Track 1: OpenMP & MPI

Goal: To learn the basics of OpenMP and to get started with MPI

Announcements

☐ OpenMP experiments due Monday
☐ Practice MPI program due Wednesday
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 ||ism
  - Compiler recognizing pragmas generates multi-threaded code
  - System in control of most aspects of ||ism

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
  ```c
  #pragma omp parallel for
  for (i=0; i < numPixels; i++) {}
  ```
  - Loop index: signed integer
  - Termination Test: <,<=,>,=> with loop invariant int
  - Incr/Decr by loop invariant int; change each iteration
  - Count up for <,<=; count down for >,=>
  - Basic block body: no control in/out except at top

- Threads are created and iterations divvied up; requirements ensure iteration count is predictable

What Does `parallel` Mean?

- “When a parallel region is encountered, a logical team of threads is formed. Each thread in the team executes all statements within a parallel region except for work-sharing constructs [such as `for`]. Work within work-sharing constructs is distributed among the threads in a team.”

- “Loop iterations must be independent before the loop can be parallelized. An implied barrier exists at the end of a parallelized statement block.”
More OpenMP Code

- Data-dependences require care [wrong code]
  ```
  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:
  - Make sum private by declaring inside loop
  ```
  #pragma omp parallel for(private sum)
  ```

Reduce Abstraction

- OpenMP has reduce
  ```
  sum = 0;
  #pragma omp parallel for reduction(+:sum)
  for (i=0; i < 100; i++) {
    sum += array[i];
  }
  ```

- Reduce ops and init() values:
  
<table>
<thead>
<tr>
<th>Operator</th>
<th>Bitwise</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
<td>~0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td>^0</td>
</tr>
</tbody>
</table>
  
  Even in OpenMP abstracting reduce helps
Sections

- Separate tasks can be performed in parallel with:
  ```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

```c
#pragma omp for nowait
for (i=0; i<100; i++) { arrayA[i]=i; }
```

- The explicit barrier has the form

```c
#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(optional name)
  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
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 argc;
char **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;
}
```

required 1st call

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

get count of peers

get my index

required last call
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:

```c
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 *marshaled*, that is, copied into a buffer, for send
  - data must be *demarshaled*, copied back, after receive
MM in MPI -- 1

MPI_Status status;
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;

A "master–slave" solution

MM in MPI -- 2

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);
umworkers = numtasks-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++)
        for (j=0; j<NCA; j++)
            b[i][j] = i;
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;
}

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*NCA;
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

```c
MPI_Recv(&a, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
count = NCA*NCB;
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 */
```

The Path of a Message

- A blocking send visits 4 address spaces
- Besides being time-consuming, it locks processors together quite tightly
Other Routines

- Besides set-up, Send(), Recv(), MPI has many other useful facilities:
  - MPI_Scatter() … distributed data from root process to others
  - MPI_Gather() … accumulate data in root process from others
  - MPI_Bcast() … broadcast a value to other processes; # and types of values must match in all participants
  - MPI_Barrier() …

---

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

Shadow buffers assist implementation

---

MPI Has Reduce and Scan

- Reduce and scan are ponderous; for $P$ vals

```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);
```
MPI Groups

- A MPI group is a subset of the processes
- Groups allow higher-level structure to be overlaid on the processes, like rows/cols

```c
int MPI_Group_incl(
    MPI_Group group,    // Existing group
    int size,     // Size of the new group
    int *ranks, // ranks of includes processes
    MPI_Group *newGroup // New group to create
);
```

- Alternatively … overlay own logical structure

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