# MPI and OpenMP Tasks

*Piotr Luszczek*

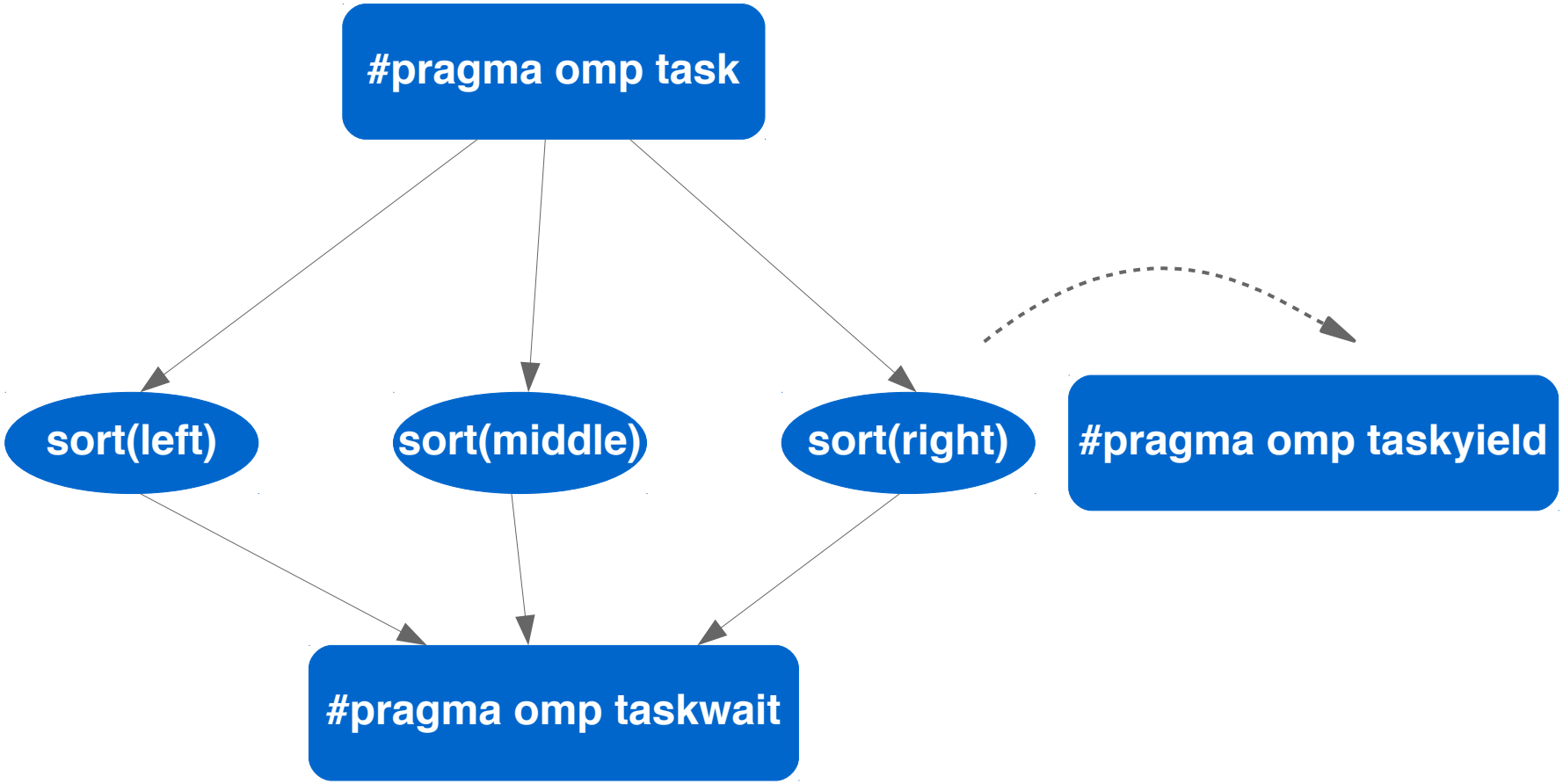
# Tasks in OpenMP

- OpenMP 1 and 2
  - Loop-based parallelism
- OpenMP 3
  - Sibling model
- OpenMP 4
  - Dataflow model
- OpenMP 4.5
  - Priorities

# OpenMP “task” Pragma

- #pragma omp task …
  - if (expr)
  - final (expr)
  - untied
  - mergeable
  - default (shared | firstprivate | none)
  - private (var1, var2, …)
  - firstprivate (var1, var2, …)
  - shared (var1, var2, …)
  - depend (in:var1) (out:var2) (inout:var3)
  - priority (value)

# Scheduling Points



# MPI versus OpenMP

- Pure MPI Pros
  - Scalability
  - Data distribution and locality
  - Communication is explicit
- Pure MPI Cons
  - Steep learning/starting curve
  - High latency
  - Low bandwidth
  - Only large granularity is a good option for performance
  - Difficult load balancing

- Pure OpenMP Pros
  - Incremental parallelism is easy
  - Low latency
  - High bandwidth
  - Implicit communication
  - Coarse and fine grain are OK
  - Dynamic load balancing
- Pure OpenMP Cons
  - Only shared memory
  - Difficult data locality management
  - Getting affinity right is complicated

# MPI Threading Levels

- Request and obtain threading levels
  - MPI_Init_thread(required, &provided)
- Known levels
  - MPI_THREAD_SINGLE
  - MPI_THREAD_FUNNELED
    - Use with #pragma omp master
  - MPI_THREAD_SERIALIZED
    - Use with #pragma omp single
  - MPI_THREAD_MULTIPLE

# SERIALIZED Level of Threading

```
#pragma omp parallel
for (int t=0; t<LARGE; ++t) {
  buf[0] = t; // what is the value stored at buf[0]?

  // barrier ensure that buf[0] has consistent value
  #pragma omp barrier
  // without barrier, buf[0] might change after MPI_Send() started

  #pragma omp single
  MPI_Send(buf, …)

  // when does MPI_Send() starts?
  // when does MPI_Send() executes?
  // when does MPI_Send() finishes?
  // a barrier assures consistent view of buf[] across threads
  // the need for a barrier is more obvious when using MPI_Recv()
  #pragma omp barrier
}
```

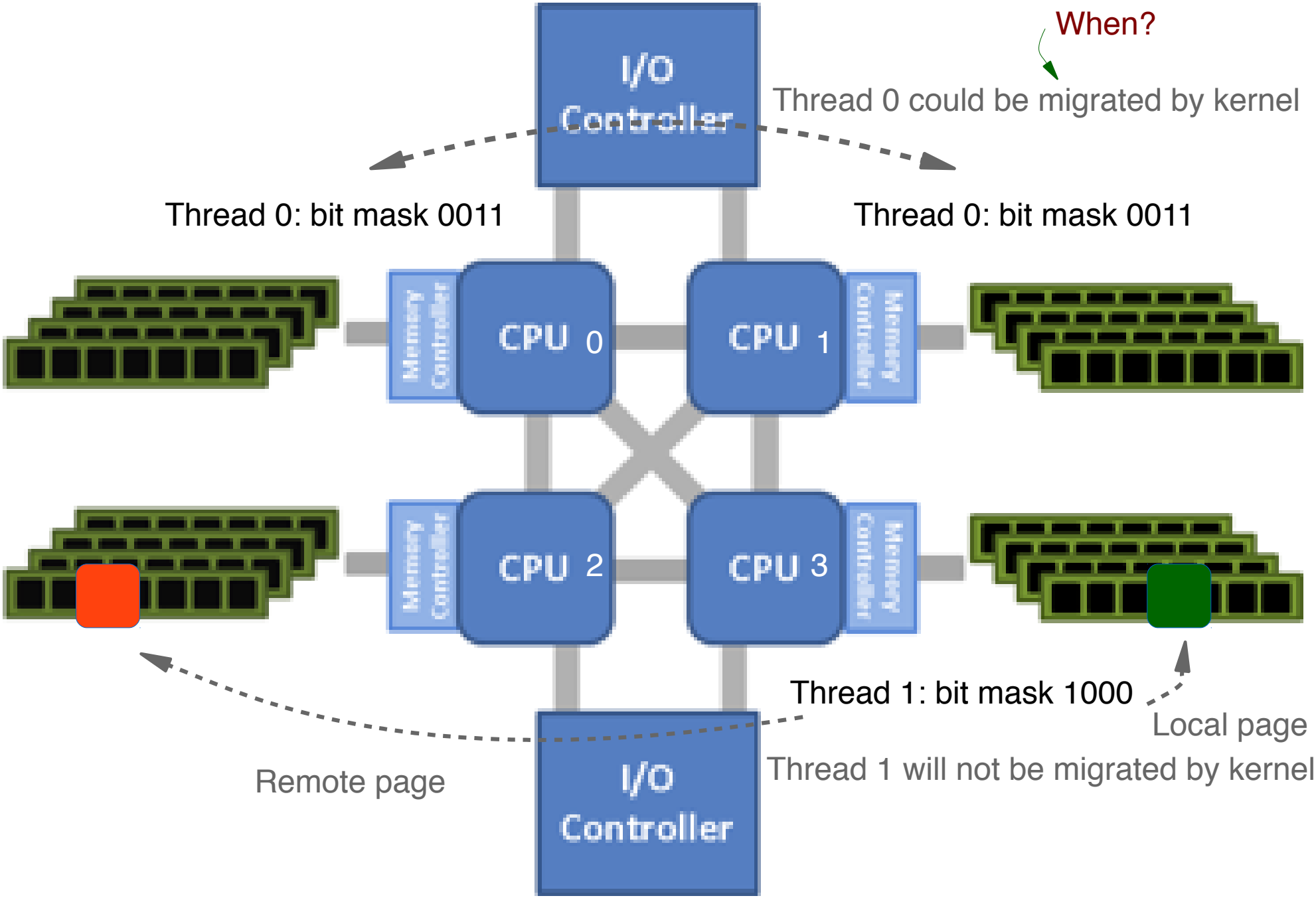
# Problem with Barriers and Ways to Avoid It

- Use of barriers limits performance
  - It also executes memory fences
- It might be possible to remove barriers
  - Algorithm-specific logic might enforce consistent memory state
    - Which thread(s) update the buffer data?
    - Which thread(s) use the buffer data?
- If barriers are removed then communication and computation may be overlapped
  - Other aspect of software and hardware might further limit the overlap

# Thread and Memory Affinity

- In threaded programming, affinity determines:
  - Which thread executes on which core
  - Which memory page ends up in which NUMA island
- On multicore processors, even sequential codes need to worry about affinity
- Thread affinity is a bit mask, each bit, if set, allows to use the corresponding core for executing thread's code
  - If only one of the bits is set then only one core will be used for it
- Memory affinity is complicated
  - Memory affinity is not decided during allocation (by default)
  - Memory affinity is decided on first use: first touch policy
  - Utilities such as numactl may change that
- Each MPI library has it's own affinity

# Affinity in NUMA Multicore Systems



# Enforcing Affinity

```
// first touch policy
// sequential
for (int i=0;i<N;++i) {
  a[i]=1;b[i]=2;c[i]=3;
}

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

```
// first touch policy
#pragma omp parallel for
for (int i=0;i<N;++i) {
  a[i]=1;b[i]=2;c[i]=3;
}

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

# Affinity in Practice

- Command line tools
  - NUMA control: `numactl`
    - May be used to override first-touch policy
  - Hardware stucture: `hwloc`
    - Shows the memory and core hierarchy and allows to choose optimal affinity

# Going Hybrid: Steps

- Starting from scratch...
  - Star with sequential code
  - Add MPI first
  - Add OpenMP next
- Starting with OpenMP code
  - Add barriers to synchronize and make the execution “almost” sequential
  - Add MPI and data decomposition
  - Incrementally remove synchronization
- Starting with MPI code...
  - Add loop parallelism
  - Add sibling tasking
  - Add dataflow tasking

# General Parallelization Guidelines

- Simplest approach
  - Use MPI outside parallel regions
  - Allow only master thread to communicate
    - MPI_THREAD_FUNNELED creates the least parallel and concurrency issues
      - Make sure parallelism is sufficient for your hardware
- If thread-safe MPI is available, use MPI calls inside parallel region
  - Add MPI calls in parallel regions incrementally to keep track of potential bugs
- Be aware of the overhead of MPI_THREAD_MULTIPLE
  - MPI might allow multiple threads to enter but not all parts of the library will run in parallel
  - Thread synchronization inside MPI will cause delays and limit parallelism