CSE 160
Lecture 20
Final Exam Review
Announcements

• Final examination
  ‣ Weds, Dec 11, 11.30a to 2.30p, CNTR 212
  ‣ You may bring a single sheet of notebook sized paper “8x10 inches” with notes

• Office hours during examination week
  ‣ Monday (2pm to 4pm)
  ‣ Tuesday (1pm to 2pm)
  ‣ Or by appointment
Global to local mapping

- In assignment #4, each process owns a subset of the data.
- We allocate only the storage owned by a process + ghost cells.
- In some applications, we need to compute a local to global mapping of array indices.
- In the 4th assignment, we set values to zero in certain physical regions of the problem and must use global rather than local indices to determine which values to set.
Improving cache locality by fusing loops

- When arrays don’t fit in cache, we get compulsory misses on both loops

```c
for (j=1; j<=m+1; j++)
    for (i=1; i<=n+1; i++){
        double *pr = &R[j][i], pe = &E[j][i];
        *pe += -dt*(kk * *pe*(*pe-a)*(*pe-1)+*pe * *pr);
    }

for (j=1; j<=m+1; j++)
    for (i=1; i<=n+1; i++){
        double *pr = &R[j][i], pe = &E[j][i];
        *pr += dt*(eps+M1 * *pr/( *pe+M2))*(-pr[0]-kk * *pe*(pe-b-1));
    }
```
Improving cache locality by fusing loops

• Merging loops can reduce the # of compulsory misses

for (j=1; j<=m+1; j++)
    for (i=1; i<=n+1; i++){
        double *pr = &R[j][i], pe = &E[j][i];
        *pe += -dt*(kk * *pe*(pe-a)*(pe-1)+*pe * *pr);
        *pr += dt*(eps+M1 * *pr/( pe+M2))*(-pr[0]-
            kk * *pe*(pe-b-1));
Today’s lecture

• Technology
• Programming
  ‣ Threads
  ‣ Message Passing
• Algorithms (applications)
• List of Keywords and Topics (cross cutting)
Technology: Terms and concepts

• Processor Memory Gap
• Caches (What are the 3 C’s of Cache Misses?)
  ‣ Cache coherence and consistency
  ‣ Snooping
  ‣ False sharing
• MIMD
• Multiprocessors: NUMAs and SMPs
An important principle: locality

- Memory accesses exhibit two forms of locality
  - Temporal locality (time)
  - Spatial locality (space)
- Often involves loops
- Opportunities for reuse
- Idea: construct a small & fast memory to cache re-used data

for \( t=0 \) to \( T-1 \)
for \( i = 1 \) to \( N-2 \)
\[ u[i]=(u[i-1] + u[i+1])/2 \]
Different types of caches

- Separate Instruction (I) and Data (D)
- Unified (I+D)
- Direct mapped / Set associative
- Write Through / Write Back
- Allocate on Write / No Allocate on Write
- Last Level Cache (LLC)
- Translation Lookaside Buffer (TLB)
Cache – a summary

- **cache hit**: an access that finds the data in cache
- **cache miss**: an access that’s not
- **hit time**: time to access the higher cache
- **miss penalty**: time to move data from lower level to upper, then to cpu
- **hit rate**: percentage of time the data is found in the higher cache [**miss rate**: (1 - hit rate)]
- **cache block size** or **cache line size**: the amount of data that gets transferred on a cache miss
- **Instruction (data) cache**: a cache that holds only instructions (data); unified cache holds both I+D
Direct mapped cache

- Simplest cache
- Look up the line indexed by the line index
- Match the stored tag against the higher order address bits
Address Space Organization

- Multiprocessors and multicomputers
- Shared memory, message passing
- With shared memory hardware automatically performs the global to local mapping using address translation mechanisms
  - **UMA**: *Uniform Memory Access* time
    Also called a Symmetric Multiprocessor (SMP)
  - **NUMA**: *Non-Uniform Memory Access* time
Today’s lecture

• Technology

• Programming
  ‣ Threads
    ‣ Message Passing

• Algorithms (applications)

• List of Keywords and Topics (cross cutting)
Threads Programming model

• Start with a single root thread
• Fork-join parallelism to create concurrently executing threads
• Threads communicate via shared memory
• A spawned thread executes asynchronously until it completes
• Threads may or may not execute on different processors
Owner computes rule

```cpp
#pragma omp parallel for schedule(static,CHUNK)
for (int j=1; j<=m+1; j++)
    for (int i=1; i<=n+1; i++)
        E[j][i] = E_prev[j][i]+alpha*(E_prev[j][i+1]+E_prev[j][i-1]-
                                   4*E_prev[j][i]+E_prev[j+1][i]+
                                   E_prev[j-1][i])
```

©2013  Scott B. Baden / CSE 160 / Fall 2013
Multithreading in perspective

• Benefits
  ‣ Harness parallelism to improve performance
  ‣ Ability to multitask to realize concurrency, e.g. display

• Pitfalls
  ‣ Program complexity
    • Partitioning, synchronization, parallel control flow
    • Data dependencies
    • Shared vs. local state (globals like errno)
    • Thread-safety
  ‣ New aspects of debugging
    • Race conditions
    • Deadlock
Implementation & techniques

• SPMD
  ‣ Threads API
  ‣ OpenMP

• Correctness: critical sections, race conditions
  ‣ Mutexes and barriers
  ‣ Atomic
  ‣ Memory fences

• Performance: Data Partitioning
  ‣ Block and cyclic decompositions
  ‣ Dynamic scheduling

• Cross cutting issues (Performance & Correctness)
  ‣ Cache coherence and consistency
  ‣ Cache locality

• Data dependencies, loop carried dependence
Today’s lecture

• Technology
• Threads Programming
  ‣ Correctness
  ‣ Performance
• Algorithms (applications)
• List of Keywords and Topics (cross cutting)
Correctness

• System: necessary condition for user level correctness
  ‣ Cache coherence and consistency
• User: avoid race conditions through appropriate program synchronization
  ‣ Migrate shared updates into main
  ‣ Critical sections
  ‣ Barriers
  ‣ Atomics
  ‣ Fork/Join
Cache Coherence Protocols

- Ensure that all processors eventually see the same value
- Two policies
  - Update-on-write (implies a write-through cache)
  - Invalidate-on-write

```
X:=2
P2
```

```
P1
X:=2
X:=2
X:=2
```

©2013 Scott B. Baden / CSE 160 / Fall 2013
Memory consistency

• Cache coherence tells us that memory will *eventually* be consistent
• The memory consistency policy tells us *when* this will happen
• A memory system is consistent if the following 3 conditions hold
  ▸ Program order (you read what you wrote)
  ▸ Definition of a coherent view of memory ("eventually")
  ▸ Serialization of writes (a single frame of reference)
Memory consistency models

• Should it be impossible for both if statements to evaluate to true?

• With sequential consistency the results should always be the same provide that
  ‣ Each processor keeps its access in the order made
  ‣ We can’t say anything about the ordering across different processors: access are interleaved arbitrarily

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=0</td>
<td>B=0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>A=1</td>
<td>B=1</td>
</tr>
<tr>
<td>if (B==0)</td>
<td>if (A==0)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Rules

• **3 rules** that mostly concern when values must be transferred between main memory and per-thread memory

• **Atomicity.** Which instructions must have indivisible effects? Only concerned with instance and static variables, including array elements, but **not** local variables inside methods

• **Visibility.** Under what conditions the effects of one thread are visible to another? The effects of interest are: writes to variables, as seen via reads of those variables

• **Ordering.** Under what conditions the effects of operations can appear out of order to any given thread? In particular, reads and writes associated with sequences of assignment statements

• All changes made in one synchronized variable or code block are atomic and visible with respect to other synchronized variables and blocks employing the same lock, and processing of synchronized methods or blocks within any given thread is in program-specified order
Data races

- We say that a program allows a data race on a particular set of inputs if there is a sequentially consistent execution, i.e. an interleaving of operations of the individual threads, in which two conflicting operations can be executed “simultaneously” (Boehm).
- We’ll say that operations can be executed “simultaneously”, if they occur next to each other in the interleaving, and correspond to different threads.
- We can guarantee sequential consistency only when the program avoids data races.
- This program has a data race (x = = y = = 0 initially)

<table>
<thead>
<tr>
<th>Execution 3</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>x = 1;</td>
<td>y = 1;</td>
</tr>
<tr>
<td>r1 = y;</td>
<td>r1 = y;</td>
<td>r2 = x;</td>
</tr>
<tr>
<td>y = 1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2 = x;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>// r1 = 1 ∧ r2 == 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Under the hood of a data race

• Assume $x$ is initially 0
  $x = x + 1$

• Generated assembly code
  - $r1 \leftarrow (x)$
  - $r1 \leftarrow r1 + #1$
  - $r1 \rightarrow (x)$

• Possible interleaving with two threads
  \begin{align*}
  P1: & \quad r1 \leftarrow x \\
  P2: & \quad r1 \leftarrow x \\
  & \quad r1 \leftarrow r1 + #1 \\
  & \quad r1 \leftarrow r1 + #1 \\
  & \quad x \leftarrow r1 \\
  & \quad x \leftarrow r1 \\
  \end{align*}

  $r1(P1)$ gets 0
  $r2(P2)$ also gets 0
  $r1(P1)$ set to 1
  $r1(P1)$ set to 1
  P1 writes its R1
  P2 writes its R1
“Happens-before”

• An important fundamental concept in understanding the memory model

• Run on two separate threads, with counter = 0
  A: counter++;
  B: prints out counter

• Even if B occurs after A, no guarantee that B will see 0 …
  … unless we establish happens-before relationship between these two statements

• What guarantee is made by a happens-before relationship?
  A guarantee that memory writes by one specific statement visible to another specific statement

• Different ways of accomplishing this in C++: synchronization, atomics, variables, thread creation and completion
The C++11 Memory model

- The C++11 memory model makes minimal guarantees about semantics of memory access, bounding effects of optimizations on execution semantics.
- Special mechanisms are needed to guarantee that communication happens between threads, that establish the “happens before” relationship.
- Memory writes made by one thread can become visible, but no guarantee.
- Without explicit communication, you can’t guarantee which writes get seen by other threads, or even the order in which they get seen.
- C++ atomic variables (and the Java volatile modifier) are a special mechanism guaranteeing that communication happens between threads.
- When one thread writes to a synchronization variable, and another thread sees that write, the first thread is telling the second about all of the contents of memory up until it performed the write to that variable.

Ready is a synchronization variable.
In C++ we use load and store member functions.

All the memory contents seen by T1, before it wrote to ready, must be visible to T2, after it reads the value true for ready.

Avoiding Data races

• C++ and Java provide synchronization variables to communicate between threads, and are intended to be accessed concurrently.
• Such concurrent accesses are not considered data races.
• Thus, sequential consistency is guaranteed so long as the only conflicting concurrent accesses are to synchronization variables.
• Any write to a synchronization variable establishes a happens-before relationship with subsequent reads of that same variable.
• Declaring a variable as a synchronization variable:
  ‣ ensures that the variable is accessed indivisibly
  ‣ prevents both the compiler and the hardware from reordering memory accesses in ways that are visible to the program.
• In C++ we have various kinds of synchronization variables, including
  ‣ The atomic types
  ‣ Mutexes
Using synchronization variables to ensure sequentially consistent execution

- This program is free from data races
- There cannot be an interleaving of the steps in which the actions \( x = 42 \) and \( r1 = x \) are adjacent
- Sequentially consistent execution is guaranteed: \( r1 = 42 \)
- The implementation of atomic ensures that
  - thread 1’s assignments to \( x \) and \( x\_init \) become visible to other threads in order
  - The assignment \( r1 = x \) operation in thread 2 cannot start until we have seen \( x\_init \) set.
- The practical significance...
  - The compiler must obey extra constraints and generate special code …
  - to prevent potential hardware optimizations, such as thread 1 making the new value of \( x\_init \) available before that of \( x \) because it happened to be faster to access \( x\_init \)’s memory

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_bool x_ready; int x;</td>
<td>atomic_bool x_ready; int x;</td>
</tr>
<tr>
<td>( x = 42 );</td>
<td>while (!x_ready) {}</td>
</tr>
<tr>
<td>x_ready = true;</td>
<td>r1 = x;</td>
</tr>
</tbody>
</table>
Visibility

- Changes to variables made by one thread are guaranteed to be visible to other threads only under certain conditions.

```java
atomic<bool> ready = false;
int answer = 0
```

All the memory contents seen by T1, before it wrote to `ready`, must be visible to T2, after it reads the value true for `ready`.

Visibility

• Changes to fields made by one thread are guaranteed to be visible to other threads only under the following conditions
• A writing thread *releases* a synchronization lock and a reading thread subsequently *acquires* that same
  ‣ Release flushes all writes from the thread’s working memory, acquire forces a (re)load of the values of accessible variables
  ‣ While lock actions provide exclusion only for the operations performed within a synchronized block, the memory effects cover all variables used by the thread performing the action
• If a variable is declared as *atomic*
  ‣ Any value written to it is flushed and made visible by the writer thread before the writer thread performs any further memory operation.
  ‣ Readers must reload the values of volatile fields upon each access.
• As a thread terminates, all written variables are flushed to main memory. Thus *join*, guarantees visibility of all thread’s writes

```java
Lock;
x++;
Unlock;
```
Acquire and release

- Why can the program tolerate non-atomic reads and writes? (Listing 5.2, Williams, p. 120)
- How are the happens-before relationships established?

```cpp
std::vector<int> data;
std::atomic<bool> data_ready(false);
void reader_thread() {
    while(!data_ready.load())
        std::this_thread::sleep(std::millisseconds(1));
    std::cout<<"The answer="<< data[0]<<"\n";
}
void writer_thread() {
    data.push_back(42);
    data_ready=true;
}
```
Acquire and release with atomics

- What can do wrong in these thread functions if there is no synchronization?

```cpp
data = 42;
rdyFlag.store(true);
While (!rdyFlag.load()) {
  ...
}
cout << data << endl;
```
The need for synchronization

• What can do wrong in these thread functions if there is no synchronization?
Where is the missing synchronization?

```c
int64_t global_sum = 0;

void sumIt(int TID) {
    mtx.lock();
    sum += (TID+1);
    mtx.unlock();
    if (TID == 0)
        cout << "Sum of 1 : " << NT << " = " << sum << endl;
}
```

% ./sumIt 5
# threads: 5
The sum of 1 to 5 is 1
After join returns, the sum of 1 to 5 is: 15
Digital Image Representation

![RGB representation](image.png)

Ryan Cuprak

©2013  Scott B. Baden / CSE 160 / Fall 2013
Image smoothing algorithm

- Repeat as many times as necessary

\[
\text{for } (i,j) \text{ in } 0:N-1 \times 0:N-1 \\
I_{\text{new}}[i,j] = \left( I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i,j+1] \right)/4 \\
I = I_{\text{new}}
\]

Original 100 iter 1000 iter
Multithreaded Smoother()

Global Change, I[:, :], I^{new}[:, :]
Local mymin = 1 + ($TID \times n/$NT),
        mymax = mymin + n/$NT - 1;
Local done = FALSE;

while (!done) do
    Local myChange = 0;
    Change = 0;
    update I^{new} and myChange:
        Change += myChange;
        if (Change < Tolerance) done = TRUE;
    end for
    Swap pointers: I \leftrightarrow I^{new}
end while

update I^{new} and myChange:
for i = mymin to mymax do
    for j = 1 to n do
        \( I^{new}[i,j] = \ldots \\
        \text{myChange} += (I^{new}[i,j] - I[i,j])^2 \\
    \) end for
end for

©2013 Scott B. Baden / CSE 160 / Fall 2013
Correctness

Global Change, I[:, :], I\text{new}[:, :]
Local mymin = 1 + ($TID \times n/$NT),
    mymax = mymin + n/$NT - 1;
Local done = FALSE;
while (!done) do
    Local myChange = 0;
    BARRIER
    Only on thread 0: Change = 0; // PRODUCE
    BARRIER
    update I\text{new} and myChange
    CRITICAL SEC: Change += myChange // PRODUCE + CONSUME
    BARRIER
    if (Change < Tolerance) done = TRUE; // CONSUMER
    Only on thread 0: Swap pointers: I \leftrightarrow I\text{new}
end while

Does this code use minimal synchronization?
Case study: Compute an Image’s Histogram

- Count the frequency of each possible pixel value in the Red, Green and Blue Channels
Implementation

• Represent histogram as a 1D array, image partitioned into strips
• Each thread computes the histogram on its part of the image, and then combine within a critical section
• In some cases we can avoid a barrier

```
for (int i = 0; i < N; ++i)
    for (int j = my_min; j < my_max; ++j)
        histoL[pixel[i][j]];
mutex_sum.lock();
for (int p=0; p<256; p++)
    histo[p] += histoL[TID][p];
mutex_sum.unlock();
```
Synchronization in Perspective

• Memory consistency and cache coherence are necessary but not sufficient conditions for ensuring program correctness

• We need to take steps to avoid race conditions through appropriate program synchronization
  ‣ Critical sections
  ‣ Barriers
  ‣ Atomics
Workload Decomposition

• Block vs. Cyclic
• Static vs. Dynamic Decomposition

[Block, *]

[Block, Block]

[Cyclic, *]

[Cyclic(2), Cyclic(2)]

©2013 Scott B. Baden / CSE 160 / Fall 2013
Tradeoffs in choosing the chunk size

- **CHUNK=1**: each box needs data from all neighbors
  - Every processor loads all neighbor data into its cache!
  - Compare with [BLOCK,BLOCK]
- **CHUNK=2**: each box in a chunk of 4 boxes needs \( \frac{1}{4} \) of the data from 3 neighboring chunks
  - Each processor loads 3 chunks of neighbor data into cache
- **CHUNK=4**: Only edge boxes in a chunk need neighbor data, 20 boxes: processor loads 1.25 chunks of neighbor data
Performance Terms and concepts

- Know the definition and significance of ....
- Parallel speedup and efficiency, super-linear speedup, strong scaling, weak scaling
- Amdahl’s law, Gustafson’s law, serial bottlenecks
- Strong and Weak Scaling
Performance questions

• You observe the following running times for a parallel program running a fixed workload \( N \)
• Assume that the only losses are due to serial sections
• What is the speedup and efficiency on 8 processors?
• What will the running time be on 4 processors?
• What is the maximum possible speedup on an infinite number of processors?
• What fraction of the total running time on 1 processor corresponds to the serial section?
• What fraction of the total running time on 2 processors corresponds to the serial section?

<table>
<thead>
<tr>
<th>NT</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000</td>
</tr>
<tr>
<td>2</td>
<td>6000</td>
</tr>
<tr>
<td>8</td>
<td>3000</td>
</tr>
</tbody>
</table>
Time constrained scaling

- Sum $N$ numbers on $P$ processors
- Let $N \gg P$
- Determine the largest problem that can be solved in time $T=10^4$ time units on 512 processors
- Let time to perform one addition $= 1$ time unit
- Let $\beta =$ time to add a value inside a critical section
Performance model

• Local additions: $N/P - 1$
• Reduction: $\beta (\lg P - 1)$
• Since $N >> P$
  \[ T(N,P) \sim (N/P) + \beta (\lg P - 1) \]
• Determine the largest problem that can be solved in time $T = 10^4$ time units on $P=512$ processors, $\beta = 1000$ time units
• Constraint: $T(512,N) \leq 10^4$
  \[ \Rightarrow (N/512) + 1000 (\lg 512 - 1) = (N/512) + 1000*(8) \leq 10^4 \]
  \[ \Rightarrow N \leq 1\times10^6 \text{ (approximately)} \]
Data parallelism

- We divide up the data, and the loops that operate on them.
- Can we parallelize the inner loops as shown?

**LOOP #1**

```plaintext
for j = 0 to n-1
    for i = 0 to n-1
        A[i, j+1] = A[i, j];
    for i = 0 to n-1
        A[i, 1:n] = A[i,0:n-1];
```

**LOOP #2**

```plaintext
for j = 0 to n-1
    for i = 0 to n-1
        A[i, j+1] = A[i, j];
    for j = 0 to n-1
        A[0:n-1, j+1] = A[0:n-1, j];
```
OpenMP’s implementation of data parallelism over loops

```
cout << "Serial\n";
N = 1000;
#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];
M = 500;
#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];
Cout << "Finish\n";
```
False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache
Eliminating false sharing

- Put each counter in its own cache line

```c
static int counts[];
for (int k = 0; k < reps; k++)
    for (int r = first; r <= last; ++r)
        if ((values[r] % 2) == 1)
            counts[TID]++;
```

```c
static int counts[][LINE_SIZE];
for (int k = 0; k < reps; k++)
    for (int r = first; r <= last; ++r)
        if ((values[r] % 2) == 1)
            counts[TID][0]++;
```

<table>
<thead>
<tr>
<th></th>
<th>NT=1</th>
<th>NT=2</th>
<th>NT=4</th>
<th>NT=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unoptimized</td>
<td>4.7 sec</td>
<td>6.3</td>
<td>7.9</td>
<td>10.4</td>
</tr>
<tr>
<td>Optimized</td>
<td>4.7</td>
<td>5.3</td>
<td>1.2</td>
<td>1.3</td>
</tr>
</tbody>
</table>
Load Balancing

- We have an int array of length 1024, and share the computation evenly among the processors. The workload consists of updating the array.
- All the updates take the same amount of time, 1 second.

a. What is the running time on 4 processors?
b. What is the running time on 5 processors?
c. What is the running time on 16 processors?
Today’s lecture

• Technology
• Programming
  ‣ Threads
    ‣ Message Passing
• Algorithms (applications)
• List of Keywords and Topics (cross cutting)
Message passing in perspective

• Benefits
  ‣ Processes communicate explicitly, no anonymous updates
  ‣ Message arrival provides synchronization

• Pitfalls
  ‣ Must replace synchronization with explicit data motion (±)
  ‣ Can’t rely on the cache to move data between processes living on separate processing nodes
  ‣ Ghost cells, collectives
  ‣ Harder to incrementally parallelize code than with threads
Message passing interface

- Collectives vs. point to point
- Message filtering
- Causality
- Immediate mode
- When is it safe to use a buffer after a send/recv, iSend/iRecv
Messaging system design and implementation

- Short vs long messages, half power point
- Eager vs. rendezvous communication
- Hypercube algorithms for collectives
- Message buffering, completion
Message passing correctness issues

- You run a program on two processes that uses immediate sends `ISend()`. Assume that $X$, $Y$, $U$, $V$ are long arrays all of the same length.

The table shows the operations:

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ISend(X, P2)</code></td>
<td><code>Recv(U)</code></td>
</tr>
<tr>
<td><code>ISend(Y, P2)</code></td>
<td></td>
</tr>
</tbody>
</table>
A Classic message passing implementation of a stencil method

- Decompose domain into sub-regions, one per process
  - Transmit halo regions between processes
  - Compute inner region after communication completes
- Loop carried dependences impose a strict ordering on communication and computation
Ghost cells

- **Image smoother**
  
  repeat until done:
  
  for \( i = 1: N-2 \)
  
  for \( j = 1: N-2 \)
  
  \[ U_{new}[i,j] = (U[i+1,j] + U[i-1,j] + U[i,j+1] + U[i,j-1])/4; \]
  
  *Swap* \( U_{new} \) and \( U \)

- What is the max number of ghost cells that a processor will need to transmit when \( P \) does not divides \( N \) evenly?

  \[ N/\sqrt(P) \]
\( \alpha - \beta \) model of communication time

- Triton.sdsc.edu

\( \alpha = 3.2 \mu \text{sec} \)

\( \beta_\infty = 1.12 \text{ GB/sec} \)

@\( N = 8\text{ MB} \)

\( N_{1/2} \approx 20 \text{ KB} \)
Communication costs - stencil method

- 1 word = double precision floating point = 8 bytes
- 1-D decomposition
  \[2(\alpha+8\beta N)\]
- 2-D decomposition
  \[4(\alpha+8\beta N/\sqrt{P})\]
Parallel speedup and efficiency

- Computation costs the same with 1-D and 2-D decomposition: $16N^2\beta$

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

• 1-D decomposition
• Let’s plot $E_P$ as a function of $N$, varying $P$ as a parameter
  \[ E_P = \frac{1}{1 + \frac{(\alpha + 8\beta N)P}{(8N^2\beta)}} \]
• Let’s also plot the fraction of time spent communicating
Parallel efficiency

N = 1024

N = 128
Communication fraction

![Communication fraction graph](image)

- N = 128
- N = 1024
Hypercubes

- Used to express collectives
- Diameter, bisection bandwidth
- Map a 1D ring onto a hypercube
- How many parallel/unique paths between any two nodes in a hypercube?
Hypercube broadcast algorithm with $p=4$

- Processor 0 is the root, sends its data to its hypercube “buddy” on processor 2 (10)
- Proc 0 & 2 send data to respective buddies
Scatter/Gather

\[ \text{Gather} \quad \rightarrow \quad \text{Scatter} \]

\[ P_0 \quad P_1 \quad P_{p-1} \]

Root

Short message algorithm

©2013 Scott B. Baden / CSE 160 / Fall 2013
Fin