

Parallel design patterns

ARCHER course

Practical four: Divide and conquer using
fork/join



Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

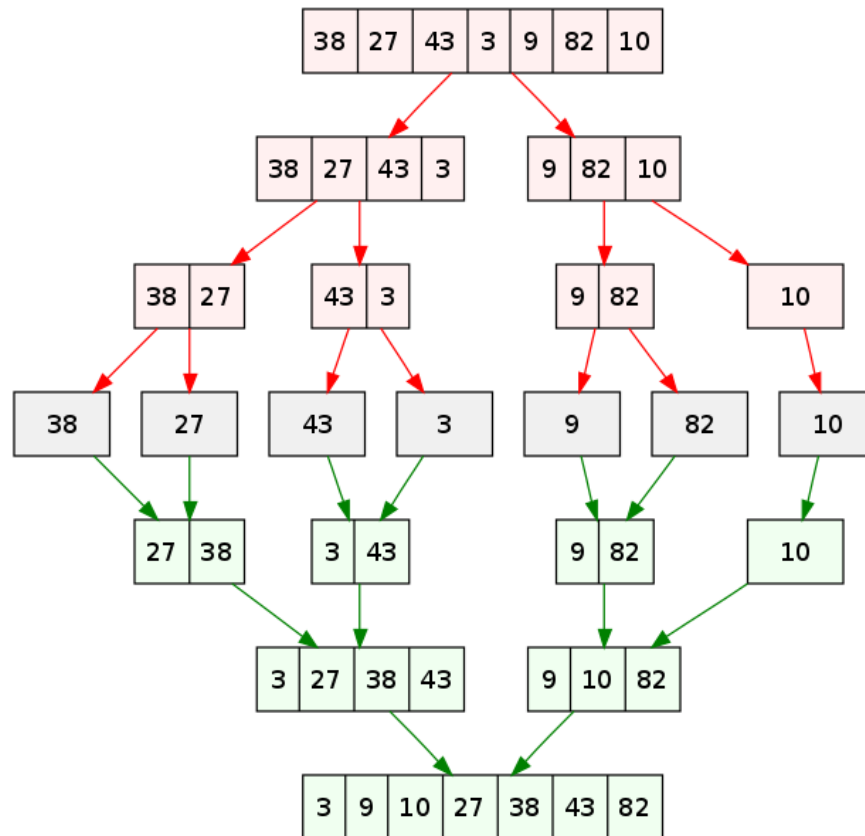
This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Acknowledge EPCC as follows: “© EPCC, The University of Edinburgh, www.epcc.ed.ac.uk”

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

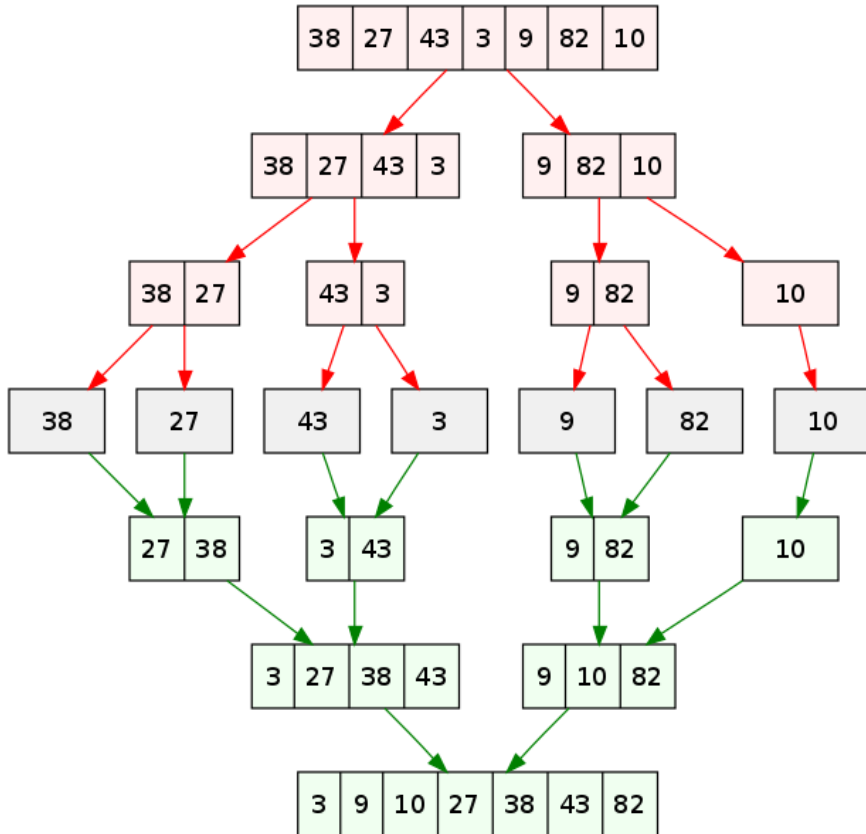
Mergesort

- Starting from some randomly generated, unsorted data.



- Repeatedly divide the data (problem) up until it is trivial to solve
- Then merge the small answers together to form the overall sorted list of numbers
- Maps very well to D&C pattern

Fork/join based mergesort

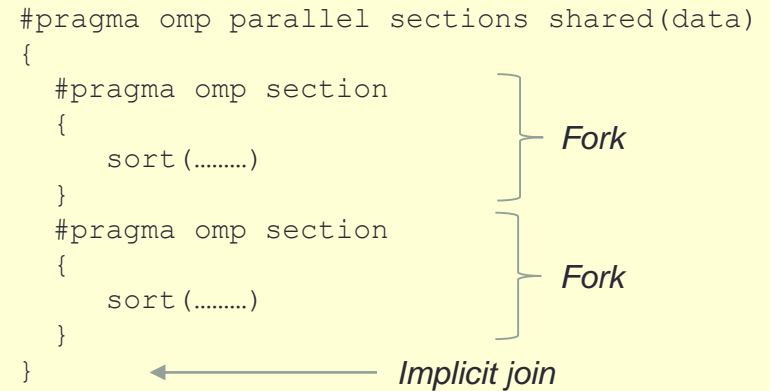


- Each division is a task, working down to some serial threshold (in the image this is 1, but in reality you probably want it to be higher than this.)
- At each division you can fork a new thread and the merge then involves a join
- Instead of threads we can also use OpenMP tasks

Wash up

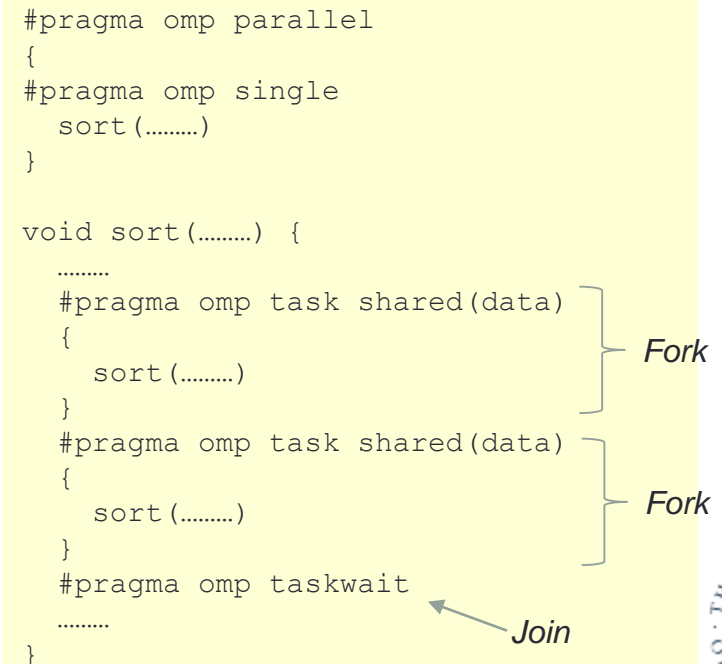
- Sample solutions are available
- Two main versions if you completed the entire exercise
 - Using sections (fork join via threads)
 - Using tasks (are scheduled and will run when a thread is available)
- Crucially with sections we are forced to create a parallel region per level, this is not so with tasks

```
#pragma omp parallel sections shared(data)
{
  #pragma omp section
  {
    sort(.....)
  }
  #pragma omp section
  {
    sort(.....)
  }
}
← Implicit join
```

The diagram shows two nested curly braces on the right side of the code. The top brace is labeled 'Fork' and spans the first #pragma omp section block. The bottom brace is also labeled 'Fork' and spans the second #pragma omp section block. A horizontal arrow labeled 'Implicit join' points from the right towards the closing brace of the outer #pragma omp parallel sections block.

```
#pragma omp parallel
{
  #pragma omp single
  sort(.....)
}

void sort(.....) {
  .....
  #pragma omp task shared(data)
  {
    sort(.....)
  }
  #pragma omp task shared(data)
  {
    sort(.....)
  }
  #pragma omp taskwait
  .....
}
← Join
```

The diagram shows two nested curly braces on the right side of the code. The top brace is labeled 'Fork' and spans the first #pragma omp task block. The bottom brace is also labeled 'Fork' and spans the second #pragma omp task block. A horizontal arrow labeled 'Join' points from the right towards the #pragma omp taskwait line.

With verbose on.....

My id 0 my depth 1 pivot=50
My id 1 my depth 1 pivot=50
My id 0 my depth 2 pivot=25
My id 1 my depth 2 pivot=25
My id 0 my depth 2 pivot=25
My id 1 my depth 2 pivot=25
My id 1 my depth 3 pivot=12
My id 0 my depth 3 pivot=12
My id 1 my depth 3 pivot=12
My id 0 my depth 3 pivot=12
My id 1 my depth 3 pivot=12
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 3 pivot=12
My id 1 my depth 3 pivot=12
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6

Sections



My id 0 my depth 1 pivot=50
My id 6 my depth 1 pivot=50
My id 0 my depth 1 pivot=25
My id 11 my depth 1 pivot=25
My id 0 my depth 1 pivot=12
My id 0 my depth 1 pivot=6
My id 3 my depth 1 pivot=12
My id 0 my depth 1 pivot=6
My id 6 my depth 1 pivot=25
My id 6 my depth 1 pivot=12
My id 3 my depth 1 pivot=6
My id 14 my depth 1 pivot=6
My id 15 my depth 1 pivot=25
My id 11 my depth 1 pivot=12
My id 6 my depth 1 pivot=6
My id 9 my depth 1 pivot=12
My id 0 my depth 1 pivot=6
My id 11 my depth 1 pivot=6
My id 3 my depth 1 pivot=6
My id 6 my depth 1 pivot=12
My id 6 my depth 1 pivot=6
My id 6 my depth 1 pivot=6
My id 15 my depth 1 pivot=12
My id 4 my depth 1 pivot=6
My id 15 my depth 1 pivot=6
My id 8 my depth 1 pivot=12
My id 1 my depth 1 pivot=6
My id 9 my depth 1 pivot=6
My id 8 my depth 1 pivot=6
My id 19 my depth 1 pivot=6

Tasks

My id 0 my depth 1 pivot=50
My id 1 my depth 1 pivot=50
My id 0 my depth 2 pivot=25
My id 1 my depth 2 pivot=25
My id 0 my depth 3 pivot=12
My id 1 my depth 3 pivot=12
My id 0 my depth 2 pivot=25
My id 1 my depth 2 pivot=25
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 3 pivot=12
My id 1 my depth 3 pivot=12
My id 0 my depth 3 pivot=12
My id 1 my depth 3 pivot=12
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6
My id 0 my depth 4 pivot=6
My id 1 my depth 4 pivot=6

Nested tasks



Performance numbers

Code	Runtime (ns)
Serial	6
Sections	82974
Sections (3 threads/region)	3868
Sections (1 section)	1546
Tasks	1882
Tasks (1 task)	1775
Nested tasks	732030
Nested tasks (2 threads/region)	1874

With 100 elements and serial threshold of 10

Code	Runtime (ms)
Serial	0.136
Sections (1 section)	1.977
Tasks	0.0265
Tasks (1 task)	0.152
Nested tasks (2 threads/region)	0.447

With 1000000 elements, serial threshold 10000

Available cores	Runtime (ms)
12	0.0486
13	0.0551

*NUMA region effects with tasks
(1000000 elements, serial threshold 100)*

Conclusions

- Sections are a useful OpenMP construct for fork/join
 - But are limited, especially if you have multiple levels as you can easily over subscribe threads to cores
- OpenMP tasks are more flexible and can avoid this problem
 - This was actually one of the main motivations behind OpenMP tasks
- Be careful of going across NUMA regions
 - Not a huge amount you can do with tasks as they will be mapped to any available thread. Hence I often limit myself to running this sort of code on a NUMA region rather than full node