Final Project Report

Adam Karl

April 19, 2021

1 Introduction

Unified Parallel C (UPC) is an extension of C that is intended for high performance computing, especially on large-scale parallel machines with a common global address space. UPC was developed at Berkeley in 1999 as an attempt to combine the best features of 3 other parallel processing extensions for C: AC, Split-C, and Parallel C Preprocessor (PCP). UPC and its compiler have been constantly updated over the years, with the most recent update to the specification in November 2013 and the most recent compiler update on 16 April 2021.

2 Syntax

When running .upc files, the entire file will be run by every thread. It is not possible to only run part of the program in parallel.

Some of the most important things to know from the UPC library are:

- THREADS
  - the total number of threads
- MYTHREAD
  - the thread index, from 0 to THREADS-1
- upc_forall()
  - divide a for loop among threads
  - discussed further later
- upc_barrier;
  - make threads wait until they've all reached the barrier
- upc_localsizeof()
  - determine the amount of bytes the shared variable has with affinity for the current thread

UPC introduces a new C type-qualifier "shared" to indicate variables that are available across processes. With exceptions, unless specifically requested the actual physical location of the variable will be in thread 0. This is referred to as the variable having an *affinity* for thread 0. Any variable that is not shared is private to its own thread.

Large shared data structures such as shared arrays can distribute their elements over all processors. I will discuss the application of these further in the "Comparison" section, but these more nuanced affinities include:

- cyclic (per element)
  - successive elements of the array have affinity with successive threads
  - shared int x [1000]
    - 1 element each, wrapped

- blocked-cyclic (user-defined)
  - the array is divided into user-defined size blocks and the blocks are cyclically distributed among threads
  - shared [10] int x [1000]
    - 10 elements each, wrapped
    - items 0-9 in thread 0, 10-19 in thread 1, ...
- blocked (run-time)
  - each thread has affinity to one contiguous part of the array. The size of the contiguous part is determined in such a way that the array is "evenly" distributed among threads
  - shared [1000/THREADS] int x [1000]
    - thread 0 has items 0 through 1000/THREADS-1
    - thread 1 has the next 1000/THREADS items

In the header of the .upc file, you should include either <upc_strict.h> or <upc_relaxed.h>. The relaxed version places the responsibility on the programmer to avoid race conditions, while the strict version runs slower but guarantees the same outcome as a sequential program.

In addition to these items specific to UPC, several other helpful libraries are included with UPC, such as: collective operations (eg broadcast, scatter, gather, etc.), Atomic Memory Operations, Blocking and Non-Blocking Data Copy, Parallel File System I/O, and High-Performance Wall-Clock Timers. UPC also contains support for (and can be used in conjunction with) both pthreads and OpenMP.

3 Comparison

After the arduous process of installing UPC and figuring out how to compile and run .upc programs, I set about converting my matrix multiplication program to use UPC. I simply changed all input and output matrices to "shared" as such:

```
//Input Array A
shared int inputArrayA [NROW][NCOL];
//Input Array B
shared int inputArrayB [NROW][NCOL];
//Weights
shared int Weight [NROW][NCOL];
//Output Array C
shared int outputArrayC [NROW][NCOL];
```

To parallelize the matrix multiplication, I changed the outer for loop to use the upc_forall function. For each element in the for loop, the thread the work is designated to is determined by a 4th argument.

```
upc_forall(i=0; i<NROW; i++; & outputArrayC[i][0]) {
    for(j=0; j<NROW; j++) {
        outputArrayC[i][j] = inputArrayB[i][j];
        for(k=0; k<NCOL; k++) {
            outputArrayC[i][j] += inputArrayA[i][k]*Weight[k][j];
        }
    }
}
```

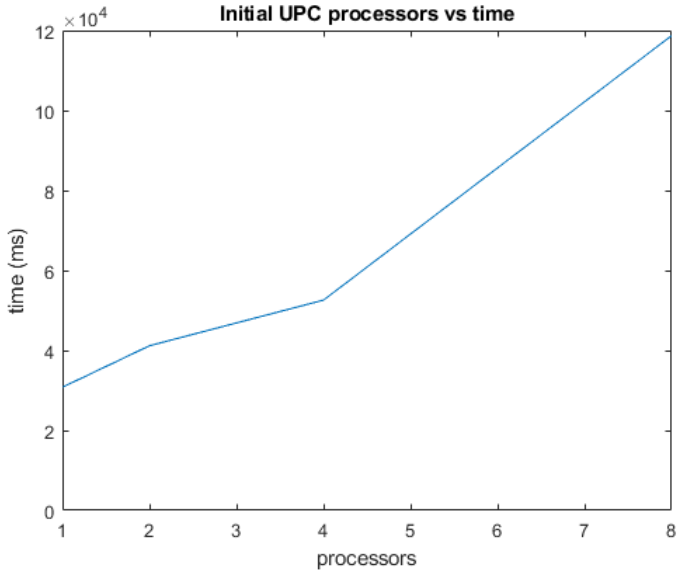
By making this $outputArrayC[i][0]$, I am saying that whichever thread "owns" that element of the output array (whichever thread $outputArrayC[i][0]$ has affinity for) will do all the calculations necessary to fill out that row. As a simple example,

```
upc_forall(i=0; i<NROW; i++; i) {
    ...
}
```

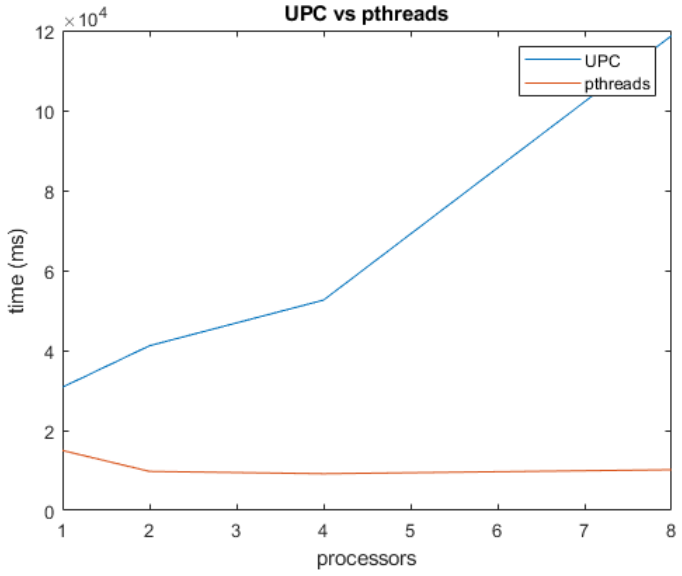
is exactly the same as

```
for(i=0; i<NROW; i++) {
    if (MYTHREAD == i % THREADS) {
        ...
    }
}
```

However, when running this file with 1-8 threads, here are the runtimes I got:



Comparing this graph to my pthreads implementation (using the exact same algorithm):



Not only is the runtime higher than pthreads for every number of processors, the runtime actually *increases* as more threads are added. To understand why, I had to learn more about affinities and how shared memory is actually managed in UPC.

A key aspect of UPC is that each shared variable only exists at one place. By default, any basic typed shared variable (int, double, etc.) will actually be located in processor 0. This is defined as the variable having an **affinity** for processor 0. Instead of having a value, each other processor will only have a pointer to the location in processor 0 where the variable is.

When instead creating an array or matrix, all the elements are spread evenly among all the processors in a cyclic manner. The 1st element will be at processor 0, the 2nd element at processor 1, the 3rd element at processor 3, and so on.

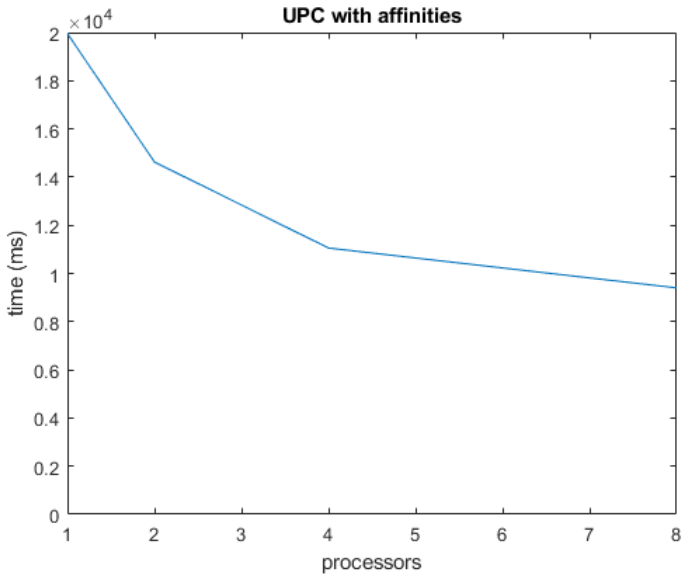
With this in mind, I analyzed what my first implementation was actually doing. While the upc_forall loop ensured that the output location in outputArrayC always had an affinity for the processor doing the calculation, the critical loop of matrix multiplication required looking up elements from inputArrayA and Weight. Since these were evenly spread across all processors, the element would only have an affinity for the processor 1/THREADS of the time. For 2 processors, the value would be located in a different processor 1/2 of the time. For 4 processors it would be elsewhere 3/4 of the time, and for 8 processors it would be elsewhere 7/8 of the time. Since the process of fetching the value from a different processor naturally takes additional time, constantly having to fetch values from other processors outweighed the benefits of multiprocessing.

This problem is familiar to anyone with experience optimizing cache hits. To improve the program, I had to find a way to organize the data to minimize the amount of "misses." After some trial and error, I came up with this solution in the matrix declarations:

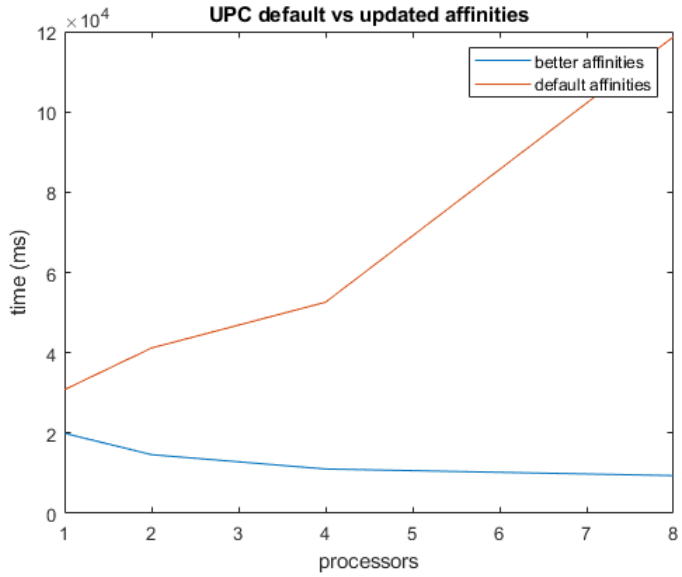
```
//Input Array A
shared [NCOL] int inputArrayA [NROW][NCOL];
//Input Array B
shared [NCOL] int inputArrayB [NROW][NCOL];
//Weights
shared int Weight [NROW][NCOL];
//Output Array C
shared [NCOL] int outputArrayC [NROW][NCOL];
```

The [NCOL] before three of the matrices indicates the **block size** for distributing the matrices. Here the first set of NCOL elements (in other words, 1 row) of the matrix will have an affinity for processor 0, the next row will have an affinity for processor 1, and so on. Note that the "hit chance" for the Weights matrix cannot be improved in this way due to the nature of matrix multiplication. Now, with the *exact same* matrix multiplication algorithm, the values for inputArrayA, inputArrayB, and outputArrayC will **always** have an affinity for the processor dealing with them.

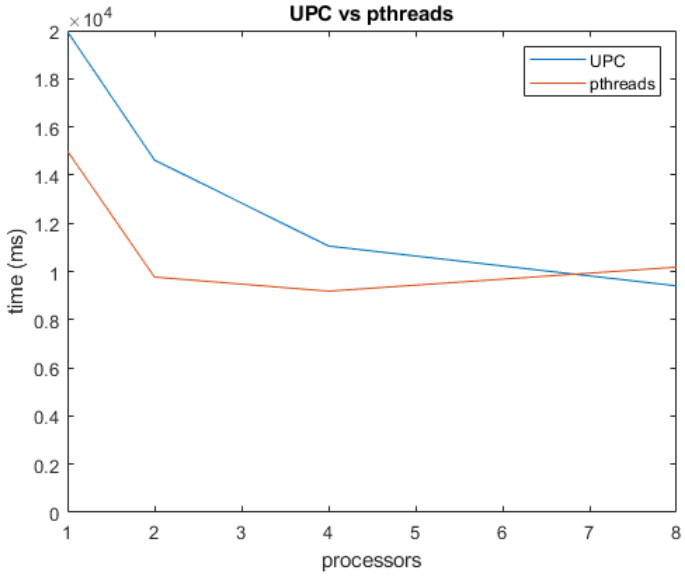
With this being the only change, I ran the tests again with 1-8 processors:



Compared to the un-optimized allocation, you can see the improvement here:

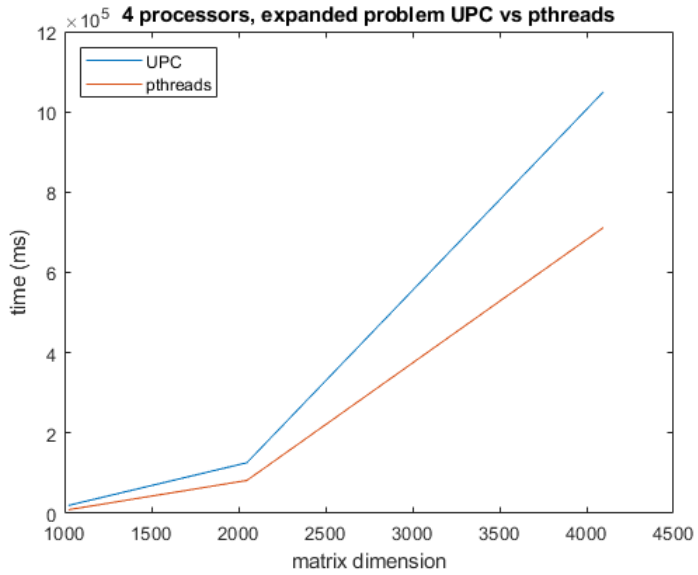


Plotted against the pthreads parallelization, UPC is now able to compete:

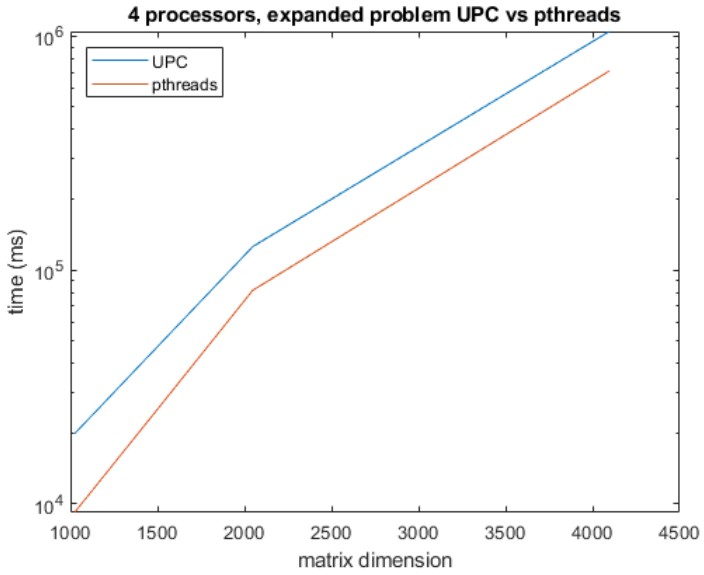


I'm sure that there's a way to further improve the blocking/affinities to minimize inter-processor communication even more, but I messed around with some ideas for a few hours and this was the best I could do.

Next I tried expanding the problem to larger matrices while setting the processors to 4. Theoretically, UPC saves memory by distributing it, and should be able to out-scale pthreads in the very long term. **For this experiment, I also included the matrix initialization in the interval time.** The initialization is another place UPC will have the advantage over pthreads since each element only has to be defined once, rather than copying the entire matrices to every threads' memory space. With the maximum tested matrix dimension being 4096, here's how UPC scaled versus pthreads:



I actually find this graph to be pretty misleading. For the dimensions I tested, doubling the dimension for the UPC program results in 6-7x the run-time. Doubling the dimension for the pthreads program results in 8-9x the run-time. In other words, despite it seeming as though the UPC program scales poorly, **the UPC program would eventually outscale the pthreads program.** It becomes more obvious that UPC is actually catching up when plotting the y-axis on a log scale:



Unfortunately, I am not able to test at higher dimensions on my (admittedly pretty wimpy) linux box, since pthreads outright rejects the matrix size as exceeding thread limitations. I *am* able to continue a bit higher with UPC since it more efficiently distributes memory, but I'm getting to the point where the programs take a significant time to finish.

However, matrix multiplication is certainly not an ideal algorithm for benchmarking UPC. Due to how each element in the product is calculated, each processor will have to fetch the majority (75% for 4 processors) of required values from Weights matrix, regardless of how I decide to block it.

Therefore, I decided to see if I could design a program where UPC outperforms pthreads. To play to the strengths of UPC's distributed memory, I decided on multiplying an array by a scalar. For this program, outside of the setup one thread will **never** have to communicate with another processor to read/write a variable. In addition, the distributed memory means that each processor will have exactly the data it needs; no more, no less.

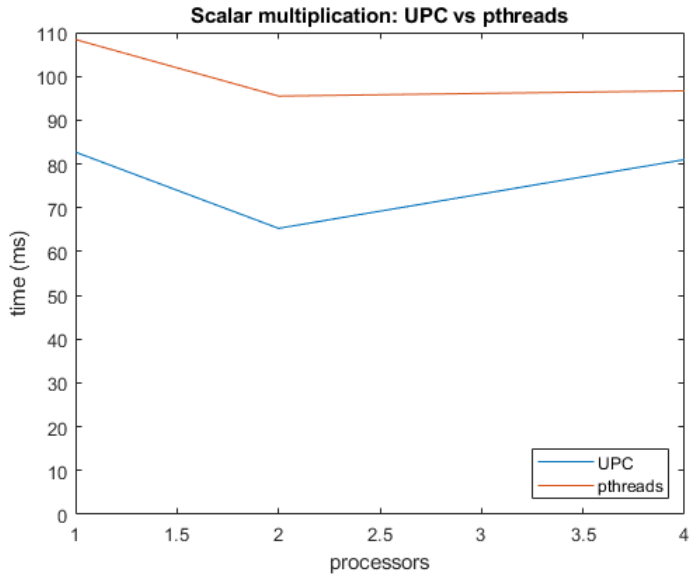
The relevant parts of my program look like this: (note, the default blocking used here distributes the array 1 element at a time)

```
#define ELEMENTS  1024*1024*8
#define WEIGHT    1024
//Input Array A
shared int inputArrayA [ELEMENTS];
//Output Array C
shared int outputArrayC [ELEMENTS];

...

// CALCULATE MATRIX MULTIPLICATION
upc_forall(i=0; i<ELEMENTS; i++; i) {
    outputArrayC[i] = inputArrayA[i] * WEIGHT;
}
```

When testing this implementation against a corresponding pthreads program, I was finally able to surpass pthreads' performance consistently:



There still seems to be some weird stuff going on: I ran the programs many times and averaged the results but UPC still runs faster for 2 processors than 4. I'm unsure of precisely why, it's possibly due to my linux box having pretty outdated hardware. Regardless, I'm still happy with this as a proof-of-concept that programs with larger data and lower processor interaction will favor UPC over pthreads.

4 Discussion

The first major roadblock for this project was the actual installation of UPC and the Berkeley Runtime package. There are no walkthroughs anywhere online I could find, so the only guide is a 19-page INSTALLATION.txt file that's incredibly dense. Several hours later, I managed to boil the installation down to:

- download the tarball from https://upc.lbl.gov/download/
- unzip the tarball and navigate inside
- $ ./configure –disable-NETWORK_API
- $ gmake
  - takes several minutes

  - known to fail, just try running gmake again until it succeeds
- test UPC is functional by running:
  - $ gmake tests-hello
  - $ env UPCC_FLAGS= ./upcc – norc –version
  - if these run without error, proceed
- $ sudo gmake install
- add bin to PATH
  - default: /usr/local/berkeley_upc/bin

Now .upc programs can be compiled and run with:

- $ upcc –network=smp -O -T 4 -o hello_world hello_world.upc
  - here, 4 is the number of threads
- $ upcrun hello_world

Compiling and running the matrix multiplication program looks something like this:

```
adam@ak-ubuntu:~/Desktop/1645_project$ upcc --network=smp -O -T 4 -o m
mult mmult_par.upc
adam@ak-ubuntu:~/Desktop/1645_project$ upcrun mmult
UPCR: UPC thread 0 of 4 on ak-ubuntu (pshm node 0 of 1, process 0 of 4
, pid=122194)
UPCR: UPC thread 2 of 4 on ak-ubuntu (pshm node 0 of 1, process 2 of 4
, pid=122213)
UPCR: UPC thread 1 of 4 on ak-ubuntu (pshm node 0 of 1, process 1 of 4
, pid=122212)
UPCR: UPC thread 3 of 4 on ak-ubuntu (pshm node 0 of 1, process 3 of 4
, pid=122214)

Total Sum = 2.93373e+12
Interval length: 9927.62 msec.
```

Figuring out to add the –network=smp option to the compilation alone set me back over an hour; there really aren't enough simple resources for using UPC and the Berkeley compiler. By default, UPC attempts to run across machines connected by a network, so it's necessary to add this flag to restrict it to one machine.

Determining if UPC is "better" than pthreads or OpenMP is a complicated question. For one, UPC includes implementations of both pthreads and OpenMP as well as a host of other supplementary libraries, so anything you can do with pthreads of OpenMP can also be run with UPC.

The main downsides for UPC are (1) performance and (2) lack of online sources. It took all of my ingenuity to get UPC to compete In addition, the fact that there are so few UPC tutorials on the internet has to be considered a significant downside. Installation was a nightmare, and the first tutorial I was able to find actually included an *incorrect* way to block arrays. This was made significantly worse because in y experience the upcc compiler doesn't produce very meaningful error messages, so it was difficult to pinpoint where the errors were.

However, once up and running the versatility of UPC is a huge upside. It can be compiled using the gcc compiler after some configuration. There are the option for either strict or relaxed multiprocessing depending on how likely race conditions are. You can either determine the number of threads at compile-time or at run-time. UPC threads can save memory by spreading a single array over all the threads rather than have each thread with its own copy of the whole array. And of course, by default UPC attempts to work over a network, so I imagine running large-scale programs over many machines would be incredibly simple with UPC.

I wouldn't mind seeing UPC added as part of the class. UPC's basic tools (including: MYTHREAD, THREADS, "shared" variables, upc_forall(), and upc_barrier) are the easiest tools I've worked with in this class.

Honestly, the biggest downside to teaching UPC over pthreads or OpenMP is that UPC is more obscure, and also seemingly designed for network computing (which makes many features irrelevant for running on single machines).

5 Conclusions

I'll be honest, my initial experience with UPC was frustrating to say the least. It was by far the most difficult setup I've had to do for this class. However, once I had my workflow down, using UPC quickly became my personal favorite multiprocessing implementation. In my opinion, UPC has the most straightforward way to divide work among threads with upc_forall(), and the "shared" tag along with an understanding of affinities made it extremely easy to visualize the memory mapping.

Now that the project's over, I view UPC as almost the "python" of multithreading (at least for small data): extremely quick and easy to implement at the cost of some performance setbacks. It is possible to increase the performance with knowledge of memory mapping and caching, but UPC will rarely be the most efficient implementation for most smaller programs. However, UPC is extremely versatile and for programs that work with big data and/or highly parallelizable algorithms (for instance, parallel quicksort rather than matrix multiplication) UPC can have the significant upside of out-scaling competing implementations.