High Performance Computing for Science and Engineering

OpenMP Part 2

Fabian Wermelinger
Computational Science & Engineering Laboratory
OUTLINE

• Lecture 1
  • Uniform and Non-Uniform Memory Access
  • OpenMP Processor Binding and Performance
  • Amdahl’s Law and Strong Scaling Analysis
UMA and NUMA Platforms

In first lecture about shared memory:

Each processor accesses the same physical memory:

Uniform Memory Access (UMA)

The discussion we shared so far was concerned with UMA platforms.

https://computing.llnl.gov/tutorials/openMP/
A set of processors have direct access to their own physical memory:

**Non-Uniform Memory Access (NUMA)**

Two different sockets with separate memory management units (MMU)

https://computing.llnl.gov/tutorials/openMP/
A set of processors have direct access to their own physical memory:

**Non-Uniform Memory Access (NUMA)**

- UMA typically features few cores (up to 8, e.g. your laptop or desktop at home).
- This allows for hardware implementations with symmetric access to memory (the bus can still service the memory requests).
- UMA is also referred to as symmetric multiprocessors (SMP).

- Supporting large processor counts requires distributed shared memory.
- Otherwise the pressure on the bus to service memory requests becomes too large (avoiding very high latencies).
- Such architectures have multiple sockets with separate memory modules.
- NUMA is also referred to as distributed shared memory (DSM).

Two different sockets with separate memory management units (MMU)

https://computing.llnl.gov/tutorials/openMP/
Both UMA and NUMA are shared memory architectures. They share the same memory space. The physical memory may be distributed.

Distributing the memory increases the bandwidth and reduces the latency to local memory.

Memory references to a remote memory module have a higher latency than local references (how does this impact performance?)

Note the difference: In distributed programming (MPI) it is not possible to access the remote memory without the assistance of software protocols running on both processes (this is exactly what MPI does)

However, also NUMA requires additional effort in software to benefit from the higher bandwidth

Further reading for the interested: https://queue.acm.org/detail.cfm?id=2513149
A common workflow in a code:

```c
1 int main(int argc, char* argv[]) {
2     double* const A = new double[1000];
3     // initialize data
4     for (int i = 0; i < 1000; ++i)
5         A[i] = 0.0;
6     // other code...
7     // perform work in parallel
8     #pragma omp parallel for
9     for (int i = 0; i < 1000; ++i)
10        A[i] = do_work(i, A);
11     delete[] A;
12     return 0;
13 }
```

1.) Memory allocation on the heap
2.) Initialize the data to some value
3.) Distribute the work among available processor cores

Assume we are working on a NUMA platform. In which memory module(s) will the allocation in step 1.) be placed?
NUMA First Touch Policy

- Memory affinity is not decided by the memory allocation but by the initialization!
- **First touch policy**: Memory is mapped to the NUMA domain that first touches it.

Core 1 first touched this data element. It will be placed in the NUMA domain of core 1.

Reads from this memory location issued from a core that is not in this NUMA domain are slower (higher latency).

Credit: C. L. Luengo Hendriks
NUMA First Touch Policy

**NUMA touch with OpenMP:**

```c
int main(int argc, char* argv[]) {
    double* const A = new double[1000];

    // initialize data: NUMA first touch policy
    #pragma omp parallel for
    for (int i = 0; i < 1000; ++i)
        A[i] = 0.0;

    // other code...

    // perform work in parallel
    #pragma omp parallel for
    for (int i = 0; i < 1000; ++i)
        A[i] = do_work(i, A);

    return 0;
}
```

1.) Memory allocation on the **heap**. No memory is touched yet.

2.) Distribute the data and perform the first touch with the corresponding core.

3.) Perform the computations. Ensure memory references hit in the correct NUMA node.

**Note:** Depending on how A is accessed inside `do_work` you might access memory outside the NUMA domain. In such case, a different data representation may be more beneficial (e.g. reorder data in a block-structured layout)
NUMA First Touch Policy

**NUMA touch with OpenMP:**

```cpp
#include <vector>
using namespace std;

int main(int argc, char* argv[]) {
    // vector<double> A(1000, 0.0); // violates first touch
    vector<double> A(1000); // depends on what default constructor does

    // might not be first touch!
    #pragma omp parallel for
    for (int i = 0; i < 1000; ++i)
        A[i] = 0.0;

    // other code...

    // perform work in parallel
    #pragma omp parallel for
    for (int i = 0; i < 1000; ++i)
        A[i] = do_work(i, A);

    return 0;
}
```

Careful with container types. The constructor may (without knowing what your intention is) touch the data.
The Euler nodes you have access to (Intel Xeon E5 2680v3):

<table>
<thead>
<tr>
<th>Node</th>
<th>Stat</th>
<th>r15s</th>
<th>r1m</th>
<th>r15m</th>
<th>Perf</th>
<th>Mem</th>
<th>MemMax</th>
<th>ISA</th>
<th>Model</th>
<th>S</th>
<th>C/S</th>
<th>T/C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 24*eu-c7-045-15</td>
<td>ok</td>
<td>0.1</td>
<td>0.0</td>
<td>0.7</td>
<td>1%</td>
<td>60672.0M</td>
<td>63.8G</td>
<td>X86_64</td>
<td>XeonE5_2680v3</td>
<td>2</td>
<td>12</td>
<td>2</td>
</tr>
</tbody>
</table>

**Query host:** lshosts -l eu-c7-045-15

**Number of cores on node:**

**Node name:**

**Memory free:**

**Max. Memory:**

**CPU model:**

**Number of sockets (2 CPUs, NUMA):**

**Cores per socket:**

**Logical threads per core (hyperthreading):**
NUMA on Euler

The Euler nodes you have access to (Intel Xeon E5 2680v3):

Query host: lshosts -l eu-c7-045-15

<table>
<thead>
<tr>
<th>Node</th>
<th>Stat</th>
<th>r15s</th>
<th>r1m</th>
<th>r15m</th>
<th>Perf</th>
<th>Mem</th>
<th>MemMax</th>
<th>ISA</th>
<th>Model</th>
<th>S</th>
<th>C/S</th>
<th>T/C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>24*eu-c7-045-15</td>
<td>ok</td>
<td>0.1</td>
<td>0.0</td>
<td>0.7</td>
<td>1%</td>
<td>60672.0 M</td>
<td>63.8G</td>
<td>X86_64</td>
<td>XeonE5_2680v3</td>
<td>2</td>
<td>12</td>
</tr>
</tbody>
</table>

Number of cores on node | Node name | Memory free | Max. Memory | CPU model | Number of sockets | Cores per socket

Logical threads per core (hyperthreading)

Get interactive node (for 1 hour): bsub -W 01:00 -n 24 -R fullnode -Is bash

1 me@eu-c7-045-15$ numactl --hardware
2 available: 2 nodes (0-1)
3 node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11
4 node 0 size: 32638 MB
5 node 0 free: 29328 MB
6 node 1 cpus: 12 13 14 15 16 17 18 19 20 21 22 23
7 node 1 size: 32768 MB
8 node 1 free: 30184 MB
9 node distances:
10 node 0 1
11 0: 10 21
12 1: 21 10

Logical cores enabled by hyperthreading

Available memory on NUMA node

Relative latency (cost) for memory references
NUMA on Euler

The Euler nodes you have access to (Intel Xeon E5 2680v3):

Another useful tool: hwloc-1s
https://www.open-mpi.org/projects/hwloc/

NUMA nodes and attached memory

Cache hierarchy

Cores with physical ID and hyperthreads

I/O devices

L1 data and instruction caches
Test NUMA First Touch Policy on Euler

Measurements on Euler using the STREAM benchmark:  [https://www.cs.virginia.edu/stream/](https://www.cs.virginia.edu/stream/)

```c
/* Get initial value for system clock. */
#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++) {
        a[j] = 1.0;
        b[j] = 2.0;
        c[j] = 0.0;
}

/* Get initial value for system clock. */
#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++) {
        a[j] = 2.0E0 * a[j];
        t = 1.0E6 * (mysecond() - t);
}
```

NUMA touch inside STREAM code

Copy loop benchmark inside STREAM code
NUMA on Euler

Roofline for Xeon E5 2680v3 Euler node (2 sockets):

You can only reach the upper ceiling in the memory bound region with a NUMA aware code!
OUTLINE

• Lecture 1
  • Uniform and Non-Uniform Memory Access
  • OpenMP Processor Binding and Performance
  • Amdahl’s Law and Strong Scaling Analysis
OpenMP Processor Binding and Thread Affinity

Where are OpenMP threads mapped to on a NUMA architecture?

- Mapping considerations are important for NUMA architectures
- OpenMP supports processor binding since the 3.0 specification

Easiest way with environment variables

Affinity control:
- `OMP_PROC_BIND='true'`, `false`

Or:
- `OMP_PROC_BIND='master'`, `close'`, `spread'

‘true’: OpenMP threads are bound to their initial place. Thread affinity is enabled.

‘false’: The execution environment may move OpenMP threads to different places. Thread affinity is disabled.
OpenMP Processor Binding and Thread Affinity

You can tell OpenMP your desired processor affinity in your code.

```c
#pragma omp parallel proc_bind(master|close|spread) new-line
  structured block
```

**Note:** You may use other clauses here, of course.

- **master**: This thread affinity policy instructs the execution environment to assign every thread in the team to the *same place* as the master thread.
- **close**: This thread affinity policy instructs the execution environment to assign the threads in the team to places *close to* the place of the parent thread (may not necessarily be the master in nested parallelism).
- **spread**: This thread affinity policy creates a *sparse* distribution of the threads in the available places.
OpenMP Processor Binding and Thread Affinity

You can tell OpenMP where it can map threads with the `OMP_PLACES` environment variable.

**Affinity control:**
- `OMP_PLACES=a list of places`

There are many possibilities! See the specification for all details.

**Example: Euler node**
- `OMP_PLACES=‘threads(48)’`
- `OMP_PLACES=‘cores(24)’`
- `OMP_PLACES=‘sockets(2)’`

**Note:** Number of places with parentheses is optional.

Specify a number of hardware threads for possible places. This includes hyper-threads.

Specify a number of cores for possible places. Each core may have a certain number of hardware threads.

Specify a number of sockets for possible places. Each socket may consist of a certain number of cores.
### Example: Euler node

- **OMP_PLACES=‘sockets(2)’**

<table>
<thead>
<tr>
<th>Socket 0</th>
<th>node 0 cpus:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>29</th>
<th>30</th>
<th>31</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket 1</td>
<td>node 1 cpus:</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
</tbody>
</table>

Threads can map anywhere on the socket.
**Example: Euler node**

- **OMP_PLACES=‘sockets(2)’**
  
  Socket 0
  
  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
  
  Thread 0, core 0
  Thread 1, core 1
  Thread 2, core 2
  ... Thread 12, core 12
  Thread 13, core 13
  Thread 14, core 14
  ...

- **OMP_PLACES=‘cores(24)’**
  
  Similar but more fine grained control

- **OMP_PROC_BIND=‘close’**
  
  Socket 0
  
  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

  Thread 0, core 0
  Thread 1, core 1
  Thread 2, core 2
  ... Thread 12, core 12
  Thread 13, core 13
  Thread 14, core 14
  ...

- **Socket 1**
  
  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
  
  Thread 12, core 12
  Thread 13, core 13
  Thread 14, core 14
  ...
OpenMP Processor Binding and Thread Affinity

Let's test this:

```c
1 int main(int argc, char* argv[])
2 {
3    int nThreads;
4    #pragma omp parallel
5    {
6        #pragma omp master
7            nThreads = omp_get_num_threads();
8    }
9    vector<string> messages(nThreads);
10    #pragma omp parallel
11    {
12        const int tid = omp_get_thread_num();
13        int cpuid, nodeid;
14        tacc_rdtscp(&cpuid, &nodeid);
15        string hostname(1024, '\0');
16        gethostname(&hostname.front(), hostname.size());
17        ostringstream mystream;
18        mystream << "[thread=\"" << tid << "]\tRunning on host \"" << hostname.c_str() << "}\tCPU \"" << cpuid << "}\tNUMA NODE \"" << nodeid;
19        #pragma omp critical
20            messages[tid] = mystream.str();
21    }
22    for (const string& message : messages)
23        cout << message << endl;
24    return 0;
25 }
26
27 // return cpu and numa node id
29 unsigned long tacc_rdtscp(int *core, int *node)
30 {
31    unsigned long a,d,c;
32    __asm__ volatile("rdtscp" : "=a" (a), "=d" (d), "=c" (c));
33    *node = (c & 0xFFF000)>>12;
34    *core = c & 0xFFF;
35    return ((unsigned long)a) | ((unsigned long)d) << 32);
36 }
```

Returns the processor ID and NUMA node (not portable)
Let’s test this:

```c
int main(int argc, char* argv[]) {
  int nThreads;
  #pragma omp parallel
  {
    #pragma omp master
    nThreads = omp_get_num_threads();
  }

  vector<string> messages(nThreads);

  #pragma omp parallel
  {
    const int tid = omp_get_thread_num();
    int cpuid, nodeid;
    tacc_rdtscp(&cpuid, &nodeid);
    string hostname(1024, '\0');
    gethostname(&hostname.front(), hostname.size());

    ostringstream mystream;
    mystream << "[thread=" << tid << "]";
    mystream << "Running on host " << hostname.c_str() << "CPU " << cpuid << "NUMA NODE " << nodeid;

    #pragma omp critical
    messages[tid] = mystream.str();
  }

  for (const string& message : messages)
    cout << message << endl;

  return 0;
}
```

Returns the processor ID and NUMA node (not portable)

```c
unsigned long tacc_rdtscp(int *core, int *node) {
  unsigned long a,d,c;
  __asm__ volatile("rdtscp" : "=a" (a), "=d" (d), "=c" (c));
  *node = (c & 0xFFF000) >> 12;
  *core = c & 0xFFF;
  return ((unsigned long)a) | (((unsigned long)d) << 32);;
}
```

In practice use:

- `OMP_DISPLAY_ENV='true', 'false'`
- `OMP_DISPLAY_AFFINITY='true', 'false'`
OpenMP Processor Binding and Thread Affinity

Let's test this:

No processor affinity:

```
1 export OMP_PROC_BIND='false'
2 export OMP_NUM_THREADS=4
3 [thread=0/4] Running on host eu-c7-061-02 CPU 37 NUMA NODE 1
4 [thread=1/4] Running on host eu-c7-061-02 CPU 25 NUMA NODE 0
5 [thread=2/4] Running on host eu-c7-061-02 CPU 26 NUMA NODE 0
6 [thread=3/4] Running on host eu-c7-061-02 CPU 38 NUMA NODE 1
```

OpenMP will schedule threads according to available resources.
OpenMP Processor Binding and Thread Affinity

Let’s test this:

No processor affinity:

```bash
1 export OMP_PROC_BIND='false'
2 export OMP_NUM_THREADS=4
3 [thread=0/4] Running on host eu-c7-061-02 CPU 37 NUMA NODE 1
4 [thread=1/4] Running on host eu-c7-061-02 CPU 25 NUMA NODE 0
5 [thread=2/4] Running on host eu-c7-061-02 CPU 26 NUMA NODE 0
6 [thread=3/4] Running on host eu-c7-061-02 CPU 38 NUMA NODE 1
```

OpenMP will schedule threads according to available resources

With processor affinity enabled:

```bash
1 export OMP_PROC_BIND='true'
2 export OMP_NUM_THREADS=4
3 [thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
4 [thread=1/4] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0
5 [thread=2/4] Running on host eu-c7-061-02 CPU 2 NUMA NODE 0
6 [thread=3/4] Running on host eu-c7-061-02 CPU 3 NUMA NODE 0
```

With processor affinity enabled, the default binding order is given by increasing physical core ID’s (see numactl and/or hwloc-1s)
OpenMP Processor Binding and Thread Affinity

With `#pragma omp parallel proc_bind(master)`:

```plaintext
1  export OMP_PROC_BIND='true'
2  export OMP_NUM_THREADS=4
3  [thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
4  [thread=1/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
5  [thread=2/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
6  [thread=3/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
```

All threads map to the place partition of the master thread
OpenMP Processor Binding and Thread Affinity

With `#pragma omp parallel proc_bind(master)`:

```plaintext
1  export OMP_PROC_BIND='true'
2  export OMP_NUM_THREADS=4
3  [thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
4  [thread=1/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
5  [thread=2/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
6  [thread=3/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
```

All threads map to the place partition of the master thread.

With `#pragma omp parallel proc_bind(close)`:

```plaintext
1  export OMP_PROC_BIND='true'
2  export OMP_NUM_THREADS=4
3  [thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
4  [thread=1/4] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0
5  [thread=2/4] Running on host eu-c7-061-02 CPU 2 NUMA NODE 0
6  [thread=3/4] Running on host eu-c7-061-02 CPU 3 NUMA NODE 0
```

All threads map to a place partition close to the parent thread (here `master`).
OpenMP Processor Binding and Thread Affinity

With `#pragma omp parallel proc_bind(master)`:

1. `export OMP_PROC_BIND='true'`
2. `export OMP_NUM_THREADS=4`
3. `[thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0`
4. `[thread=1/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0`
5. `[thread=2/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0`
6. `[thread=3/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0`

All threads map to the place partition of the master thread

With `#pragma omp parallel proc_bind(close)`:

1. `export OMP_PROC_BIND='true'`
2. `export OMP_NUM_THREADS=4`
3. `[thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0`
4. `[thread=1/4] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0`
5. `[thread=2/4] Running on host eu-c7-061-02 CPU 2 NUMA NODE 0`
6. `[thread=3/4] Running on host eu-c7-061-02 CPU 3 NUMA NODE 0`

All threads map to a place partition close to the parent thread (here `master`)

With `#pragma omp parallel proc_bind(spread)`:

1. `export OMP_PROC_BIND='true'`
2. `export OMP_NUM_THREADS=4`
3. `[thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0`
4. `[thread=1/4] Running on host eu-c7-061-02 CPU 12 NUMA NODE 1`
5. `[thread=2/4] Running on host eu-c7-061-02 CPU 24 NUMA NODE 0`
6. `[thread=3/4] Running on host eu-c7-061-02 CPU 36 NUMA NODE 1`

Threads are sparsely mapped over the set of all possible places. We have 48 threads total and 4 threads to map, the stride will be 12 in this case.
OpenMP Processor Binding and Thread Affinity

What happens when we request more threads than the hardware can deliver?

Oversubscription

With an arbitrary number of threads, OpenMP continues the affinity pattern at the end. In this example, CPU 0 and 1 are assigned two threads.
What happens when we request more threads than the hardware can deliver?

**Oversubscription**

With an arbitrary number of threads, OpenMP continues the affinity pattern at the end. In this example, CPU 0 and 1 are assigned two threads.

If the number of threads is evenly divisible by the number of available places, OpenMP will distribute the threads evenly with this pattern.
OpenMP Processor Binding and Thread Affinity

Use `OMP_PLACES` for more control if you must work with oversubscription

With `#pragma omp parallel proc_bind(close)`:

1. `export OMP_PROC_BIND='true'`
2. `export OMP_PLACES='cores(24)'`
3. `export OMP_NUM_THREADS=96`
4. `[thread=8/96] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0`
5. `[thread=1/96] Running on host eu-c7-061-02 CPU 24 NUMA NODE 0`
6. `[thread=2/96] Running on host eu-c7-061-02 CPU 24 NUMA NODE 0`
7. `[thread=3/96] Running on host eu-c7-061-02 CPU 24 NUMA NODE 0`
8. `[thread=4/96] Running on host eu-c7-061-02 CPU 24 NUMA NODE 0`
9. `[thread=5/96] Running on host eu-c7-061-02 CPU 24 NUMA NODE 0`
10. `[thread=6/96] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0`
11. `[thread=7/96] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0`
...
92. `[thread=88/96] Running on host eu-c7-061-02 CPU 22 NUMA NODE 1`
93. `[thread=89/96] Running on host eu-c7-061-02 CPU 22 NUMA NODE 1`
94. `[thread=90/96] Running on host eu-c7-061-02 CPU 22 NUMA NODE 1`
95. `[thread=91/96] Running on host eu-c7-061-02 CPU 22 NUMA NODE 1`
96. `[thread=92/96] Running on host eu-c7-061-02 CPU 47 NUMA NODE 1`
97. `[thread=93/96] Running on host eu-c7-061-02 CPU 23 NUMA NODE 1`
98. `[thread=94/96] Running on host eu-c7-061-02 CPU 23 NUMA NODE 1`
99. `[thread=95/96] Running on host eu-c7-061-02 CPU 23 NUMA NODE 1`

If we specify `OMP_PLACES='cores(24)'`, then 4 threads are mapped to each core. Because of `OMP_PLACES='cores(24)'` OpenMP threads can map to any hardware thread.

**Note:** CPU 0 and 24 are on the same core (two hardware threads).
OpenMP Overheads

What is the cost of spawning a parallel region?

• A useful benchmark for measuring overhead can be found here in this link: https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite
• You can measure the overhead of various OpenMP constructs
  ‣ Synchronization (parallel, parallel for, for, barrier, atomic, reduction, …)
  ‣ Loop scheduling (static, dynamic, guided and different chunk sizes)
  ‣ Tasking (not covered in this course)
OpenMP Overheads

Measurements on Euler with gcc:

OMP_PROC_BIND='false'

OMP_PROC_BIND='true'

gcc does benefit slightly from processor binding.
OpenMP Overheads

Measurements on Euler with clang:

OMP_PROC_BIND='false'

OMP_PROC_BIND='true'

clang reductions do slightly improve with processor binding. Overall performance is superior to gcc for this platform.

Roughly constant overhead for large number of threads.
OpenMP Overheads

Measurements on Euler with Intel (icc):

- **OMP_PROC_BIND='false'**
- **OMP_PROC_BIND='true'**

*icc* and *clang* have similar overhead for this platform.
Spawning and joining threads is expensive
- They are system calls to the OS

The OpenMP runtime library spawns threads only once
- When it encounters the first parallel region
- The threads are re-used at the next parallel region

This means that after the end of a parallel region
- Only the master thread continues
- The other threads become idle

```c
int main(int argc, char* argv[])
{
  #pragma omp parallel
  {
    // first parallel region
    // threads are spawned here
  }

  #pragma omp parallel
  {
    // second parallel region
    // threads are re-used here
  }

  return 0;
}
```
OpenMP Performance

How should idle threads spend their time?

• You can specify a hint to OpenMP how threads should wait with the `OMP_WAIT_POLICY` environment variable.
• The variable can not be changed during runtime.
• It can take two possible values (default is implementation defined):
  ‣ `OMP_WAIT_POLICY='active'`: Waiting threads should mostly be active, consuming processor cycles while waiting (busy-wait, spinning).
  ‣ `OMP_WAIT_POLICY='passive'`: Waiting threads should mostly be passive, not consuming processor cycles while waiting (threads yield the processor to other threads or go to sleep).
• Lecture 1
  • Uniform and Non-Uniform Memory Access
  • OpenMP Processor Binding and Performance
  • Amdahl’s Law and Strong Scaling Analysis
Recall Amdahl’s Law from the first lecture:

\[ S_p = \frac{1}{f + \frac{1-f}{p}} \]

I wrote a shared memory code. How well does my code run in parallel?
Recall Amdahl’s Law from the first lecture:

$$S_p = \frac{1}{f + \frac{1-f}{p}}$$

I wrote a shared memory code. How well does my code run in parallel?

In a picture:

- **Serial execution**: $f$
- **Parallel execution (p processors)**: $\frac{1-f}{p}$
- **Serial fraction of the code**: $f$
- **Parallel fraction of the code**: $1-f$

Time
Implicit assumptions in Amdahl’s Law:

- Fixed problem size
  - Makes sense if $p$ is relatively small
  - Often we want to keep the execution time constant and increase the problem size (weak scaling)

- Negligible communication cost
  - The number of processors $p$ should be small

- All-or-None parallelism
  - A more realistic model would be:
    \[
    S_p = \frac{1}{f_1 + \sum_{i=2}^{p} \frac{f_i}{i}} \quad \text{with} \quad \left( \sum_{i=1}^{p} f_i = 1 \right)
    \]

Problems for which those assumptions are reasonable can use this model for performance analysis. Such analysis is called **Strong Scaling**.

**Recall:** Shared memory architectures cannot implement a large number of processors due to limitations on the memory bus as well as related cost issues. Communication cost using shared memory is still relatively low compared to distributed memory models.
Implication of fixed problem size:

Speed of a certain task: \( \frac{W}{t} \) associated work (problem size)

time needed to complete the work

Speed for serial task: \( \frac{w}{t_1} \)

Speed for parallel task: \( \frac{w}{t_p} \)

**Strong scaling speedup:**

\[
S_p = \frac{\frac{w}{t_p}}{\frac{w}{t_1}} = \frac{t_1}{t_p}
\]
**Strong Scaling Analysis**

*Implication of serial fraction $f$:*

![Graph showing the impact of serial fraction on strong scaling](image)

- **0.1% serial fraction!**
- **1% serial fraction!**
- **10% serial fraction!**

With only about 6x faster with 24 threads...

The serial fraction implies a performance upper-bound:

$$\lim_{p \to \infty} S_p = \frac{1}{f}$$

Even with an infinite amount of processors, this is the best we could do. Strong scaling analysis is very sensitive towards the serial fraction. Communication overhead (e.g. synchronization) further degrades performance.
Strong Scaling Analysis

Recall last week’s false sharing problem (approximation of $\pi$):

We would expect a large parallel fraction in that code given the algorithm to approximate $\pi$.

We used strong scaling analysis to test our implementation.

Strong scaling will tell you whether there is an issue in the implementation!

\[
S_p = \frac{t_1}{t_p}
\]
(Speedup is computed by benchmarking execution time)

- Speedup is computed by benchmarking execution time
- We would expect a large parallel fraction in that code given the algorithm to approximate $\pi$
- We used strong scaling analysis to test our implementation
- Strong scaling will tell you whether there is an issue in the implementation!