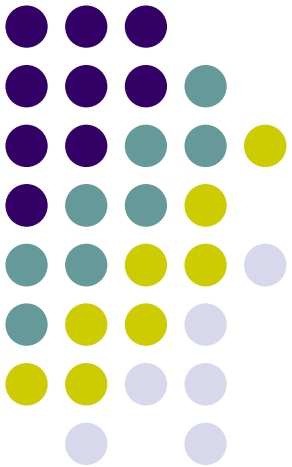
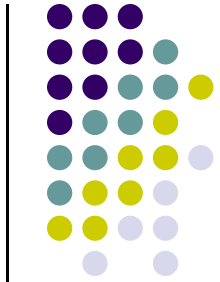
ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

Variable Sharing in OpenMP
OpenMP synchronization issues
OpenMP performance issues



November 4, 2015
Lecture 22

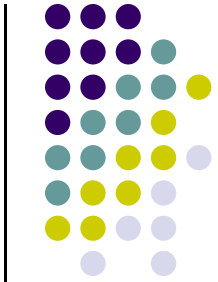
Quote of the Day



“You have power over your mind - not outside events. Realize this, and you will find strength.”

-- Marcus Aurelius, Roman Emperor
121 AD -- 180 AD

Before We Get Started



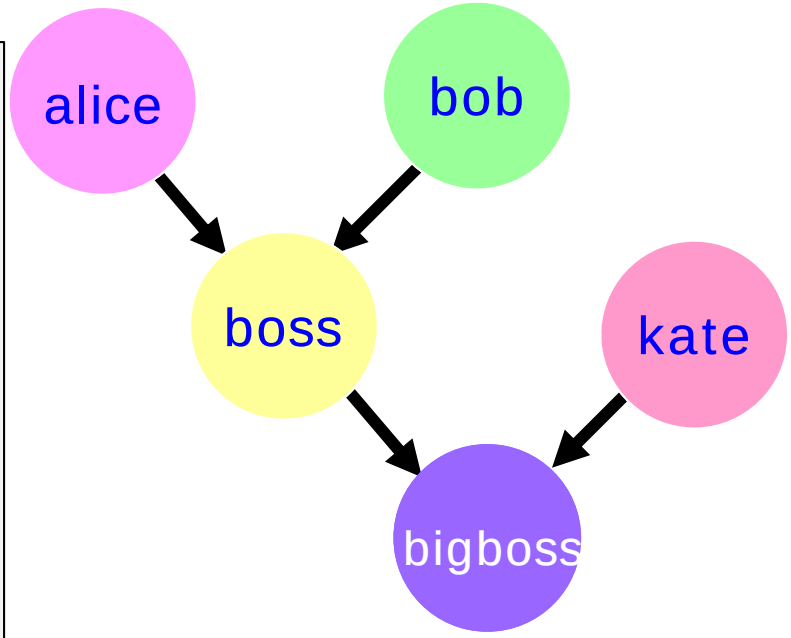
- Issues covered last time:
  - Work sharing in OpenMP
    - Parallel for
    - Parallel sections
    - Parallel tasks
  - Data sharing OpenMP

- Today's topics
  - Data sharing in OpenMP, wrap up
  - OpenMP, caveats

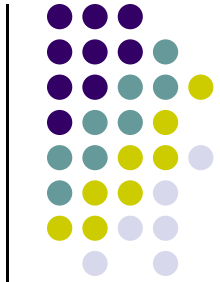
- Other issues:
  - Assignment: HW07 - due today at 11:59 PM
  - HW08 assigned today, posted on the class website
  - Final project proposal: 2 pages, due on 11/13 at 11:59 pm (Learn@UW dropbox)

Functional Level Parallelism Using omp sections

```
#pragma omp parallel sections
{
#pragma omp section
    double a = alice();
#pragma omp section
    double b = bob();
#pragma omp section
    double k = kate();
}

 double s = boss(a, b);
 printf ("%6.2f\n", bigboss(s,k));
```



Left: not good. “a” value garbage outside scope of the section
Right: good.



```
#pragma omp parallel sections
{
#pragma omp section
  {
    double a = alice();
  }
#pragma omp section
 {
    double b = bob();
 }
#pragma omp section
 {
    double k = kate();
 }
}

 double s = boss(a, b);
 printf ("%6.2f\n", bigboss(s,k));
```

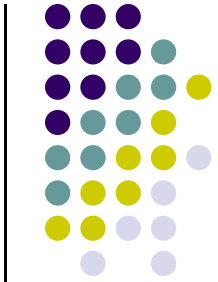
```
#pragma omp parallel sections
{
#pragma omp section
    a = alice();
#pragma omp section
    b = bob();
#pragma omp section
    k = kate();
}

 double s = boss(a, b);
 printf ("%6.2f\n", bigboss(s,k));
```

[IOMPP]→

```
using namespace std ;
typedef list<double> LISTDBL;

void doSomething(LISTDBL::iterator& itrtr) {
    *itrtr *= 2.;
}
int main() {
    LISTDBL test;                    // default constructor
    LISTDBL::iterator it;

    for( int i=0;i<4;++i)
        for( int j=0;j<8;++j) test.insert(test.end(), pow(10.0,i+1)+j);
    for( it = test.begin(); it!= test.end(); it++ ) cout << *it << endl;

    it =  test.begin();
#pragma omp parallel num_threads(8)
    {
#pragma omp single
        {
            while( it != test.end() ) {
#pragma omp task firstprivate(it)
                {
                    doSomething(it);
                }
                it++;
            }
        }
    }
    for( it = test.begin(); it != test.end(); it++ ) cout << *it << endl;
    return 0;
}
```

```
#include <omp.h>
#include <list>
#include <iostream>
#include <math.h>
```

Data Scoping, Words of Wisdom



- When in doubt, explicitly indicate who's what

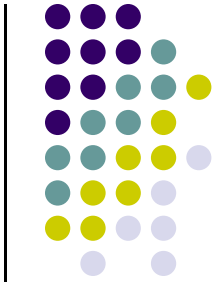
- Data scoping: common sources of errors in OpenMP
  - It takes some practice before you understand default behavior
  - Scoping: Not always intuitive

```
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
  {
  tid = omp_get_thread_num();
  if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  printf("Thread %d starting...\n",tid);

  #pragma omp sections nowait
    {
    #pragma omp section
      {
      printf("Thread %d doing section 1\n",tid);
      for (i=0; i<N; i++)
        {
        c[i] = a[i] + b[i];
        printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
        }
      }

    #pragma omp section
      {
      printf("Thread %d doing section 2\n",tid);
      for (i=0; i<N; i++)
        {
        d[i] = a[i] * b[i];
        printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
        }
      }
    }  /* end of sections */

    printf("Thread %d done.\n",tid);
  }  /* end of parallel section */
```

When in doubt, explicitly indicate who's what

Q: Do you see any problem with this piece of code?

Example: Shared & Private Vars.

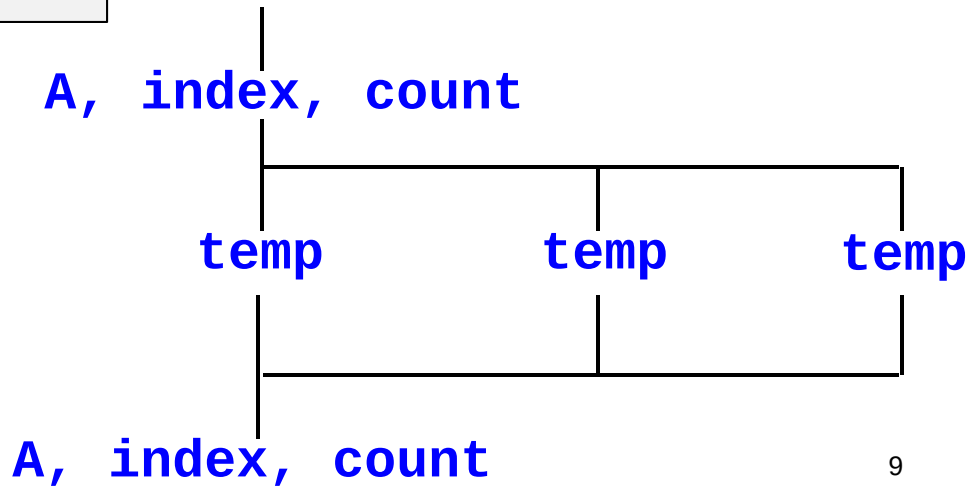


```
float A[10];
main () {
   int index[10];
#pragma omp parallel
   {
      Work (index);
   }
   printf ("%d\n", index[1]);
}
```

```
extern float A[10];
void Work (int *index)
{
   float temp[10];
   static int count;
   <...>
}
```

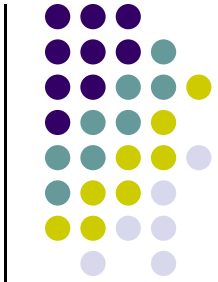
Assumed to be in another translation unit

A, index, and count are shared by all threads, but temp is local to each thread

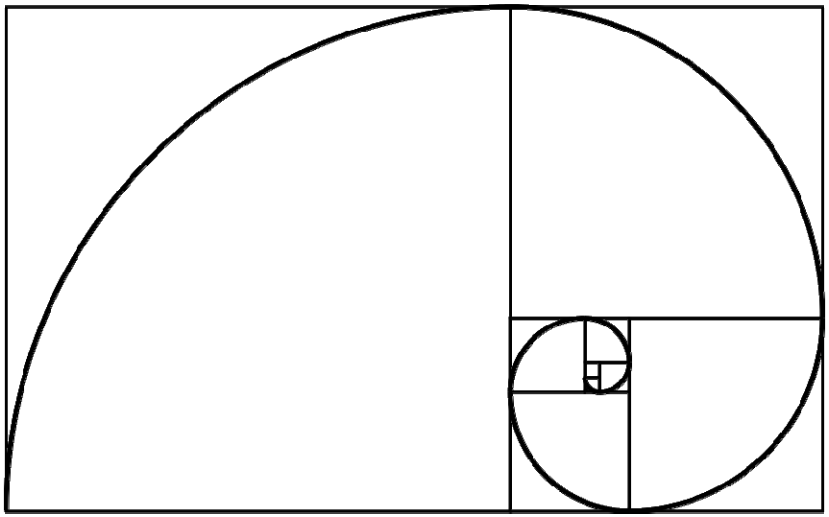


[IOMPP] →

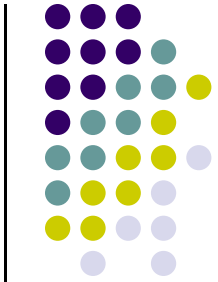
More on Variable Scoping: Fibonacci Sequence



- Start with
  - $F_0 = 0$
  - $F_1 = 1$
- Recursion formula
  - $F_N = F_{N-1} + F_{N-2}$
    - $N \geq 2$

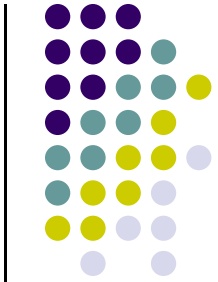


Example: Data Scoping Issue - `fib`



```
#include <stdio.h>
#include <omp.h>

int fib(int);

int main()
{
    int n = 10;
    omp_set_num_threads(4);

#pragma omp parallel
    {
#pragma omp single
        printf("fib(%d) = %d\n", n, fib(n));
    }
}
```

Example: Data Scoping Issue - `fib`



Assume that the parallel region exists outside of `fib` and that `fib` and the tasks inside it are in the dynamic extent of a parallel region

```
int fib ( int n ) {

    int x, y;
    if ( n < 2 ) return n;
#pragma omp task
    x = fib(n-1);
#pragma omp task
    y = fib(n-2);
#pragma omp taskwait

    return x+y;
}
```

n is private in both tasks

x is a private variable
y is a private variable

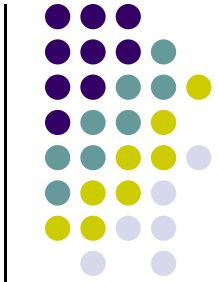
This is important - wait here on the completion of the child tasks spawned (two of them)

What's wrong here?

Values of the private variables not available outside of tasks

[IOMPP] →

Example: Data Scoping Issue - `fib`



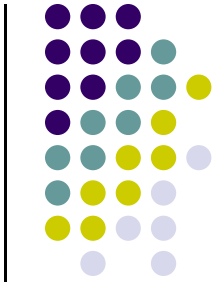
```
int fib ( int n ) {

    int x, y;
    if ( n < 2 ) return n;
#pragma omp task
{
    x = fib(n-1);
}
#pragma omp task
{
    y = fib(n-2);
}
#pragma omp taskwait

return x+y
}
```

x is a private variable
y is a private variable

Values of the private variables not available outside of task definition

[IOMPP] →

Example: Data Scoping Issue - `fib`



```
int fib ( int n ) {

    int x, y;
    if ( n < 2 ) return n;
#pragma omp task shared(x)
    x = fib(n-1);
#pragma omp task shared(y)
    y = fib(n-2);
#pragma omp taskwait

return x+y;
}
```

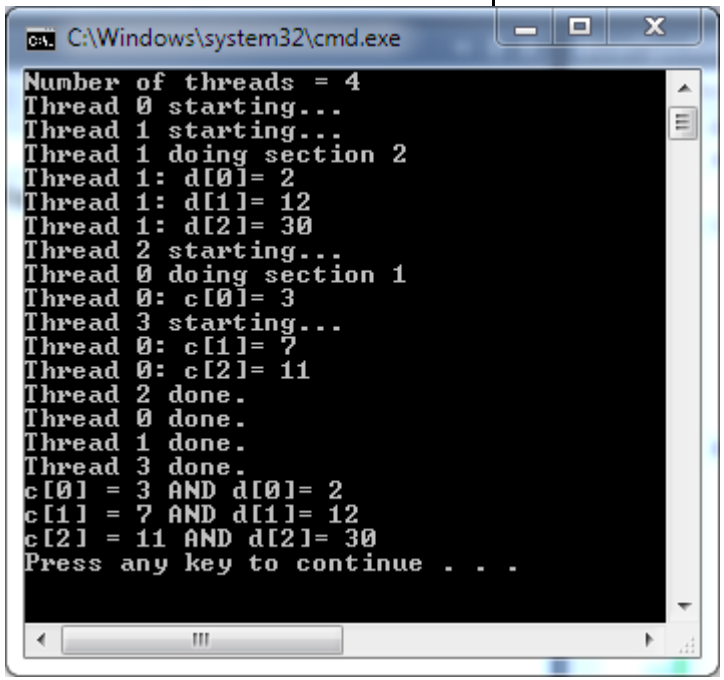
n is private in both tasks

x & y are now shared
we need both values to
compute the sum

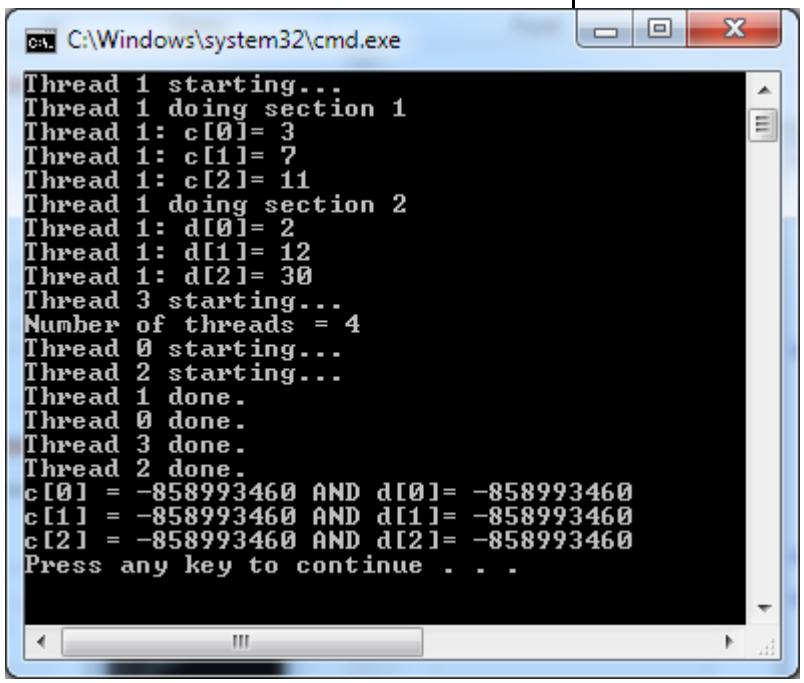
The values of the x & y variables will be available outside each task construct – after the taskwait

[IOMPP] →

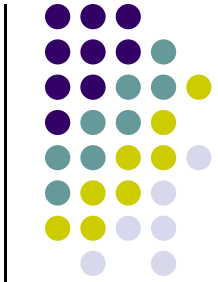
```c
#include <stdio.h>
#include <omp.h>

int main(void) {
    const int N = 3;
    int a[3] = { 2, 4, 6 };
    int b[3] = { 1, 3, 5 };
    int c[3], d[3];
    int i, tid, nthreads;

#pragma omp parallel private(i,tid) shared(a,b)
    {
        tid = omp_get_thread_num();
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

        printf("Thread %d starting...\n", tid);

#pragma omp sections
        {
#pragma omp section
            {
                printf("Thread %d doing section 1\n", tid);
                for (i = 0; i < N; i++) {
                    c[i] = a[i] + b[i];
                    printf("Thread %d: c[%d]= %d\n", tid, i, c[i]);
                }
            }
#pragma omp section
            {
                printf("Thread %d doing section 2\n", tid);
                for (i = 0; i < N; i++) {
                    d[i] = a[i] * b[i];
                    printf("Thread %d: d[%d]= %d\n", tid, i, d[i]);
                }
            }
        }  /* end of sections */

        printf("Thread %d done.\n", tid);
    }  /* end of parallel section */
    for (i = 0; i < N; i++) {
        printf("c[%d] = %d AND d[%d]= %d\n", i, c[i], i, d[i]);
    }

    return 0;
}
```



```c
#include <stdio.h>
#include <omp.h>

int main(void) {
    const int N = 3;
    int a[3] = { 2, 4, 6 };
    int b[3] = { 1, 3, 5 };
    int c[3], d[3];
    int i, tid, nthreads;

#pragma omp parallel private(i,tid, c, d) shared(a,b)
    {
        tid = omp_get_thread_num();
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

        printf("Thread %d starting...\n", tid);

#pragma omp sections
        {
#pragma omp section
            {
                printf("Thread %d doing section 1\n", tid);
                for (i = 0; i < N; i++) {
                    c[i] = a[i] + b[i];
                    printf("Thread %d: c[%d]= %d\n", tid, i, c[i]);
                }
            }
#pragma omp section
            {
                printf("Thread %d doing section 2\n", tid);
                for (i = 0; i < N; i++) {
                    d[i] = a[i] * b[i];
                    printf("Thread %d: d[%d]= %d\n", tid, i, d[i]);
                }
            }
        }  /* end of sections */

        printf("Thread %d done.\n", tid);
    }  /* end of parallel section */
    for (i = 0; i < N; i++) {
        printf("c[%d] = %d AND d[%d]= %d\n", i, c[i], i, d[i]);
    }

    return 0;
}
```



Work Plan



What is OpenMP?

Parallel regions

Work sharing

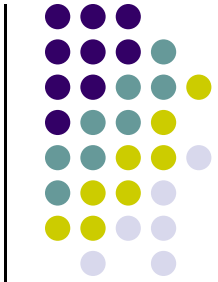
Data environment

Synchronization

- Advanced topics

[IOMPP] →

Implicit Barriers



- Several OpenMP constructs have *implicit* barriers
  - parallel – necessary barrier – cannot be removed
  - for
  - single

- Unnecessary barriers hurt performance and can be removed with the **nowait** clause
  - The **nowait** clause is applicable to:
    - for clause
    - single clause

Credit: IOMPP

Nowait Clause

```
#pragma omp for nowait
   for(...)
     {...};
```

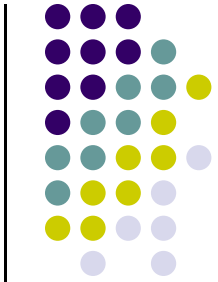
```
#pragma single nowait
{ [...] }
```

- Use when threads unnecessarily wait between independent computations

```
#pragma omp for schedule(dynamic,1) nowait
 for(int i=0; i<n; i++)
   a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
 for(int j=0; j<m; j++)
   b[j] = bigFunc2(j);
```


Credit: IOMPP

Barrier Construct

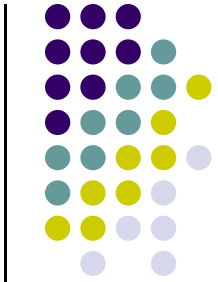


- Explicit barrier synchronization
- Each thread waits until all threads arrive

```
#pragma omp parallel shared(A, B, C)
{
    DoSomeWork(A,B); //input is A, output is B
#pragma omp barrier

    DoMoreWork(B,C); //input is B, output is C
}
```


Credit: IOMPP

[Preamble]
Atomic Construct



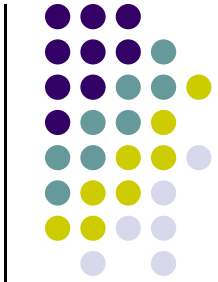
```
index[0] = 5;
index[1] = 3;
index[2] = 4;
index[3] = 0;
index[4] = 5;
index[5] = 5;
index[6] = 2;
index[7] = 1;
```

- Code runs ok sequentially (left)
- OpenMP code doesn't run ok (right)

```
for (i = 0; i < 8; i++) {
    x[index[i]] += work1(i);
    y[i] += work2(i);
}
```

```
#pragma omp parallel for
    for (i = 0; i < 8; i++) {
        x[index[i]] += work1(i);
        y[i] += work2(i);
    }
```

Atomic Construct

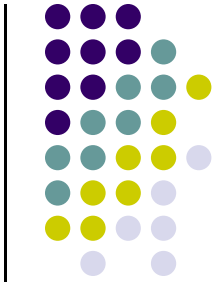


- Applies only to simple update of memory location
- Special case of a `critical` section, discussed shortly
  - Atomic introduces less overhead than `critical`

```
index[0] = 5;
index[1] = 3;
index[2] = 4;
index[3] = 0;
index[4] = 5;
index[5] = 5;
index[6] = 2;
index[7] = 1;
```

```
#pragma omp parallel for shared(x, y, index)
    for (i = 0; i < 8; i++) {
#pragma omp atomic
        x[index[i]] += work1(i);
        y[i] += work2(i);
    }
```

Synchronisation, Words of Wisdom

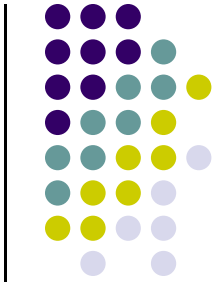


- Barriers can be very expensive
  - Typically 1000s cycles to synchronise

- Avoid barriers via:
  - *Careful* use of the NOWAIT clause
  - Parallelize at the outermost level possible
    - May require re-ordering of loops +/ indexes
  - Choice of CRITICAL / ATOMIC / lock routines may impact performance

Credit: Alan Real

Example: Dot Product

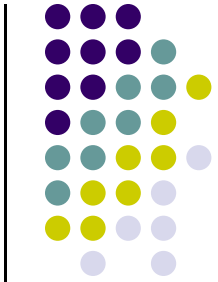


```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
#pragma omp parallel for
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

This is not good.

Credit: IOMPP

Race Condition



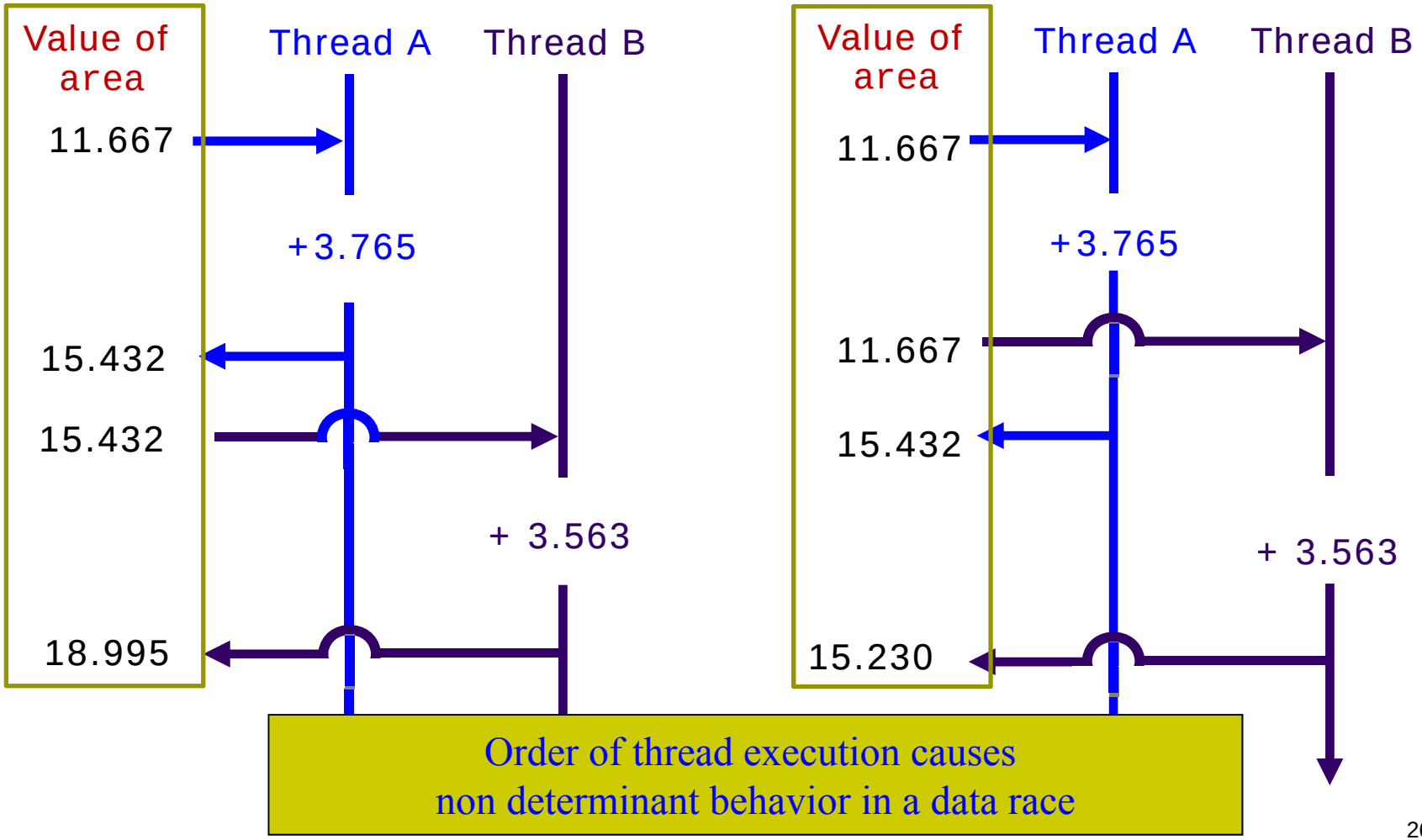
- Definition, *race condition* : two or more threads access a shared variable at the same time.
  - Leads to nondeterministic behavior

- For example, suppose that `area` is shared and both Thread A and Thread B are executing the statement

```
area += 4.0 / (1.0 + x*x);
```

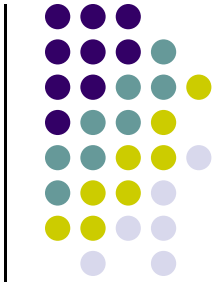

Credit: IOMPP

Two Possible Scenarios



Credit: IOMPP

Protect Shared Data



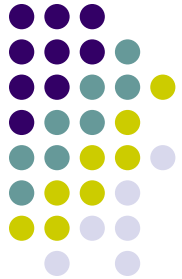
- The `critical` construct: protects access to shared, modifiable data
- The critical section allows only one thread to enter it at a given time

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
#pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
#pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```


Credit: IOMPP

OpenMP Critical Construct

```
#pragma omp critical [(lock_name)]
```



- Defines a critical region on a structured block

Threads wait their turn – only one at a time calls `consum()` thereby preventing race conditions

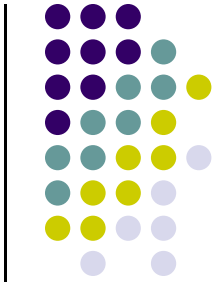
Naming the critical construct RES_lock is optional but highly recommended

```
float RES;
#pragma omp parallel
{
#pragma omp for
   for(int i=0; i<niters; i++) {
       float B = big_job(i);

#pragma omp critical (RES_lock)
       consum(B, RES);
   }
}
```

The "for" loop

Includes material from IOMPP

`reduction` Example

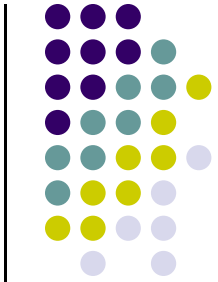


```
#pragma omp parallel for reduction(+:sum)
   for(i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
```

- Local copy of `sum` for each thread engaged in the reduction is private
  - Each local sum initialized to the identity operand associated with the operator that comes into play
    - Here we have “+”, so it’s a zero (0)
- All local copies of `sum` added together and stored in “global” variable

Credit: IOMPP

OpenMP `reduction` Clause

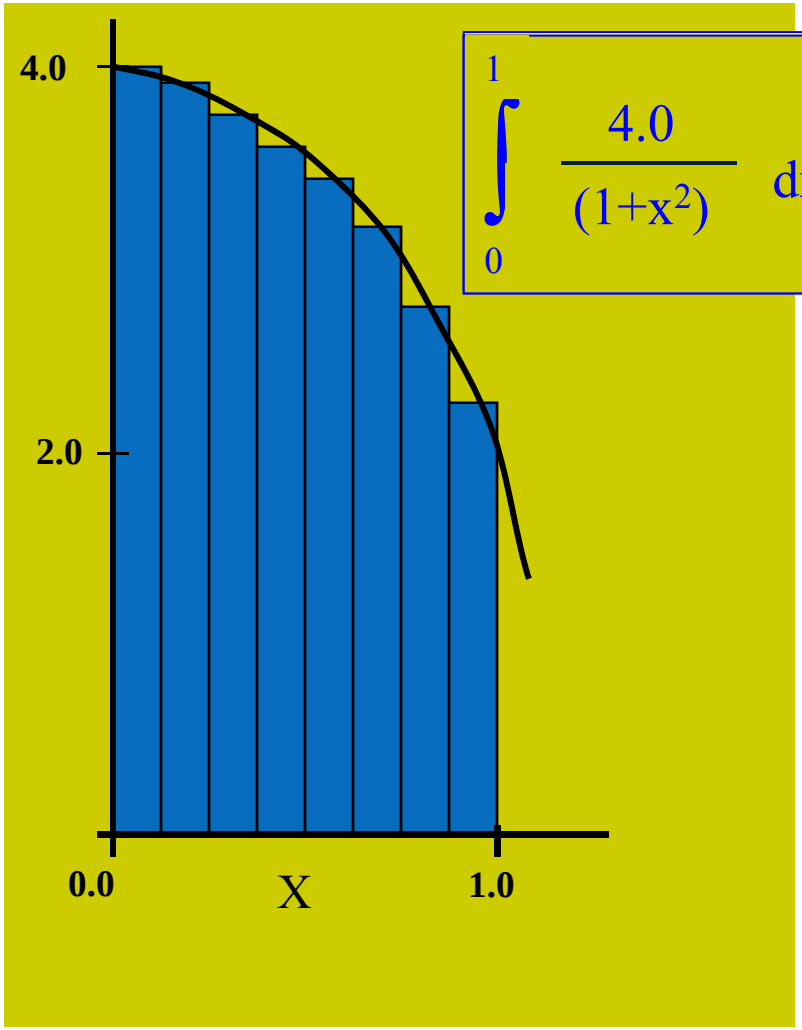


```
reduction (op:list)
```

- The variables in `list` will be shared in the enclosing parallel region
- Here's what happens inside the parallel or work-sharing construct:
  - A private copy of each list variable is created and initialized depending on the "op"
  - These copies are updated locally by threads
- At end of construct, local copies are combined through "op" into a single value

Credit: IOMPP

OpenMP Reduction Example: Numerical Integration

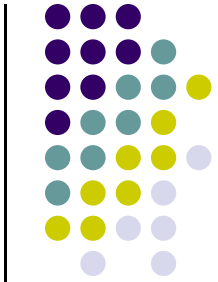


$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

```
static long num_steps=100000;
double step, pi;

void main() {
   int i;
   double x, sum = 0.0;

   step = 1.0/(double) num_steps;
   for (i=0; i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %f\n",pi);
}
```

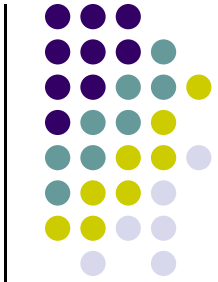

Credit: IOMPP

OpenMP Reduction Example: Numerical Integration



```
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"

int main(int argc, char* argv[]) {
    int num_steps = atoi(argv[1]);
    double step = 1./(double(num_steps));
    double sum;

#pragma omp parallel for reduction(+:sum)
{
     for(int i=0; i<num_steps; i++) {
         double x = (i + .5)*step;
         sum += 4.0/(1.+ x*x);
     }
}

    double my_pi = sum*step;
    printf("Value of integral is: %f\n", my_pi);

    return 0;
}
```

C/C++ Reduction Operations

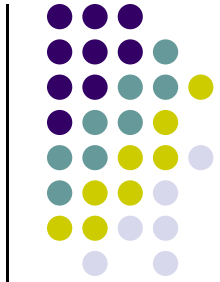


- A range of associative operands can be used with reduction

Operand	Initial Value
+	0
*	1
-	0
^	0

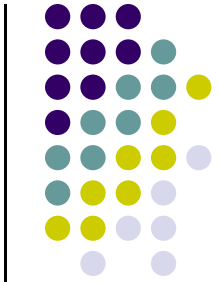
Operand	Initial Value
&	~0
|	0
&&	1
||	0

Credit: IOMPP



OpenMP Performance Issues

Performance

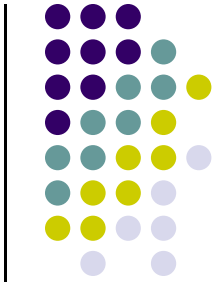


- Easy to write OpenMP yet hard to write an efficient program

- Five main causes of poor performance:
  - Sequential code
  - Communication
  - Load imbalance
  - Synchronisation
  - Compiler (non-)optimisation.

Credit: Alan Real

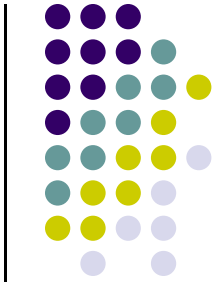
Sequential Code



- Corollary to Amdahl's law: A code that has been parallelized will never run faster than the sum of the parts executed sequentially
  - If most of the code continues to run sequentially, parallelization not going to make a difference
- Solution: Go back and understand whether you can approach the solution from a different perspective that exposes more parallelism
- Thinking within the context of OpenMP
  - All code outside of parallel regions and inside MASTER, SINGLE and CRITICAL directives is sequential
  - This code should be as small as possible.

[Alan Real]→

Communication



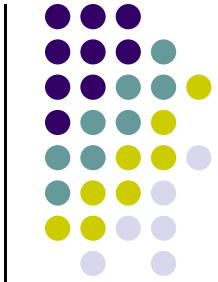
- On shared memory machines, which is where OpenMP operates, communication is “disguised” as increased memory access costs.
  - It takes longer to access data in main memory or another processor’s cache than it does from local cache.

- Memory accesses are expensive
  - ~100 cycles for a main memory access compared to 1-3 cycles for a flop.

- Unlike message passing, communication is spread throughout the program
  - Hard to analyse and monitor
  - NOTE: Message passing discussed next week. Programmer manages the movement of data through messages

Credit: Alan Real

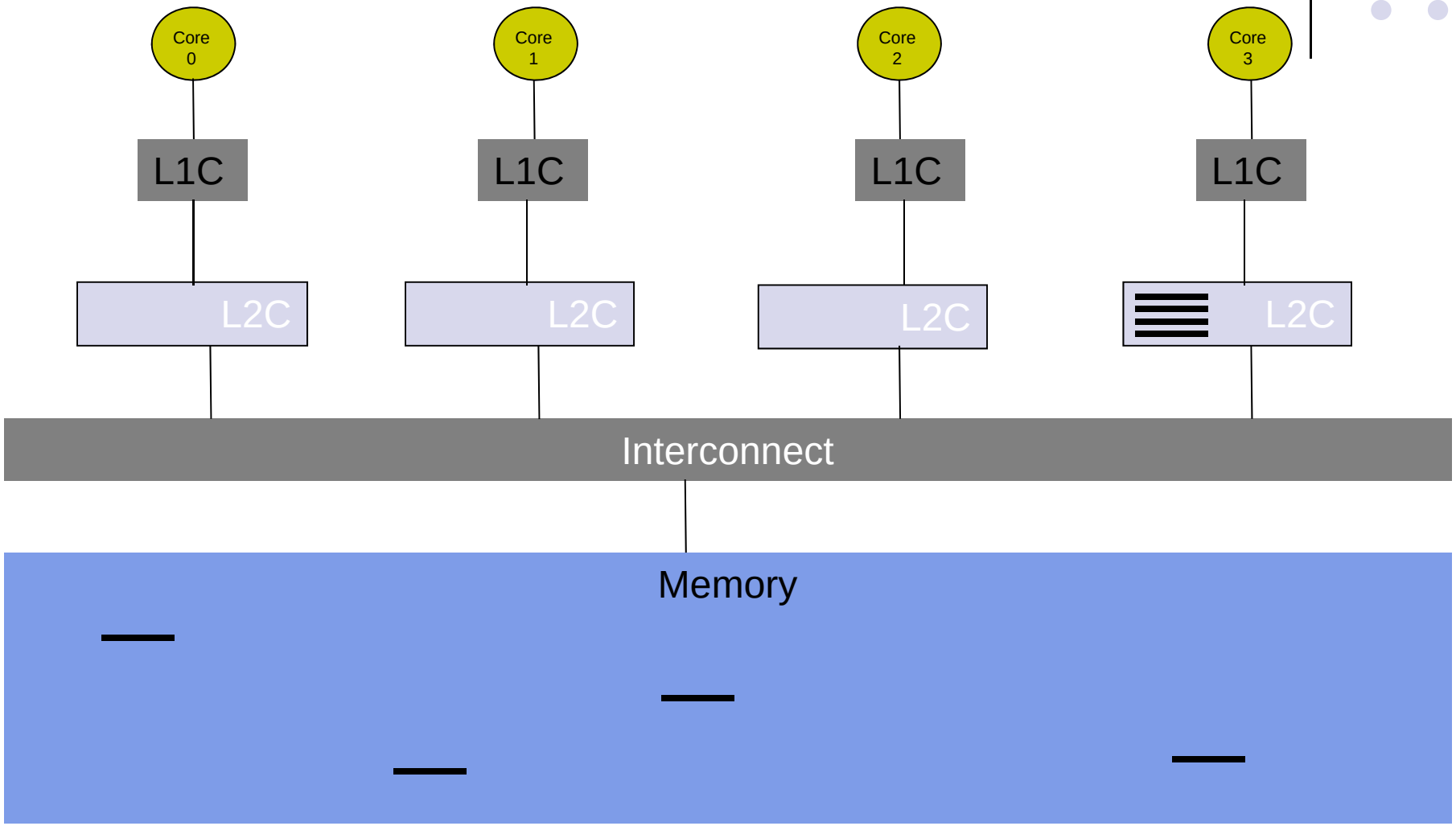
Caches and Coherency



- Shared memory programming assumes that a shared variable has a unique value at a given time
- Speeding up sequential computation: done through use of large caches
- Caching consequence: multiple copies of a physical memory location may exist at different hardware locations
- For program correctness, caches must be kept *coherent*
- Coherency operations are usually performed on the cache lines in the level of cache closest to memory
  - LLC last level cache: high end systems these days: LLC is level 3 cache
    - Can have 45 MB of L3 cache on a high end Intel CPU
- There is much overhead that goes into cache coherence

Credit: Alan Real

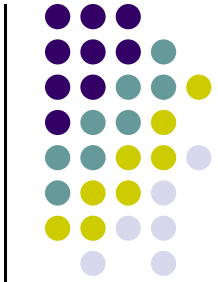
Memory Hierarchy, Multi-Core Scenario



Credit: Alan Real

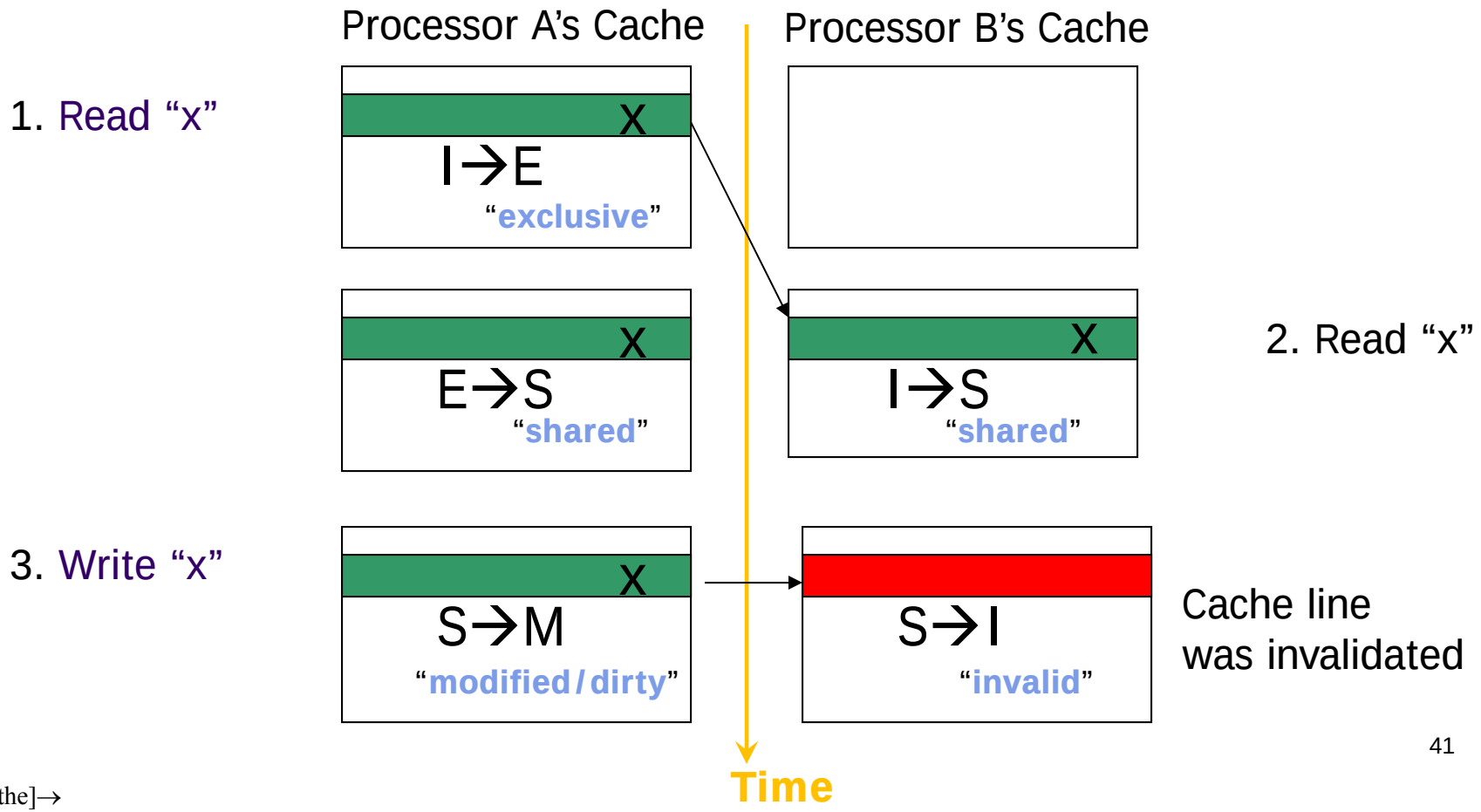
What Does MESI Mean to You?

MESI: Invalidation-Based Coherence Protocol



- Cache lines have state bits.
- Data migrates between processor caches, state transitions maintain coherence.

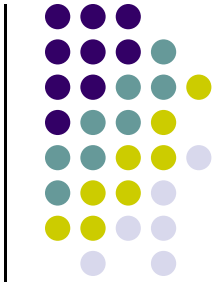
- **MESI** Protocol has four states: M: Modified, E: Exclusive, S: Shared, I: Invalid



[J. Marathe]→

Coherency – Simplified Further

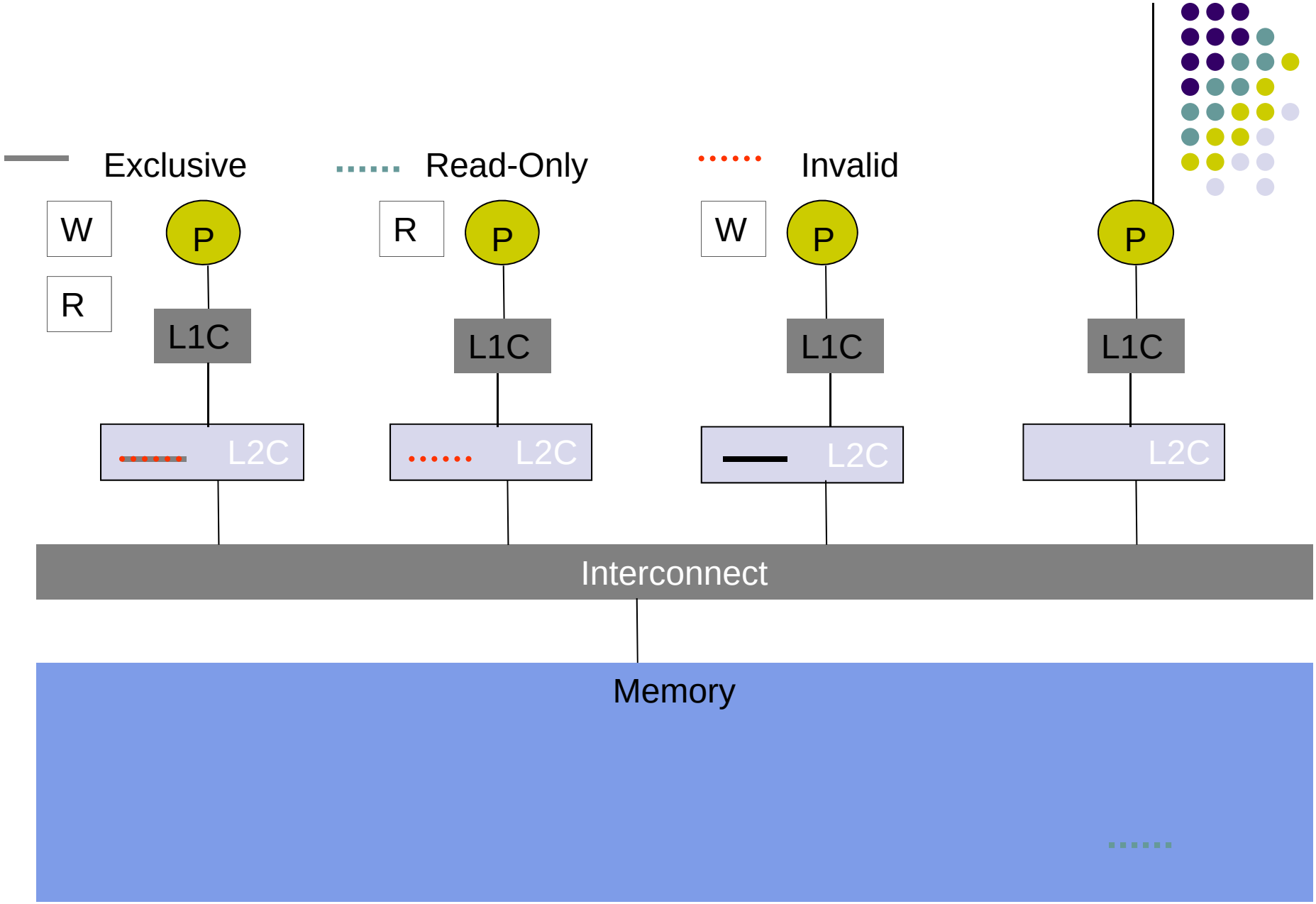
[cooked up example]



- Further simplify MESI for sake of simple discussion on next slide
  - Assume now that each cache line can exist in one of 3 states:
    - Exclusive: the only valid copy in any cache
    - Read-only: a valid copy but other caches may contain it
    - Invalid: out of date and cannot be used

- In this simplified coherency model
  - A READ on an *invalid* or *absent* cache line will be cached as *read-only* or *exclusive*
  - A WRITE on a line not in an *exclusive* state will cause all other copies to be marked *invalid* and the written line to be marked *exclusive*

Credit: Alan Real



Coherency example

Credit: Alan Real