LAU
الجامعة اللبنانية الأميركية
Lebanese American University

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Parallel Algorithms

Instructor: Haidar M. Harmanani

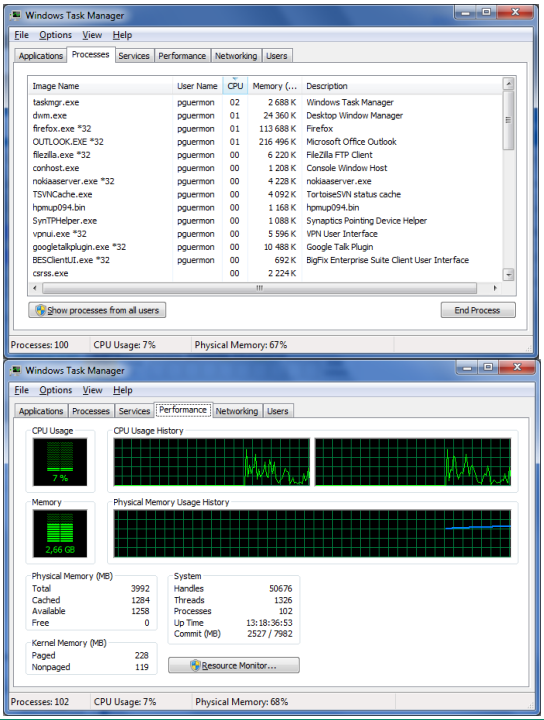
Spring 2017

Processes

- Remember you have the choice to launch several independent software at the same time to take advantage of several cores (browser, email, music player, ...).
- It's easy, the operating system is taking care of associating in real time your software with cores and memory.
- You are using processes everyday !

intel Software

Windows

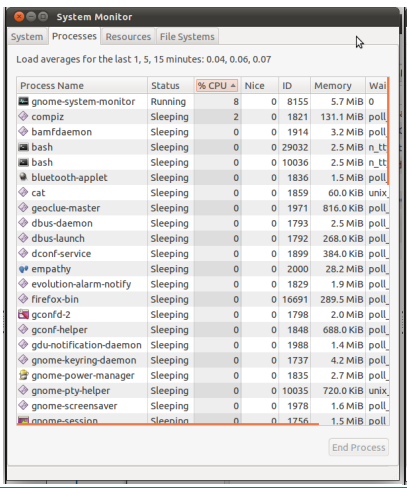


Process view

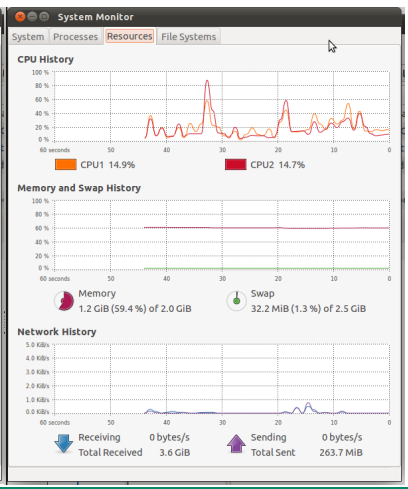
Core activity view

Linux

Process view



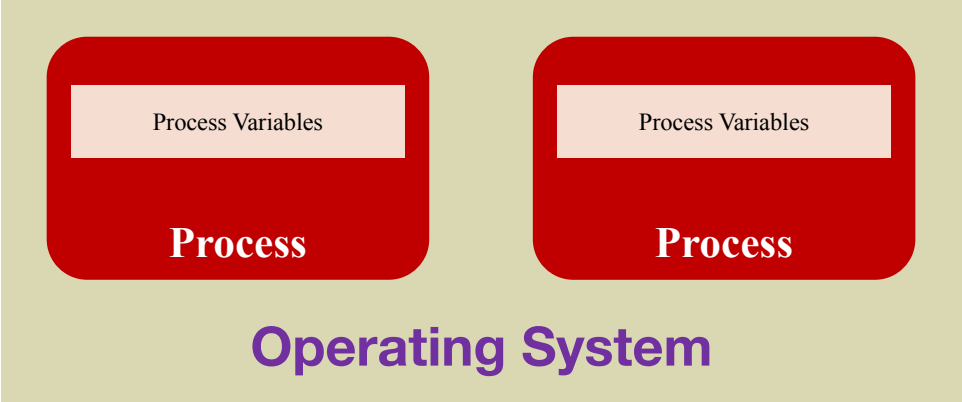
Core activity view



Processes

- Good side :
  - Each process is totally independent : it's secure.
- Bad side :
  - But each process has a dedicated memory space so you can't easily share a variable in memory or code between processes.
  - Processes can communicate with other processes on the same machine or other machines over the network, but it's slow and generates a lot of CPU overhead.

Processes Memory Model



Example : Apache Web Server

- Apache web server is typically using processes. Dozens, hundreds of processes are launched automatically to answer client requests coming from the network.
- Good side : It's simple, secure and easy to code.
- Bad side : Communication between processes is complex, but web pages served to different clients usually do not need to share information.

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Threads

Threads

- Inside processes, you have thread(s).
- When your main() function begins, that's the beginning of the first thread of your software.
- If you create more threads, the operating system will allocate them transparently with cores and memory in real time, just like processes.
- Danger : But unlike processes, threads from the same process share memory (data and code).
  They can communicate easily, but it's dangerous if you don't protect your variables correctly.

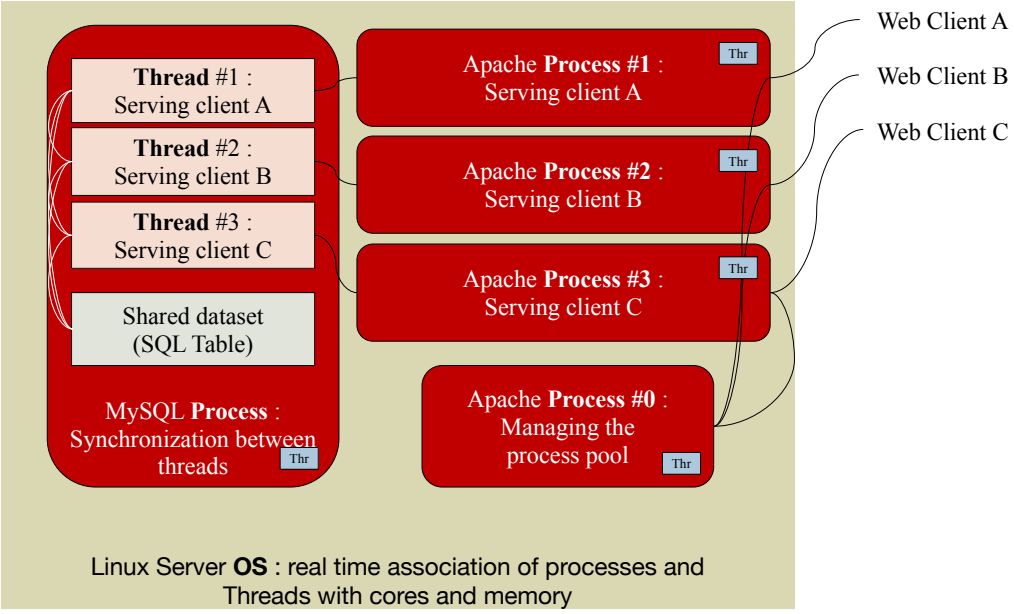
Example : MySQL Database

- MySQL Database server is launching threads to serve clients requests.
- Clients typically request information from the same dataset (SQL Table) : communication between threads is needed.
  That's why MySQL could not have easily used processes.
- MySQL developers had to take care of parallel programming risks to protect shared information. “Synchronization”.

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Threads + Processes

Example : LAMP



intel Software

Example : LAMP

- Each PHP generated page is processed from a different process : 3rd party developers coding in PHP or coding PHP itself don't need to worry about synchronization. Nothing is shared.

- MySQL is using threads to processes various SQL requests. MySQL developers had to code synchronization to protect tables from concurrent access. SQL users can also lock tables using SQL transactions.

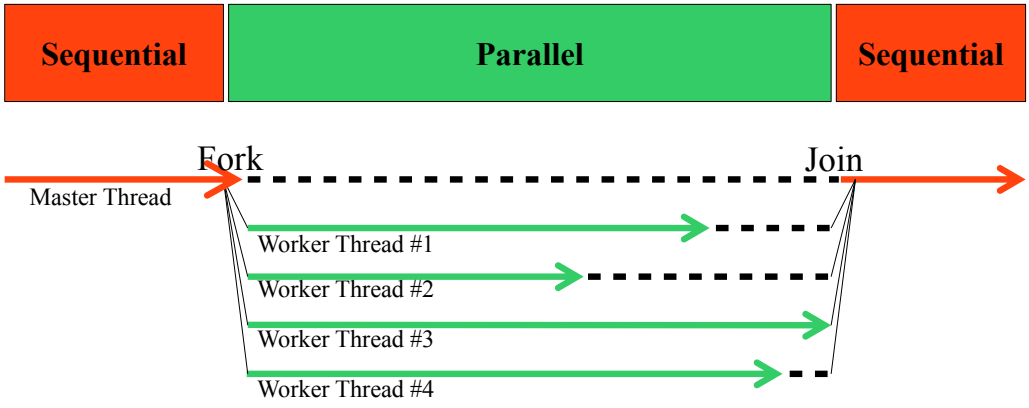
LAU Lebanese American University

Fork/Join Model

Fork/Join Model

- When your main() function begins, that's the beginning of the first thread of the software.
  - You are in serial mode!
- When you enter the parallel part of your mutithreaded software, threads are launched.
  - Master thread forks worker threads.
- When all the worker threads have finished, they join.
- You are in serial mode again.
- End of main() function, end of process.

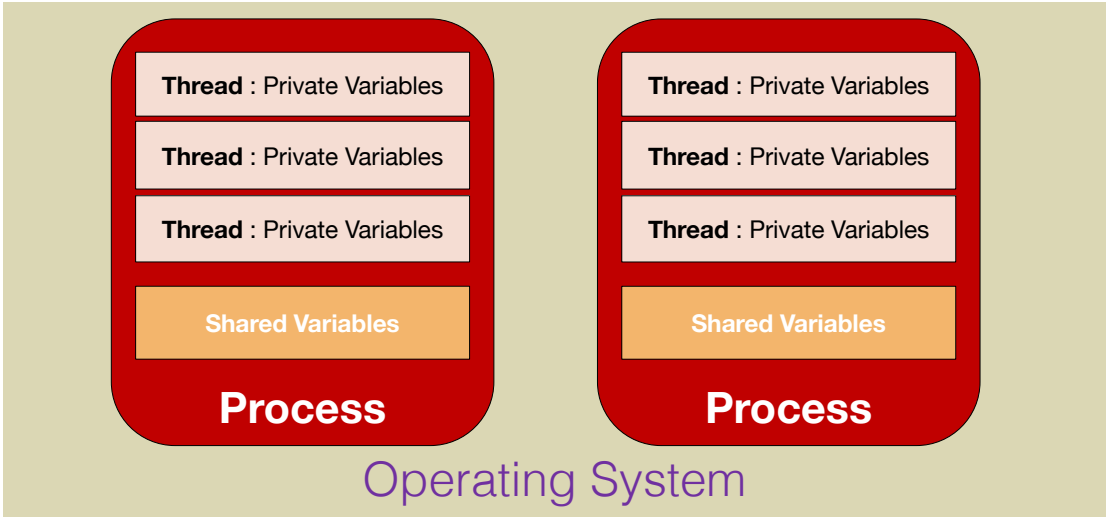
Fork/Join Model



LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Threads Memory Model

Threads Memory Model



intel Software

Sharing variables – Simple code

- Let's compute the sum of integers between 0 and n using the following loop :

```
int sum = 0;
int i, n = 3;

for (i = 1; i < n; i++) {
    sum += i;
}
printf("%d\n", sum);
```


Sharing variables – Serial run

- If you run it serially (2 iterations, one at a time) :

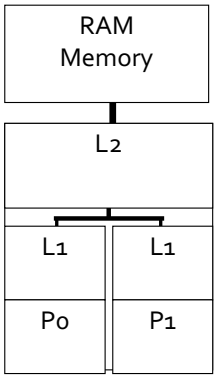
```
int n = 3;
for (i = 1; i < n; i++) {
    sum += i;
}

Starting point : sum=0
After 1st iteration : i=1 sum=1
After 2nd iteration : i=2 sum=3
Final result : sum=3
```

Write A Parallel Program

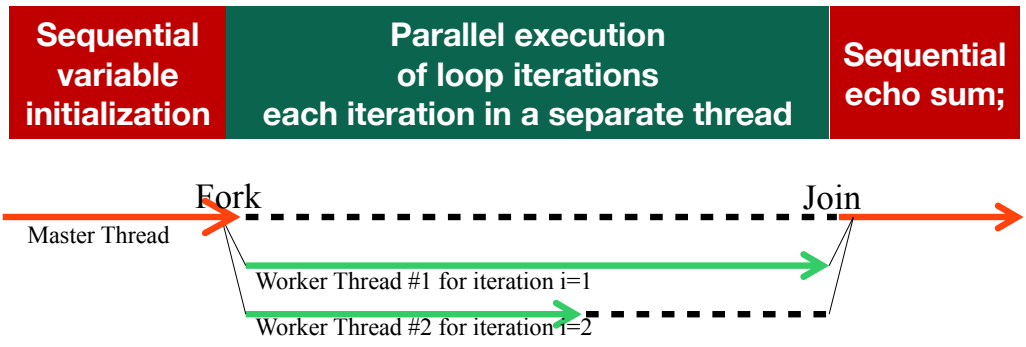
- Need to know something about machine … use multicore architecture

How would you solve it in parallel?



Sharing variables – Parallel run

- Now, let's say you find a way (let's see later how) to compute iteration i=1 and i=2 in two separate threads to save execution time.



intel Software

Sharing variables – Parallel run

- When you run it in parallel, each thread has to execute independently “sum += i;”
- It means 3 operations :
  - Read sum
  - Compute +i
  - Write sum.
- Depending on the timing of the two execution, you'll have a different result. Let's see why.

Sharing variables – Parallel run

- Case 1 :
  thread #1 and #2 read at the same time,
  thread #1 is writing first, then thread #2.

```
Starting point for thread #1 : sum=0
After thread #1 1rst iteration : i=1 sum=1
Starting point for thread #2 : sum=0
After thread #2 1rst iteration : i=2 sum=2
Final result : sum=2 (instead of 3)
```

Sharing variables – Parallel run

- Case 2 :
  thread #1 and #2 read at the same time,
  thread #2 is writing first, then thread #1.

```
Starting point for thread #2 : sum=0
After thread #2 1rst iteration : i=2 sum=2
Starting point for thread #1 : sum=0
After thread #1 1rst iteration : i=1 sum=1
Final result : sum=1 (instead of 3)
```


Sharing variables – Parallel run

- Case 3 :
  thread #1 read after thread #2 finished writing.
  (or the contrary)

```
Starting point for thread #1 : sum=0
After thread #1 1rst iteration : i=1 sum=1
Starting point for thread #2 : sum=1
After thread #2 1rst iteration : i=2 sum=3
Final result : sum=3
```

Race condition

- Launching threads is easy, and computing +i in parallel will use *multiple core*.
- But, depending on various uncontrollable conditions, your code running in parallel will return variable results.
  - It's called a *race condition*.
- Other types of typically parallel bugs are also possible.

Race condition

- Cause : sum is shared between threads.
- Solution : explain to the compiler that sum has to be protected from random parallel access.

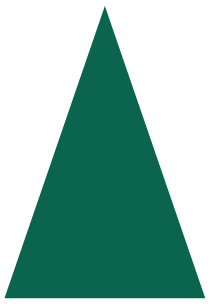
Serial bugs and parallel bugs are different.
If you are aware of the problem, solutions are easy to implement.

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

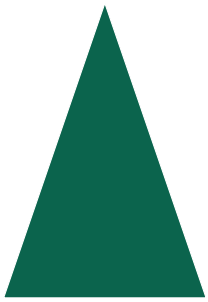
Program A Parallel Sum

- Return to problem of writing a parallel sum
- Sketch solution in class when n > P = 8
- Use a logical binary tree?



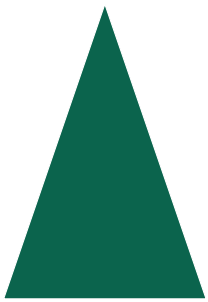
Program A Parallel Sum

- Return to problem of writing a parallel sum
- Sketch solution in class when n > P = 8
- Assume
  - Communication time = 30 ticks
  - n = 1024
- compute performance



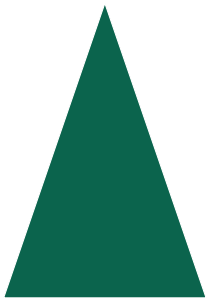
Program A Parallel Sum

- Return to problem of writing a parallel sum
- Sketch solution in class when $n > P = 8$
- Assume
  - Communication time = 30 ticks
  - $n = 1024$
- compute performance
- Now scale to 64 processors



Program A Parallel Sum

- Return to problem of writing a parallel sum
- Sketch solution in class when $n > P = 8$
- Assume
  - Communication time = 30 ticks
  - $n = 1024$
- compute performance
- Now scale to 64 processors



This analysis will become standard, intuitive

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Another Example

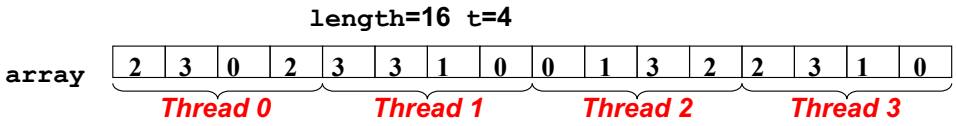
Consider a Simple Problem

- Count the 3s in `array[]` of `length` values
- Definitional solution …
  - Sequential program

```
count = 0;
for (i=0; i<length; i++)
  {
    if (array[i] == 3)
       count += 1;
  }
```


Divide Into Separate Parts

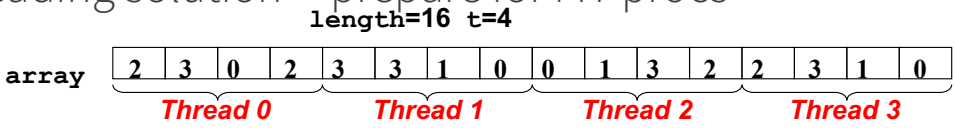
- Threading solution -- prepare for MT procs




```
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

- Threading solution -- prepare for MT procs

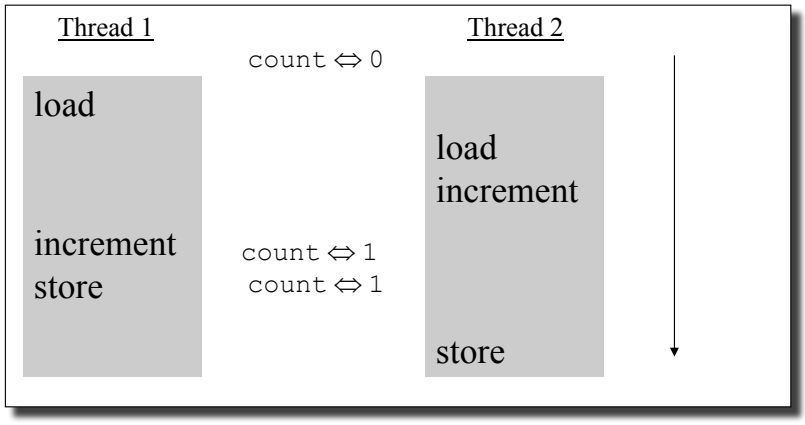



```
int length_per_thread = length/t;
int start = id * length_per_thread;
for (i=start; i<start+length_per_thread; i++)
    {
       if (array[i] == 3)
         count += 1;
    }
```

Doesn't actually get the right answer

Races

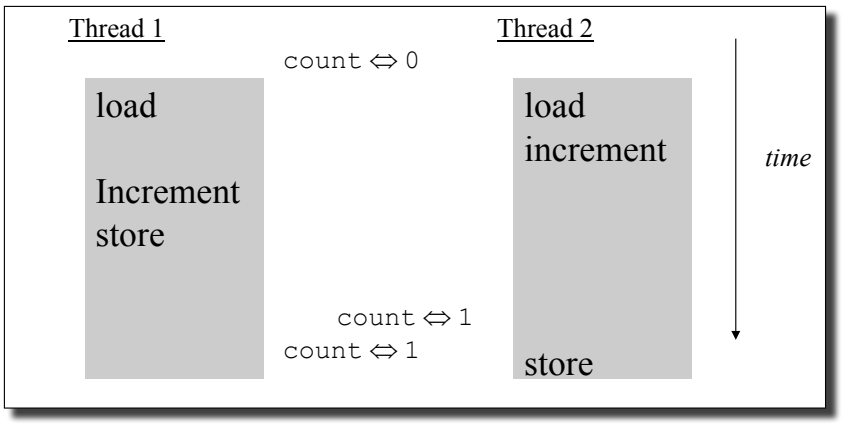
- Two processes interfere on memory writes



Races

- Two processes interfere on memory writes

Try 1



Protect Memory References

- Protect Memory References

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

Protect Memory References

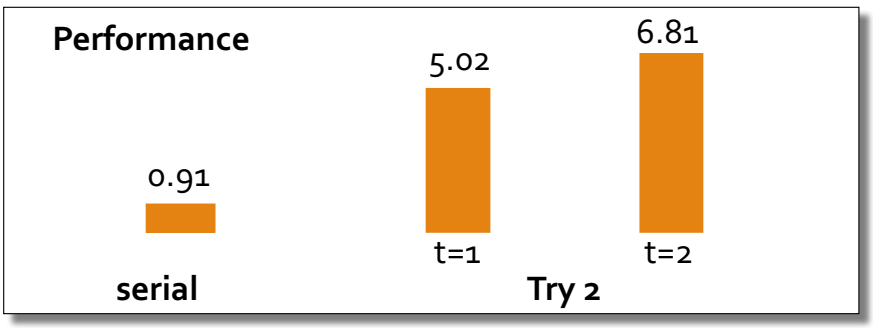
- Protect Memory References

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

Try 2

Correct Program Runs Slow

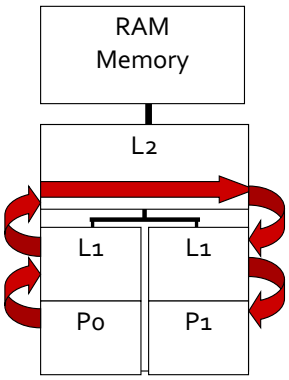
- Serializing at the mutex



  - The processors wait on each other

Closer Look: Motion of count, m

- Lock Reference and Contention




```
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 Count

- Each processor adds into its own memory; combine at the end

```
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);
```

Accumulate Into Private Count

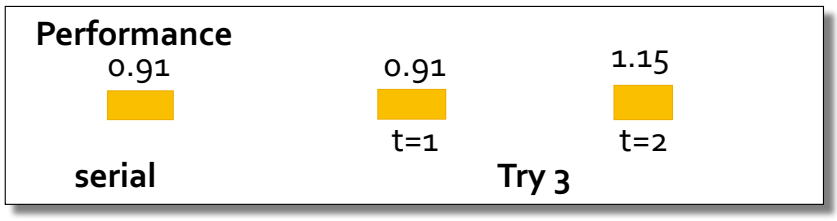
- Each processor adds into its own memory; combine at the end

Try 3

```
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);
```

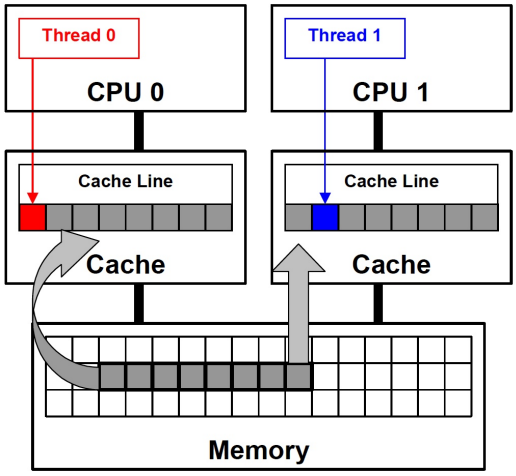

Keeping Up, But Not Gaining

- Sequential and 1 processor match, but it's a loss with 2 processors



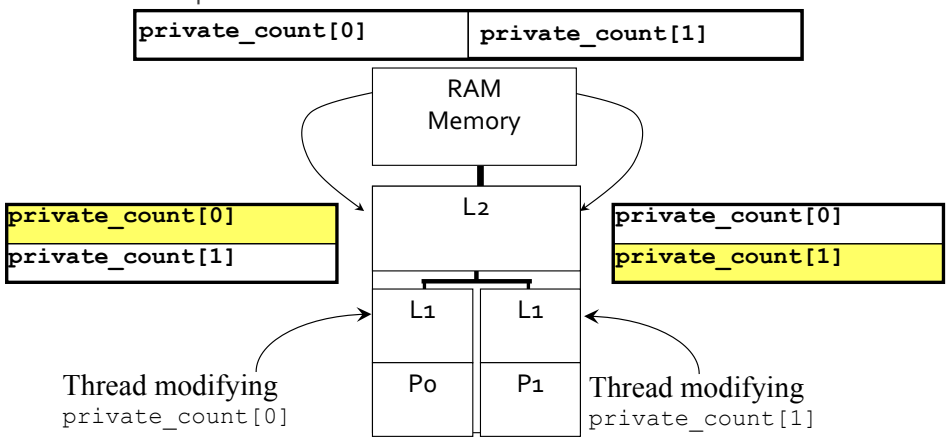
False Sharing

- A well-known performance issue on SMP systems where each processor has a local cache
- False sharing occurs when threads on different processors modify variables that reside on the same cache line
  - It is called false sharing because each thread is not actually sharing access to the same variable



False Sharing

- Private var ≠ private cache-line

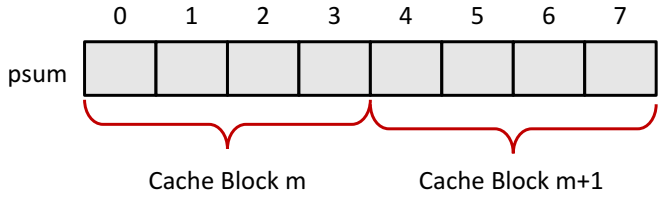


False Sharing: OMP Example

```
double sum=0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
   int me = omp_get_thread_num();
   sum_local[me] = 0.0;
   #pragma omp for
   for (i = 0; i < N; i++)
      sum_local[me] += x[i] * y[i];
   #pragma omp atomic
      sum += sum_local[me];
}
```


False Sharing

- Coherency maintained on cache blocks
- To update psum[i], thread i must have exclusive access
  - Threads sharing common cache block will keep fighting each other for access to block



Force Into Different Lines

- Padding the private variables forces them into separate cache lines and removes false sharing

```
struct padded_int
  {  int value;
     char padding[128];
  }  private_count[MaxThreads];
```


Force Into Different Lines

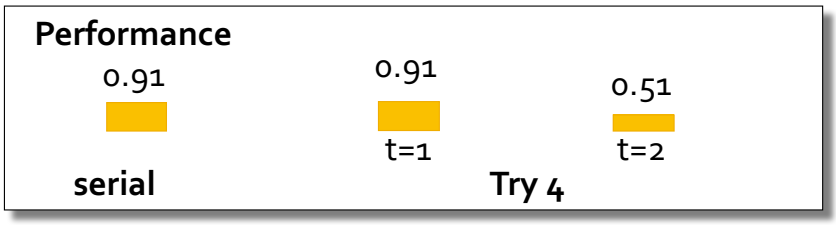
- Padding the private variables forces them into separate cache lines and removes false sharing

```
struct padded_int
  {  int value;
     char padding[128];
  }  private_count[MaxThreads];
```

Try 4

Success!!

- Two processors are almost twice as fast



- Is this the best solution???

Count 3s Summary

- Recapping the experience of writing the program, we
  - Wrote the obvious "break into blocks" program
  - We needed to protect the count variable
  - We got the right answer, but the program was slower … lock congestion
  - Privatized memory and 1-process was fast enough, 2- processes slow … false sharing
  - Separated private variables to own cache line

Finally, success

Variations

- What happens when more processors are available?
  - 4 processors
  - 8 processors
  - 256 processors
  - 32,768 processors

Parallel Programming Goals

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

Parallel Prefix

Consider A Simple Task …

- Adding a sequence of numbers A[0],…,A[n-1]
- Standard way to express it

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

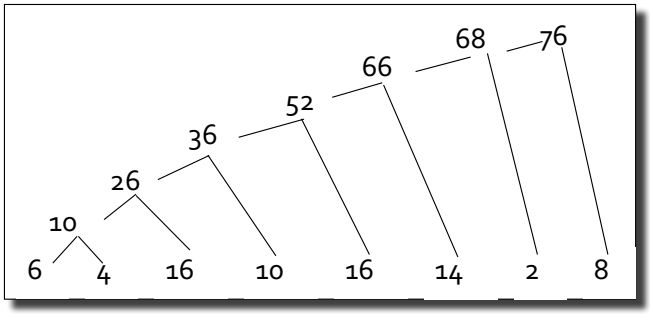
- Semantics require: (…((sum+A[0])+A[1])+…)+A[n-1]
  - That is, sequential
- Can it be executed in parallel?

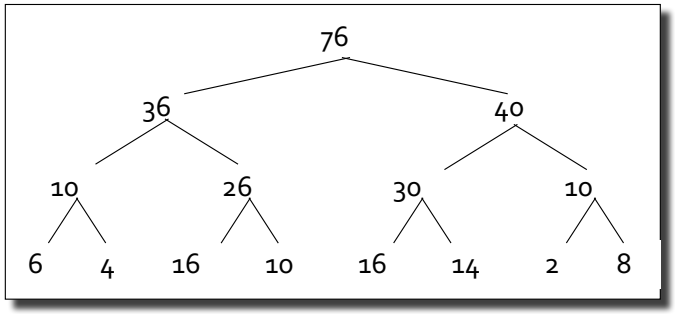
Parallel Summation

- To sum a sequence in parallel
  - add pairs of values producing 1st level results,
  - add pairs of 1st level results producing 2nd level results,
  - sum pairs of 2nd level results …
- That is, (…((A[0]+A[1]) + (A[2]+A[3])) + … + (A[n-2]+A[n-1]))…)

Express the Two Formulations

- Graphic representation makes difference clear





  - Same number of operations; different order

The Dream …

- Since 70s (Illiac IV days) the dream has been to compile sequential programs into parallel object code
  - Three decades of continual, well-funded research by smart people implies it's hopeless
    - For a tight loop summing numbers, its doable
    - For other computations it has proved extremely challenging to generate parallel code, even with pragmas or other assistance from programmers

What's the Problem?

- It's not likely a compiler will produce parallel code from a C specification any time soon…
- Fact
  - For most computations, a "best" sequential solution (practically, not theoretically) and a "best" parallel solution are usually fundamentally different …
    - Different solution paradigms imply computations are not "simply" related
    - Compiler transformations generally preserve the solution paradigm

Therefore… the programmer must discover the || solution

A Related Computation

- Consider computing the prefix sums

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

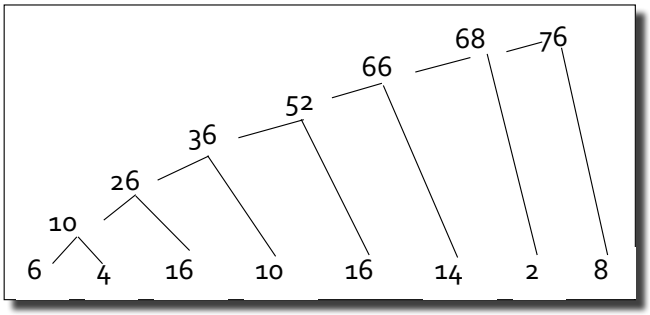
A[i] is the sum of the first i + 1 elements

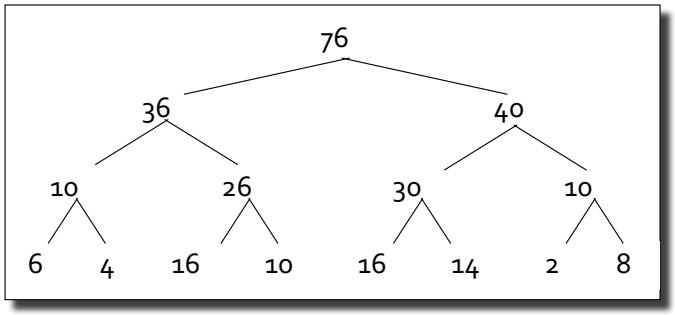
- Semantics …
  - A[0] is unchanged
  - A[1] = A[1] + A[0]
  - A[2] = A[2] + (A[1] + A[0])
  - …
  - A[n-1] = A[n-1] + (A[n-2] + (… (A[1] + A[0]) …)

What advantage can ||ism give?

Comparison of Paradigms

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





- Or does it?

Parallel Prefix

- Assume that *n* operands are input to the leaves of the complete binary tree
- Algorithm
  - Compute the grand total at the root by pair-wise sum;
  - On completion, root receives a 0 from its (nonexistent) parent;
  - All non-leaf nodes receive a value from their parent, relay that value to their left child, and send their right child the sum of the parent's value and their left child's value that was computed on the way up;
  - Leaves add the prefixes from computed above values and saved input.
- Time: Phase1 : log n - 1, Phase2: 2 , Phase 3: 2, Phase 4: log n - 1

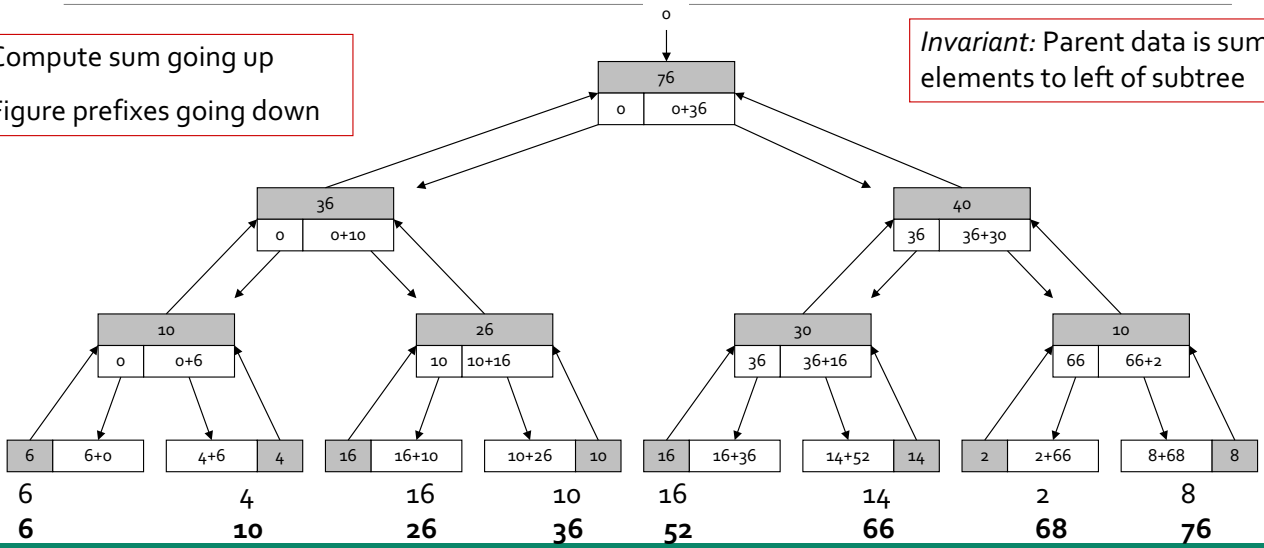
non-leaf nodes receive a value from their parent, relay that value to their left child, and send their right child the sum of the parent's value and their left child's value that was computed on the way up;

Parallel Prefix Algorithm

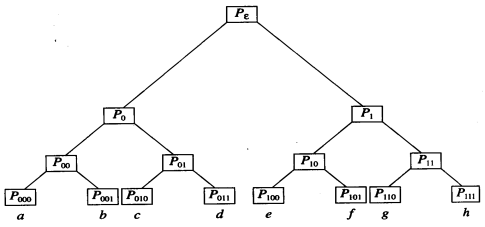
Compute sum going up

Figure prefixes going down

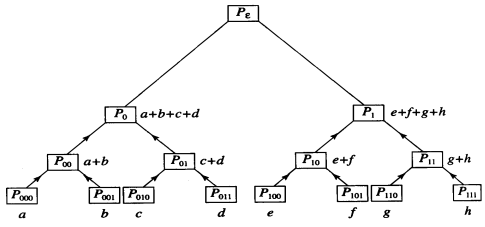
Invariant: Parent data is sum of elements to left of subtree



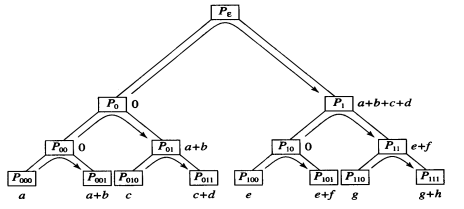
Parallel Prefix : On the complete binary tree



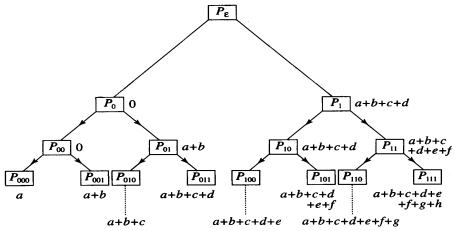
(a) Phase 1: Input the numbers in the leaves of PT_{2n-1}.



(b) Compute binary fan-in sums.



(c) Phase 2: For leaves, add sums in siblings and leave resulting sum in right child sibling. Phase 3: For non-root, non-leaf, left children, transfer binary fan-in sum to sibling then zero out own sum.



(d) Phase 4: Compute binary fan-out sums. Parallel prefix sums now reside in leaves.

Fundamental Tool of || Pgmming

- Original research on parallel prefix algorithm published by

  R. E. Ladner and M. J. Fischer

  Parallel Prefix Computation

  Journal of the ACM 27(4):831-838, 1980

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

Applies to a wide class of operations

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Matrix Multiplication

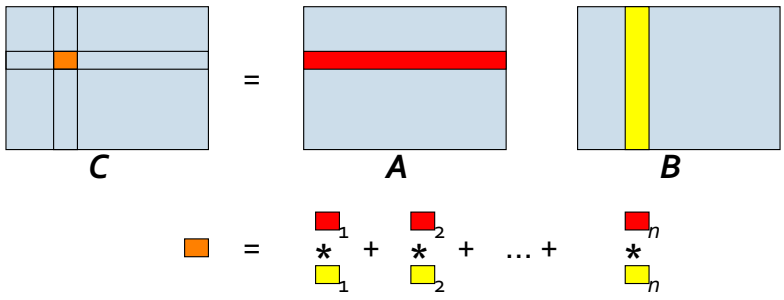
Matrix Product: || Poster Algorithm

- Matrix multiplication is most studied parallel algorithm (analogous to sequential sorting)
- Many solutions known
  - Illustrate a variety of complications
  - Demonstrate great solutions
- Our goal: explore variety of issues
  - Amount of concurrency
  - Data placement
  - Granularity

Exceptional by requiring $O(n^3)$ ops on $O(n^2)$ data

Recall the computation…

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

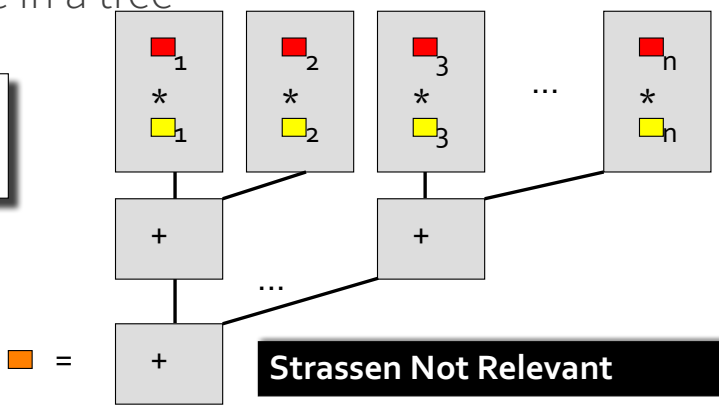


Extreme Matrix Multiplication

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

O(*n*) processors for each result element implies O(n^3) total

Time: O(*log n*)



Strassen Not Relevant

O(*log n*) MM in the real world …

Good properties

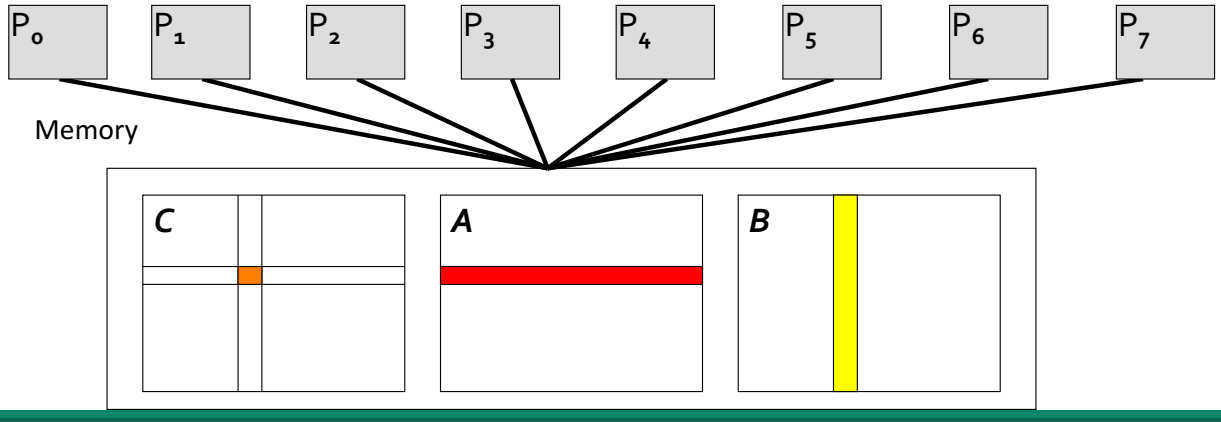
- Extremely parallel … shows limit of concurrency
- Very fast -- $log_2 n$ is a good bound … faster?

Bad properties

- Ignores memory structure and reference collisions
- Ignores data motion and communication costs
- Under-uses processors -- half of the processors do only 1 operation

Where is the data?

- Data references collisions and communication costs are important to final result … need a model … can generalize the standard RAM to get PRAM



Parallel Random Access Machine

- Any number of processors, including nc
- Any processor can reference any memory in "unit time"
- Resolve Memory Collisions
  - Read Collisions -- simultaneous reads to location are OK
  - Write Collisions -- simultaneous writes to loc need a rule:
    - Allowed, but must all write the same value
    - Allowed, but value from highest indexed processor wins
    - Allowed, but a random value wins
    - Prohibited

Caution: The PRAM is *not* a model we advocate

PRAM says O(log n) MM is good

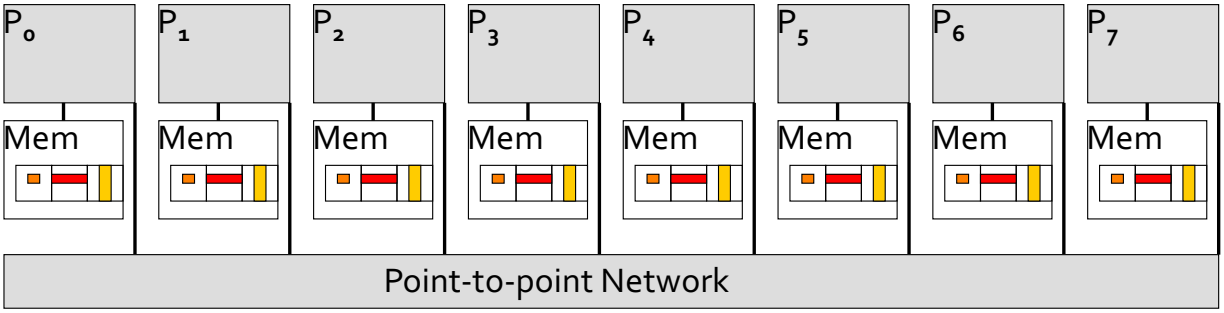
- PRAM allows any # processors => O(n^3) processors are OK
- ***A*** and ***B*** matrices are read simultaneously, but that's OK
- ***C*** is written simultaneously, but no location is written by more than 1 processor => OK

PRAM model implies O(*log n*) algorithm is best … but in real world, we suspect not

We return to this point later

Where else could data be?

- Local memories of separate processors …

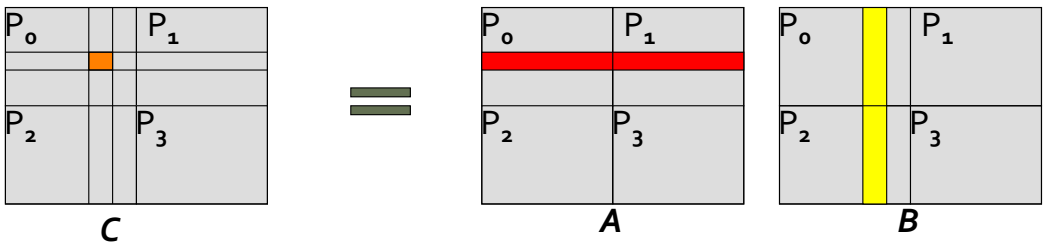


- Each processor could compute block of C
  - Avoid keeping multiple copies of A and B

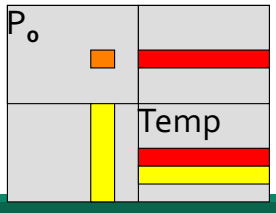
Architecture common for servers

Data Motion

- Getting rows and columns to processors

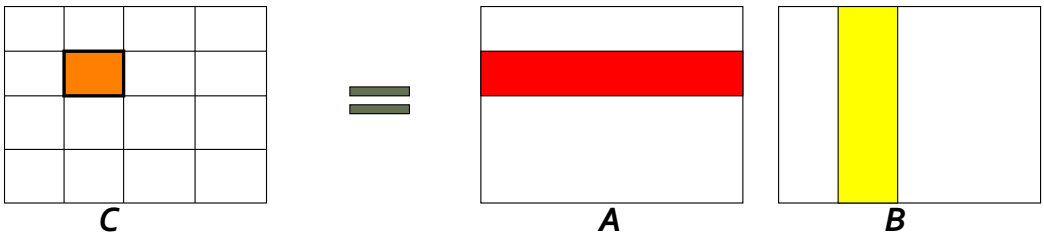


  - Allocate matrices in blocks
  - Ship only portion being used



Blocking Improves Locality

- Compute a b x b block of the result



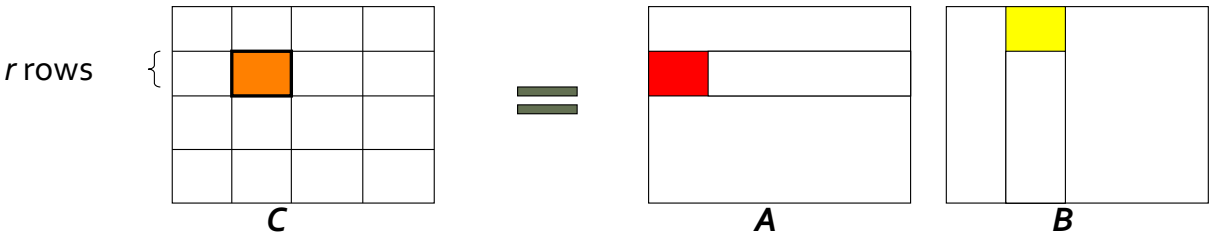
- Advantages
  - Reuse of rows, columns = caching effect
  - Larger blocks of local computation = hi locality

Caching in Parallel Computers

- Blocking = caching … why not automatic?
  - Blocking improves locality, but it is generally a manual optimization in sequential computation
  - Caching exploits two forms of locality
    - Temporal locality -- refs clustered in time
    - Spatial locality -- refs clustered by address
- When multiple threads touch the data, global reference sequence may not exhibit clustering features typical of one thread -- thrashing

Sweeter Blocking

- It's possible to do even better blocking …



- Completely use the cached values before reloading

Best MM Algorithm?

- We haven't decided on a good MM solution
- A variety of factors have emerged
  - A processor's connection to memory, unknown
  - Number of processors available, unknown
  - Locality--always important in computing--
    - Using caching is complicated by multiple threads
    - Contrary to high levels of parallelism
- Conclusion: Need a better understanding of the constraints of parallelism

Next week, architectural details + model of ||ism