

Potential correctness problems

- Both reads and writes to shared data must be locked, if a concurrent write is possible

```
typedef struct {  
    int egg_count;  
    double milk_qty;  
    pthread_mutex_t lock;  
} fridge;
```

```
EatEggOrDieTrying(fridge *f) {  
    pthread_mutex_lock(f->lock);  
    if(f->egg_count > 0) {  
        f->egg_count --;  
    }  
    else {  
        printf("Plan B: cereal\n");  
    }  
    pthread_mutex_unlock(f->lock);  
}
```

```
EggRun(fridge *f) {  
    pthread_mutex_lock(f->lock);  
    int eggs_left = f->egg_count;  
    if(eggs_left == 0) {  
        eggs_left = buy_carton();  
        f->egg_count += eggs_left;  
    }  
    pthread_mutex_unlock(f->lock);  
  
    printf("Eggs refilled: %d remaining!", f->egg_count);  
}
```

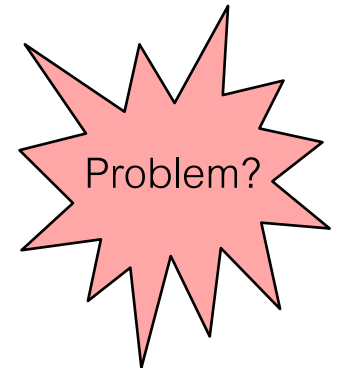
Potential correctness problems

- Both reads and writes to shared data must be locked, if a concurrent write is possible

```
typedef struct {  
    int egg_count;  
    double milk_qty;  
    pthread_mutex_t lock;  
} fridge;
```

```
EatEggOrDieTrying(fridge *f) {  
    pthread_mutex_lock(f->lock);  
    if(f->egg_count > 0) {  
        f->egg_count --;  
    }  
    else {  
        printf("Plan B: cereal\n");  
    }  
    pthread_mutex_unlock(f->lock);  
}
```

```
EggRun(fridge *f) {  
    pthread_mutex_lock(f->lock);  
    int eggs_left = f->egg_count;  
    if(eggs_left == 0) {  
        eggs_left = buy_carton();  
        f->egg_count += eggs_left;  
    }  
    pthread_mutex_unlock(f->lock);  
  
    printf("Eggs refilled: %d remaining!", f->egg_count);  
}
```



Potential correctness problems

- Both reads and writes to shared data must be locked, if a concurrent write is possible

```
typedef struct {  
    int egg_count;  
    double milk_qty;  
    pthread_mutex_t lock;  
} fridge;
```

```
EatEggOrDieTrying(fridge *f) {  
    pthread_mutex_lock(f->lock);  
    if(f->egg_count > 0) {  
        f->egg_count --;  
    }  
    else {  
        printf("Plan B: cereal\n");  
    }  
    pthread_mutex_unlock(f->lock);  
}
```

```
EggRun(fridge *f) {  
    pthread_mutex_lock(f->lock);  
    int eggs_left = f->egg_count;  
    if(eggs_left == 0) {  
        eggs_left = buy_carton();  
        f->egg_count += eggs_left;  
    }  
    pthread_mutex_unlock(f->lock);  
  
    printf("Eggs refilled: %d remaining!", f->egg_count);  
}
```

No lock around printf in the yellow box so possible bogus output of

Eggs refilled:
0 remaining!

Potential correctness problems

- Careful about losing track of a lock without unlocking
 - e.g., what happens here:

```
bool CanEatEggs(fridge *f) {  
    pthread_mutex_lock(f->lock);  
    int eggs_left = f->egg_count;  
    if(eggs_left == 0){  
        printf("Oh no!\n");  
        return false;  
    }  
    printf("Yummy, eggs!\n");  
    pthread_mutex_unlock(f->lock);  
    return true;  
}
```

- If a thread never releases a lock, all other waiting threads are stuck
 - Such concurrency bugs are called **deadlocks**! (more on this later...)

Locking – coarse vs fine-grained

- Locking large sections of code may not be efficient => limits concurrency
- What if T1 wants to do a MilkRun, while T2 does an EggRun?
 - Locking the fridge for the EggRun won't allow a MilkRun to happen
- Solution: fine-grained locking
 - Use smaller locks, lock **only what is needed**...
- Advantage: reduces unnecessary waiting/blocking, more parallelism

Example

- Separate locks => can run in parallel, higher degree of concurrency

```
typedef struct {  
    int egg_count;  
    pthread_mutex_t egg_lock;  
  
    double milk_qty;  
    pthread_mutex_t milk_lock;  
} fridge;
```

```
MilkRun(fridge *f) {  
    pthread_mutex_lock(f->milk_lock);  
    double milk_left = f->milk_qty;  
    if(milk_left == 0) {  
        milk_left = buy_carton();  
        f->milk_qty += milk_left;  
    }  
    pthread_mutex_unlock(f->milk_lock);  
}
```

```
EggRun(fridge *f) {  
    pthread_mutex_lock(f->egg_lock);  
    int eggs_left = f->egg_count;  
    if(eggs_left == 0) {  
        eggs_left = buy_carton();  
        f->egg_count += eggs_left;  
    }  
    pthread_mutex_unlock(f->egg_lock);  
}
```

Careful with fine-grained locking

- Morning routine includes eating eggs and drinking milk
- Must have both eggs and milk to eat breakfast, otherwise breakfast is ruined
- MorningRoutine thread executes concurrently with other threads that perform other breakfast routines => Locks are used (see code)
- Problem?

```
MorningRoutine(fridge *f, int e, double m) {
    pthread_mutex_lock(f->egg_lock);
    int eggs_left = f->egg_count;
    if(eggs_left > 0) {
        pthread_mutex_unlock(f->egg_lock);

        pthread_mutex_lock(f->milk_lock);
        double milk_left = f->milk_qty;
        if(milk_left > 0) {
            pthread_mutex_unlock(f->milk_lock);

            pthread_mutex_lock(f->egg_lock);
            f->egg_count -= e;
            pthread_mutex_unlock(f->egg_lock);

            pthread_mutex_lock(f->milk_lock);
            f->milk_qty -= m;
            pthread_mutex_unlock(f->milk_lock);
        } else {
            printf("Breakfast is ruined\n");
            pthread_mutex_unlock(f->milk_lock);
        }
    } else {
        printf("Breakfast is ruined\n");
        pthread_mutex_unlock(f->egg_lock);
    }
}
```

Careful with fine-grained locking

- Morning routine includes eating eggs and drinking milk
- Must have both eggs and milk to eat breakfast, otherwise breakfast is ruined
- MorningRoutine thread executes concurrently with other threads that perform other breakfast routines => Locks are used (see code)
- Problem? Two threads might try to eat the same egg!

```
MorningRoutine(fridge *f, int e, double m) {
    pthread_mutex_lock(f->egg_lock);
    int eggs_left = f->egg_count;
    if(eggs_left > 0) {
        pthread_mutex_unlock(f->egg_lock);

        pthread_mutex_lock(f->milk_lock);
        double milk_left = f->milk_qty;
        if(milk_left > 0) {
            pthread_mutex_unlock(f->milk_lock);

            pthread_mutex_lock(f->egg_lock);
            f->egg_count -= e;
            pthread_mutex_unlock(f->egg_lock);

            pthread_mutex_lock(f->milk_lock);
            f->milk_qty -= m;
            pthread_mutex_unlock(f->milk_lock);
        } else {
            printf("Breakfast is ruined\n");
            pthread_mutex_unlock(f->milk_lock);
        }
    } else {
        printf("Breakfast is ruined\n");
        pthread_mutex_unlock(f->egg_lock);
    }
}
```


Fine-grained locking and atomicity

- Must be aware of the program semantics to correctly use fine-grained locking and guarantee atomicity where race conditions are possible
- Solutions:
 - Restructure the code if possible
 - Lock larger sections to guarantee atomicity
- Let's fix the example...

Fine-grained locking and atomicity

```
MorningRoutine(fridge *f, int e, double m) {  
    pthread_mutex_lock(f->egg_lock);  
    int eggs_left = f->egg_count;  
    if(eggs_left > 0) {  
  
        pthread_mutex_lock(f->milk_lock);  
        double milk_left = f->milk_qty;  
        if(milk_left > 0) {  
  
            f->egg_count -= e;  
            pthread_mutex_unlock(f->egg_lock);  
  
            f->milk_qty -= m;  
            pthread_mutex_unlock(f->milk_lock);  
        } else {  
            printf("Breakfast is ruined\n");  
            pthread_mutex_unlock(f->milk_lock);  
        }  
    } else {  
        printf("Breakfast is ruined\n");  
        pthread_mutex_unlock(f->egg_lock);  
    }  
}
```

We made our locks
abit more coarse
grained

Fine-grained locking and atomicity


```
MorningRoutine(fridge *f, int e, double m) {  
    pthread_mutex_lock(f->egg_lock);  
    int eggs_left = f->egg_count;  
    if(eggs_left > 0) {  
  
        pthread_mutex_lock(f->milk_lock);  
        double milk_left = f->milk_qty;  
        if(milk_left > 0) {  
  
            f->egg_count -= e;  
            pthread_mutex_unlock(f->egg_lock);  
  
            f->milk_qty -= m;  
            pthread_mutex_unlock(f->milk_lock);  
        } else {  
            printf("Breakfast is ruined\n");  
            pthread_mutex_unlock(f->milk_lock);  
        }  
    } else {  
        printf("Breakfast is ruined\n");  
        pthread_mutex_unlock(f->egg_lock);  
    }  
}
```

Identify a glaring
correctness problem
in this code...

Fine-grained locking and atomicity

```
MorningRoutine(fridge *f, int e, double m) {  
    pthread_mutex_lock(f->egg_lock);  
    int eggs_left = f->egg_count;  
    if(eggs_left > 0) {  
  
        pthread_mutex_lock(f->milk_lock);  
        double milk_left = f->milk_qty;  
        if(milk_left > 0) {  
  
            f->egg_count -= e;  
            pthread_mutex_unlock(f->egg_lock);  
  
            f->milk_qty -= m;  
            pthread_mutex_unlock(f->milk_lock);  
        } else {  
            printf("Breakfast is ruined\n");  
            pthread_mutex_unlock(f->milk_lock);  
        }  
    } else {  
        printf("Breakfast is ruined\n");  
        pthread_mutex_unlock(f->egg_lock);  
    }  
}
```

The egg_lock might
never get unlocked if a
thread makes it to this
line



Other problems with fine-grained locking

- If we have two morning routines, eating egg then milk, eating milk then egg

```
MorningRoutine_1(fridge *f, int e, double m) {
    pthread_mutex_lock(f->egg_lock);
    int eggs_left = f->egg_count;
    if(eggs_left > 0) {

        pthread_mutex_lock(f->milk_lock);
        double milk_left = f->milk_qty;
        if(milk_left > 0) {

            f->egg_count -= e;
            pthread_mutex_unlock(f->egg_lock);

            f->milk_qty -= m;
            pthread_mutex_unlock(f->milk_lock);

        } else {
            printf("Breakfast is ruined\n");
            pthread_mutex_unlock(f->milk_lock);
            pthread_mutex_unlock(f->egg_lock);
        }

    } else {
        printf("Breakfast is ruined\n");
        pthread_mutex_unlock(f->egg_lock);
    }
}
```

```
MorningRoutine_2(fridge *f, int e, double m) {
    pthread_mutex_lock(f->milk_lock);
    double milk_left = f->milk_qty;
    if(milk_left > 0) {

        pthread_mutex_lock(f->egg_lock);
        int eggs_left = f->egg_count;
        if(eggs_left > 0) {

            f->milk_qty -= m;
            pthread_mutex_unlock(f->milk_lock);

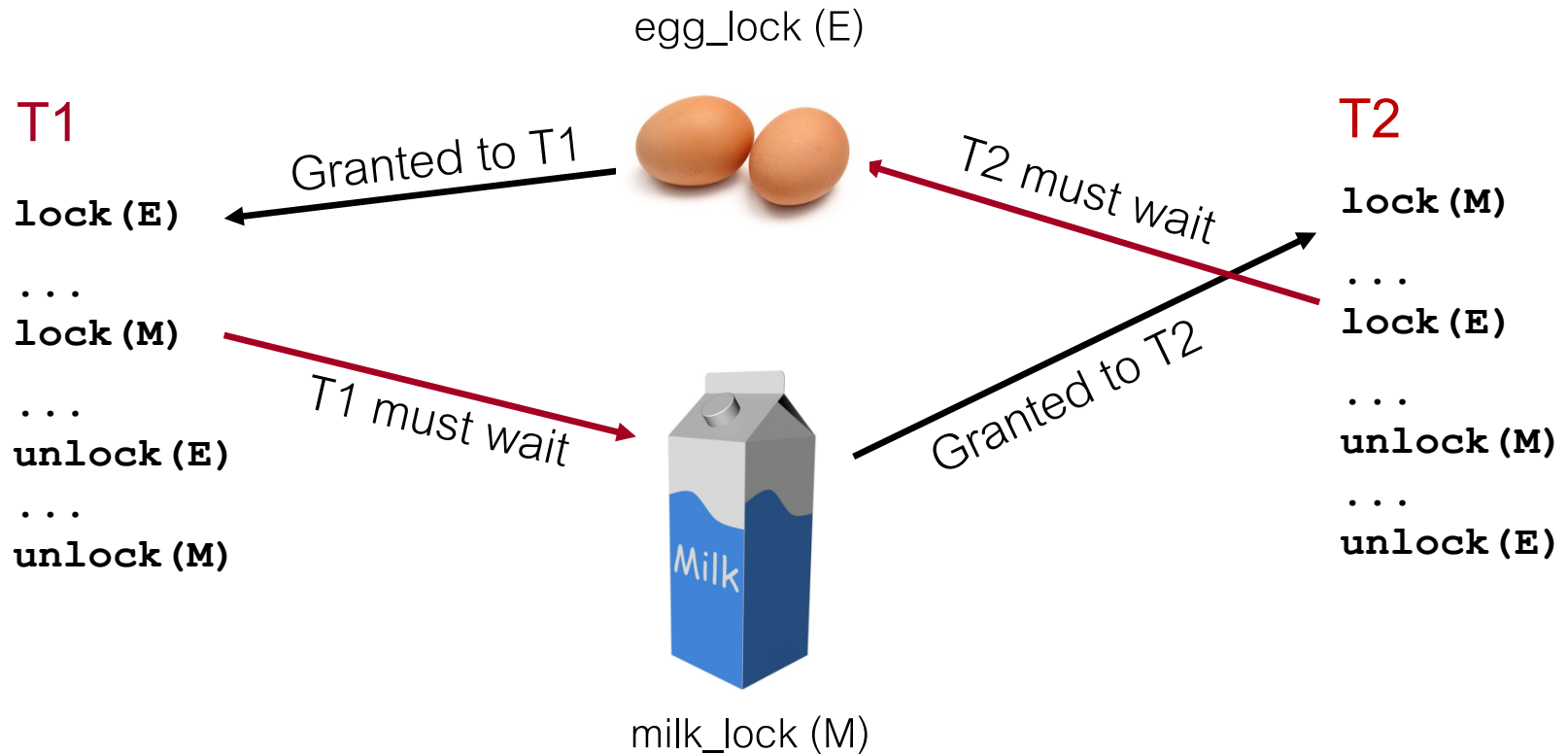
            f->egg_count -= e;
            pthread_mutex_unlock(f->egg_lock);

        } else {
            printf("Breakfast is ruined\n");
            pthread_mutex_unlock(f->milk_lock);
            pthread_mutex_unlock(f->egg_lock);
        }

    } else {
        printf("Breakfast is ruined\n");
        pthread_mutex_unlock(f->milk_lock);
    }
}
```

Deadlocks

- The **mutual** blocking of a set of threads (or processes)
- Each process/thread in the set is blocked, waiting for a lock which can only be unlocked by another process/thread in the set



- Simplest way to **break the deadlock**: always acquire locks in the same order!
 - Must enforce the **same ordering in every piece of code** where we acquire more than 1 lock

Next up...

- Overheads of locking
- Barrier construct

Overheads of locking

- When threads access the same locks => **lock contention!**
- If lots of threads are contending for the same lock, impacts performance
- Example: simple access to a shared counter by a handful of threads

```
void* do_work(void* arg) {  
    int i;  
    for (i = 0; i < loops; i++) {  
        pthread_mutex_lock(&mutex);  
        counter++;  
        pthread_mutex_unlock(&mutex);  
    }  
    return NULL;  
}
```

- Even when no lock contention, acquiring a lock has overheads
 - Try with 1 thread and lots of iterations, with and without the mutex

"Localize" your computations

- Idea: Compute as much as possible locally, use synchronization scarcely
 - Mind you, do not break mutual exclusion when needed for correctness!
- Example:

```
void* do_work(void* arg) {  
    int i;  
    for (i = 0; i < loops; i++) {  
        pthread_mutex_lock(&mutex);  
        counter++;  
        pthread_mutex_unlock(&mutex);  
    }  
    return NULL;  
}
```

- Idea: use local counter, update the global shared counter much more rarely...

Other synchronization primitives

- Semaphores
- Condition variables
- We'll focus only on the latter - powerful semantics

Barriers

- Threads that reach the barrier stop until all threads have reached it as well
 - If execution has stages, barrier ensures that data needed from a previous stage is ready
- POSIX has built-in barrier implementation
 - `int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrier_attr_t *attr, unsigned count);`
 - `int pthread_barrier_wait(pthread_barrier_t *barrier);`
 - `int pthread_barrier_destroy(pthread_barrier_t* barrier);`
 - Check pthread documentation for more details...

Next up ...

- Shared memory architecture
- Parallel programming models: shared memory
- Pthreads: Synchronization, Races, Locks
- OpenMP
- Cache coherency

Summary of Programming with Threads

- POSIX Threads are based on OS features
- Pitfalls
 - Overhead of thread creation is high (1-loop iteration probably too much)
 - Data race bugs are very nasty to find because they can be intermittent
 - Deadlocks are usually easier, but can also be intermittent
- OpenMP is commonly used today as an alternative
 - Helps with some of these, but doesn't make them disappear

What is OpenMP?

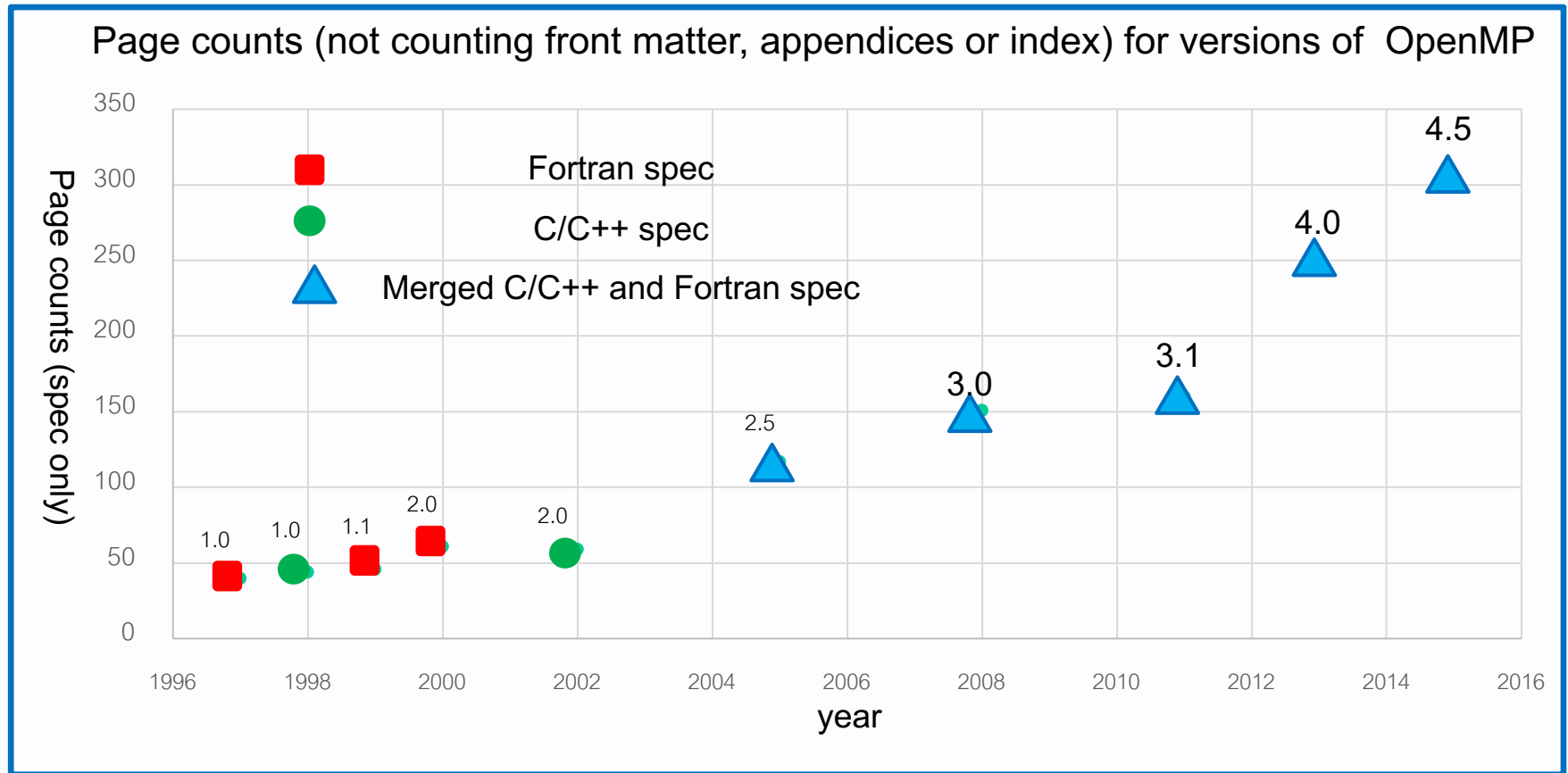
- OpenMP = Open specification for Multi-Processing
 - openmp.org – Talks, examples, forums, etc.
 - Spec controlled by the ARB
- Motivation: capture common usage and simplify programming
- OpenMP Architecture Review Board (ARB)
 - A nonprofit organization that controls the OpenMP Spec
 - Latest spec: OpenMP 4.5 (Nov. 2015), working on 5.0
- High-level API for programming in C/C++ and Fortran
 - Preprocessor (compiler) directives (~ 80%)
#pragma omp construct [clause [clause ...]]
 - Library Calls (~ 19%)
#include <omp.h>
 - Environment Variables (~ 1%)
all caps!

A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with “light” syntax
 - Requires compiler support (C, C++ or Fortran)
- OpenMP will:
 - Allow a programmer to separate a program into *serial regions* and *parallel regions*, rather than P concurrently-executing threads.
 - Hide stack management
 - Provide synchronization constructs
- OpenMP will not:
 - Parallelize automatically
 - Guarantee speedup
 - Provide freedom from data races

The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science. The complexity has grown considerably over the years!



The complexity of the full spec is overwhelming, so we focus on the 16 constructs most OpenMP programmers restrict themselves to ... the so called “OpenMP Common Core”

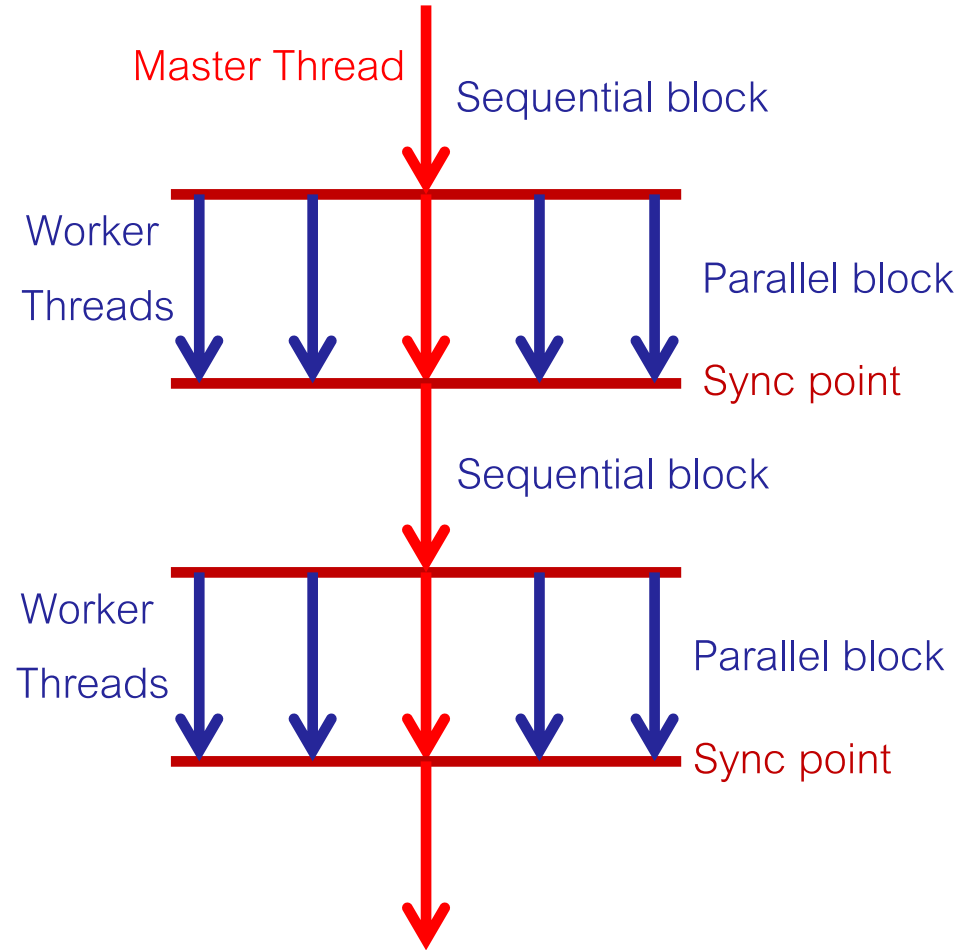
The OpenMP Common Core: Most OpenMP programs only use these 19 items

OpenMP pragma, function, or clause	Concepts
<code>#pragma omp parallel</code>	Parallel region, teams of threads, structured block, interleaved execution across threads
<code>int omp_get_thread_num()</code> <code>int omp_get_num_threads()</code>	Create threads with a parallel region and split up the work using the number of threads and thread ID
<code>double omp_get_wtime()</code>	Speedup and Amdahl's law.
<code>setenv OMP_NUM_THREADS N</code>	Internal control variables. Setting the default number of threads with an environment variable
<code>#pragma omp barrier</code> <code>#pragma omp critical</code>	Synchronization and race conditions.
<code>#pragma omp for</code> <code>#pragma omp parallel for</code>	Worksharing, parallel loops, loop carried dependencies
<code>reduction(op:list)</code>	Reductions of values across a team of threads
<code>schedule(dynamic [,chunk])</code> <code>schedule (static [,chunk])</code>	Loop schedules, loop overheads and load balance
<code>private(list)</code> , <code>firstprivate(list)</code> , <code>shared(list)</code>	Data environment
<code>nowait</code>	Disabling implied barriers on workshare constructs
<code>#pragma omp single</code>	Workshare with a single thread
<code>#pragma omp task</code> and <code>#pragma omp section</code> <code>#pragma omp taskwait</code>	Tasks including the data environment for tasks.

The course project might use some other routines, see the following link for more routines
<https://computing.llnl.gov/tutorials/openMP/>

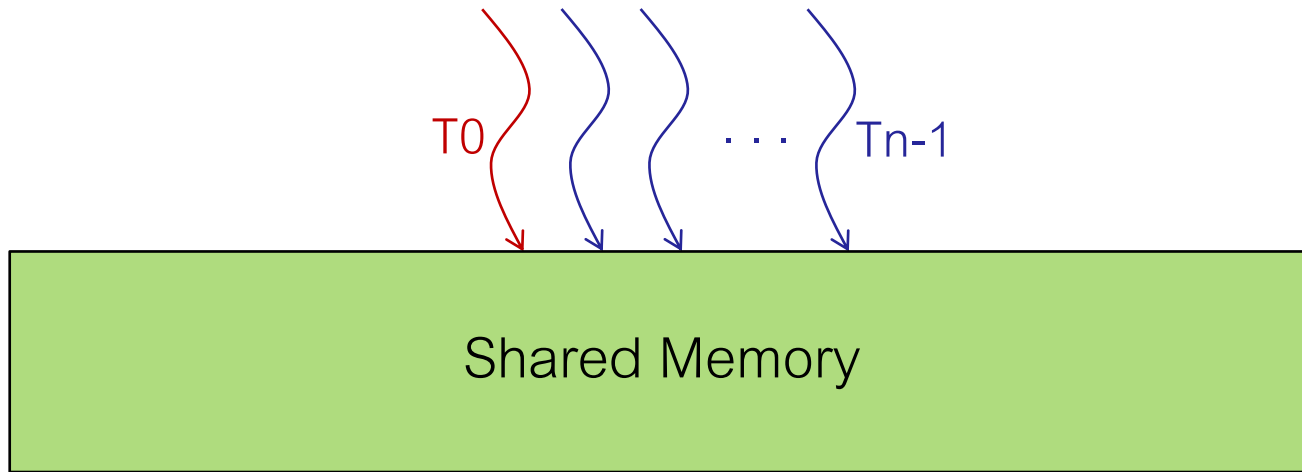
Execution model: Fork-and-Join

- Start with a single (**master**) thread
- Fork:** Create a group of worker threads
 - Everything in a parallel block is executed by every thread
- Join:** All threads synchronize at the end of the parallel block
 - Execution continues with only the initial (master) thread
- Threads can do the **exact same work**, share the **same tasks (work sharing)**, or perform **distinct tasks** in parallel!



Reminder: Shared Memory Model

- All worker threads share the same address space



- OpenMP provides the ability to declare variables private or shared within a parallel block (more on this later...)

OpenMP Programming Basics

- Programming model: provide "hints" or "**directives**" to the compiler as to what you intend to parallelize

- C/C++ **#pragma** compiler directives, followed by various clauses:

#pragma omp directivename [clause list]

- Must include OpenMP function headers

#include <omp.h>

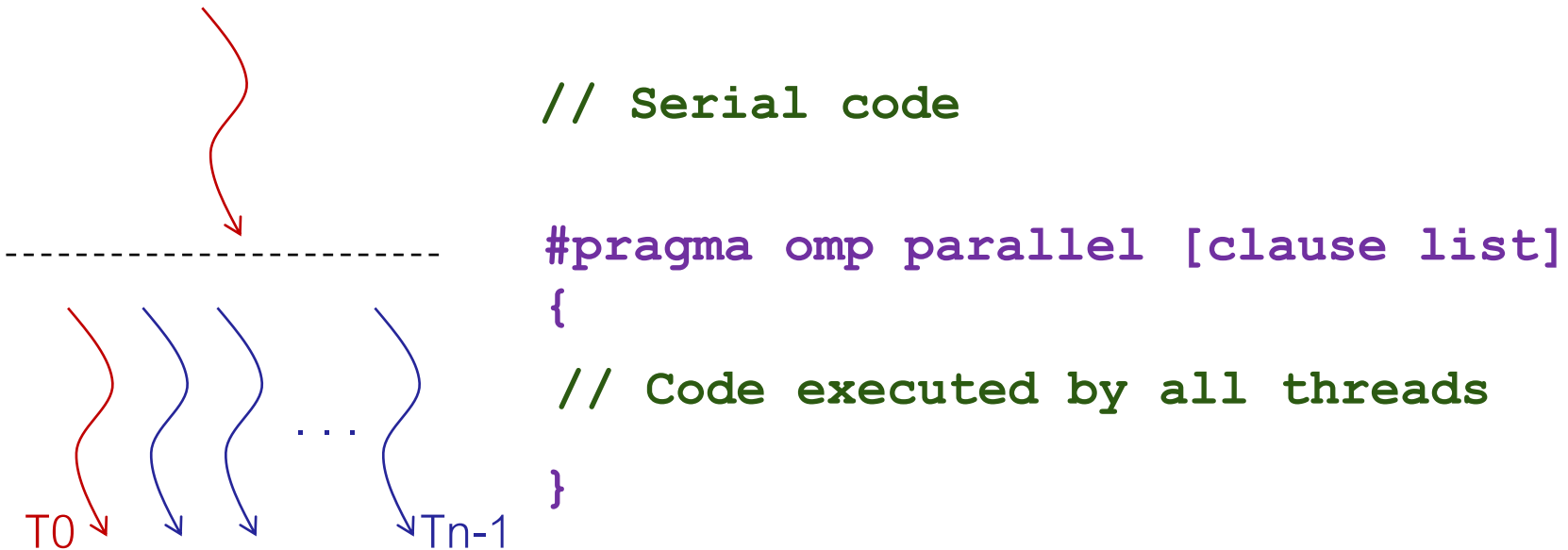
- Compile like a regular C program, link with omp library

gcc -fopenmp ... [-std=c99/gnu99]

- Debug with gdb and valgrind, or dedicated parallel debuggers e.g., TotalView, DDT, etc. DDT should be on Scinet.

Parallel directive

- Regular serial execution until it hits a `parallel` directive
 - Creates a group of threads, main thread becomes the *master* of the group



A first parallel program

- Default number of threads to be spawned is stored in the environment variable OMP_NUM_THREADS
 - Useful to set a default

```
int main(int argc, char *argv[]) {  
    printf("Starting a parallel region, spawning threads\n");  
  
    #pragma omp parallel  
    {  
        printf("Hello world\n");  
    }  
  
    return 0;  
}
```

```
$ export OMP_NUM_THREADS=8  
$ gcc -o hello hello.c -fopenmp
```

- All threads execute the exact same code from the parallel region

A first parallel program

- Parallel region spawns a block of code or a line (no braces needed for latter)

```
int main() {  
    printf("Starting a parallel region, spawning threads\n");  
  
    #pragma omp parallel  
    {  
        printf("Hello world\n");  
    }  
    printf("Bye world\n");  
  
    return 0;  
}
```

```
$ export OMP_NUM_THREADS=8  
$ gcc -o hello hello.c -fopenmp
```

If you do not set the number of threads, X number of threads will be automatically spawned in the parallel region, where X is the number of cores (or the number of threads with hyperthreading if enabled)

OpenMP: language extension + library

- Language extensions: `#pragma omp` (ignored if not compiled with `-fopenmp`), great feature for monitoring the behavior of the serial version of your code
- Library functions (must include `omp.h`, even if compiled with `-fopenmp`):

```
int omp_get_num_threads(); /* # of threads running when this is invoked */
                          /* refers to closest enclosing parallel block*/

void omp_set_num_threads(int n); /* # of threads to use in the next
                                parallel section */

int omp_get_max_threads(); /* max number of threads that can be created */

int omp_get_thread_num(); /* thread id in a group of threads */

int omp_get_num_procs(); /* number of processors available */

int omp_in_parallel(); /* non-zero if called within a parallel region */
```


A first parallel program

- Threads run in parallel - no guarantees about ordering

```
int main() {  
    printf("Starting a parallel region, spawning threads\n");  
  
    #pragma omp parallel  
    {  
        printf("Hello world, I am thread %d out of %d!\n",  
               omp_get_thread_num(),  
               omp_get_num_threads());  
    }  
  
    return 0;  
}
```

A first parallel program

- Threads run in parallel - no guarantees about ordering

```
int main() {  
    printf("Starting a parallel region, spawning threads\n");  
  
    #pragma omp parallel  
    {  
        printf("Hello world, I am thread %d out of %d running threads!\n",  
               omp_get_thread_num(),  
               omp_get_num_threads());  
    }  
    printf("There are %d threads running!\n", omp_get_num_threads());  
  
    return 0;  
}
```

- Notice anything?

A first parallel program

- Threads run in parallel - no guarantees about ordering

```
int main() {  
    printf("Starting a parallel region, spawning threads\n");  
  
    #pragma omp parallel  
    {  
        printf("Hello world, I am thread %d out of %d running threads!\n",  
               omp_get_thread_num(),  
               omp_get_num_threads());  
    }  
    printf("There are %d threads running!\n", omp_get_num_threads());  
  
    return 0;  
}
```

- Notice anything? The last printf is only visited by the main thread and prints
There are 1 threads running!

Parallel directive clauses

- Conditional Parallelization: `if` clause (only one!)

- Only create threads if an expression holds

```
#pragma omp parallel if(to_parallelize == 1)
```

- Degree of concurrency: `num_threads` clause

- How many threads to spawn, overrides the default OMP_NUM_THREADS

```
#pragma omp parallel num_threads(8)
```

- Data handling: `private/firstprivate/shared` clauses (different visibility)

```
#pragma omp parallel private(v1) shared(v2,v3) firstprivate(v4)
```

Up Next...

Variable semantics: shared, private, firstprivate

The reduction directive

The Omp for directive

Examples on the above

Variable semantics - summary

- Semantics of each variable
 - Shared: all threads share the same copy of the variable(s)
 - Private: variable(s) are local to each thread
 - Firstprivate: like private, but value of each copy is initialized to the value before the parallel directive

```
#pragma omp parallel private(v1) shared(v2,v3) firstprivate(v4)
```

Data sharing

- Consider this example of PRIVATE and FIRSTPRIVATE

```
variables: A = 1, B = 1, C = 1  
#pragma omp parallel private(B) firstprivate(C)
```

- Are A,B,C private to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

Data sharing

- Consider this example of PRIVATE and FIRSTPRIVATE

```
variables: A = 1, B = 1, C = 1  
#pragma omp parallel private(B) firstprivate(C)
```

- Are A,B,C private to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

Inside this parallel region ...

- "A" is shared by all threads; equals 1
- "B" and "C" are private to each thread.
 - B's initial value is undefined
 - C's initial value equals 1

Following the parallel region ...

- B and C revert to their original values of 1
- A is either 1 or the value it was set to inside the parallel region

Careful with variable state

- Think carefully about the intended variable visibility between threads
- Should a variable be shared or private?
 - Private: a thread can modify its own copy without impacting other threads
 - Shared: all threads see the exact same copy, tricky if the data is not read-only
- Dramatic impact on correctness if you get this wrong!

Default state

- Can specify default state using `default(...)` clause
 - `default(shared)` = by default, a variable is shared by all threads
 - `default(none)` = must explicitly specify how every single variable used in parallel region should be handled, otherwise compile errors raised
- If not explicitly indicated
 - Variables declared *within* a parallel block are *implicitly private*
 - Variables declared *outside* a parallel block become *shared* when parallel region starts (with some exceptions like *some* loop counters .. more later)
- Safer to use `default(none)` to **force you to specify intended visibility**
 - Recommended to **avoid possible bugs** from unintentionally sharing data

Reduction

- Important primitive – supported seamlessly using `reduction` clause
 - Multiple local copies of a variable are combined into a single copy at the master when the parallel block ends

`reduction(operator: variable list)`

- Operators: +, *, -, &, |, ^, &&, ||

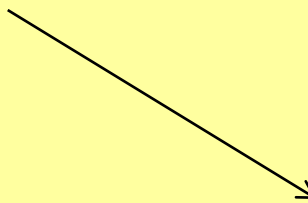
```
int s = 0;
#pragma omp parallel reduction(+:s) num_threads(8)
{
    // compute sum s
    s += ...
}
// variable s now has the sum of all local dim copies
```

Reduction: Simple example

- Calculate the sum of the thread ids

```
int sum = 0;
#pragma omp parallel reduction(+: sum) num_threads(8)
{
    int tid = omp_get_thread_num();

    sum += tid;
}
printf("Sum of thread ids = %d\n", sum);
```



*Each thread gets a
private sum copy*

- Simple but useful construct for more complex computations (as we'll see later ..)

Reduction: More complex example

- Calculate the dot product of two arrays 'a' and 'b' of length 'size'
- Can use the reduction clause for the parallel directive

```
int dotprod = 0;
#pragma omp parallel shared(size, a, b) \
    reduction(+: dotprod) num_threads(8)
{
    int nthr = omp_get_num_threads();
    int chunk = (size+nthr-1) / nthr;
    int tid = omp_get_thread_num();

    for(int i = tid*chunk;
        i < (tid+1)*chunk && i < size; i++) {
        dotprod += a[i] * b[i];
    }
}
```

*Each thread gets a
private dotprod copy*

*Loop counter i is
implicitly private*

- Notice that we have to explicitly partition the data, and make sure that each thread only operates on its assigned chunk

For loops in OpenMP

- Previous example: Every thread executes the same loop, just on different value ranges of a and b
- A simple example:

```
int i, size=8;
int *a = (int*)malloc(size*sizeof(int));
for(i = 0; i < size; i++) {
    a[i] = i+1;
}

#pragma omp parallel shared(size, a) private(i) num_threads(8)
{
    int tid = omp_get_thread_num();
    for(i = 0; i < size; i++) {
        printf("TID[%d] - a[%d] = %d\n", tid, i, a[i]);
    }
}
```

- Kind of useless work because each thread executes all iterations of the loop.
- What is the output?

For loops in OpenMP

- Previous example: Every thread executes the same loop, just on different value ranges of a and b
- A simple example:

```
int i, size=8;
int *a = (int*)malloc(size*sizeof(int));
for(i = 0; i < size; i++) {
    a[i] = i+1;
}

#pragma omp parallel shared(size, a) private(i) num_threads(8)
{
    int tid = omp_get_thread_num();
    for(i = 0; i < size; i++) {
        printf("TID[%d] - a[%d] = %d\n", tid,
    }
}
```

```
TID[1] - a[0] = 1
TID[1] - a[1] = 2
TID[1] - a[2] = 3
TID[0] - a[0] = 1
TID[0] - a[1] = 2
TID[0] - a[2] = 3
TID[0] - a[3] = 4
TID[0] - a[4] = 5
TID[0] - a[5] = 6
TID[0] - a[6] = 7
```

...

- Kind of useless work.

Concurrent tasks – loop scheduling

- Use `parallel` directive to create concurrency across iterations
 - Recall task parallelism!
- Then, automatically divvy up the iterations across threads using `for` directive

```
#pragma omp for [clause list]  
// for loop
```

- Clauses:
 - `private`, `firstprivate`, `lastprivate`
 - `reduction`
 - `schedule`
 - `nowait`
 - `ordered`
 - `private` and `firstprivate` – same semantics as for `parallel` directive
 - `lastprivate` handles writing back a single copy from multiple copies

Example: for directive

- A simple example:

```
int i, size=8;
int *a = (int*)malloc(size*sizeof(int));
for(i = 0; i < size; i++) {
    a[i] = i+1;
}

#pragma omp parallel shared(size, a) private(i) num_threads(8)
{
    #pragma omp for
    int tid = omp_get_thread_num();
    for(i = 0; i < size; i++) {
        printf("TID[%d] - a[%d] = %d\n", tid, i, a[i]);
    }
}
```

- What will this print?

Example: for directive

- A simple example:

```
int i, size=8;
int *a = (int*)malloc(size*sizeof(int));
for(i = 0; i < size; i++) {
    a[i] = i+1;
}

#pragma omp parallel shared(size, a) private(i) num_threads(8)
{
    #pragma omp for
    int tid = omp_get_thread_num();
    for(i = 0; i < size; i++) {
        printf("TID[%d] - a[%d] = %d\n", tid, i, a[i]);
    }
}
```

Compiler error



- What will this print?

Example: for directive

- A simple example:

```
int i, size=8;
int *a = (int*)malloc(size*sizeof(int));
for(i = 0; i < size; i++) {
    a[i] = i+1;
}

#pragma omp parallel shared(size, a) private(i) num_threads(8)
{
    int tid = omp_get_thread_num();
    #pragma omp for
    for(i = 0; i < size; i++) {
        printf("TID[%d] - a[%d] = %d\n", tid, i, a[i]);
    }
}
```

- What will this print?

Example: for directive

- A simple example:

```
int i, size=8;
int *a = (int*)malloc(size*sizeof(int));
for(i = 0; i < size; i++) {
    a[i] = i+1;
}

#pragma omp parallel shared(size, a) private(i) num_threads(8)
{
    int tid = omp_get_thread_num();
    #pragma omp for
    for(i = 0; i < size; i++) {
        printf("TID[%d] - a[%d] = %d\n", tid, i, a[i]);
    }
}
```

- What will this print?
 - Loop iterations partitioned across threads now
 - Each thread takes care of different iterations

```
TID[1] - a[1] = 2
TID[7] - a[7] = 8
TID[4] - a[4] = 5
TID[6] - a[6] = 7
TID[3] - a[3] = 4
TID[5] - a[5] = 6
TID[0] - a[0] = 1
TID[2] - a[2] = 3
```

How omp for makes loop parallelism easy

An example to how how parallelizing a loop can be made easy with **omp for**

(A) Sequential code

```
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

(B) OpenMP parallel region

```
#pragma omp parallel
```

```
{
```

```
    int id, i, Nthrds, istart, iend;
```

```
    id = omp_get_thread_num();
```

```
    Nthrds = omp_get_num_threads();
```

```
    istart = id * N / Nthrds;
```

```
    iend = (id+1) * N / Nthrds;
```

```
    if (id == Nthrds-1) iend = N;
```

```
    for(i=istart;i<iend;i++) { a[i] = a[i] + b[i];}
```

```
}
```

(C) OpenMP parallel region and a worksharing for construct replaces (B)

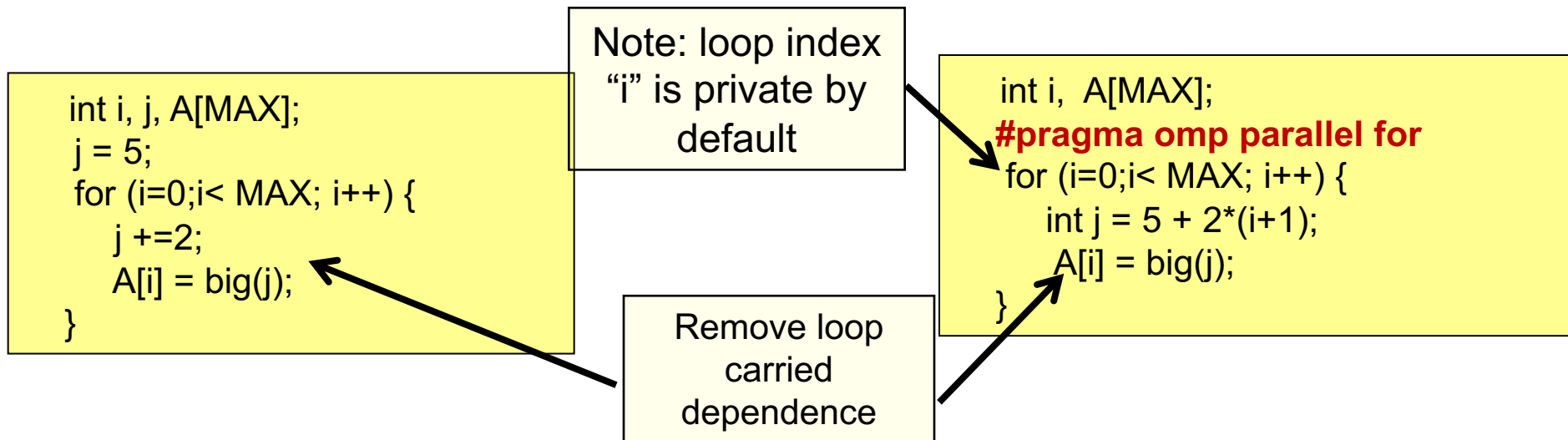
```
#pragma omp parallel
```

```
#pragma omp for
```

```
    for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

Basic approach for working with loops

- Basic approach
 - Find compute intensive loops
 - Make the loop iterations independent ... So they can safely execute in any order without loop-carried dependencies
 - Place the appropriate OpenMP directive and test



More examples: private clause

- `private(var)` creates a new local copy of `var` for each thread.
 - The value of the private copies is uninitialized
 - The value of the original variable is unchanged after the region

```
void wrong() {  
    int tmp = 0;  
    #pragma omp parallel for private(tmp)  
        for (int j = 0; j < 1000; ++j)  
            tmp += j;  
    printf("%d\n", tmp);  
}
```

tmp was not
initialized

tmp is 0 here

More examples: Firstprivate clause

- Variables initialized from a shared variable

```
incr = 0;  
#pragma omp parallel for firstprivate(incr)  
for (i = 0; i <= MAX; i++) {  
    if ((i%2)==0) incr++;  
    A[i] = incr;  
}
```

Each thread gets its own copy of incr
with an initial value of 0

More examples: default clause

- You can put the default clause on parallel and parallel + workshare constructs.

```
#include <omp.h>
int main()
{
    int i, j=5;    double x=1.0, y=42.0;
    #pragma omp parallel for default(none) reduction(*:x)
    for (i=0;i<10;i++){
        for(j=0; j<3; j++)
            x+= foobar(i, j, y);
    }
    printf(" x is %f\n",(float)x);
}
```

What does the compiler
complain about?

More examples: default clause

- You can put the default clause on parallel and parallel + workshare constructs.

```
#include <omp.h>
int main()
{
    int i, j=5;    double x=1.0, y=42.0;
    #pragma omp parallel for default(none) reduction(*:x)
    for (i=0; i<10; i++){
        for(j=0; j<3; j++)
            x+= foobar(i, j, y);
    }
    printf(" x is %f\n", (float)x);
}
```

The compiler would complain about j and y, which is important since you don't want j to be shared

Loop counter i is implicitly private

The reduction also implicitly defines the state of x