Lecture 14

Performance Profiling
Under the hood of MPI
Parallel Matrix Multiplication
MPI Communicators
Announcements
Today’s lecture

- Performance Profiling
- MPI
  - What goes on under the hood
  - Performance
  - Working with Communicators
- Application
  - Parallel Matrix Multiplication
Profiling Code

• Gprof: a standard Unix profiling utility
  ‣ Compile with –pg –g
  ‣ Run: ./lu -n 1024 –h
  ‣ Get Profile: gprof <executable> [gmon.out]

• Cachegrind
  ‣ Detailed line by line memory system behavior
  ‣ Difficult to use with multithreaded code
  ‣ Compile with –g, run (slowdown), collect profile
    • valgrind --tool=cachegrind ./lu -n 512 –h
    • cg_annotate cachegrind.out.24798 --auto=yes
GPROF

Flat profile: Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>time</th>
<th>cumulative</th>
<th>self</th>
<th>seconds</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>96.12</td>
<td>1.24</td>
<td>1.24</td>
<td>2</td>
<td>620.00</td>
<td>elim</td>
</tr>
<tr>
<td>3.10</td>
<td>1.28</td>
<td>0.04</td>
<td>2</td>
<td>20.00</td>
<td>mtxdya</td>
</tr>
<tr>
<td>0.78</td>
<td>1.29</td>
<td>0.01</td>
<td>2</td>
<td>5.00</td>
<td>initHMat</td>
</tr>
</tbody>
</table>

Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.78% of 1.29 seconds

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.24</td>
<td>0.00</td>
<td>2/2</td>
<td></td>
<td></td>
<td>main [1]</td>
</tr>
<tr>
<td>[2]</td>
<td>96.1</td>
<td>1.24</td>
<td>0.00</td>
<td>2</td>
<td></td>
<td>elim(int) [2]</td>
</tr>
<tr>
<td></td>
<td>0.01</td>
<td>0.04</td>
<td>2/2</td>
<td></td>
<td></td>
<td>main [1]</td>
</tr>
<tr>
<td>[3]</td>
<td>3.9</td>
<td>0.01</td>
<td>0.04</td>
<td>2</td>
<td>initHMat(..) [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.04</td>
<td>0.00</td>
<td>2/2</td>
<td></td>
<td>mtxdya(..) [4]</td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td>3.1</td>
<td>0.04</td>
<td>0.00</td>
<td>2</td>
<td>initHMat(..) [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.04</td>
<td>0.00</td>
<td>2/2</td>
<td></td>
<td>mtxdya(..) [4]</td>
<td></td>
</tr>
</tbody>
</table>
Cachegrind applied to LU

./lu -n 512 –h Cachegrind → cachegrind.out.21351

==21351== I refs: 723,912,962
==21351== I1 misses: 1,898
==21351== L2i misses: 1,743
==21351== I1 miss rate: 0.00%
==21351== L2i miss rate: 0.00%

==21351== D refs: 303,085,975 (205,600,429 rd + 97,485,546 wr)
==21351== D1 misses: 12,123,062 (11,987,562 rd + 135,500 wr)
==21351== L2d misses: 72,258 (5,773 rd + 66,485 wr)
==21351== D1 miss rate: 3.9% (5.8% + 0.1%)
==21351== L2d miss rate: 0.0% (0.0% + 0.0%)
==21351== L2 refs: 12,124,960 (11,989,460 rd + 135,500 wr)
==21351== L2 misses: 74,001 (7,516 rd + 66,485 wr)
==21351== L2 miss rate: 0.0% (0.0% + 0.0%)
The image contains a page from a document titled "Getting line by line behavior." It appears to be a part of a computer science course, possibly related to a programming assignment. The text includes command output and some annotations related to the execution of a program. Here is a structured representation of the content:

### Getting line by line behavior

The document describes the use of `cg_annotate` tool, along with some input and output data. Below is a structured representation of the key parts of the text:

#### Command Output

- **cg_annotate cachegrind.out.21351 --auto=yes**

<table>
<thead>
<tr>
<th>Ir</th>
<th>I1mr</th>
<th>I2mr</th>
<th>Dr</th>
<th>D1mr</th>
<th>D2mr</th>
<th>Dw</th>
<th>D1mw</th>
<th>D2mw</th>
</tr>
</thead>
<tbody>
<tr>
<td>723,912,962</td>
<td>1,898</td>
<td>1,743</td>
<td>205,600,429</td>
<td>11,987,562</td>
<td>5,773</td>
<td>97,485,546</td>
<td>135,500</td>
<td>66,485</td>
</tr>
</tbody>
</table>

**PROGRAM TOTALS**

<table>
<thead>
<tr>
<th>Ir</th>
<th>I1mr</th>
<th>I2mr</th>
<th>Dr</th>
<th>D1mr</th>
<th>D2mr</th>
<th>Dw</th>
<th>D1mw</th>
<th>D2mw</th>
</tr>
</thead>
<tbody>
<tr>
<td>632,246,864</td>
<td>5</td>
<td>5</td>
<td>181,131,170</td>
<td>11,641,637</td>
<td>0</td>
<td>89,814,500</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

#### Auto-annotated Source

```
```

---

This document appears to be part of a computer science course, possibly for Winter 2010, as indicated by the footer, which reads: "©2010 Scott B. Baden / CSE 160 / Winter 2010."
Drilling down

<table>
<thead>
<tr>
<th>Ir</th>
<th>Ilmr</th>
<th>I2mr</th>
<th>Dr</th>
<th>D1mr</th>
<th>D2mr</th>
<th>Dw</th>
<th>file:function</th>
</tr>
</thead>
<tbody>
<tr>
<td>632,246,864</td>
<td>5</td>
<td>5</td>
<td>181,131,170</td>
<td>11,641,637</td>
<td>0</td>
<td>89,814,500</td>
<td>/home/linux/ieng6/cs160w/public/HW/A3/elim.cpp:elim(int)</td>
</tr>
</tbody>
</table>

- 788,984 | 0 | 0 | 2,044 | 0 | 0 | 1,022 for ( i = k+1; i < N; i++ ) {
  1,308,160 | 0 | 0 | 1,046,528 | 273,748 | 0 | 0 double Aik = A[i][k] |
  . . . . . . . |
  . . . . . . . |
  . . . . . . . |
  double *Ai = A[i];
  267,911,168 | 0 | 0 | 0 | 0 | 0 | 0 for ( j = k+1; j < N; j++ )
  356,866,048 | 0 | 0 | 178,433,024 | 11,190,884 | 0 | 89,216,512 Ai[j] -= Aik * A[k][j];
Today’s lecture

• Performance Profiling
• MPI
  ‣ What goes on under the hood
  ‣ Performance
  ‣ Working with Communicators
• Application
  ‣ Parallel Matrix Multiplication
Buffering

- If there is no pending receive, then an incoming message is placed in an \textit{anonymous} system buffer
- When a receive is \textit{posted}, the message is moved into the buffer specified in the call
- Double copying reduces communication performance
- Non-blocking communication can help avoid this problem
- \textit{MPI: The Complete Reference}, by Marc Snir et al.  
  “Buffering and Safety”
Correctness and fairness

- When there are multiple outstanding iRecvs, MPI doesn’t say how incoming messages are matched…
- Or even if the process is fair

1. Iteration 1: 1 → 2&0  0 → 1  (0 → 2)  2 → 0&1
2. 1 begins iteration 2: 1 → 2
3. 0 → 2 (but for iteration 1)
4. Problem: irecv in P2 receiving data from P1 in iteration 2
   But it expects data from P0 in iteration 1

for i = 1 to n
  MPI_Request req1, req2; MPI_Status status;
  MPI_Irecv(buff, len, CHAR, ANY_NODE, TYPE, WORLD,&req1);
  MPI_Irecv(buff2,len, CHAR, ANY_NODE, TYPE, WORLD,&req2);
  MPI_Send(buff, len, CHAR, nextnode, TYPE, WORLD);
  MPI_Send(buff, len, CHAR, prevnode, TYPE, WORLD);
  MPI_Wait(&req1, &status);
  MPI_Wait(&req2, &status);
end for
Rendezvous

• When a message is to be sent, can MPI just send the message?

• For “short” message, it can. This is **eager mode**

• The **eager limit** is the longest message that can be sent in eager mode

• For long messages, MPI first sends a scout to get permission to send the message

• This is called **rendezvous mode**
Where does the time go?

- Under ideal conditions…
  - There is a pending receive waiting for an incoming message, which is transmitted directly to and from the users message buffer
  - There is no other communication traffic
- Assume a contiguous message
- LogP model (Culler et al, 1993)
Communication performance

• The simpler $\alpha, \beta$ model is often good enough
• Let the message length $= n$
• Message passing time $= \alpha + \beta^{-1} \infty \ n$

$\alpha =$ message startup time

$\beta_\infty =$ peak bandwidth (bytes per second)

• “Short” messages: startup term dominates

$\alpha >> \beta^{-1} \infty \ n$

• “Long” messages: bandwidth term dominates

$\beta^{-1} \infty \ n >> \alpha$
Typical bandwidth curve  (SDSC Triton)

\[ \beta_\infty = 1.12 \text{ GB/sec} \]

@ \[ N = 8 \text{ MB} \]

\[ \alpha = 3.2 \mu\text{sec} \]

\[ N_{1/2} \approx 20 \text{ KB} \]
Short message behavior
Intermediate length message behavior

\[ N_{1/2} \approx 20 \text{ KB} \]

\[ \sim 4K \quad \sim 32K \]
Measuring communication performance with the Ring program

- Configure the processors logically in a ring and pass messages around the ring multiple times
- Assume there are $p$ processors
- Neighbors of processor $k$ are
  - $(k + 1) \mod p$
  - $(k + p - 1) \mod p$
Measurement technique with Ring

for (int len = 1, l=0; len <= maxSize; len *= 2, l++)
    if (myid == 0) {
        // (WARM UP CODE)
        const double start = MPI_Wtime();
        for (int i = 0; i < trips; i++) {
            PROCESSOR 0 CODE
        }
        const double delta = MPI_Wtime() - start;
        Bandwidth = (long)((trips*len*nodes)/ delta /1000.0);
    } else { // myid != 0
        // (WARM UP CODE)
        for (int i = 0; i < trips; i++) {
            ALL OTHER PROCESSORS
        }
    }
}
The Ring program

Processor 0:

```c
MPI_Request req;
MPI_Irecv(buffer, len, MPI_CHAR, (rank + p - 1)%p,
          tag, MPI_COMM_WORLD, &req);
MPI_Send(buffer, len, MPI_CHAR, (rank + 1) % p,
         tag, MPI_COMM_WORLD);
MPI_Status status;
MPI_Wait(&req,&status);
```

All others:

```c
MPI_Status status1;
MPI_Recv(buffer, len, MPI_CHAR, (rank + p - 1)%p,
         tag, MPI_COMM_WORLD, &status1);
MPI_Send(buffer, len, MPI_CHAR, (rank+1)%p,
         tag, MPI_COMM_WORLD);
```
Today’s lecture

• Performance Profiling
• MPI
  ‣ What goes on under the hood
  ‣ Performance
  ‣ Working with Communicators
• Application
  ‣ Parallel Matrix Multiplication
Matrix multiplication

• The simplest formulation of matrix multiply the “ijk” formulation

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

• The “kij” formulation is the basis for an efficient parallel algorithm

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

• The matrices may be non-square

\[
\text{for } k := 0 \text{ to } n3-1 \\
\quad \text{for } i := 0 \text{ to } n1-1 \\
\quad \quad \text{for } j := 0 \text{ to } n2-1 \\
C[i,j] += A[i,k] \times B[k,j]
\]

\[
\text{C[i,:]} += A[i,k] \times B[k,:]
\]

• The two innermost loop nests compute \( n3 \) outer products

\[
\text{for } k := 0 \text{ to } n3-1 \\
C[::] += \text{A[::,k]} \times B[k,:]
\]

where \( \times \) is outer product
Sidebar on 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 $b = \text{matrix } C$

\[
\begin{pmatrix}
ax & ay & az \\
bx & by & bz \\
 cx & cy & cz \\
\end{pmatrix}
\]

\[
\begin{array}{|ccc|}
\hline
* & 1 & 2 & 3 \\
\hline
10 & 11 & 20 & 30 \\
20 & 20 & 40 & 60 \\
30 & 30 & 60 & 90 \\
\hline
\end{array}
\]

• Multiplication table with rows formed by $a[:]$ and the columns by $b[:]$
Outer Product Formulation

- The new algorithm computes $n$ partial outer products:

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

```
A       B        C      D
A0      B0     C0    D0
A1      B1     C1    D1
A2     B2     C2     D2
A3     B3     C3     D3
```

“Inner product” formulation:

for $i := 0 \text{ to } n-1, \ j := 0 \text{ to } n-1$
$$C[i,j] += A[i,:] \cdot B[:,j]$$
Serial algorithm

• Each row $k$ of $B$ contributes to the $n$ partial outer products:

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

- Processors organized into rows and columns, process rank an ordered pair
- Processor geometry $P = px \times py$
- Blocked (serial) matrix multiply, panel size $= b << N/\max(px,py)$

$$\text{for } k := 0 \text{ to } n-1 \text{ by } b$$

- $\text{multicast } A[ :, k:k+b-1 ]$ \quad Along processor rows
- $\text{multicast } B[ k:k+b-1, : ]$ \quad Along processor columns
- $C += A[:,k:k+b-1] * B[k:k+b-1,:]$ \quad Local MM

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

©2010 Scott B. Baden / CSE 160 / Winter 2010
Parallel matrix multiplication

- Assume \( p \) is a perfect square
- Each processor gets an \( \frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}} \) 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

\[
\begin{array}{ccc}
p(0,0) & p(0,1) & p(0,2) \\
p(1,0) & p(1,1) & p(1,2) \\
p(2,0) & p(2,1) & p(2,2) \\
\end{array}
\]
What is the performance?

for k := 0 to n−1 by b

// Tree broadcast: $\lg \sqrt{p} \left( \alpha + b\beta n/\sqrt{p} \right)$
// For long messages: $2(\sqrt{p} - 1)/p)bn$

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: $\sim 2n^3/p + 4\beta bn/\sqrt{p}$
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 ⇒ less memory, lower efficiency
  - $b$ large ⇒ 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
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
• Specify subsets of communicating processes
• A processor may be a member of more than one communication domain
Splitting communicators for Summa

• Create a communicator for each row and column
• Group the processors by row
  \[ \text{key} = \frac{\text{myid}}{\sqrt{P}} \]
• A communicator groups processes together that have the same key
• 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 same **splitKey** value

• Each process computes a key based on its rank
Establishing row communicators

\[
\text{MPI} \_\text{Comm} \ \text{rowComm};
\]

\[
\text{MPI} \_\text{Comm}\_\text{split}( \text{MPI} \_\text{COMM} \_\text{WORLD}, \text{myRank} / \sqrt{P}, \text{myRank}, &\text{rowComm});
\]

\[
\text{MPI} \_\text{Comm}\_\text{rank}(\text{rowComm}, &\text{myRow});
\]

- Ranks apply only to the respective communicator
- Ordered according to myRank

<table>
<thead>
<tr>
<th>Y0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>P1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P4</td>
<td>P5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P8</td>
<td>P9</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P12</td>
<td>P13</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
More on Comm_split

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

• Ranks assigned arbitrarily among processes sharing the same rankKey value
• May exclude a process by passing the constant MPI_UNDEFINED as the splitKey
• Return a special MPI_COMM_NULL communicator
• If a process is a member of several communicators, it will have a rank within each one
• Each process has a rank relative to the new communicator
Panel Broadcast

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

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

  Ring Bcast(current row, comm_row)

DGEMM ( )

Multicast A

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 );
Summary

• Performance profiling: gprof & cachegrind
• MPI
  ‣ Internal behavior
  ‣ Performance: $\alpha$, $\beta$ model and the Ring program
  ‣ Grouping processes with communicators
• Application: Matrix Multiplication
Fin