CSEP 524 – Parallel Computation
University of Washington

Lecture 1: Motivation; Administrative; Introduction

Michael Ringenburg
Spring 2015
What is parallelism?
What is parallelism?

- **Parallelism**: Using multiple resources to complete a task
  - E.g., the cashier to collect $$ and the barista to make coffee
  - Or, multiple gardeners, or multiple instructors, or multiple programmers, etc...
What is parallelism?

• **Parallelism**: Using multiple resources to complete a task
  – E.g., the cashier to collect $$ and the barista to make coffee
  – Or, multiple gardeners, or multiple instructors, or multiple programmers, etc...

• **Key question**: How do you divide the work?
  – Each gardener working on a separate patch?
  – Each worker handling a separate piece of the pipeline (e.g., cashier and barista)?
What is parallelism?

- **Parallelism:** Using multiple resources to complete a task
  - E.g., the cashier to collect $$ and the barista to make coffee
  - Or, multiple gardeners, or multiple instructors, or multiple programmers, etc...

- **Key question:** How do you divide the work?
  - Each gardener working on a separate patch?
  - Each worker handling a separate piece of the pipeline (e.g., cashier and barista)?

- Why would we do this?
  - Complete work faster
  - Complete task that is infeasible for single worker
What is parallel computing?
What is parallel computing?

• **Parallel Computing**: Using multiple *compute* resources to complete a task
  – Typically, processors and their memory
  – May include accelerators: GPUs, FPGAs, etc…
What is *parallel computing*?

- **Parallel Computing**: Using multiple *compute* resources to complete a task
  - Typically, processors and their memory
  - May include accelerators: GPUs, FPGAs, etc...
- **Key question**: How do you divide the work?
  - *Data parallelism*: Divide the data across processors, compute the same task on each (like the gardeners)
  - *Task parallelism*: Execute separate tasks on each processor (like the cashier and the barista) on the same or different data
What is parallel computing?

• **Parallel Computing**: Using multiple *compute* resources to complete a task
  – Typically, processors and their memory
  – May include accelerators: GPUs, FPGAs, etc...

• **Key question**: How do you divide the work?
  – *Data parallelism*: Divide the data across processors, compute the same task on each (like the gardeners)
  – *Task parallelism*: Execute separate tasks on each processor (like the cashier and the barista) on the same *or* different data

• Why would we do this?
  – Complete *a computation* faster
  – Complete *computation* that is infeasible for *one processor*
Parallel Computations Vary in Difficulty

Matrix Addition: Quite straightforward

\[
\begin{array}{c}
\text{Green} \\
\end{array}
\begin{array}{c}
\text{Blue} \\
\end{array}
\begin{array}{c}
\text{Yellow} \\
\end{array}
\end{array}
= \\
\begin{array}{c}
\text{Addition} \\
\end{array}
\begin{array}{c}
\text{Green} \\
\end{array}
\begin{array}{c}
\text{Blue} \\
\end{array}
\begin{array}{c}
\text{Yellow} \\
\end{array}
\end{array}
Parallel Computations Vary in Difficulty

Matrix Multiplication: Far more involved
Two Key Concerns

• **Parallelism:** “What should execute simultaneously?”
  – without parallelism, no speedup

• **Locality:** “Where should things execute?”
  – Minimize time spent sending, waiting for data
  – Necessary for top performance
Why study parallel computing?
The Traditional Answer(s)

- It is a fundamental departure from the “normal” computer model, therefore it is inherently cool/interesting
- Deep intellectual challenges for CS -- models, programming languages, algorithms, HW, ...
- HPC/Supercomputing: The extra power from parallel computers is very useful in science, engineering, business, ...
My Employer: CRAY
THE SUPERCOMPUTER COMPANY

CSE341: Programming Languages
Lecture 1
Course Mechanics
ML Variable Bindings
Dan Grossman
Fall 2011

Spring 2015
UW CSEP 524 (PMP Parallel Computation):
Ringenburg

14
For more information: [http://www.olcf.ornl.gov/titan/](http://www.olcf.ornl.gov/titan/)
The New Answer(s)

• Why does this matter to non-HPC/supercomputing developers?
  – The “multicore revolution” – everything is a parallel computer now
    • Desktops
    • Laptops
    • Even telephones!
  – Big Data Analytics
    • Large data sets that are too big to fit on a single machine
    • And too large to compute efficiently with a single processor
      – Many applications are time-sensitive
    • Most popular frameworks for data analytics are parallel
      – Hadoop MapReduce
      – Spark
      – Storm
      – …
Multicore Processors: How did we get here?

- Transistor density has continued following Moore’s Law
  - But, see the caveat in this week’s second reading ...
- But clock speeds have mostly stopped increasing
  - Physical limitations: heat, power, leakage
- So what do we do with the extra transistors? How do we provide the performance boosts we’re used to?
  - Answer: Add parallelism

This Course
About me

• UW CSE PhD alum – graduated early last year
  – Researched architectures and programming models for Approximate Computing (reducing energy consumption by relaxing accuracy/precision guarantees)

• Worked at Cray since 2006
  – Part-time in 2012-13 while finishing PhD
  – ~7 years working on an automatically parallelizing compiler
    • Take non-parallel C/C++ code, plus (optional) pragmas, convert to a parallel program via automatic loop parallelization
  – More recently: working on parallel Big Data Analytics
    • Important new application of parallel computing
    • Will have a lecture on this towards the end of the course
What am I doing here? ;-)

• Give something back to the department
• Enjoy teaching, meeting students
  – First time teaching this format of class, so bear with me.
• Parallel computing is a broad, fascinating, ever-changing subject – always more to learn
  – I hope to learn as much from you as you learn from me!
Overall Course Goals

• Expose you to as much information about parallel computing as possible within the (short) timeframe
  – foundations
  – best practices
  – recent trends

• Teach you principles of parallel programming

• Give you the background needed to read the state-of-the-art research in the field
  – Will gain practice through reading/reviewing research papers in homeworks, discussing in class
  – Final project will give you practice going in depth on a specific topic
Class Sessions

- Don’t worry, I won’t lecture for three hours straight...
  - You would fall asleep; I would lose my voice
- Class will generally start with a lecture (about 1.5 hours, with short break)
- Then a break
- Then a discussion of the readings
  - Discussion session is for you to discuss/debate *(politely)* the papers and related topics
  - I am just here to moderate/keep things on track
  - So, please be prepared: do the readings and the homework on time
  - Otherwise discussions will not be valuable
  - Discussion participation *is* part of your grade for the course
  - **Note:** This is my first time with a distance course, but I will work to make sure both classrooms are able to participate in discussions.
- Today’s discussion will be short, since the first readings aren’t due until next week
  - Introduce yourselves, why you are here, etc.
Your Work

• **Assignments:**
  – Most weeks will include 1-2 articles/research papers to read and review
  – May also include a couple short written and/or programming problems

• **Review format:**
  – 0.5 - 1 pages (using a “reasonable” font size)
  – Include:
    • Summary of articles key points
    • Do you agree/disagree? Why? ⇐ **Important**
    • 2-3 discussion questions related to the article(s)

• **Late policy:** At most twice during the quarter, you may turn in an assignment late (max 1 week). This is intended for use with work/family emergencies – don’t abuse.
Your Work, cont.

• **End-of-term project:**
  – Learn about and report on some technology we didn’t cover
    • Or go in significantly more depth on a topic we did cover
  – Will include written report and oral presentation (last 2 days of class)
    • Sign-ups available soon
    • East-side students may come to Seattle campus to present (recommended, but not required)
  – May include programming component, but not required
  – Grading will be based on both content *and* delivery
  – More details available soon on course web
  – Homeworks may include project “checkpoints”

• **Grading breakdown (tentative):**
  – Project: 100 points
  – Homework: 60-80 points *total* (about 10 points each)
  – Class/discussion participation: ~40 points
Nuts and Bolts

• **TA:** Amnon Horowitz, amnonh@cs
• **Text:** Lin & Snyder, *Principles of Parallel Programming* (2nd edition)
  – Meant as supplementary material to lecture – read at your leisure, but note that homeworks may rely on it.
• **Office Hours:**
  – Difficult with a distance course, and with all of us having day jobs
  – I will be in my office (CSE 278) before class, starting at roughly 5:30 Tuesdays
  – Amnon office hours: TBD – let us know your thoughts
• **Webpage:**
  – Discussion boards, slides (after class), homeworks, dropbox, project info, etc.
• **Guest lecture on April 28:** Brad Chamberlain
  – Taught this course two years ago
  – Technical lead for Cray’s Chapel parallel programming language
  – I will be at a conference, but your attendance is still expected – there will be homework related to the lecture
Introduction to Parallel Computing
Rest of this Lecture

• Goal: To give a general idea of the challenges of parallel computation
  – Examine a few problems
  – Think about how to make them parallel tasks
  – Understand some of the challenges, e.g., locality and caching

• Motivate future lectures!
First, the dream …

• Since 70s (Illiac IV days) the dream has been to automatically compile sequential programs into parallel programs
  – Decades of research by academy and industry implies it’s hopeless for general computations
  – But didn’t your instructor work on exactly that?!?
    • For individual loops, it is possible (sometimes with semantic help from programmer)
    • For complete applications/algorithms it has proved extremely difficult to efficiently parallelize
    • MTA/XMT programmers would come up with a parallel algorithm, rely on out compiler to deliver fine-grained loop parallelism within the algorithm
What’s the Problem?

• Compilers are good at local optimizations (including parallelization and vectorization)
  – C/C++ aliasing makes this harder, but user pragmas/type qualifiers can solve

• But, for most algorithms, a “best” sequential solution and a “best” parallel solution are usually fundamentally different.
  – Different solution paradigms imply good parallelization is not a local optimization.

Therefore... the programmer must discover the || solution!
Consider a simple task

- Adding sequence of numbers $A[0], \ldots, A[n-1]$
- Standard way to express it

```c
sum = 0;
for (i=0; i<n; i++) {
    sum += A[i];
}
```

- Language semantics require we execute as:
  - $\ldots((\text{sum}+A[0])+A[1])+\ldots)+A[n-1]$
  - That is, **sequential**
- Can we execute this in parallel?
Parallel Summation

• To sum a sequence in parallel
  – add pairs of values producing 1st level results,
  – sum pairs of 1st level results producing 2nd level results,
  – sum pairs of 2nd level results producing 3rd level results,
  – etc.

• E.g., replace:

\[
\]

• With:

\[
\]
Express the Two Formulations

- Graphic representation makes difference clear

- Same number of operations; different order allows parallelism
Express the Two Formulations

- Graphic representation makes difference clear

- Same number of operations; different order allows parallelism
Express the Two Formulations

- Graphic representation makes difference clear

- In class exercise: sketch what happens when \#summands > cores/2
Our Goals In Parallel Programming

• Goal: Scalable programs with performance and portability
  – Scalable: More processors can be “usefully” added to solve the problem faster
  – Portability: The solutions run well on all parallel platforms
  – Performance: Programs run as fast as those produced by experienced parallel programmers for the specific machine

• Not always possible to achieve both performance and portability, due to architectural differences, but a good goal.
Scaling a Parallel Sum

• **Exercise part 2:** Compute performance of your generalized parallel sum:
  – Start with $N = 1024$, and $P = 4$
  – Assume sending a small message takes 30 ticks
  – And loading, adding and storing a result takes a total of 3 ticks (cached array, unrolled loop).

• What if we scale to $P = 16$?
• How about $P = 64$?
• Now, repeat with $N = 1,048,576$ ($2^{20}$)
Scaling a Parallel Sum

• **Exercise part 2:** Compute performance of your generalized parallel sum:
  – Start with $N = 1024$, and $P = 4$
  – Assume sending a small message takes 30 ticks
  – And loading, adding and storing a result takes a total of 3 ticks (cached array, unrolled loop).

• What if we scale to $P = 16$?
• How about $P = 64$?
• Now, repeat with $N = 1,048,576 (2^{20})$

**Key takeaway:** Scalability depends on problem size
A Related Computation

- Consider computing the prefix sums of an array

```c
for (i=1; i<n; i++) {
    A[i] += A[i-1];
}
```

- Semantics ...
  - A[0] is unchanged
    ...

How can we compute this in parallel?
Comparison of Paradigms

- The sequential solution computes the prefixes … the parallel solution computes only the last value

- Or does it?
Comparison of Paradigms

• The sequential solution computes the prefixes … the parallel solution computes only the last value

Notice that the left input to a node is the sum of all elements in its left (first) subtree.

• Or does it?
Comparison of Paradigms

- The sequential solution computes the prefix for the entire input, while the parallel solution computes only the prefix for the rightmost element.

Notice that the left input to a node is the sum of all elements in its left (first) subtree.

Can we use this to help compute the prefixes for the right?

- Or does it?
Parallel Prefix Algorithm

Compute sum going up
Figure prefixes going down

Invariant: Parent data is sum of elements to left of subtree

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>4</th>
<th>16</th>
<th>10</th>
<th>16</th>
<th>14</th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parallel Prefix Algorithm

Compute sum going up
Figure prefixes going down

Invariant: Parent data is sum of elements to left of subtree
Fundamental Tool of Parallel Programming

• Original research on parallel prefix algorithm published by
  Richard E. Ladner and Michael J. Fischer
  Parallel Prefix Computation

  The Ladner-Fischer algorithm requires $2\log n$ time, twice as much as simple tournament global sum, not linear time

Applies to a wide class of operations
Parallel Compared to Sequential Programming

• Has different costs, different advantages
• Requires different, unfamiliar algorithms
• Must use different abstractions
• More complex to understand a program’s behavior
• More difficult to control the interactions of the program’s components
• Knowledge/tools/understanding more primitive
  – Although this is rapidly changing
Consider Another Simple Problem

• This time, let's consider how it runs on a real machine as well.

• First, the problem:
  – Count the 3s in array[] of n values:

```cpp
count = 0;
for (i=0; i<n; i++) {
  if (array[i] == 3) {
    count += 1;
  }
}
```
Write A Parallel Program

- Need to know something about machine... use multicore architecture

How would you solve it in parallel?

2-level cache, L2 shared
Divide Into Separate Parts

- Idea 1: assign each thread a chunk of the array to count

```
array = [2, 3, 0, 2, 3, 3, 1, 0, 0, 1, 3, 2, 2, 3, 1, 0]
length = 16  t = 4

Thread 0
Thread 1
Thread 2
Thread 3

int length_per_thread = length/t;
int start = id * length_per_thread;
for (i=start; i<start+length_per_thread; i++) {
    if (array[i] == 3)
        count += 1;
}
```
Divide Into Separate Parts

• THIS GETS THE WRONG ANSWER!
  – Any ideas why?

length=16  t=4

array

| 2 | 3 | 0 | 2 | 3 | 3 | 1 | 0 | 0 | 1 | 3 | 2 | 2 | 3 | 1 | 0 |

Thread 0  Thread 1  Thread 2  Thread 3

int length_per_thread = length/t;
int start = id * length_per_thread;
for (i=start; i<start+length_per_thread; i++) {
    if (array[i] == 3) {
        count += 1;
    }
}
Divide Into Separate Parts

• THIS GETS THE WRONG ANSWER!
  – Any ideas why?

<table>
<thead>
<tr>
<th>array</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
int length_per_thread = length/t;
int start = id * length_per_thread;
for (i=start; i<start+length_per_thread; i++) {
    if (array[i] == 3)
        count += 1;
}
```
Race conditions

- Two processes interfere on memory writes

Thread 1

- count $\leftarrow 0$
- load
- increment
- store

Thread 2

- load
- increment
- store
- count $\leftarrow 1$
- count $\leftarrow 1$
- count $\leftarrow 1$
Protect Memory References

- 2nd attempt: Protect memory references with a mutex (mutual exclusion) lock:

```c
mutex m;
for (i=start; i<start+length_per_thread; i++) {
    if (array[i] == 3) {
        mutex_lock_acquire(m);
        count += 1;
        mutex_lock_release(m);
    }
}
```

- Only one thread may hold the lock \( m \) at any given time. Others must wait until it is released.
Correct Program!

• But look what happens to performance…

Performance

<table>
<thead>
<tr>
<th></th>
<th>serial</th>
<th>parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>t=1</td>
<td>0.91</td>
<td>5.02</td>
</tr>
<tr>
<td>t=2</td>
<td></td>
<td>6.81</td>
</tr>
</tbody>
</table>

– Performs worse than the serial version of the code!
Closer Look: Motion of count, m

- Problem 1: Threads waste time waiting on lock
- Problem 2: Contention on lock and data causes constant cache misses and invalidations!
- Problem 3: Lock operations expensive – must ensure visible to all threads

```c
mutex m;
for (i=start; i<start+length_per_thread; i++){
    if (array[i] == 3) {
        mutex_lock(m);
        count += 1;
        mutex_unlock(m);
    }
}
```
Accumulate Into Private Counter

• 3rd attempt: each processor adds into its own memory; combine at the end (single lock acquire/release per thread)

```c
for (i=start; i<start+length_per_thread; i++) {
    if (array[i] == 3) {
        private_count[t] += 1;
    }
}
mutex_lock(m);
count += private_count[t];
mutex_unlock(m);
```
Keeping Up, But Not Gaining

• Sequential and 1 processor match, but it’s a loss with 2 processors
False Sharing

- Got rid of time waiting on lock, and most of the expensive lock operations
- But, private variable ≠ private cache-line
Force Into Different Lines

- 4th attempt: padding the private variables forces them into separate cache lines and removes false sharing

```c
// Assume 64 byte cache lines
struct padded_int {
    int32 value;
    char padding[60];
} private_count[MaxThreads];
```
Success!!

- Two processors are almost twice as fast

Performance

<table>
<thead>
<tr>
<th>t=1</th>
<th>t=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.91</td>
<td>0.51</td>
</tr>
</tbody>
</table>

serial  Try 4
Count 3s Summary

• Recapping:
  – Started with obvious “break into blocks” program
  – Needed to protect the count variable
    • Prevent race conditions – repeated theme
  – Got the right answer, but the program was slower
    … lock and data contention
  – Privatized memory and 1-process was fast
    enough, 2- processes slow … false sharing
  – Separated private variables to own cache line
  – Success! 2 cores were almost twice as fast as 1
Recall the Matrix Multiplication

- Matrix multiplication of (square n x n) matrices $A$ and $B$ producing n x n result $C$ where $C_{rs} = \sum_{1\leq k\leq n} A_{rk} * B_{ks}$
Extreme Matrix Multiplication

• The multiplications are independent (do in any order) and the adds can be done in a tree

\[
\begin{align*}
O(n) \text{ processors for each result element implies } O(n^3) \text{ total for } n \times n \text{ matrix} \\
\text{Time: } O(\log n) \\
\textbf{In-class question:} \text{ How would you generalize this to work when } P < n^3? 
\end{align*}
\]
In the real world…

• Good properties
  – Extremely parallel
  – Very fast – $\log n$ is a good bound

• Bad properties
  – Ignores memory structure and reference collisions
  – Ignores data motion and communication costs
  – Work imbalance between processors – half only participate in first round.
Where is the data?

- Reference collisions and communication costs are important to final result.
- Need a model for this! One simple possibility is the PRAM (parallel RAM) model:

```
P_0  P_1  P_2  P_3  P_4  P_5  P_6  P_7
```

Memory:

```
C   A   B
```

- Spring 2015
- UW CSEP 524 (PMP Parallel Computation): Ringenburg
PRAM: Parallel Random Access Machine

• Use as many execution units (cores, threads, etc.) as you like
• All units access a single shared memory
  – Any processor can reference any memory location in \textit{unit time}
• How do we resolve memory collisions?
  – Read Collisions -- simultaneous reads to location are OK
  – Write Collisions -- simultaneous writes to location need a rule. Typical options:
    • Allowed, but must all write the same value
    • Allowed, but value from highest indexed processor wins
    • Allowed, but a random value wins
    • Prohibited
PRAM: Parallel Random Access Machine

- Use as many execution units (cores, threads, etc.) as you like
- All units access a single shared memory
  - Any processor can reference any memory location in \textit{unit time}
- How do we resolve memory collisions?
  - Read Collisions -- simultaneous reads to location are OK
  - Write Collisions -- simultaneous writes to location need a rule. Typical options:
    - Allowed, but must all write the same value
    - Allowed, but value from highest indexed processor wins
    - Allowed, but a random value wins
    - Prohibited

Is this realistic??
PRAM likes our algorithm

- Allows any # of execution units: $O(n^3)$ OK
- $A$ and $B$ matrices are read simultaneously, but that’s OK
  - Read in “unit time”
- $C$ is written simultaneously, but no location is written by more than 1 processor
  - Write in “unit time”

PRAM model implies $O(\log n)$ algorithm is good … but in real world, we suspect not
Where else could data reside?

- Cluster-like model: data split between local memories of separate processors

  - Each processor could hold blocks of $A$ and $B$, and compute block of $C$
Data Motion

- Getting rows and columns to processors

- Allocate matrices in blocks
- Ship only portion being used
Data Motion

- Getting rows and columns to processors

- Allocate matrices in blocks
- Ship only portion being used
- *Can we reuse shipped data?*

\[ \begin{array}{ccc}
P_0 & P_1 & \text{C} \\
P_2 & P_3 & \\
\end{array} \quad = \quad \begin{array}{ccc}
P_0 & P_1 & \text{A} \\
P_2 & P_3 & \\
\end{array} \quad = \quad \begin{array}{ccc}
P_0 & P_1 & \text{B} \\
P_2 & P_3 & \\
\end{array} \]
Data Motion

• Getting rows and columns to processors

- Allocate matrices in blocks
- Ship only portion being used
- Can we reuse shipped data? Yes!

C

\[
\begin{array}{c|c}
| P_0 & P_1 \\
|-----|-----|
\end{array}
\]

A

\[
\begin{array}{c|c}
| P_0 & P_1 \\
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-----</td>
<td>-----</td>
</tr>
</tbody>
</table>
\end{array}
\]

B

\[
\begin{array}{c|c}
| P_0 & P_1 \\
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-----</td>
<td>-----</td>
</tr>
</tbody>
</table>
\end{array}
\]
Blocking Improves Locality

- Reuse of rows, columns => caching effect
- Large blocks => big chunks of needed rows/columns local
What we learned

• Many factors matter when choosing/designing a parallel algorithm
  – A processor’s connection to memory
  – Number of processors available
  – Locality: always important in computing
    • But locality is often at odds with high levels of parallelism
    • Using caching is complicated by multiple threads – don’t want data “bouncing” between caches

• Need a better understanding of parallel architectures and models of parallelism!
  – Coming up next week!
Discussion

• Today will be short (we can go home early!), since you haven’t read any papers yet.

• Briefly introduce yourself:
  – Name
  – Where you work
  – What you do
  – Why you are interested in this course
  – Any other interesting facts about yourself/relevant background you bring/jokes/etc.