

CS267: Fun with UPC

Marghoob Mohiyuddin
marghoob@eecs.berkeley.edu

Outline

- The Knapsack Problem
- The UPC Solution
- Things to be done
- and Reported

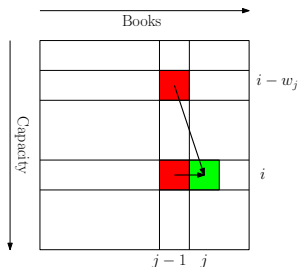
The Knapsack Problem

- N old books $\{B_1, B_2, \dots, B_N\}$.
- Book B_i has weight w_i and is worth p_i dollars.
- Want to choose books such that their total weight $\leq C$ (capacity) and their total worth is maximized.
 - By the way, the weights w_i and C are integers.

The Dynamic Programming Solution

- Dynamic programming: Store solutions of subproblems in table and use them for solving larger problems
- Use a $C \times N$ table T :
 - $T[i, j]$ is the optimal worth for the problem when the maximum weight is limited to i and the books have to be chosen from $\{B_1, \dots, B_j\}$
- Observation:
 - $T[i, j] = \max(T[i, j - 1], p_j + T[i - w_j, j - 1])$, i.e.,
 - To pick or not to pick book B_j , that is the question
- The Dynamic Programming Solution:
 - Fill in the table, $T[C, N]$ is the optimal value
 - Backtrack to get the list of books for the optimal value

Parallelizing the Dynamic Programming Solution

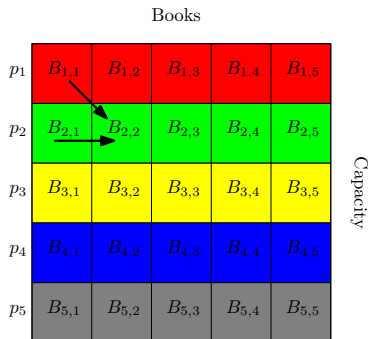


We simply need to fill in the $C \times N$ table in parallel. Possible ways of splitting:

- Split books across processors, i.e., split by columns
 - Lots of communication at the boundaries
 - Creates a dependence in time
- Split capacity across processors, i.e., split by rows
 - Lesser communication, lesser time dependence
 - This is done by the given UPC code.

Pipelining

Since there is a dependence on
Assume the following layout:



The arrows show the
dependencies.

We now have the following
sequence of computations:

- 1 Fill in block $B_{1,1}$.
- 2 Fill in blocks $B_{1,2}, B_{2,1}$.
- 3 Fill in blocks $B_{1,3}, B_{2,2}, B_{3,1}$.
- 4 Fill in blocks $B_{1,4}, B_{2,3}, B_{3,2}, B_{4,1}$.
- 5 And so on ...

We also need a way to tell
other processors to start.

Sending signals

- Method 1 (slow)
 - Processor p keeps a counter of what book it is on. Subsequent processors ask p what book it is on.

Code for Method 1

- Here is the code for processor p :

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

- Here is the code for waiting processor

```
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 book 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;
}
```

What is wrong with Method 1?

- 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` ...

Method 2: 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 */
    }
}
```

Method 2: 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 Method 2 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 for Method 2

```
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

Simple Example to Motivate Ideas

Each processor computes N vectors of length r that are destined for different processors.

- Vector i sent to processor $i \% \text{THREADS}$ 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_mempup(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);

for(i=0; i<N; i++) {
    /* cant 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_mempup_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 Non-Blocking

- 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 Stuff

- Simple collective library developed 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>

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/Titanium 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.

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: Feel free to email me code if you have specific questions

Questions