CS420 – Lecture 5

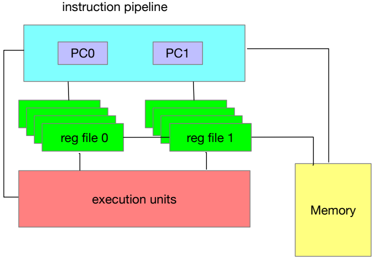
Marc Snir

Fall 2018

Multithreading

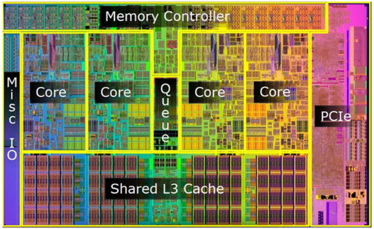
Why Multithreading?

- *Thread of execution:* The execution of a sequence of instructions.
- Notwithstanding caches, a thread may often wait for slow memory accesses, idling the ALUs
- ⇒ Can improve ALU utilization by having it shared by multiple hardware threads: *simultaneous multithreading* (SMT).
- Modern processors support 2-4 HW threads per core
- User can control how many HW threads use a core
- Higher thread count good for memory latency bound executions
- Lower thread count can work better for memory bandwidth bound executions



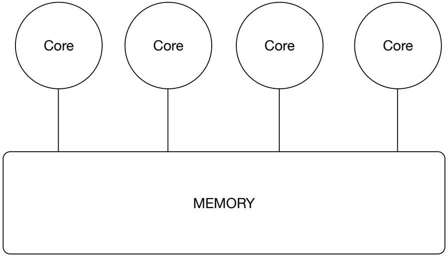
The OS may handle a large number of concurrent SW threads, but only one per HW thread can run simultaneously. one at a time

- Run out of opportunities to enhance single CPU performance, and have room for more transistors on chip
- $\Rightarrow$ Can put multiple cores on a chip

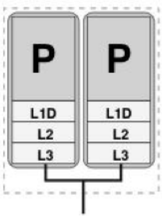


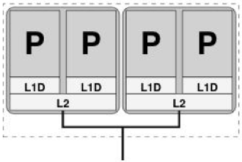
Multicore processors

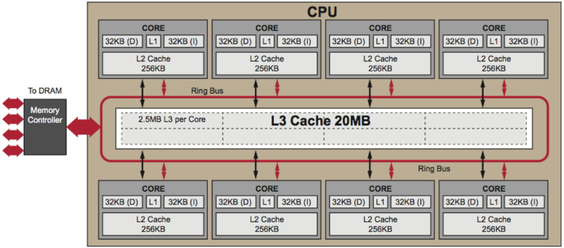
- Multiple cores each running multiple hardware threads connected to common shared memory
- Each hardware thread executes its own program
- All cores can read and write shared variables in shared memory
- Each thread can run an independent program; but if we want to speed up a computation, the threads need to collaborate on one computation.



Can be private or shared, too



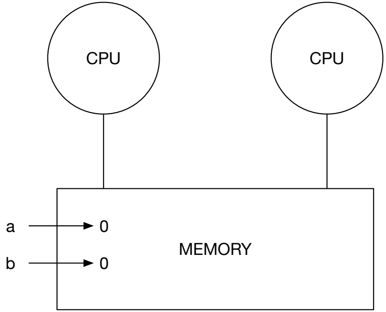




Sharing without caches

Thread 0	Thread 1
a=5;	...
barrier;	barrier;
...	b=a

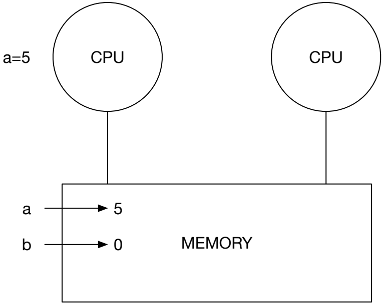
No caches



Sharing without caches

```
Thread 0          Thread 1
a=5;              ...
barrier;          barrier;
...                 b=a
```

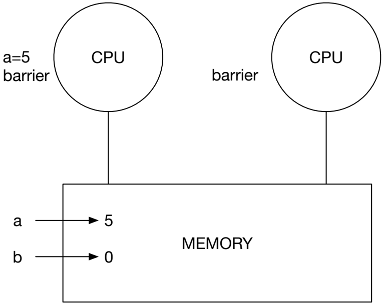
No caches



Sharing without caches

```
Thread 0          Thread 1
a=5;              ...
barrier;          barrier;
...                 b=a
```

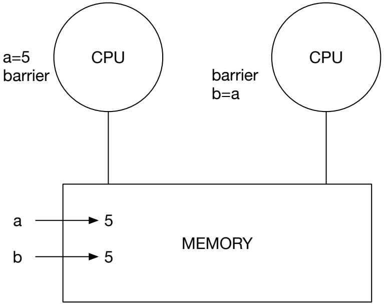
No caches



Sharing without caches

```
Thread 0        Thread 1
a=5;            ...
barrier;        barrier;
...                 b=a
```

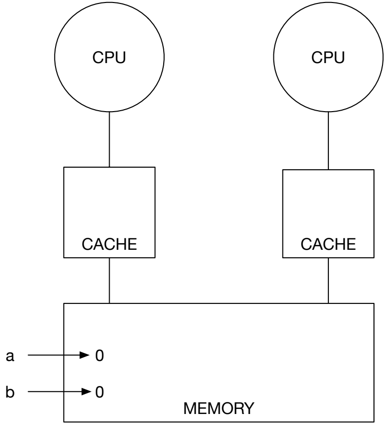
No caches



Problem with caches

```
Thread 0         Thread 1
a=5;             ...
barrier;         barrier;
...                  b=a
```

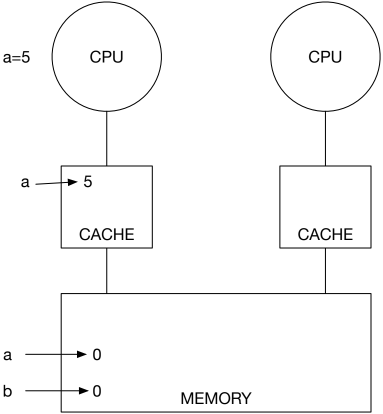
With caches



Problem with caches

```
Thread 0          Thread 1
a=5;              ...
barrier;          barrier;
...                  b=a
```

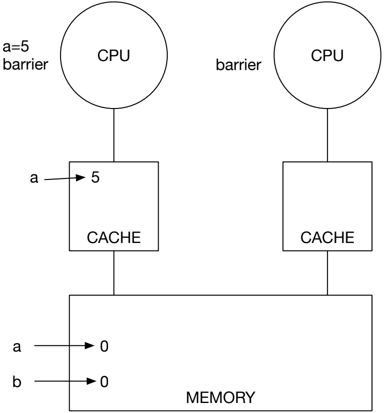
With caches



Problem with caches

Thread 0	Thread 1
a=5;	...
barrier;	barrier;
...	b=a

With caches

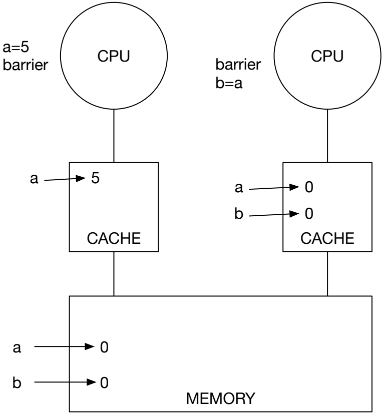


Problem with caches

```
Thread 0        Thread 1
a=5;            ...
barrier;        barrier;
...                 b=a
```

Caches are not consistent – wrong result!

With caches



Coherence protocol

Protocol that ensure that all caches have same value for variable cached in multiple caches
MESI protocol (AKA *Illinois protocol*):
A cache line can have four states:

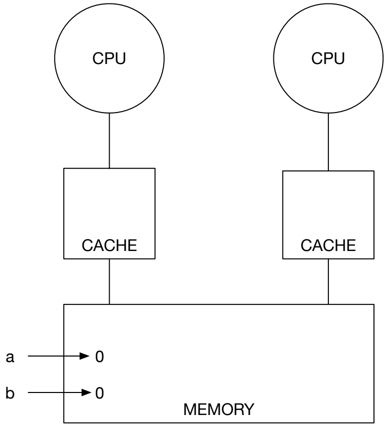
- Modified: Only in that cache and different from memory value (Need to be written back to memory, if evicted)
- Exclusive: Only in that cache (can be modified)
- Shared: In other caches as well (cannot be modified)
- Invalid: Invalid

More states can be used to further reduce coherence traffic

Snoopy protocol

All caches "snoop" on memory accesses and intervene, if needed

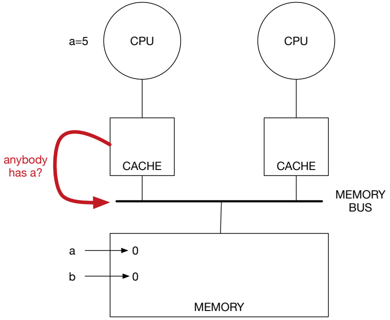
```
Thread 0          Thread 1
a=5;              ...
barrier;          barrier;
...                   b=a
```



Snoopy protocol

All caches "snoop" on memory accesses and intervene, if needed

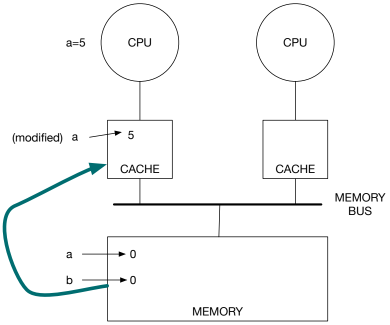
```
Thread 0          Thread 1
a=5;              ...
barrier;          barrier;
...                  b=a
```



Snoopy protocol

All caches “snoop” on memory accesses and intervene, if needed

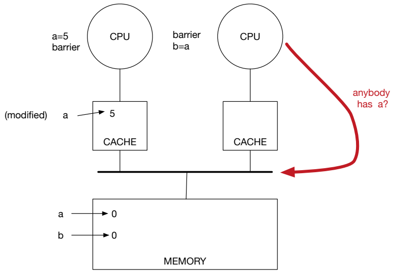
```
Thread 0        Thread 1
a=5;            ...
barrier;        barrier;
...               b=a
```



Snoopy protocol

All caches "snoop" on memory accesses and intervene, if needed

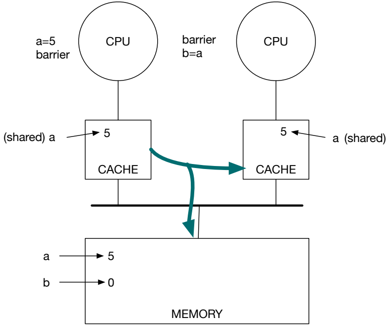
```
Thread 0        Thread 1
a=5;            ...
barrier;        barrier;
...               b=a
```



Snoopy protocol

All caches "snoop" on memory accesses and intervene, if needed

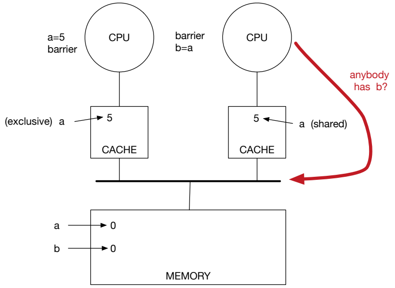
```
Thread 0        Thread 1
a=5;            ...
barrier;        barrier;
...               b=a
```



Snoopy protocol

All caches "snoop" on memory accesses and intervene, if needed

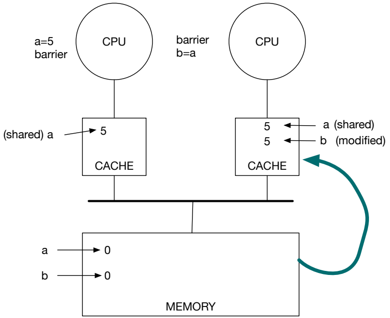
```
Thread 0        Thread 1
a=5;            ...
barrier;        barrier;
...                b=a
```



Snoopy protocol

All caches "snoop" on memory accesses and intervene, if needed

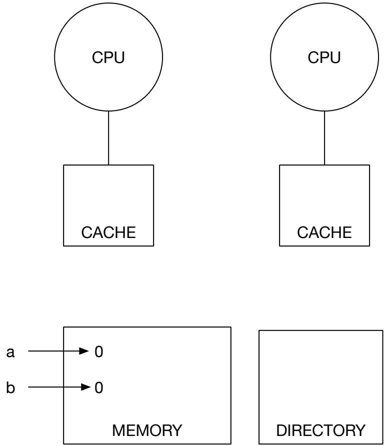
```
Thread 0          Thread 1
a=5;              ...
barrier;          barrier;
...                   b=a
```



Directory protocol

A (logical central) directory keeps track which lines are in which cache

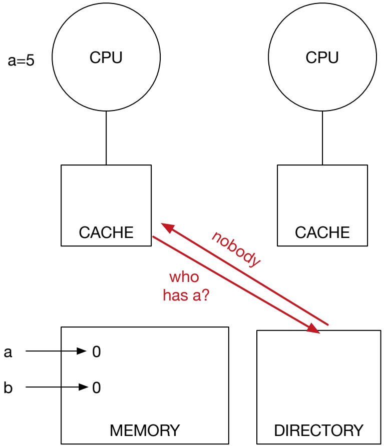
```
Thread 0          Thread 1
a=5;              ...
barrier;          barrier;
...                   b=a
```



Directory protocol

A (logical central) directory keeps track which lines are in which cache

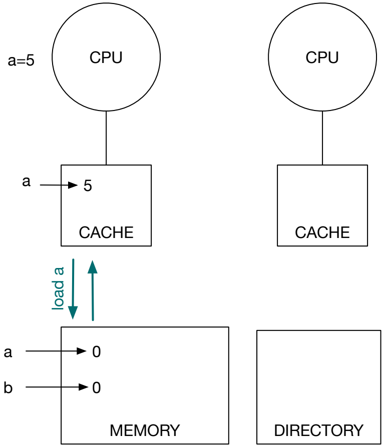
```
Thread 0          Thread 1
a=5;              ...
barrier;          barrier;
...                  b=a
```



Directory protocol

A (logical central) directory keeps track which lines are in which cache

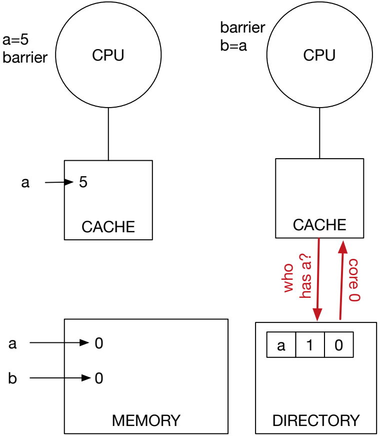
```
Thread 0          Thread 1
a=5;              ...
barrier;          barrier;
...                  b=a
```



Directory protocol

A (logical central) directory keeps track which lines are in which cache

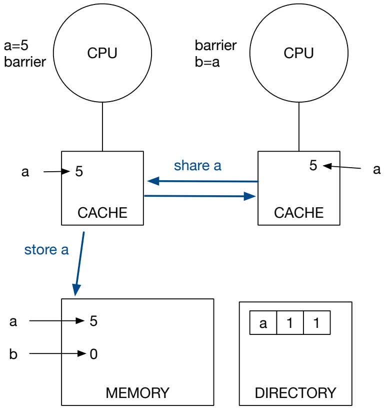
```
Thread 0          Thread 1
a=5;              ...
barrier;          barrier;
...                   b=a
```



Directory protocol

A (logical central) directory keeps track which lines are in which cache

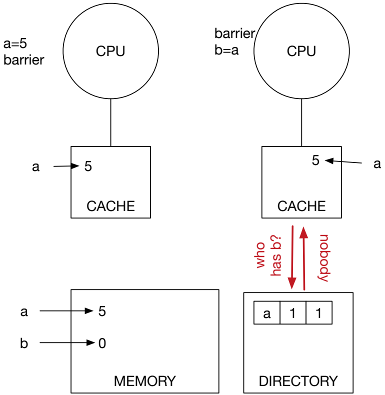
```
Thread 0          Thread 1
a=5;              ...
barrier;          barrier;
...                 b=a
```



Directory protocol

A (logical central) directory keeps track which lines are in which cache

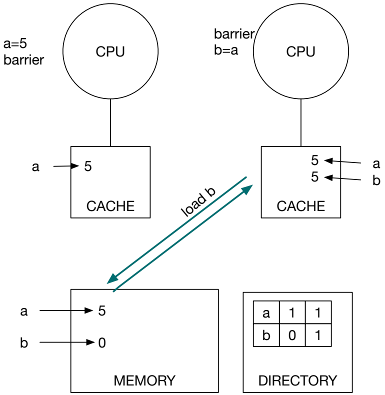
```
Thread 0          Thread 1
a=5;              ...
barrier;          barrier;
...                   b=a
```



Directory protocol

A (logical central) directory keeps track which lines are in which cache

```
Thread 0          Thread 1
a=5;              ...
barrier;          barrier;
...                  b=a
```



Snooping vs. directory

Snooping

- Good: Lower latency – fewer communications per transaction
- Bad: All caches are involved in all memory accesses – does not scale

Directory

- Bad: Higher latency – more communications per transaction
- Good: Scales: Fixed amount of traffic per transaction

Variants of snooping used for low core counts and directories used for high core counts

False sharing

Coherence protocol works on full cache lines, not individual words

```
Thread 0        Thread 1
while(1)       while(1)
  a++;         b++;
```

Case 1: `a` and `b` are in distinct cache lines; there is no coherence traffic

False sharing

Coherence protocol works on full cache lines, not individual words

```
Thread 0        Thread 1
while(1)       while(1)
  a++;         b++;
```

Case 1: `a` and `b` are in distinct cache lines; there is no coherence traffic

Case 2: `a` and `b` happen to be in same cache line; cache line "ping-pongs" from one cache to the other

False sharing

Coherence protocol works on full cache lines, not individual words

```
Thread 0        Thread 1
while(1)        while(1)
  a++;            b++;
```

Case 1: `a` and `b` are in distinct cache lines; there is no coherence traffic

Case 2: `a` and `b` happen to be in same cache line; cache line "ping-pongs" from one cache to the other

Keep variables mostly used by different cores in distinct cache lines

Programming with threads

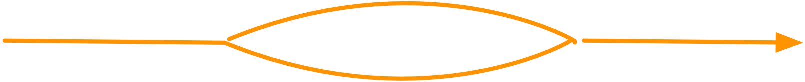
Use a thread library, such as Posix Pthreads library

Thread creation `pthread_create(&thread, atttr, function, arg)`

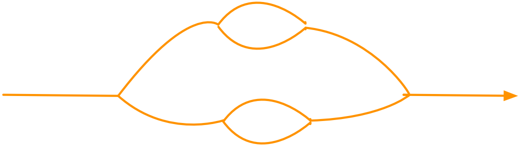
```
pthread_exit(status)
pthread_cancel(thread)
```

thread synchronization `pthread_join(thread,&status)`

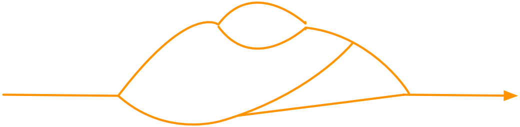
Fork-join model



Well-nested code: Forking thread Join in reverse order of forks



Spaghetti code



Further synchronization – locks

```
pthread_mutex_init
pthread_mutex_lock
pthread_mutex_unlock
pthread_mutex_destroy
```

```
Thread 0        Thread 1
Lock             Lock
A=2              A=A+1
Unlock          Unlock
```

Mutual exclusion – but order is unknown; program is non-deterministic!

Further synchronization – locks

```
pthread_mutex_init
pthread_mutex_lock
pthread_mutex_unlock
pthread_mutex_destroy
```

Thread 0	Thread 1
Lock	Lock
A=2	A=A+1
Unlock	Unlock

Mutual exclusion – but order is unknown; program is non-deterministic!
Final value of `A` is either 2 or 3

Why lock?

Thread 0	Thread 1
(2) str #2 A	(1) ldr r1 A
	(3) add r1 r1 #1
	(4) str r1 A

Why lock?

```
Thread 0                 Thread 1
(2)   str #2  A        (1) ldr r1 A
                       (3) add r1 r1 #1
                       (4) str r1 A
```

- Without lock, final result of `A`=1 is possible

Why lock?

Thread 0	Thread 1
(2) str #2 A	(1) ldr r1 A
	(3) add r1 r1 #1
	(4) str r1 A

- Without lock, final result of `A`=1 is possible
- Lock ensures atomicity of `A=A+1`

Ordering synchronization – condition variables

```
pthread_cond_init(&condition, attr)  pthread_cond_destroy(condition)
pthread_cond_wait(condition,mutex)
pthread_cond_signal(condition)  (pthread_cond_broadcast)
```

Thread 0	Thread 1
A=2	Wait
Signal	A=A+1

Final result will be `A=3`

Programming with threads – Use a language that supports multithreading

C++, Java,...

```
static const int num_threads = 10;
void *call_from_thread(void *) {
...
}
int main() {
std::thread t[num_threads];
//Launch a group of threads
for (int i = 0; i < num_threads; ++i) {
 t[i] = std::thread(call_from_thread);
}
std::cout << "Launched from the main\n";
//Join the threads with the main thread
for (int i = 0; i < num_threads; ++i) {
 t[i].join();
}
```

OpenMP – Open Multi-Processing

- Language developed by committee – Architecture Review Board (ARB)
- OpenMP V1.0 came out October 1997.
- Current version is V4.5 (your compiler may be V4.0)
- Mostly used in scientific computing (provides better control of how hardware threads are used)
- Exists as Fortran or as C/C++ extension

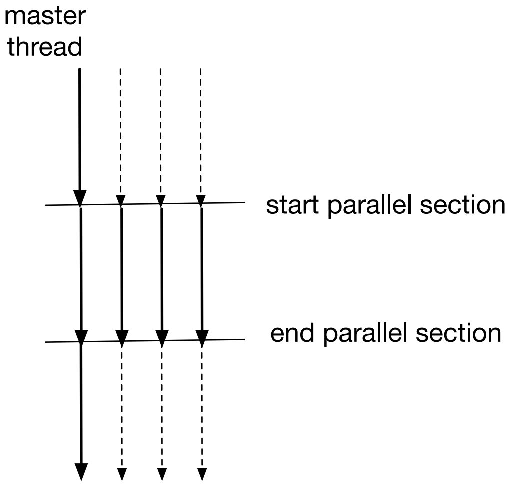
OpenMP

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

int main(int argc, char** argv) {

 omp_set_num_threads(4);
 printf("running with  max %d threads \n", omp_get_max_threads());
 printf("master is thread number %d of %d \n",
 omp_get_thread_num(), omp_get_num_threads());
#pragma omp parallel
  {
  printf("thread number %d of %d says hello world \n",
  omp_get_thread_num(), omp_get_num_threads());
 }
}
```

- Code consists of C/C++ (or Fortran), augmented with *pragmas* (or *directives*)
- It is executed by a *team* of threads
- Thread 0 starts execution; when the `parallel` statement is encountered, then each thread in the team executes the ensuing block; thread 0 resumes afterward



Output

```
running with  max 4 threads
master is thread number 0 of 1
thread number 0 of 4 says hello world
thread number 2 of 4 says hello world
thread number 1 of 4 says hello world
thread number 3 of 4 says hello world
```

- The number of threads in the team may exceed the number of hardware threads
- The calls to `printf()` will occur in an arbitrary order

How many threads in the team?

- Can be set as an environment variable, or by default
- Can be fixed (static) or varying (dynamic) – but does not change during the execution of a parallel section
- Good code is written to work with an arbitrary number of threads (possibly constant).
- In particular, code should work with pragmas ignored (one thread)

Example

- Sum squares of numbers from 1 to N

```
#include<omp.h>
#include <stdio.h>
...
int sum;
#pragma omp parallel
{
 int myid,numthreads,mysum,first ,next ,i;
 myid = omp_get_thread_num();
 numthreads=omp_get_num_threads();
 mysum=0;
 first=myid*N/numthreads+1;
 next=(myid+1)*N/numthreads+1;
 for(i=first;i<next;i++)
  mysum+=i*i;
 sum += mysum;
}
```

Example

- Sum squares of numbers from 1 to N
- Smart way: $sum = n(n+1)(2n+1)/6$

```
#include<omp.h>
#include <stdio.h>
...
int sum;
#pragma omp parallel
{
 int myid,numthreads,mysum,first ,next ,i;
 myid = omp_get_thread_num();
 numthreads=omp_get_num_threads();
 mysum=0;
 first=myid*N/numthreads+1;
 next=(myid+1)*N/numthreads+1;
 for(i=first;i<next;i++)
  mysum+=i*i;
 sum += mysum;
}
```

Example

- Sum squares of numbers from 1 to N
- Smart way: $sum = n(n+1)(2n+1)/6$
- Dumb way:
  - Split the range $1..N$ across threads
  - Have each thread sum its squares
  - Sum the results

```
#include<omp.h>
#include <stdio.h>
...
int sum;
#pragma omp parallel
{
  int myid,numthreads,mysum,first ,next ,i;
  myid = omp_get_thread_num();
  numthreads=omp_get_num_threads();
  mysum=0;
  first=myid*N/numthreads+1;
  next=(myid+1)*N/numthreads+1;
  for(i=first;i<next;i++)
    mysum+=i*i;
  sum += mysum;
}
```

```c
#include<omp.h>
#include <stdio.h>
...
int sum;
#pragma omp parallel
{
 int myid,numthreads,mysum,first ,next ,i;
 myid = omp_get_thread_num();
 numthreads=omp_get_num_threads();
 mysum=0;
 first=myid*N/numthreads+1;
 next=(myid+1)*N/numthreads+1;
 for(i=first;i<next;i++)
  mysum+=i*i;
 sum += mysum;
}
...
```

N=10 and team has 2 threads

1 2 3 4	5 6 7 8 9

thread 0	first=1	next=6
thread 1	first=6	next=11

```
...
int sum;
#pragma omp parallel
{
 int myid,numthreads,
 mysum,first ,next ,i;
...
}
...
```

- `sum` is a *shared* variable: all threads can access it
- `myid, numthreads...` are *private* variables: Each thread has its own copy.
- (This follows the usual scoping rules of C/C++)

Code is Wrong!

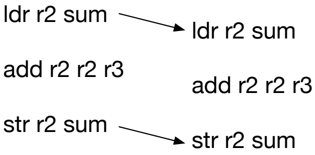
```
int sum;
#pragma omp parallel
{
 int myid,numthreads,mysum,first ,next ,i;
 myid = omp_get_thread_num();
 numthreads=omp_get_num_threads();
 mysum=0;
 first=myid*N/numthreads+1;
 next=(myid+1)*N/numthreads+1;
 for(i=first;i<next;i++)
  mysum+=i*i;
 sum += mysum;
}
```

For N=3 the program computed wrong sum=3270

Races

```
Thread 0                    Thread 1

ldr     r2  sum            load    r2  sum
add     r2  r2  r3         add     r2  r2  r3
str     r2  sum            str     r2  sum
```



- *The final value of* sum *need not be the sum of all the added local sums!*
- Need to ensure that the increments be *atomic*

```
...
int sum;
#pragma omp parallel
{
  int myid,numthreads,mysum,first ,next,i;
  myid = omp_get_thread_num();
  numthreads=omp_get_num_threads();
  mysum=0;
  first=myid*N/numthreads+1;
  next=(myid+1)*N/numthreads+1;
  for(i=first;i<next;i++)
   mysum+=i*i;
#pragma omp atomic
  sum += mysum;
}
...
```

The accesses to variable `sum` are atomic; the outcome is as if the variable is it read and written back by each thread in turn, in some order

Alternative

```
...
int sum;
#pragma omp parallel
{
 int myid,numthreads,mysum,first ,next,i;
 myid = omp_get_thread_num();
 numthreads=omp_get_num_threads();
 mysum=0;
 first=myid*N/numthreads+1;
 next=(myid+1)*N/numthreads+1;
 for(i=first;i<next;i++)
 mysum+=i*i;
#pragma omp critical
 sum += mysum;
}
...
```

- `atomic` can be used on for operations of the form `share_var = shared_var op val`
- `critical` ensures that the entire code within the critical section is executed atomically; the outcome is as if the section are executed by each thread in turn, in some order
- `atomic` can be significantly faster than `critical`, but is less general

Yet another way

```
...
int N,sum;
...
#pragma omp parallel reduction(+:sum)
{
 int myid,numthreads,first,next,i;
 myid = omp_get_thread_num();
 numthreads=omp_get_num_threads();
 first=myid*N/numthreads+1;
 next=(myid+1)*N/numthreads+1;
 for(i=first;i<next;i++)
 sum+=i*i;
}
...
```

- Each thread gets a copy of `sum`, initialized to 0
- when the threads exit the parallel section, the global variable `sum` is set to the sum of the local copies
- works for *, &, &&, max, min... (with the suitable initial value)

Data sharing

- *reduction* reduces values of copies to one master value at exit from parallel section
- Can also broadcast the master value to the local copies at the entry to the parallel section

```
int first=3;
omp_set_num_threads(2);
#pragma omp parallel firstprivate(first)
{
 int myid = omp_get_thread_num();
 printf("at thread %d, first=%d,
     \n", myid, first);
 first=myid;
}
 printf("when done first=%d \n", first);
```

output is:

```
at thread 0, first=3,
at thread 1, first=3,
when done first=3
```