SimCenter NHERI
Center for Computational Modeling and Simulation

2020 Programming Bootcamp

Parallel Programming - OpenMP

Frank McKenna
University of California at Berkeley

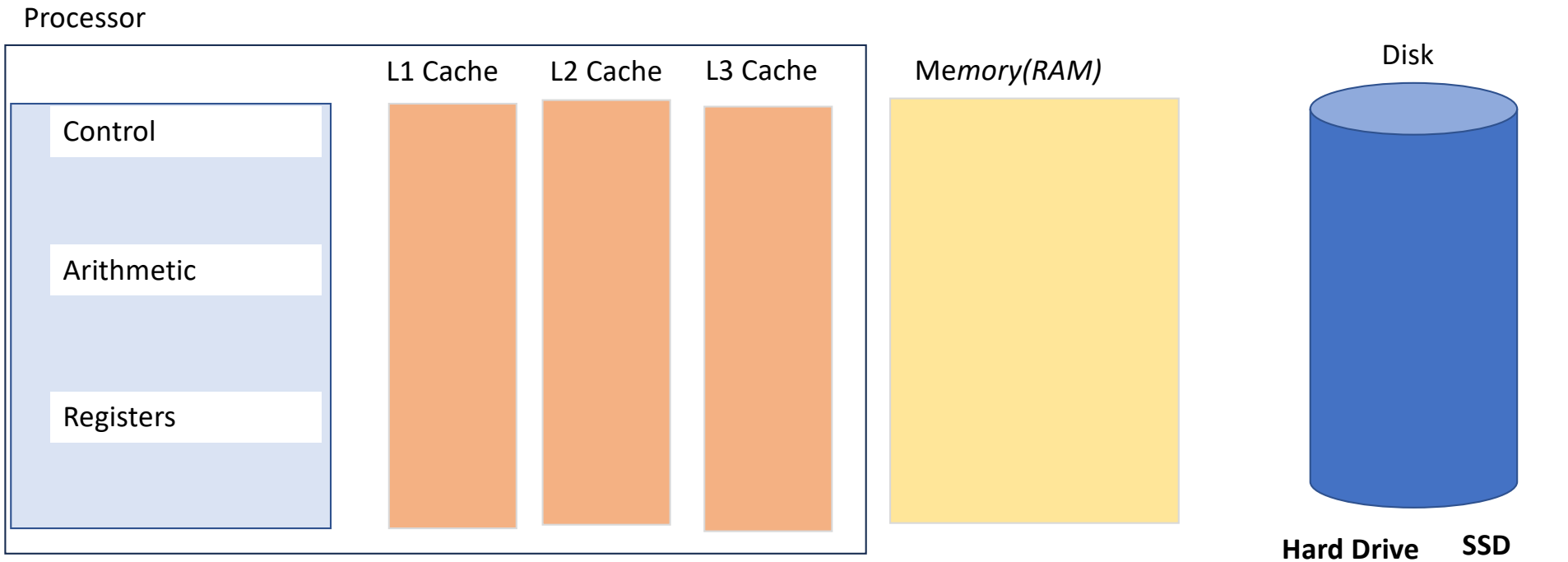
NSF award: CMMI 1612843

Did you remember?

```
git fetch upstream
git merge upstream/master
```

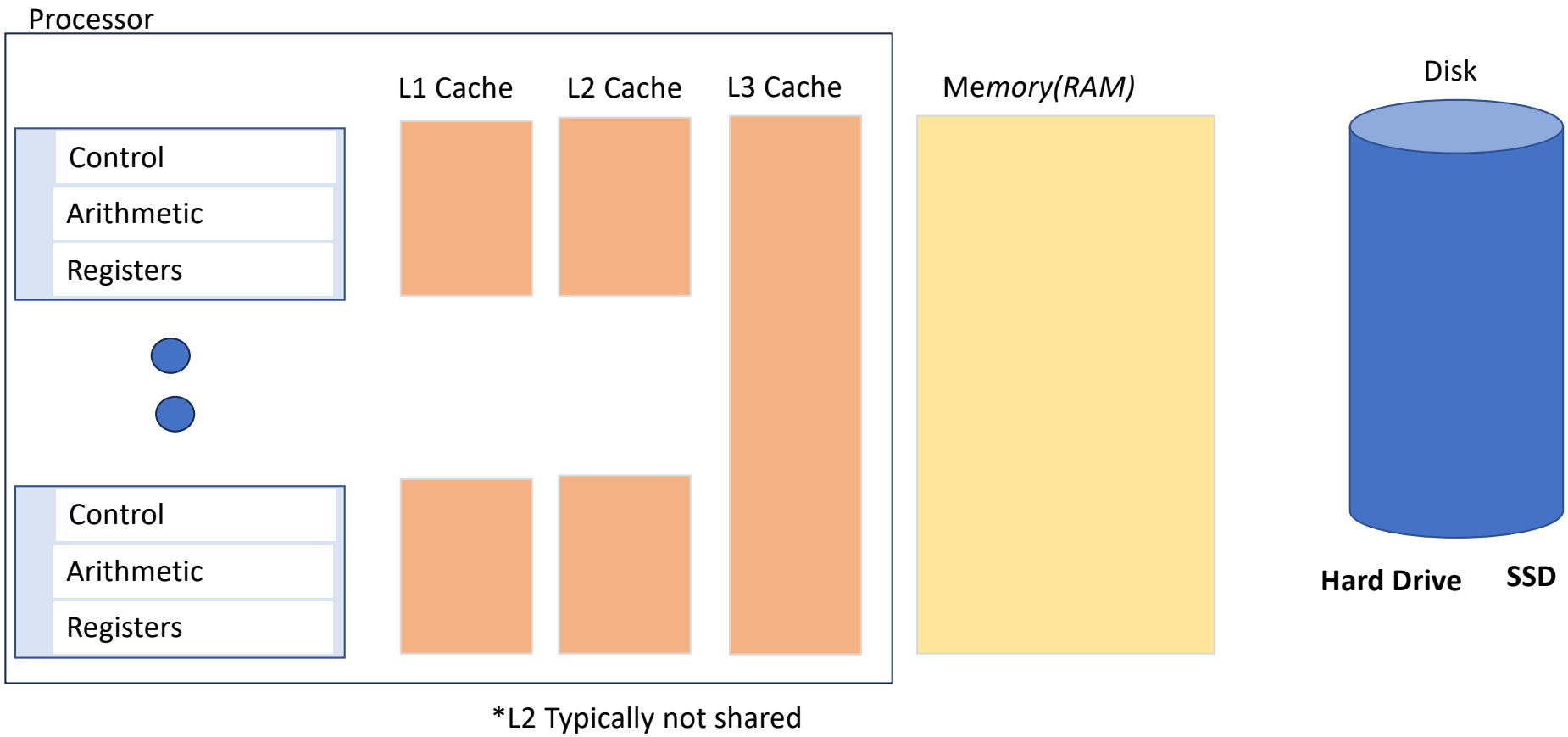
```
idev
```

Idealized Processor Model

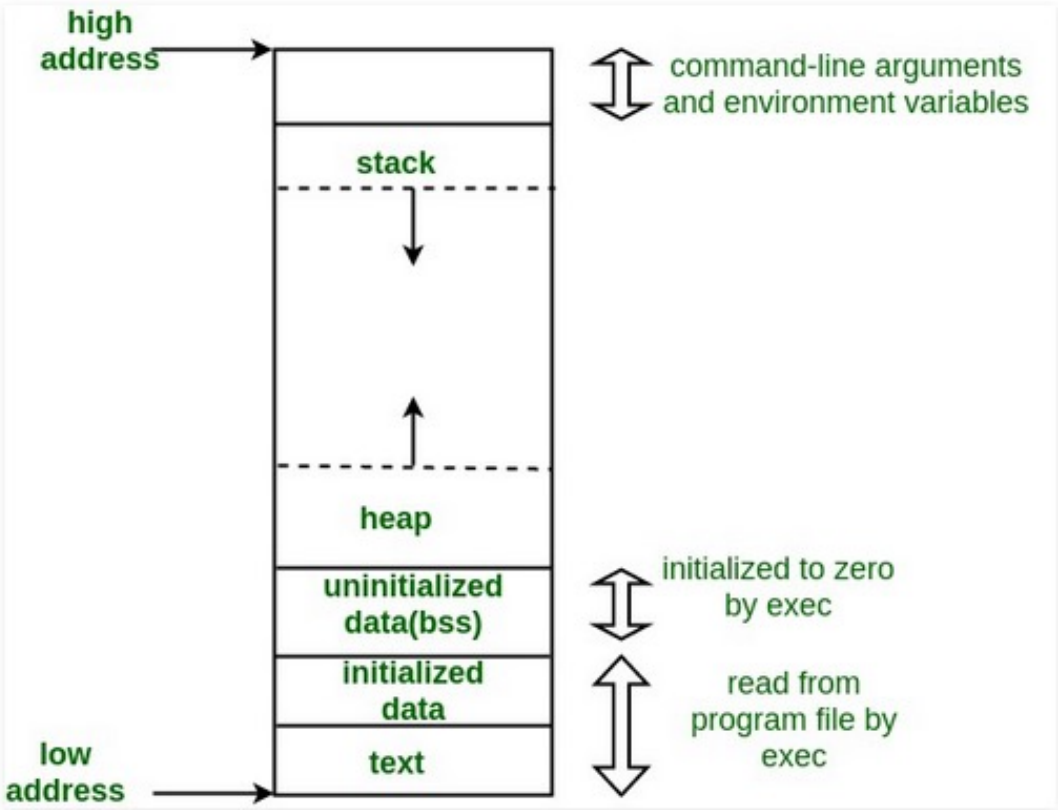


Size	1000 Bytes	64 KB	256 KB	2-4 MB	4-32 GB	4-16 TB	*.25-1TB*
Latency	0.3 ns	1 ns	3-10 ns	20-30 ns	50-100 ns	5-10e6 ns	25-50e3 ns
	Compiler	HW	HW	HW	Operating System	Operating System	

For Threaded Programming Let Us Revise

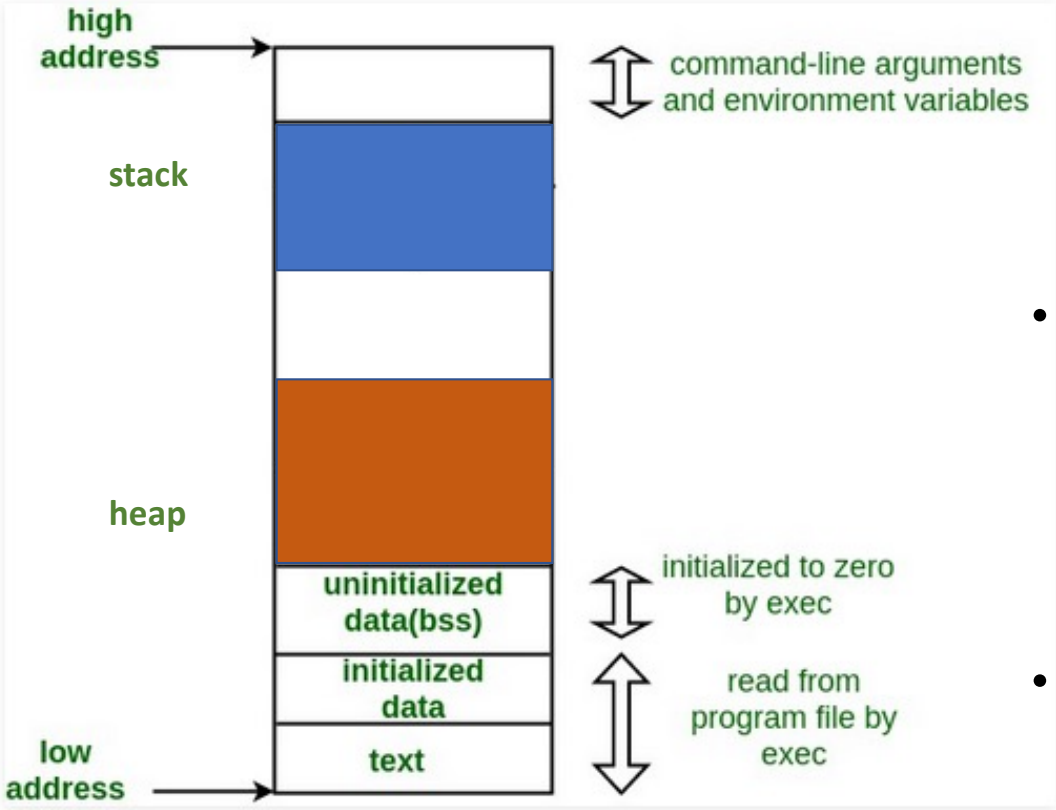


Process



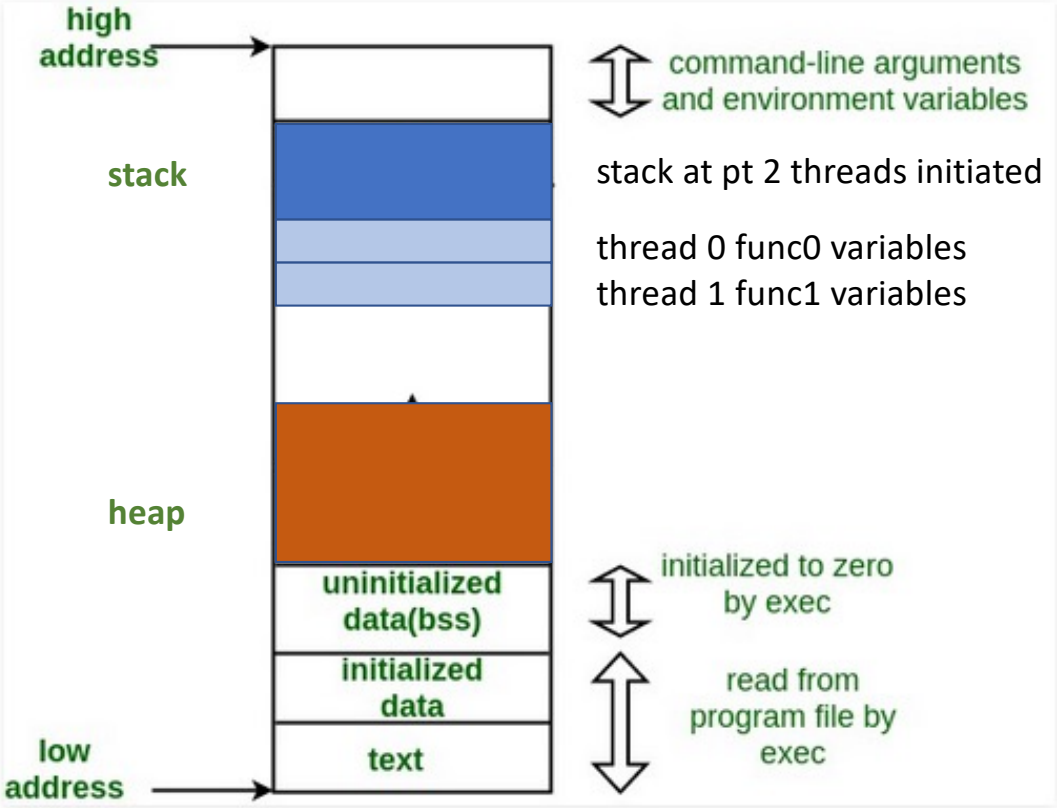
- An instance of a program execution.
- A process executes a program, you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.
- you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.
- The execution context of a running program, i.e. resources associated with program, current state of memory, current instruction being executed, pc.

Process:



State of Memory at some point during program execution

- An instance of a program execution.
- A process executes a program, you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.
- you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.
- The execution context of a running program, i.e. resources associated with program, current state of memory, current instruction being executed, pc.

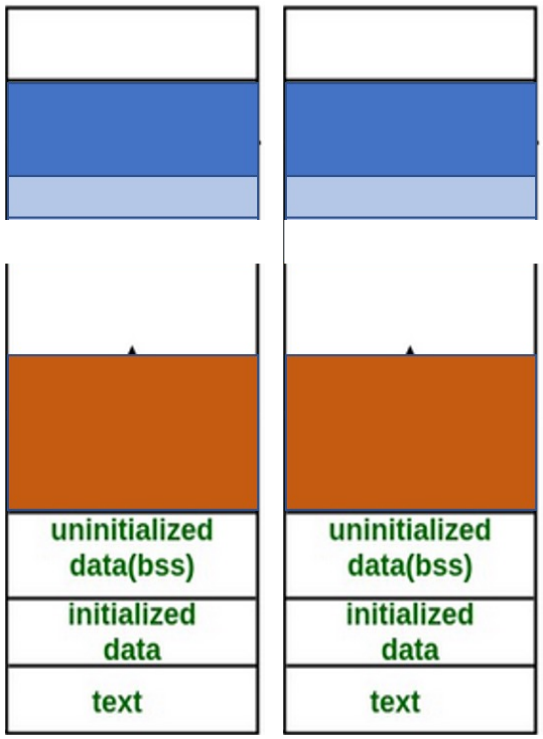


THREAD:

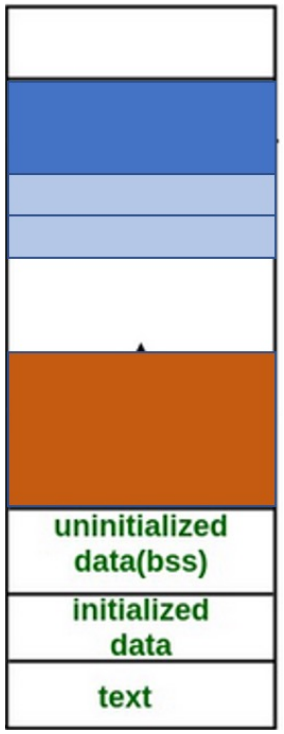
- A light weight process.
- Thread shares the Process state among multiple threads, has unique stack, shares contents of stack with other share heap.

State of Memory at point 2 threads are created (thread 0 in func0 with variables for func0, And thread1 in func1 with variables for func1. threads 0 and 1 share stack at point variables created, and share heap.

MPI v OpenMP

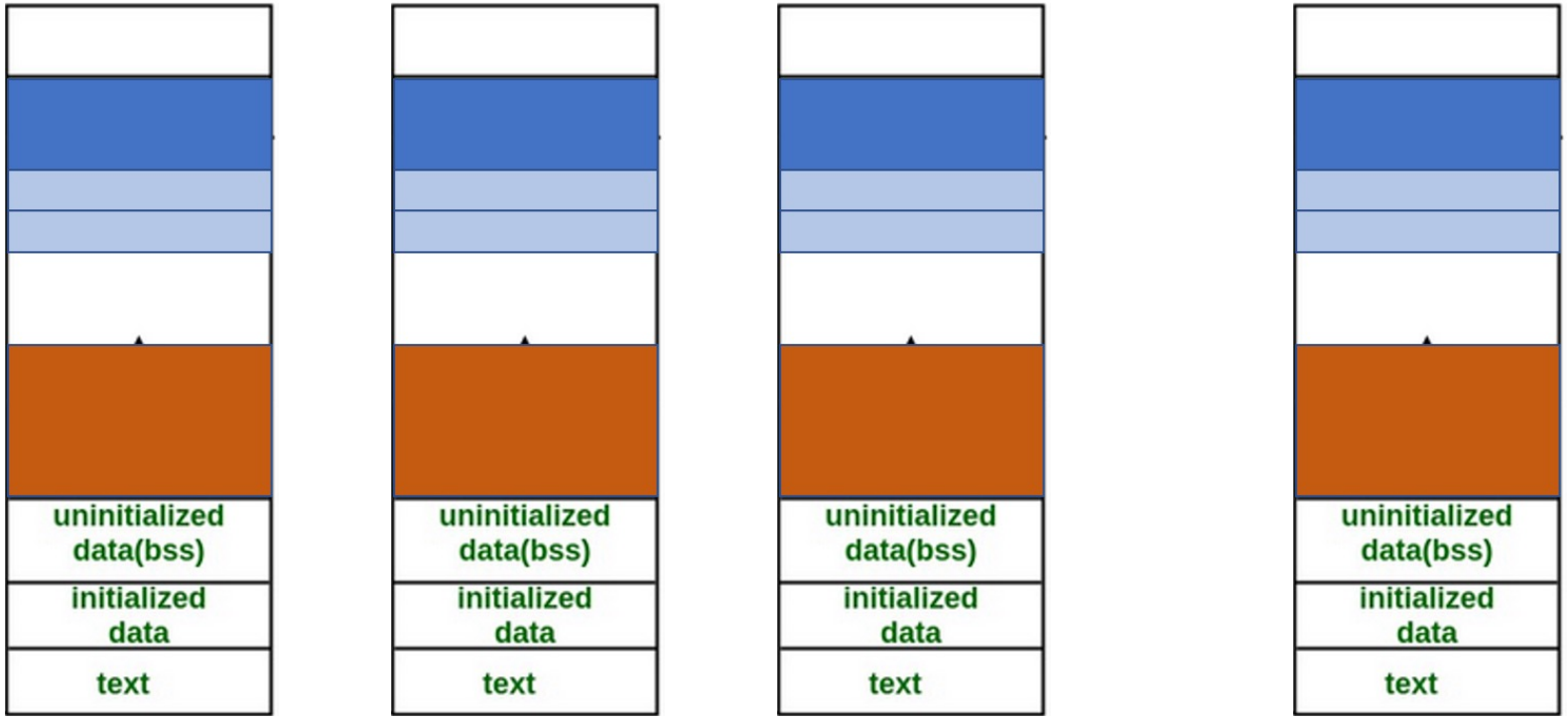


Multiple Processes



Multiple Threads
in a Single Processes

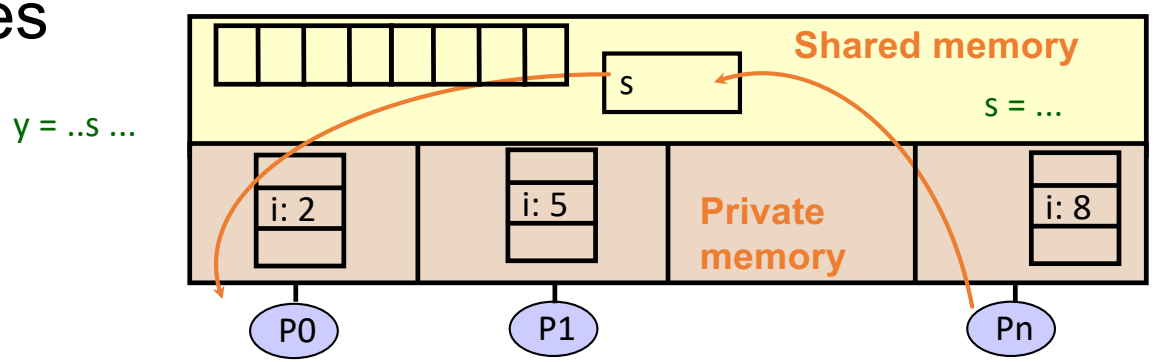
Hybrid – MPI + OpenMP



Multiple Processes each with Multiple Threads

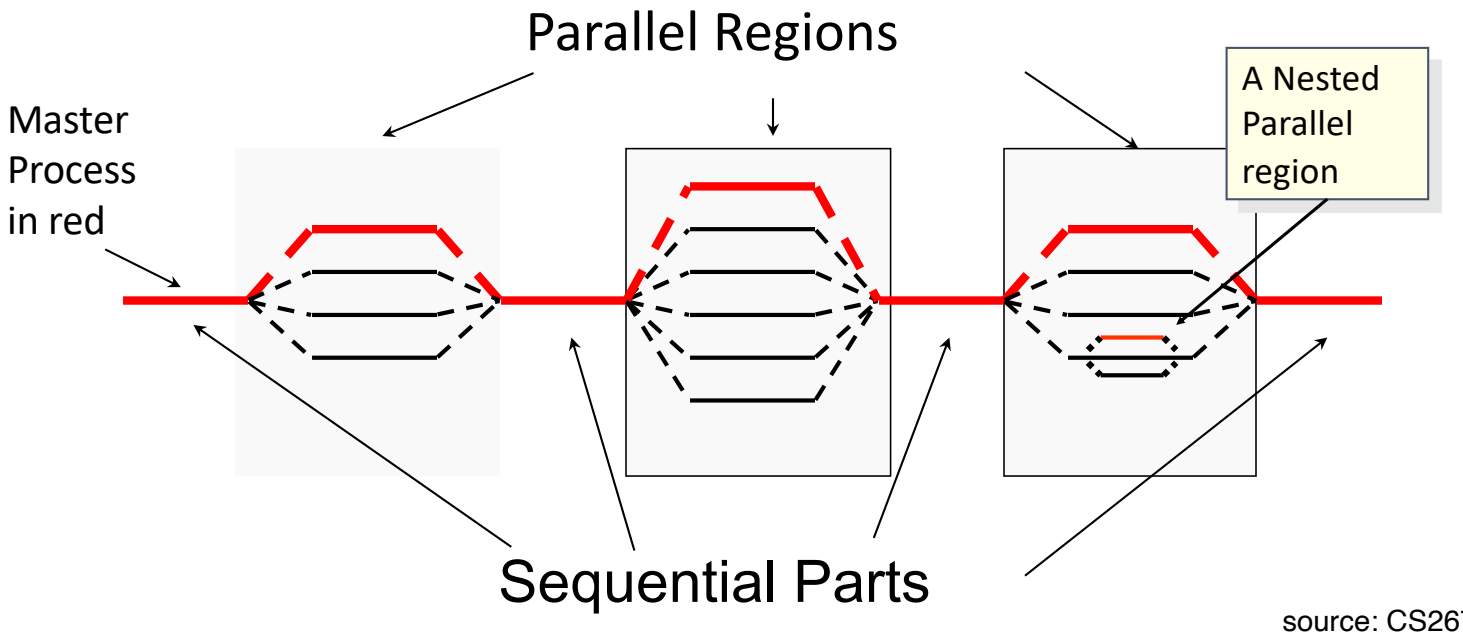
Threads

- Can be created dynamically, mid-execution, in some languages
- Each thread has a set of **private variables**, e.g., local stack variables
- Also a set of **shared variables,** e.g., static variables, shared common blocks, or global heap.
- Threads communicate **implicitly by writing and reading shared variables.**
- Threads **coordinate by synchronizing** on shared variables



Programming for Threads

- Master Process spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



source: CS267, Jim Demmell

Runtime Library Options for Shared Memory

POSIX Threads (pthreads)
OpenMP

- OpenMP provides multi-threaded capabilities to C, C++ and Fortran Programs
- In a threaded environment, threads communicate by sharing data
- Unintended sharing of data causes **race conditions**
- **Race Condition**: program output is different every time you run the program, a consequence of the threads being scheduled differently
- OpenMP provides constructs to control what blocks of code are run in parallel and also constructs for providing access to shared data using synchronization
- Synchronization has overhead consequences, you have to minimize them to get good speedup.
- INTERFACE: https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

- Mostly Set of Compiler directives (#pragma) applying to structured block

```
#pragma omp parallel
{

}
```

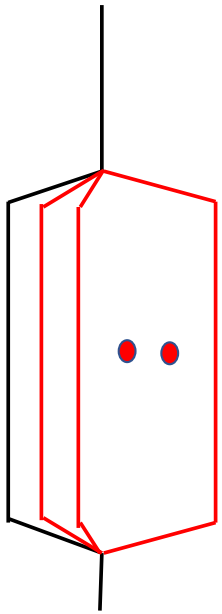
- Some runtime library calls

```
omp_num_threads(4);
```

- Being compiler directives, they are built into most compilers .
- Just have to activate it when compiling

```
gcc hello.c –fopenmp
icc hello.c  -qopenmp
```

Hello World



Code/Parallel/openmp/hello1.c

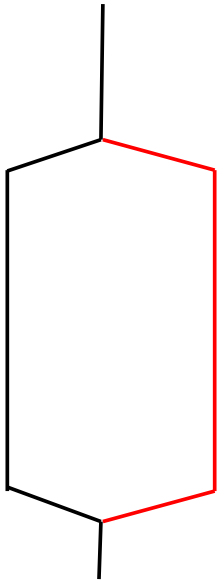
```
#include <omp.h>
#include <stdio.h>

int main( int argc, char *argv[])
{
  #pragma omp parallel
  {
     int id = omp_get_thread_num();
     int numP = omp_get_num_threads();
     printf("Hello World, I am  %d of %d\n",id,numP);
  }
  return 0;
}
```

```
openmp >gcc-7.2 hello1.c -fopenmp; ./a.out
Hello World, I am  0 of 4
Hello World, I am  3 of 4
Hello World, I am  1 of 4
Hello World, I am  2 of 4
openmp >
```

Each thread executes code within structured block

```
openmp >export env OMP_NUM_THREADS=2
openmp >./a.out
Hello World, I am  0 of 2
Hello World, I am  1 of 2
openmp >
```

Hello World – changing num threads



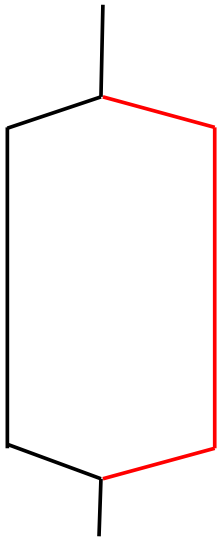
```
#include <openmp.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
  omp_set_num_threads(2);
  #pragma omp parallel
   {
        int id = omp_get_thread_num();
        int numP = omp_get_num_threads();
        printf("Hello World, I am  %d of %d\n",id,nu
   }
  return 0;
}
```

Code/Parallel/openmp/hello2.c

Actually an upper limit ..
You might not get all requesetd

Runtime function to request a certain number of threads

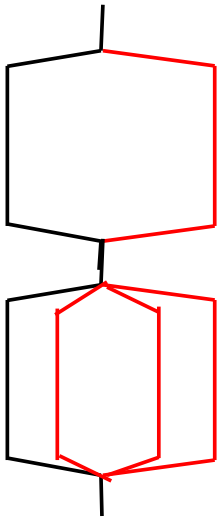
Runtime function to return actual number of threads in the team



Code/Parallel/openmp/hello3.c

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

int main( int argc, char *argv[] )
{
#pragma omp parallel num_threads(2)
   {
        int id = omp_get_thread_num();
        int numP = omp_get_num_threads();
        printf("Hello World, I am  %d of %d\n",id,numP);
   }
  return 0;
}
```

Different # threads in different blocks



Code/Parallel/openmp/hello4.c

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

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

#pragma omp parallel num_threads(2)
  {
    int id = omp_get_thread_num();
    int numP = omp_get_num_threads();
    printf("Hello World, I am  %d of %d\n",id,numP);
  }
#pragma omp parallel num_threads(4)
  {
    int id = omp_get_thread_num();
    int numP = omp_get_num_threads();
    printf("Hello World Again, I am  %d of %d\n",id,numP);
  }
  return(0);
}
```

```
openmp >gcc-7.2 hello4.c -fopenmp; ./a.out
Hello World, I am  0 of 2
Hello World, I am  1 of 2
Hello World Again, I am  1 of 4
Hello World Again, I am  2 of 4
Hello World Again, I am  3 of 4
Hello World Again, I am  0 of 4
openmp >
```

For Those Who Need to Know Why Stack

- It's Implementation under the hood – **THUNKS**!

```
#pragma omp parallel num_threads(4)
{
    foobar ();
}
```

=

```
void thunk ()
{
    foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
   pthread_create (
         &tid[i],0,thunk, 0);
thunk();

for (int i = 1; i < 4; ++i)
    pthread_join (tid[i]);
```

RACE CONDITIONS (can get different answers)
a consequence of variable sharing

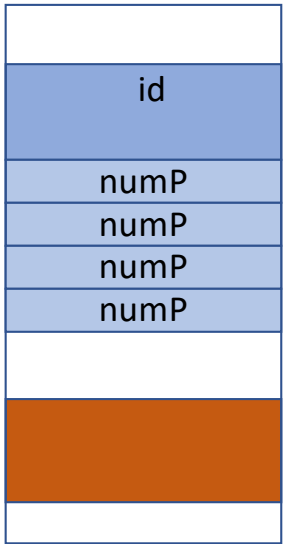
Code/Parallel/openmp/hello5.c

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

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

 int id;

#pragma omp parallel num_threads(4)
  {
    id = omp_get_thread_num();
    int numP = omp_get_num_threads();
    printf("Hello World from %d of %d th
  }

  return(0);
}
```

```
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 1 of 4 threads
Hello World from 2 of 4 threads
Hello World from 3 of 4 threads
Hello World from 0 of 4 threads
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 0 of 4 threads
Hello World from 3 of 4 threads
Hello World from 2 of 4 threads
Hello World from 1 of 4 threads
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 0 of 4 threads
Hello World from 0 of 4 threads
Hello World from 2 of 4 threads
Hello World from 3 of 4 threads
```

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

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

 int id;

#pragma omp parallel num_threads(4)
  {
    id = omp_get_thread_num();
    int numP = omp_get_num_threads();
    printf("Hello World from %d of %d threads\n",id,numP);
  }

  return(0);
}
```

```
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 1 of 4 threads
Hello World from 2 of 4 threads
Hello World from 3 of 4 threads
Hello World from 0 of 4 threads
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 0 of 4 threads
Hello World from 3 of 4 threads
Hello World from 2 of 4 threads
Hello World from 1 of 4 threads
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 0 of 4 threads
Hello World from 0 of 4 threads
Hello World from 2 of 4 threads
Hello World from 3 of 4 threads
```



Threads 0-4 share id

thread 0 has private var numP
thread 1 has private var numP
thread 2 has private var numP
thread 3 has private var numP

Between time thread 1 sets and prints the variable id, Process 0 has come in and changed it's value

Race Conditions Bane of Parallel Programming

Race Conditions in OpenMP occur when threads share the same data for writing

Race Conditions are Controlled through Synchronization

Simple Vector Sum

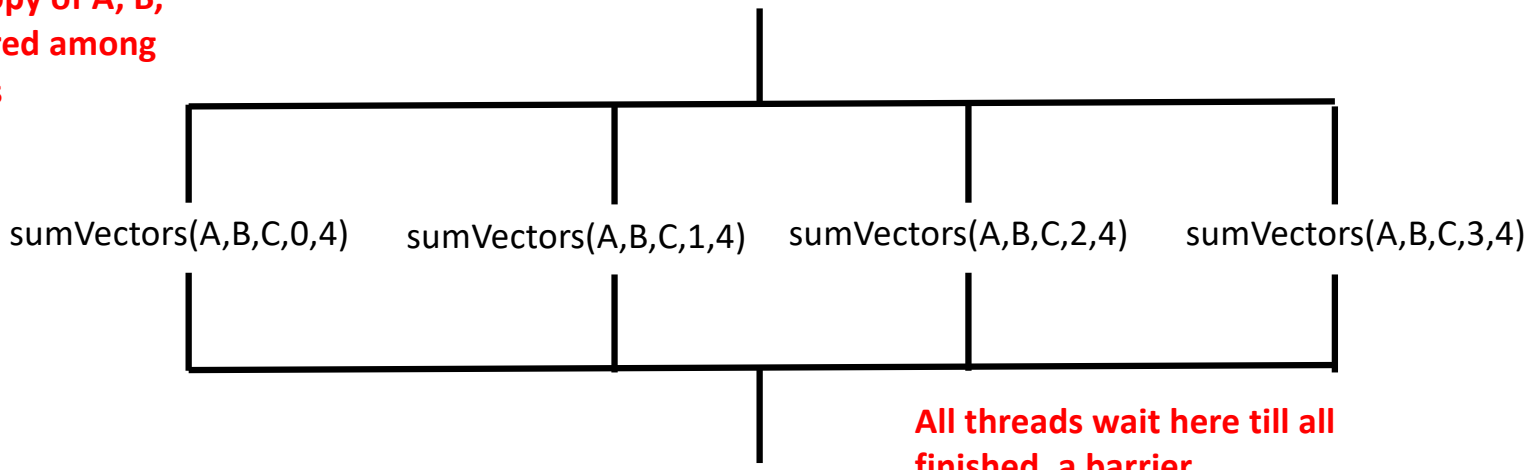
Code/Parallel/openmp/sum1.c

```
#include <omp.h>
#include <stdio.h>
#define DATA_SIZE 10000

void sumVectors(int N, double *A, double *B, double *C, int tid, int numT);
int main(int argc, const char **argv) {
  double a[DATA_SIZE], b[DATA_SIZE], c[DATA_SIZE];
  int num;
  for (int i=0; i<DATA_SIZE; i++) { a[i] = i+1; b[i] = i-
  double tdata = omp_get_wtime();
#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    num = numT;
    sumVectors(DATA_SIZE, a, b, c, tid, numT);
  }
  tdata = omp_get_wtime() - tdata;
  printf("first %f last %f in time %f using %d threa
  return 0;
}
```

```
void sumVectors(int N, double *A, double *B, double *C, int tid, int numT) {
  // determine start & end for each thread
  int start =   tid * N / numT;
  int end = (tid+1) * N / numT;
  if (tid == numT-1)
    end = N;


  // do the vector sum for threads bounds
  for(int i=start; i<end; i++) {
    C[i] = A[i]+B[i];
  }
}
```

Implicit Barrier in Code

A single copy of A, B, and C shared among all threads



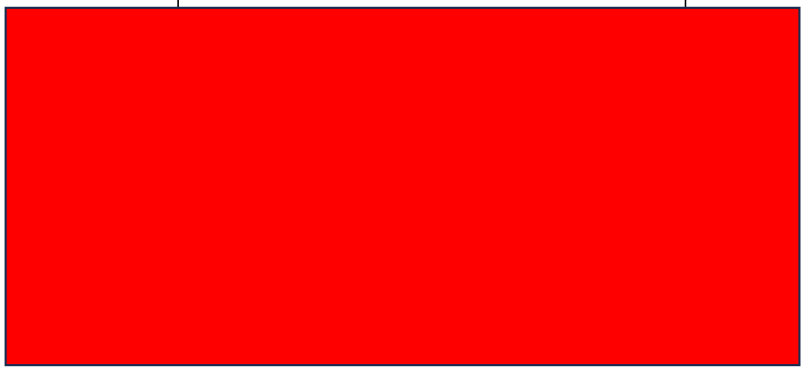

```
openmp >export env OMP_NUM_THREADS=1; ./a.out
first 2.000000 last 200000.000000 in time 0.000902 using 1 threads
openmp >export env OMP_NUM_THREADS=2; ./a.out
first 2.000000 last 200000.000000 in time 0.000678 using 2 threads
openmp >export env OMP_NUM_THREADS=4; ./a.out
first 2.000000 last 200000.000000 in time 0.000652 using 4 threads
openmp >export env OMP_NUM_THREADS=8; ./a.out
first 2.000000 last 200000.000000 in time 0.000693 using 8 threads
```

The for is such an obvious candidate for threads:

code/Parallel/openmp/sum2.c

```
#include <omp.h>
#include <stdio.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  double a[DATA_SIZE], b[DATA_SIZE], c[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++) { a[i] = i+1; b[i] = i+1; }
  double tdata = omp_get_wtime();
#pragma omp parallel
  {
   #pragma omp for
   for (int i=0; i<DATA_SIZE; i++)
      c[i] = a[i]+b[i];
  }
  tdata = omp_get_wtime() - tdata;
  printf("first %f last %f in time %f \n",c[0], c[DATA_SIZE-1], tdata);
  return 0;
}
```

Code/Parallel/openmp/sum3.c



How About Dot Product?

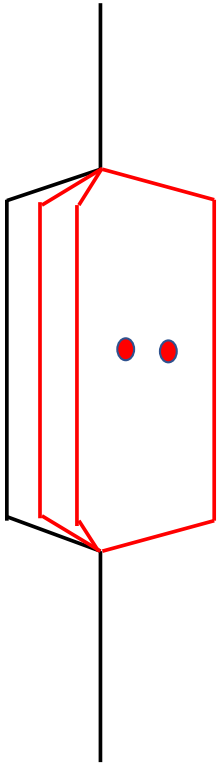
code/Parallel/openmp/dot1.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  int nThreads = 0;
  double dot = 0, a[DATA_SIZE], sum[64];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;
  for (int i=0; i<64; i++) sum[i] = 0;

#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    if (tid == 0)  nThreads = numT;
    for (int i=tid; i<DATA_SIZE; i+= numT)
      sum[tid] += a[i]*a[i];
  }
  for (int i=0; i<nThreads; i++)
      dot += sum[i];
  dot = sqrt(dot);
  printf("dot %f\n", dot);
  return 0;
}
```

Create a shared array to store data

Iterate over big array
using thread id
and number of threads

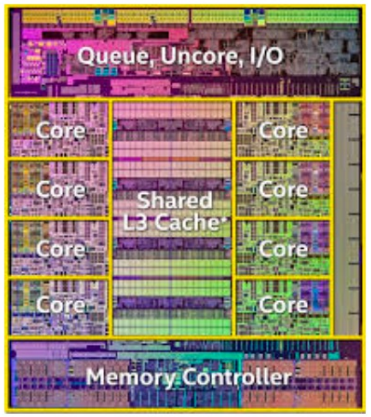
Combine sequentially



Poor Performance?

- Want high performance for shared memory: Use Caches!
  - Each processor has its own cache (or multiple caches)
  - Place data from memory into cache
  - Writeback cache: don't send all writes over bus to memory

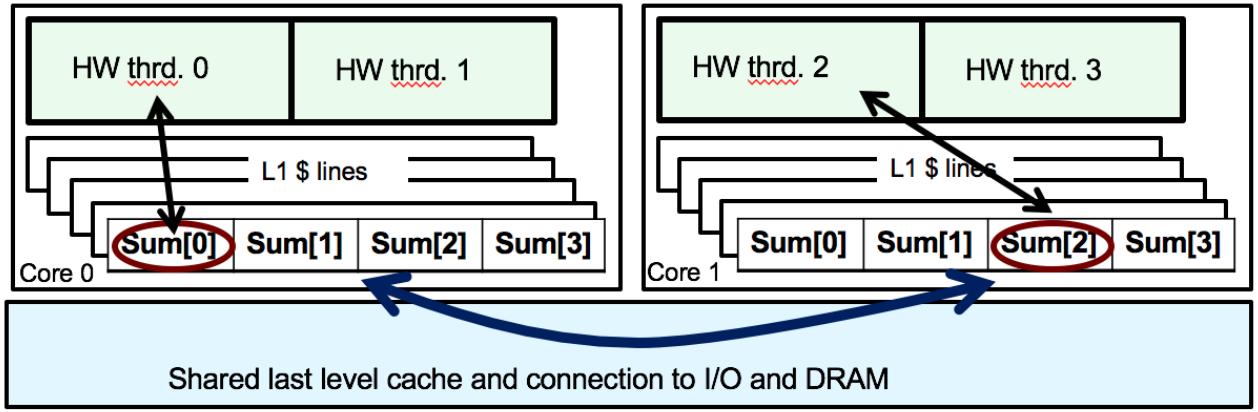
L3 (sometime L2 & LS)
Cache Actually Shared



- Problem is in multi-threaded model with all threads wanting to WRITE same spatially temporal data we have contention at the cache line in the L3 cache

False Sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to "slosh back and forth" between threads … This is called **false sharing** or sequential **consistency**.



- Sequential Consistency problem is pervasive and performance critical in shared memory

Solution?

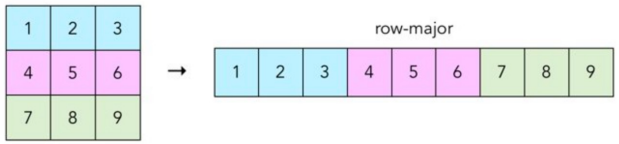
source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere

AVOID FALSE SHARING

code/Parallel/openmp/dot2.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000
#define PAD 64

int main(int argc, const char **argv) {
  int nThreads = 0;
  double dot = 0, a[DATA_SIZE], sum[64][PAD];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;
  for (int i=0; i<64; i++) sum[i][0]= 0;

#pragma omp parallel
  {
     int tid = omp_get_thread_num();
     int numT = omp_get_num_threads();
     if (tid == 0)  nThreads = numT;
     for (int i=tid; i<DATA_SIZE; i+= numT)
       sum[tid][0]+= a[i]*a[i];
  }
  for (int i=0; i<nThreads; i++)
      dot += sum[i][0];
  dot = sqrt(dot);
  printf("dot %f \n", dot);
  return 0;
```

C:



Pad the shared array to store data to avoid false sharing

SYNCHRONIZATION

code/Parallel/openmp/dot3.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  double dot = 0;
  double a[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;

#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    double sum = 0.;
    for (int i=tid; i<DATA_SIZE; i+= numT)
      sum += a[i]*a[i];
#pragma omp_critical
    dot += sum;
  }
  dot = sqrt(dot);
    printf("dot %f  \n", dot);
  return 0;
}
```

Only 1 thread will be in here at any one time

```
#pragma omp_critical
  {
    ir # of lines of code
  }
```

REDUCTION

code/Parallel/openmp/dot4.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  double dot = 0;
  double a[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;

#pragma omp parallel reduction(+:dot)
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    double sum = 0.;
    for (int i=tid; i<DATA_SIZE; i+= numT)
      sum += a[i]*a[i];

    dot += sum;
  }
  dot = sqrt(dot);
   printf("dot %f  \n", dot);
  return 0;
}
```

dot5.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  double dot = 0;
  double a[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;

#pragma omp parallel reduction(+:dot)
  {
  #pragma parallel for
      for (int i=tid; i<DATA_SIZE; i+= numT)
      dot += a[i]*a[i];
  }
  dot = sqrt(dot);
   printf("dot %f  \n", dot);
  return 0;
}
```

Additional Reduction Operators

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

Pitfalls to Parallel Loops

might only occur for experienced programmer!

- 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

```
x = 0.5*dx;
for (int i=0; i<numSteps; i++) {
    pi += 4./(1.+x*x);
    x += dx;
  }
```

Note: loop index "i" is private by default

```
#pragma omp parallel for reduction(+:pi)
  for (int i=0; i<numSteps; i++) {
        x = (i+0.5)*dx;
        pi += 4./(1.+x*x);
  }
```

Remove loop carried dependence

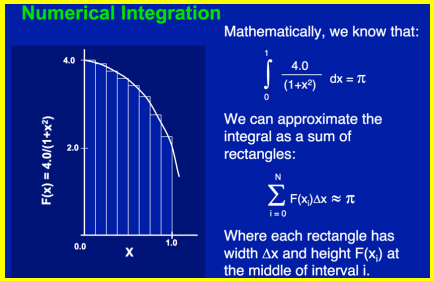
```
#pragma omp parallel for
reduction(+:pi)
   for (int i=0; i<numSteps; i++) {
         pi += 4./(1.+x*x);
         x+=dx;
   }
```

Exercise 1: Compute Vector Norm using OpenMP

$$||u||_2 = \sqrt{u_1^2 + u_2^2 + \cdots + u_n^2}$$

normVector.c in assignments/C-Day4/ex1

Exercise 2: Parallelize Compute PI using OpenMP



pi.c in assignments/C-Day5/ex2