UNIFIED PARALLEL C - A STUDY AND PERFORMANCE COMPARISON WITH MPI (PROJECT REPORT)

Pawas Ranjan, P. Sadayappan
Department of Computer Science and Engineering, The Ohio State University
ranjan, saday@cse.ohio-state.edu

***Abstract* – With serial execution hitting its performance limits, the world has now started to turn to parallelism over the last few years. Many new programming models and languages have been suggested and created to this effect. Of these, the MPI programming language follows the message passing model while UPC aims for a partitioned global address space model. Over the last few years, MPI has become extremely popular due to the amount of parallelism it offers, but at the price of increased complexity in programming. UPC, on the other hand, tries to keep the performance of MPI and offer an easier and more intuitive programming environment. The purpose of this project is to study and compare these two languages that are based on two different programming models and to compare their performance and scalability on distributed systems.**

I. INTRODUCTION TO UPC

A. The Partitioned Global Address Space (PGAS)

The UPC models a PGAS and is, therefore, called a PGAS language. PGAS languages are explicitly parallel, i.e. The programmer has to explicitly specify where what and, sometimes, even how to parallelize a given code.

The amount of parallelism is fixed at the startup. This allows the programmer, as well as the compiler, to make certain optimizations based on the pre-knowledge of the number of threads available.

These languages, like MPI, are SPMD (Single Program Multiple Data) based. This means that each thread runs the exact same program, but operates on different data sets.

As the name suggests, these languages presents the programmer with a global address space memory model which makes it easier to represent distributed data structures. This address space has a two-level hierarchy and is logically partitioned into local and remote memory.

A point to be noted is that there need not be a physical globally shared memory in the system. PGAS uses pointers and division of per-node memory into private and shared regions to cleverly produce the illusion of global memory.

PGAS languages also provide constructs to easily represent distributed data structures and their split-up across the threads. This allows the programmer to control performance critical decisions in a simple way.

Besides C, the PGAS model has also been implemented for parallel programming in FORTRAN (CAF) and Java (Titanium).

A pictorial view of the PGAS model is shown in fig. 1

B. Execution model

UPC has a very simplistic execution model. All codes are SPMD parallel with the amount of parallelism fixed at program startup. This is available through a constant called THREADS within the program.

Each thread also has a unique id assigned to it, and can be found in the MYTHREAD constant.

UPC also provides functions and constructs for declaring shared variables data structures, thread synchronization, collective operations, work division and dynamic shared memory allocation. These will be discussed in the remaining parts of section 1.

C. Shared variables and arrays

Normally, variables and objects created by a program or function are allocated in the private memory space of each thread or function. UPC, however, provides keywords to declare shared variables which reside in the global memory.

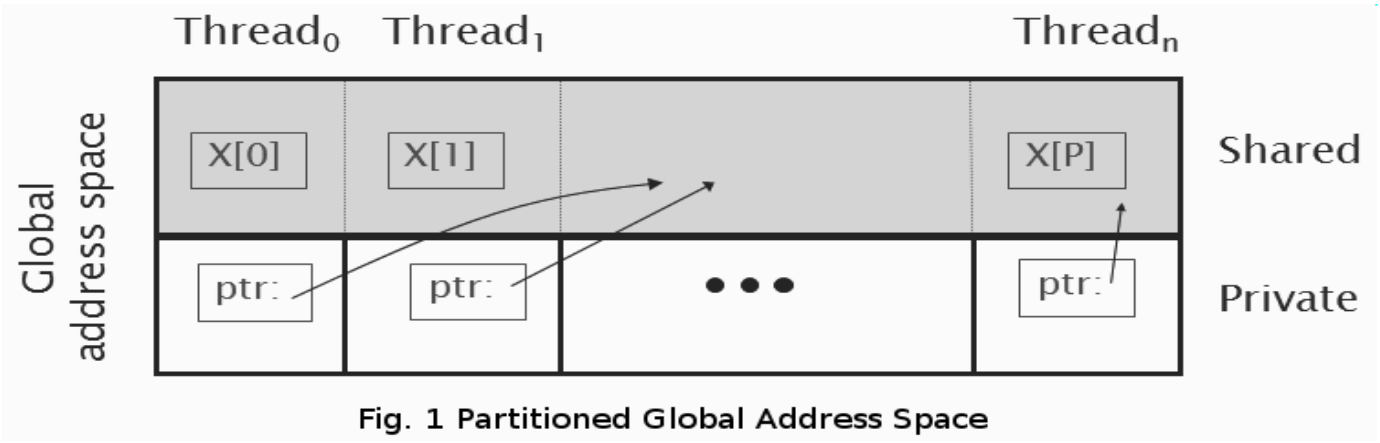


Fig. 1 Partitioned Global Address Space

Shared scalars are allocated only once, and reside in the shared memory available on thread 0.

For arrays, however, the programmer can decide on type type of distribution desired across the threads. This means that although the entire array is shared, each thread has an affinity to some of the elements since they reside on the global memory space allocated in the memory of the node on which the thread runs.

The layout can be cyclic, blocked or striped (blocked cyclic). For 2-D arrays, it can imply 1-D row based, 1-D column based, row-block, or column blocked distributions. The possibilities are presented in the table below:

Declaration	*Meaning*
`shared int ours;`	Shared scalar on thread 0
`shared int a[N];`	Cyclic distributed array
`shared [] int a[N];`	Shared array on thread 0
`shared [*] int a[N];`	Block distributed array
`shared [A] int a[N];`	Block distributed with fixed block size

Table 1. Declaring shared variables

D. Synchronization

UPC provides two basic synchronization primitves. These are barriers and locks. Barriers themselves are convention barriers in which a thread blocks until all other threads reach that point in code.

There is also available another type of barrier called a split phase barrier which can be used to overlap the wait-time on a barrier with later computations that are not dependent on the barrier. This barrier utilizes notify and wait calls.

UPC also provides a fence primitive that can be used to enforce strict consistency.

Locks in UPC are shared. They locks can be used to ensure mutual exclusion of critical sections. Based on the needs, the pointer to the lock can be given to all threads or to just one thread. In either case, the lock resides in shared memory.

Primitive	*Meaning*
`upc_barrier`	Block until all other threads arrive
`upc_notify`	Notify that thread is ready for barrier
`upc_wait`	Wait for others to be ready
`upc_fence`	Ensures that all shared references issued before `upc_fence` are complete

Table 2. Synchronization primitives

A process can perform computation unrelated to the barrier between `upc_notify` & `upc_wait`.

Statement	*Meaning*
`upc_lock_t *lck;`	Declare a lock
`lck = upc_all_lock_alloc();`	Allocate lock, return pointer to all threads
`lck = upc_global_lock_alloc();`	Allocate lock, return pointer to thread 0 only
`upc_lock(lck);`	Lock
`upc_unlock(lck);`	Unlock
`upc_lock_free(lck);`	Free the lock

Table 3. Using locks in UPC

E. Collectives

Collective operations are executed by all threads are are used for communication between threads. The collectives can be data movement collectives like scatter, gather, broadcast, permute etc. or computation collectives like reduce and prefix sum. The George Washington University maintains a library of UPC collectives. More details can be found on [1].

F. Bringing it all together: PI estimation

The following codes a simple UPC program to estimate the value of PI.

```
#include <upc_relaxed.h>
#include <stdio.h>

shared [THREADS] double sum[THREADS];

int main () {
  int i;
  double pi, step, trials, my_trials;
  double x, tmp;

  trials = 1024000.0;
  step = 1.0/trials;
  my_trials = trials/THREADS;
  tmp = (double)0.0;

  for(
  i = MYTHREAD*my_trials;
  i < (MYTHREAD+1)*my_trials;
  i++) {
    x = ((double)i-0.5)*step;
    tmp += 4.0/(1.0 + (x*x));
  }
  sum[MYTHREAD] = tmp;

  upc_barrier;

  if (MYTHREAD == 0) {
    for(i = 0; i < THREADS; i++)
      pi += sum[i]*step;
    printf("PI = %f.\n", pi);
  }

  return 0;
}
```

Listing 1. UPC code for PI estimation

Here, we use a shared array sum to store the partial results from each thread and then use thread 0 to access all these and then sum them up to give us the result. Due to the way the array is declared, a thread `i` gets element `sum[i]` assigned to it.

An alternate method s to eliminate the shared array and instead use a collective reduce operation to obtain the result from all the arrays. Note that the collective has an implicit barrier and hence does not need an explicit `upc_barrier`. It however is important in Listing 1 to make sure that all the values have been filled before thread 0 starts accessing it.

```
#include <upc_relaxed.h>
#include <bupc_collectivev.h>
#include <stdio.h>

int main () {
  int i;
  double pi, step, trials, my_trials;
  double x, sum;

  trials = 1024000.0;
  step = 1.0/trials;
  my_trials = trials/THREADS;
  sum = (double)0.0;

  for(
  i = MYTHREAD*my_trials;
  i < (MYTHREAD+1)*my_trials;
  i++) {
    x = ((double)i-0.5)*step;
    sum += 4.0/(1.0 + (x*x));
  }

  sum = sum*step;

  pi=bupc_allv_reduce(double,sum,0,UPC_ADD);

  if (MYTHREAD == 0)
    printf("PI = %f.\n", pi);

  return 0;
}
```

Listing 2. using a collective reduce

G. Work distribution: `upc_forall`

Consider the code given in Listing 3 for a simple vector addition. Once the sharing of the data has been decided, we control which thread executes which iteration by using and if statement. This ensures that a particular iteration `i` is performed by the thread only if it owns the corresponding elements of the arrays (viz. `sum[i]`, `v1[i]` and `v2[i]`.

UPC provides a special for loop work division construct which takes an extra parameter called affinity to decide the iterations executed by each thread. In other words, a Thread executes an iteration only if it has an

affinity to the object passed as affinity parameter to `upc_forall`. The code is Listing 4 is equivalent to the one in Listing 3.

```
#include <upc_relaxed.h>

#define N 100*THREADS

shared int v1[N], v2[N], sum[N];

int main() {
  int i;
  for(i = 0; i < N; i++)
    if(MYTHREAD == i%THREADS)
      sum[i]=v1[i]+v2[i];
}
```

Listing 3. Vector addition in parallel

The affinity parameter `i` can also be replaced by `&sum[i]` or `&v1[i]` or `&v2[i]`.

```
#include <upc_relaxed.h>

#define N 100*THREADS

shared int v1[N], v2[N], sum[N];

int main() {
  int i;
  upc_forall(i = 0; i < N; i++, i)
    sum[i]=v1[i]+v2[i];
}
```

Listing 4. Vector addition using `upc_forall`

H. Notes on pointers

Pointers in UPC are broadly classified into four different types based on where they reside and where they point to. Table 4 gives these categories and how to create these pointers.

		Where does the pointer point?	
		Local	Shared
Where does the pointer reside?	Private	`int *p1;`	`int *shared p3;`
	Shared	`shared int *p2;`	`shared int *shared p4;`

Table 4. Declaring pointers in UPC

Generally, using shared pointers to private memory (type p2) is not recommended. Pointer arithmetic in UPC is powerful enough to implicitly support blocked and non-blocked array distributions. Also, casting of shared pointers to private is allowed, but not vice versa.

Casting of shared to local is well defined only if the object pointed to by the pointer to shared has affinity with the thread performing the cast.

I. Dynamic memory allocation

UPC provides collective and non-collective versions of functions for dynamic shared memory allocation.

The non-collective version is called by one thread only and gets back a pointer to a contiguous block of shared memory.

The collective version, on the other hand, is expected to be called by every thread. The call returns the same pointer to each thread, and the memory is allocated on per-thread shared basic with a fixed block size passed to the function.

The syntax for the two functions is given in Listing 5. Once done, the memory can be freed. Note that this function is not collective and has to be executed on every thread for a shared data structure.

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

shared void *upc_all_alloc (
             size_t nblocks, size_t nbytes);


nblocks : number of blocks
nbytes  : block size

void upc_free (shared void* ptr)
```

Listing 5. Dynamic memory allocation

J. Consistency models

A memory consistency model defines the order in which one thread may see another threads accesses to memory. There are two different consistency models available in UPC viz. strict and relaxed, and can be enforced by including `upc_strict.h` or `upc_relaxed.h` header file, respectively.

With the strict model, all accesses will always appear in order. On the other hand, accesses may appear out of order to other threads in the relaxed model.

It is usually preferred to use a relaxed model and enforce strictness only when necessary since the strict

model disables all compiler optimizations.

```
/* Initialize MPI */
int MPI_Init(int *argc, char **argv)

/* Determine number of processes within  a
communicator */
int MPI_Comm_size(MPI_Comm comm, int *size)

/* Determine processor rank within a
communicator */
int MPI_Comm_rank(MPI_Comm comm, int *rank)

/* Exit MPI */
MPI_Finalize()

/* Send a message */
int MPI_Send(void *buf, int count,
  MPI_Datatype datatype, int dest, int tag,
  MPI_Comm comm)

/* Receive a message */
int MPI_Recv(void *buf, int count,
  MPI_Datatype datatype, int src, int tag,
  MPI_Comm comm, MPI_Status *status)

/* Scatter */
int MPI_Scatter(void* sendbuf,
  int sendcount, MPI_Datatype sendtype,
  void* recvbuf, int recvcount,
  MPI_Datatype recvtype, int root,
  MPI_Comm comm);

/* Gather */
int MPI_Gather(void* sendbuf,
  int sendcount, MPI_Datatype sendtype,
  void* recvbuf, int recvcount,
  MPI_Datatype recvtype,int root,
  MPI_Comm comm);

/* Prefix Sum */
int MPI_Scan(void* sendbuf, void* recvbuf,
  int count, MPI_Datatype datatype,
  MPI_Op op,MPI_Comm comm);

/*Reduce */
int MPI_Reduce(void* sendbuf,
  void* recvbuf,int count,
  MPI_Datatype datatype,MPI_Op op,
  int root,MPI_Comm comm);

/* All Gather */
int MPI_Allgather(void* sendbuf,
  int sendcount, MPI_Datatype sendtype,
  void* recvbuf, int recvcount,
  MPI_Datatype recvtype, MPI_Comm comm);
```

Listing 6. MPI subroutines

```
#include "mpi.h"
#include <stdio.h>

int main (int argc, char **argv) {
  int i;
  int THREADS, MYTHREAD;
  double pi, step, trials, my_trials
  double x, sum;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &THREADS);
  MPI_Comm_rank(MPI_COMM_WORLD, &MYTHREAD);

  trials = 1024000.0;
  step = 1.0/trials;
  my_trials = trials/THREADS;
  sum = (double)0.0;

  for(
  i = MYTHREAD*my_trials;
  i < (MYTHREAD+1)*my_trials;
  i++) {
    x = ((double)i-0.5)*step;
    sum += 4.0/(1.0 + (x*x));
  }

  MPI_Barrier(MPI_COMM_WORLD);

  MPI_Reduce(&sum,&pi,1,MPI_DOUBLE, MPI_SUM,
                        0, MPI_COMM_WORLD);

  if (MYTHREAD == 0)
    printf("PI = %f.\n", pi);

  MPI_FINALIZE();

  return 0;
}
```

Listing 7. PI estimation in MPI

K. Hand-tuning UPC code

Some basic guidelines for increasing parallel performance are:

- Use local pointers instead of shared pointers when dealing with local shared data, through casting and assignments
- Use block copy instead of copying elements one by one with a loop (`upc_memget`)
- Overlap remote accesses with local processing using split-phase barriers
- Improve single-node performance by using standard libraries

II. Brief Review of MPI

MPI follows an SPMD message passing based parallel programming model. There is no concept of a shared memory and all communications and synchronizations in MPI occur through explicit message passing. Messages can be point-to-point or collective.

For point-to-point messages, any MPI_Send call from sender must have a corresponding MPI_Recv called by the receiver. Although MPI is extremely powerful, this constraint makes it harder to code and debug programs in MPI.

Some of the commonly used MPI functions are presented in Listing 6, followed by a simple PI estimation program in Listing 7. An exhaustive list of MPI subroutines can be found on [2]. All commutation subroutines have blocking and non-blocking versions.

III. Installing and Using UPC

There are many implementations of UPC available. The following is an incomplete list of various versions available:

- Berkeley UPC - UC, Berkeley's implementation using GASNet
- HP UPC - for all HP-branded platforms, including Tru64, HPUX and Linux systems
- Cray UPC - for Cray X1 and future Cray vector-family platforms
- GCC UPC - for x86, SGI IRIX, Cray T3E
- IBM UPC - for IBM Blue Gene and AIX SMP's

We installed Berkeley UPC 2.6.0 on Glenn, which is an OSC production cluster. The cluster contains dual-core Opterons connected together by Infiniband. Berkeley UPC uses GASNet as its underlying communication layer, which, in turn, needs a working implementation of MPI in order to spawn threads on remote nodes.

Also, Berkeley UPC utilizes an HTTP-based UPC-to-C translator. It connects to a remote server via HTTP to convert the UPC code into C and then compiles it locally. Alternately, one can compile and use a local translator or run it on an HTTP server node within the distributed system. We used the default HTTP-based UPC-to-C translator.

Steps for compiling Berkeley UPC 2.6.0 are presented in Listing 8. A more comprehensive set of instructions can be found on [3].

0. Get Berkeley UPC 2.6.0

```
# wget -c
http://upc.lbl.gov/download/release/berkeley
_upc-2.6.0.tar.gz
```

1. Extract

```
# tar -zvxf berkeley_upc-2.6.0.tar.gz
```

2. Obtain an interactive node

```
# qsub -I nodes:2:ppn:2
```

3. change directory

```
# cd  berkeley_upc-2.6.0
```

4. configure UPC

```
#  ./configure CC=mpicc CXX=mpiCC -disable-
debug -disable-udp -disable-aligned-segments
--prefix=$HOME/opt
```

the configure script should automatically detect the network API being used and locate its libraries. Also, nabling debug, udp or aligned-segments may break the package.

5. make the package

```
# make
```

6. install the package

```
# make install
```

Listing 8. Compiling Berkeley UPC 2.6.0

This creates the Berkeley UPC compiler called `upcc` in `$HOME/opt/bin`. To use UPC, simply create a file using vi, or any other text editor of choice, save it with a `.upc` extension and follow the steps in Listing 9.

1. get interactive nodes

```
# qsub -I -l nodes=8:ppn=2
```

2. export PATH

```
# export PATH=$PATH:$HOME/opt/bin
```

3. compile the code

```
# upcc <-T=16> file.upc -o file
```

4. run the code

```
# mpiexec <-n=16> file
```

Listing 9. Running UPC programs

One point to remember is that if the code uses THREADS or MYTHREAD constants to declare arrays or for deciding the sharing block sizes, then the number of threads must be passed at compile time and when running the program. `upcc` uses `-T` switch for this purpose.

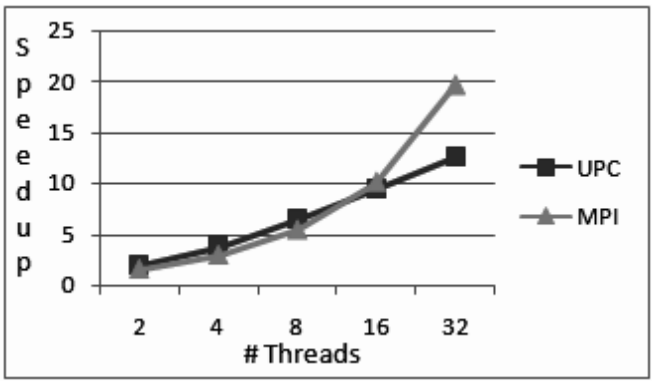
Also, the default thread spawner, `upirun`, uses `mpirun`, which is not used on OSC clusters. Hence, all codes must to be run by using `mpiexec`.

IV. Performance Comparison

The performance data was collected for four different problems by comparing the runtime of serial execution with that of execution of UPC codes and MPI codes. The UPC code along with the performance data are presented in sub-sections *A-D*. The codes were run with 16 and 32 threads, except for matrix multiplication, which was run with 64 threads due to its extremely computation intense nature. All source code is present in the tar file submitted with the project.

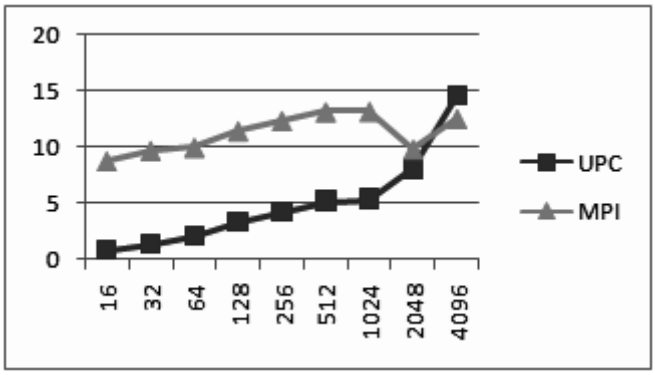
A. PI Estimation

The same code, as presented in Listing 2, was used. The results are summarized in the graph below.



Graph 1. MPI vs UPC for PI Estimation

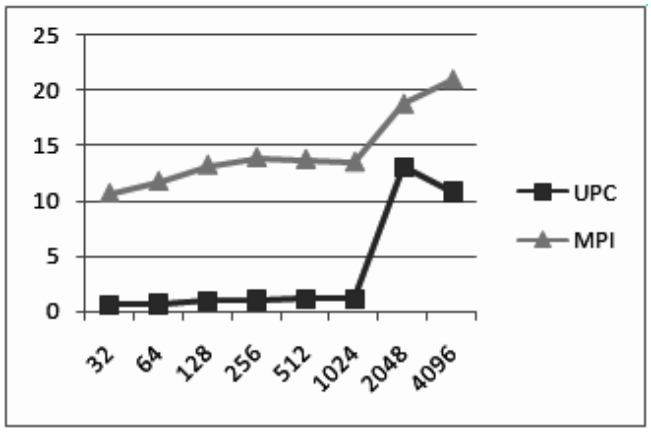
This program requires very little communication amongst the processes. As seen above, UPC scales linearly with the number of threads, but MPI shows an exponential increase. The code in Listing 1, however, does not scale well and gives poorer results due to usage of a shared array.



Graph 2. Matrix Vector with 16 Threads

B. Matrix-Vector Multiplication

The matrix a is row-block divided amongst the threads while arrays b and c are block distributed. Each thread is responsible for evaluating the value of elements of the c array assigned to it. The program proceeds by pre-fetching the b array since its needed by all for all computations. Then, the iterations are work-distributed amongst the threads, based on affinity with elements of array c. A possibility is to cast shared data pointers to local pointers if it resides on the node itself. It turns out that code with casting runs almost twice as fast the code without casting. Graphs 2 and 3 show a comparison of MPI code and UPC code (with pointer casting).

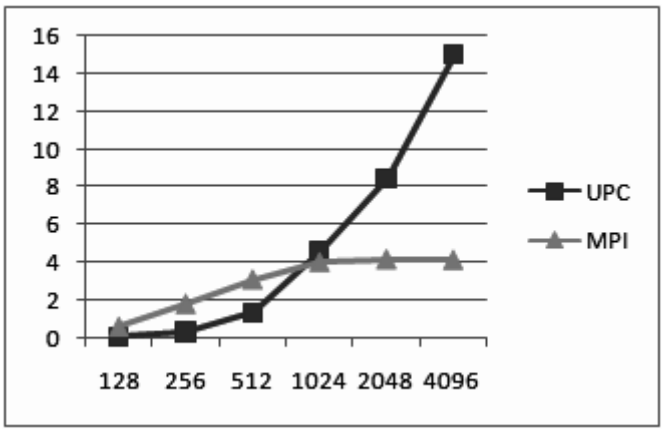


Graph 3. Matrix-Vector with 32 Threads

This code requires an initial data movement followed by computations with local data. Hence the computations play a more important role than communication, an MPI scales nicely, leveling out at large

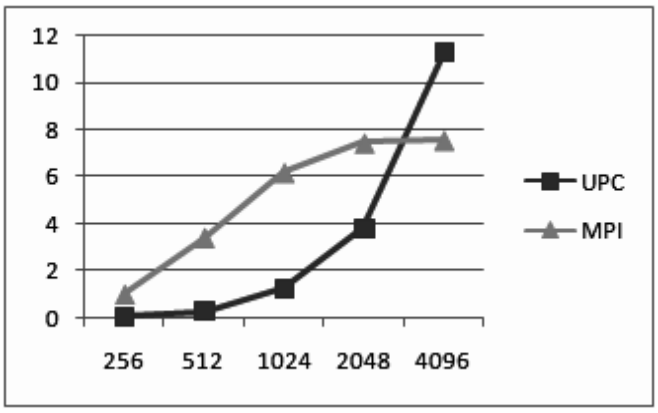
data size, probably due a naïve implementation of the computations. UPC on the other hand shows remarkable improvements at large data sizes but with lesser threads.

C. *Two Matrix-Vectors with dependency*



Graph 4. M-V M-V with 16 Threads

Here, we have two matrix-vectors in which the result of one is used for calculations in the other. Hence, we have to compute array c first and then copy it to all nodes again perform the second matrix-vector. Graphs 4 and 5 give the performance comparison with MPI.

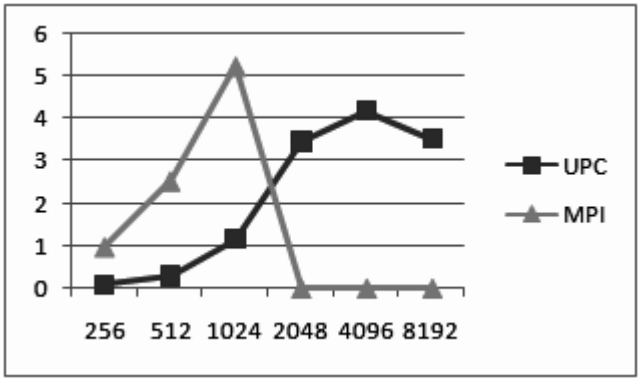


Graph 5. M-V M-V with 32 Threads

This code is fairly communication intensive since each outer iteration requires two communications. Here again, MPI levels out at large data sizes while UPC simply zooms up. Using libraries (like BLAS) for computation may help smoothen out the differences.

D. *Matrix-Matrix Multiplication*

This is an extremely computation as well as communication intense task. The implementation used is a naïve row-block distributed implementation. The MPI code suffers and does not run for matrices larger than 2048x2048, probably due to memory allocation limitations. Graph 6 summarizes the results.



Graph 6. Matrix-Matrix on 64 Threads

The peak MFLOPS achieved for all the problems by UPC is presented in Table 5.

Problem	Problem Size	MFLOPS
M-V	4096	5240
M-V M-V	4096	5543
M-M	4096	1168

Table 5. UPC Peak MFLOPS

V. Conculsion

The performance comparison clearly shows that even though UPC is easier to code and provides a more intuitive programming environment, for the very same code, MPI give a much better performance.

With MPI, a lot of responsibility rests with the programmer. But, if done correctly, if the most naïve of implementations can give a good scale-up. Hence, its not a wonder that MPI has become a de facto standard for parallel programming.

Also, UPC's limitation on maximum shared block size (1048576) and lack of explicit construct for 2-D

distribution of data structures limits its capabilities. Also, it needs large data sets to perform well on a large distributed systems. MPI on the other hand, scales up as well as down gracefully.

VI. Acknowledgments

I would like to take this opportunity to thank Prof. P. Sadayappan for his guidance and for motivating to study parallel programming in greater detail. His courses (CSE 621 and CSE 721) have been extremely interesting and intellectually stimulating. I also wish to thank Jim Dinan for helping me out with installing Berkeley UPC on Ohio Supercomputer Center's clusters.

VII. References

[1] George Washington University UPC [Online]. Available: *http://www.gwu.edu/~upc/documentation.html*

[2] MPI Subroutine Reference [Online]. Available: http://www.nersc.gov/vendor_docs/ibm/pe/am107mst02.html

[3] Berkeley UPC installation guide [Online]. Available: *http://upc.lbl.gov/download/dist/INSTALL*

[4] Berkeley UPC Homepage [Online]. Available: http://upc.lbl.gov/

[5] George Washington University High Performance Computing Laboratory UPC Manual v1.2 [Online]. Available: *http://upc.gwu.edu/downloads/Manual-1.2.pdf*

[6] Kathy Yelick, *Unified Parallel C, Spring 2006 Tutorial for CS 267* [Online]. Available:

[7] CSE 621 and CSE 721 Lecture Notes and Slides