

Παράλληλη Επεξεργασία

Εαρινό Εξάμηνο 2022-23
«OpenMP - II»

Παναγιώτης Χατζηδούκας, Ευστράτιος Γαλλόπουλος

Goals

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

Outline

- Hands-On: finding the max
- False sharing and how to avoid it
- Synchronization: implicit barriers
- Parallel regions and thread management
- Tuning the OpenMP runtime library
 - Measuring OpenMP overheads
 - OMP_PROC_BIND: core binding
 - NUMA considerations
 - OMP_WAIT_POLICY: "sleep or spin?"
- *Loop scheduling policies*
- *Nested parallelism and the collapse clause*

I. Hands-On: finding the max

```
double find_max(double *A, int N)
{
    double mx = A[0];

    for (int i=0; i<N; i++){
        if (A[i] > mx) mx = A[i];
    }

    return mx;
}
```

Using the reduction Clause

```
double find_max(double *A, int N)
{
    double mx = A[0];

    #pragma omp parallel for
    for (int i=0; i<N; i++){
        if (A[i] > mx) {
            #pragma omp critical
            {
                mx = A[i];
            }
        }
    }

    return mx;
}
```

Using parallel for

```
double find_max(double *A, int N)
{
    int nthreads;
    #pragma omp parallel
    #pragma omp master
    nthreads = omp_get_num_threads();

    double mx = A[0];
    double local_mx[nthreads]; // false sharing (discussed later)
    for (int i = 0; i < nthreads; i++) local_mx[i] = A[0];

    #pragma omp parallel for
    for (int i=0; i<N; i++) {
        int me = omp_get_thread_num(); // called too many times
        if (A[i] > local_mx[me]) { local_mx[me] = A[i]; }
    }

    for (int i = 0; i < nthreads; i++)
        if (local_mx[i] > mx) mx = local_mx[i];

    return mx;
}
```

Manual Reduction with critical

```
double find_max(double *A, int N)
{
    double mx = A[0];

    #pragma omp parallel
    {
        double local_mx = A[0];
        #pragma omp for
        for (int i=0; i<N; i++) {
            if (A[i] > local_mx) local_mx = A[i];
        }
        #pragma omp critical
        if (local_mx > mx) mx = local_mx;
    }
    return mx;
}
```

II. False Sharing

```
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);
}
```


Pi Computation with Worksharing

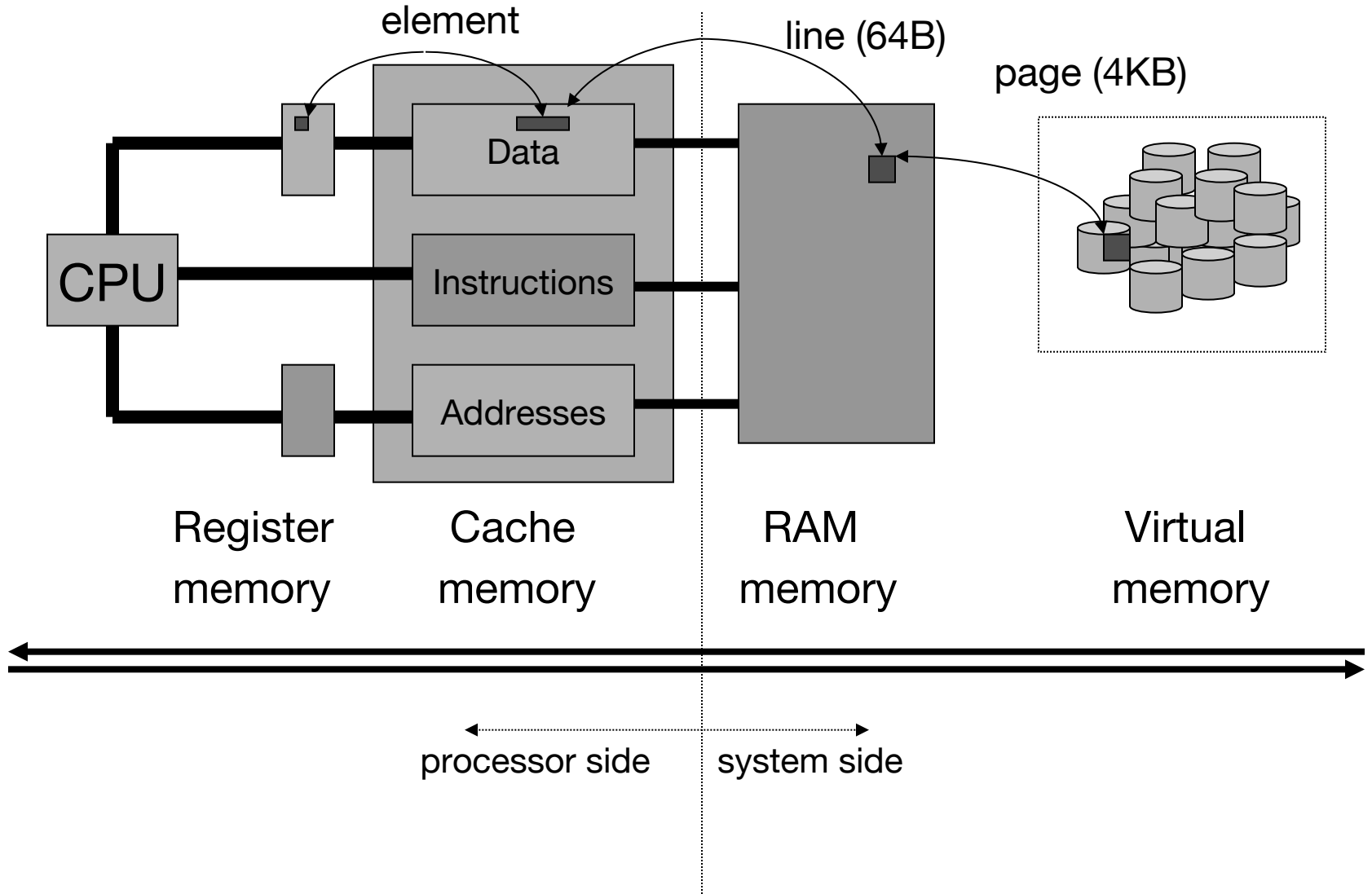
```
long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);

    #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;
}
```

threads write to
different but **neighboring**
memory locations

different: no race condition
neighboring: false sharing

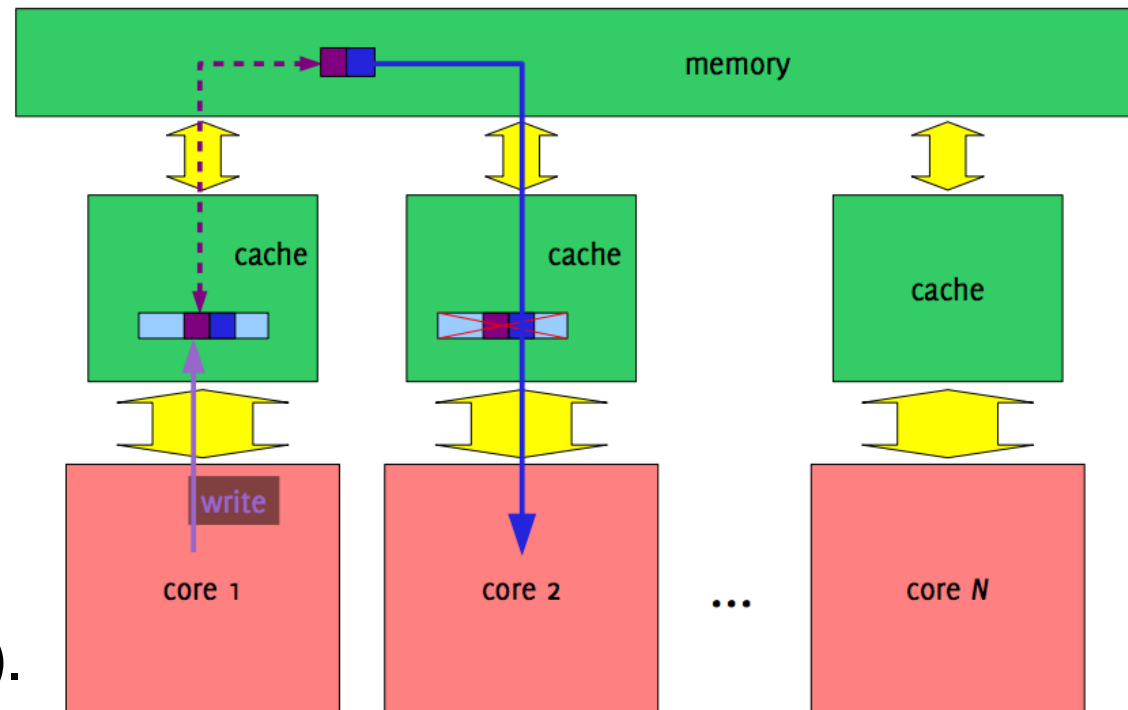
Memory Hierarchy



False Sharing

- The previous implementations suffers 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

Credit: C. L. Luengo Hendriks


Memory Padding

```
long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
    double x, pi, sum[NUM_THREADS][PADSIZE];
    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][0] += 4.0/(1.0+x*x);
      }
    }
    pi=0.0;
    for(int i=0; i<NUM_THREADS; i++) pi += sum[i][0]*step;
}
```

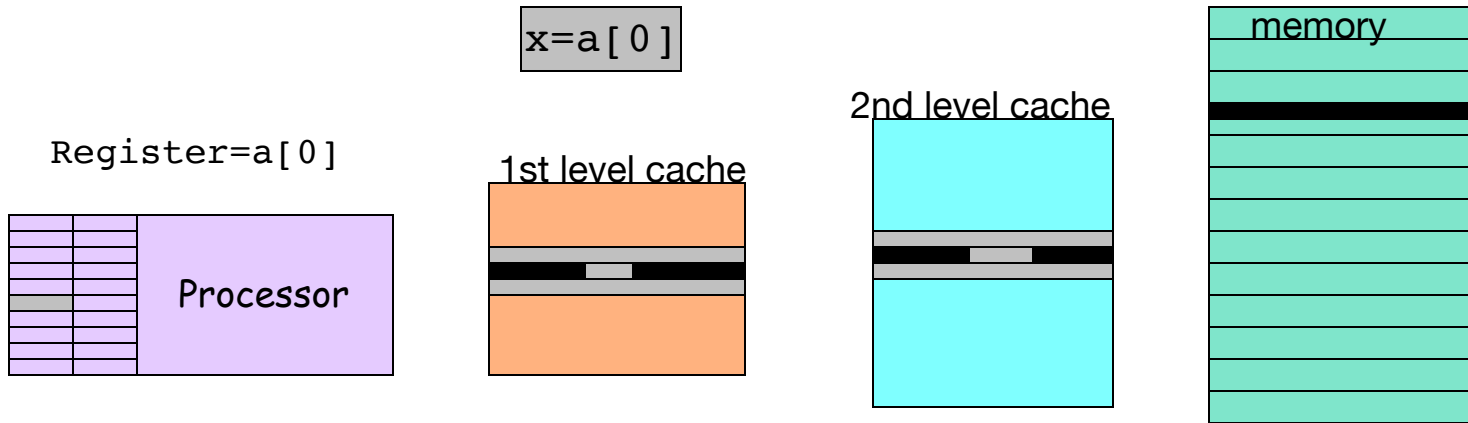
#define PADSIZE 8

adding some distance
between the updated
memory locations



Cache Lines

More than one element is transferred each time



- Transfer unit: cache line
- Cache line includes neighboring memory places
- Cache line size depends on the processor architecture
- Typical sizes
 - 32/64 bytes for 1st level cache
 - 64/128 bytes for 2nd level cache
 - Common case: 64 bytes for both levels

$$\text{PADSIZE} = 8 = 64/8 = \text{cache_line_size} / \text{sizeof(double)}$$

Worksharing + Local Sum

```
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;
}
```

goal: minimize accesses to sum[]

local sum is a private variable

each thread updates sum[] only once

Worksharing + Local Sum + Atomic

```
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;
    }
}
```

goal: minimize accesses to sum

atomic operations faster than
mutexes (omp critical)

Result of the reduction Clause

```
omp_set_num_threads(4);  
#pragma omp parallel for reduction(+:sum)  
for (long i=0; i<1000; i++){  
    sum += a[i];  
}
```

- 1 OpenMP thread (sequential code)
 - $\text{sum} = a[0] + a[1] + a[2] + \dots + a[1023]$
- 4 OpenMP threads and local sum
 - $\text{lsum0} = a[0] + a[1] + \dots + a[249]$
 - $\text{lsum1} = a[250] + a[251] + \dots + a[499]$
 - $\text{lsum2} = a[500] + a[501] + \dots + a[749]$
 - $\text{lsum3} = a[750] + a[751] + \dots + a[999]$
- array: sequential addition of the local sums
- atomic: addition of the local sums in any order

III. Synchronization - Implicit Barriers

- 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();
```

OpenMP Quiz 1

- Identify and fix any issues in the following OpenMP code

```
1 #pragma omp parallel
2 {
3     if( omp_get_thread_num() % 2 ){
4 #pragma omp barrier
5
6         // ...
7     }
8 }
```

OpenMP Quiz 1

- Identify and fix any issues in the following OpenMP code

```
1 #pragma omp parallel
2 {
3     if( omp_get_thread_num() % 2 ){
4 #pragma omp barrier
5
6         // ...
7     }
8 }
```

Every OpenMP parallel region has its own explicit barrier *
All threads must reach the barrier, otherwise deadlock occurs
This is the case for the above example

```
if (omp_get_thread_num() %2) {
    #pragma omp barrier
    // ...
}
else {
    #pragma omp barrier
}
```

possible solution:
now all threads reach the barrier

Implementation detail: the parallel region includes also barriers for the worksharing constructs (for, sections, single) - one for each construct ¹⁹

Explicit and Implicit barriers

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

    #pragma omp for
    for(i=0; i<N; i++){
        C[i]=big_calc3(I,A);
    }

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

    A[id] = big_calc3(id);
}
```

explicit barrier

end of omp for: implicit barrier

nowait: no barrier

end of parallel: implicit barrier

omp for nowait

```
#include <math.h>

void
a8(int n, int m, float *a, float *b, float *y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;

        #pragma omp for nowait
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
```

no data dependencies, a thread can
safely proceed to the next loop

omp single vs omp master

```
#pragma omp parallel  
{  
  do_many_things();
```

```
  #pragma omp single  
  exchange_boundaries();
```

executed by one of the threads

```
  do_many_other_things();
```

----- end of single: implicit barrier

```
#pragma omp parallel  
{
```

```
  do_many_things();  
  #pragma omp master  
  exchange_boundaries();
```

executed only by the master thread

```
  #pragma barrier  
  do_many_other_things();
```

necessary explicit barrier

```
}
```

omp single

```
#include <stdio.h>
void work1();
void work2();
```

```
void a10()
{
    #pragma omp parallel
    {
        #pragma omp single
        printf("Beginning work1.\n");
```

end of single: implicit barrier

```
work1();
        #pragma omp single
        printf("Finishing work1.\n");
```

end of single: implicit barrier

```
        #pragma omp single nowait
        printf("Finished work1 and beginning work2.\n");
```

no barrier

```
work2();
    }
}
```

end of parallel: implicit barrier

OpenMP Quiz 2

- Identify and fix any issues in the following OpenMP code

```
void do_work(int, float);    /* assume no barriers inside */
float new_value(int);
```

```
void testsingle()
```

```
{
```

```
    float t = 0;
```

```
    #pragma omp parallel
```

```
    {
```

```
        for (int step = 0; step < 100; step++)
```

```
        {
```

```
            //<probably some code here>
```

```
            #pragma omp barrier
```

```
            do_work(step, t);
```

```
            #pragma omp single
```

```
                t = new_value(step);
```

```
        }
```

```
    }
```

```
}
```

Hint: consider two threads in the parallel region, one being very fast and the other very slow

OpenMP Quiz 2 - Solution

- Identify and fix any issues in the following OpenMP code

```
void do_work(float);    /* assume no barriers inside */
double new_value(int);
```

```
void testsingle()
{
```

```
    float t = 0;
```

```
    #pragma omp parallel
```

```
        for (int step = 0; step < 100; step++)
```

```
        {
```

```
            //<some code here>
```

```
            #pragma omp barrier
```

```
            do_work(t);
```

```
            #pragma omp barrier
```

```
            #pragma omp single
```


```
                t = new_value(step);
```

```
        }
```

```
    }
```

Race condition:

A thread might execute
do_work() and new_value()
before another thread
executes do_work()



IV. Parallel Regions & Thread Management

- Use OpenMP to parallelize the following code

```
for (int timestep = 0; timestep < Nsteps; timestep++) {  
    for (int i = 0; i < N; i++)  
        work(timestep, i);  
}
```

evolution in time
parallel loop

- First approach

```
for (int timestep = 0; timestep < Nsteps; timestep++) {  
    #pragma omp parallel for  
    for (int i = 0; i < N; i++)  
        work(timestep, i);  
}
```

parallel region at every timestep

- Second approach

```
#pragma omp parallel  
{  
    for (int i = stand; i < end; i++)  
        work(timestep, i);  
}  
#pragma omp  
for (int i = stand; i < end; i++)  
    work(timestep, i);  
}
```

parallel region only once

Thread Management

- Spawning and joining thread is expensive
 - they are system calls to the operating system
- The OpenMP runtime library spawns threads only once
 - at the first parallel region
 - reuses the threads at the next parallel regions
- This means that after the end of a parallel region
 - only the master thread continues
 - the other threads become idle, waiting to execute the work defined by the next parallel region

```
#pragma omp parallel
{
    // first parallel region
}

#pragma omp parallel
{
    // second parallel region
}
```

Thread Management - Example

- Shows the mapping between OpenMP and POSIX threads

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

#define OMP_ID omp_get_thread_num()
#define PTHREAD_ID pthread_self()

int main()
{
    printf("main(),          thread=%d, pthread_t=%lx\n", OMP_ID, PTHREAD_ID);

    #pragma omp parallel
    {
        printf("1st region, thread=%d, pthread_t=%lx\n", OMP_ID, PTHREAD_ID);
    }

    #pragma omp parallel
    {
        printf("2nd region: thread=%d, pthread_t=%lx\n", OMP_ID, PTHREAD_ID);
    }
    return 0;
}
```

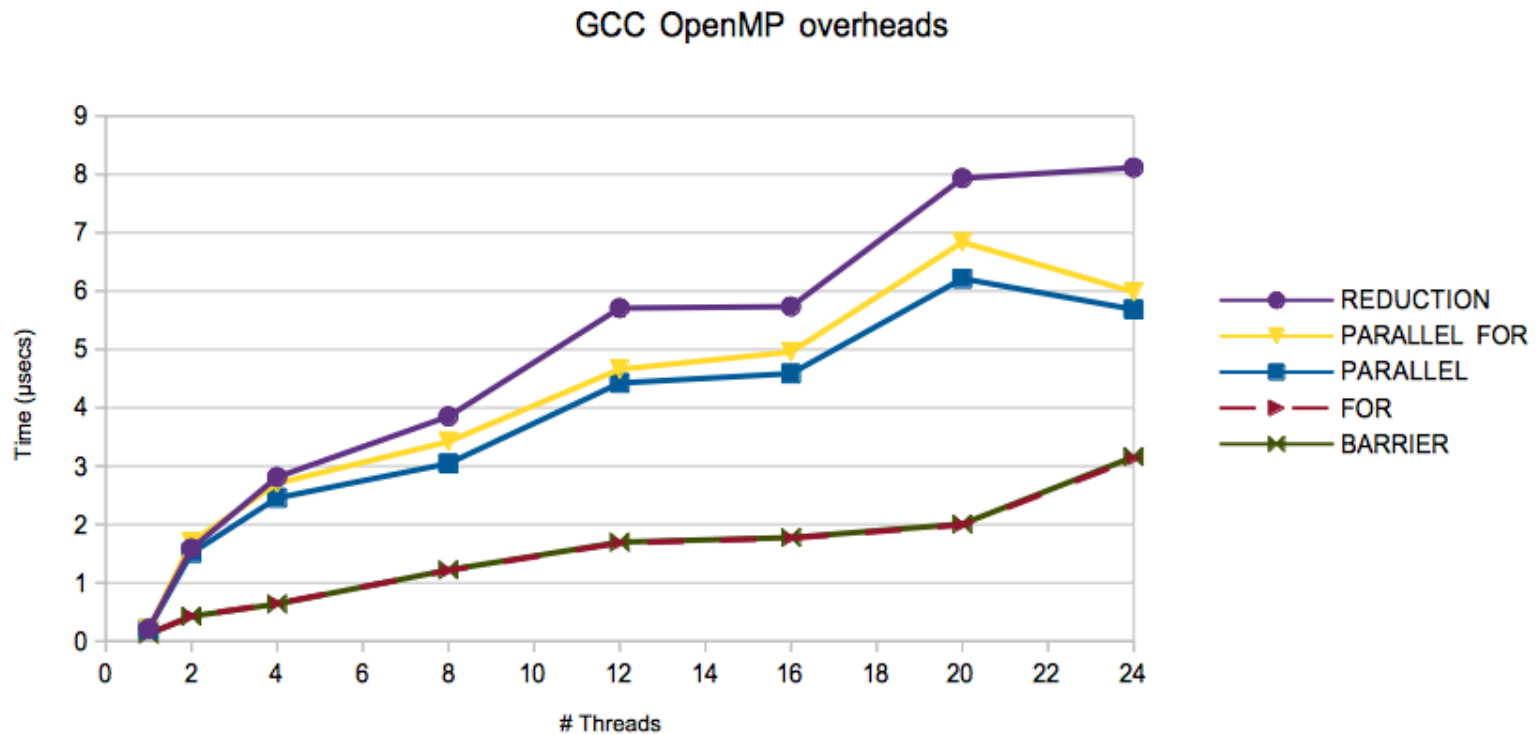
we observe the same pthread_t values

V. Tuning the OpenMP runtime library

- Question: what is the cost of spawning a parallel region?
- EPCC OpenMP micro-benchmark suite
 - <https://www.epcc.ed.ac.uk/>
- Measures overhead of
 - synchronization
 - parallel, for, parallel for, barrier, critical, reduction...
 - loop scheduling
 - {static, dynamic, guided} + various chunk size
 - tasking
 - not covered this semester

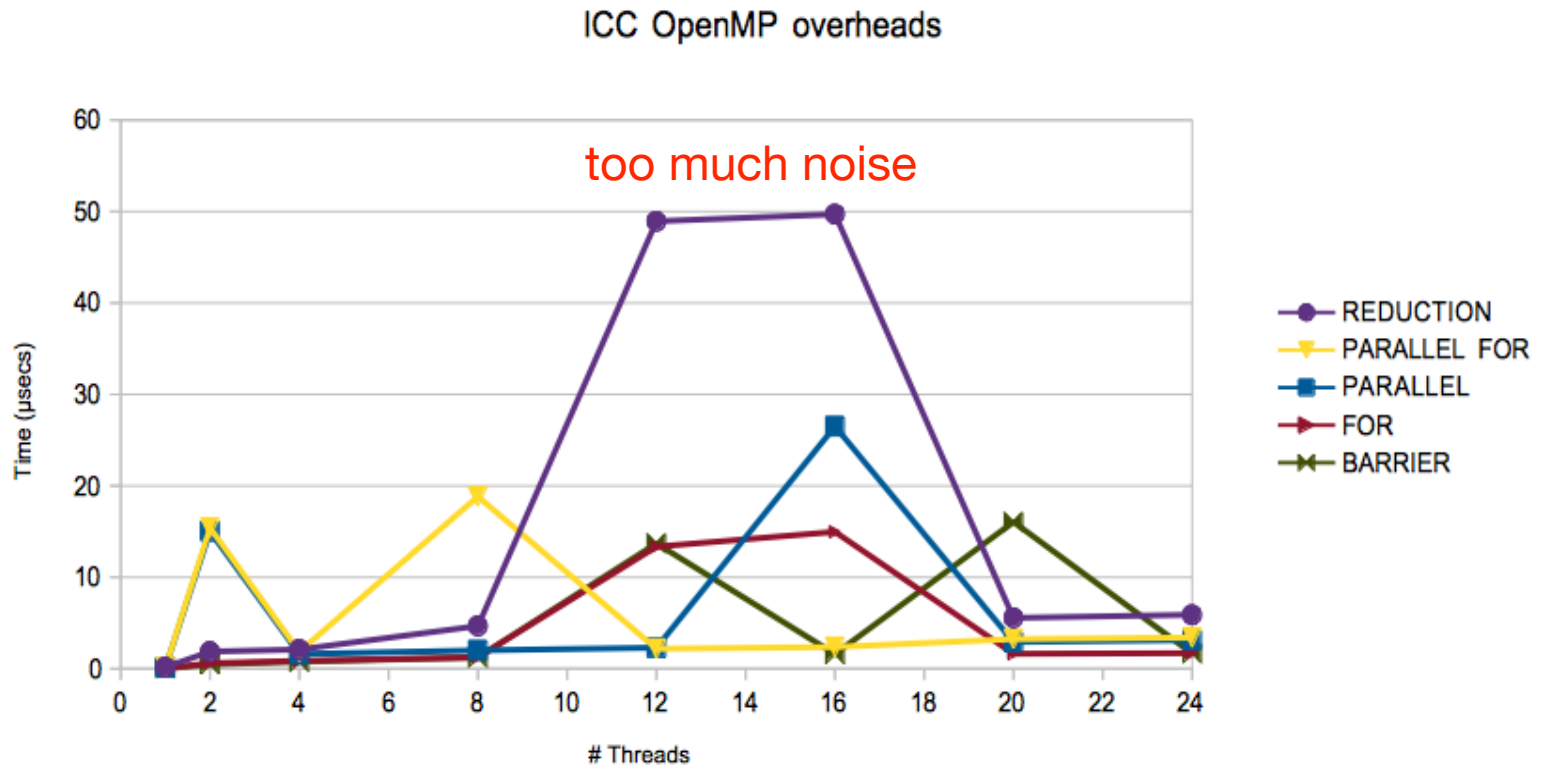
OpenMP Overheads on Euler - I

- Results for default runtime options



OpenMP Overheads on Euler - II

- Results for default runtime options



Thread-Core Binding

- `OMP_PROC_BIND`: “Supported since OpenMP 3.0. Set to `TRUE` to bind threads to processors and disable migration to other processors. Important on NUMA architectures”

```
void test_proc_bind()
{
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int core = sched_getcpu(); /* linux specific */

        #pragma omp critical
        printf("Thread %d running on core %d\n", tid, core);
    }
}
```

Execution on a 8-core system

```
$ export OMP_PROC_BIND=FALSE && ./getcpu_linux | sort -n -k 2
```

```
Thread 0 running on core 3
```

```
Thread 1 running on core 5
```

```
Thread 2 running on core 7
```

```
Thread 3 running on core 3
```

```
Thread 4 running on core 0
```

```
Thread 5 running on core 4
```

```
Thread 6 running on core 2
```

```
Thread 7 running on core 6
```

no binding, the can run anywhere

thread 3 or 0 finished its work, reached the barrier at the end of the parallel region and released core #3

```
$ export OMP_PROC_BIND=TRUE && ./getcpu_linux | sort -n -k 2
```

```
Thread 0 running on core 0
```

```
Thread 1 running on core 1
```

```
Thread 2 running on core 2
```

```
Thread 3 running on core 3
```

```
Thread 4 running on core 4
```

```
Thread 5 running on core 5
```

```
Thread 6 running on core 6
```

```
Thread 7 running on core 7
```

12 OpenMP threads on 8 cores

```
$ export OMP_NUM_THREADS=12  && export OMP_PROC_BIND=TRUE  &&  
./getcpu_linux | sort -n -k 2
```

Thread 0 running on core 0

Thread 1 running on core 1

Thread 2 running on core 2

Thread 3 running on core 3

Thread 4 running on core 4

Thread 5 running on core 5

Thread 6 running on core 6

Thread 7 running on core 7

Thread 8 running on core 0

Thread 9 running on core 1

Thread 10 running on core 2

Thread 11 running on core 3

processor/core oversubscription

16 OpenMP threads on 8 cores

```
$ export OMP_NUM_THREADS=16  && export OMP_PROC_BIND=TRUE  &&  
./getcpu_linux | sort -n -k 2
```

Thread 0 running on core 0

Thread 1 running on core 0

processor/core oversubscription

Thread 2 running on core 1

Thread 3 running on core 1

Thread 4 running on core 2

Thread 5 running on core 2

Thread 6 running on core 3

Thread 7 running on core 3

Thread 8 running on core 4

Thread 9 running on core 4

Thread 10 running on core 5

Thread 11 running on core 5

Thread 12 running on core 6

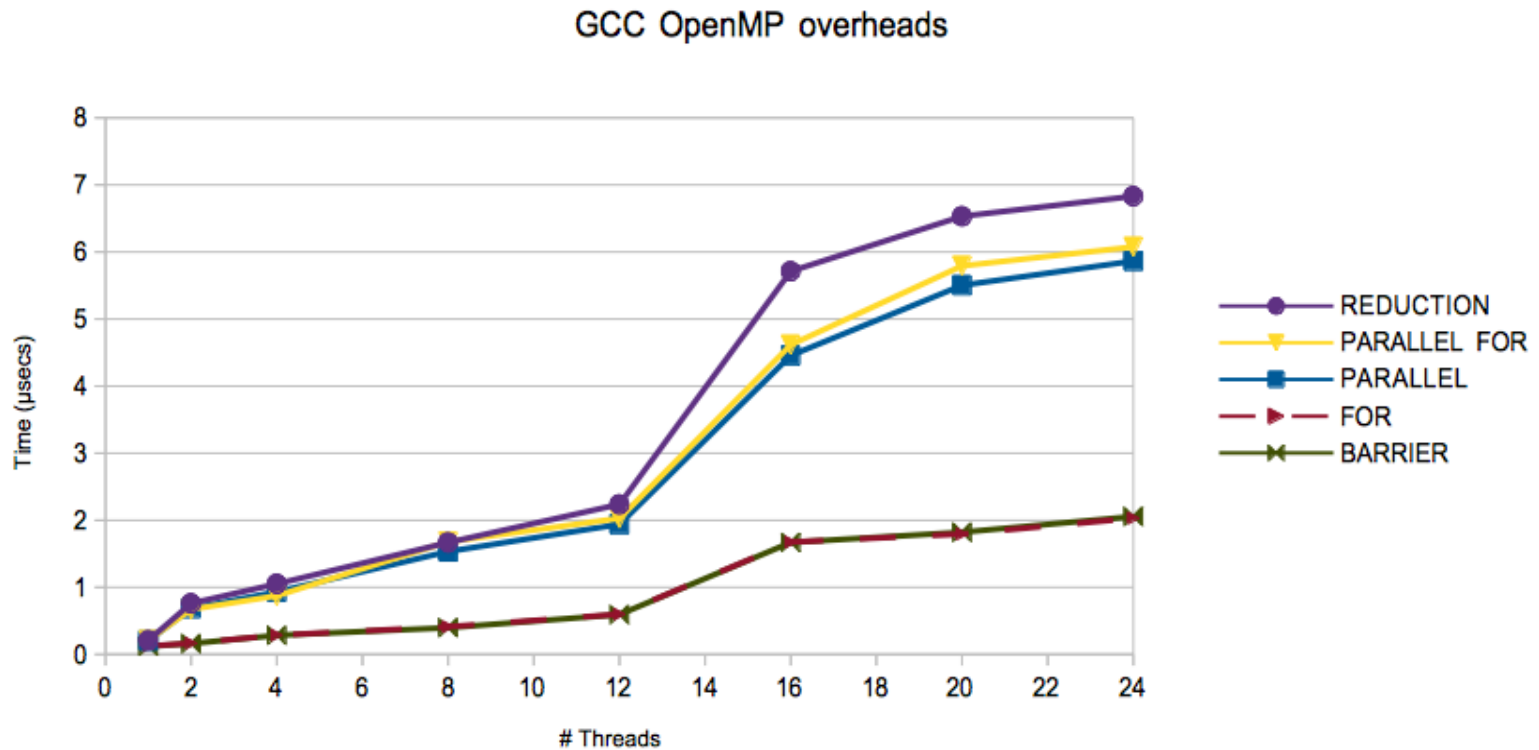
Thread 13 running on core 6

Thread 14 running on core 7

Thread 15 running on core 7

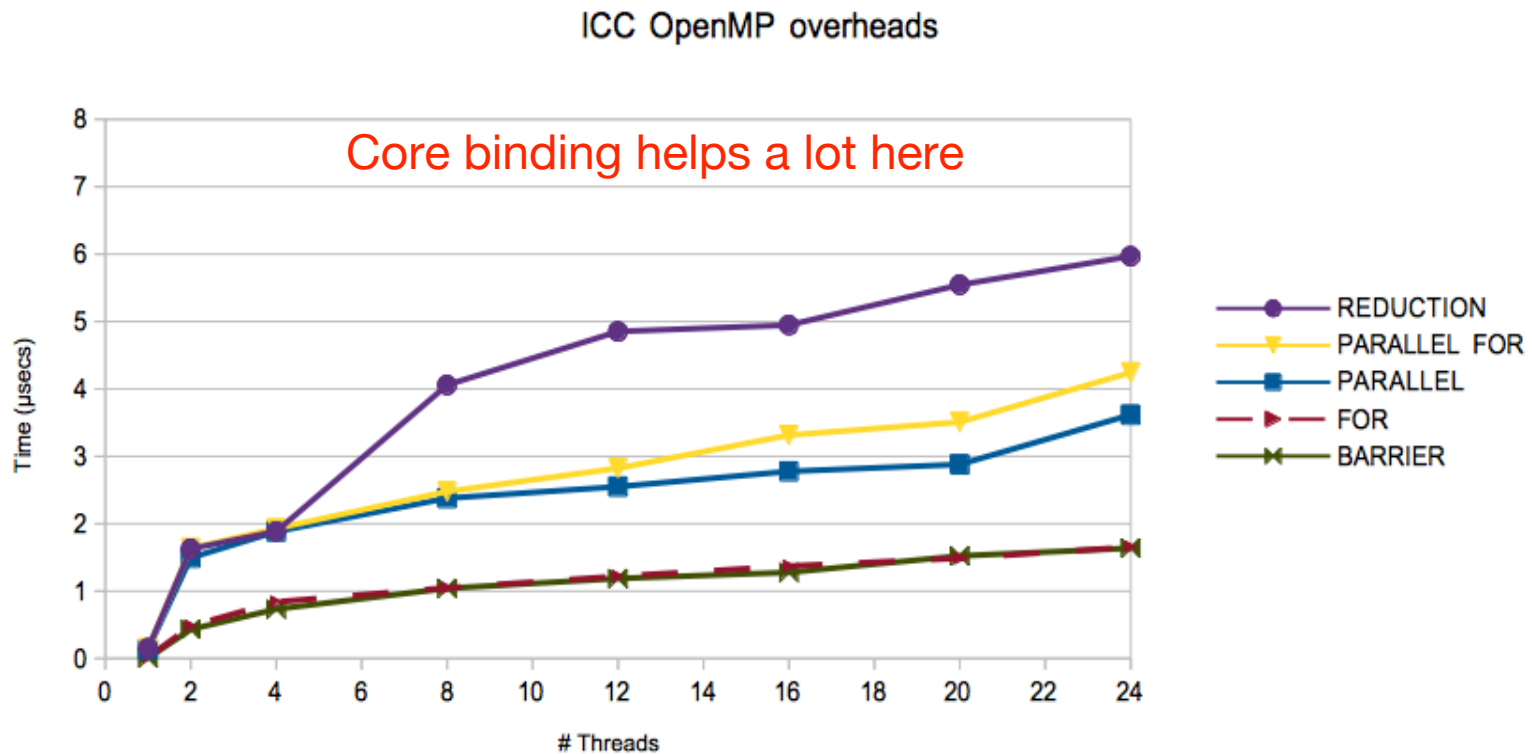
OpenMP Overheads on Euler - III

- Results for default runtime options and OMP_PROC_BIND=TRUE



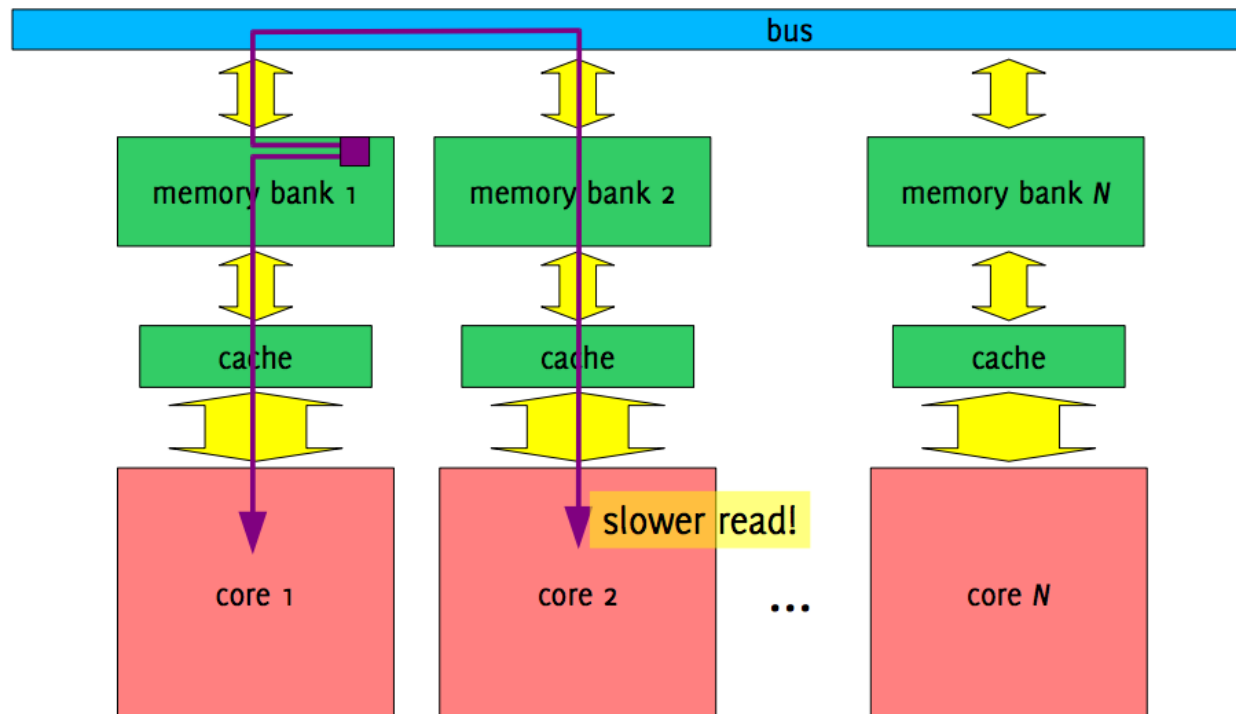
OpenMP Overheads on Euler - IV

- Results for default runtime options and OMP_PROC_BIND=TRUE



NUMA and First Touch

- Memory affinity is not decided by the memory allocation but by the initialization
- First-touch principle: memory mapped to the NUMA domain that first touches the data



Credit: C. L. Luengo Hendriks

NUMA on Euler

Euler (2016)

```
[[chatzidp@e1329 ]$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11
node 0 size: 32733 MB
node 0 free: 30899 MB
node 1 cpus: 12 13 14 15 16 17 18 19 20 21 22 23
node 1 size: 32767 MB
node 1 free: 32025 MB
node distances:
node  0  1
  0:  10  20
  1:  20  10
```

Hyper-threading enabled (virtual cores)

Euler (2017)

```
[[chatzidp@eu-c7-021-14 ~]$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 24 25 26 27 28 29 30 31 32 33 34 35
node 0 size: 32733 MB
node 0 free: 26453 MB
node 1 cpus: 12 13 14 15 16 17 18 19 20 21 22 23 36 37 38 39 40 41 42 43 44 45 46 47
node 1 size: 32767 MB
node 1 free: 24752 MB
node distances:
node  0  1
  0:  10  20
  1:  20  10
```


NUMA on Euler

```
[[chatzidp@eu-c7-021-14 ~]$ hwloc-ls
Machine (64GB total)
  NUMANode L#0 (P#0 32GB)
    Package L#0 + L3 L#0 (30MB)
      L2 L#0 (256KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0
      PU L#0 (P#0)
      PU L#1 (P#24)
      L2 L#1 (256KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1
      PU L#2 (P#1)
      PU L#3 (P#25)
      L2 L#2 (256KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2
      PU L#4 (P#2)
      PU L#5 (P#26)
      L2 L#3 (256KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3
      PU L#6 (P#3)
      PU L#7 (P#27)
      L2 L#4 (256KB) + L1d L#4 (32KB) + L1i L#4 (32KB) + Core L#4
      PU L#8 (P#4)
      PU L#9 (P#28)
      L2 L#5 (256KB) + L1d L#5 (32KB) + L1i L#5 (32KB) + Core L#5
      PU L#10 (P#5)
      PU L#11 (P#29)
      L2 L#6 (256KB) + L1d L#6 (32KB) + L1i L#6 (32KB) + Core L#6
      PU L#12 (P#6)
      PU L#13 (P#30)
      L2 L#7 (256KB) + L1d L#7 (32KB) + L1i L#7 (32KB) + Core L#7
      PU L#14 (P#7)
      PU L#15 (P#31)
      L2 L#8 (256KB) + L1d L#8 (32KB) + L1i L#8 (32KB) + Core L#8
      PU L#16 (P#8)
      PU L#17 (P#32)
      L2 L#9 (256KB) + L1d L#9 (32KB) + L1i L#9 (32KB) + Core L#9
      PU L#18 (P#9)
      PU L#19 (P#33)
      L2 L#10 (256KB) + L1d L#10 (32KB) + L1i L#10 (32KB) + Core L#10
      PU L#20 (P#10)
      PU L#21 (P#34)
      L2 L#11 (256KB) + L1d L#11 (32KB) + L1i L#11 (32KB) + Core L#11
      PU L#22 (P#11)
      PU L#23 (P#35)
  HostBridge L#0
    PCIBridge
      PCI 14e4:168e
        Net L#0 "eth0"
      PCI 14e4:168e
        Net L#1 "eth1"
    PCIBridge
      PCI 103c:323b
        Block(Disk) L#2 "sda"
    PCIBridge
      PCI 15b3:1003
        Net L#3 "ib0"
        Net L#4 "ib1"
        OpenFabrics L#5 "mlx4_0"
    PCIBridge
      PCI 102b:0533
        GPU L#6 "card0"
        GPU L#7 "controlD64"
```

```
NUMANode L#1 (P#1 32GB) + Package L#1 + L3 L#1 (30MB)
  L2 L#12 (256KB) + L1d L#12 (32KB) + L1i L#12 (32KB) + Core L#12
  PU L#24 (P#12)
  PU L#25 (P#36)
  L2 L#13 (256KB) + L1d L#13 (32KB) + L1i L#13 (32KB) + Core L#13
  PU L#26 (P#13)
  PU L#27 (P#37)
  L2 L#14 (256KB) + L1d L#14 (32KB) + L1i L#14 (32KB) + Core L#14
  PU L#28 (P#14)
  PU L#29 (P#38)
  L2 L#15 (256KB) + L1d L#15 (32KB) + L1i L#15 (32KB) + Core L#15
  PU L#30 (P#15)
  PU L#31 (P#39)
  L2 L#16 (256KB) + L1d L#16 (32KB) + L1i L#16 (32KB) + Core L#16
  PU L#32 (P#16)
  PU L#33 (P#40)
  L2 L#17 (256KB) + L1d L#17 (32KB) + L1i L#17 (32KB) + Core L#17
  PU L#34 (P#17)
  PU L#35 (P#41)
  L2 L#18 (256KB) + L1d L#18 (32KB) + L1i L#18 (32KB) + Core L#18
  PU L#36 (P#18)
  PU L#37 (P#42)
  L2 L#19 (256KB) + L1d L#19 (32KB) + L1i L#19 (32KB) + Core L#19
  PU L#38 (P#19)
  PU L#39 (P#43)
  L2 L#20 (256KB) + L1d L#20 (32KB) + L1i L#20 (32KB) + Core L#20
  PU L#40 (P#20)
  PU L#41 (P#44)
  L2 L#21 (256KB) + L1d L#21 (32KB) + L1i L#21 (32KB) + Core L#21
  PU L#42 (P#21)
  PU L#43 (P#45)
  L2 L#22 (256KB) + L1d L#22 (32KB) + L1i L#22 (32KB) + Core L#22
  PU L#44 (P#22)
  PU L#45 (P#46)
  L2 L#23 (256KB) + L1d L#23 (32KB) + L1i L#23 (32KB) + Core L#23
  PU L#46 (P#23)
  PU L#47 (P#47)
[[chatzidp@eu-c7-021-14 ~]$ numactl --hardware
```

- Portable Hardware Locality (hwloc)
 - <https://www.open-mpi.org/projects/hwloc/>
 - available on Euler

Euler: `bsub -W 00:30 -n 24 -ls bash`
Interactive shell on a compute node for 30 minutes

Parallel Initialization - Stream Benchmark

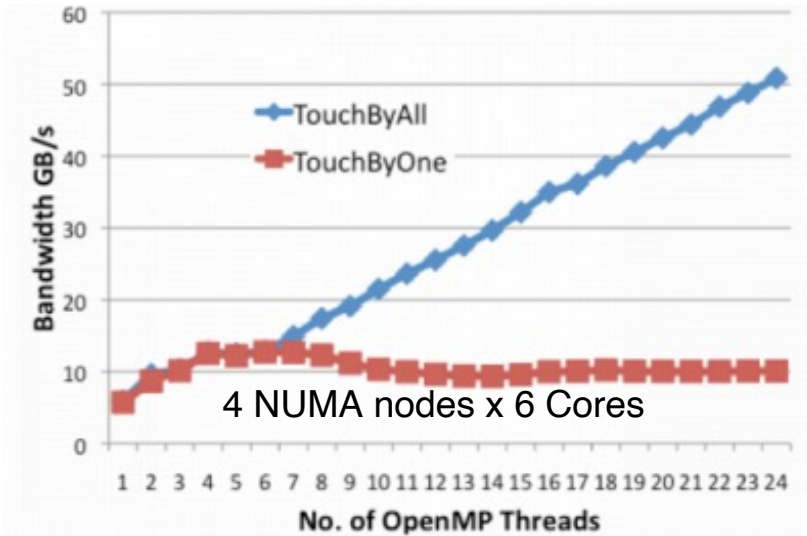
```
#pragma omp parallel for
for(int i=0; i<N; i++) {
    a[i] = 1.0; b[i] = 2.0; c[i] = 0.0;
}
```

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

- Without the first parallel region, the arrays would
 - be initialized only by the master thread
 - located to the NUMA node of the master thread
- Parallel initialization allows the memory of the arrays to be distributed to the NUMA nodes

Stream Benchmark - Results

- TouchByAll: parallel initialization
- TouchByOne: single-threaded initialization



Credit: <http://www.nersc.gov/assets/Uploads/COENERSCTrainingFeb82011.pdf>

WARNING: If your code performs data initialization then you should not study its performance (speedup) by using a for loop that calls `omp_set_num_threads()`

```
for (int t = 0; t < 24; t++) {  
    omp_set_num_threads(t);  
    run_benchmark(); // OpenMP code here  
}
```

Wait Policy: Active or Passive?

- `OMP_WAIT_POLICY`: “provides a hint to an OpenMP implementation about the desired behavior of waiting threads”
 - Possible values: `ACTIVE`, `PASSIVE`
- `ACTIVE`: waiting threads should mostly be active, consuming processor cycles, while waiting.
 - e.g., waiting threads spin
- `PASSIVE`: waiting threads should mostly be passive, not consuming processor cycles, while waiting.
 - e.g., waiting threads yield the processor to other threads or go to sleep

Spin or Sleep?

```
int pthread_mutex_trylock(pthread_mutex_t * mutex);
```

- Allows a thread to try to lock a mutex
- If the mutex is available then the thread locks the mutex
- If the mutex is locked then the function informs the user by returning a special value (EBUSY):

```
while (pthread_mutex_trylock(&mut) == EBUSY)  
    action;
```
- Possible options for **action**
 - **nothing** = continuous check = the thread spins on the core
 - **sched_yield()** = the thread releases the core for a very short period -the operating system can schedule another thread
 - **sleep** = the thread releases the core for a longer time period
 - combination of the above, e.g. the thread spins for a while, then sleeps

the same options can be applied for threads waiting at barriers

OpenMP Quiz 3

- Implement an equivalent version of the following code without using parallel sections

```
void XAXIS();
void YAXIS();
void ZAXIS();

void a9()
{
    #pragma omp parallel sections
    {
        #pragma omp section
            XAXIS();
        #pragma omp section
            YAXIS();
        #pragma omp section
            ZAXIS();
    }
}
```

OpenMP Quiz 3 - Solution

```
void XAXIS();
void YAXIS();
void ZAXIS();

void a9()
{
    #pragma omp parallel for
    for (int i = 0; i < 3; i++)
        if (i == 0) XAXIS();
        if (i == 1) YAXIS();
        if (i == 2) YAXIS();
}
}
```

OpenMP Quiz 3 - Solution

```
void XAXIS();
void YAXIS();
void ZAXIS();

void a9()
{
    #pragma omp parallel
    {
        #pragma omp single nowait
        XAXIS();

        #pragma omp single nowait
        YAXIS();

        #pragma omp single nowait
        YAXIS();
    }
}
```


OpenMP Quiz 4

- Identify and fix any issues in the following OpenMP codes

```
1  int A[N], B[N];
2  int auxdot = 0, dot = 0;
3
4  #pragma omp parallel
5  {
6      #pragma omp for
7      for (int i=0 ; i< N; i++ ){
8          auxdot += A[i]*B[i];
9      }
10
11     #pragma omp critical
12     dot += auxdot ;
13 }
```

OpenMP Quiz 4 - Solution

- Identify and fix any issues in the following OpenMP codes

```
1  int A[N], B[N];
2  int auxdot = 0, dot = 0;
3
4  #pragma omp parallel
5  {
6      #pragma omp for
7      for (int i=0 ; i< N; i++ ){
8          auxdot += A[i]*B[i];
9      }
10
11     #pragma omp critical
12     dot += auxdot ;
13 }
```

race condition on auxdot

Simplest solution:
auxdot must be firstprivate

Examples in OpenMP Specs, v3.1

- A.1 simple parallel loop
- A.3 conditional compilation
- A.5 parallel
- A.7 num_threads and omp_set_dynamic
- A.10 nowait
- A.11 collapse
 - ignore ordered
 - ignore lastprivate
- A.12 parallel sections
- A.13 firstprivate + sections
- A.14 single
- A.18 master
- A.19 critical
- A.21 binding of barrier regions
- A.22 atomic
- A.23 Restrictions on atomic
- A.25 Placement of barrier
- A.30 default(none)
 - ignore threadprivate
- A.32 private
- A.36 reduction
- A.39 nested loop
- A.40 restrictions on nesting of regions
- A.41 omp_set_dynamic and omp_set_num_threads
- A.42 omp_get_num_threads
- A.43-45 locks