HPCSE - I

«OpenMP Programming Model - Part I»

Panos Hadjidoukas
Schedule and Goals

• 13.10.2017: OpenMP - part 1
  • study the basic features of OpenMP
  • able to understand and write OpenMP programs

• 20.10.2017: OpenMP - part 2
  • how OpenMP works
  • how to optimize OpenMP / parallel code
  • study and discuss more examples

“\textit{We need to create learning situations where we ask students to practice program reading, to predict program execution, and to understand program idioms.}”

Mark Guzdial, Communications of the ACM, Vol. 60 No. 6, Pages 10-11
doi:10.1145/3077227
Example 1

• Identify and fix any issues in the following OpenMP codes

```c
int A[N], B[N];
int auxdot = 0, dot = 0;

#pragma omp parallel
{
    #pragma omp for
    for (int i=0 ; i < N ; i++){
        auxdot += A[i]*B[i];
    }
    #pragma omp critical
    dot += auxdot;
}
```

```c
#pragma omp parallel
{
    if (omp_get_thread_num() % 2)
    {
        #pragma omp barrier
        // ...
    }
}
```
Example 2

• Implement an equivalent version of the following code without using OpenMP worksharing

```c
// double A[N];
// int i;

#pragma omp parallel for schedule(dynamic, 1)
for (i = 0; i < N; i++)
{
    A[i] = work(i);
}
```
Example 3

- Parallelize the following code using OpenMP

```c
void compute_max_density()
{
    // This routine finds the value of max density (max_rho) and
    // its location (max_i, max_j) – there are no duplicate values
    double max_rho;
    int max_i, max_j;
    max_rho = rho_[0];
    max_i = 0;
    max_j = 0;

    for (int i = 0; i < N_; ++i)
        for (int j = 0; j < N_; ++j)
            if (rho_[i*N_ + j] > max_rho)
                { max_rho = rho_[i*N_ + j];
                  max_i = i;
                  max_j = j;
                }
}
```
Outline

- Introduction to OpenMP
- Parallel regions
- Worksharing constructs
  - loops, sections, single
- Combined parallel worksharing
- Data environment
- Synchronization
  - critical, atomic, barrier, master
- Library routines
- Environment variables
- Examples
OpenMP

- OpenMP: An Application Program Interface (API) for writing multithreaded applications
  - simple, portable, widely supported standard
  - facilitates the development of multithreaded code in Fortran, C and C++
  - suitable for shared memory platforms

- Three primary components
  - compiler directives - instruct the compiler to generate multithreaded code
  - library calls
  - environment variables
## Evolution of OpenMP

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oct 1997</td>
<td>Fortran 1.0</td>
</tr>
<tr>
<td>Oct 1998</td>
<td>C/C++ 1.0</td>
</tr>
<tr>
<td>Nov 1999</td>
<td>Fortran 1.1</td>
</tr>
<tr>
<td>Nov 2000</td>
<td>Fortran 2.0</td>
</tr>
<tr>
<td>Mar 2002</td>
<td>C/C++ 2.0</td>
</tr>
<tr>
<td>May 2005</td>
<td>OpenMP 2.5</td>
</tr>
<tr>
<td>May 2008</td>
<td>OpenMP 3.0</td>
</tr>
<tr>
<td>Jul 2011</td>
<td>OpenMP 3.1</td>
</tr>
<tr>
<td>Jul 2013</td>
<td>OpenMP 4.0</td>
</tr>
<tr>
<td>Nov 2015</td>
<td>OpenMP 4.5</td>
</tr>
</tbody>
</table>

- OpenMP specifications at [www.openmp.org](http://www.openmp.org)
  - OpenMP 4.0 (2013): Examples in a separate PDF file
Syntax Format

- Compiler directives
  - C/C++
    - `#pragma omp construct [clause [clause] ...]`
  - Fortran
    - `C$OMP construct [clause [clause] ... ]`
    - `!$OMP construct [clause [clause] ... ]`
    - `*$OMP construct [clause [clause] ... ]`

- Since we use directives, **no changes** need to be made to a program for a compiler that does not support OpenMP
OpenMP Directive

• Program executes serially until it encounters a parallel directive
  
  #pragma omp parallel [clause list]
  /* structured block of code */

• Clause list is used to specify conditions
  - Conditional parallelism: if (cond)
  - Degree of concurrency: num_threads(int)
  - Data handling: private(vlist), firstprivate(vlist), shared(vlist)
Programming Model

• Fork-join type of parallelism:
  – The master thread spawns teams of threads according to the user/application requirements
  – Parallelism is added incrementally
    • the sequential code is transformed to parallel

http://computing.llnl.gov/tutorials/openMP/
Typical Usage

• OpenMP is generally used for loop parallelization
  – Find the most time-consuming loops
  – Distribute the loop iterations to the threads

Assign this loop to different threads

```c
void main()
{
    double Res[1000];
    for (int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

Sequential code

```c
void main()
{
    double Res[1000];
    #pragma omp parallel for
    for (int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

Parallel code

But OpenMP is not just that!
Using OpenMP

• Some compilers can automatically place directives with option
  - –qsmp=auto (IBM xlc)
  - some loops may speed up, some may slow down

• Compiler option required when you use directives
  - –fopenmp (GNU compilers)
  - –openmp (Intel compilers)
  - –qsmp=omp (IBM)

• Scoping variables can be sometimes the hard part!
  - shared variables, thread private variables
Hello World!

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

int main() {
    #pragma omp parallel
    {
        int me = omp_get_thread_num();
        int nthr = omp_get_num_threads();
        printf("Hello world from thread \%d of \%d\n", me, nthr);
    }
    return 0;
}
```

- Compilation with the GNU GCC and Intel compilers
  ```bash
  $ gcc -fopenmp -o hello hello.c
  $ icc -openmp -o hello hello.c
  ```

MacOS: brew install gcc
Usage

• Execution

$ export OMP_NUM_THREADS=4
$ ./hello
Hello world from thread 0 of 4
Hello world from thread 2 of 4
Hello world from thread 1 of 4
Hello world from thread 3 of 4
$ export OMP_NUM_THREADS=1
$ ./hello
Hello world from thread 0 of 1
Thread Interaction

• OpenMP is a shared-memory programming model
  – Threads communicate through shared variables
• Data sharing can lead to \textit{race conditions}
  – the output of some code can change due to thread scheduling, e.g. their order of execution
• Synchronization at the right places can eliminate race conditions
  • However, \textit{synchronization is expensive}
  • the way data is stored might need to change to minimize the need for synchronization
OpenMP Directives

• 5 categories
  • Parallel Regions
  • Worksharing
  • Data Environment
  • Synchronization
  • Runtime functions & environment variables

• Basically the same between C/C++ and Fortran
Parallel Regions

- Create threads with `omp parallel`
- The following code will create a parallel region of 4 threads:

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

- Threads share A (default behavior)
- Master thread creates the threads
- Threads all start at same time then synchronize at a barrier at the end to continue with code
- Each threads calls pooh for its own ID (0 to 3)
Parallel Regions

- Each thread runs the same code
- All threads share A
- Execution continues when all threads have finished their work (barrier)

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

structured_block

Clauses

if (scalar_expression)
num_threads (integer-expression)
private (list)
shared (list)
firstprivate (list)
default (shared | none)
reduction (operator: list)
copyin (list)
Structured Blocks

• Most OpenMP directives are applied to structured blocks of code
  – Structured block: piece of code with a single entry point at the beginning and a single exit point at the end.

```c
#pragma omp parallel
{
    int id = omp_get_thread_num();
    res[id] = work(id);
}
printf("after parallel\n");
```

Structured block

```c
#pragma omp parallel
{
    int id = omp_get_thread_num();
    res[id] = work(id);
    if (res[id] == 0) goto out;
}
out: printf("after parallel\n");
```

Unstructured block
### Clauses for omp parallel

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (scalar_expression)</td>
<td>Only parallelize if the expression is true. Can be used to stop parallelization if the work is too little</td>
</tr>
<tr>
<td>num_threads (integer-expression)</td>
<td>Set the number of threads</td>
</tr>
<tr>
<td>private (list)</td>
<td>The specified variables are thread-private</td>
</tr>
<tr>
<td>shared (list)</td>
<td>The specified variables are shared among all threads</td>
</tr>
<tr>
<td>firstprivate (list)</td>
<td>The specified variables are thread-private and initialized from the master thread</td>
</tr>
<tr>
<td>reduction (operator: list)</td>
<td>Perform a reduction on the thread-local variables and assign it to the master thread</td>
</tr>
<tr>
<td>default (shared</td>
<td>none)</td>
</tr>
</tbody>
</table>

```c
#pragma omp parallel private(i) shared(n) if(n > 10)
{
    //...
}
```
Actual Number of Threads

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  1. Evaluation of the `if` clause
  2. Setting of the `num_threads` clause
  3. Use of the `omp_set_num_threads()` library function
  4. Setting of the `OMP_NUM_THREADS` environment variable
  5. Implementation default - usually the number of CPUs on a node, though it could be dynamic.

- Reminder: threads are numbered from 0 (master thread) to N-1
Static and Dynamic modes

• Dynamic mode (default):
  – The number of threads can differ between parallel regions of the same program
  – The specified number of threads actually defines the maximum number - the actual number of threads can be smaller

• Static mode:
  – The number of threads is fixed and exactly equal to the number specified by the programmer

• OpenMP supports nested parallel regions but…
  – The compiler is allowed to serialize all the inner levels
  – This means that it uses a single OpenMP thread for those parallel regions
Worksharing Constructs

- the for construct splits up loop iterations

```c
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i<N; ++i){
        do_work(i);
    }
}
```

- By default, there is a barrier at the end of the `omp for`.
- Use the `nowait` clause to turn off the barrier.
Rule

- In order to be made parallel, a loop must have canonical “shape”

```plaintext
for (index=start; index <= end; )
    <
    index++;
    ++index;
    index--;
    --index;
    index += inc;
    index -= inc;
    index = index + inc;
    index = inc + index;
    index = index – inc;
```
Sections construct

• The sections construct gives a different structured block to each thread

```c
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
    x_calculation();
#pragma omp section
    y_calculation();
#pragma omp section
    z_calculation();
}
```

• By default there is a barrier at the end. The `nowait` clause turns it off
Single construct

- The structured block is executed only by one of the threads
- An implicit barrier exists at the end of `single`
- Can be considered as a synchronization construct

```c
#pragma omp parallel
{
    do_many_things();
#pragma omp single
{
    exchange_boundaries();
}
    do_many_other_things();
}
```

implicit barrier here

and here, end of parallel region
Combined Directives

• Parallel regions can be combined with the `for` and sections worksharing constructs

• `omp parallel + omp for → omp parallel for`

```c
#pragma omp parallel for
for (i=0; i<N; i++){
    do_work(i);
}
```
Combined Directives

- `omp parallel + omp sections → omp parallel sections`

```c
#pragma omp parallel sections
{
#pragma omp section
    x_calculation();
#pragma omp section
    y_calculation();
#pragma omp section
    z_calculation();
}
```
Directive Scoping

- OpenMP directives can be extended in multiple files
- Orphan directives: appear outside a parallel region

```c
//foo.c

#pragma omp parallel
{
    whoami();
}

//bar.c

void whoami()
{
    int iam = omp_get_thread_num();
    #pragma omp critical synchronization
    {
        printf("Hello from %d\n", iam);
    }
    return;
}
```

- foo.c: Static (lexical) extent of parallel region
- bar.c: Dynamic extent of parallel region
Data Scoping

- **OpenMP** is a shared memory programming model
  - most variables are shared by default
- Global variables are shared
- But not everything is shared
  - loop index variables
  - stack variables in called functions from parallel region
Storage Attributes

- The programmer can change the storage attributes of variables with the following clauses:
  - shared
  - private
  - firstprivate
  - threadprivate

- The value of a private variable used in a parallel loop can be exported as global value with the clause:
  - lastprivate

- The default behavior can be changed using:
  - default(private | shared | none)

- The data clauses are applied to the parallel region and worksharing constructs - however, shared is only valid for parallel regions.

- Data scoping clauses are valid only in the lexical extent of the OpenMP directive.
Data Environment

- Example of `private` and `firstprivate`

```c
int A, B, C;
A = B = C = 1;
#pragma omp parallel private(B) firstprivate(C)
{
    // ...
}
```

- Within the parallel region:
  - “A” is shared between threads and equal to 1
  - Both “B” and “C” are private for each thread
    - B has undefined initial value
    - C has initial value equal to 1

- After the parallel region:
  - Both B and C have the same value as before the parallel region
private

• **private**(var) creates a private copy of var in each thread
  
  – The value of the copy is not initialized
  – The private copy is not related to the original variable with respect to the memory location

```c
int is = 0;
#pragma omp parallel for private(is)
for (int j=1; j<=1000; j++)
    is = is + j;

printf("%d\n", is);
```

• IS has not been initialized inside the loop
firstprivate

- **firstprivate**: special case of **private**
  - The private copy of each thread is initialized with the value of the original variable, which belongs to the master thread

    ```c
    int is = 0;
    #pragma omp parallel for firstprivate(is)
    for (int j=1; j<=1000; j++)
        is = is + j;
    printf("%d\n", is);
    ```

- Each thread has a private copy of IS with initial value 0
lastprivate

• Copies the value of the private variable, as assigned by the last loop iteration, to the original (global) variable

```c
int is = 0;
#pragma omp parallel for firstprivate(is) \ lastprivate(is)
for (int j=1; j<=1000; j++)
    is = is + j;

printf("%d\n", is);
```

• Each thread has a private copy of IS with initial value 0
• IS has the value it was assigned by the last loop iteration (i.e. for j=1000)
Synchronization

• OpenMP supports several synchronization constructs:
  – critical section
  – atomic
  – barrier
  – master (in fact, not a synchronization construction)
  – ordered not studied
  – flush not studied
Synchronization – critical

- No two threads will simultaneously be in the critical section
- Critical sections can be named
  - `omp critical (name)`

```c
#pragma omp parallel for private(b) shared(res)
for (i=0; i<niters; i++) {
    b = doit(i);
    #pragma omp critical
    {
        update(b, &res);
    }
}
```

lock mutex
unlock mutex

res: initialized before the parallel region
Synchronization – atomic

• Special case of critical section that can be used only for simple instructions.
• Can be applied only when a single memory location (variable) is updated

```c
#pragma omp parallel private(b)
{
    int i = omp_get_thread_num();
b = doit(i);
    #pragma omp atomic
    res = res + b;
}
```

use of some hardware-supported atomic operation

res: initialized before the parallel region
Synchronization – barrier

• Barrier: all threads wait until each thread has reached the barrier

```c
#pragma omp parallel shared (A, B) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier

    #pragma omp for
    for(int i=0; i<N; i++){
        B[i]=big_calc2(i,A);
    }
}
```

initialization of A
necessary synchronization
these computations depend on A
Synchronization – master

- The structured block is executed only by the master thread - the other threads of the team ignore it
- There is no barrier at the end of master

```c
#pragma omp parallel
{
  do_many_things();
  #pragma omp master
  {
    exchange_boundaries();
  }
  #pragma barrier
  do_many_other_things();
}
```
• A barrier is implicitly called at the end of the following constructs:
  – **parallel**
  – **for** (except when nowait is used)
  – **sections** (except when nowait is used)
  – **single** (except when nowait is used)

  - **for**, **sections** and **single** accept the **nowait** clause

    ```
    int nthreads;
    #pragma omp parallel
    #pragma omp single nowait
    nthreads = omp_get_num_threads();
    ```
Reductions

• The reduction clause modifies the way variables are “shared”:
  – `reduction (op : list)`
• Variables included in `list` must be shared in the parallel region where the reduction clause exists
• Allowed reduction operations: `+,-,*,&,^,|,&&,||,min, max`
• Within a parallel region or a worksharing construct:
  – A local copy for each variable in the list is created and initialized accordingly to the reduction operation
    – 0 for “+”
  – The values of the local copies are combined (reduced) to a single value that is stored to the original variable after the end of the construct
Reduction - Example

```c
#include <omp.h>
#define NUM_THREADS 2

double func(int i);

void main ()
{
    int i;
    double ZZ, res=0.0;
    omp_set_num_threads(NUM_THREADS);

#pragma omp parallel for reduction(+:res) private(ZZ)
    for (i=0; i< 1000; i++){
        ZZ = func(i);
        res = res + ZZ;
    }
}
```
Loop Scheduling

- **Usage:** `
#pragma omp parallel for <schedule clause>
  - schedule ( static | dynamic | guided [, chunk] )
  - schedule (runtime)

- **static [,chunk]**
  - Loop iterations are divided into segments of size `chunk` and distributed cyclically to the threads of the parallel region.
  - If `chunk` is not specified, it is equal to N/P and each thread executes a single chunk of iterations.

- **dynamic [,chunk]**
  - Loop iterations are divided into segments of size `chunk`.
  - An idle thread gets dynamically the next available chunk of iterations.

- **guided [,chunk]**
  - Similar to dynamic but the chunk size decreases exponentially.
  - `chunk` specifies the minimum segment size.

- **runtime**
  - Decide at runtime depending on the OMP_SCHEDULE environment variable.

- **auto**
  - Decided by the compiler and/or the underlying OpenMP runtime library.
```c
#pragma omp parallel for num_threads(4) schedule(*)
for (int i = 0; i < 500; i++) do_work(i);
```

Example

500 iterations on 4 threads

Thread ID

* = guided, 5

* = dynamic, 5

* = static

Multiple chunks

More details in the next lecture and the exercises
Library Calls

- **OpenMP locks**
  - `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`

- **Functions that control the runtime environment:**
  - Number of threads
    - `omp_set_num_threads()`, `omp_get_num_threads()`, `omp_get_thread_num()`, `omp_get_max_threads()`
  - Dynamic mode and nested parallelism
    - `omp_set_dynamic()`, `omp_set_nested()`, `omp_get_dynamic()`, `omp_get_nested()`
  - Check if code is in a parallel region
    - `omp_in_parallel()`
  - Number of processors / cores
    - `omp_get_num_procs()`

- **Wall-clock time measurement (in seconds)**
  - `omp_get_wtime()`
OpenMP Locks

omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int tmp = do_lots_of_work(id);

    omp_set_lock(&lck);
    printf("%d %d\n", id, tmp);
    omp_unset_lock(&lck);
}

omp_destroy_lock(&lck);
Libraries Calls

Dynamic mode is disabled and then the number of threads is specified. This ensures that the parallel region will have 4 threads.

```c
#include <omp.h>
void main()
{
    omp_set_dynamic(0);
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        int id=omp_get_thread_num();
        do_lots_of_stuff(id);
    }
}
```
Environment Variables

- Default number of threads
  - **OMP_NUM_THREADS** int Literal

- Control of dynamic mode
  - **OMP_DYNAMIC** TRUE || FALSE

- Control of nested parallelism
  - **OMP_NESTED** TRUE || FALSE

- Control of loop scheduling if the programmer has used *omp for schedule(RUNTIME)*
  - **OMP_SCHEDULE** “schedule[, chunk_size]”

- Control of threads binding
  - **OMP_PROC_BIND** TRUE || FALSE
«Test Cases»
Case 1: Loop & Parallel Region

- Parallelize the following sequential code with
  - parallel regions
  - worksharing

```c
#define N 1024
for(int i=0; i<N; i++) { a[i] = a[i] + b[i];}
```

- OpenMP parallel region

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

adjustment for the last thread
Loop & Worksharing

• Sequential code

```c
#define N 1024
for(int i=0; i<N; i++) { a[i] = a[i] + b[i];}
```

• OpenMP parallel region with worksharing

```c
#pragma omp parallel
{
    #pragma omp for schedule(static)  // default scheduling
    for(int i=0; i<N; i++) { a[i] = a[i] + b[i];}
}
```

or simply:

```c
#pragma omp parallel for
for(int i=0; i<N; i++) { a[i] = a[i] + b[i];}
```
Case 2: Functional parallelism

- Parallelize the following sequential code
  - what is the total execution time if each function takes one second?

```c
V = alpha();
W = beta();
X = gamma(V, W);
Y = delta();
printf("%f\n", epsilon(X,Y));
```

**total time = 5s**
#pragma omp parallel num_threads(3)
#pragma omp sections
{
    #pragma omp section
    V = alpha();

    #pragma omp section
    W = beta();

    #pragma omp section
    Y = delta();
}
X = gamma(V, W);
printf("%f\n", epsilon(X,Y));

total time = 3s
#pragma omp parallel num_threads(2)  
{
    #pragma omp sections
    {
        #pragma omp section
        V = alpha();

        #pragma omp section
        W = beta();
    }
    #pragma omp sections
    {
        #pragma omp section
        X = gamma(V, W);

        #pragma omp section
        Y = delta();
    }
}
printf("%f\n", epsilon(X,Y));
Case 3 - Reductions

• Parallelize the following sequential code

```c
long num_steps = 100000;
double step;

void main ()
{
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;

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

    printf("Pi is %lf\n", pi);
}
```

Using the reduction clause

long num_steps = 100000;
double step;

void main ()
{
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;

    #pragma omp parallel for reduction(+:sum) private(x)
    for (long i=0; i<num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;

    printf("Pi is %lf\n", pi);
}
Version with parallel region

long num_steps = 100000; double step;
#define NUM_THREADS 2

void main()
{
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);

#pragma omp parallel private(x)
{
    int id = omp_get_thread_num();
    sum[id]=0.0
    for (long i=id; i<num_steps; i+=NUM_THREADS) {
        x = (i+0.5)*step;
        sum[id] += 4.0/(1.0+x*x);
    }
}

pi=0.0;
for(int i=0; i<NUM_THREADS; i++) pi += sum[i]*step;
printf("Pi is %lf\n", pi);
}
long num_steps = 100000; double step;
define NUM_THREADS 2
void main ()
{
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);

#pragma omp parallel private(x)
    { int id = omp_get_thread_num();
        sum[id] = 0.0;
        #pragma omp for
        for (long i=0; i< num_steps; i++) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    pi=0.0;
    for(int i=0; i<NUM_THREADS; i++) pi += sum[i]*step;
    printf("Pi is %lf\n", pi);
}
False Sharing

- The previous two implementations suffer from *cache thrashing* due to *false sharing*
- False sharing degrades performance when all the following conditions occur:
  - Shared data is modified by multiple processors.
  - Multiple processors update data within the same cache line.
  - This updating occurs very frequently (for example, in a tight loop).

Source: Sun Studio 12: OpenMP API User's Guide
long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(x)
    {
        int id = omp_get_thread_num();
        double lsum=0;
        #pragma omp for
        for (long i=0; i<num_steps; i++){
            x = (i+0.5)*step;
            lsum = lsum + 4.0/(1.0+x*x);
        }
        sum[id] = lsum;
    }
    pi=0.0;
    for(int i=0; i<NUM_THREADS; i++) pi += sum[i]*step;
    printf("Pi is %lf\n", pi);
}
long num_steps = 100000;
double step;
#define NUM_THREADS 2

void main ()
{
    double x, pi, sum = 0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(x)
    {
        int id = omp_get_thread_num();
        double lsum=0;
        #pragma omp for
        for (long i=0; i<num_steps; i++){
            x = (i+0.5)*step;
            lsum = lsum + 4.0/(1.0+x*x);
        }
        #pragma omp atomic
        sum += lsum;
    }
    printf("Pi is %lf\n", pi);
}
References

• OpenMP Specifications & Quick Reference Card
  • www.openmp.org

• OpenMP tutorial at LLNL, Blaise Barney
  • https://computing.llnl.gov/tutorials/openMP/

• An Overview of OpenMP, Ruud van der Pas – Sun Microsystems