A “Hands-on” Introduction to OpenMP*

Tim Mattson

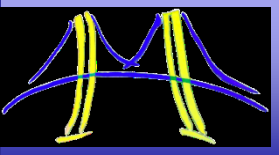
Intel Corp.

timothy.g.mattson@intel.com

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

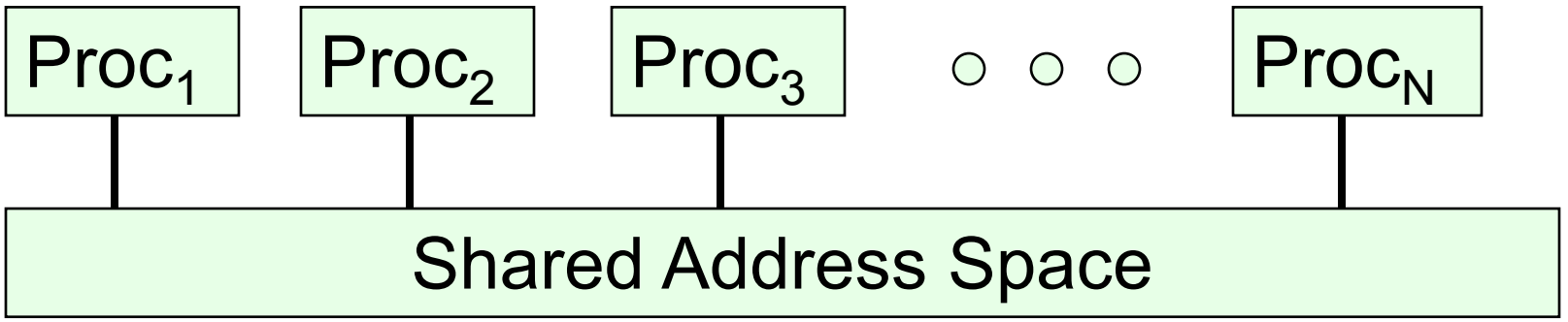
Introduction

- OpenMP is one of the most common parallel programming models in use today.
- It is relatively easy to use which makes a great language to start with when learning to write parallel software.
- Assumptions:
  - We assume you know C. OpenMP supports Fortran and C++, but we will restrict ourselves to C.
  - We assume you are new to parallel programming.
  - We assume you have access to a compiler that supports OpenMP (more on that later).



Shared memory Computers

- **Shared memory computer** : any computer composed of multiple processing elements that share an address space. Two Classes:
  - **Symmetric multiprocessor** (**SMP**): a shared address space with "equal-time" access for each processor, and the OS treats every processor the same way.
  - **Non Uniform address space multiprocessor** (**NUMA**): different memory regions have different access costs … think of memory segmented into "Near" and "Far" memory.



OpenMP* Overview:

C$OMP FLUSH

#pragma omp critical

C$OMP THREADPRIVATE(/ABC/)

CALL OMP SET NUM THREADS(10)

C$OM

C$OM

C$C

C

#pr

ED

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice

C$OMP PARALLEL COPYIN(/blk/)

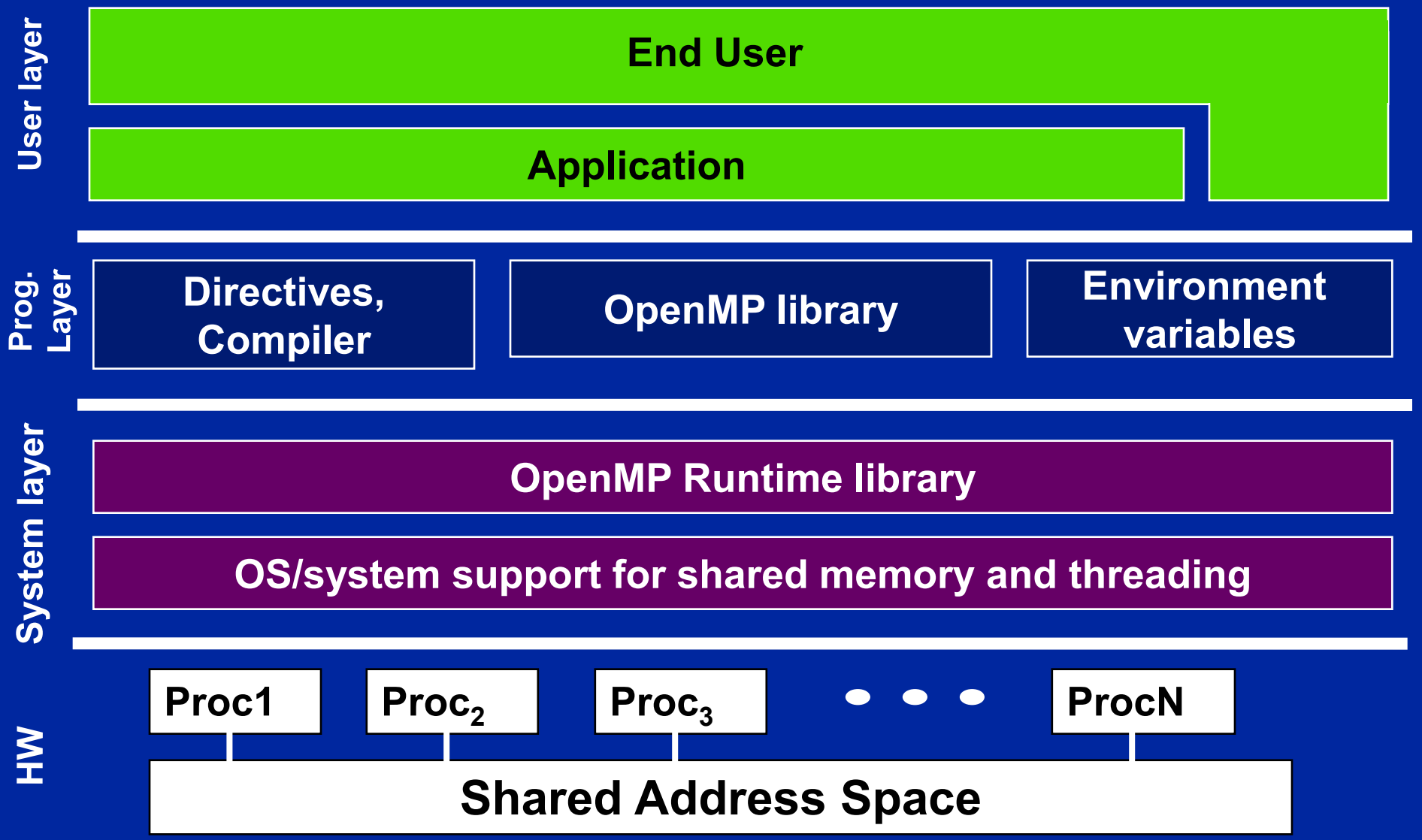
C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

omp_set_lock(lck)

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

OpenMP Basic Defs: Solution Stack



OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.

  #pragma omp *construct [clause [clause]…]*

  - Example

    #pragma omp parallel num_threads(4)

- Function prototypes and types in the file:

  #include <omp.h>

- Most OpenMP* constructs apply to a "structured block".
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  - It's OK to have an exit() within the structured block.

Exercise 1, Part A: Hello world

Verify that your environment works

- **Write a program that prints “hello world”.**

```
int main()
{



    int ID = 0;

    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);

}
```

Exercise 1, Part B: Hello world

Verify that your OpenMP environment works

- Write a multithreaded program that prints “hello world”.

Linux and OS X	gcc -fopenmp
PGI Linux	pgcc -mp
Intel windows	icl /Qopenmp
Intel Linux and OS X	icpc –openmp

```
#include <omp.h>
int main()
{
  #pragma omp parallel
  {
   int ID = 0;
   printf(“ hello(%d) ”, ID);
   printf(“ world(%d) \n”, ID);
  }
}
```

Exercise 1: Solution
A multi-threaded "Hello world" program

- **Write a multithreaded program where each thread prints "hello world".**

```
#include "omp.h"
int  main()
{

#pragma omp parallel
 {

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

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Exercise 1: Solution
A multi-threaded "Hello world" program

- **Write a multithreaded program where each thread prints "hello world".**

```
#include "omp.h"
int  main()
{

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

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Sample Output:

hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

world(2)

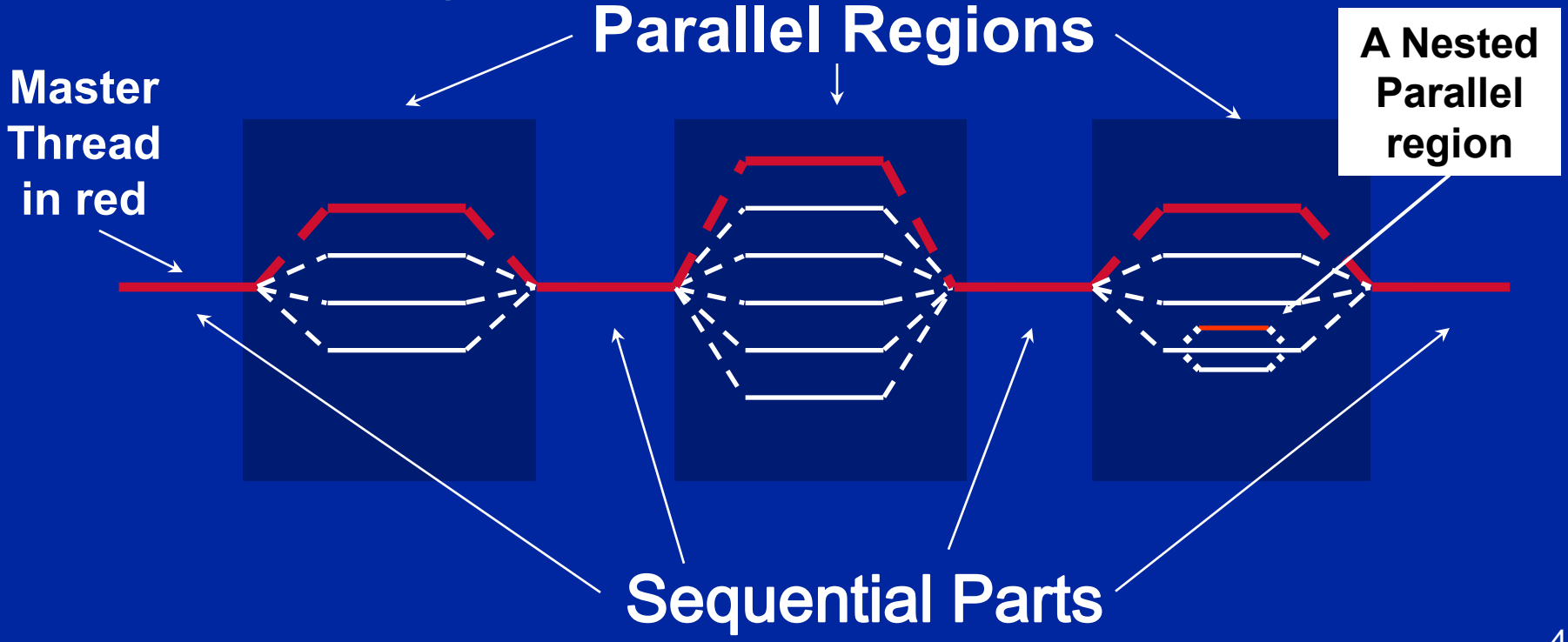
OpenMP Overview:
How do threads interact?

- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

OpenMP Programming Model:

Fork-Join Parallelism:

- Master thread 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.



Thread Creation: Parallel Regions

- **You create threads in OpenMP* with the parallel construct.**
- **For example, To create a 4 thread Parallel region:**

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

Runtime function to request a certain number of threads

Runtime function returning a thread ID

Each thread executes a copy of the code within the structured block

- **Each thread calls `pooh(ID,A)` for `ID = 0 to 3`**

* The name "OpenMP" is the property of the OpenMP Architecture Review Board

Thread Creation: Parallel Regions

- You create threads in OpenMP* with the parallel construct.
- For example, To create a 4 thread Parallel region:

```
double A[1000];

#pragma omp parallel num_threads(4)
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

clause to request a certain number of threads

Each thread executes a copy of the code within the structured block

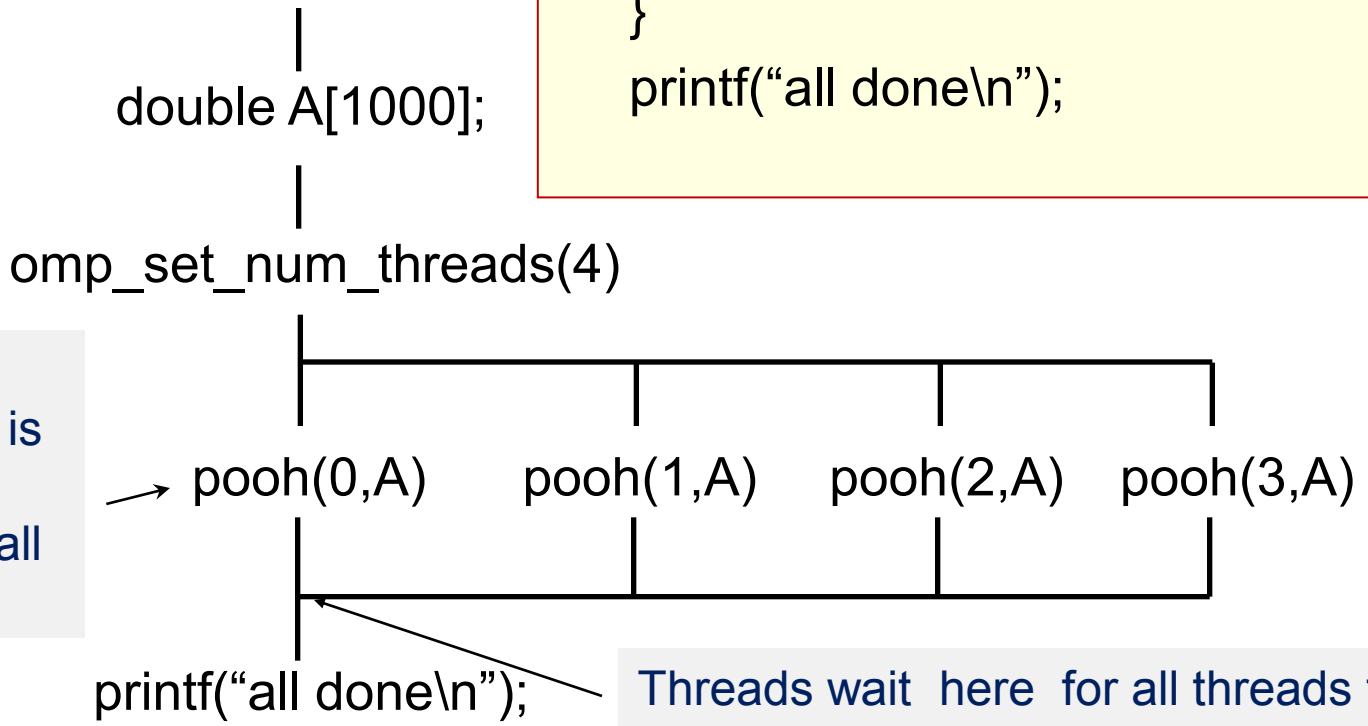
Runtime function returning a thread ID

- Each thread calls `pooh(ID,A)` for `ID = 0 to 3`

Thread Creation: Parallel Regions

- Each thread executes the same code redundantly.

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



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

OpenMP: what the compiler does

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

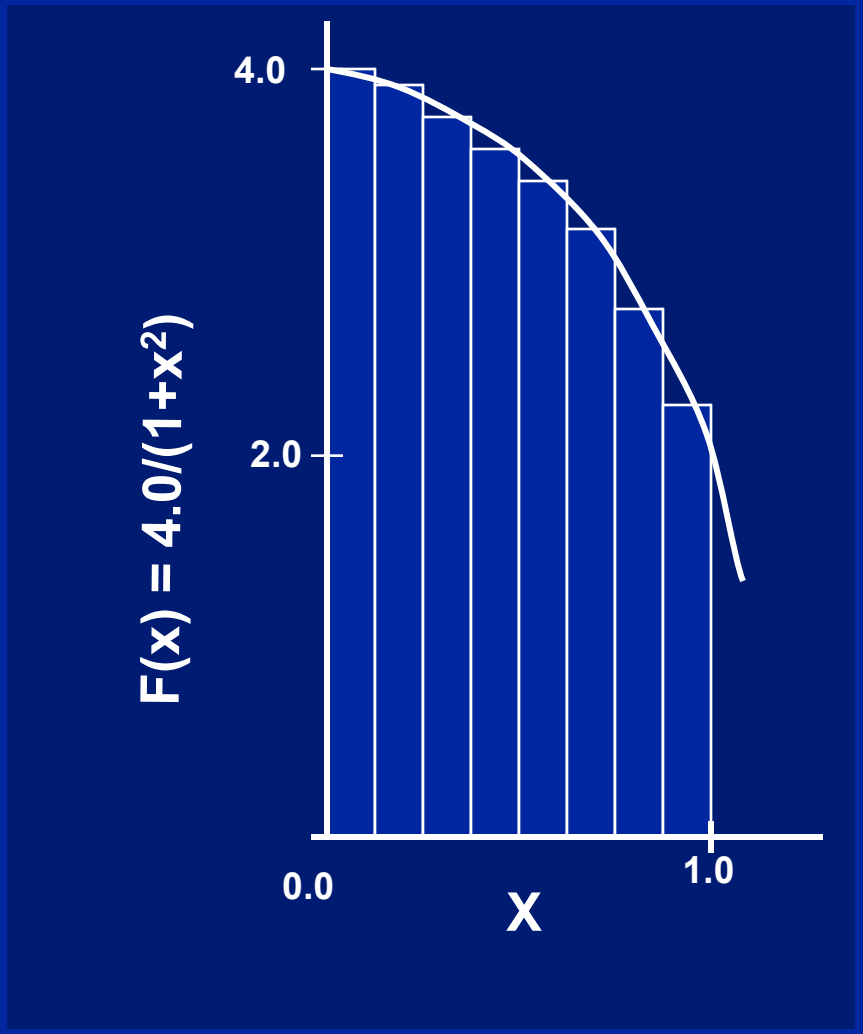
- The OpenMP compiler generates code logically analogous to that on the right of this slide, given an OpenMP pragma such as that on the top-left
- All known OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for reach parallel region.
- Only three threads are created because the last parallel section will be invoked from the parent thread.

```
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]);
```

Exercises 2 to 4:
Numerical Integration



Mathematically, we know that:

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

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 Δx and height $F(x_i)$ at the middle of interval i.

Exercises 2 to 4: Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{       int i;   double x, pi, sum = 0.0;

        step = 1.0/(double) num_steps;

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

Exercise 2

- Create a parallel version of the pi program using a parallel construct.
- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();

Number of threads in the team

Thread ID or rank

Time in Seconds since a fixed point in the past

Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{       int i;   double x, pi, sum = 0.0;

        step = 1.0/(double) num_steps;

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

Example: A simple Parallel pi program

```
#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

Algorithm strategy:
The SPMD (Single Program Multiple Data) design pattern

- 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.

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.

Results*

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

Example: A simple Parallel pi program

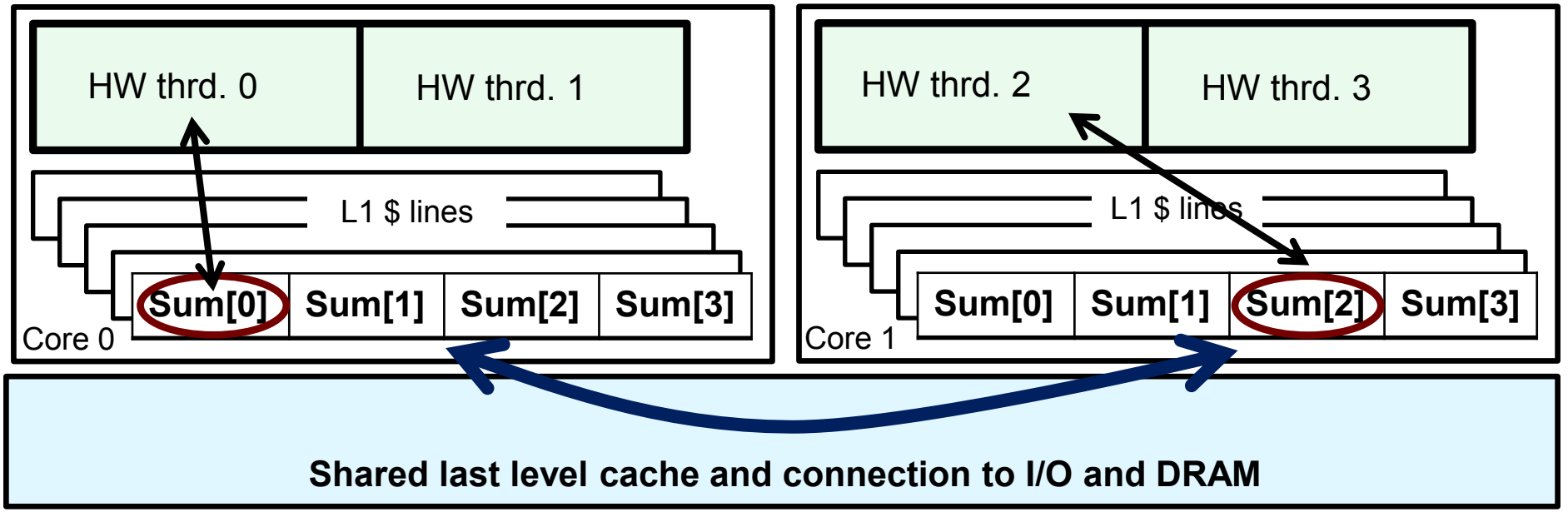
```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{          int i, nthreads;  double pi, sum[NUM_THREADS];
           step = 1.0/(double) num_steps;
           omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
          int i, id,nthrds;
          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;
}
```

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

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Why such 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 … This is called **"false sharing"**.



- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines … Results in poor scalability.
- Solution: Pad arrays so elements you use are on distinct cache lines.

Example: eliminate False sharing by padding the sum array

```
#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

Results*: pi program padded accumulator

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

Example: eliminate False sharing by padding the sum array

```
#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;
}
```

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

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

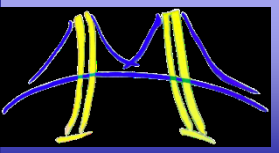
Do we really need to pad our arrays?

- Padding arrays requires deep knowledge of the cache architecture. Move to a machine with different sized cache lines and your software performance falls apart.
- There has got to be a better way to deal with false sharing.

OpenMP Overview:
How do threads interact?

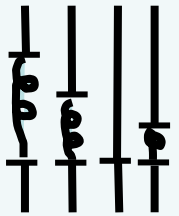
Recall our high level overview of OpenMP?

- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

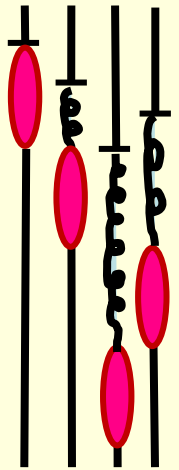


Synchronization:

- Synchronization: bringing one or more threads to a well defined and known point in their execution.
- The two most common forms of synchronization are:



Barrier: each thread wait at the barrier until all threads arrive.



Mutual exclusion: Define a block of code that only one thread at a time can execute.

Synchronization

- High level synchronization:
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

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

Discussed later

Synchronization: Barrier

- **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);
}
```

Synchronization: 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);
     }
}
```

Threads wait their turn – only one at a time calls consume()

Exercise 3

- In exercise 2, you probably used an array to create space for each thread to store its partial sum.
- If array elements happen to share a cache line, this leads to false sharing.
  - Non-shared data in the same cache line so each update invalidates the cache line … in essence "sloshing independent data" back and forth between threads.
- Modify your "pi program" from exercise 2 to avoid false sharing due to the sum array.

Pi program with false sharing*

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

Example: A simple Parallel pi program

```
#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;
}
```

Recall that promoting sum to an array made the coding easy, but led to false sharing and poor performance.

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

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Example: Using a critical section to remove impact of false sharing

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

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes “out of scope” beyond the parallel region … so you must sum it in here. Must protect summation into pi in a critical region so updates don’t conflict

Results*: pi program critical section

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

Example: Using a critical section to remove impact of false sharing

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

threads	1st SPMD	1st SPMD padded	SPMD critical
1	1.86	1.86	1.87
2	1.03	1.01	1.00
3	1.08	0.69	0.68
4	0.97	0.53	0.53

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Example: Using a critical section to remove impact of false sharing

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

Be careful where you put a critical section

What would happen if you put the critical section inside the loop?

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?
  - This is called worksharing
    - Loop construct
    - Sections/section constructs
    - Single construct
    - Task construct

Discussed later

The loop worksharing Constructs

- **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);
        }
}
```

Loop construct name:

- C/C++: for
- Fortran: do

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

Loop worksharing Constructs
A motivating example

Sequential code

```
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 a worksharing for construct

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

loop worksharing constructs: The schedule clause

- **The schedule clause affects how loop iterations are mapped onto threads**
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - schedule(dynamic[,chunk])
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library).
  - schedule(auto)
    - Schedule is left up to the runtime to choose (does not have to be any of the above).

loop work-sharing constructs:
The schedule clause

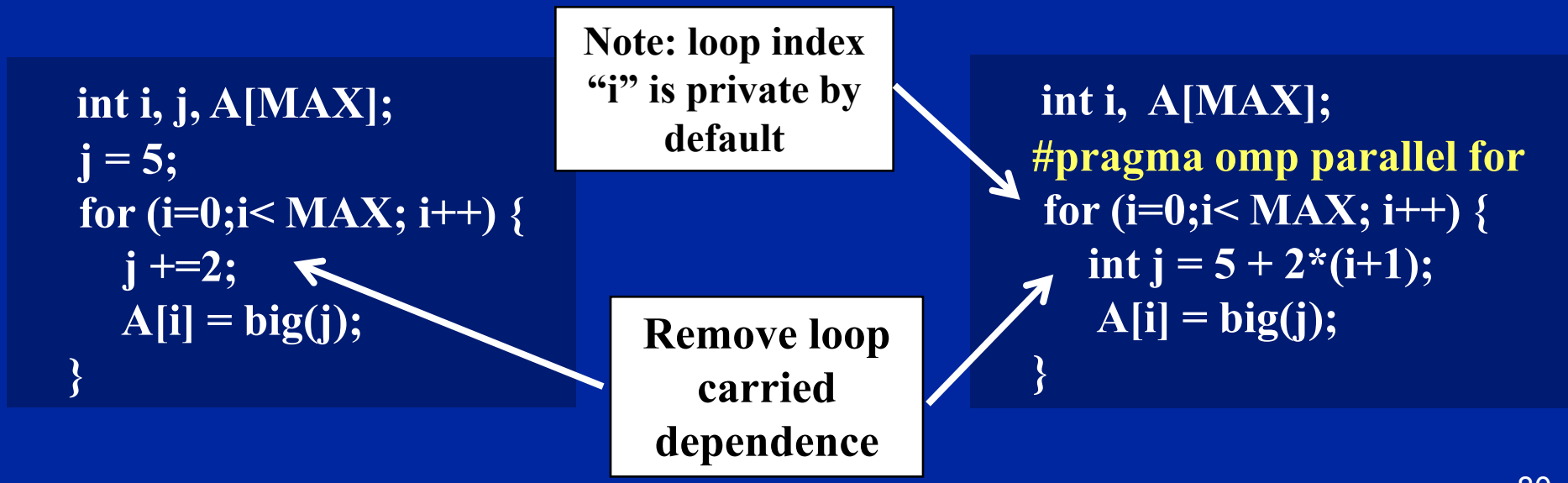
Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead
AUTO	When the runtime can "learn" from previous executions of the same loop

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

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



Nested loops

- For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
  for (int j=0; j<M; j++) {
          .....
  }
}
```

Number of loops to be parallelized, counting from the outside

- Will form a single loop of length NxM and then parallelize that.
- Useful if N is O(no. of threads) so parallelizing the outer loop makes balancing the load difficult.

Reduction

- How do we handle this case?

```
double  ave=0.0, A[MAX];    int i;
 for (i=0;i< MAX; i++) {
       ave + = A[i];
 }
 ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation … it is called a "reduction".
- Support for reduction operations is included in most parallel programming environments.

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;
```

OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

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

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.