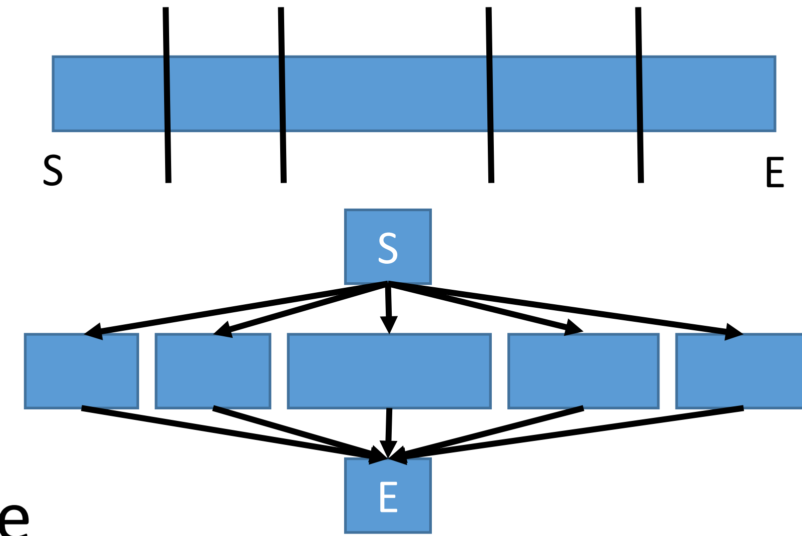
Introduction to OpenMP Tasks

COSC 462

Tasks



- Available starting with OpenMP 3.0
- Remember the Bernstein conditions ? They define dependence relationship between all computational entities (including OpenMP tasks)
  - In this case the dependencies are explicitly declared
- A task is composed of
  - Code to be executed
  - Data environment (inputs to be used and outputs to be generated)
  - A location where the task will be executed (a thread)

Tasks

- The tasks were initially implicit in OpenMP
  - A *parallel* construct constructs implicit tasks, one per thread
  - Teams of threads are created (or declared)
  - Threads in teams are assigned to each task
  - They synchronize with the master thread using a barrier once all tasks are completed
- Allowing the application to explicitly create tasks provide support for different execution models
  - More elastic as now the threads have a choice between multiple existing tasks
  - Require scheduling strategies to be more powerful
  - Move away from the original fork/join model of OpenMP constructs

Technical Notions

- When a thread encounters a task construct, it may choose to execute the task immediately or defer its execution until a later time. If deferred, the task in placed in a conceptual pool of tasks associated with the current parallel region. All team threads will take tasks out of the pool and execute them until the pool is empty. A thread that executes a task might be different from the thread that originally encountered it.
- The code associated with a task construct will be executed only once. A task is **tied** if the code is executed by the same thread from beginning to end. Otherwise, the task is **untied (**the code can be executed by more than one thread).

Types of tasks

- Undeferred: the execution is not deferred with respect to its generating task region, and the generating task region is suspended until execution of the undeferred task is completed (such as the tasks created with the if clause)
- Included: execution is sequentially included in the generating task region (such as a result from a final clause)
- Subtle difference: for undeferred task, the generating task region is suspended until execution of the undeferred task is completed, even if the undeferred task is not executed immediately.
  - The undeferred task may be placed in the conceptual pool and executed at a later time by the encountering thread or by some other thread; in the meantime, the generating task is suspended. Once the execution of the undeferred task is completed, the generating task can resume.

task Construct

#pragma omp task [clause[,clause]*]

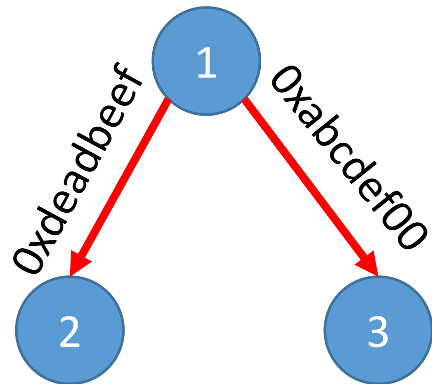
- Clause can be
  - if (expression)
  - final (expression)
  - untied
  - mergeable
  - default (shared | firstprivate | none)
  - private (list)
  - firstprivate (list)
  - shared (list)
  - depend (list)
  - priority (value)

task Construct

#pragma omp task [clause[,clause]*]

- Clause can be
  - depend (list)
  - if (expression)
  - final (expression)
  - untied
  - mergeable
  - default (shared | firstprivate | none)
  - private (list)
  - firstprivate (list)
  - shared (list)
  - priority (value)

- Enforces additional constraints between tasks
- in, out, inout, ~~source, sink~~
- The list if depend contains storage locations (memory addresses) on which the dependency will be tracked



#pragma omp task depend (out 0xdeadbeef, 0xabdcef00)
 task1(…)
#pragma omp task depend (in 0xdeadbeef)
 task2(…)
#pragma omp task depend (in 0xabcdef00)

task Construct

#pragma omp task [clause[,clause]*]

- Clause can be
  - depend (list)
  - if (expression)
  - final (expression)
  - untied
  - mergeable
  - default (shared | firstprivate | none)
  - private (list)
  - firstprivate (list)
  - shared (list)
  - priority (value)

- When the if expression argument evaluate to false
  - The task is executed immediately by the encountering thread
- Allow for user-defined optimizations
  - Example: a model can be used to predict the cost of executing the task and if the cost is too small the cost of deferring the task would jeopardize any possible benefit
  - Allow to define a critical path with respect to cache friendliness and memory affinity

task Construct

#pragma omp task [clause[,clause]*]

- Clause can be
  - depend (list)
  - if (expression)
  - final (expression)
  - untied
  - mergeable
  - default (shared | firstprivate | none)
  - private (list)
  - firstprivate (list)
  - shared (list)
  - priority (value)

- When the final expression evaluates to true the task will not have descendants (leaf in the DAG of tasks) that will be created in the shared pool of tasks
- Allows the runtime to stop generating and deferring new tasks and instead execute all future tasks from the current task directive directly in the context of the execution thread

task Construct

#pragma omp task [clause[,clause]*]

- Clause can be
  - depend (list)
  - if (expression)
  - final (expression)
  - untied
  - mergeable
  - default (shared | firstprivate | none)
  - private (list)
  - firstprivate (list)
  - shared (list)
  - priority (value)

- Different parts of the task can be executed by different threads. Implies the tasks will yield, allowing the executing thread to switch context and execute another task instead.
- If the task is tied, it is guaranteed that the same thread will execute all the parts of the task, even if the task execution has been temporarily suspended
- An untied task generator can be moved from thread to thread allowing the tasks to be generated by different entities.

task Construct

#pragma omp task [clause[,clause]*]

- Clause can be
  - depend (list)
  - if (expression)
  - final (expression)
  - untied
  - mergeable
  - default (shared | firstprivate | none)
  - private (list)
  - firstprivate (list)
  - shared (list)
  - priority (value)

A **merged** task is a task whose data environment is the same as that of its generating task region. When a **mergeable** clause is present on a **task** construct, then the implementation may choose to generate a merged task instead. If a merged task is generated, then the behavior is as though there was no task directive at all

task Construct

#pragma omp task [clause[,clause]*]

- Clause can be
  - depend (list)
  - if (expression)
  - final (expression)
  - untied
  - mergeable
  - default (shared | firstprivate | none)
  - private (list)
  - firstprivate (list)
  - shared (list)
  - priority (value)

- Priority is a hint for the scheduler. A non-negative numerical value, that recommend a task with a high priority to be executed before a task with lower priority
- Default defines the data-sharing attributes of variables that are referenced
  - firstprivate: each construct has a copy of the data item, and it is initialized from the upper construct before the call
  - lastprivate: each construct has a non-initialized copy, and it's value is updated once the task in completed
  - shared: All references to a list item within a task refer to the storage area of the original variable
  - private: each task receive a new item

Scheduling

- OpenMP defines the following task scheduling points:
  - The point of encountering a task construct
  - The point of encountering a taskwait construct
  - The point of encountering a taskyield construct
  - The point of encountering an implicit or explicit barrier
  - The completion point of the task
- all explicit tasks generated within a parallel region are guaranteed to be complete on exit from the next implicit or explicit barrier within the parallel region

Fibonacci – task based

```
int fib(int n) {
  int l, r;

  if (n<2) return n;

 #pragma omp task shared(l) firstprivate(n) \
               final(n <= THRESHOLD)
 l = fib(n-1);

 #pragma omp task shared(r) firstprivate(n) \
               final(n <= THRESHOLD)
 r = fib(n-2);

 #pragma omp taskwait
 return l+r;
}
```

```
int main() {
  int n = 30;

  omp_set_dynamic(0);
  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
   #pragma omp single
   printf ("fib(%d) = %d\n", n, fib(n));
  }
}
```

- Why shared and firstprivate ?
- Why taskwait ?

Task generation

```
#pragma omp single {


   for ( i = 0; i < ONEZILLION; i++)
      #pragma omp task
         process(items[i]);
}
```

- Many tasks will be generated
  - At some point, when the list of deferred tasks is too long, the implementation is allowed to stop generating new tasks, and switches every thread in the team on executing already generated tasks
  - If the thread that generated the tasks is executing a long lasting task, we might eventually reach a starvation scenario where the other threads do not have anything else to execute, and there is nobody to generate new tasks

Task generation

```
#pragma omp single {
   #pragma omp task untied
   for ( i = 0; i < ONEZILLION; i++)
      #pragma omp task
         process(items[i]);
}
```

- Many tasks will be generated
  - At some point, when the list of deferred tasks is too long, the implementation is allowed to stop generating new tasks, and switches every thread in the team on executing already generated tasks
  - If the thread that generated the tasks is executing a long lasting task, we might eventually reach a starvation scenario where the other threads do not have anything else to execute, and there is nobody to generate new tasks
- If the generator task is untied, any other thread in the team can pick it up, and start generating new tasks

Adding MPI to OpenMP

Hybrid programming: MPI + X

MPI vs. OpenMP

- Pure MPI Pro:
  - Portable to distributed and shared memory machines
  - Scales beyond one node
  - No data placement problem
  - Explicit communication
- Pure MPI Con:
  - Difficult to develop and debug
  - High latency, low bandwidth (max PCI-x bus)
  - Large granularity
  - Difficult load balancing

- Pure OpenMP Pro:
  - Easy to implement parallelism
  - Low latency, high bandwidth (max memory bus)
  - Implicit Communication
  - Coarse and fine granularity
  - Dynamic load balancing
- Pure OpenMP Con:
  - Difficult to develop and debug
  - Only on shared memory machines
  - Scale within one node
  - Possible data placement problem (on NUMA architectures)
  - No specific thread order

Why hybrid programming ?

- Hybrid MPI+X paradigm is the software trend for dealing with complexities of hybrid hierarchical architectures (such as heterogeneous multi-core architectures prevalent nowadays).
- Elegant in concept and architecture: using MPI across nodes and OpenMP within nodes. Good usage of shared memory system resource (memory, latency, and bandwidth).
- Avoids the extra communication overhead with MPI within node. Reduce memory footprint.
- OpenMP adds fine granularity (larger message sizes) and allows increased and/or dynamic load balancing.
- Some problems have two-level parallelism naturally.
- Some problems could only use restricted number of MPI tasks.
- Possible better scalability than both pure MPI and pure OpenMP.

Example 1

```
int main(int argc, char* argv[]) {
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  #pragma omp parallel private(omp_rank)
  {
    omp_rank = omp_get_thread_num();
    printf(”Rank %d thread %d\n”, rank, omp_rank);
  }
  MPI_Finalize();
}
```

- What is the expected outcome ?

Example 1

```
int main(int argc, char* argv[]) {
  MPI_Init(NULL, NULL);
  #pragma omp parallel private(omp_rank)
  {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    omp_rank = omp_get_thread_num();
    printf(”Rank %d thread %d\n”, rank, omp_rank);
  }
  MPI_Finalize();
}
```

- What is the expected outcome ?

Initializing MPI with thread support

- MPI_INIT_THREAD (required, &provided, ierr)
  - IN: required, desired level of thread support (integer).
  - OUT: provided, provided level of thread support (integer).
  - Beware: Returned provided maybe less than required.
- Thread support levels:
  - MPI_THREAD_SINGLE: Only one thread will execute.
  - MPI_THREAD_FUNNELED: Process may be multi-threaded, but only master thread will make MPI calls (all MPI calls are ’’funneled'' to master thread)
  - MPI_THREAD_SERIALIZED: Process may be multi-threaded, multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are ’’serialized'').
  - MPI_THREAD_MULTIPLE: Multiple threads may call MPI, with no restrictions.

MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE

OMP MASTER calls MPI

- The OMP master thread is the thread that entered main
  - In some OSes it might have specific properties and behaviors (signals, pid, …)
- MPI_THREAD_FUNNELED is required
- Inside a parallel region there are no implicit synchronizations

```
#pragma omp parallel
for(i = 0; i < BIG_NUMBER; i++)
  buf[i] = i;

  #pragma omp master
    MPI_Send(buf, …);
```

OMP MASTER calls MPI

- The OMP master thread is the thread that entered main
  - In some OSes it might have specific properties and behaviors (signals, pid, …)
- MPI_THREAD_FUNNELED is required
- Inside a parallel region there are no implicit synchronizations
  - An explicit barrier before the MPI call is needed to ensure correctness of the input data
  - An explicit barrier after the MPI call is needed to ensure correctness of the output data
  - It also implies that all the other threads are wasting time

```
#pragma omp parallel
for(i = 0; i < BIG_NUMBER; i++)
  buf[i] = i;

  #pragma omp master
    MPI_Send(buf, …);
```

OMP SINGLE calls MPI

- The OMP single directive ensure the only one thread executes the corresponding block
- MPI_THREAD_SERIALIZED is required
- Inside a parallel region there are no implicit synchronizations
  - An explicit barrier before the MPI call is needed to ensure correctness of the input data
  - An explicit barrier after the MPI call is needed to ensure correctness of the output data
  - It also implies that all the other threads are wasting time

```
#pragma omp parallel
for(i = 0; i < BIG_NUMBER; i++)
  buf[i] = i;

  #pragma omp master
    MPI_Send(buf, …);
```

OMP SINGLE calls MPI

- The OMP single directive ensure the only one thread executes the corresponding block
- MPI_THREAD_SERIALIZED is required
- Inside a parallel region there are no implicit synchronizations
  - An explicit barrier before the MPI call is needed to ensure correctness of the input data
  - An explicit barrier after the MPI call is needed to ensure correctness of the output data
  - It also implies that all the other threads are wasting time

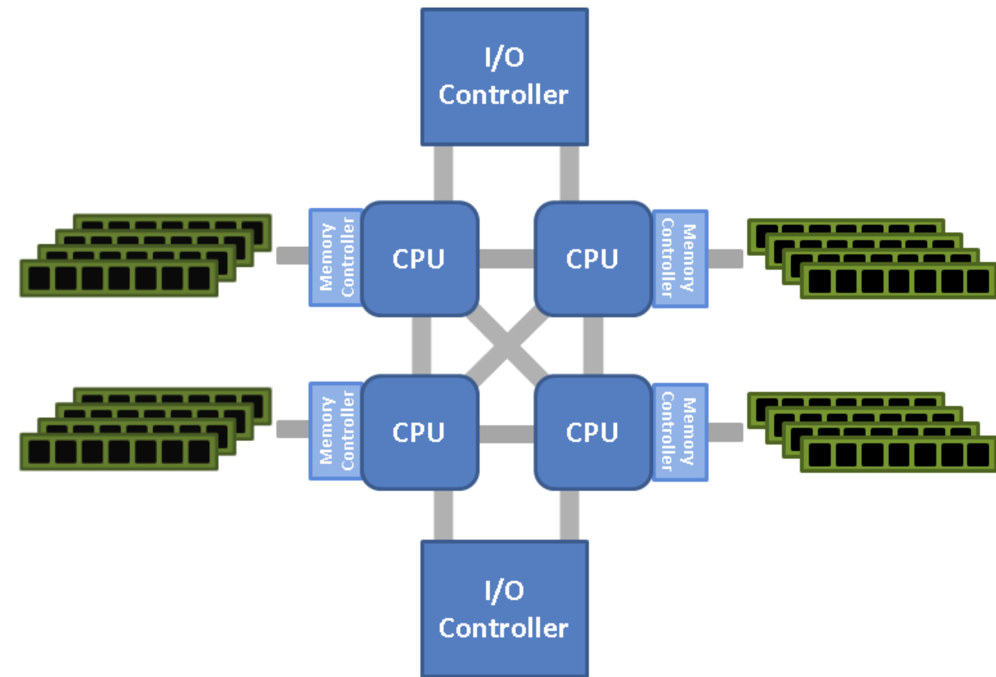
```
#pragma omp parallel
{
   for(i = 0; i < BIG_NUMBER; i++)
      buf[i] = i;
   #pragma omp barrier
   #pragma omp single
      MPI_Send(buf, …);
   #pragma omp barrier
}
```

No pain, no gain

- Enforcing barriers limit the performance
- Removing the barriers depends on the algorithm and on the other implicit synchronizations between parts of the algorithm
  - When was the data updated ? Outside the parallel section ?
  - When will be the data used ? Outside this parallel section ?
- Without the barrier automatic overlap between computations and communications become automatic

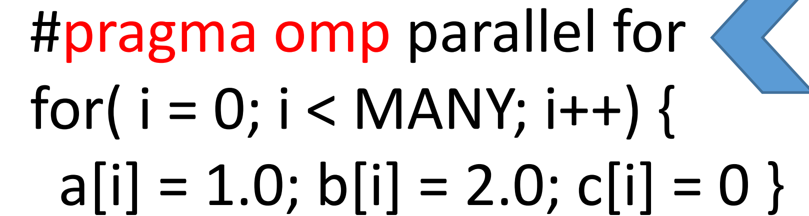
A word (or two) about affinity

- Single threaded MPI applications rarely raise affinity issues
- Unleashing multiple threads in the context of the same application is a different topic:
  - Thread affinity: floating vs. bound
    - Memory issues
  - Memory affinity: allocate memory as close as possible to the core that will use it most
    - Affinity is not decided during the allocation
    - The default policy is ”first touch”
- Each MPI library has it’s own affinity settings (read the man/documentation…)



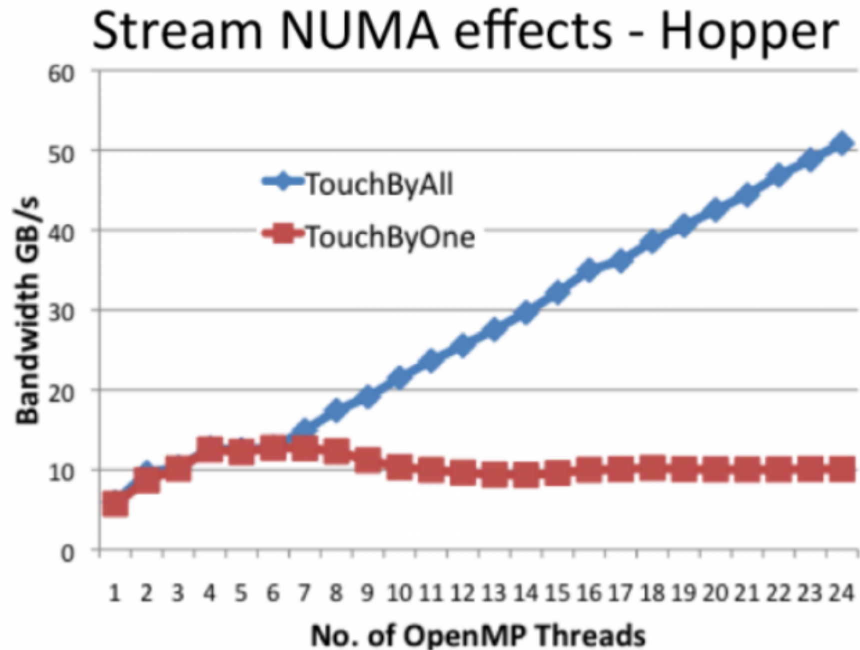
More words about affinity

- Performance with and without correct data initialization
- HWLOC is the tool to use !



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

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



Courtesy Hongzhang Shan

Hybrid Parallelization steps

- From sequential code, decompose with MPI first, then add OpenMP
- From OpenMP code, treat as serial code.
- From MPI code, add OpenMP.
- Simplest and least error-prone way is to use MPI outside parallel region, and allow only master thread to communicate between MPI tasks. MPI_THREAD_FUNNELED is usually the best choice.
    - Keep in mind the cost and implications of serializations
- Could use MPI inside parallel region with thread-safe MPI.
- MPI_THREAD_MULTIPLE comes with a performance cost. Inside the MPI library, thread synchronizations might be necessary, and this might show on the overheads of the MPI calls.
- Special care should be taken regarding collective communications (where clearly only one thread per node should call the collective)
    - Multiple collective calls of the same type in the same communicator is explicitly prohibited by the MPI standard

MPI + MPI (2-level hybridization)

- MPI point-to-point used to exchange data between nodes, MPI-3.0 shared memory regions used inside the node to share content
  - Advantages
    - Lower communication overheads: No message passing inside of the SMP nodes
    - Simplicity: only one parallel programming standard
    - No thread-related data races (e.g., thread-safety isn't an issue)
  - Problems
    - Application responsibility to split communicators into shared memory islands
    - To minimize shared memory communication overhead: the data accessed by the neighbors must be stored in MPI shared memory windows (memory regions visible to other processes where explicit synchronizations are necessary)
    - Load-balancing is as complicated as in pure MPI