Study of parallelization of sorting algorithms

CSC 746, High Performance Computing

Fabian Weiland, Fall 24

Problem Statement

- Focus is on optimizing sorting algorithms, specifically mergeSort and quickSort
- Sequential mergeSort and quickSort have O(n*log(n)) time complexity
- Opportunity for parallelization due to the presence of independent, repetitive tasks
- Benchmarking parallelized quickSort and mergeSort against their serial counterparts
- Using the openMP (Open Multi-Processing) API

Codebase

- **benchmark.cpp** -> driver program which does test runs, creates problem sizes and runs the different algorithms for each problem size and measures the runtime
- **setUpArray.cpp** -> initialises array with random numbers of problem size
- **mergeSort** & **quickSort.cpp** -> sequential version of the sorting algorithms
- **mergeSort_openMP** & **quickSort_openMP.cpp** -> parallelized openMP versions of the sorting algorithms

Methodology

- two different performance metrics were calculated using elapsed runtime
- runtime was measured by adding instrumentation code around the sorting methods (C++ Chrono Timer)

➡ runtime across problem sizes and concurrency level

➡ speed up across concurrency levels

  - speedup(x) = (runtime_at_concurrency_1)/(runtime_at_concurrency_x)

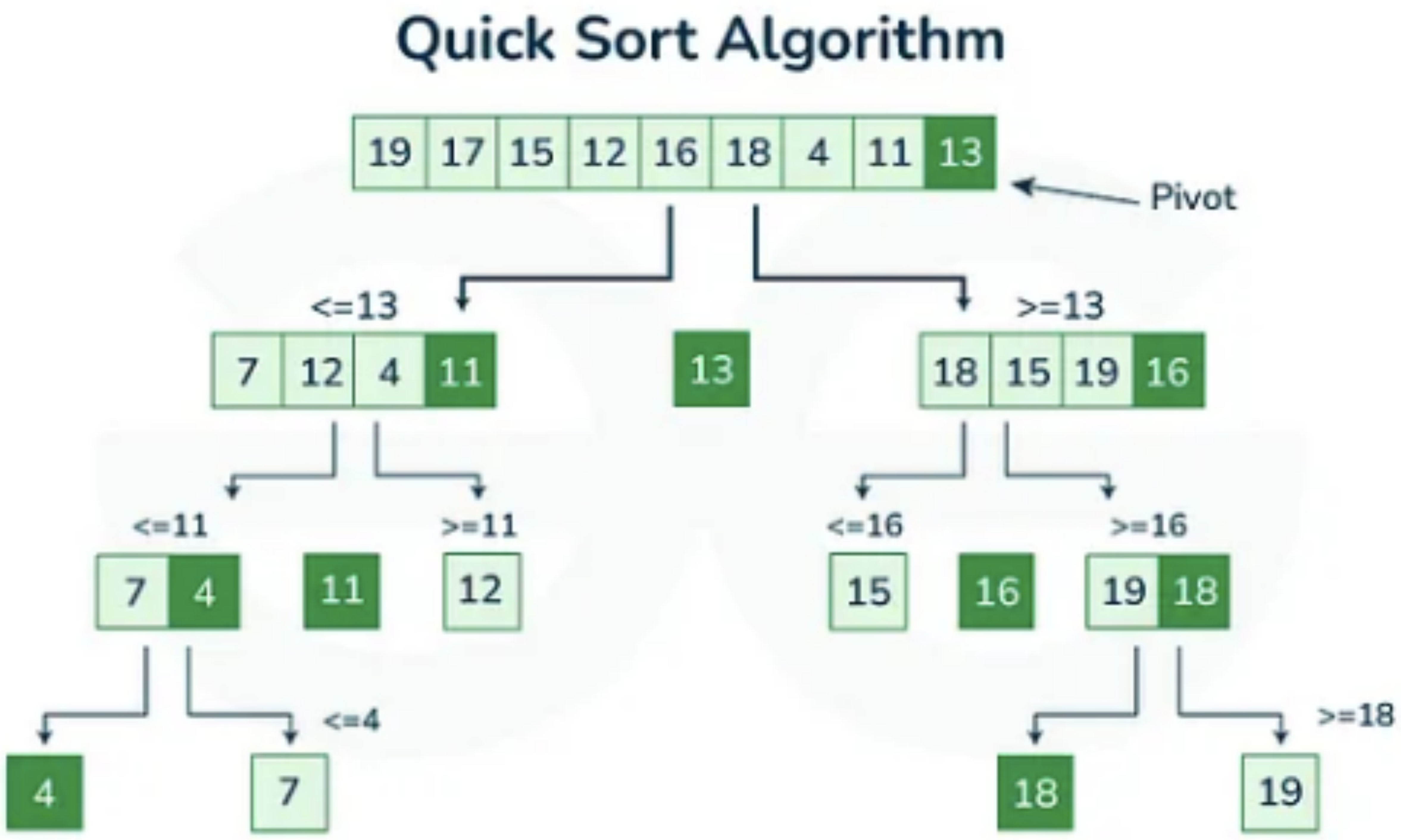
- tests were conducted over 6 different concurrency levels, ranging from 1, 4, 9, 15, 40, 60 with problem sizes ranging from 16, 32, 64, 128, 265 - million

Computational environment and platform

- experiments were conducted on a single CPU node of the Perlmutter supercomputer (NERSC-9)
- a CPU node features 2 AMD EPYC 7763 (Milan) CPU’s, each with a base clock speed of 2.45 GHz and 256 MB of L3 Cache and 512 GB of DDR4 memory
- OS is the Cray Linux Environment
- source code was compiled with C++ (GCC version 11.4.0)

Implementation of quickSort

- select pivot and sort the array recording to this pivot in place




```
// lb = lowerBound
// ub = upperBound
partition(lb, ub, Array){
    pivot := Array[ub]
    i := lb

    for i to ub:
        if Array[i] < pivot:
            swap Array[i] and Array[lb]
            lb++

    swap pivot and Array[lb]
    return lb  //new pivot index
}

QuickSort(lb, ub, Array){
    if lb < ub:
        pivotIndex = partition(lb, ub, Array)

        QuickSort(lb, pivotIndex-1, Array)
        QuickSort(pivotIndex+1, ub, Array)
}
```

openMP task construct

- break up a big task into multiple smaller tasks and execute these tasks across the available threads.

```
#pragma omp task
{
   // this task gets written on a sticky note
   // and added to the list of tasks
}
```

sticky notes

```
#pragma omp single
{
   #pragma omp task
   // this task gets written on a sticky note
   // and added to the list of tasks

   #pragma omp task
   // this task gets written on a sticky note
   // and added to the list of tasks
}
```

- one master thread to write the tasks down to sticky notes
- all threads execute the sticky notes

Team of threads

quickSort with openMP task construct

parallelized version

- breaking down recursive calls into independent tasks
- those independent tasks can run in parallel
- openMP handles dependencies (sub arrays depending on partitioning)
- small array sizes are sorted sequential to avoid overhead
- shared (Array) = array is shared among all tasks
- threashold (taskLimit) is 300

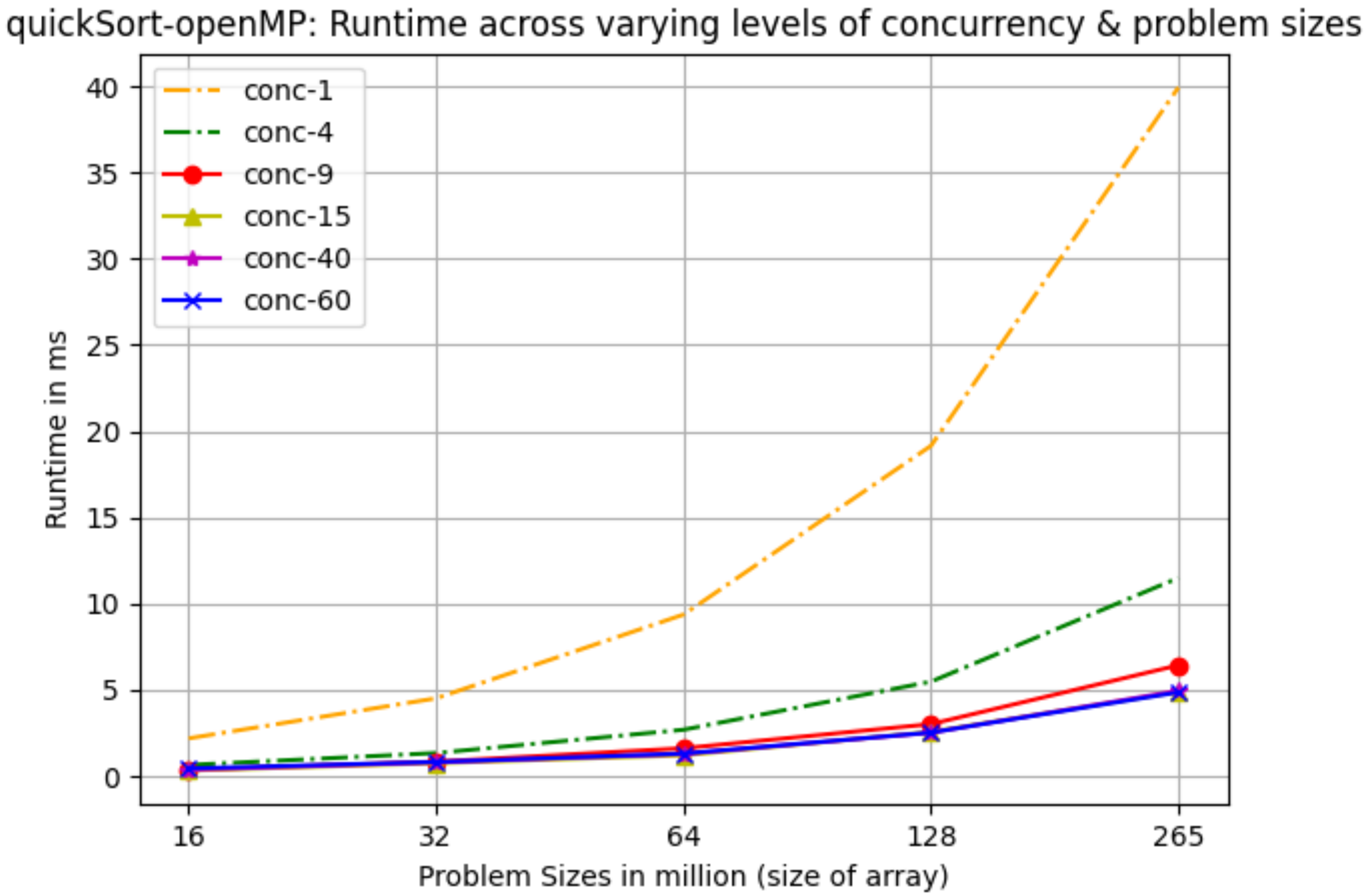
```
void quickSort(int64_t lb, int64_t ub, uint64_t* Array, int64_t taskLimit){

    if(lb < ub)
    {
        if ((ub - lb) < taskLimit ){
            return quickSort_sequential(lb, ub, Array);

        }else{
                int64_t pivot_index = partition_openMP(lb, ub, Array);

                #pragma omp task shared(Array)
                    quickSort(lb, pivot_index - 1, Array, taskLimit);
                #pragma omp task shared(Array)
                    quickSort(pivot_index + 1, ub, Array, taskLimit);
        }
    }
}
```

```
void quickSort_openMP(int64_t lb, int64_t ub, uint64_t* Array){
    #pragma omp parallel
        {
            #pragma omp single
            quicksort(lb, ub, Array, 300);
            #pragma omp taskwait
        }
}
```

Result: quickSort

Runtime across problem sizes and concurrency level

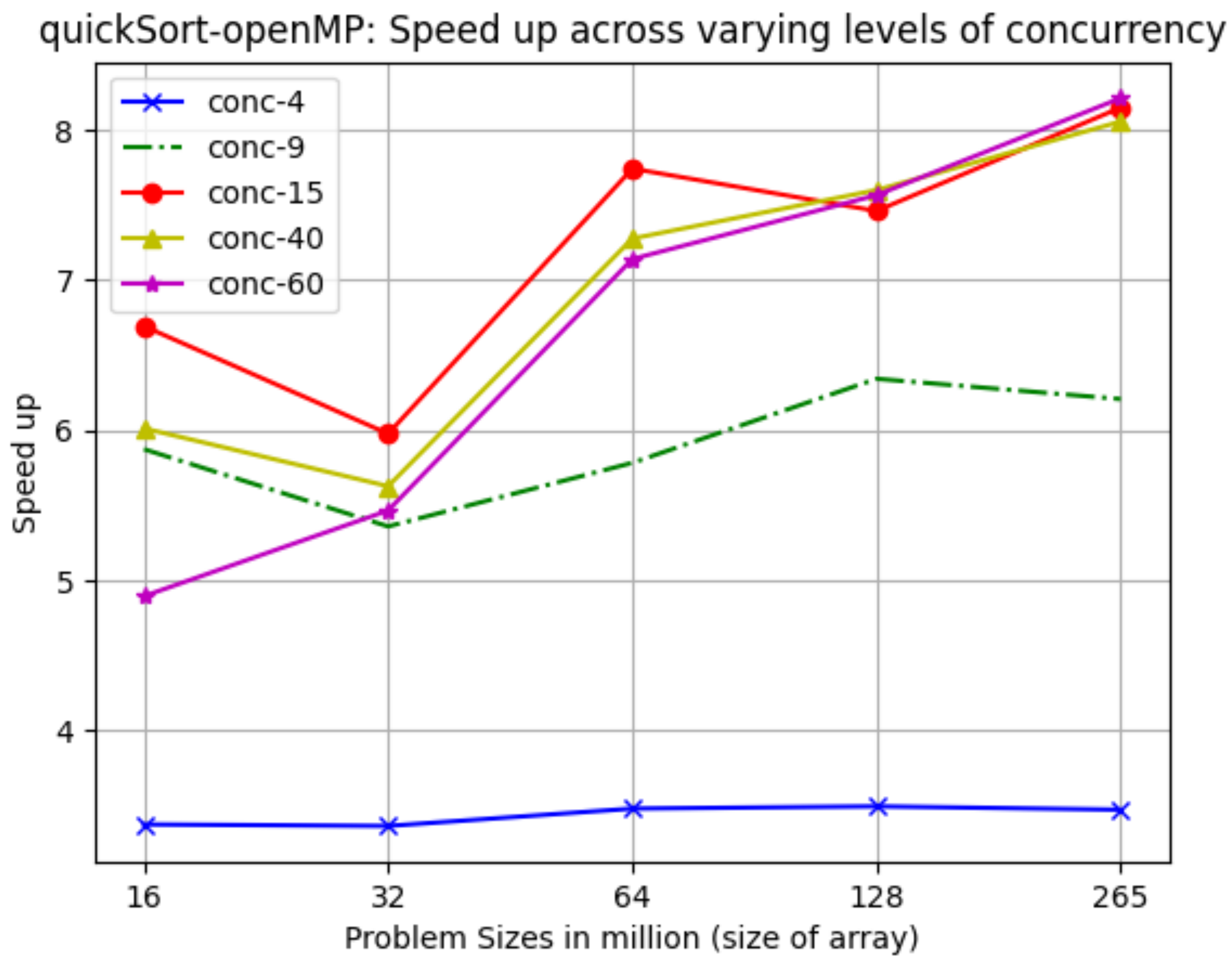
- conc-1 (sequential) has highest runtimes
- all other conc-levels are very similar for smaller problem sizes
- for larger problem sizes conc-4 and conc-9 show bigger runtimes
- conc-15, conc-40 and conc-60 remain very similar



Result: quickSort

Speed up across varying levels of concurrency

- speedup(x) = (conc-1) / (conc-x)
- speedup(4) remains steady rate of ~ 3.4
- a decline in speed up for problem size 32 million
- conc-15 has the best speed up across all problem sizes
- conc-15, conc-40 and conc-60 are more then 8 times faster then the sequential version



Findings

- significant performance improvement was achieved by parallelizing the quickSort algorithm (especially with concurrency level 15)
- openMP task construct is a good approach for parallelizing recursive sorting algorithms like quickSort
- increasing threads beyond a certain point no longer improves runtime
- minimal or no benefit when parallelizing very small arrays due to the overhead of task creation and thread assignment

The End