



# Knapsack and UPC

---

By

Rajesh Nishtala

March 22, 2006



# What would make a good report

---

- Write up a better UPC implementation than what we have given
  - Tune for your favorite platform with the native conduit (GM/LAPI/VAPI)
  - But run the same code on the other platforms
  - Give an outline of what improvements you have made to the code
    - If possible include what kind of performance improvement it got
- Write a small paragraph at the end that has the pros/cons of the UPC language as a whole.
  - Feel free to gripe away, user feedback is invaluable to the development team and I'll make sure to pass it on.



# How I write UPC programs

---

- Write a serial version in your favorite language to understand the nuances of the problem
  - I prefer Matlab, but C is just as good
- Make sure that any serial algorithms you have work in all the corner cases
  - As you have seen parallel code is 10x harder to write and debug
- Correctness First!
  - Then performance

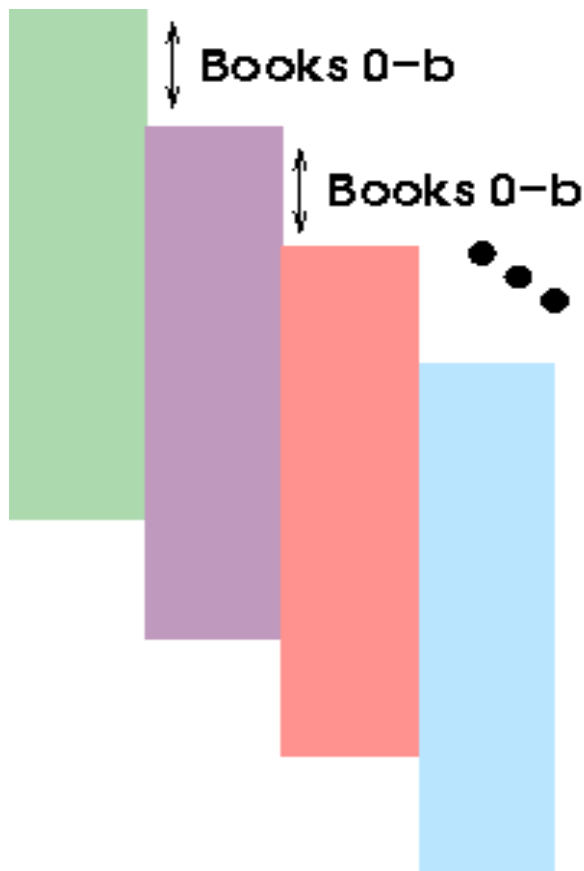


# Knapsack

---

- You are basically given a  $N \times C$  matrix that you need to fill in with the code algorithm that we have given you
- Two ways of parallelizing
  - Split books across processors
    - Incurs a lot of communication at the boundaries
    - Creates a dependence in time
  - Split capacity across processors
    - Only a dependence in time
    - Lets use this one.

# Pipelining



- Since there is a dependence on the capacity processor  $p+1$  has to wait for processor  $p$  to finish computing it's portion of capacity vector for book  $b$  before it can start.
- Necessitates a way to send signals across processors to tell them when they can start



# Sending Signals: UPC Style

---

- Method 1 (slow)
  - Processor  $p$  can keep a counter of what book its on and subsequent processors constantly ask processor  $p$  what iteration of the loop they have finished.



# Code for Computing Processor

---

Code for processor p (assume processor 0 for simplicity):

```
/*counters allocated in shared space*/
shared int counters[THREADS];
int *count; /*declare local copy*/
/*create a local pointer to my
  counter*/
count = (int*) (counters+MYTHREAD);
for(*count=0; *count<N; *count++) {
  do stuff;
}
```



# Code for Waiting Processor

---

```
/*counters already declared*/  
int *count; /*declare local copy*/  
/*create a local pointer to my counter*/  
count = (int*) (counters+MYTHREAD);  
/*need to wait for my neighbor to get to  
  b before I can start*/  
while(counters[MYTHREAD-1]<b);  
for(*count=0; *count<N; *count++) {  
    do stuff;  
}
```



# Combined Code

---

```
shared int counters[THREADS];
int *count;
count = (int*) (counters+MYTHREAD);
if (MYTHREAD > 0)
    while (counters[MYTHREAD-1] < b);
for (*count = 0; *count<N; *count++) {
    do stuff;
}
```



# Way Way too much communication

---

- The spin loop:
  - `while(counters[MYTHREAD-1] < b)`
  - Incurs communication for every iteration!!
  - Might be ok on SMP but horrible for Distributed Memory
    - Why?
- Better way (Method 2)
  - Local counters but global flags array
  - More code
  - Has a `upc_barrier` and a `upc_fence` ...



# Code for Computing Processor

---

```
int count;
shared int flags[THREADS];
flags[MYTHREAD]=0;
/* make sure everyone has reset the flag*/
upc_barrier;
for(count=0; count<N; count++) {
    do stuff;
    if(count == b) {
        flags[MYTHREAD+1] = 1;
        upc_fence;
        /*makes sure that all the shared memory references
        until this point are finished ... ensures us that
        processor MYTHREAD+1 got the message */
    }
}
```



# Code For Waiting Processor

---

(code until the UPC barrier in previous slide is the same)

```
int *myflag; /*local pointer to myflag*/
myflag = (int*) &flags[MYTHREAD];
while(*myflag == 0); /*spin wait on local
    variable only*/
For(count = 0; count<N; count++) {
    do stuff;
}
```



# Why this method is better

---

- spin wait is only on local flag
  - Only one communication event to signal
  - Communicate a very small piece of data (one int)
  - Doesn't slow the other side down to force it to spin on communication



# Full Code

---

```
int count;
int *myflag;
shared int flags[THREADS];
flags[MYTHREAD] = 0;
upc_barrier;
myflag = (int*) &flags[MYTHREAD];
if(MYTHREAD > 0)
    while(*myflag == 0);
for(count=0; count<N; count++){
    do stuff;
    if(count == b && MYTHREAD != THREADS-1){
        flags[MYTHREAD+1] = 1;
        upc_fence;
    }
}
```



# Overlapping Communication and Computation

---

- Berkeley UPC has non-blocking communication extensions
  - proven to lead to large performance improvements
  - Have to be careful with data management
  - Make sure buffers are around until remote acknowledgement that the data has reached safely.
    - Usually done through a barrier synchronization
  - See Documentation for full details:
    - [http://upc.lbl.gov/publications/upc\\_memcpy.pdf](http://upc.lbl.gov/publications/upc_memcpy.pdf)



# Simple Example to Motivate Ideas

---

- Each processor computes  $N$  vectors or length  $R$  that are destined for different processors.
  - vector  $i$  sent to processor  
 $i \bmod \text{THREADS}$  into row  
 $\text{MYTHREAD} * N/\text{THREADS} + i \% (N/\text{THREADS})$
  - Every processor thus receives  $N$  different vectors from different processors



# Global Data Allocation

---

```
shared int *destination_data; /*allocation*/
shared [] int** dest_ptrs; /*local copy*/
/* perform the allocation */
destination_data =
    upc_all_alloc(THREADS, sizeof(int)*N*R);
/*allocate space for the local copy*/
dest_ptrs = (shared [] int**)
    malloc(sizeof(shared [] int*) * THREADS);
/* exchange pointers*/
for(t=0; t<THREADS; t++) {
    dest_ptrs[t] = (shared [] int*)
        &(destination_data[t]);
}
upc_barrier;
```



# Blocking Version (less memory)

---

```
/* allocate local data .. Only need R ints*/
local_data = malloc(sizeof(int)*R);
for(i=0; i<N; i++) {
    /*safe to overwrite local_data because
    blocking put semantics*/
    compute vector i into local_data;
    /*remote position computation*/
    rp = MYTHREAD*N/THREADS+i%(N/THREADS)
    /* send the data over */
    upc_mempout(dest_ptrs[i%THREADS]+rp*R,
        local_data, sizeof(int)*R);
}
upc_barrier;
```



# Non-Blocking Version (more memory)

---

```
/* allocate data ... need N*R ints*/
local_data= malloc(sizeof(int)*N*R);

/* allocate nonblocking handles
   to know when the communication events
   finish
*/
bupc_handle_t *nbhandles;
nbhandles = (bupc_handle_t*)
    malloc(sizeof(bupc_handle_t)*N);
```



# Non-Blocking Version (continued)

---

```
for(i=0; i<N; i++) {
    /* can't overwrite because nonblocking semantics say that
    until put is cleared the data has to be available */
    compute vector i into local_data+i*R
    rp = MYTHREAD*N/THREADS+i%(N/THREADS);
    /* same call ... but handle is returned*/
    nbhandles[i] =
    bupc_memput_async(dest_ptrs[i%THREADS]+rp*R,
    local_data+i*R, sizeof(int)*R);
}
/* sync all handles indicating the put has finished */
for(i=0; i<N; i++) {
    bupc_waitsync(nbhandles[i]);
}
/* make sure everyone has finished their puts */
upc_barrier;
```



# Pros / Cons Of Nonblocking

---

- Cons

- semantics cause the possibility of more bugs
- have handles floating around
- more memory used

- Pros

- almost always faster
- hide communication costs
- allow for fine-grained communication / computation overlap



# Other Useful features

---

- Simple collective library developed last year for 267 students
  - <http://upc.lbl.gov/docs/user/README-collectivev.txt>
- Portable High-Precision Timers
  - <http://upc.lbl.gov/docs/user/index.html#timer>
- Debugging and Trace Functionality
  - <http://upc.lbl.gov/docs/user/index.html#debugging>



# Conclusions

---

- Sending Processors Signals
  - Will be useful in setting up the pipeline
- Non blocking versions ...
  - Overlapping communication and computation
- Read UPC Documentation
  - A lot of UPC documentation
- Email Me
  - Since this is a new language .. Feel free to email me code if you have specific questions