Lecture 12

Performance Measurement

Advanced Communication
Announcements

• No class on 11/17 and 11/19
• Makeup lectures
  – Friday 11/20. 3:00 to 4:20
  – Weds 12/2 5-7PM
• CSE 260 Symposium
  – Week 10: Tues, Weds, Thurs
Parallel print function
Parallel print function

- Debugging output can be hard to sort out on the screen
- Many messages say the same thing
  Process 0 is alive!
  Process 1 is alive!
  ...
  Process 15 is alive!

- Compare with
  Processes[0–15] are alive!

- Parallel print facility
  http://www.llnl.gov/CASC/ppf
Summary of capabilities

- Compact format list sets of nodes with common output
  
```c
PPF_Print( MPI_COMM_WORLD, "Hello world" );
0–3: Hello world
```

- `%N` specifier generates process ID information
  
```c
PPF_Print( MPI_COMM_WORLD, "Message from %N\n" );
Message from 0–3
```

- Lists of nodes
  
```c
PPF_Print(MPI_COMM_WORLD,
    (myrank % 2)
    ? "[%N] Hello from the odd numbered nodes!\n"
    : "[%N] Hello from the even numbered nodes!\n")
[0,2] Hello from the even numbered nodes!
[1,3] Hello from the odd numbered nodes!
```
Practical matters

• Installed in $(PUB)/lib/PPF
• Use a special version of the arch file called arch.ppf.pgi
• Each module that uses the facility must
  
  #include "ptools_ppf.h"

• Look in $(PUB)/Examples/PPF for example programs ppfexample_cpp.C and test_print.c
Send_Recv

• Instead of Send and Recv, we can use MPI_Sendrecv_replace() to simplify the coding and improve performance

• Sends then receives a message using a single buffer

```c
int MPI_Sendrecv_replace
( void *buf,
  int count,  MPI_Datatype datatype,
  int dest,   int sendtag,
  int source, int recvtag,
  MPI_Comm comm, MPI_Status *status )
```
Parallel Matrix Multiplication
Parallel matrix multiplication

• Assume $p$ is a perfect square
• Each processor gets an $n/\sqrt{p} \times n/\sqrt{p}$ chunk of data
• Organize processors into rows and columns
• Process rank is an ordered pair of integers
• Assume that we have an efficient serial matrix multiply

\[
\begin{array}{ccc}
p(0,0) & p(0,1) & p(0,2) \\
p(1,0) & p(1,1) & p(1,2) \\
p(2,0) & p(2,1) & p(2,2) \\
\end{array}
\]
Loop reordering

• The simplest formulation of matrix multiply is the so called “ijk” formulation, named after the order of the loops

\[
\text{for } i := 0 \text{ to } n-1, \ j := 0 \text{ to } n-1, \ k := 0 \text{ to } n-1 \\
C[i,j] += A[i,k] \times B[k,j]
\]

• Now consider the “kij” formulation

\[
\text{for } k := 0 \text{ to } n-1, \ i := 0 \text{ to } n-1, \ j := 0 \text{ to } n-1 \\
C[i,j] += A[i,k] \times B[k,j]
\]
Formulation

• The matrices may be non-square

\[
\begin{align*}
\text{for } k & := 0 \text{ to } n_3 - 1 \\
\quad \text{for } i & := 0 \text{ to } n_1 - 1 \\
\quad \quad \text{for } j & := 0 \text{ to } n_2 - 1 \\
C[i,j] & += A[i,k] \ast B[k,j] \\
C[i,:] & += A[i,k] \ast B[k,:] \\
\end{align*}
\]

• The two innermost loop nests compute \( n_3 \) outer products

\[
\begin{align*}
\text{for } k & := 0 \text{ to } n_3 - 1 \\
C[:,:] & += A[:,k] \ast B[k,:] \\
\end{align*}
\]

where \( \ast \) is outer product
Outer product

• Recall that when we multiply an \( m \times n \) matrix by an \( n \times p \) matrix… we get an \( m \times p \) matrix

• Outer product of column vector \( a^T \) and vector \( b = \text{matrix } C \)
  an \( m \times 1 \) times a \( 1 \times n \)
  \[a[1,3] \cdot x[3,1]\]

  \[
  (a,b,c) \ast (x,y,z)^T = \begin{pmatrix}
  ax & ay & az \\
  bx & by & bz \\
  cx & cy & cz
\end{pmatrix}
  \]

  Multiplication table with rows formed by \( a[:] \) and the columns by \( b[:] \)

• The SUMMA algorithm computes \( n \) partial outer products:
  \[
  \text{for } k := 0 \text{ to } n-1 \\
  C[:,:] += A[:,k] \cdot B[k,:]
  \]
Serial algorithm

• Each row \( k \) of \( B \) contributes to the \( n \) partial outer products:

\[
\text{for } k := 0 \text{ to } n-1 \\
C[:,,:] += A[:,k] \cdot B[k,:] 
\]
Parallel algorithm

- Set up a processor geometry $P = px \times py$
- Blocked multiply, panel size $= b << N/\max(px,py)$
  
  \[
  \text{for } k := 0 \text{ to } n-1 \text{ by } b
  \]
  
  multicast $A[:,k:k+b-1]$ along processor rows
  multicast $B[k:k+b-1,:]$ along processor columns
  
  $C += A[:,k:k+b-1] * B[k:k+b-1,:]$ // Local MM

- Each row and column of processors independently participate in a broadcast of a panel
- Owner of the panel changes with $k$
What is the performance?

for k := 0 to n−1 by b

// Tree broadcast:  \(1g \sqrt{p} (\alpha + b\beta n/\sqrt{p})\)
// For long messages:  \(2((\sqrt{p} -1)/ p)b\beta n\)

multicast \(A[:,k:k+b-1]\) along rows
multicast \(B[k:k+b-1,:]\) along columns

// Built in matrix multiply:  \(2(n/\sqrt{p})^2b\)

\(C += A[:,k:k+b-1] * B[k:k+b-1,:]\)

• Total running time:  \(~2n^3/p + 4\beta bn/ \sqrt{p}\)
Highlights of SUMMA

• **Performance**
  – Running time = \(2n^3/p + 4\beta bn/\sqrt{p}\)
  – Efficiency = \(O(1/(1 + \sqrt{p/n^2})\)

• **Generality**: non-square matrices, non-square geometries

• Adjust b to **tradeoff latency cost** against **memory**
  – b small \(\Rightarrow\) less memory, lower efficiency
  – b large \(\Rightarrow\) more memory, high efficiency

• **Low temporary storage**
  – grows like \(2bn/\sqrt{p}\)

• A variant used in SCALAPACK

  R. van de Geign and J. Watts,
  “SUMMA: Scalable universal matrix multiplication algorithm,”
  [www.netlib.org/lapack/lawns/lawn96.ps](http://www.netlib.org/lapack/lawns/lawn96.ps)
Communication domains

- Summa motivates MPI *communication domains*
- Derive communicators that naturally reflect the communication structure along rows and columns of the processor geometry
Communication domains

• A communicator is name space specified by an MPI communicator
• Messages remain within their domain
• Communication domains simplify the code, by specifying subsets of processes that may communicate
• A processor may be a member of more than one communication domain
Splitting communicators

- Each process computes a key based on its rank
- Derived communicators group processes together that have the same key
- Each process has a rank relative to the new communicator
- If a process is a member of several communicators, it will have a rank within each one
Splitting communicators for Summa

• Create a communicator for each row and column
• Group the processors by row
  \[ \text{key} = \text{myid div } \sqrt{P} \]
• Thus, if \( P=4 \)
  – Processes 0, 1, 2, 3 are in one communicator because
    they share the same value of key (0)
  – Processes 4, 5, 6, 7 are in another (1), and so on
MPI support

• **MPI_Comm_split()** is the workhorse

```c
MPI_Comm_split(MPI_Comm comm,
               int splitKey,
               int rankKey,
               MPI_Comm* newComm);
```

• A collective call

• Each process receives a new communicator, `newComm`, which it shares in common with other processes having the same `splitKey` value
Establishing row communicators

```c
MPI_Comm rowComm;
MPI_Comm_split( MPI_COMM_WORLD, myRank / √P, myRank, &rowComm);
MPI_Comm_rank(rowComm,&myRow);
```

- Ranks are unique within a communicator
- Ordered according to `rankKey = myRank / √P`
More on Comm_split

MPI_Comm_split(MPI_Comm comm,
    int splitKey,
    int rankKey,
    MPI_Comm* newComm);

• Ranks are assigned arbitrarily among processes sharing the same rankKey value
• May exclude a process by passing the constant MPI_UNDEFINED as the splitKey
• A special MPI_COMM_NULL communicator will be returned
• Alternative is to enumerate all nodes in a set and call MPI_Group_incl() and MPI_Comm_create()
Panel Broadcast

- Each row/column calls Bcast, a multicast
- Contributing row/column circulates across and downward

Foreach step in 0 to n by panel
  - Ring Bcast(current column, comm_col)
  - Ring Bcast(current row, comm_row)
  - DGEMM(

![Diagram of Panel Broadcast with multicast A and multicast B]
Ring BCast

RING_Bcast( double *buf, int count, 
    MPI_Datatype type, int root, 
    MPI_Comm comm )

MPI_Comm_rank( comm, &rank );
MPI_Comm_size( comm, &np );

if ( rank ≠ root )
    MPI_Recv( buf, count, type, (rank-1+np) mod np, 
        MPI_ANY_TAG, comm, &status );

if ( ( rank +1 ) mod np ≠ root )
    MPI_Send(buf, count, type, (rank+1)%np, 0, comm );
Occupancy Calculator
1.) Select Compute Capability (click): 1.3

2.) Enter your resource usage:
- Threads Per Block: 256
- Registers Per Thread: 8
- Shared Memory Per Block (bytes): 2048

(Don’t edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:

<table>
<thead>
<tr>
<th>GPU Occupancy Data</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Threads per Multiprocessor</td>
<td>1024</td>
</tr>
<tr>
<td>Active Warps per Multiprocessor</td>
<td>32</td>
</tr>
<tr>
<td>Active Thread Blocks per Multiprocessor</td>
<td>4</td>
</tr>
<tr>
<td>Occupancy of each Multiprocessor</td>
<td>100%</td>
</tr>
</tbody>
</table>

Physical Limits for GPU:

<table>
<thead>
<tr>
<th>Physical Limit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads / Warp</td>
<td>32</td>
</tr>
<tr>
<td>Warps / Multiprocessor</td>
<td>32</td>
</tr>
<tr>
<td>Threads / Multiprocessor</td>
<td>1024</td>
</tr>
<tr>
<td>Thread Blocks / Multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>Total # of 32-bit registers / Multiprocessor</td>
<td>16384</td>
</tr>
<tr>
<td>Register allocation unit size</td>
<td>512</td>
</tr>
<tr>
<td>Shared Memory / Multiprocessor (bytes)</td>
<td>16384</td>
</tr>
<tr>
<td>Warp allocation granularity (for register allocation)</td>
<td>2</td>
</tr>
</tbody>
</table>

Allocation Per Thread Block

<table>
<thead>
<tr>
<th>Allocation Per Thread Block</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warps</td>
<td>8</td>
</tr>
<tr>
<td>Registers</td>
<td>2048</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>2048</td>
</tr>
</tbody>
</table>

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor

<table>
<thead>
<tr>
<th>Maximum Thread Blocks Per Multiprocessor</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited by Max Warps / Multiprocessor</td>
<td>4</td>
</tr>
<tr>
<td>Limited by Registers / Multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>Limited by Shared Memory / Multiprocessor</td>
<td>8</td>
</tr>
</tbody>
</table>

Thread Block Limit Per Multiprocessor highlighted RED

CUDA Occupancy Calculator

<table>
<thead>
<tr>
<th>Version</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.5</td>
</tr>
<tr>
<td>Threads</td>
<td>Warps/Multiprocessor</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>48</td>
<td>16</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>80</td>
<td>24</td>
</tr>
<tr>
<td>96</td>
<td>24</td>
</tr>
<tr>
<td>112</td>
<td>32</td>
</tr>
<tr>
<td>128</td>
<td>32</td>
</tr>
<tr>
<td>144</td>
<td>30</td>
</tr>
<tr>
<td>160</td>
<td>30</td>
</tr>
<tr>
<td>176</td>
<td>30</td>
</tr>
<tr>
<td>192</td>
<td>30</td>
</tr>
<tr>
<td>208</td>
<td>28</td>
</tr>
<tr>
<td>224</td>
<td>28</td>
</tr>
<tr>
<td>240</td>
<td>32</td>
</tr>
<tr>
<td>256</td>
<td>32</td>
</tr>
<tr>
<td>272</td>
<td>27</td>
</tr>
<tr>
<td>288</td>
<td>27</td>
</tr>
<tr>
<td>304</td>
<td>30</td>
</tr>
<tr>
<td>320</td>
<td>30</td>
</tr>
<tr>
<td>336</td>
<td>22</td>
</tr>
<tr>
<td>352</td>
<td>22</td>
</tr>
<tr>
<td>368</td>
<td>24</td>
</tr>
<tr>
<td>384</td>
<td>24</td>
</tr>
<tr>
<td>400</td>
<td>26</td>
</tr>
<tr>
<td>416</td>
<td>26</td>
</tr>
<tr>
<td>432</td>
<td>28</td>
</tr>
<tr>
<td>448</td>
<td>28</td>
</tr>
<tr>
<td>464</td>
<td>30</td>
</tr>
<tr>
<td>480</td>
<td>30</td>
</tr>
<tr>
<td>496</td>
<td>32</td>
</tr>
<tr>
<td>512</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>