Power 775 Overview

The P775 [35] system employs a hierarchical design allowing highly scalable deployments of up to 512K processor cores. The basic compute node of the P775 consists of four Power7 (P7) [38] CPUs and a HUB chip, all managed by a single OS instance. The HUB provides the network connectivity between the four P7 CPUs participating in the cache coherence protocol.

Additionally the HUB acts as a switch supporting communication with other HUBs in the system. There is no additional communication hardware present in the system (no switches). Each P7 CPU has 8 cores and each core can run four hardware threads (SMT), leading to compute nodes (*octants*) of 32 cores, 128 threads, and up to 512GB memory. The peak performance of a compute node is 0.98 Tflops/sec.

A large P775 system is organized, at a higher level, in *drawers* consisting of 8 octants (256 cores) connected in an all-to-all fashion for a total of 7.86 Tflops/s.

The links between any two nodes of a drawer are referred to as L_{local} links with a peak bandwidth of 24 Gbytes/s in each direction. A *supernode* consists of four drawers (1024 cores, 31.4flops/s). Within a supernode each pair of octants is connected with an L_{remote} link with 5 Gbytes/s in each direction. A full P775 system may contain up to 512 super nodes (524288 cores) with a peak performance of 16 Pflops/s. Between each pair of supernodes multiple optical D-links are used, each D-link having a peak performance of 10 Gbytes/s in each direction. The whole machine thus has a partial all-to-all topology where any two compute nodes are at most three hops away. Additional hardware description and key parameters are included elsewhere [23, 38, 4, 35].

3 IBM XL UPC compiler

The major components of the XL UPC compiler are depicted in Figure 1. These include the front end, high-level optimizer and low-level optimizer, and the PGAS runtime, which leverages the IBM Parallel Active Message Library (PAMI [2]).

The compiler front end tokenizes and parses UPC source code, performs syntactic and semantic analysis, and diagnoses violations of the Unified Parallel C (v 1.2) language rules. It then generates an intermediate language representation (IR) of the UPC program, augmented with UPC specific artifacts such as the layout of shared arrays, and the affinity expression of a `upc_forall` loop, for example. The augmented intermediate representation (W-Code + UPC artifacts) is consumed by the high-level optimizer, or TPO (Toronto Portable Optimizer). The high-level optimizer component has been extended to perform UPC specific optimizations; it also performs a subset of the traditional control-flow, data-flow and loop optimizations designed for the C language on UPC source code.

The high-level optimizer interfaces with the PGAS runtime through an internal API that is used to translate operations on shared objects such as dereferencing (reading/writing) a pointer to a shared object (such as a shared array for example) and performing pointer arithmetic operations. Finally, the high-level optimizer produces a modified version of the IR (Optimized Wcode) that lacks UPC-specific artifacts (operations on UPC artifacts are either translated to PGAS runtime calls or resolved to the base IR through optimizations). The IR produced by the high-level optimizer is consumed by the low-level optimizer (TOBEY), which performs further optimizations that are UPC unaware. After optimizing the low level IR, TOBEY generates machine code for the target architecture (POWER 7). This process is repeated for each

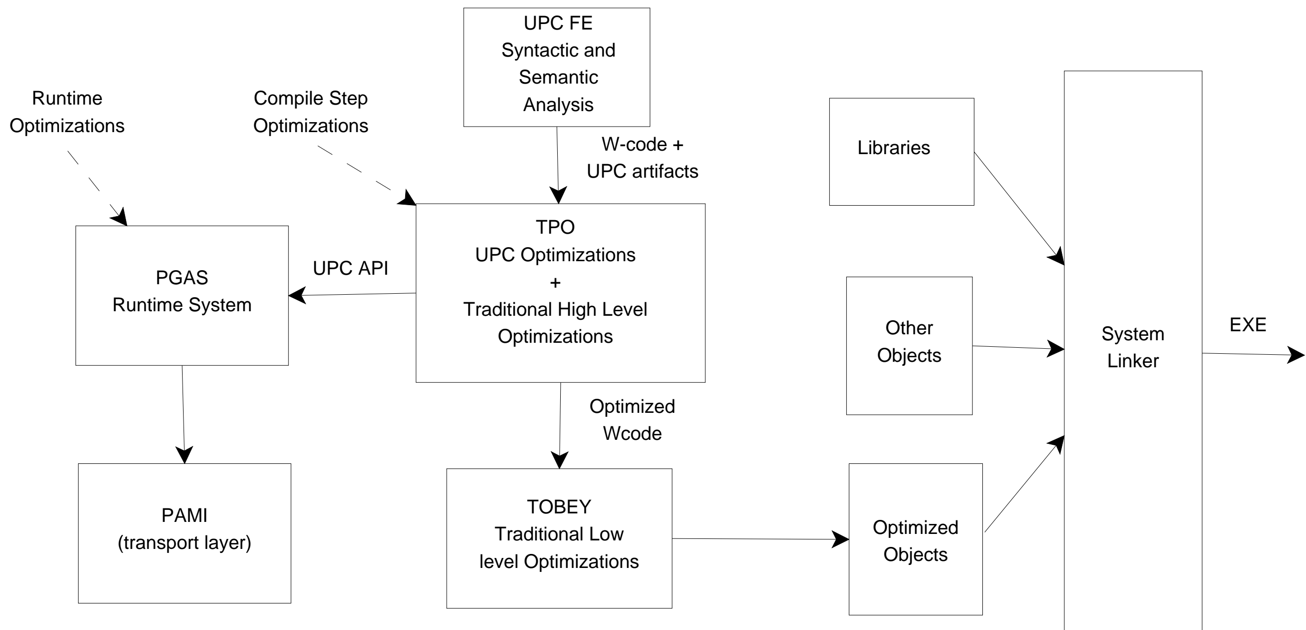


Figure 1: IBM XL UPC Compiler.

compilation unit. To complete the compilation process, the XL UPC compiler invokes the system linker, which links compiler-generated objects and any required libraries (such as the PGAS runtime library) to form an executable.

In this section we discuss the main high level optimizations that are UPC specific. All experimental results included in Section 5 employ these optimizations apropriately.

3.1 Shared object access optimizations

The XL UPC compiler implements a set of performance optimizations on shared array accesses. The compiler can partition shared array accesses performed in a `upc_forall` work-sharing loop into two categories: shared local accesses (accesses that have affinity with the issuing thread) and shared remote accesses. Shared array accesses that are proven to have affinity with the issuing thread are optimized by the compiler in such a way as to eliminate unnecessary runtime calls. Shared array accesses that are remote (have affinity to another thread) can be coalesced by the compiler to reduce the communication time required to perform them.

3.2 Shared object access privatization

In general the XL UPC compiler translates accesses to shared arrays by generating an appropriate set of runtime function calls. In the context of a `upc_forall` loop the compiler can often prove that the memory read and/or written during an array access operation resides in

the local address space of the accessing thread; in such cases the compiler generates code that performs the indexing (pointer arithmetic) operations required to access the shared array directly. To do so the compiler retrieves the address of the shared array partition in the local address space (the private address space of the accessing thread) via a runtime function call. It then schedules the runtime call outside the `upc_forall` loop nest containing the shared array access being translated. This is legal because the local shared array address is loop invariant. Finally the compiler uses the local array address to index the appropriate shared array element, doing any pointer arithmetic operations that are necessary locally.

3.3 Shared object access coalescing

In general the XL UPC compiler translates a remote shared array access by generating a call to the appropriate runtime function. For example, reading from (or writing to) multiple shared array elements that have affinity to the same remote thread causes the compiler to generate a runtime call for each of the array elements read. In the context of a `upc_forall` loop nest the compiler can often determine that several array elements are read from the same remote partition. When that is the case the compiler combines the read operations and generates a single call to the runtime system to retrieve the necessary elements together, thus reducing the number of communication messages between the accessing thread and the thread that has affinity with the remote array partition.

3.4 Shared object remote updating

This optimization targets read-modify-write operations on a shared array element. To translate a read operation followed by a write operation on the same shared array element the compiler would normally generate two runtime calls: one to retrieve the shared array element and one to write the modified value back. When the compiler can prove that the array elements being read and written have the same index, it can generate a single runtime call to instruct the runtime system to perform the update operation on the thread that has affinity with the array elements accessed. This optimization reduces the number of calls required to translate the read-modify-write pattern from two to one, therefore reducing the communication requirement associated with the operation. On the P775 architecture this compiler optimization also allows the PGAS runtime to generate RDMA messages as we will see in Section 4.

3.5 Array idiom recognition

Unified Parallel C programs often include loops that simply copy all elements of a shared array into a local array, or vice versa, or loops used to set all elements of a shared array with an initial value. The XL UPC compiler is able to detect these common initialization idioms and substitute the fine grained communication in such loops with coarser grained communication. The compiler achieves this goal by replacing the individual shared array accesses with calls to

one of the UPC string handling functions: `upc_memget`, `upc_memset`, `upc_memcpy`, or `upc_memput`.

3.6 Parallel loop optimizations

The XL UPC compiler implements a set of optimizations that remove the overhead associated with the evaluation of the affinity expression in a `upc_forall` loop. The affinity expression in a `upc_forall` loop could be naively translated by using a branch to control the execution of the loop body. For example an integer affinity expression "i" could be translated by inserting the conditional expression `(i == MYTHREAD)` around the `upc_forall` loop body. The compiler instead translates the `upc_forall` loop into a for loop (or a for loop nest) using a strip-mining transformation technique that avoids the insertion of the affinity branch altogether, and therefore removes a major obstacle to the parallel scalability of the loop.

4 XL UPC runtime

The XL UPC runtime is the component responsible for orchestrating data transfer between UPC threads, managing shared memory allocation and deallocation, and implementing synchronization primitives, collective operations and remote atomic operations. The runtime relies exclusively on the IBM PAMI [2, 42] library for all its communication needs, similar to other IBM middleware such as MPI[31], OpenShmem and X10[6]. We describe in this section how the runtime system takes advantage of the P775 specific architecture features to provide good performance.

4.1 Memory Management

One important feature of the P775 architecture is the availability of multiple modes of data transfer, including optimized short message transfers (SHORT), arbitrary size messages (FIFO) and Remote Direct Memory Access (RDMA). SHORT data transfers are used for message sizes up to 128 bytes. FIFO data transfers are used for messages larger than 128 bytes. In the `RDMA` mode a process can read or write data on a remote compute node without involving its CPU. On top of these hardware transfer modes PAMI provides short active messages, active messages with large data transfers and RDMA transfers. While the user doesn't need to know the effective remote address for active messages, active message performance is often slower than RDMA due to CPU involvement on the remote side. In the RDMA mode the data transfer is fast but it requires the thread initiating the transfer to know the remote memory address where data will be read or written.

In this section we describe the XL UPC memory allocator designed to facilitate data exchanges using RDMA. The two key constraints for an efficient RDMA operation on P775 are the following:

- a thread invoking such an operation needs to know both the local and remote virtual memory address
- memory involved in RDMA data exchanges needs to be registered with the operating system, and this can be achieved with a special registration call available in the PAMI library.

While the complete details about the memory allocator are outside the scope of this paper, here we include the key design decisions. First we distinguish in UPC two types of memory allocations: *local memory* allocations performed using malloc and which are currently outside of the tracking capability of the runtime, and *shared memory* allocated using UPC specific constructs such as `upc_all_alloc()`, `upc_alloc()` and `upc_global_alloc()`.

Within the UPC specific family of allocation functions, the runtime currently employs a set of optimizations to help both with remote memory address inference for RDMA operations and with memory registration. First, shared arrays allocated using `upc_all_alloc()` are allocated symmetrically at the same virtual memory address on all locations and automatically registered for RDMA operations. For this the XL UPC allocator creates a reserved area in the virtual address space of each process called the symmetric partition. The starting virtual address of each symmetric partition, called the origin, is identical across all threads and distributed shared arrays are then stored in blocks of memory isomorphically located in each symmetric partition. The READ and WRITE operations on elements of a shared array allocated with `upc_all_alloc`, therefore, know the virtual address in the remote locations. This enables RDMA operations, which do not interrupt the remote process and are accelerated in P775 network hardware. Shared memory allocated using the `upc_global_alloc` primitive is handled in a similar fashion.

Using `upc_alloc` the user can declare shared arrays allocated in one thread's address space. In this situation, other threads can only obtain references to these arrays using another explicit data exchange. If such a reference is obtained, a remote thread can perform RDMA reads or writes to the data of the shared array. In the current XL UPC implementation however the memory allocated using `upc_alloc` is not registered with PAMI.

Another case for which we don't currently have a scalable solution is the situation where memory accessed by `upc_memput` or `upc_memget` are from arrays explicitly allocated with system `malloc`. For a `upc_memput` for example, the destination must always be a shared memory array, while the source can actually be from an array allocated with `malloc`. In this situation RDMA operations won't be efficiently exploited unless the user explicitly performs a PAMI or runtime call to register the memory employed in transfer. Similar to the `upc_alloc` memory allocated with system `malloc` must be register explicitly by the user using PAMI calls.

4.2 Point to point data transfers

Point to point data transfers are employed by UPC whenever a remote array index is accessed or whenever `upc_memput`/`upc_memget` are invoked. As introduced in the previous section, the underlying network supports three modes of data transfer, which are referred to as active messages with short and large data and RDMA. All three modes are exploited by the XL UPC runtime system. In the current pre release version RDMA must be explictly requested by the user using `-xlpgashfiupdate` runtime flag.

In this section we describe how we enforce the ordering constraints imposed by the UPC memory consistency model. The underlying P775 interconnect and PAMI library do not preserve the order of messages between a source and destination process. For this reason, when mapping UPC constructs to PAMI primitives, we currently explicitly wait for a message to be remotely delivered before sending the next message to *the same destination*. However we can send a message to a different destination without waiting for the acknowledgment of messages on previous destinations. This was an important design decision we had to employ in order to satisfy consistency of memory reads and writes in UPC. While in principle this can be seen as a performance bottleneck, in practice this decision did not affect the performance of most benchmarks we evaluated. This is because often the threads send to different destinations before sending again to a particular one. On a fast architecture like P775 the amount of time between sends to the same destination from the current threads is often long enough that the acknowledgment is already received.

The above implementation for point to point data transfers simplifies the complicated UPC fence mechanism, because now the information on when memory accesses from a thread are completed is always locally available.

4.3 Atomic Operations

The XL Unified Parallel C compiler implements the atomic extension of the Unified Parallel C language as proposed by Berkeley UPC [1]. This extension, which is on track to being adopted as part of the UPC 1.3 specification, allows users to atomically read and write private and shared memory in a UPC program. With atomic functions, you can update variables within a synchronization phase without using a barrier. The atomic functions are declared in the `upc_ext_atomics.h` header file and are included in Figure 2. The function prototypes have different variants depending on the values of `type, X`, and `RS`. `X` and `type` can take any pair of values in (I, int), (UI, unsigned int), (L, long), (UL, unsigned long), (I64,int64_t), (U64,uint64_t), (I32,int32_t), (U32,uint32_t). `RS` can be either 'strict' or 'relaxed'.

4.4 Accelerated Collectives

UPC collectives are implemented completely within the runtime system. The compiler redirects UPC calls to runtime entries with only some minimal analysis in certain situations. In

```
type xlupc_atomicX_read_RS(shared void *ptr);
void xlupc_atomicX_set_RS(shared void *ptr, type val);
type xlupc_atomicX_swap_RS(shared void *ptr, type val);
type xlupc_atomicX_cswap_RS(shared void *ptr, type oldval, type newval);
type xlupc_atomicX_fetchadd_RS(shared void *ptr, type op);
type xlupc_atomicX_fetchand_RS(shared void *ptr, type op);
type xlupc_atomicX_fetchor_RS(shared void *ptr, type op);
type xlupc_atomicX_fetchxor_RS(shared void *ptr, type op);
type xlupc_atomicX_fetchnot_RS(shared void *ptr);
```

Figure 2: XL UPC Atomics Interface

turn the runtime transforms UPC pointers-to-shared to regular pointers and calls appropriate collectives within PAMI. Reduce and barrier collectives in XL UPC exploit the Collective Accelerated Unit (CAU) available on P775 when low latency performance is crucial.

The CAU unit integrated in the IBM P775 HUB provides offload and acceleration for broadcast and reduce up to 64 bytes of data. For reduce, it supports NO-OP, SUM, MIN, MAX, AND, OR and XOR operations with 32-bit/64-bit signed/unsigned fixed-point operands or single/double precision floating-point operands. The benefits of deploying the CAUs are most evident in low latency operations. For operations requiring large bandwidth like broadcast or reduction on large data the CAU unit may not perform better than the point to point versions. For a complete description on how collectives exploit shared memory and CAU on P775 please consult [42].

Due to machine specific configuration accelerated collectives need to be enabled with proper environment variables or load leveler directives in order to be available in XL UPC.

4.5 Other Runtime Optimizations

We briefly describe here a number of other optimizations available at the runtime level. On a platform that allows up to 128 threads per shared memory node, thread binding turns out to be crucial. The runtime system provides flexible control on how to bind UPC threads to hardware threads. By default no binding is performed while a specific mapping is performed with proper command line arguments. The autobinding option (e.g., -xlpgasbind=auto) is the option recommended. Huge memory pages is another runtime option, which instructs the memory allocator to use huge memory pages available on P775. As we will explain in Section 5.3, certain applications benefits from this.

```
// Next is ok if P=2^n
// #define PERM(ME,TOT,ITER) ((ME)^(ITER))
#define PERM(ME,TOT,ITER) (((ME)+(ITER))%(TOT))
int64 all2all(TYPE *dst, uint64 *src,
              int64 len, int64 nwrd){
int64 i, j, pe;
len = len - (len % (nwrd * THREADS));
for (i = 0; i < len; i += THREADS*nwrd){
 for (j = 0; j < THREADS; j++) {
  pe = PERM (MYTHREAD, THREADS, j);
  shared[0] uint64 *ptr=(shared[0] uint64*)&dst[pe];
  upc_memput(&ptr[i+MYTHREAD*nwrd],&src[i+pe*nwrd],
       nwrd * sizeof(uint64));
 }
}
return len;
}
```

Figure 3: All2All

5 Performance Evaluation

In this section we evaluate and analyze the performance of various benchmarks and applications. We discuss the results obtained and compare them to the theoretical limits imposed by the system architecture. We identify performance bottlenecks in existing benchmarks and discuss possible solutions.

5.1 Methodology

In our experiments we have used the IBM XL Unified Parallel C compiler prototype for Linux. Each node runs Linux with kernel 2.6.32. Unless specified all runs use one process per UPC thread and schedule one UPC thread per Power7 core. Each UPC thread communicates with other UPC threads through the network interface or interprocess communication. The UPC threads are grouped in blocks of 32 per octant and each UPC thread is bound to a different processor core. All benchmarks are compiled using the '`-qarch=pwr7 -qtune=pwr7 -O3`' compiler flags.

5.2 The MPP Benchmark

To evaluate the performance of a parallel machine using UPC point to point and collective operations we employed the MPP benchmark suite. The MPP benchmark code came to us as

```
1 /* mpp accumulate */
2 int64 mpp_accum_long (int64 val)
3 {
4   shared static uint64 src[THREADS];
5   shared static uint64 dst;
6   src[MYTHREAD] = val;
7   upc_all_reduceUL(&dst, src, UPC_ADD,
8                    THREADS, 1, NULL,
9                    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
10  return dst;
11 }
12 // main
13 do_all2all_warmup(...);
14 //start timed loop
15 for(i=0;i<n_iterations;++i)
16  mpp_accum_long(...);
```

Figure 4: Reduce Benchmark.

```
1 void   mpp_broadcast   (TYPE *dst, TYPE *src,
2                         int64 nelem, int64 root)
3 {
4   upc_all_broadcast(dst, &src[root],
5                     nelem * sizeof(uint64),
6                     UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
7 }
8 // main
9 do_all2all_warmup(...);
10 for(nelems=4;nelems<100000;n*2)
11  //start timed loop for each input size
12  for(i=0;i<n_iterations;++i)
13   mpp_broadcast(dst,src,nelems,i%THREADS);
```

Figure 5: Broadcast.

part of an interaction with IBM clients, and is used in this paper with their permission. We describe and analyze the performance of the five MPP benchmarks next.

5.2.1 Alltoall

The all-to-all benchmark is one of the most important benchmarks we evaluated for measuring the bandwidth of the system. The pattern employed by this benchmark is very important as it shows up in a large number of scientific applications including FFT, Sort, and Graph 500. The kernel for this benchmark is included in Figure 3. This measurement allocates a shared buffer, using `upc_all_aloc`, initializes it with random data and then executes the equivalent of a global matrix transpose using `upc_memput` transfers. In Figure 3, Lines 7-9, we notice the large message being broken in chunks of size `nwrd`, which are subsequently sent to all other threads according to a given permutation (Line 10). In addition to being used for measurement purposes, the all-to-all communication phase is employed by all other benchmarks as a "warmup exercise" and sanity check; the communication pattern exercises all links in the interconnect and warms up caches.

While some initial permutations specifying the order in which data will be written are provided with the code (Figure 3, Lines 1-3) they do not lead to optimal performance on the P775 system. We have evaluated the performance for different permutations and what proved to be the best strategy is to use a random enumeration of the destination threads. The enumeration is unique for each thread and unless explicitly specified this is the communication pattern employed for measuring the results included in this section. As we will see later in this section, the measured bandwidth also depends on the granularity of the data exchanged, so unless specified the results included are for `nwrd`=8192 and using one megabyte as the size of the big message exchanged (`len=1MB`).

Discussion of machine peak versus measured performance: In Table 1 we include for reference the peak all-to-all bandwidth of the machine for various configurations and how much we are able to achieve using XL UPC all-to-all.

There are two main factors that need to be considered when estimating the bandwidth of a P775 system: the peak bandwidth of the various links (LL, LR, D), and the peak bandwidth at which the HUB chip can push data in and out of a node (HUB limit). The data included in Table 1 is essential for understanding the performance of the measured all-to-all bandwidth included in Figures 6(a),(b) and 7(a),(b) and of subsequent benchmarks like Guppie described in Section 5.3 and FFT described in Section 5.4.1.

The following are the columns of the table:

- "Scenario" describes various configurations of the machine considered.
- "Links" describes the physical links used which are LL, LR, DL or a combination of these.

Scenario	Links	Max All2All BW (GB/s)	Max HUB Agg BW (GB/s)	Agg Measured BW(GB/s)	% Max All2All	% Max HUB
1 Octant	HUB		37.1	37.1875		100
2 Octants	1 LL	96	74.36	61.5	64	**82.7**
4 Octants	4 LL	384	148.72	71.6875	18	**96.4**
1Dr	16 LL	1536	297.44	287.5	18	**96.6**
1Dr+1oct	16LL+1LR	720	334.64	227.81	31	**68**
1SN	256LR	5120	1189.7	1046.25	20	**87.9**
2SN	8DL	320	2379.5	187.5	**58**	7.88

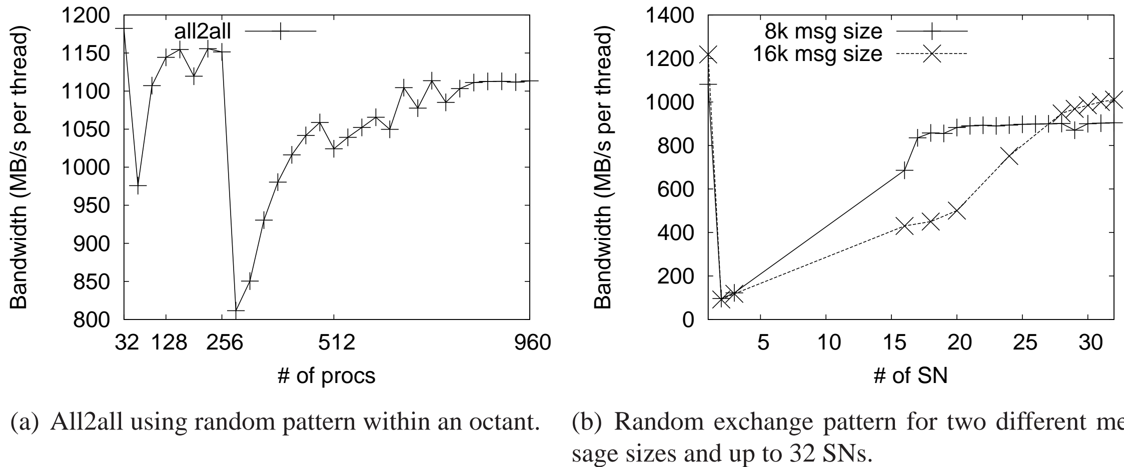
Table 1: Estimated peak all-to-all bandwidth versus measured bandwidth for different machine configurations.

- "Max All2All BW (GB/s) contains the peak all-to-all bandwidth the interconnect can provide.
- "Max HUB Agg BW (GB/s)" represents the maximum all-to-all bandwidth measurable on the system when taking into account the HUB's maximum ever measured bidirectional bandwidth. We estimate this column by multiplying the number of HUBs employed in a particular configuration with the maximum measured capacity of a HUB.
- "Agg Measured BW (GB/s)" is the aggregated bandwidth we measure using MPP all-to-all benchmark.
- "% Max All2All" is the percentage of the measured bandwidth relative to what the links can provide.
- "% Max HUB" is the percentage we measure relative the maximum it can be measured on this platform when taking into account the HUB limit.

Next we discuss the five scenarios considered in Table 1 corresponding to the different components of a P775 supercomputer:

One Octant: When running the benchmark within an octant, all communication is pushed through the local HUB chip using RDMA operations only, and overall we measure 1190 MB/s/thread for an aggregated bandwidth of 37.18 GB/s. Essentially this gives the upper limit on how much data the HUB can move in and out of the octant. This is the HUB limit value we used to compute the values of Column 4.

Two Octants: When using two octants, the communication passes through one LL link whose capacity is 24 GB/s bidirectional. The all-to-all peak bandwidth in this case is $24GB/s\times$



(a) All2all using random pattern within an octant.

(b) Random exchange pattern for two different message sizes and up to 32 SNs.

Figure 6: All2All performance within a supernode and at scale using 32 SNs.

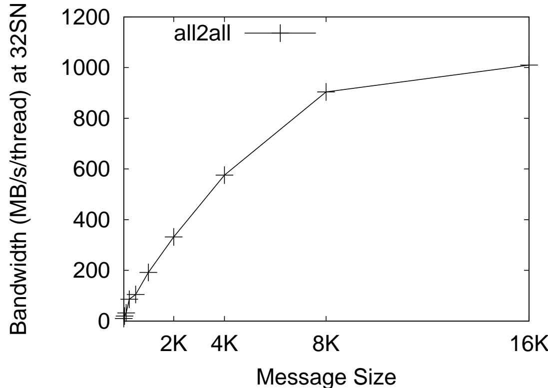
2×2 where the first 2 is because the link support 24 GB/s in both directions, and the second 2 is because half the traffic stays within the octant. Thus the peak bandwidth is 96 GB/s (column 3, row 3). In column 5, row 3 and correspondingly in Figure 6(a) we see the aggregated measured bandwidth to be 61.5 GB/s which is about 60% of the peak bandwidth of the link. However taking into account that the maximum bandwidth measured for the HUB chip is 37.18 GB/s, two HUB chips can push 74.36 GB/s, and our result of 61.5 GB/s is 82.7% of that. However when `nwrd` is increased to 16k words per round, we achieve 87% of the HUBs capacity and this essentially is an indication of some small software overheads when moving smaller data messages through the network.

Four Octants: Similarly we estimate the bandwidth for four octants (one drawer) where essentially we see that the immense bandwidth available through the LL links can not be efficiently exploited due to the HUB chip limit. The results achieved using XL UPC are similar to other results obtained using either PAMI directly or lower level C interfaces. As we can see in Figure 6(a), and in Table 1, when using four octants the network bandwidth is not a limit and essentially we achieve 96% of the highest measured HUB bandwidth.

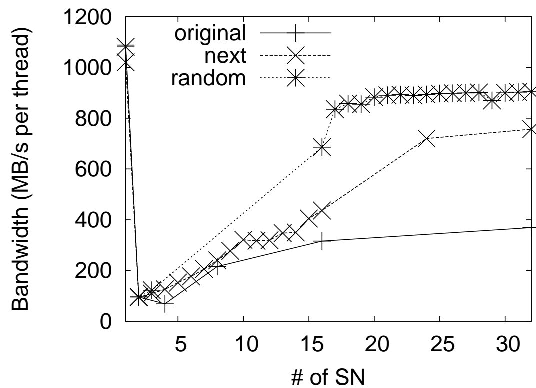
One drawer plus one octant: When using a drawer and an octant we start exercising the LR links which can only carry 5 GB/s bidirectional. In Figure 6(a) we see the bandwidth dropping to 810 MB/s per thread or 227 GB/s aggregated bandwidth. This is only 68% of what the HUB can achieve.

One supernode: With a full supernode we see again a huge aggregated peak bandwidth of 5120 GB/s out of which we can only achieve 20%. Taking into account the HUB limitation though we achieve 87%.

Two supernodes and large scale case: As we move from one super node to two supernodes, the bandwidth drops considerably (see Figure 6(b)). This is mainly due to limited cross-section bandwidth provided by the eight D links between two supernodes. In Table 1 we see the peak



(a) Random exchange pattern for two different message sizes.



(b) Three different exchange patterns.

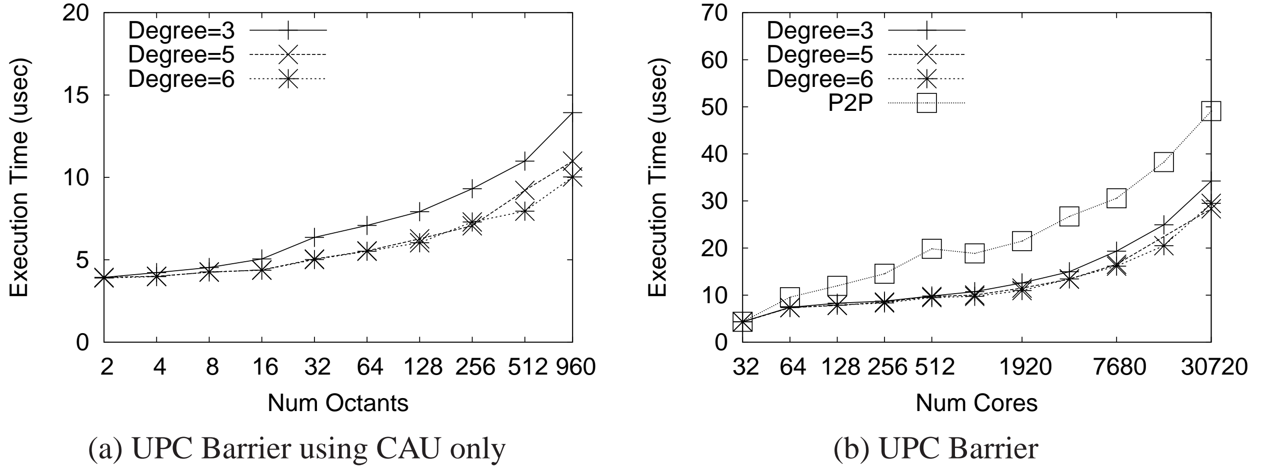
Figure 7: Three different communication schedules(a) and message size study with up 32k words (b).

aggregated bandwidth dropping to 320 GB/s out of which we can achieve 60% using XL UPC all-to-all. In Figure 6(b) we include the all-to-all performance when using up to 32 supernodes and two different message sizes : `nwrd` of 8k and 16k respectively. We observe that at scale, the 16k case achieves 1000 MB/s per thread. Considering the HUB limit, XL UPC achieves 85% of the maximum performance at scale. An independent experiment conducted on the machine, using only C and low level communication primitives (thus avoiding the overhead of UPC and PAMI) achieved the same peak bandwidth at scale (Ram Rajamony).

Message size study: Figure 7 (a) shows the performance of the random all-to-all computation on 32 supernodes when the big message of size 1MB is split into smaller messages with sizes between 32 and 16384 words. In the code included in Figure 3 this is encompassed by the variable `nwrd`. As we increase the message size we measure higher bandwidth, and this is due to less overhead per message injected into the network. On 32 supernodes using `nwrd`=16k, we achieve 1000MB/s per thread or 85% of the HUB capacity. We recommend that, to fully exploit the bandwidth of this system, UPC memput messages be aggregated in larger chunks.

Different communication schedules: In Figure 7(b) we include results for the three different communication schedules we considered, all using `nwrd`=8k. A first schedule sends messages consecutively, starting with the next logical thread and in a cyclic manner (Figure 3, Line 3). In a second schedule, each thread sends across octants from step zero. This is achieved incrementing the destination each iteration with 32 (the octant size) and starting from current thread ID. A third schedule uses the random exchange pattern used for all previous experiments.

The results for the first schedule are included in Figure 7(b) and marked as “original” as this is the default schedule included in the benchmark. The maximum performance we achieved with this is only 380 MB/s per thread, which is only 32% of the HUB capacity. The second



(a) UPC Barrier using CAU only

(b) UPC Barrier

Figure 8: The performance for UPC barrier using CAU only (a) and at scale(b).

pattern improves the performance to almost 800 MB/s per thread which is around 80% of the HUB capacity. It turns out that while so far we have not found a perfect communication schedule for the topology employed by this machine, the best option is to employ a completely random pattern on each communicating thread. Such a random scheme avoids communication hot spots and achieves the highest performance among all schemes we devised. As shown in Figure 7(b), in the random experiment we achieved 950 MB/sec on 32 supernodes, and very good performance overall starting with 16 supernodes (80% of the HUB capacity).

5.2.2 Barrier

This benchmark performs a timed loop of barrier invocations performed by all threads. Average barrier latency is reported. In general the UPC barrier requires a check to be performed on an optional input argument to ensure proper barrier matching across different threads. For this reason the XL UPC barrier is actually implemented as a PAMI short allreduce operation, not a barrier.

Figure 8(a) shows the performance of the UPC barrier when using CAU units only. The barrier is implemented as an allreduce operation consisting of a reduce on node zero followed by a broadcast from node zero. For this experiment we used a large P7IH cluster of up to 960 SMP nodes and we employ one UPC thread per SMP node, thus using only one of the 32 cores/node. The CAU collectives are enabled using the PAMI `MP_COLLECTIVE_GROUPS` environment variable. For comparison purposes we include in the plot the performance of 3 different CAU versions employing CAU trees with 3, 5 or 6 connections per CAU node.

As expected, the performance of the CAU collective is affected by the number of children in the CAU tree; the more connections per CAU node, the lower the operation latency. Overall

the latency of the barrier slowly increases as we increase the depth of the CAU tree. On average we introduce 1 μs as we increase the depth by one.

The benefits of deploying the CAUs are most evident in low latency operations. For operations requiring large bandwidth, such as broadcast or reduction on large data, the CAU unit may not perform better than the point to point versions [42].

In Figure 8 (b) we include the performance of the barrier when using 32 threads per octant and up to 30720 threads in total. The algorithm in this case has a shared memory component within the octant and a CAU component across octants [42]. For comparison purposes we compare the CAU accelerated collective for various CAU trees and a generic version of the barrier implemented using point to point messages. On 32 super nodes we measured 49 μs for the point to point barrier, 34μs for the CAU with tree of degree 3 and 28μs for trees of higher degree.

5.2.3 Allreduce

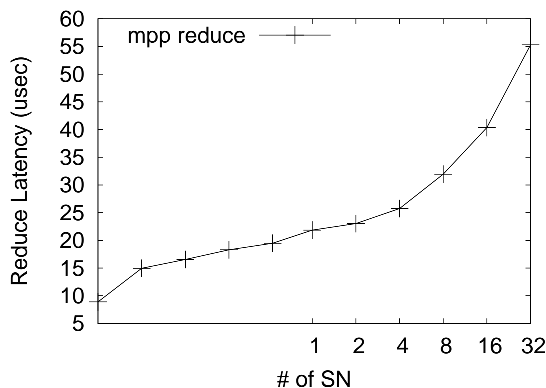


Figure 9: MPP Allreduce.

The code for this benchmark is included in Figure 4. The benchmark performs a loop consisting of the following operations: reduce a set of values from a shared array to a shared scalar integer, and return said shared value to all calling threads. The return of a shared value as a local value to all threads is a covert form of broadcast. Implemented naively (i.e. with assignments) this operation can lead to major performance degradation.

We believe that the particular way in which the operation is expressed reflects the fundamental inadequacy of the `upc_all_reduce` operation in the UPC specification. In any case the current formulation of the benchmark presents a major performance and scalability challenge to the UPC compiler which required us to look for a compiler optimization to improve the scaling of the allreduce MPP benchmark.

We observe in Figure 4 that the benchmark allocates a shared integer, followed by a reduction of data from a shared array into the shared integer and returning the shared integer value

```
int64 mpp_accum_long (int64 val)
{
  shared static uint64 dst;
  //code that assigns a value to dst
  int64 ret = dst;
  return ret;
}
```

(a) Original Code

```
int64 mpp_accum_long (int64 val)
{
  shared static uint64 dst;
  //code that assigns a value to dst
  int64 ret;
  broacast dst to ret;
  return ret;
}
```

(b) Optimized

Figure 10: Optimizing shared scalars access

to all threads. A straightforward implementation of this code translates into the following sequence of operations: `barrier, reduce, barrier` and read shared scalar from thread zero. The last step of this sequence is not scalable and performance for this kernel greatly degrades as we use an increasing number of threads.

Essentially what the reduce kernel achieves is an allreduce operation where all threads get a copy of the reduction. Based on our experience with UPC this pattern is commonly used. Thus we employed a compiler optimization that replaces the shared scalar reads from all threads with a broadcast operation. This way rather than using a linear time operation in the number of threads we use a logarithmic operation.

Collective Idiom Identification (i.e. replacing a shared scalar read operation with a broadcast operation). This optimization has the goal of transforming operations that result in fine grained P2P communication with a suitable collective operation. In particular the optimization scans each procedure in a UPC program to look for assignment statements that are concurrently executed by all threads, and where the right hand side of the assignment is a shared scalar variable and the left hand side of the assignment is a non-shared variable (a local stack variable or a global thread-local variable for example).

Consider the example in Figure 10(a). The assignment statement at line 5 reads a shared scalar variable (dst) and assigns a local automatic variable (ret). The assignment statement can be replaced with a broadcast operation if the compiler can prove that the statement is executed concurrently by all threads (`upc_all_brodcast` must be executed collectively by all threads). The optimized code for Figure 10(a) is shown in Figure 10(b).

The performance for the optimized MPP allreduce benchmark is included in Figure 9(b). We observe good logarithmic speedup with 55 μs latency when using 32 supernodes. In [41] we include a more in-depth analysis of other factors affecting the performance like the shared memory component and the CAU component of the allreduce collective.

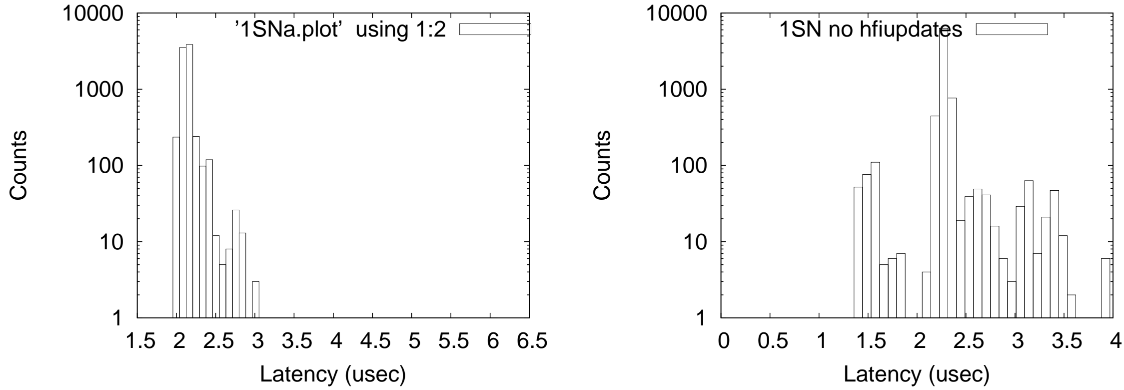
Size	1 SN (B/s)	2 SN	4 SN	8 SN	16 SN	32 SN
8	97800	82170	76790	61950	46520	35700
16	287930	252370	222480	175030	139420	100410
32	623030	526360	459600	379310	299660	222950
64	1140000	1020000	890770	746390	579690	425970
128	2180000	2010000	1720000	1390000	1160000	823830
256	4380000	4000000	3370000	2940000	2240000	1770000
512	9000000	8140000	6480000	5630000	4260000	3060000
1000	17030000	15350000	12940000	10610000	8050000	6240000
2000	29370000	25980000	22300000	18170000	14800000	10350000
4000	53910000	48410000	40940000	34100000	28320000	21210000
8000	80390000	72210000	61710000	54890000	45290000	34490000
16000	103560000	90550000	81440000	70510000	59040000	47040000
32000	154070000	138680000	122170000	107620000	93270000	78180000
64000	182810000	197920000	177050000	158900000	140390000	121520000
98000	250510000	229530000	208790000	187020000	168500000	143570000

Table 2: Broadcast bandwidth per thread for various message sizes. The size is expressed in bytes and the bandwidth is expressed in B/s per thread.

5.2.4 Broadcast

The code for this benchmark is included in Figure 5. It evaluates the performance of the broadcast collective for various data sizes (powers-of-two word buffers up to a given maximum size). Average wall time (in seconds), per thread bandwidth, and aggregate (per job) bandwidth are reported. For each iteration the root is randomly chosen. For reduce and broadcast the `UPC_IN_ALLSYNC`, `UPC_OUT_ALLSYNC` flags are used in the benchmark. This increases execution time due to a fence and a barrier call at the beginning and the end of the function call..

Table 2 shows the performance of broadcast in terms of bandwidth per thread, for different buffer sizes varying from 8 bytes to 98 Kbytes. For large buffers we see the bandwidth per thread slowly decreasing as we increase the number of threads. The algorithm employed is a P2P binomial algorithm whose execution time is logarithmic with the number of threads used. Since the broadcast is not pipelined, every doubling of the number of UPC threads causes the depth of the binomial tree to increase by one, affecting execution time and thus reducing measured bandwidth. For the largest message used we achieve 250 MB/s per thread within one supernode, 229 MB/s/thread on 2 supernodes and 143 MB/s/thread on 32 supernodes.



(a) Latency on 1 SNs with -xlpgashfiupdates (RDMA enabled).

(b) Latency on 1 SN without -xlpgashfiupdates.

Figure 11: MPP Ping pong latency on one supernode with and without -xlpgashfiupdates flag.

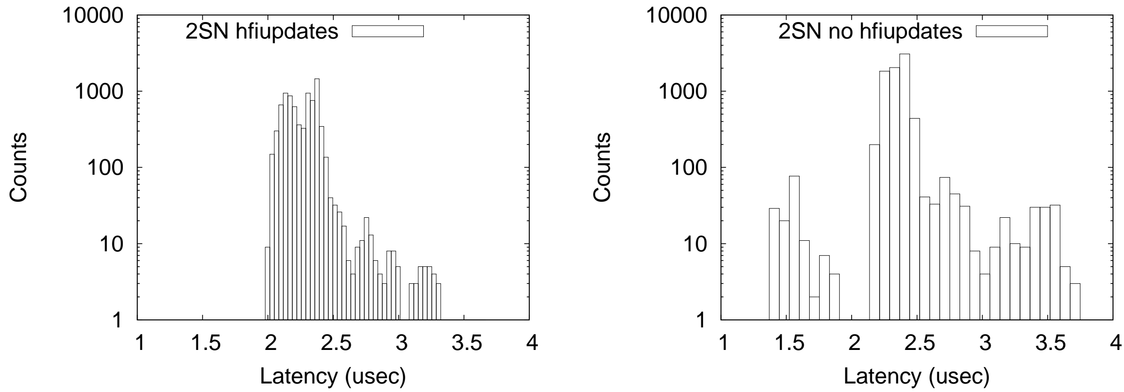
5.2.5 Ping-Pong Test

The ping pong benchmark tests the latency of reading and writing remote shared memory locations. The benchmark writes a remote memory location while the remote thread is spinning waiting for the data to arrive. When data arrives the remote thread will write back to the sender, which in turn spins waiting for the value to arrive. This essentially measures the latency of a round trip communication performed using two `mem_put` operations.

The experiment is conducted using one and two supernodes where we make random pairs of communicating processors. Once the pairing is decided processors start exchanging messages in a loop. We report averages for all communicating pairs in the histogram plots included in Figure 11 and 12. For both one and two supernodes we include results when using two underlying protocols for data exchange: RDMA and SHORT as described in Section 4. Figure 11 (a) shows the measurements for one supernode and using RDMA. This is enabled using the runtime flag `-xlpgashfiupdate`. We observe the majority of measured latencies clustering around 2.15 μs value (98% of measurements) with a second smaller clustering around 2.7 μs value (2%). The plotting scale is logarithmic to emphasize the small clusters.

In Figure 11(b) we show the same ping pong experiment, this time disabling RDMA transfer (i.e., we don't specify `-xlpgashfiupdate` flag to the runtime). The short message protocol is used in this situation. We observe in this case that the majority of the measured latencies are around 2.27 μs (93%) with two additional peaks at 1.5 μs (2%) and 3.22 μs (2%). The peak at 1.5 μs is due to the pairs communicating within the same octant where short messages translate to simple memory copies. The reported average within a drawer is 2.22 μs and between drawers is 2.36 μs.

Overall without RDMA the latency increases from 2.15 to 2.27 μs for the majority of the samples. Elsewhere we reported that the latency of simple PAMI message sends is also around



(a) Latency on 2 SNs with -xlpgashfiupdates (RDMA enabled).

(b) Latency on 2 SNs without -xlpgashfiupdates.

Figure 12: MPP Ping pong latency on two supernodes with and without -xlpgashfiupdates flag.

2 μs and this shows that the MPP ping pong testing methodology and XL UPC adds only a minimal overhead to what PAMI can provide.

For the two supernodes case included in Figure 12(a), we see two major distinct peaks at 2.13 μs and 2.37 μs. These peaks correspond to the cluster of intra supernode pairs and across super nodes pairs of communicating threads. The intra supernode time is basically the same as the time reported in the one supernode experiment (Figure 11(a)). Figure 12(b) shows the latency when using short messages and here again we notice the cluster corresponding to intra octant data exchange and the rest.

5.3 Guppie

Guppie is a variant of the HPCC RandomAccess benchmark. Guppie measures the aggregate rate of random remote updates in a large distributed array. Performance is measured in billions, or giga-updates, per second, hence the name of the benchmark. In Guppie the remote update operation is always a bit-wise XOR operation; the distributed array consists of 64-bit words and fills at least half the total available memory in the system.

Guppie performance in general is gated by (a) how fast any particular process can generate and process remote updates and (b) how fast the interconnection network can transmit the updates. The p775 system features special purpose hardware for remote updates as follows: up to four different update requests can be packed into a single 128-byte flit, reducing bandwidth requirements, and the network hardware processes remote updates when received - there are no CPU requirements on the receiving end of an update. Thus on the p775 system performance is determined by the following factors:

- **Network limits:** We have already measured these for MPP alltoall. The traffic pattern generated by Guppie is very similar to alltoall, since every Guppie thread sprays update

packets in every direction. Since we know that each update takes $128/4 = 32$ bytes on the network, we can convert alltoall aggregate bandwidth numbers directly into Guppie numbers by dividing them by 32.

- **Huge pages:** An additional constraining factor in Guppie is the fact that system memory is not accessed in a quasi-linear fashion. While MPP alltoall sends messages in contiguous multi-kilobyte chunks, Guppie sends updates to random addresses 8 bytes at a time. This puts a higher pressure on the update target's paging system. The p775 network chip has a 6144-entry page cache; when this cache is exhausted every remote update has to look up the CPU's page table, a cumbersome and time consuming process that decreases performance.

  6144 "normal" (64 KByte) pages cover much less than half the memory of a p775 octant. Thus we are forced to use 16 MByte pages for Guppie. 6144 16 MBytes pages cover 96 GBytes of the 128 GBytes per octant in p775.

- **HUB limit:** The HUB limit applies to Guppie the same way it applies to MPP alltoall. Again, bandwidth numbers have to be divided by 32.

- **Software overhead and SMT-2 operation:** Another difference between Guppie and alltoall is that UPC index arithmetic is necessary for every 8 byte update instead of e.g. once 8 KByte send. UPC index arithmetic is computationally intensive. Thus on the p775 system we never reach the HUB limit with Guppie - the UPC runtime cannot generate remote updates fast enough to saturate the network hub. We run Guppie in SMT-2 mode to double the number of threads generating addresses; still we reach no more than approximately 70% of the HUB limit. A new version of the XL UPC compiler, currently in development, reduces index arithmetic overhead and will allow substantial improvements.

Figure 13 shows measured Guppie aggregate performance in Giga updates per second on a system consisting of 1 to 32 supernodes. Unsurprisingly the figure has the same shape as the MPP alltoall measurement. On 1 and 32 supernodes respectively, performance is gated by per node throughput. Given the HUB limit - 35 GBytes/s/octant - we should expect $\frac{35}{32} \approx 1.1$ gups/octant. The actual numbers - 22 gups for 30 octants in one supernode, 650 gups for 960 octants in 32 supernodes - tell us that UPC address generation overhead limits us to between 60 and 70 % of the HUB limit.

In all other cases Guppie performance is limited by available network bandwidth. We therefore expect aggregate numbers in line with those measured for MPP alltoall (Table 1). For the 2 supernode setup, performance is gated by the 8 D-links connecting the supernodes; Table 1 lists 187 Gbytes/s aggregate bandwidth. Divided by 32, this translates into 5.87 gups, which is indeed what we measure with Guppie.

By nature in the p775 system network, doubling the number of supernodes in a measurement quadruples the cross-section of the network; we therefore expect $4\times$ performance on 4

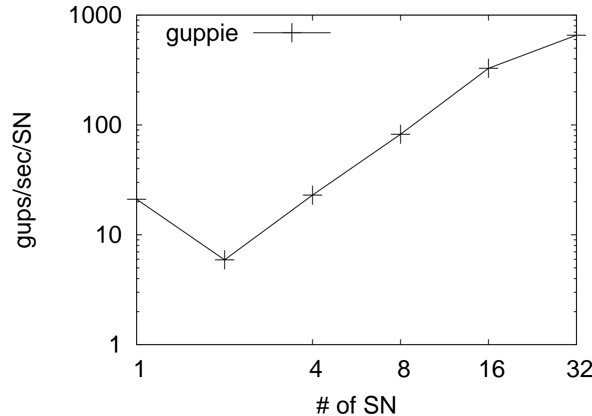


Figure 13: Guppie performance. For 1 and 32 supernodes, performance is limited by software overhead; for 2, 4, 8, 16 supernodes the network cross-section limits performance.

supernodes ($\approx 24gups$) and so on. Figure 13 bears out these expectations exactly. Performance quadruples with every doubling of the system, until at 32 supernodes the cross-section bandwidth is large enough that per-node overhead becomes the limiting factor again.

5.4 HPCC Benchmarks

We use three benchmarks from the HPC Challenge [25] suite to evaluate the performance: Global HPL (HPL), Global FFT (FFT), and EP Stream Triad (EP).

5.4.1 FFT

Global FFT performs a Discrete Fourier Transform on a one-dimensional array of complex values. The arrays in the benchmark are sized to fill 40% of the total memory of the machine. The arrays are evenly distributed across the memory of the participating processors. The cluster implementation of FFT calls for a sequence of local DFT transformations followed by global and local transpositions across the nodes.

The FFT uses local DFTs by calling the ESSL `dcft` function. In order to transpose the arrays, there are two strategies. In strategy (a) transposition is executed line by line (Point-to-Point), which tends to generate a large number of short network transactions. Strategy (b) involves first coalescing the buffers into pieces that can be exchanged over the network with a single call to all-to-all (Collective).

Figure 14 shows the results for the FFT benchmark. We compare three different versions: the Point-to-Point and the coalesced all-to-all versions. We also include a version of the Point-to-Point with the proper loop scheduling to increase the bandwidth, as discussed in

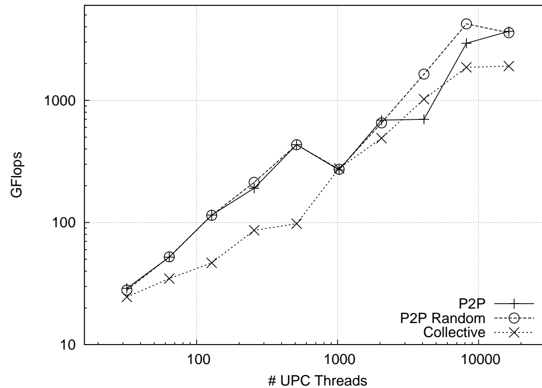


Figure 14: FFT performance.

```
upc_forall(i = 0; i<VectorSize; i++; i)
  a[i] = b[i] + alpha * c[i];
```

Figure 15: UPC version of Stream Triad.

section 5.2.1. The results show that the Point-to-Point approach is faster than the all-to-all version, due to the additional overhead of marshaling and unmarshaling of the shared data. Furthermore, the proper scheduling of loop iterations gives from $1.03X$ up to $2.3X$ increase on average in the performance. With the exception of 1024 threads, the random access is faster, from 1.10X up to 2.1X than when using the collective. Overall the curve has a similar shape as the all-to-all performance as this step dominates the execution time at scale.

5.4.2 StreamTriad

The UPC code for this benchmark has a work distribution loop. Figure 15 shows the code for the UPC version of the stream code. All array references in the `upc_forall` loop above are actually local. The shared array privatization allows the compiler to recognize that the loop does not contain any communication, and the compiler can therefore privatize the 3 array accesses in the stream kernel.

Figure 16 presents Stream performance. In the Stream benchmark, the compiler can recognize that consecutive array accesses performed by each thread have a physical distance in memory equal to the size of the array element (stride one access pattern). The compiler remaps the iteration space of the `upc_forall` loop from the UPC index space to the physical array distribution and privatizes the shared access. The optimization places the calls outside of the loop to obtain their local array address. Finally, the compiler applies the traditional stream prefetching optimization to the loop. The overall scaling is linear as expected, reaching 207.2 TB/s on 32 supernodes (30720 cores).

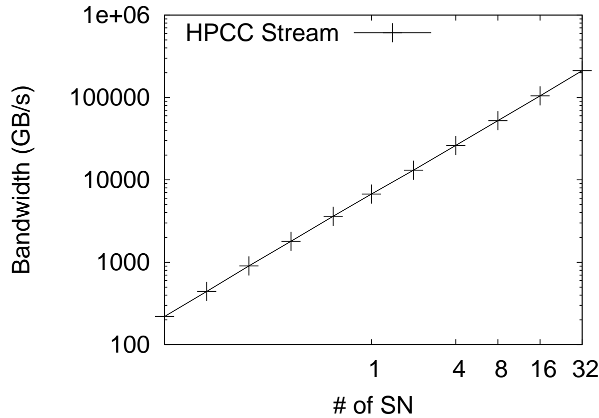


Figure 16: Stream performance.

5.4.3 HPL

The main kernel of HPL solves a dense linear system of equations using LU factorization with row partial pivoting. The basis of the implementation is single-processor blocked LU factorization code similar to that found in the LAPACK routines `dgetf2` and `dgetrf`. The benchmarks uses a distributed tiled data structure in order to facilitate parallelism and readability of the code. Therefore, the benchmark includes the tiled array library with a UPC port.

The main purpose of the tiled array library is to perform the distributed index calculation for multi-dimensional tiles laid out using a block-cyclic array distribution. This alleviates programmer effort and decreased readability in the needed (i.e modulo-arithmetic) steps to compute which tiles in a distributed array are local or remote to the given thread. The tiled array library's main interesting feature is its reliance on one-sided communication.

Figure 17(a) compares the HPL benchmark performance with the peak performance of the machine. The benchmark achieves from 46% to 65% of the peak performance of the machine. The benchmark achieves 20 GFlop/s per core in 32 UPC threads and drops to 14.36 GFlop/s in 32768 UPC threads. There are two reasons that the efficiency of the benchmark decreases as the number of UPC threads increase: (a) communication overhead and (b) load imbalance. Figure 17(b) presents the breakdowns for different numbers of UPC threads. Moreover, for 32 UPC threads, the figure shows that while we spend 80+% of execution time in computation, we still only achieve 65% of peak performance. This is due to ESSL itself only achieving 83% of peak performance.

5.5 K-Means

The K-Means benchmark is an implementation of Lloyd's algorithm [26]. The problem, in a nutshell, is to cluster a set of N points $\{x_1, x_2, ...x_N\}$ in D-dimensional Euclidean space into

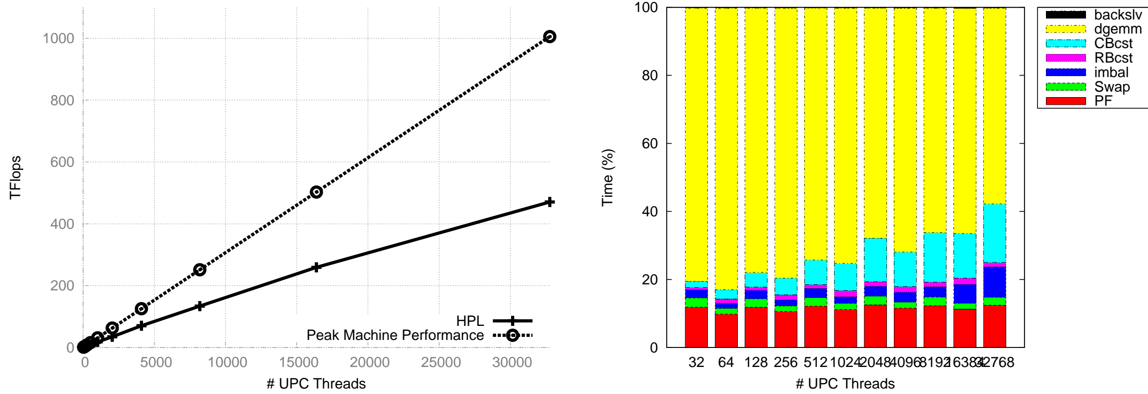


Figure 17: HPL results compared with the peak machine performance (left) and breakdowns (right).

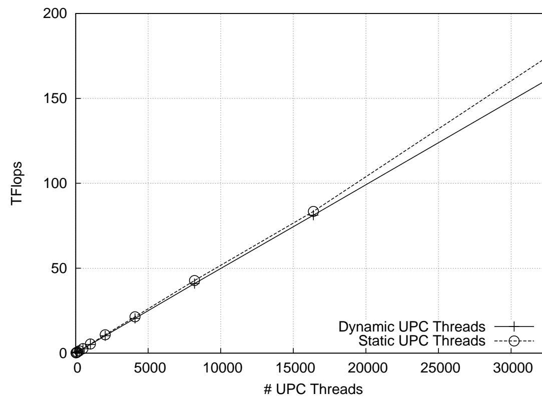


Figure 18: K-Means performance.

K sets $\{S_1, S_2, ...S_k\}$ so as to minimize the sum of Euclidean distances from the points to the clusters' centers $\mu_j, j \in 1..k$, i.e. calculate:

$$\arg\min_S \sum_{i=1}^{k} \sum_{x_j \in S_i} ||x_j - \mu_i||^2$$

A *naive* implementation of the algorithm is fairly computationally intensive, as the classification phase involves the calculation of the Euclidean distance between every pair of points and centroids, resulting in $O(N \times K \times D)$ complexity. There are complicated tree-based algorithms in existence that cut down computation to a fraction of the above number by eliminating redundant distance calculations. However, as far as we are aware, a three-level nested loop computing pairs of Euclidean distances exists in some form in every variant of optimized code, as shown in Figure 19.

```
 1  do p=1, N
 2    kmin=-1; dmin=\inf;
 3    do k=1, K
 4      d0 = 0
 5      do d=1, D
 6        d0 = d0 + (points[p][d]-clusters[k][d])^2
 7      end do
 8      if (d0 < dmin) { dmin = d0; kmin = k; }
 9    end do
10    nearest[p] = kmin;
11  end do
```

Figure 19: K-means: pseudo-code for classification phase.

Figure 18 shows the performance of the k-means benchmark. We measured the performance for two versions of code, to compare the compilation with static and dynamic number of threads. The results show that the benchmark scales well. Moreover, XL UPC is able to simplify the shared pointer arithmetic when the user specifies the number of UPC threads at compile time. Thus for 32768 UPC threads the static compiled version is 9% faster than the version with dynamic number of threads.

The K-means benchmark has three different dis phases. First the benchmark makes the classification, then calculates the average, first locally and then globally through reduction, and finally calculates the difference between successive operations. The XL UPC compiler privatizes most of the shared accesses except one shared read in the classification and one in calculation of average. The shared reference in the average calculation is actual local, however XL UPC misses the opportunity to privatize due to the complexity of the index expression.

5.6 UTS

The Unbalanced Tree Search benchmark [33, 34] belongs in the category of state-space search problems. The Unbalanced Tree Search benchmark measures the rate of traversal of a tree generated on the fly using a splittable random number generator. The overall task of a state-space search problem is to calculate some metric over the set of all good configurations – e.g. the number of such configurations, the ”best” configuration etc.

The challenge of this benchmark category, is to parallelize the computation across a potentially very large number of hosts while achieving high parallel efficiency. The computation is typically initiated at a single host with the root configuration. A good solution must quickly divide up the work across all available hosts and solve the global load-balancing problem. It must ensure that once a host runs out of work it is able to quickly find work, if in fact work exists at any host in the system.

In Figure 20 we show the performance for the UTS benchmark using a mixed weak and strong scaling approach. We fix a particular tree size and for that we perform a strong scaling

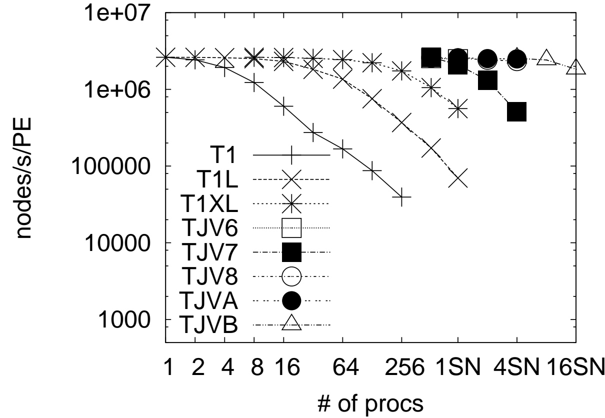


Figure 20: UTS performance.

experiment and each line in the plot corresponds to a strong scaling experiment. Subsequently we increase the tree size by a power of two and we perform another strong scaling experiment starting with double the number of processors in the previous run. When doubling the input size while doubling the number of threads we essentially achieve the same amount of load per UPC thread. From the plot we can see good weak scaling as we increase the number of processors and the problem size. The benchmark achieves 2.5 million nodes traversed per second per thread. The same good performance is obtained as we scale up to 16 supernodes.

5.7 Sobel

To evaluate the performance of a parallel machine using fine-grained communication we use the Sobel benchmark. The Sobel benchmark computes an approximation of the gradient of the image intensity function, performing a nine-point stencil operation. In the UPC version [14] the image is represented as a two-dimensional shared array and the outer loop is a parallel *upc_forall* loop. The evaluation uses a constant data set size per thread (weak scaling), thus the overall problem size increases with the number of threads. We start from 32768×32768 as input image size in 32 UPC threads, up to 1048576×1048576 using 32768 UPC threads. The maximum allocated memory is two TBytes in 32768 UPC threads.

The low communication efficiency of fine grain data accesses has been identified by many researchers [8, 10] as one of the main bottlenecks of PGAS languages. The XL UPC compiler coalesces shared references, when possible, to decrease the overhead of many runtime calls and increase the network efficiency. The Shared Object Access Coalescing optimization [11] reduces the number of remote and local shared accesses. The optimization identifies opportunities for coalescing, manages the temporary buffers, including modifying shared references to read-from/write-to the buffers, and inserts calls to the coalescing runtime functions. The

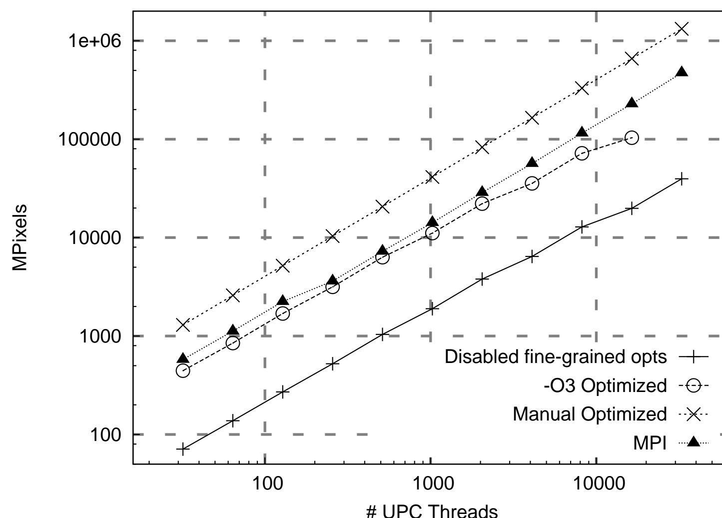


Figure 21: Sobel performance compared with MPI.

runtime allows the compiler to specify a shared symbol, a stride to use when collecting shared references, the number of shared references to collect and the temporary buffer to use. The compiler checks some requirements for coalescing the elements: (i) all shared references must have the same base symbol; (ii) all candidates must be owned by the same UPC thread; (iii) all candidates must be either read or write.

Figure 21 presents the performance numbers for the Sobel benchmark in mega-pixel per second. We use four benchmark versions: (i) Fine-grained with disabled the static coalescing and privatization; (ii) Fine-grained with all optimizations enabled (-O3); (iii) Manual optimized with coarse grain transfers; (iv) MPI version with coarse grain accesses.

The fine-grained optimized version is around $6X$ faster than the non-optimized version. The XL UPC compiler applies two optimizations for fine grained accesses, privatization and static coalescing. The Sobel benchmark communicates with the neighboring UPC threads only at the start and end of the computation. One of the goals for the optimizations is to provide comparable performance to the hand-optimized version with coarse grain accesses. The fine-grained version of the Sobel benchmark achieves from 45% up to 77% of the performance of the MPI version. One interesting observation is that the UPC hand-optimized version is from $2.1X$ up to $2.9X$ faster than the MPI version, because of the better overlap of one-sided communication. The MPI version requires the synchronization of the two processes to transfer the data. Thus, the synchronization with large number of threads is harder to achieved in low latency, resulting in better speedup for the UPC versions.

1 Megapixel = 1000000 pixels

6 Related Work

Evaluating the performance of various UPC implementations has been the focus of numerous publications [19, 27, 32]. These discuss either point to point operations like shared memory reads or writes, or whole applications or benchmarks (NPB, BenchC, etc).

Previous UPC performance evaluations [45, 15] have focused either on basic data movement primitives, NPB NAS benchmarks [22], or small applications, such as sobel [14] In [28] the authors compare the NPB NAS benchmarks in MPI and UPC. The results show that MPI versions is always better than the UPC versions. Authors in [18] present the performance of the Berkeley compiler, compared with the HP UPC compiler [17]. The performance of the open source compiler is similar or better than the commercial HP compiler. The authors in [21] use the MuPC [29] runtime system to evaluate the collective calls.

In [23] the authors perform an in depth performance study of the P7IH various intra and inter node communication links describing achievable latencies and bandwidths. They measure the performance using simple point to point kernels and discuss it relative to the maximum specified values.

In [40] the authors present a comprehensive evaluation of two UPC compilers, GNU UPC and Berkley UPC, on a cluster of SMPs evaluating distributed, shared memory and hybrid mode of executions. The authors deployed their own kernels similar to Intel MPI collectives benchmark. In [27] the UPC (GNU UPC and Berkley UPC) collectives are evaluated in the context of a multicore architecture. The authors of [30] discuss the performance of UPC collectives tuned for Linux/Myrinet and compare them with the default Berkley implementation.

Various papers [36] discuss implementation trade-offs and performance improvements for various UPC collectives. The authors of [42] present a set of collectives optimized for PERCS architecture. While they were initially implemented as part of the IBM XL UPC runtime they were migrated into PAMI, the IBM messaging library.

Optimizations for data coalescing using static analysis exist in Unified Parallel C [8, 11] and High Performance Fortran [7, 16]. A compiler uses data and control flow analysis to identify shared accesses to specific threads and creates one runtime call for accessing the data from the same thread. Another approach for minimizing the communication latency in the PGAS programming model is to split request and completion of shared accesses. The technique is called either "split-phase communication" [9] or "scheduling" [12, 16]. Other techniques to reduce communication latency at runtime include decomposing the coarse-grained transfer segments, into strips and transferring them in pipeline [20], and the usage of a software-cache mechanism [44].

Moreover, the inspector-executor strategy is a well-know optimization technique in PGAS languages. There are approaches [24] for compiler support using a global name space programming model, or language-targeted optimizations such as: High Performance Fortran [5, 43], Titanium language [39], X10 [13], Chapel [37], and UPC [3]. The inspector loop analyzes the communication pattern and the executor loop performs the actual communication based on the results of the analysis performed in the inspector loop.

7 Conclusion

In this paper we included a comprehensive discussion on the XL UPC compiler, the optimization employed by the high level optimizer and runtime, and evaluated the performance of various benchmarks. We have shown that XL UPC provide scalable performance when using more than 30000 cores on the PERCS machine and that the runtime efficiently exploit the underlying PAMI communication library and the hardware features of the machine like RDMA and accelerated collectives.

We also describes pitfalls of the PGAS programming model employed by UPC and how they can be either addressed at the programming level or with additional compiler optimizations. We highlighted the key characteristics of the PERCS architecture emphasizing the low latency of the network and the huge bandwidth available on the machine.We have shown that XL UPC can achieve up to 60% using an all to all pattern which is common in a large number of applications like sort, FFT matrix transposition and others. While the current performance achievable through software on the architecture is notable relative to other supercomputer architectures, there is still room for improvement to further reduce latency of collectives and further improve bandwidth utilization.

References

[1] The berkeley upc user's guide version 2.16.0. http://upc.lbl.gov.

[2] *Parallel Environment Runtime Edition for AIX, PAMI Programming Guide, Version 1 Release 1.0, IBM*.

[3] M. Alvanos, M. Farreras, E. Tiotto, and X. Martorell. Automatic Communication Coalescing for Irregular Computations in UPC Language. In *Conference of the Center for Advanced Studies*, CASCON '12.

[4] R. X. Arroyo, R. J. Harrington, S. P. Hartman, and T. Nguyen. Ibm power7 systems. *IBM Journal of Research and Development*, 55(3):2:1 –2:13, may-june 2011.

[5] P. Brezany, M. Gerndt, and V. Sipkova. SVM Support in the Vienna Fortran Compilation System. Technical report, KFA Juelich, KFA-ZAM-IB-9401, 1994.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[7] D. Chavarria-Miranda and J. Mellor-Crummey. Effective Communication Coalescing for Data-Parallel Applications. In *In Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 14–25, 2005.

[8] C. I. W. Chen and K. Yelick. Communication optimizations for fine-grained upc applications. In *In 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[9] W.-Y. Chen, D. Bonachea, C. Iancu, and K. Yelick. Automatic nonblocking communication for partitioned global address space programs. In *Proceedings of the 21st annual international conference on Supercomputing (ICS '07)*, pages 158–167.

[10] Christopher Barton, Calin Cascaval, and Jose Nelson Amaral. A characterization of shared data access patterns in upc programs. In *In Workshop on Languages and Compilers and Parallel Computing (LCPC)*, pages 111–125, New Orlean, LO, November 2006.

[11] Christopher Barton, George Almasi, Montse Farreras, and Jose Nelson Amaral. A Unified Parallel C compiler that implements automatic communication coalescing. In *14th Workshop on Compilers for Parallel Computing*, 2009.

[12] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A Multi-Platform Co-Array Fortran Compiler. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 29–40.

[13] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: Programming for hierarchical parallelism and non-uniform data access. In *Proceedings of the International Workshop on Language Runtimes, OOPSLA*, 2004.

[14] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: a NPB experimental study. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–26, Los Alamitos, CA, USA, 2002.

[15] T. A. El-Ghazawi, F. Cantonnet, Y. Yao, S. Annareddy, and A. S. Mohamed. Benchmarking parallel compilers: a UPC case study. *Future Gener. Comput. Syst.*, 22(7):764–775, Aug. 2006.

[16] M. Gupta, E. Schonberg, and H. Srinivasan. A unified framework for optimizing communication in data-parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 7:689–704, 1996.

[17] Hewlett-Packard. Compaq UPC compiler, 2011. http://www.hp.com/go/upc.

[18] P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 63–73, New York, NY, USA, 2003. ACM.

[19] P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the berkeley upc compiler. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 63–73, New York, NY, USA, 2003. ACM.

[20] C. Iancu, P. Husbands, and P. Hargrove. HUNTing the Overlap. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 279–290.

[21] A. K. Jain, L. Hong, and S. Pankanti. Biometrics: Promising frontiers for emerging identification market. Technical Report MSU-CSE-00-2, Department of Computer Science, Michigan State University, East Lansing, Michigan, February 2000.

[22] H. Jin, R. Hood, and P. Mehrotra. A practical study of UPC using the NAS Parallel Benchmarks. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 8:1–8:7, New York, NY, USA, 2009.

[23] D. Kerbyson and K. Barker. Analyzing the performance bottlenecks of the power7-ih network. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 244 –252, sept. 2011.

[24] C. Koelbel and P. Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Trans. Parallel Distrib. Syst.*, 2, 1991.

[25] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06. ACM, 2006.

[26] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. L. Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[27] D. A. Mallón, A. Gómez, J. C. Mouriño, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, R. Doallo, and B. Wibecan. Upc performance evaluation on a multicore system. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 9:1–9:7, New York, NY, USA, 2009. ACM.

[28] D. A. Mallón, A. Gómez, J. C. Mouriño, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, R. Doallo, and B. Wibecan. UPC performance evaluation on a multicore system. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 9:1–9:7, New York, NY, USA, 2009. ACM.

[29] Michigan Technological University. UPC Projects, 2011. http://www.upc.mtu.edu.

[30] A. Mishra and S. Seidel. High performance unified parallel c (upc) collectives for linux/myrinet platforms, 2004.

[31] MPI Forum. Mpi:a message-passing interface standard (version 1.1). technical report (june 1995), January 2012. available at: http://www.mpi-forum.org (Jan. 2012).

[32] R. Nishtala, G. Almasi, and C. Cascaval. Performance without pain = productivity: data layout and collective communication in upc. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 99–110, New York, NY, USA, 2008. ACM.

[33] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An Unbalanced Tree Search Benchmark. In G. Almási, C. Cascaval, and P. Wu, editors, *LCPC*, volume 4382 of *Lecture Notes in Computer Science*, pages 235–250. Springer, 2006.

[34] J. Prins, J. Huan, B. Pugh, and C. wen Tseng. UPC implementation of an unbalanced tree search benchmark, Univ. North Carolina at Chapel Hill. Technical report, 2003.

[35] R. Rajamony, L. B. Arimilli, and K. Gildea. PERCS: The IBM POWER7-IH high-performance computing system. *IBM Journal of Research and Development*, 55(3):3:1 –3:12, may-june 2011.

[36] R. A. Salama and A. Sameh. Potential performance improvement of collective operations in upc. In *PARCO'07*, pages 413–422, 2007.

[37] A. Sanz, R. Asenjo, J. Lopez, R. Larrosa, A. Navarro, V. Litvinov, S.-E. Choi, and B. L. Chamberlain. Global data re-allocation via communication aggregation in Chapel. In *SBAC-PAD*. IEEE Computer Society, 2012.

[38] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. Ibm power7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1:1 –1:29, may-june 2011.

[39] J. Su and K. Yelick. Automatic Support for Irregular Computations in a High-Level Language. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[40] G. L. Taboada, C. Teijeiro, J. Tourino, B. B. Fraguela, R. Doallo, J. C. Mourino, D. A. Mallon, and A. Gomez. Performance evaluation of unified parallel c collective communications. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, HPCC '09, pages 69–78, Washington, DC, USA, 2009. IEEE Computer Society.

[41] G. Tanase, G. Almási, E. Tiotto, Y. Liu, H. Xue, and C. Archer. Performance Analysis of the IBM xlUPC Collective Operations on PERCS. In *The 6th Conference on Partitioned Global Address Space Programming Models*, PGAS'12, 2012.

[42] G. I. Tanase, G. Almási, H. Xue, and C. Archer. Composable, non-blocking collective operations on power7 ih. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 215–224, New York, NY, USA, 2012. ACM.

[43] D. Yokota, S. Chiba, and K. Itano. A New Optimization Technique for the Inspector-Executor Method. In *International Conference on Parallel and Distributed Computing Systems*, pages 706–711, 2002.

[44] Z. Zhang, J. Savant, and S. Seidel. A UPC Runtime System Based on MPI and POSIX Threads. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:195–202, 2006.

[45] Z. Zhang and S. Seidel. Benchmark measurements of current UPC platforms. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, page 8 pp., april 2005.

A Compiler, runtime and environment configuration

A.1 MPP Benchmark

Compiler options used: `-O3 -qarch=pwr7 -qtune=pwr7`
Runtime options: `-xlpgashfiupdate -xlpgashugepages -msg_api pgas`
Load Leveler Job File:

```
all2all
MP_CHECKPOINT=no
MP_RDMA_ROUTE_MODE=hw_direct_striped
MP_SHARED_MEMORY=yes
MP_USE_BULK_XFER=yes

bcast
MP_CHECKPOINT=no
MP_SHARED_MEMORY=yes
MP_USE_BULK_XFER=yes
XLPGAS_PAMI_BROADCAST=I1:Broadcast:P2P:P2P

pingpong, barrier, reduce
MP_CHECKPOINT=no
MP_SHARED_MEMORY=yes
MP_USE_BULK_XFER=yes
```

A.2 Guppie

Compiler options used: `-O3 -DVLEN=1 -DLTABSIZE=${LTABSIZE}`
Runtime options: `-xlpgashfiupdate -xlpgashugepages -msg_api pgas`
Load Leveler Job File:

```
SMT2
node_topology = island
job_type = parallel
network.pgas = sn_single,not_shared,us
checkpoint = no
bulkxfer = yes
task_affinity = cpu
collective_groups = 4
```

A.3 uts

Compiler options used: `-O3 -qarch=pwr7 -qtune=pwr7 -DMAIN_TIMING -DMAXSTACKDEPTH=`
Runtime options: `-xlpgasautobind -xlpgasqlock`
Load Leveler Job File:

```
MP_ADAPTER_USE=shared
MP_CHECKPOINT=no
MP_CPU_USE=multiple
MP_DEVTYPE=hfi
```

A.4 Sobelk

Compiler options used: `-O3 -qarch=pwr7 -qtune=pwr7 -qupc=threads=${THREADS}`
Runtime options: `-xlpgasautobind -xlpgashfiupdate`
Load Leveler Job File:

```
MP_ADAPTER_USE=dedicated
MP_CHECKPOINT=no
MP_CPU_USE=unique
```

A.5 Stream

Compiler options used: `-O3 -qarch=pwr7 -qtune=pwr7 -qdebug=finalc` **Runtime options**: `-xlpgasbind=auto -xlpgashfiupdate -msg_api pgas`

Load Leveler Job File:

```
MP_ADAPTER_USE=dedicated
MP_CPU_USE=unique
```

A.6 FFT

Compiler options used: `-q64 -O3 -qarch=pwr7 -qtune=pwr7 -qprefetch=aggressive -qinline -qhot`
Runtime options: `-xlpgashfiupdate -xlpgasautobind`
Load Leveler Job File:

```
MP_ADAPTER_USE=dedicated
MP_CPU_USE=unique
```