Lecture 6

Scalability and performance metrics
Communicators
Matrix multiplication

Announcements

• Corrections to Lecture #5 have been posted: reprint from page 33 to the end
• For the Blue Horizon IBM SP system at the Supercomputer Center (with Power3 CPUs)
  \[ \phi \approx 10^{-8} \approx 4\beta \]
  
  \[ T(1,(m,n)) = 4\beta mn \]
• Datum are 8-byte double precision numbers, message passing time for a message of length N is
  \[ \alpha + 8\beta N \]
Valkyrie Announcements

• Valkryie is up, with all but node 7 operation
• If you are having troubles with a job terminating silently (i.e. no output)... You may have runaway processes
• Use the cluster-ps command to find them

```
/usr/sbin/cluster-ps cs160x**
```

```
compute-0-0:
    cs160x** 26947  40.3  2.1  22936  21644 ? R  03:44  56:11
    /home/cs160x/cs160x**/HW1A/message

compute-0-1:
    cs160x** 25906  42.7  2.1  22940  21648 ? R  03:44  59:31
    /home/cs160x/cs160x**/HW1A/message

compute-0-2:
    cs160x** 24547  42.5  2.1  22940  21648 ? R  03:44  59:17
    /home/cs160x/cs160x**/HW1A/message

compute-0-3:
    cs160x** 24423  38.7  2.1  22940  21648 ? R  03:44  54:02
    /home/cs160x/cs160x**/HW1A/message
```
Valkyrie Announcements

• To remove these processes
/usr/sbin/cluster-kill cs160x**

• Follow the instructions on the “Getting started with Valkyrie” web page

Parallel speedup and efficiency

• 1-D decomposition
\[ S_P = \frac{T_1}{T_P} = \frac{16N^2\beta}{(16N^2\beta/P + 2(\alpha+8\beta N))} \]
\[ E_P = \frac{S_P}{P} = \frac{16N^2\beta}{(16N^2\beta + 2P(\alpha+8\beta N))} = \frac{1}{1 + (\alpha+8\beta N)P/(8N^2\beta)} \]

• 2-D decomposition
\[ S_P = \frac{T_1}{T_P} = \frac{16N^2\beta/(16N^2\beta/P+4(\alpha+8\beta N\sqrt{P}))}{16N^2\beta+4(\alpha P+8\beta N\sqrt{P})/(4N^2\beta)} \]
\[ E_P = \frac{S_P}{P} = \frac{1}{1 + (\alpha P+8\beta N\sqrt{P})/(4N^2\beta)} \]
Putting these formulas to work

- 1-D decomposition
  \[ E_p = \frac{1}{1 + (\alpha+8\beta N)P \over (8N^2\beta)} \]
- What is the efficiency for \( N=64, P=8? \)
  0.29

Today’s lecture

- Scalability and performance metrics
- Matrix Multiplication
- Communicators
Scalability

- Earlier we talked about the isoefficiency function…
- This function tells us how quickly serial work $W$ must grow as we increase $P$. so that the efficiency will remain constant
- We now consider scalability in greater detail

Overhead

- Ideally, $T_P \equiv W/P$, or $W \equiv P T_P$
- In practice, $T_P > W/P$. Why is this?
- We define $T_o \equiv PT_P - W$ as the total overhead or the overhead function
- $T_o$ is the total time spent by all the processors in excess of that spent by the serial algorithm
- We call $PT_P$ as the cost, the work, or the processor-time product
- Note that $E_P = W / (P T_P) = W / cost$
Cost optimality

• If the cost of solving a problem in parallel grows at the same rate as that of the serial algorithm, then we say that the problem is cost optimal.
• This implies that the cost should grow at the same rate as $W$, i.e. $PT_p = \Theta(W)$
• Consider adding $n$ numbers: the serial algorithm runs in time $n \log n$
• $T_n = (\log n)^2$ on $P=n$ processors: the cost is $n(\log n)^2$
• The algorithm is not cost optimal—if it were, the cost would be $n \log n$—but is not far off.

A Cost optimal computation

• Adding $n$ numbers on $n$ processors is not cost-optimal
• But adding $n$ numbers on $p<n$ processors is cost-optimal
• Each processor locally sums its $n/p$ values: $\Theta(n/p)$
• Then a log-time global summation $\Theta(\log p)$
• $T_p = \Theta(n/p + \log p)$
• Cost = $\Theta(n + p \log p)$
• So long as $n = \Omega(p \log p)$, cost = $\Theta(n) = W$, and the algorithm is cost-optimal.
Why does efficiency decrease with P?

- Recall that efficiency \( E_P = \frac{W}{PT_p} \)
- Plugging in the overhead equation \( T_o \equiv PT_p - W \) we have \( E_P = \frac{1}{1 + T_o/W} \)
- Note that if \( W \) remains fixed, then overhead increases with \( P \), and efficiency must therefore decrease

Scalability

- We say that a system is **scalable** if we can maintain a (nearly) constant running time as we increase the work with the number of processors
- Equivalently, a system is scalable if we can maintain a constant level of parallel efficiency by increasing the work with the number of processors
- When we think about scalability we ask: “how quickly must the computational work grow with \( P \)?”
Scalability and cost optimality are connected

- We can always make a scalable parallel system cost-optimal if we choose an appropriate value of P and N

Isoefficiency

- The isoefficiency function of a computation tells us how quickly the workload must grow as a function of P in order to maintain a constant level of efficiency.
- If the isoefficiency function \( f \) is related to \( W \) as \( W = \Omega(f(p)) \), we have a cost-optimal computation.
- Since \( E = 1/(1 + T_o/W) \), let’s re-write \( W \) as \( (E/(1-E)) T_o \).
- The isoefficiency function is \( KT_o \), where \( K=E/(1-E) \).
Looking at isoefficiency

- The larger the isoefficiency function, the less scalable is the system

- Consider the ODE solver
  - \( N \) = Problem size
  - \( P \) = Number of processors
  - Computational work = \( W = 3N = T_1 \)

Isoefficiency function for the ODE solver

- Let a floating point multiply or add take unit time
- Normalized message start time = \( \alpha \)
- Parallel running time \( T_P \)
  - Perfect parallelization of \( W + \) communication verheads
    \[ \frac{N}{P} + 2\alpha \]
- Parallel efficiency
  \[ E_P = \frac{T_1}{P T_P} = \frac{3N}{2\alpha P + 3N} \]
- Rewriting to obtain isoefficiency function
  \[ N = (2/3) \alpha P \left( \frac{E_P}{1 - E_P} \right) = \Theta(P) \]
- \( T_o = PT_P - W = (N + 2\alpha P) - N = 2\alpha P = \Theta(P) \)
- The solver is cost optimal since \( T_o = \Omega(P) \)
Isoefficiency of summation

- Summing N numbers on P processors
  - \( W = N-1, T_p = (N-1)/P + \alpha \log P \)
  - \( E_p = N/(N + \alpha P \log(P)) \)
- Isoefficiency function
  - \( N = (2E_p/(1 - E_p)) \alpha P \log(P) = \Theta(p \log p) \)
- Overhead \( T_o = PT_p - W \)
  - \( = N + \alpha P \log P - N = \alpha P \log P = \Omega(p \log p) \)
- Summation is cost-optimal
  - But not very scalable
  - As we increase the number of processors from 32 to 1024 (x32), we must increase the work by a factor 160
  - We may run out of memory

Interpreting the result

- While summing N numbers on P processors is cost-optimal, the system is not very scalable
- For example, as we increase the number of processors from 32 to 1024 (x32), we must increase the work by a factor 160
  - We may ultimately run out of memory
Matrix multiplication

Matrix Multiplication

• Given two *conforming* matrices $A$ and $B$, form the matrix product $A \times B$
• Second dimension of $A$ must equal first dimension of $B$
  
  $A$ is $m \times n$
  $B$ is $n \times p$

• Let’s assume that the matrices are square:
  $n \times n$
• Operation count is $O(n^3)$
Matrix multiply algorithm

```cpp
function MM(Matrix A, Matrix B, Matrix C)
    for i := 0 to n - 1
        for j := 0 to n - 1 do
            C[i, j] = 0;
            for k := 0 to n - 1
                C[i, j] += A[i, k] * B[k, j];
        end
    end MM
```

Memory locality

- The order that we access data affects performance
- Go “with” the grain of the cache:
  - IJ loop: 0.48706 secs
    for (i=0; i<N; i++)
    for (j=0; j<N; j++)
    a[i][j] += b[i][j];
  - JI loop: 2.18245 secs
    for (int j=0; j<N; j++)
    for (i=0; i<N; i++)
    a[i][j] += b[i][j];

<table>
<thead>
<tr>
<th>0 1 2 3</th>
<th>0 4 8 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 5 6 7</td>
<td>1 5 9 13</td>
</tr>
<tr>
<td>8 9 10 11</td>
<td>2 6 10 14</td>
</tr>
<tr>
<td>12 13 14 15</td>
<td>3 7 11 15</td>
</tr>
<tr>
<td>8 Misses</td>
<td>16 Misses</td>
</tr>
</tbody>
</table>
Reuse

• Memory access times are much slower than cache
• An L2 cache miss might cost 50 to 100 cycles, and is increasing
• The success of caching depends on the ability to re-use previously cached data
  – Such re-use exhibits temporal locality
  – Re-use depends on the ability of the application to live within the capacity of the cache

Blocking for cache (tiling)

• Amortize memory accesses by increasing memory reuse
• Discussion follows from James Demmel, UC Berkeley
  (http://www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect02.html)
Matrix Multiplication

\[
C += C + A \times B
\]

for \(i := 0\) to \(n-1\)

for \(j := 0\) to \(n-1\)

for \(k := 0\) to \(n-1\)

\[
C[i,j] += A[i,k] \times B[k,j]
\]

Analysis of performance

for \(i = 0\) to \(n-1\)

// for each iteration \(i\), load all of \(B\) into cache

for \(j = 0\) to \(n-1\)

// for each iteration \((i,j)\), load \(A[i,:]\) into cache

// for each iteration \((i,j)\), load and store \(C[i,j]\)

for \(k = 0\) to \(n-1\)

\[
C[i,j] += A[i,k] \times B[k,j]
\]
Analysis of performance

for i = 0 to n-1
    // n × n^2 / b loads = n^3/b, where b=cache line size
    for j = 0 to n-1
        // for each iteration (i,j), load A[i,:] into cache
        // for each iteration (i,j), load and store C[i,j]
        for k = 0 to n-1
            C[i,j] += A[i,k] * B[k,j]
Analysis of performance

for i = 0 to n-1
  // n × n^2 / b loads = \( n^3 / b \), where b=cache line size
  for j = 0 to n-1
    // n^2 × b loads = n^2 / b
    // n^2 / b loads and n^2 / b stores = 2n^2 / b
    for k = 0 to n-1
      C[i,j] += A[i,k] * B[k,j]

Total cost: \( (n^3 + 3n^2) / b \)
Memory reuse

• Total cost: \( \frac{(n^3 + 3n^2)}{b} \)

• Reuse = \( q \)

\[
\frac{\text{total number of different blocks accessed}}{\text{total number of accesses}}
\]

• \( \frac{2n^3}{(n^3 + 3n^2)} \)
  \( \approx 2 \) as \( n \to \infty \)

Blocked Matrix Multiply

• Let’s consider \( A, B, C \) to be \( N \) by \( N \) matrices consisting of \( b \times b \) subblocks
  – \( b = n/N \) is called the blocksize
  – how do we establish \( b \)?
  – assume we have a good quality library to perform matrix multiplication on subblocks

\[
C(i,j) = C(i,j) A(i,k) * B(k,j)
\]
The algorithm

for i = 0 to N-1
    for j = 0 to N-1
        // read block C[i,j] into cache
        for k = 0 to N-1
            // read block A[i,k] into cache
            // read block B[k,j] into cache
        //write block C[i,j] to memory

Analysis of performance

for i = 0 to N-1
    for j = 0 to N-1
        // read each block C[i,j] once : \( n^2 \)
        for k = 0 to N-1
            // read blocks of A & B: N^3 times
            // \( = N^3 \times (n/N)^2 = 2Nn^2 \)
        // write each block C[i,j] once : \( n^2 \)
Reuse

- Total cost: \( \frac{n^3 + 3n^2}{b} \)

- Reuse = \( q \)
  
  \[
  \frac{\text{total number of different blocks accessed}}{\text{total number of accesses}}
  \]

- \( \frac{2n^3}{(2N+2)n^2} = \frac{n}{(N+1)} \)

  \[
  \approx \frac{n}{N} = b \quad \text{as } n \to \infty
  \]

More on blocked algorithms

- Data in the sub-blocks are contiguous within rows only
- We may incur conflict cache misses
- Idea: since re-use is so high… let’s copy the subblocks into contiguous memory before passing to our matrix multiply routine


http://www-suif.stanford.edu/papers/lam91.ps

![Miss rate vs. Blocking factor graph](image-url)
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

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

A simple parallel algorithm

• Apply the basic algorithm but treat each element $A[i,j]$ as a block rather than a single element
• Thus, $A[i,k] \times B[k,j]$ is a matrix multiply in $C[i, j] += A[i, k] \times B[k, j]$

\[
\begin{array}{ccc}
 p(0,0) & p(0,1) & p(0,2) \\
p(1,0) & p(1,1) & p(1,2) \\
p(2,0) & p(2,1) & p(2,2)
\end{array}
\]
A simple parallel algorithm

- Apply the basic algorithm but treat each element $A[i,j]$ as a block rather than a single element.
- Thus, $A[i,k] \times B[k,j]$ is matrix multiply in $C[i,j] += A[i,k] \times B[k,j]$.
Cost

- Each processor performs $n^3/p$ multiply-adds
- But a significant amount of communication is needed to collect a row and a column of data onto each processor
- Each processor broadcasts a chunk of data of size $n^2/p$ within a row and a column of $\sqrt{p}$ processors
- Disruptive - distributes all the data in one big step
- High memory overhead
  - needs $2\sqrt{p}$ times the storage needed to hold A & B

Observation

- In the broadcast algorithm each processor multiplies two skinny matrices of size $n^2/\sqrt{p}$
- But we can form the same product by computing $\sqrt{p}$ separate matrix multiplies involving $n^2/p \times n^2/p$ matrices and accumulating partial results

$$
\text{for } k := 0 \text{ to } n - 1
\text{ do } \text{accumulate } C[i, j] \text{ += } A[i, k] \times B[k, j];
$$
A more efficient algorithm

- Take advantage of the organization of the processors into rows and columns
- Move data incrementally in $\sqrt{p}$ phases, using smaller pieces than with the broadcast approach
- Circulate each chunk of data among processors within a row or column
- In effect we are using a ring broadcast algorithm
- Buffering requirements are $O(1)$
A more efficient algorithm

- Take advantage of the organization of the processors into rows and columns
- Move data incrementally in $\sqrt{p}$ phases, using smaller pieces than with the broadcast approach
- Circulate each chunk of data among processors within a row or column
- In effect we are using a ring broadcast algorithm
- Buffering requirements are $O(1)$

Canon’s algorithm

- Based on the above approach
- A slight reformulation to make things work out
- Consider iteration $i=1, j=2$:

Canon’s algorithm

- \( C[1,2] += A[1,k] \times B[k,2] \), for \( k=0, 1, 2 \)
- We want \( A[1,0] \) and \( B[0,2] \) to reside on the same processor initially
- We then shift each row and column so that the next pair of values \( A[1,1] \) and \( B[1,2] \) line up on the same processor
- And so on with \( A[1,2] \) and \( B[2,2] \)

The steps of Canon’s algorithm

- This works out if we *pre-skew* the matrices
Skewing the matrices

- Canon’s algorithm requires that we **pre-skew** the matrices
- Shift each row $i$ by $i$ columns to the left using sends and receives
- Communication wraps around
- Do the same for each column

```
<table>
<thead>
<tr>
<th>A(0,0)</th>
<th>A(0,1)</th>
<th>A(0,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(1,1)</td>
<td>A(1,2)</td>
<td>A(1,0)</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(2,2)</td>
<td>A(2,0)</td>
<td>A(2,1)</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>B(0,0)</th>
<th>B(1,1)</th>
<th>B(2,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(1,0)</td>
<td>B(2,1)</td>
<td>B(0,2)</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(2,0)</td>
<td>B(0,1)</td>
<td>B(1,2)</td>
</tr>
</tbody>
</table>
```

Shift and multiply

- Takes $\sqrt{p}$ steps
- Circularly shift
  - each row by 1 column to the left
  - each column by 1 row to the left
- Each processor forms the product of the two local matrices adding into the accumulated sum

$$C(1,2) = A(1,0) \times B(0,2) + A(1,1) \times B(1,2) + A(1,2) \times B(2,2)$$
Cost of Canon’s algorithm – Pre skewing

forall i=0 to q-1  // q = √p
    CShift-left A[i; :] by i  // T = a+βn^2/p

forall j=0 to q-1
    Cshift-up B[: , j] by j  // T = a+βn^2/p

Cost of computational loop

for k=0 to q-1
    forall i=0 to q-1 and j=0 to q-1
    forall i=0 to q-1
        CShift-leftA[i; :) by 1  // T= a+βn^2/p
    forall j=0 to q-1
        Cshift-up B[ : , j] by 1  // T= a+βn^2/p
Cost of Canon’s algorithm

- \( T_p = 2n^3/p + 4qa + 4\beta n^2/q \)
- \( E_p = 2n^3/(pT_p) = 1/(1 + 2a(q/n)^3 + 2\beta q/n) = 1/(1 + O(\sqrt{p/n})) \)

- Grows to 1 as \( n/\sqrt{p} = \sqrt{\text{data per processor}} \)
- But there are drawbacks
  - We need to provide added storage for the shifted in matrices
  - Various constraints make the algorithm hard to generalize to real world situations

Drawbacks

- Awkward if …
  - \( P \) is not a perfect square
  - \( A \) and \( B \) are not square, and not evenly divisible by \( \sqrt{p} \)
- Interoperation with applications and other libraries difficult or expensive
MPI Communicators

- MPI Communicators provide a way of hiding internal behavior of a library written using MPI
- If we call a library routine, we don’t want the message passing activity in the library to interfere with our program
- A communicator specifies a name space called a Communication Domain
- Messages remain within their communication domain

Implementing Cannon’s algorithm

- Cannon’s algorithm provides a good motivation for using MPI communication domains
- Communication domains simplify the code, by specifying subsets of processes that may communicate
- We may structure the sets in any way we like
- Each processor may be a member of more than one communication domain
- We will define new sets of communicators that naturally reflect the structure of communication along rows and columns
Splitting communicators

- We can create a set of communicators, one for each row and column of the geometry
- Each process computes a key based on its rank
- We then 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 Cannon’s algorithm

- In Cannon’s algorithm, each processes needs to communicate with process within its row and column
- Let’s create a communicator for each row and one for each column
- Consider a grouping of processors by row $\text{key} = \text{myid div } \sqrt{P}$
- Thus, if $P=9$, then
  - Processes 0, 1, 2 are in one communicator because they share the same value of key (0)
  - Processes 3, 4, 5 are in another (1)
  - Processes 6, 7, 8 are in a third (2)
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, which it shares in common with other processes having the same key value

---

Comm_split

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

• Each process receives a unique rank within its respective communicator according to the value of `rankKey`

• The relative values of the ranks follows the ordering of the rankKeys across the processes

• I.e. if A give a rank key of 1, and B a rank key of 10, then A’s rank < B’s rank
More on Comm_split

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

- Ranks are assigned arbitrarily among processes sharing the same `rankKey` value.
- It is also possible to exclude a process from a communicator, by passing the constant `MPI_UNDEFINED` as the `splitKey`.
- A special `MPI_COMM_NULL` communicator will be returned.

Splitting into rows

```c
MPI_Comm rowComm;
int myRow = myid / \sqrt{P};
MPI_Comm_split(MPI_COMM_WORLD,
myRow,
myid,
&rowComm);
```
A ring shift

MPI_Comm_rank(rowComm,&myidRing);
MPI_Comm_size(rowComm,&nodesRing);

int I = myrow, X = ..., XR;

int next = (myidRng + 1) % nodesRing;
MPI_Send(&X,1,MPI_INT,next,0, rowComm);
MPI_Recv(&XR,1,MPI_INT,
         MPI_ANY_SOURCE,
         0, rowComm, &status);

Other kinds of communication domains

- Cartesian grids permit us to work in different coordinate systems such that the rank is no longer a scalar
- But we can accomplish a good deal of what we want using the splitting method
The Code

- Cartesian grids permit us to work in different coordinate systems such that the rank is no longer a scalar
- But we can accomplish a good deal of what we want using the splitting method
- We use another routine `MPI_Sendrecv_replace()` to simplify the coding
- Sends then receives a message using one buffer
- Code listing on separate handout

`MPI_Sendrecv_replace()`

- Sends then receives a message using one 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 )
```