Announcements

- Thanks to everyone who found the UD-scan bugs in the slides and book!
- Approval for MS cluster happened today!
  - Accounts assigned this week
  - Documents up this week
  - Should be developing off-line anyhow (more later)
- Next week is ZPL, but if you’re expecting to use it, read text early

There is still time for project revisions
Tonight’s Assignment

- UD-Scan of your choice ... what did you choose?

Multicast Logic

- Recall the question of multicasting in Peril-L in a way that is suitable for SUMMA
- This is the set up: tile rows want column segments and tile columns want row seg.s
Memory Allocation

- The problem memory is in global space
  
  ```c
  double A_G[n][n], B_G[n][n], C_G[n][n];
  int p=sqrt(P), t=n/p;  // Define constants
  ```

- Allocate materialized memory for row/col for seg.s
  
  ```c
  double Acol_GO[p][p][t], Brow_GO[p][p][t];
  ```

- Next, induce a tree on the tile rows, columns

![Flow down tree](image)

Assume p is a power of 2

Logical Tree for (a Row) of Tiles

- A node waits on its _GO memory; when it gets it’s value, it simply fills out its sibling nodes if any

```c
if (v == 0)
  stride=p;
else
  stride = pow(2,loOrdZeroes(v));  // No. least sig zeroes
a[0:t-1] = Acol_GO[u][v][0:t-1];  // Grab col segment
while (stride > 1) {
  stride = stride>>1;  // Reference next one
  Acol_GO[u][v+stride][0:t-1]=a[0:t-1];  // Fill GO mem
}
```

Make it into a procedure: `mcast()`
Make diagonal tiles responsible

```c
for (k=0; k<n; k++) {
    for (i=0; i<p; i++) {
        Acol_GO[i][0][0:t-1] = A_G[i*t:i*t+t-1][k];
        Brow_GO[0][i][0:t-1] = B_G[k][i*t:i*t+t-1];
    }
}
```

Do multicast, if needed

```c
a[0:t-1] = mcast(u, v, Acol);
b[0:t-1] = mcast(u, v, Brow);
```

---

### SUMMA Preamble Code

```c
double A_G[n][n], B_G[n][n], C_G[n][n];
int p = sqrt(P), t = n/p;  // Define constants
forall u, v in (0..p-1, 0..p-1) {  // Thread in 2D
    double C[t][t] = is_local(C_G)  // Ref local tile
    int i, j, k; double a[t], b[t];
    for (i=0; i<t; i++) {
        for (j=0; j<t; j++) {
            C[i][j] = 0.0;  // Initialize C
        }
    }
}
```
Inner Loop of SUMMA

for (k=0;k<n;k++) {
    /* Multicasting code goes here */
    for (i=0;i<t;i++) {
        for (j=0;j<t;j++) {
            c[i][j] += a[i]*b[j];  \textit{Figure k^{th} terms}
        }
    }
}

Multicast Improvement

- The diagonal tile processors fill all of the _GO memory, so processors wait on them
- Is there a better solution? Issues …
  - A specific tile (diagonal) initializes result
  - All row (col) roots initialized together
Revision ...

- The first processor in row/col ready, call it a “winner,” becomes tree root and tree shifts
  - To shift tree, simply use the $u(v)$ value of winner to offset all node positions, doing all addressing math mod $p$
  - To ensure a uniquely identified winner, use an exclusive block (per row/col) & global; if your $k$ is higher than the value stored there, you’re the winner; update it and fire-up the tree

How much can the computation skew?

Parallel Programming Support

- For decades it’s been assumed: parallel code = compiler(sequential code)
- Compilers have progressed, but cannot do the conversion in general circumstances
- Two basic solutions
  - Library + serial programming language
    - Main issue: || abstractions limited
  - Legitimate parallel language
    - Main issue: quality of emitted code
Programming Language Topic

- There is much to say about parallel languages
- Book’s Strategy
  - 3 representative approaches (will be 2 more)
  - Give key features, basics; enough to write code but other materials needed for full study
- Course Strategy
  - Discuss several approaches including book’s
  - Focus on abstract ideas
  - Summarize and discuss current state of the art

Avoid teaching detail

OpenMP

- Nonproprietary extensions (as pragmas) to C, C++, Fortran
- Mainly used to exploit hyper-threading parallelism in a single fetch/execute engine
- Use
  - Programmer inserts pragmas identifying parallelism
  - Compiler recognizing pragmas generates multi-threaded code
  - System in control of most aspects of parallelism

http://www.openmp.org
OpenMP Code Examples

- All pragmas begin: `#pragma`
- Convert 32-bit RGB image to 8-bit gray scale
  
  ```
  #pragma omp parallel for
  for (i=0; i < numPixels; i++) {
      pGrayScaleBitmap[i] = (unsigned BYTE)
          (pRGBBitmap[i].red * 0.299 +
           pRGBBitmap[i].green * 0.587 +
           pRGBBitmap[i].blue * 0.114);
  }
  
  ||ism is “element-wise” … each item independent
  Also called “work sharing”
  ```

Limitations and Semantics

- Not all “element-wise” loops can be ||ised
  ```
  #pragma omp parallel for
  for (i=0; i < numPixels; i++) {} 
  
  n Loop index: signed integer
  n Termination Test: <,<=,>,=> with loop invariant int
  n Incr/Decr by loop invariant int; change each iteration
  n Count up for <,<=; count down for >,>=
  n Basic block body: no control in/out except at top
  ```
- Threads are created and iterations divvied up; requirements ensure iteration count is predictable
More OpenMP Code

- Data-dependences require care [wrong code]
  ```c
  sum = 0;
  #pragma omp parallel for
  for (i=0; i < 100; i++) {
    sum += array[i];
  }
  ```
- A race exists in this code (sum); fix 2 ways:
  1. Make sum private by declaring inside loop
  2. #pragma omp parallel for(private sum)

Reduce Abstraction

- OpenMP has reduce
  ```c
  sum = 0;
  #pragma omp parallel for reduction(+:sum)
  for (i=0; i < 100; i++) {
    sum += array[i];
  }
  ```
- Reduce ops and init() values:
  + 0 bitwise & ~0 logical & 1
  - 0 bitwise | 0 logical | 0
  * 1 bitwise ^ 0
  Even in OpenMP abstracting reduce helps
Sections

- Separate tasks can be performed in ||
  ```c
  #pragma omp sections {
      #pragma omp section (TaskA());
      #pragma omp section (TaskB());
      #pragma omp section (TaskC());
  }
  ```
- The tasks must not have dependences
  - Each section runs to completion
  - Order not guaranteed
  - Private is allowed

Care with Parallel

- Check out this code
  ```c
  int i;
  #pragma omp parallel for
  for (i='a'; i<='z'; i++){printf("%c",i);}  
  int i;
  #pragma omp parallel private (i)
  for (i='a'; i<='z'; i++){printf("%c",i);}  
  ```
- Red prints alphabet once; blue unknown #
- The compiler decides on concurrency
Treads are created/destroyed

- Threads are created at start of parallel block; destroyed, with implied barrier, at end of || block
- Good advice: Set up threads at start; stay with ‘em
- Avoid waiting overhead by using nowait
  
  ```
  #pragma omp for nowait
  for (i=0; i<100; i++) {arrayA[i]=i; }
  #pragma omp for
  for (j=0; j<500; j++) {arrayB[j]=0; }
  ```

- The explicit barrier has the form

  ```
  #pragma omp barrier
  ```

Dependences

- Handling dependences is entirely up to the programmer

- Tools for protecting code:
  - Privatizing variables -- requires “cleanup code”
  - Reduce
  - Atomic operations
  - Critical sections
Synchronizing

- Any statement’s execution can in principle be interrupted, so atomicity help would help
- Achieve atomicity using: `atomic`
  
  ```
  #pragma omp atomic
  a[i] += x; // never interrupted
  ```
- Atomic operations are:
  
  ```
  expr++, expr--, ++expr, --expr, +=, -=,
  *=, /=, <<=, >>=, &=, |=, ^=
  ```
- Could save cost of using “heavy” protection for some variables

Critical Sections

- Guarantee exclusive execution with a critical section
  
  ```
  #pragma omp critical(maxvalue) 
  if (max < next_value)
    max = next_value;
  ```
- Only 1 thread enters critical section at a time
- Naming avoids all threads but 1 excluded from all critical sections, usually a big win
Loop Scheduling

- When loop iterations are not balanced ...
  
  #pragma omp parallel for schedule(kind [,chunk_size])

- The choices for kind are
  
  * static assign chunk size units of work; default is loop_bound/threads; 1 implies interleaving iterations
  * dynamic work queue with chunk size iterations per thread; default is 1
  * guided work queue with diminishing chunks down to chunk size
  * runtime choose 1 of above at run-time w/environ var

OpenMP Summary

- Simple facility, low entry cost, potential to exploit parallelism with little programming effort

- Simplicity is somewhat deceptive:
  
  * Programmers are responsible for all potential “gotchas” … still need to think very carefully!
  * Few higher-level abstractions beyond reduce
  * Programming model is threaded von Neumann rather than true parallel
  * De facto control over features that give performance are generally ceded to compiler

There is more in the spec
Break

Threading

- Threading facilities like the POSIX library Pthreads are popular shared-memory parallel programming tools
- Unlike OpenMP, where a compiler takes over to give limited capabilities, threading systems give primitive ops, but little help
  
  anything’s possible, nothing is easy
- PRAM-like model tied to shared memory
- Pthreads is a library that’s widely available
Pthread Standard Structure

- Create threads and wait for their completion

```c
#include <pthread.h>
int err;
void main ()
{
    pthread_t tid[MAX]; /* Thread ID Array */
    for (i=0; i<t; i++) {
        err = pthread_create (&tid[i], NULL,
                              count3s_thread, i);
    }
    for (i=0; i<t; i++) {
        err = pthread_join_(tid[i], &status[i])
    }
}
```

Mutual Exclusion

- Race conditions are avoided using locks

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
void count3s_thread (int id) {
    /* Compute local part of array */
    int length_per_thread = length/t;
    int start = id * length_per_thread;
    for (i=start; i<start+length_per_thread; i++) {
        if (array[i] == 3) {
            pthread_mutex_lock(&lock);
            count++;
            pthread_mutex_unlock(&lock);
        }
    }
}
```
Thread-specific Data

- Mostly Pthreads reference global data
- Pthreads allows for thread-specific data
  - Accessed indirectly via a key
  - Procedural interface makes it somewhat kludgy
- Facilities
  - `pthread_key_create()`
  - `pthread_key_delete()`
  - `pthread_setspecific()`
  - `pthread_getspecific()`

Not suitable for non-trivial data structures

Condition Variables

- Threads wait on a condition to come true, and then some waiting thread is chosen
  - Non-deterministic and possibly unfair
- Illustrate with a circular buffer

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t nonempty = PTHREAD_COND_INITIALIZER;
pthread_cond_t nonfull = PTHREAD_COND_INITIALIZER;
Item buffer[SIZE];
int in = 0; // Buff index for next insert
int out = 0; // Buff index for next remove
```
Get and Put For Circular Buffer

```c
void put (Item x) {            // Producer thread
    pthread_mutex_lock(&lock);
    while (in - out == SIZE) // While buffer full
        pthread_cond_wait(&nonfull, &lock);
    buffer[in % SIZE] = x; in++;
    pthread_cond_signal(&nonempty);
    pthread_mutex_unlock(&lock);
}

Item get() { // Consumer thread
    Item x;
    pthread_mutex_lock(&lock);
    while (out == in)  // While buffer is empty
        pthread_cond_wait(&nonempty, &lock);
    x = buffer[out % SIZE]; out++;
    pthread_cond_signal(&nonfull);
    pthread_mutex_unlock(&lock);
    return x;
}
```

Split-phase Barrier

- Barriers stop threads until everyone’s there
- Often wasteful bc barriers are interposed between regions that shouldn’t overlap

```plaintext
earliest place for barrier
... computational work
... latest place for barrier
```

- Sending, receiving also have this feature

Break the coupling between arriving and leaving
Arrive and Wait

- Split the barrier into two parts
  - `barrier.arrived()` -- has reached the first safe pt
  - `barrier.wait()` -- the last point before overlap

Thread 1  Thread 2  Thread 3  Thread 4

```
arrived

x = a + ...

wait

```

`a = f(b, c, d);`

---

Split phase has many uses

---

Summary on Pthreads

- Pthreads is gives power and flexibility
- It is possible to build gadgets for general concurrent operations
  - Nearly everything is possible
  - Often the complexity can be deceptive
  - Ideas are quite traditional, and newer concepts are available
    - Transactions
    - Shared memory parallel languages like CiLC, SAC
    - Not easy to apply performance (CTA) ideas

---

There is more in the spec
Message Passing

- Message passing is the principle alternative to shared memory parallel programming
  - Based on Single Program, Multiple Data (SPMD) Model with `send()` and `recv()` primitives
  - Message passing is universal, but low-level
  - Parallel Virtual Machine (PVM), Message Passing Interface (MPI) are main libraries, but there've been many
  - More even than threading, message passing is locally focused -- what does each processor do?
  - Isolation of separate address spaces can be a programming asset -- no races--and a pain!

  **Clear distinction between local, non-local**

A Typical Process Structure

- SPMD idea => 1 pgm to run on every node

  ```c
  int main (argc, argv)
  {
    int myID, size;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &myID);
    /* compute stuff in parallel */
    MPI_Finalize()
    return 0;
  }
  ```

  **MPI_Comm_World** is a communicator a set of logically related comm's

  **required 1st call**

  **get count of peers**

  **required last call**

  **get my index**
General Process Structure

- Most computations have two kinds of procs
  - Worker -- performing a share of work
  - Leader -- performing 1-time work needed by all and, perhaps, its share of task
- Common structure:

```
if (rootproc == myId) {
  ... /* do stuff for all */
} else {
  ... /* work on local part */
}
```

Sending A Message

- The general form of an MPI send() is:

```c
int MPI_Send ( // Blocking Send routine
  void *       buffer, // Address of data to send
  int          count,  // No. data elements to send
  MPI_Datatype type,     // Type of data elements
  int          dest,     // ID of destination process
  int          tag,      // Tag for this message
  MPI_Comm *   comm      // An MPI communicator
)
```

```
MPI_Send(&a[offset][0], count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
```
Receiving A Message

- The general form of an MPI recv() is:

```c
int MPI_Recv ( // Blocking Receive routine
    void *       buffer, // Address receiving data
    int          count,  // No. elements to receive
    MPI_Datatype type,   // Type of each element
    int          source, // ID of sending process
    int          tag,    // Tag for this message
    MPI_Comm     comm,   // MPI communicator
    MPI_Status * status  // Status of this receive
);
```

- `MPI_Recv(&a, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);`

Marshalling

- MPI assumes data comes from consecutive locations and goes to consecutive locations
- When not true (columns in rmo-allocation)
  - data must be marshalled, copied into buffer, for send
  - data must be demarshalled, copied back, for receive
main(int argc, char **argv) {
    int numtasks, /* number of tasks in partition */
    taskid, /* a task identifier */
    numworkers, /* number of worker tasks */
    source, /* task id of message source */
    dest, /* task id of message destination */
    nbytes, /* number of bytes in message */
    mtype, /* message type */
    intsize, /* size of an integer in bytes */
    dbsize, /* size of a double float in bytes */
    rows, /* rows of matrix A sent to each worker */
    averow, extra, offset, /* used to determine rows sent to each worker */
    i, j, k, /* misc */
    count;
    MPI_Status status;
}

A “master−slave” solution

MM in MPI -- 1

double a[NRA][NCA], /* matrix A to be multiplied */
       b[NCA][NCB], /* matrix B to be multiplied */
       c[NRA][NCB]; /* result matrix C */
intsize = sizeof(int);
dbsize = sizeof(double);
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
numworkers = numtasks−1−1;
/****** master task *******/
if (taskid == MASTER) {
    for (i=0; i<NRA; i++)
        for (j=0; j<NCA; j++)
            a[i][j]= i+j;
    for (i=0; i<NCB; i++)
        b[i][j]= i*j;
MM in MPI -- 3

/* send matrix data to the worker tasks */
averow = NRA/numworkers;
extra = NRA%numworkers;
offset = 0;
mtype = FROM_MASTER;
for (dest=1; dest<=numworkers; dest++) {
    rows = (dest <= extra) ? averow+1 : averow;
    MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    count = rows*NCA;
    MPI_Send(&a[offset][0], count, MPI_DOUBLE, dest, mtype,
             MPI_COMM_WORLD);
    count = NCA*NCB;
    MPI_Send(&b, count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
    offset = offset + rows;
}

46

MM in MPI -- 4

/* wait for results from all worker tasks */
mtype = FROM_WORKER;
for (i=1; i<=numworkers; i++) {
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    count = rows*NCB;
    MPI_Recv(&c[offset][0], count, MPI_DOUBLE, source, mtype,
              MPI_COMM_WORLD,&status);
}

/**************************************************************************** worker task */
if (taskid > MASTER) {
    mtype = FROM_MASTER;
    source = MASTER;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    count = rows*NCA;
}
MM in MPI -- 5

MPI_Recv(&a, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
MPI_Recv(&b, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);

for (k=0; k<NCB; k++)
  for (i=0; i<rows; i++) {
    c[i][k] = 0.0;
    for (j=0; j<NCA; j++)
      c[i][k] = c[i][k] + a[i][j] * b[j][k];
  }

mtype = FROM_WORKER;
MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
} /* end of worker */

91 “Net” Lines

The Path of a Message

- A blocking send visits 4 address spaces
- Besides being time-consuming, it locks processors together quite tightly
Alternative Send/Recvs

- Variants of the operations have other properties:
  - MPI_Rsend() -- assumes sending, receiving processes are synchronized, so no handshaking needed; it's risky
  - MPI_Bsend() -- use a user-space buffer rather than kernel space buffer; resume when buffer loaded
  - MPI_Isend() -- non-blocking send; does not wait for operation to complete; use MPI_Wait()

Overlapping Comm and Comp

- Using MPI_Isend()/MPI_Irecv() to overlap communication and computation is smart
- General protocol:
  - Receive “edge” values from neighbors
  - Send “edge” values to neighbors
  - Compute “interior” elements
  - Wait on arrival of edge elements
  - Complete “edge” computations

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>13</th>
<th>21</th>
<th>7</th>
<th>29</th>
<th>23</th>
<th>20</th>
<th>4</th>
<th>25</th>
<th>1</th>
<th>31</th>
</tr>
</thead>
</table>

Shadow buffers assist implementation
MPI Has Reduce and Scan

- Reduce and scan apply are ponderous

```c
int MPI_Reduce ( // Reduce routine
    void *       sendBuffer,  // Address of local val
    void *       recvBuffer,  // Place to receive into
    int          count,       // No. of elements
    MPI_Datatype datatype,    // Type of each element
    MPI_OP       op,          // MPI operator
    int          root,        // Process to get result
    MPI_Comm     comm         // MPI communicator
);
```

```c
MPI_Reduce (&myCount,&globalCount, 1, MPI_INT, MPI_SUM, RootProcess, MPI_COMM_WORLD);
```

Message Passing Critique

- Message passing is a very simple model
- Extremely low level; heavy weight
  - Expense comes from λ and lots of local code
  - Communication code is often more than half
  - Tough to make adaptable and flexible
  - Tough to get right and know it
  - Tough to make perform in most cases
- Programming model of choice for scalability
  - Not as portable as it’s claimed to be
One-sided Communication

- Intermediate between and shared and message passing is one-sided communication.
- Process model with global address space:
  - `get()` loads a value from a non-local address
  - `put()` stores a value into a non-local address
  - No memory consistency: *caveat emptor*
  - Popularized by Cray machines: called `shmem`
- Libraries are available to implementing `Co-Array Fortran based on concept`

Working On Project

- Parallel computers of any size are generally tough to use ... put it off as long as possible
- Most programming systems allow for development on a workstation
- My recommended steps (not including your personal development techniques):
  - Sketch solution w/diagram + Peril-L
  - Work out logic on sequential platform
  - Consider moving to || platform in parts
Homework

- Reading: Chapter 8
  - Read, but do not study …
  - Goal is to conceptualize ZPL’s approach

- Project
  - Most projects approved; after reading ZPL, proceed
  - Recommendation: Sketch in Peril-L first
  - Bring paper statement of progress to class