Lecture 7

Programming with message passing
MPI
Announcements

• Presenters for papers – 2 next week
• More projects posted
• Starter code for Canon’s algorithm in $PUB/Examples/MPI/Cannon_Starter
• Trestles
Today’s lecture

• A look at Matrix Multiply results
• Programming with message passing
Results – shared memory

• Correction to Lecture 5 (slide 19)
• N=512, single and double precision
• Different thread geometries
• Baseline: 23 GFlops on 4 cores of Lilliput
  69 Gflops on 8 cores of Triton (double)

<table>
<thead>
<tr>
<th>Geometry</th>
<th>16 × 16</th>
<th>8 × 8</th>
<th>4 × 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncoalesced</td>
<td>9.2</td>
<td>8.9</td>
<td>8.2</td>
</tr>
<tr>
<td>Coalesced</td>
<td>125 (57)</td>
<td>53 (41)</td>
<td>12 (15)</td>
</tr>
</tbody>
</table>
Results with Matrix multiply: Fermi & Tesla

Matthew Fedder

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Max Gflop</th>
<th>Equiv. Cores</th>
<th>Theor Gflop (FMA)</th>
<th>% Theor</th>
<th>GB Moved</th>
<th>Time (s)</th>
<th>Eff BW</th>
<th>Theor BW</th>
<th>% Theor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tesla</td>
<td>SP</td>
<td>14.29</td>
<td>2.56</td>
<td>622</td>
<td>2.3%</td>
<td>10</td>
<td>1.88</td>
<td>5.3</td>
<td>105</td>
<td>5.1%</td>
</tr>
<tr>
<td></td>
<td>DP</td>
<td>11.97</td>
<td>2.15</td>
<td>77.76</td>
<td>15.4%</td>
<td>20</td>
<td>2.24</td>
<td>8.9</td>
<td>105</td>
<td>8.5%</td>
</tr>
<tr>
<td>Fermi</td>
<td>SP</td>
<td>94.26</td>
<td>16.91</td>
<td>1405</td>
<td>6.7%</td>
<td>10</td>
<td>0.28</td>
<td>35.1</td>
<td>152</td>
<td>23.1%</td>
</tr>
<tr>
<td></td>
<td>DP</td>
<td>81.69</td>
<td>14.65</td>
<td>702</td>
<td>11.6%</td>
<td>20</td>
<td>0.33</td>
<td>60.9</td>
<td>152</td>
<td>40.0%</td>
</tr>
<tr>
<td>Lillput</td>
<td>DP</td>
<td>22.3</td>
<td>4.00</td>
<td>32</td>
<td>69.7%</td>
<td>8</td>
<td>0.096 /iter</td>
<td>83.2</td>
<td>19.2</td>
<td>433%</td>
</tr>
</tbody>
</table>
Results with Matrix multiply on Fermi

Mike Folkerts
Results with C1060

Mike Folkerts

© 2011 Scott B. Baden / CSE 262 / Spring 2011
Programming with Message Passing

• **The** primary model for implementing parallel applications
• Programs execute as a set of P processes
  ‣ We specify P when we run the program
  ‣ Assume each process is assigned a different physical processor
• Each process
  ‣ is initialized with the same code, but has private state
    SPMD = “Same Program Multiple Data”
  ‣ executes instructions at its own rate
  ‣ has an associated *rank*, a unique integer in the range 0:P-1
  ‣ may or may not be assigned a different physical processor
• The sequence of instructions each process executes depends on its rank and the messages it sends and receives
• Program execution is often called “bulk synchronous” or “loosely synchronous”
Message Passing

• Messages are like email; to send one, we specify
  ‣ A destination
  ‣ A message body (can be empty)

• To receive a message we need similar information, including a receptacle to hold the incoming data

• Requires a sender and an explicit recipient that must be aware of one another

• Message passing performs two events
  ‣ Memory to memory block copy
  ‣ Synchronization signal on receiving end: “Data arrived”
A minimal interface

• Query functions
  \[ \text{nproc}() = \# \text{processors} \]
  \[ \text{myRank}() = \text{this process's rank} \]

• Point-to-point communication
  ‣ Simplest form of communication
  ‣ Send a message to another process
    \[ \text{Send}() \]
    \[ \text{Send(Object, Destination process ID)} \]
  ‣ Receive a message from another process
    \[ \text{Receive(Object)} \]
    \[ \text{Receive(Source process, Object)} \]
Send and Recv

- When `Send()` returns, the message is “in transit”
  - A return doesn’t tell us if the message has been received
  - Somewhere in the system
  - Safe to overwrite the buffer
- `Receive()` blocks until the message has been received
  - Safe to use the data in the buffer
Message completion

• A `Send()` may or may not complete…
• … before a `Recv()` has been posted
• “May or may not” depends on the implementation
• Some programs may deadlock on certain message passing implementations

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (x,1)</td>
<td>Send(y,0)</td>
</tr>
<tr>
<td>Recv (y,1)</td>
<td>Recv(x,0)</td>
</tr>
</tbody>
</table>

This program may hang

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (x,1)</td>
<td>Recv(x,0)</td>
</tr>
<tr>
<td>Recv (y,1)</td>
<td>Send(y,0)</td>
</tr>
</tbody>
</table>

This program is “safe”
Buffering

- Where does the data go when you send it?
- It might be buffered
- Preferable to avoid the extra copy
Causality

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent.
- If different processes send messages to the same destination, the order of receipt isn’t defined across sources.
Asynchronous, non-blocking communication

- Immediate return, does not wait for completion
  - Required to express certain algorithms
  - Optimize performance: message flow problems

- Split-phased
  - Phase 1: initiate communication with the immediate ‘I’ variant of the point-to-point call
    \[ \text{IRrecv( ), ISend( )} \]
  - Phase 2: synchronize
    \[ \text{Wait( )} \]
  - Perform unrelated computations between the two phases

- Building a blocking call
  \[ \text{Recv( )} = \text{IRrecv( ) + Wait( )} \]
Restrictions on non-blocking communication

• The message buffer may not be accessed between an `IRecv()` (or `ISend()`) and its accompanying `Wait()`

`ISend(data,destination)`
`Wait()` on `ISend()`
Use the data

• Each pending `IRecv()` must have a distinct buffer
### Overlap

<table>
<thead>
<tr>
<th>Overlap</th>
<th>No Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRecv(x, req)</td>
<td>IRecv(x)</td>
</tr>
<tr>
<td>Send(...)</td>
<td>Send(…)</td>
</tr>
<tr>
<td>Compute(y)</td>
<td>Wait(x)</td>
</tr>
<tr>
<td>Wait(req)</td>
<td>Compute(x)</td>
</tr>
<tr>
<td>Compute(x)</td>
<td>Compute(y)</td>
</tr>
</tbody>
</table>

A message buffer may not be accessed between an `IRrecv()` (or `ISend()`) and its accompanying `wait()`.
• We’ll program with a library called **MPI**
  “Message Passing Interface”
  ‣ 125 routines in MPI-1
  ‣ 7 minimal routines needed by every MPI program
    • start, end, and query MPI execution state (4)
    • non-blocking point-to-point message passing (3)
  • Reference material: see [http://www-cse.ucsd.edu/users/baden/Doc/mpi.html](http://www-cse.ucsd.edu/users/baden/Doc/mpi.html)
  • Callable from C, C++, Fortran, etc.
  • All major vendors support MPI, but implementations differ in quality
Functionality we’ll cover

• Point-to-point communication
• Collective communication
• Non-blocking communication
• Message Filtering
• Communicators
A first MPI program: “hello world”

#include "mpi.h"

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello, world! I am process %d of %d.\n", rank, size);
    MPI_Finalize();
    return(0);
}
Send and Recv

```c
const int Tag=99;
int msg[2] = { rank, rank * rank};
if (rank == 0) {
    MPI_Status status;
    MPI_Recv(msg, 2,
             MPI_INT, 1,
             Tag, MPI_COMM_WORLD, &status);
}
else  MPI_Send(msg, 2,
               MPI_INT, 0,
               Tag, MPI_COMM_WORLD);
```
Message status

- An MPI_Status variable: a struct that contains the sending processor and the message tag
- This information is useful when we haven’t filtered the messages
- We may also access the length of the received message (may be shorter than the message buffer)

```c
MPI_Recv( message, count, TYPE, MPI_ANY_SOURCE, MPI_ANY_TAG, COMMUNICATOR, &status);
MPI_Get_count(&status, TYPE, &recv_count);
```

```c
status.MPI_SOURCE    status.MPI_TAG
```
Asynchronous, non-blocking communication

- An extra request argument is required
  
  ```c
  MPI_Request request;
  MPI_Irecv(buf, count, type, source, tag, comm,&request)
  ```

- We use the request variable to specify which message we are synchronizing in `MPI_Wait()`
  ```c
  MPI_Wait(&request, &status)
  ```

- Making above 3 calls in succession is equivalent to
  ```c
  MPI_Recv(buf, count, type, source, tag, comm, &status)
  ```
Buffering

- If there is not a pending receive, then an incoming message is placed in an anonymous system buffer
- When the receive gets posted, the message is moved into the user specified buffer
- Double copying reduces communication performance
- Non-blocking communication can help avoid this problem
- *MPI: The Complete Reference*, by Marc Snir et al.  
  “Buffering and Safety”
- *Send modes* are also useful  
  www-unix.mcs.anl.gov/mpi/sendmode.html