Concepts and Models of Parallel and Data-centric Programming

Parallel Algorithms V

Lecture, Summer 2020

Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

i12 High Performance Computing | RWTH AACHEN UNIVERSITY

Outline

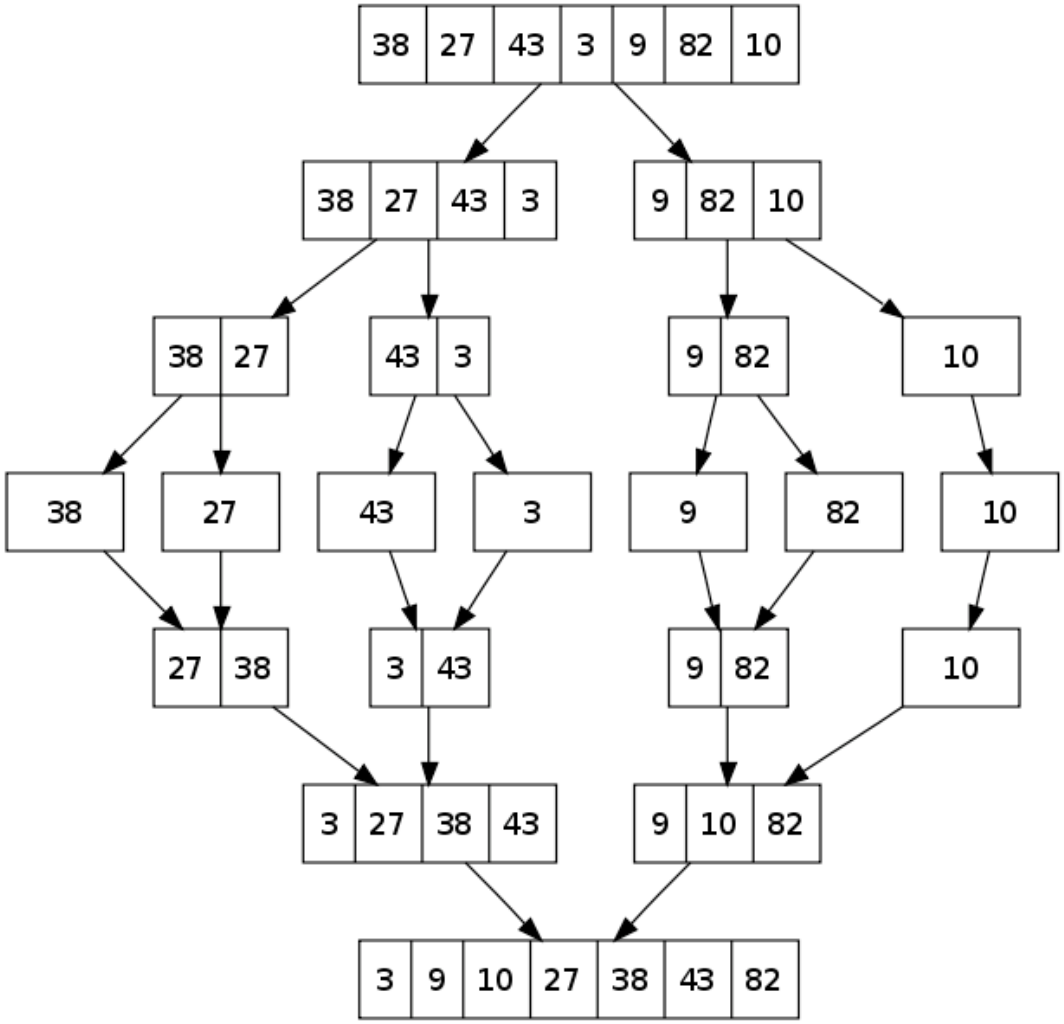
0. Organization
1. Foundations
2. Shared Memory
3. GPU Programming
4. Bulk-Synchronous Parallelism
5. Message Passing
6. Distributed Shared Memory
7. **Parallel Algorithms**
8. Parallel I/O
9. MapReduce
10. Apache Spark

a. Berkeley DWARFS
b. Dense Linear Algebra
c. Sparse Linear Algebra
d. Monte Carlo Methods
e. Graph Traversal

Graph Traversal

Merge-Sort

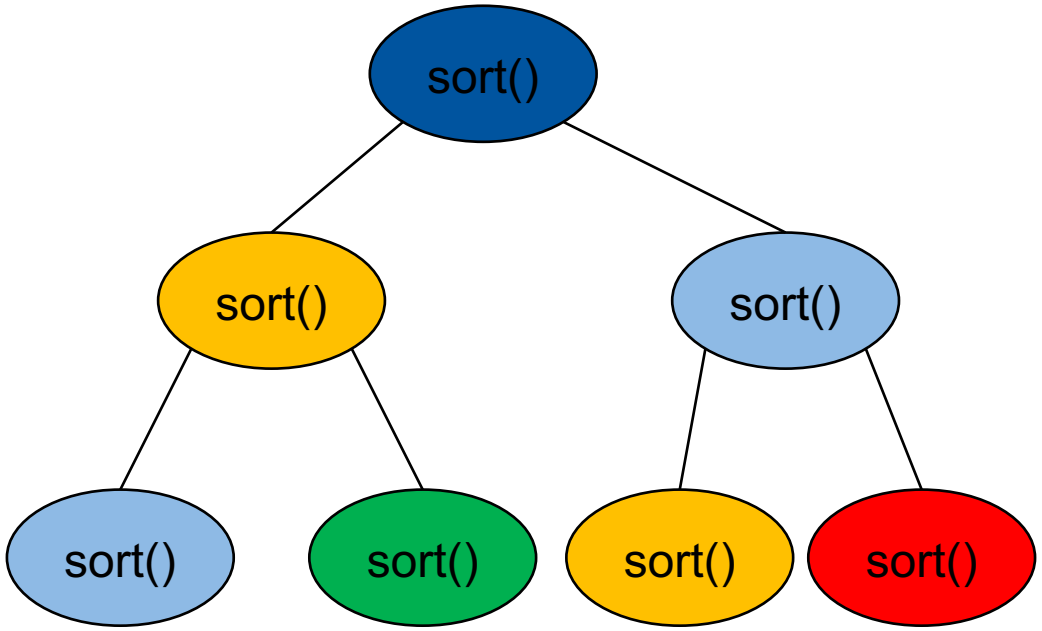
- Algorithm
  - Not in-place (Array tmp)
  - Recursive



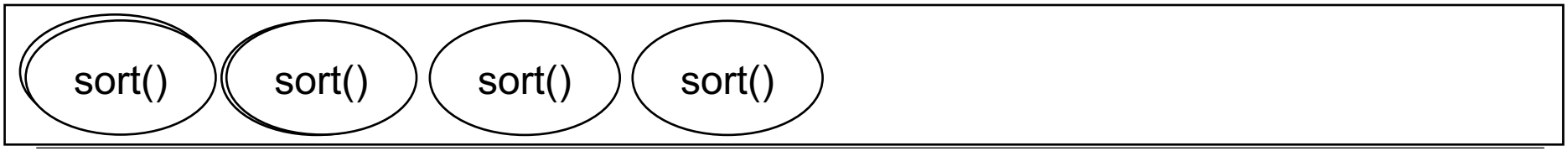
- What is the intuitive approach to parallelize such a problem?

Tasking illustrated

- T1 enters sort()
- T1 creates child tasks
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new child tasks
- T1 - T4 execute tasks



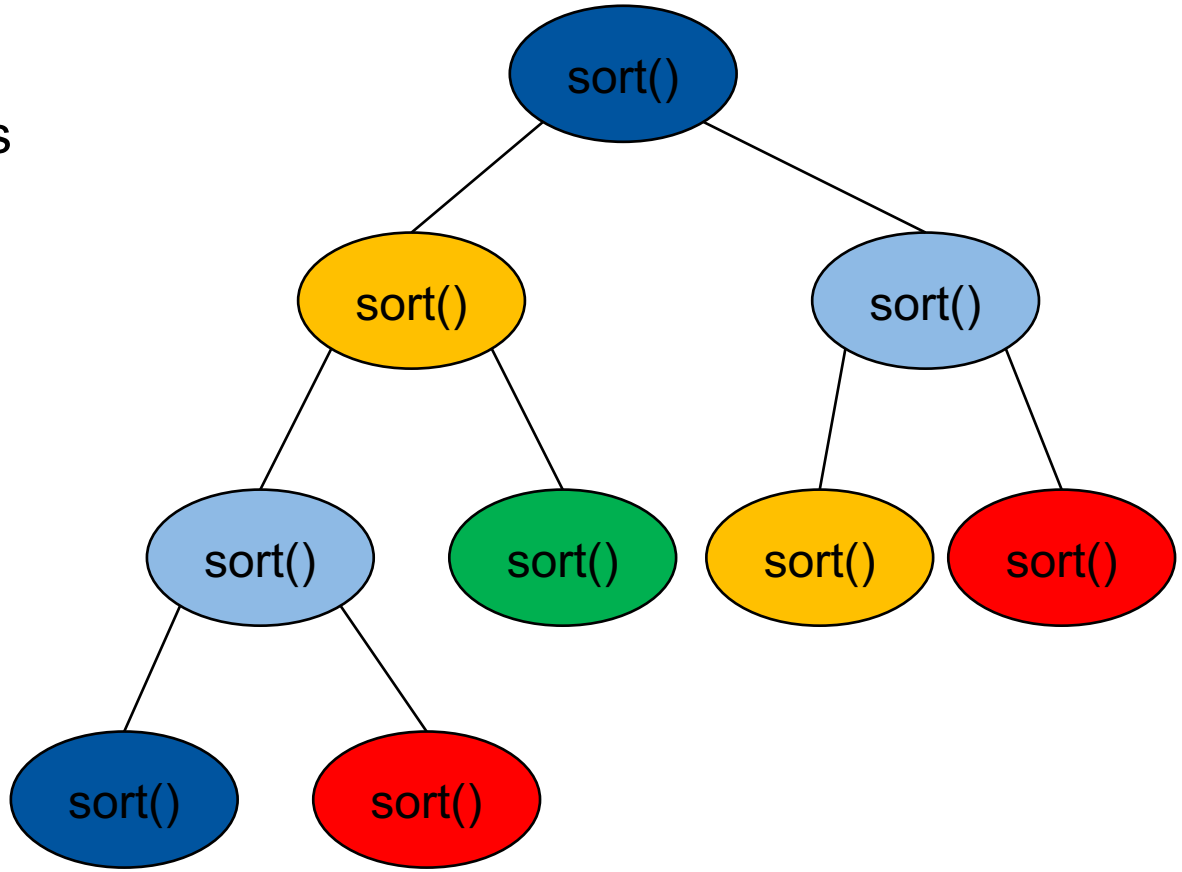
Task Queue



Tasking illustrated

- T1 enters sort()
- T1 creates child tasks
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new child tasks
- T1 - T4 execute tasks

...



Solution in OpenMP / 1

- Task-parallel Partitioning step
  - Cut-off: >= 10 times number of cores tasks created
- Task-parallel Merge step
  - Cut-off similar
- Most profitable optimization: NUMA-aware task scheduling
  - See publication: Approaches for Task Affinity in OpenMP, by C. Terboven, J. Hahnfeld, X. Teruel, S. Mateo, A. Duran, M. Klemm, S. L. Olivier, and B. R. de Supinski, at IWOMP 2016 in Japan, 2016.

Solution in OpenMP / 2

- Task-parallel partitioning

```
void MsParallel(int *array, int *tmp, long begin, long end, int deep) {
   /* cut-off based on deep left out for brevity */
   long half;
   if (begin < (end - 1)) {
      half = (begin + end) / 2;
#pragma omp task default(shared)
      MsParallel(array, tmp, begin, half, deep - 1);
#pragma omp task default(shared)
      MsParallel(array, tmp, half, end, deep - 1);
#pragma omp taskwait
   }
   MsMergeParallel(tmp, array, begin, half, half, end, begin, deep);
}
```

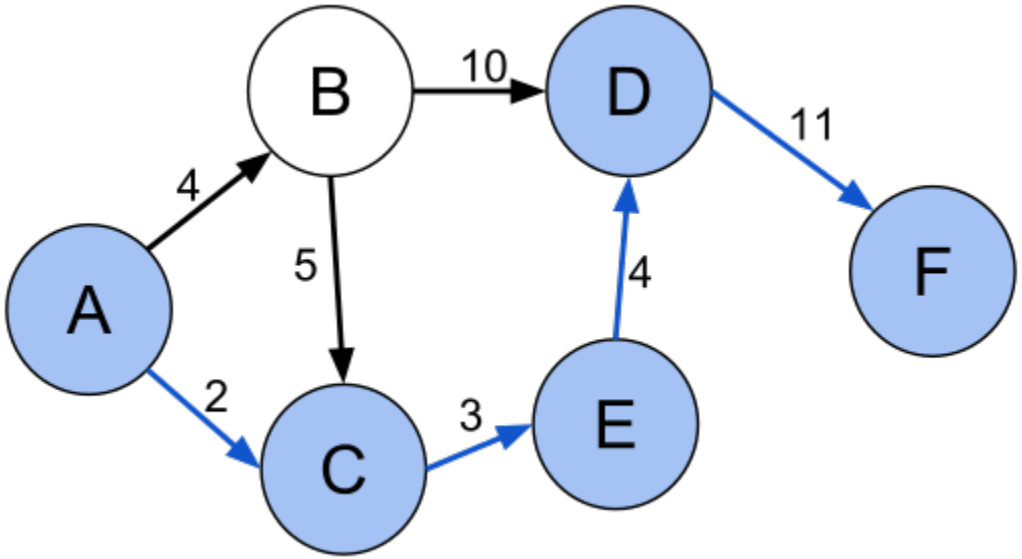
Solution in OpenMP / 2

- Task-parallel merging

```
void MsMergeParallel(int *out, int *in, long begin1, long end1, long begin2, long
end2, long outBegin, int deep)

{

  /* index computation left out for brevity */


#pragma omp task default(shared)

   MsMergeParallel(out, in, begin1, half1, begin2, half2, outBegin, deep-1);

#pragma omp task default(shared)

   MsMergeParallel(out, in, half1, end1, half2, end2, outBegin2, deep-1);

#pragma omp taskwait

}
```

Single-source shortest path

- Shortest path problem: finding a path between two nodes in a graph such that the sum of the weights of its edges is minimized



  - Single-source: shorts path from a given source to all other nodes
  - Shortest path from A to F: (A, C, E, D, F)

- What is the intuitive approach to parallelize such a problem for a large graph?

Simple approach for parallel BFS in DASH / 1

```
// This class represents a simple implementation of a directed
// graph using adjacency list representation

class Graph

{

    int V;             // No. of vertices

    // Pointer to an array containing adjacency lists

    dash::Array<int> *adj;

public:

   Graph(int V)

   {

      this->V = v; adj = new dash::Array<int>(V);

   }

   void BFS(int s);   // performs BFS traversal from source s

};
```

Simple approach for parallel BFS in DASH / 2

```
void Graph::BFS(int s)
{
    // Mark all vertices as not visited
    dash::Array<bool> *visited = new dash:Array<bool>(V);
    for (auto i = 0; i < visited->lsize(); i++)
        visited.local[i] = false;

    // Create a queue for BFS
    std::list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);
```

Simple approach for parallel BFS in DASH / 3

```
while(!queue.empty())
{
    s = queue.front();      // Dequeue a vertex from queue
    /* do something with s */

    // Get all adjacent vertices of the dequeued vertex s. If an adjacent
    // has not been visited, then mark it visited and enqueue it
    for (list<int>::iterator i = adj[s].lbegin(); i != adj[s].lend(); ++i)
    {
        if (!visited[*i])
        {
            visited[*i] = true;
            queue.push_back(*i);
        }
} } }
```

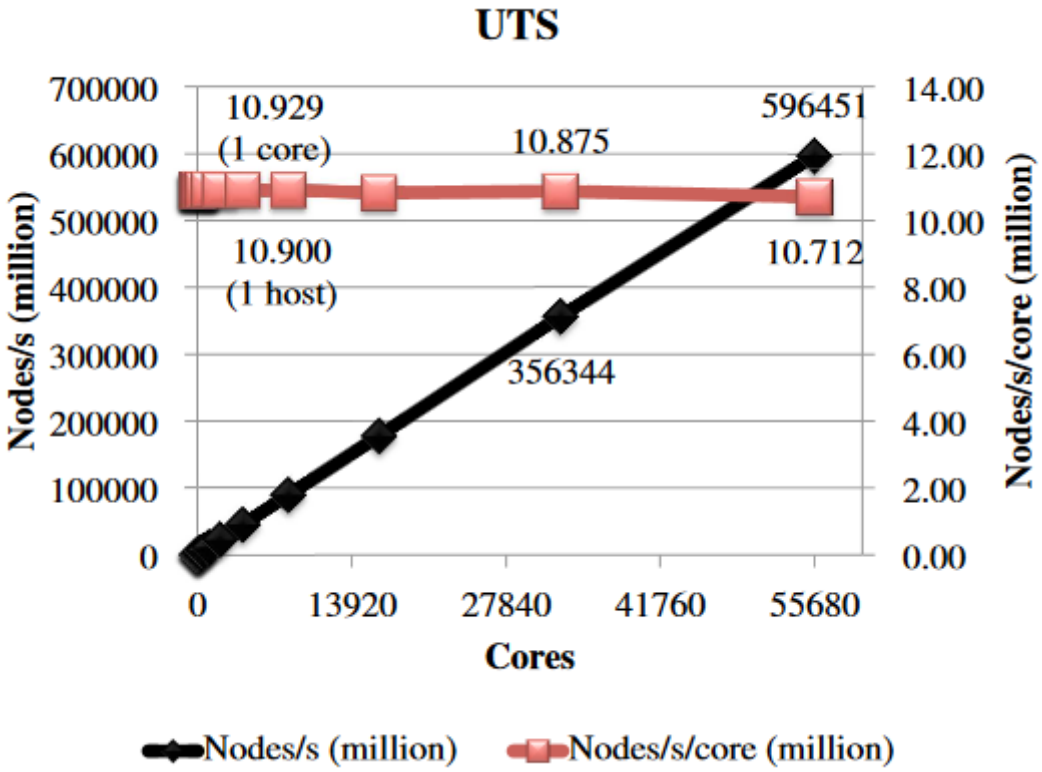
Benchmark results for UTS / 1

- UTS: Unbalanced tree search
  - benchmark that measures the rate of traversal of a tree generated on the fly
  - requires advanced dynamic distributed load balancing techniques

- Work stealing idea
  - Every worker (one per place) maintains a list of pending nodes to process
  - If the list becomes empty, the idle worker attempts to steal nodes from another worker
    - steal attempts are first random and synchronous
  - Past a few unsuccessful random attempts, “steal” from a fixed precomputed list of victims called *lifelines*
    - which will actively share work with any connected quiescent node

Benchmark results for UTS / 2

- Results from 2012 (IBM Power 775 system)



  - UTS benchmark: https://sourceforge.net/p/uts-benchmark/wiki/Home/