SALVATORE DI GIROLAMO <DIGIROLS@INF.ETHZ.CH>

DPHPC: Introduction to OpenMP

Recitation session

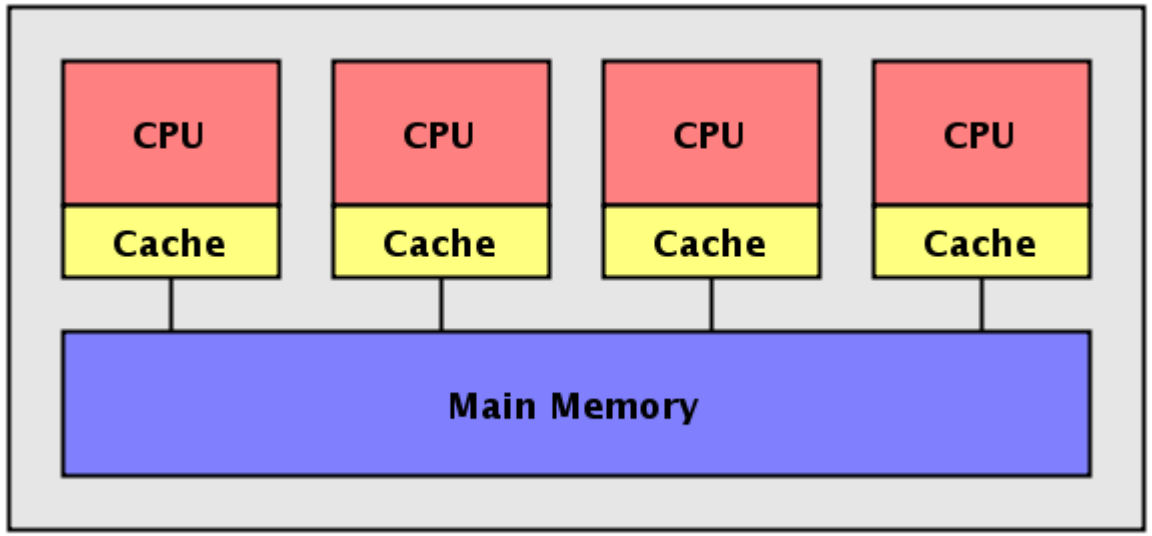
Systems@ETH zürich

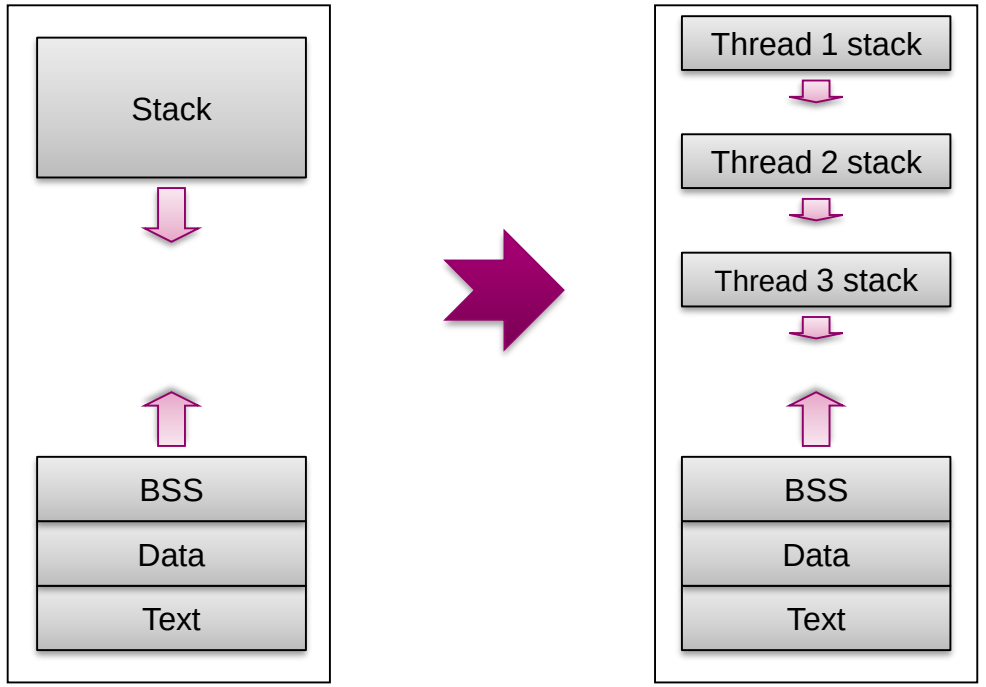
Based on http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf

OpenMP – An Introduction

- **What is it? A set of compiler directives and a runtime library**
  - ***#pragma omp parallel num_threads(4)***
  - **#include <omp.h>**

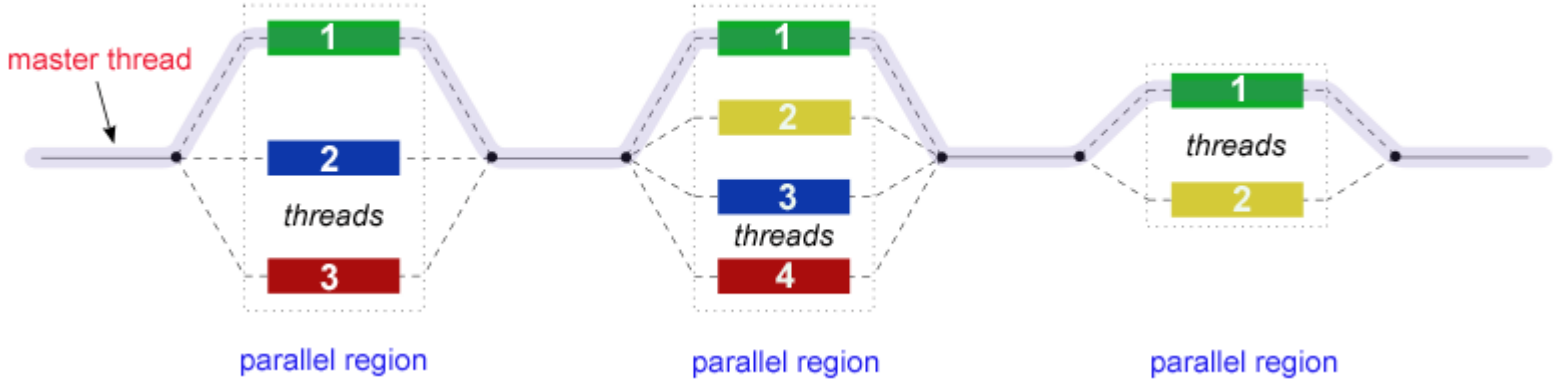
- **Why do we care? Simplify (& standardizes) how multi-threaded application are written**
  - Fortran, C, C++





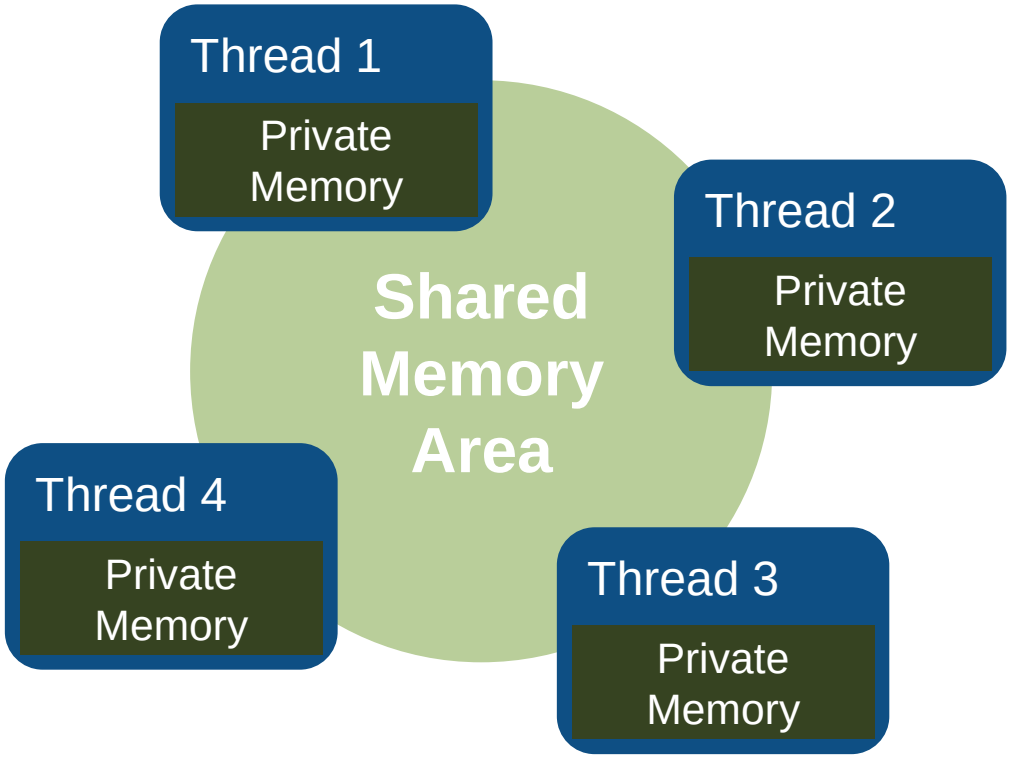
OpenMP – An Introduction

- **OpenMP is based on Fork/Join model**
  - When program starts, one Master thread is created
  - Master thread executes sequential portions of the program
  - At the beginning of parallel region, master thread forks new threads
  - All the threads together now forms a “team”
  - At the end of the parallel region, the forked threads die



What's a Shared-Memory Program?

- **One process that spawns multiple threads**
- **Threads can communicate via shared memory**
  - Read/Write to shared variables
  - Synchronization can be required!
- **OS decides how to schedule threads**



OpenMP: Hello World

- **Make "Hello World" multi-threaded**

```
int main(){
    int ID=0;
    printf("hello(%d) ", ID);;
    printf("world(%d)\n", ID);;
}
```

Include OpenMP header

Start parallel region with "default" number of threads

Who am I?

```
#include "omp.h"

int main(){
 #pragma omp parallel
 {
    int ID = omp_get_thread_num();
    printf("hello(%d) ", ID);;
    printf("world(%d)\n", ID);;
 }
}
```

Parallel Regions

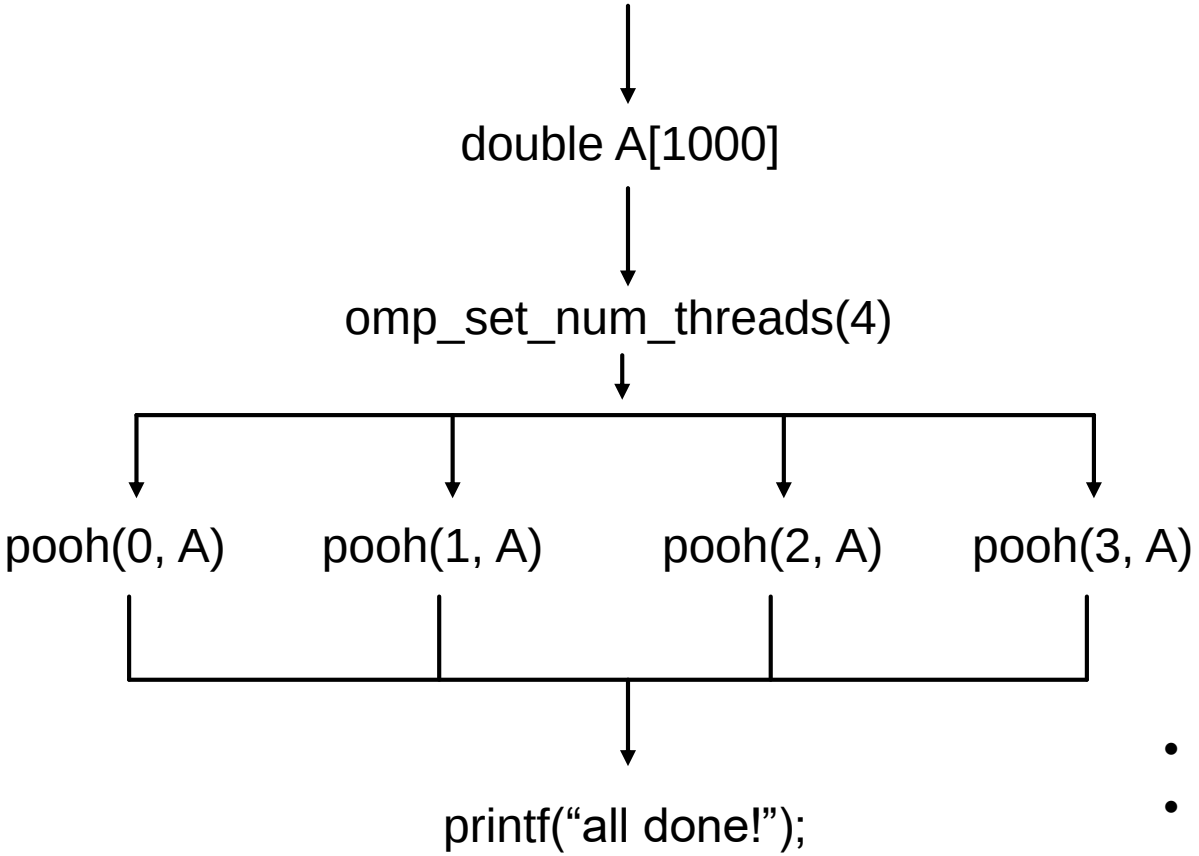
- **A parallel region identifies a portion of code that can be executed by different threads**
- **You can create a parallel region with the "parallel" directive**
- **You can request a specific number of threads with *omp_set_num_threads(N)***

```
double A[1000];
omp_set_num_threads(4);
 #pragma omp parallel
 {
    int ID = omp_get_thread_num();
    pooh(ID,A);
 }
printf("all done!");
```

```
double A[1000];

 #pragma omp parallel num_threads(4)
 {
    int ID = omp_get_thread_num();
    pooh(ID,A);
 }
printf("all done!");
```

- **Each thread will call pooh with a different value of ID**

Parallel Regions



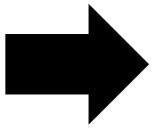

```
double A[1000];
omp_set_num_threads(4);
 #pragma omp parallel
 {
    int ID = omp_get_thread_num();
    pooh(ID,A);
 }
printf("all done!);
```

- **All the threads execute the same code**
- **The A array is shared**
- **Implicit synchronization at the end of the parallel region**

Behind the scenes

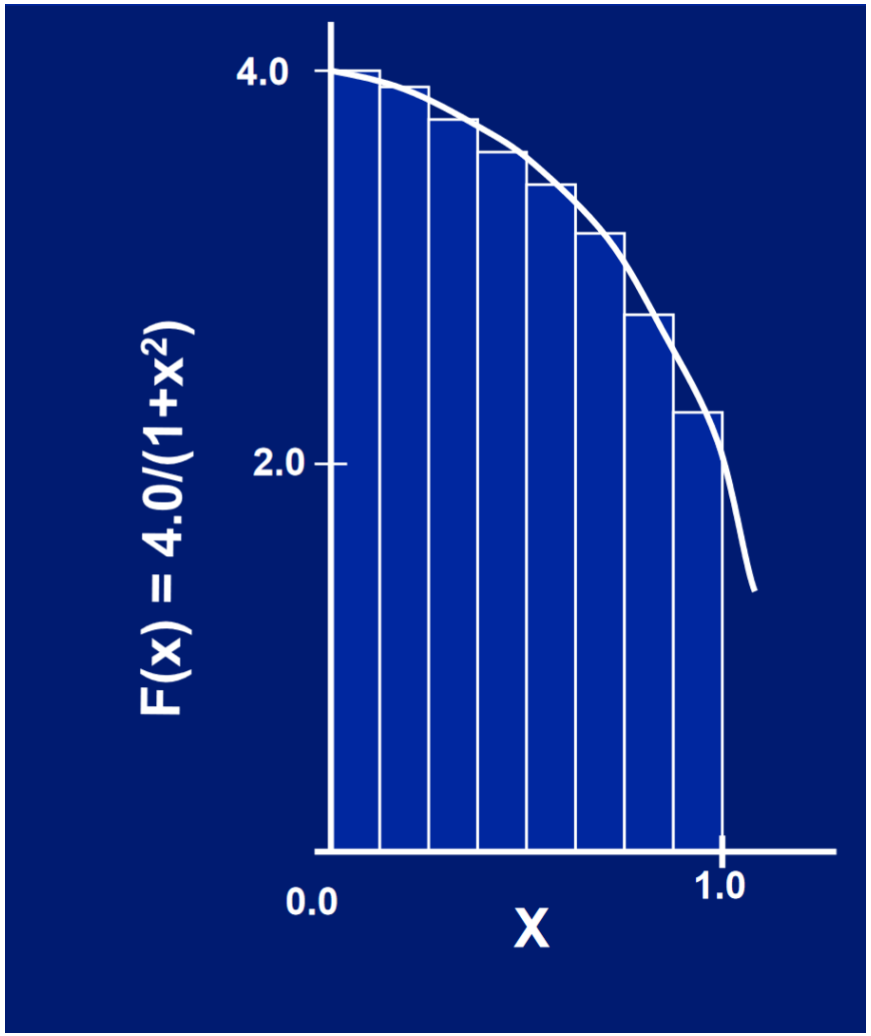
- The OpenMP compiler generates code logically analogous to that on the right
- All known OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for each parallel region.
- Only three threads are created because the last parallel section will be invoked from the parent thread.

```
#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]);
```

Exercise: Compute PI



- **Mathematically, we know that computing the integral of**

  4/(1+x*x)

  from 0 to 1 will give us the value of pi – which is great since it gives us an easy way to check the answer.

- **We can approximate the integral as a sum of rectangles:**

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

- **Where each rectangle has width 'x and height F(x_i) at the middle of interval i.**

Computing PI: Sequential Version

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000000;
double step;
int main () {
        int i;
        double x, pi, sum = 0.0;
        double start_time, run_time;

        step = 1.0/(double) num_steps;

        start_time = omp_get_wtime();

        for (i=0;i<num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }

        pi = step * sum;
        run_time = omp_get_wtime() - start_time;
        printf("\n pi with %ld steps is %lf in %lf seconds\n ",num_steps,pi,run_time);
}
```

Parallel PI

- **Create a parallel version of the pi program using a parallel construct.**
- **Pay close attention to shared versus private variables.**

- **In addition to a parall**
  - int omp_get_num_threa
  - int omp_get_thread_nu
  - double omp_get_wtime

This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

MPI programs almost always use this pattern ... it is probably the most commonly used pattern in the history of parallel programming.

- **Possible strategy:**
  - Run the same program on P processing elements where P can be arbitrarily large.
  - Use the rank ... an ID ranging from 0 to (P-1) ... to select between a set of tasks and to manage any shared data structures.

Simple Parallel PI

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i, nthreads;  double pi, sum[NUM_THREADS];
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
           int i, id,nthrds;
           double x;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)   nthreads = nthrds;
           for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                       x = (i+0.5)*step;
                       sum[id] += 4.0/(1.0+x*x);
           }
    }
           for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don't conflict.

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations

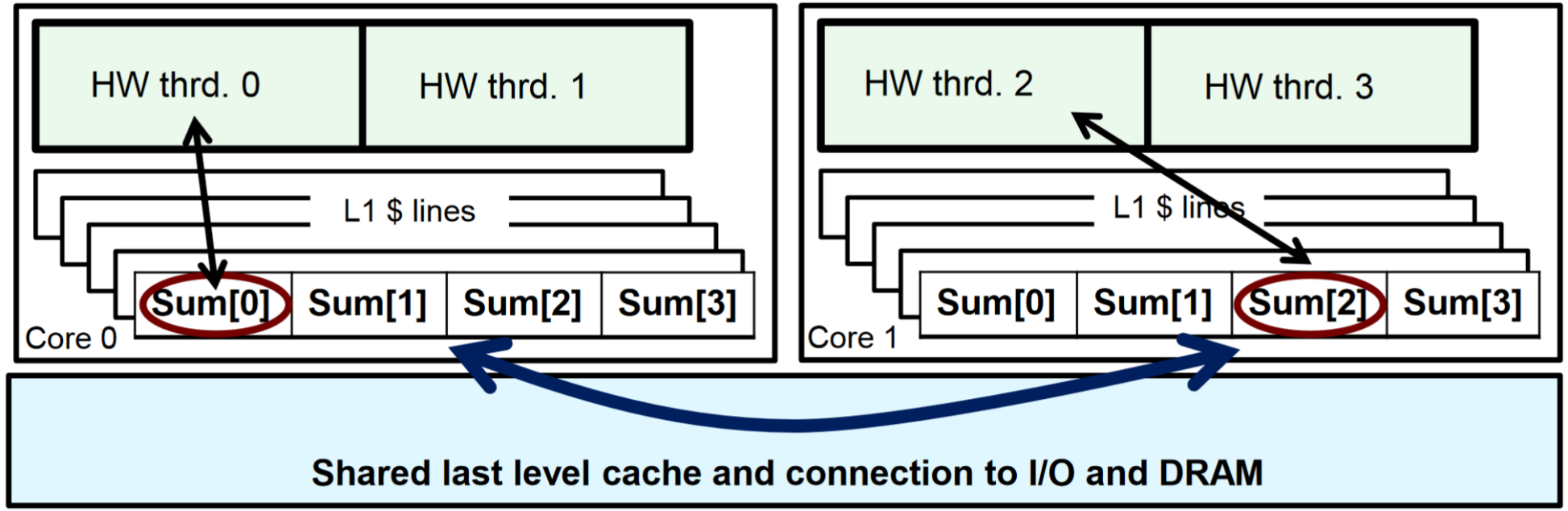
Original Serial pi program with 100000000 steps ran in 1.83 seconds.

threads	1st SPMD
1	1.86
2	1.03
3	1.08
4	0.97

Poor scaling!!!

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



HotFix: Pad arrays so elements you use are on distinct cache lines.

Padding the PI

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define    PAD      8          // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{             int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
              step = 1.0/(double) num_steps;
              omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {        int i, id,nthrds;
            double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;
             for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                        x = (i+0.5)*step;
                        sum[id][0] += 4.0/(1.0+x*x);
             }
    }
             for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

Pad the array so each sum value is in a different cache line

threads	1st SPMD	1st SPMD padded
1	1.86	1.86
2	1.03	1.01
3	1.08	0.69
4	0.97	0.53

SPMD vs WorkSharing

- **A parallel construct by itself creates an SPMD or "Single Program Multiple Data" program ... i.e., each thread redundantly executes the same code.**

- **How do you split up pathways through the code between threads within a team?**
  - WorkSharing: The OpenMP loop construct (not the only way to go)

- **The loop worksharing construct splits up loop iterations among the threads in a team**

```
#pragma omp parallel
{
    #pragma omp for
    for (I=0;I<N;I++){
          NEAT_STUFF(I);
    }
}
```

The variable I is made "private" to each thread by default. You could do this explicitly with a "private(I)" clause

Why should we use it (the loop construct)?

Sequential:

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

OpenMP parallel region:

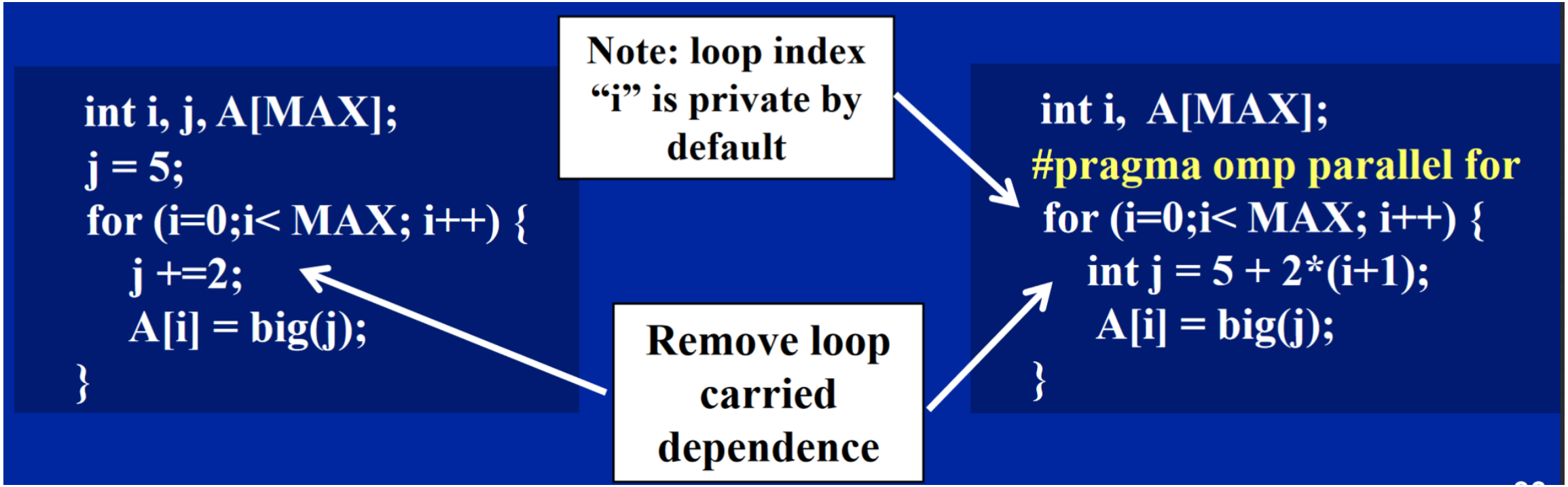
```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        if (id == Nthrds-1)iend = N;
        for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and worksharing:

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

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



Reduction

- **OpenMP reduction clause:**
  - ***reduction (op : list)***
- **Inside a parallel or a work-sharing construct:**
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- **The variables in “list” must be shared in the enclosing parallel region.**

```
 double  ave=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:ave)
 for (i=0;i< MAX; i++) {
        ave + = A[i];
 }
 ave = ave/MAX;
```

PI with loop construct

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{    int i;            double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
     #pragma omp parallel
     {
        double x;
       #pragma omp for reduction(+:sum)
          for (i=0;i< num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
          }
      }
           pi = step * sum;
}
```

Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum. Note … the loop indix is local to a thread by default.

OpenMP - Synchronization

Synchronization is used to impose order constraints and to protect access to shared data

- **High level synchronization:**
  - Critical, Atomic, Barrier, Ordered
- **Low level synchronization**
  - Flush, Locks (both simple and nested)

http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf

Barrier

- **Each thread waits until all threads arrive.**

```
#pragma omp parallel
{
        int id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier

        B[id] = big_calc2(id, A);
}
```

http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf

Critical

- **Mutual exclusion: Only one thread at a time can enter a critical region**

```
float  res;
#pragma omp parallel
{    float B;   int i, id, nthrds;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
     for(i=id;i<niters;i+=nthrds){
          B =  big_job(i);
#pragma omp critical
          res += consume (B);
     }
}
```

http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf

Atomic

- **Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)**

```
#pragma omp parallel
{
        double tmp, B;
        B =  DOIT();
        tmp = big_ugly(B);
 #pragma omp atomic
        X +=  tmp;
}
```

The statement inside the atomic must be one of the following forms:

- x binop= expr
- x++
- ++x
- x—
- --x

X is an lvalue of scalar type and binop is a non-overloaded built in operator.

http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf

Locks in OpenMP

- **Simple Locks**
  - A simple lock is available if not set
  - Routines:

    omp_init_lock/omp_destroy_lock: create/destroy the lock

    omp_set_lock: acquire the lock

    omp_test_lock: test if the lock is available and set it if yes. Doesn't block if already set.

    omp_unset_lock: release the lock

- **Nested Locks**
  - A nested lock is available if it is not set OR is set and the calling thread is equal to the current owner
  - Same functions of the simple lock: omp_*_nest_lock

Note: a lock implies a memory fence of all the thread variables

http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf

Locks in OpenMP: An example

```
class data{
 private:
   std::set<int> flags;
 #ifdef _OPENMP
   omp_lock_t lock;
 #endif
 public:
   data() : flags(){
 #ifdef _OPENMP
     omp_init_lock(&lock);
 #endif
   }
   ~data(){
 #ifdef _OPENMP
     omp_destroy_lock(&lock);
 #endif
   }

   bool set_get(int c){
   #ifdef _OPENMP
     omp_set_lock(&lock);
   #endif
     bool found = flags.find(c) != flags.end();
     if(!found) flags.insert(c);
   #ifdef _OPENMP
     omp_unset_lock(&lock);
   #endif
     return found;
   }
 };
```

http://bisqwit.iki.fi/story/howto/openmp/#Locks

Locks in OpenMP: An example

```
#ifdef _OPENMP
struct MutexType{
  MutexType() { omp_init_lock(&lock); }
  MutexType(omp_lock_t _lock) { lock=_lock; omp_init_lock(&lock); }
  ~MutexType() { omp_destroy_lock(&lock); }
  void Lock() { omp_set_lock(&lock); }
  void Unlock() { omp_unset_lock(&lock); }
  MutexType(const MutexType& ) { omp_init_lock(&lock); }
  MutexType& operator= (const MutexType& ) { return *this; }
public:
  omp_lock_t lock;
};
#else
/* A dummy mutex that doesn't actually exclude anything, but as the
struct MutexType{
  void Lock() {}
  void Unlock() {}
};
#endif

/* An exception-safe scoped lock-keeper. */
struct ScopedLock{
  explicit ScopedLock(MutexType& m) : mut(m), locked(true) { mut.Lock(); }
  ~ScopedLock() { Unlock(); }
  void Unlock() { if(!locked) return; locked=false; mut.Unlock(); }
  void LockAgain() { if(locked) return; mut.Lock(); locked=true; }
private:
  MutexType& mut;
  bool locked;
private: // prevent copying the scoped lock.
  void operator=(const ScopedLock&);
  ScopedLock(const ScopedLock&);
};
```

```
#include <set>

class data
{
private:
  std::set<int> flags;
  MutexType lock;
public:
  bool set_get(int c)
  {
    ScopedLock lck(lock); // locks the mutex

    if(flags.find(c) != flags.end()) return true; // was found
    flags.insert(c);
    return false; // was not found
  } // automatically releases the lock when lck goes out of scope.
};
```

http://bisqwit.iki.fi/story/howto/openmp/#Locks

Scoped Locking VS OMP Critical

Scoped Locking

- it is allowed to leave the locked region with jumps (e.g. break, continue, return), this is forbidden in regions protected by the critical-directive
- scoped locking is exception safe, critical is not
- all criticals wait for each other, with guard objects you can have as many different locks as you like

OMP Critical

- no need to declare, initialize and destroy a lock
- no need to include, declare and define a guard object yourself, as critical is a part of the language (OpenMP)
- you always have explicit control over where your critical section ends, where the end of the scoped lock is implicit
- works for C and C++ (and FORTRAN)
- less overhead, as the compiler can translate it directly to lock and unlock calls, where it has to construct and destruct an object for scoped locking

http://www.thinkingparallel.com/2006/08/21/scoped-locking-vs-critical-in-openmp-a-personal-shootout/