OpenMP: tasking

Erik Saule
esaule@uncc.edu

Intro to Parallel Programming

Learning Outcomes

After this lecture you will be able to

- implement an algorithm expressed as a graph of task
- write a simple parallel recursive algorithm

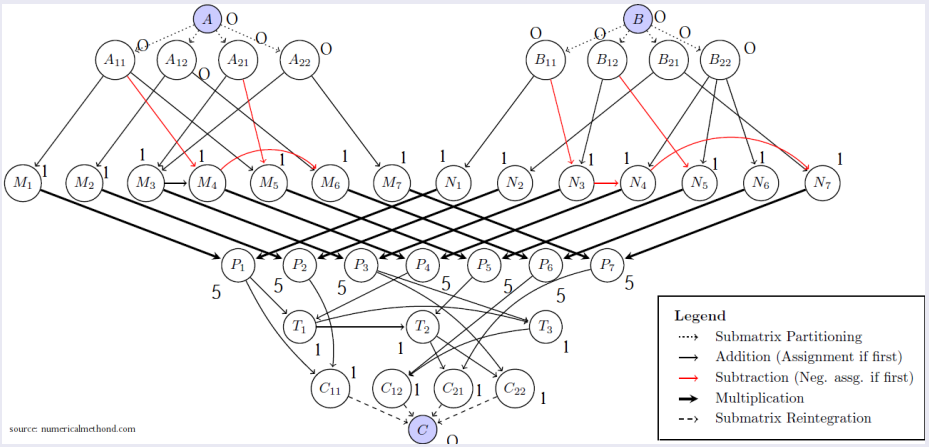
The assignment will make you

- write parallel for loops as recursive algorithms
- write parallel recursive algorithm with tasking construct

Outline

Limits of parallel for

Some dependency structure can not be represented



Recursive algorithms

If you implemented MergeSort, you know how cumbersome it is to implement a recursive algorithm with for loops.
For-loops force a level per level synchronisation.
And they force a significant rewrite of the code to fit a for-loop structure.

Structure of for-based MergeSort.

OpenMP tasks

task

A `#pragma omp task` defines a block that can be done asynchronously with the rest of the code. Note that it does not do anything if not in a parallel section.

taskwait

A task can wait for all DIRECT child tasks by executing `#pragma omp taskwait`. Note a task can complete before its child task if taskwait is not used.

Note that tasking construct where poorly implemented in OpenMP before OpenMP 4.0. In GCC that came in version 4.9.

An example

```
#include <iostream>

void A1()
{std::cout<<"A1"<<std::endl;}
void A2()
{std::cout<<"A2"<<std::endl;}
void B1()
{std::cout<<"B1"<<std::endl;}
void B2()
{std::cout<<"B2"<<std::endl;}

void A() {
#pragma omp task
  A1();
#pragma omp task
  A2();
}

void B() {
#pragma omp task
  B1();
#pragma omp task
  B2();
}
```

```
void maintask() {
#pragma omp task
  A();
#pragma omp task
  B();
#pragma omp taskwait
  std::cout<<"main"<<std::endl;
}

int main() {
#pragma omp parallel
  {
#pragma omp single
    maintask();
  }
  return 0;
}
```

Can produce

mainB2
A2
A1
B1

An example

```
void A() {
#pragma omp task
  A1();
#pragma omp task
  A2();
#pragma omp taskwait
}

void B() {
#pragma omp task
  B1();
#pragma omp task
  B2();
#pragma omp taskwait
}

void maintask() {
#pragma omp task
  A();
#pragma omp task
  B();
#pragma omp taskwait
  std::cout<<"main"<<std::endl;
}
```

main is always last

B2A2
A1
B1
main

Initialization

Main

```
int main() {
#pragma omp parallel
  {
#pragma omp single
    maintask();
  }
  return 0;
}
```

Note only one parallel. And the use of single to execute the tasks once. If single is not used, each thread will execute `maintask`, so many A1 will end up being executed.

Scheduling points and untied task

Model

An OpenMP thread will execute a task until it reaches a scheduling points (such as an `openmp_yield`, the end of a task, the creation of a task, a `barrier`, a `taskwait`, ...).
Upon reaching a scheduling point, it may execute another task or keep executing the current task.

Preemption and Migration

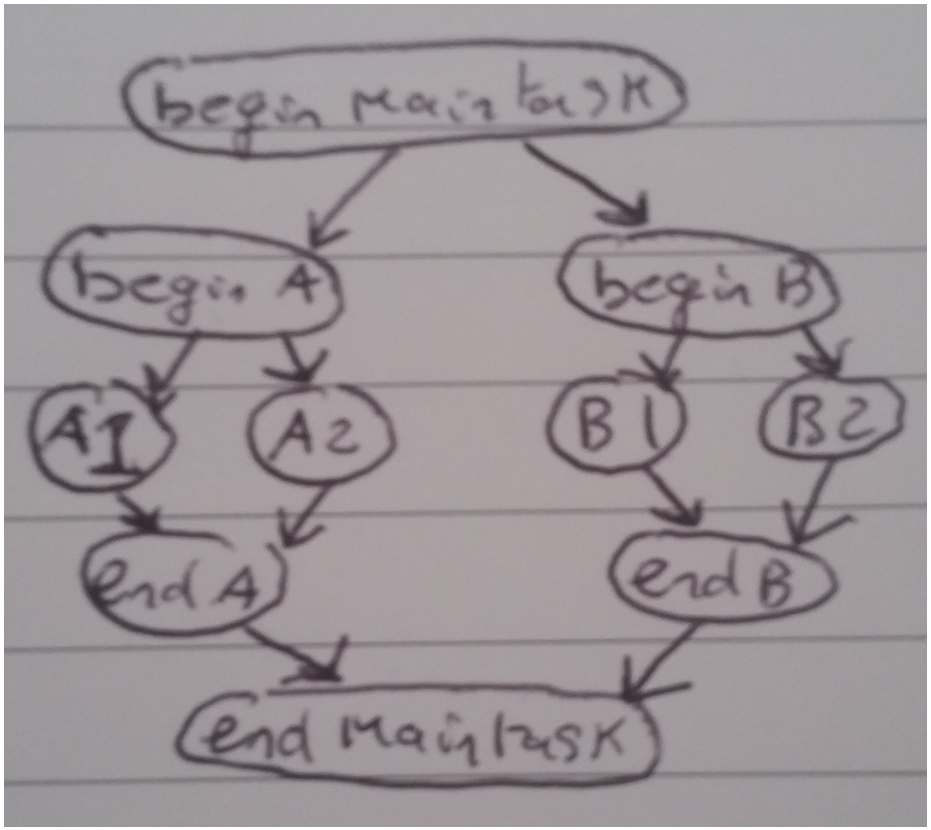
By defaut a task is `tied`, it can be preempted at any scheduling point and then resumed. But it must resumed by the same thread.
A task can be marked `untied` to allow it to be resumed by a different OpenMP thread.
This is important if the code uses thread IDs. (And has performance consequences.)

Equivalent graph

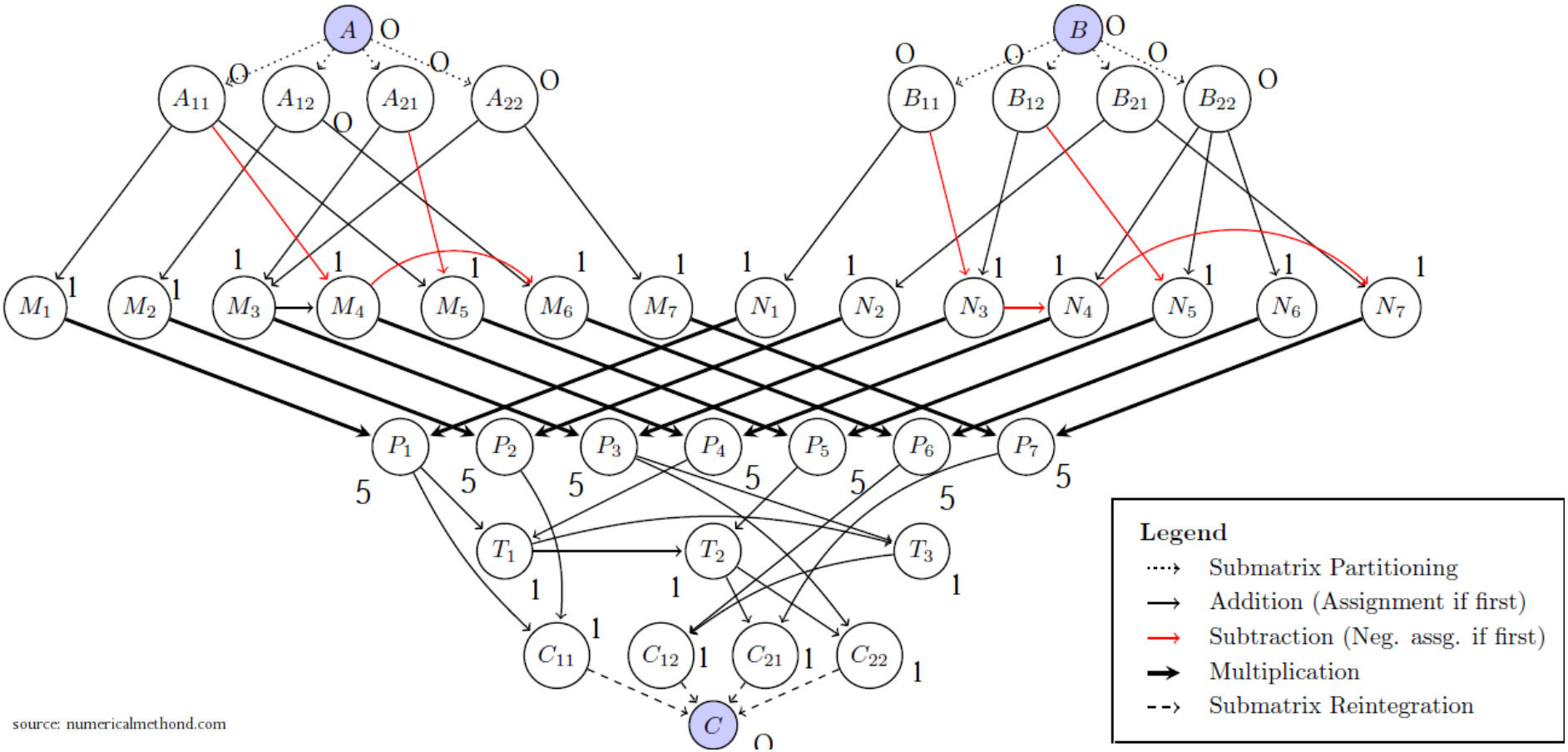
```
void A() {
#pragma omp task
  A1();
#pragma omp task
  A2();
#pragma omp taskwait
}

void B() {
#pragma omp task
  B1();
#pragma omp task
  B2();
#pragma omp taskwait
}

void maintask() {
#pragma omp task
  A();
#pragma omp task
  B();
#pragma omp taskwait
  std::cout<<"main"<<std::endl;
}
```



More expressive, but still not general enough for Strassen



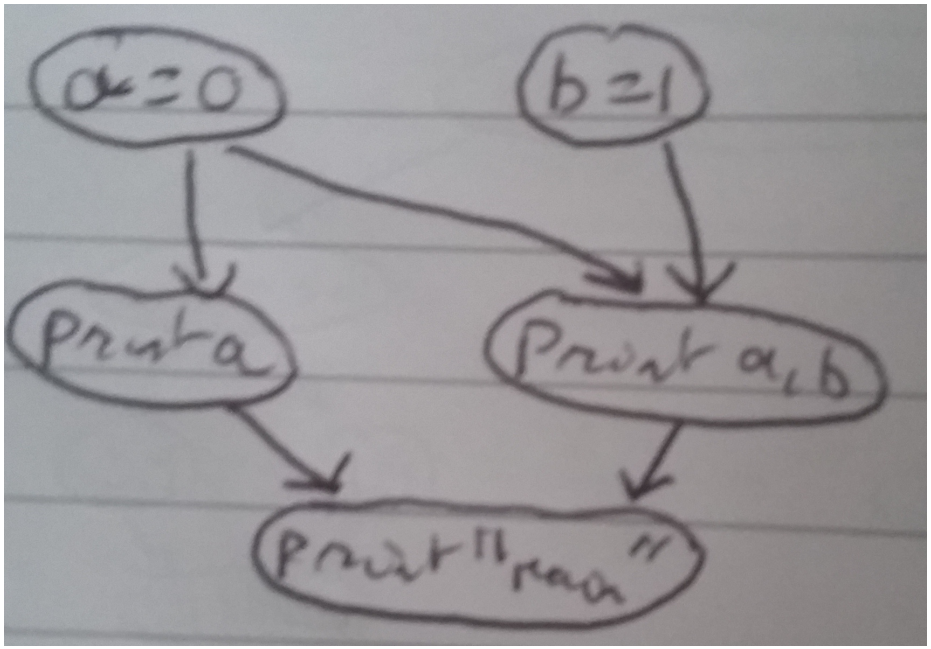
The "Ps" are usually implemented recursively.

But this graph can not be formed with forks and joins. (It is not Series-Parallel).

Arbitrary Graphs

The depend clause

All tasks can take a depend clause to indicate dependencies between tasks. The dependencies can be either `in`, `in`, or `inout`.




```
void maintask() {
  int a, b;
#pragma omp task depend(out:a)
  a=0;
#pragma omp task depend(out:b)
  {b=1; sleep(10);}
#pragma omp task depend(in:a)
  std::cout<<"a="<<a<<std::endl;
#pragma omp task depend(in:a,b)
  std::cout<<"a="<<a<<" b="<<b<<std::endl;
#pragma omp taskwait
  std::cout<<"main"<<std::endl;
}
```

Outline

Producer Consumer

Key Idea

Some threads produce some data.
Other threads consume these data.

Some example

Unix pipes: ls | grep
Webserver: collect clients. pass clients to processing threads.
The graphics pipeline
Video rendering: read file, decompression, filling video buffer

Workstealing

Key Idea

Threads do not take work one piece at a time.
They manage their own work queues.
When idling, take half the work of an other thread.
Support static decomposition (like we did) or decomposition upon workstealing

Some example

Cilk middleware
Algorithm that decompose work when needed (merge sort)

Outline

External

OpenMP:

- OpenMP 4.0 reference card: `http://openmp.org/mp-documents/OpenMP-4.0-C.pdf`
- OpenMP 4.0 API: `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`
- OpenMP Examples: `http://openmp.org/mp-documents/OpenMP_Examples_4.0.1.pdf`
- OpenMP in GCC: `https://gcc.gnu.org/wiki/openmp`

Other task based middleware:

- Intel CilkPlus: `https://www.cilkplus.org/`
- StarPU: `http://starpu.gforge.inria.fr/`
- Intel TBB: `https://software.intel.com/en-us/node/506216`

Workstealing algorithm:

- Blumofe, Robert D.; Leiserson, Charles E. (1999). "Scheduling multithreaded computations by work stealing". JACM. 46 (5): 720748.
- Daouda Traoré, Jean-Louis Roch, Nicolas Maillard, Thierry Gautier, Julien Bernard. Deque-Free Work-Optimal Parallel STL Algorithms. Euro-Par 2008: 887-897

Producer Consumer:

- Wikipedia `https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem`
- In TBB `https://software.intel.com/en-us/node/506217`