Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

*2018-2019 Q2*

**par2110**

Maurici Abad Gutierrez

Nicolas Schrack

# Introduction

The goal of this deliverable is to observe and practise different parallelization strategies on the Multisort algorithm.

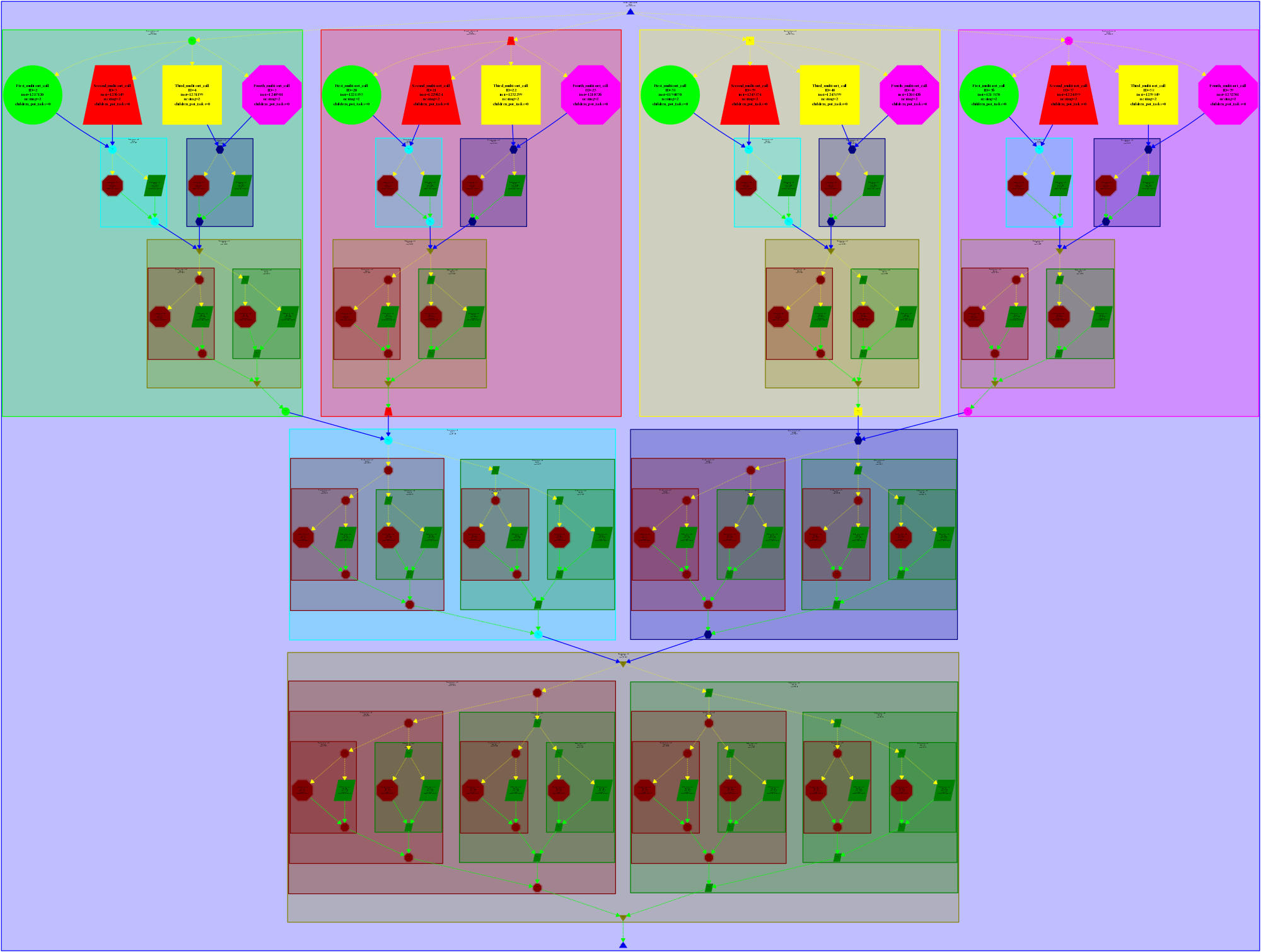
This recursive algorithm is ideal for this, because it can be splitted in a lot of little different tasks with smaller problem size and then merged into one (following the divide and conquer strategy).

In the first part we will analyze the parallelism of multisort-tareador.c code provided in order to fully understand its dependencies and develop a proper parallelized version of it.

Next, we will consider several different parallelization strategies: Leaf strategy, Tree strategy, Tree strategy using a cut-off mechanism and Tree depend strategy.

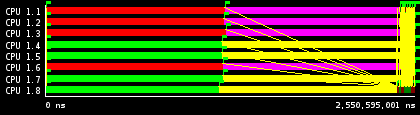
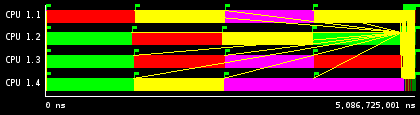
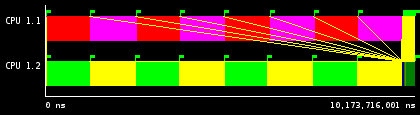
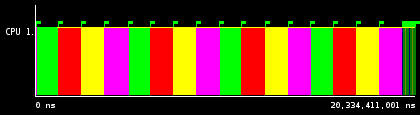
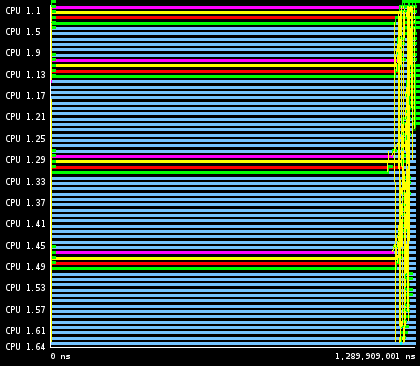
# Task decomposition analysis for Mergesort

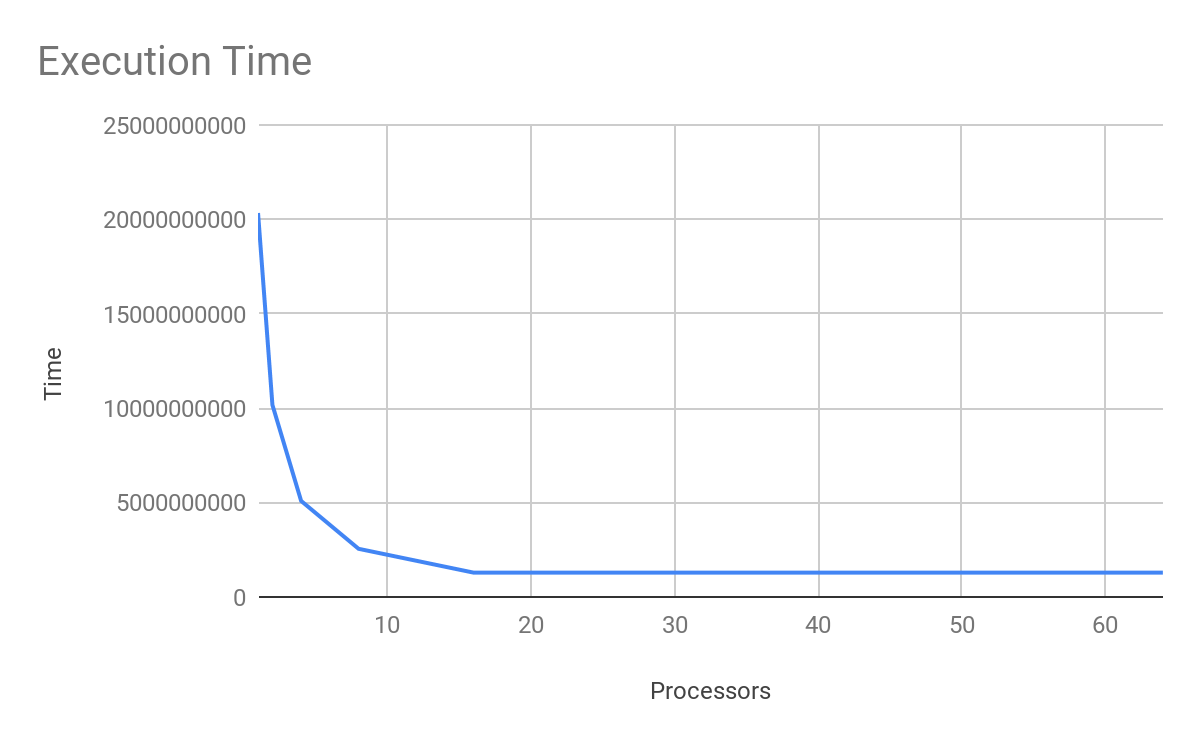
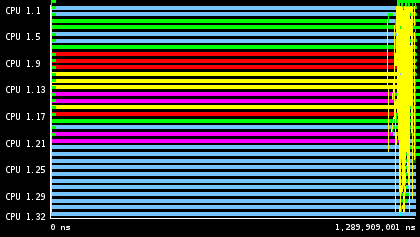
|  |
| --- |
| void merge(long n, T left[n], T right[n], T result[n\*2], long start, long length) {  if (length < MIN\_MERGE\_SIZE\*2L) {  // Base case  basicmerge(n, left, right, result, start, length);  } else {  // Recursive decomposition  tareador\_start\_task("Fourth merge call");  merge(n, left, right, result, start, length/2);  tareador\_end\_task("Fourth merge call");  tareador\_start\_task("Fifth merge call");  merge(n, left, right, result, start + length/2, length/2);  tareador\_end\_task("Fifth merge call");  } } void multisort(long n, T data[n], T tmp[n]) {  if (n >= MIN\_SORT\_SIZE\*4L) {  // Recursive decomposition  tareador\_start\_task("First multisort call");  multisort(n/4L, &data[0], &tmp[0]);  tareador\_end\_task("First multisort call");  tareador\_start\_task("Second multisort call");  multisort(n/4L, &data[n/4L], &tmp[n/4L]);  tareador\_end\_task("Second multisort call");  tareador\_start\_task("Third multisort call");  multisort(n/4L, &data[n/2L], &tmp[n/2L]);  tareador\_end\_task("Third multisort call");  tareador\_start\_task("Fourth multisort call");  multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L]);  tareador\_end\_task("Fourth multisort call");   tareador\_start\_task("First merge call");  merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);  tareador\_end\_task("First merge call");  tareador\_start\_task("Second merge call");  merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L);  tareador\_end\_task("Second merge call");    tareador\_start\_task("Third merge call");  merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);  tareador\_end\_task("Third merge call");  } else {  // Base case  basicsort(n, data);  } } |



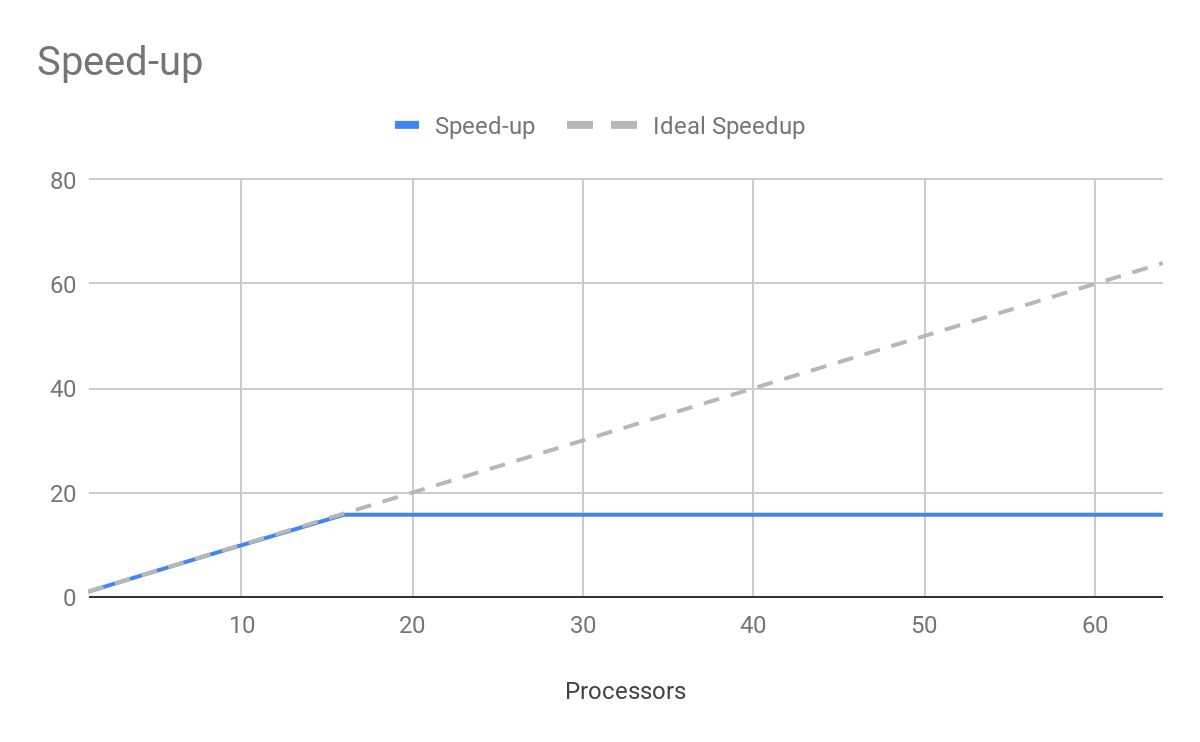
We see that the dependencies are mainly in the merging part of the algorithm.

# Execution Time and Speedup Table



|  |  |  |
| --- | --- | --- |
| Processors | Time | Speed-up |
| 1 | 20334411001 | 1 |
| 2 | 10173716001 | 1,998720133 |
| 4 | 5086725001 | 3,997544785 |
| 8 | 2550595001 | 7,97241859 |
| 16 | 1289922001 | 15,76406247 |
| 32 | 1289909001 | 15,76422134 |
| 64 | 1289900001 | 15,76433133 |



This results are close to ideal until we reach the critical path, at 16 processors.

# Shared-memory parallelization with OpenMP tasks

## Leaf strategy

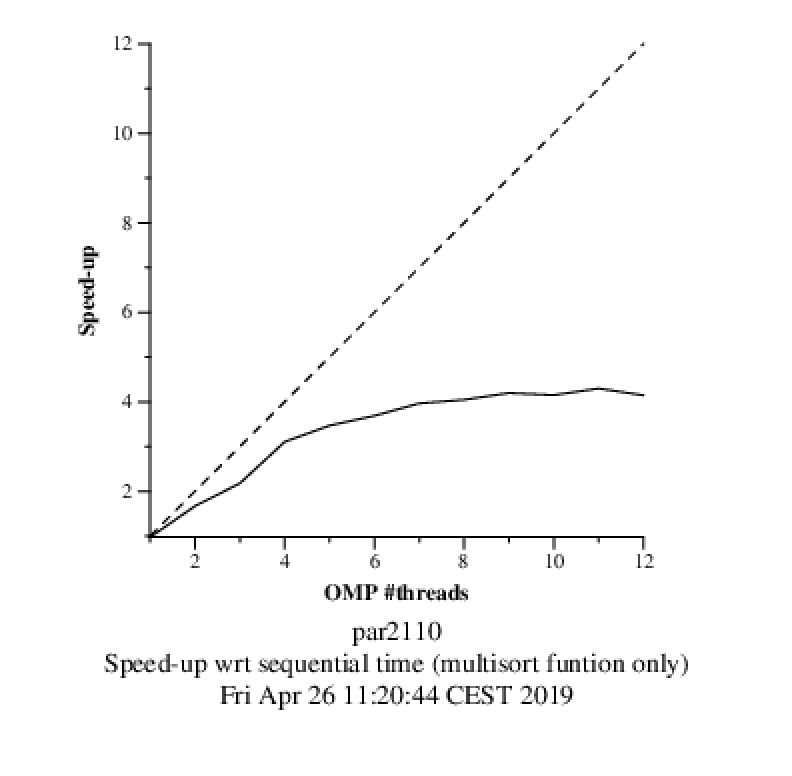
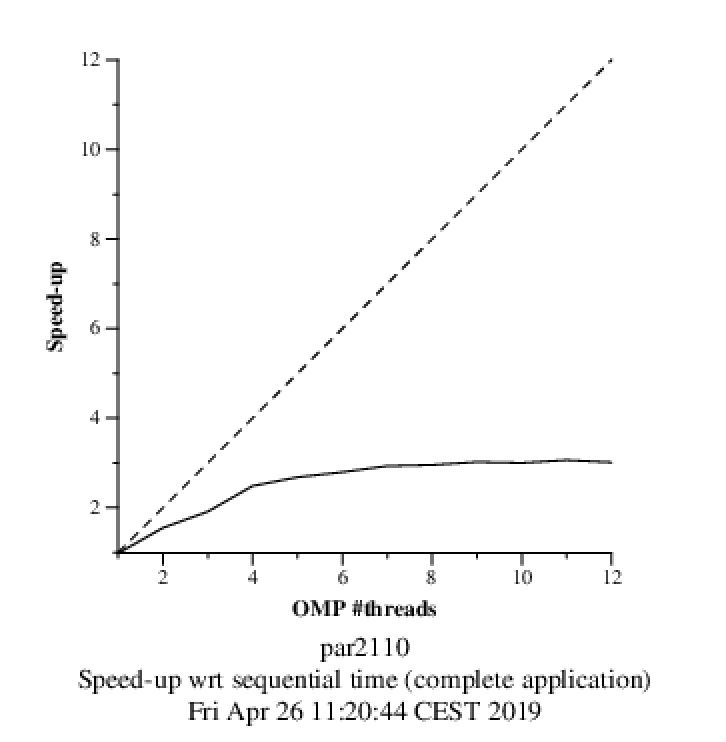
### Code

|  |
| --- |
| **void** **merge**(**long** n, T left[n], T right[n], T result[n\*2], **long** start, **long** length) {  **if** (length < MIN\_MERGE\_SIZE\*2L) {  *// Base case*  **#pragma omp task**  basicmerge(n, left, right, result, start, length);  } **else** {  *// Recursive decomposition*  merge(n, left, right, result, start, length/2, depth+1);  merge(n, left, right, result, start + length/2, length/2, depth+1);  } }  **void** **multisort**(**long** n, T data[n], T tmp[n]) {  **if** (n >= MIN\_SORT\_SIZE\*4L) {  *// Recursive decomposition*  multisort(n/4L, &data[0], &tmp[0], depth+1);  multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);  multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);  multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L], depth+1);   **#pragma omp taskwait**   merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);  merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L, depth+1);   **#pragma omp taskwait**   merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);   } **else** {  *// Base case*  #pragma omp task  basicsort(n, data);  } } ... **int** **main**(**int** argc, **char** \*\*argv) {  ...  **#pragma omp parallel**  **#pragma omp single**  multisort(N, data, tmp);  ...  } |

### Conclusion

Initialization time in seconds: 0.818585  
Multisort execution time: 1.553404  
Check sorted data execution time: 0.020209

We will never get a better speedup than 4, because synchronization is inefficient. We don’t create tasks until we reach the leaf, so they can’t be done before.



## 

## Tree without cutoff strategy

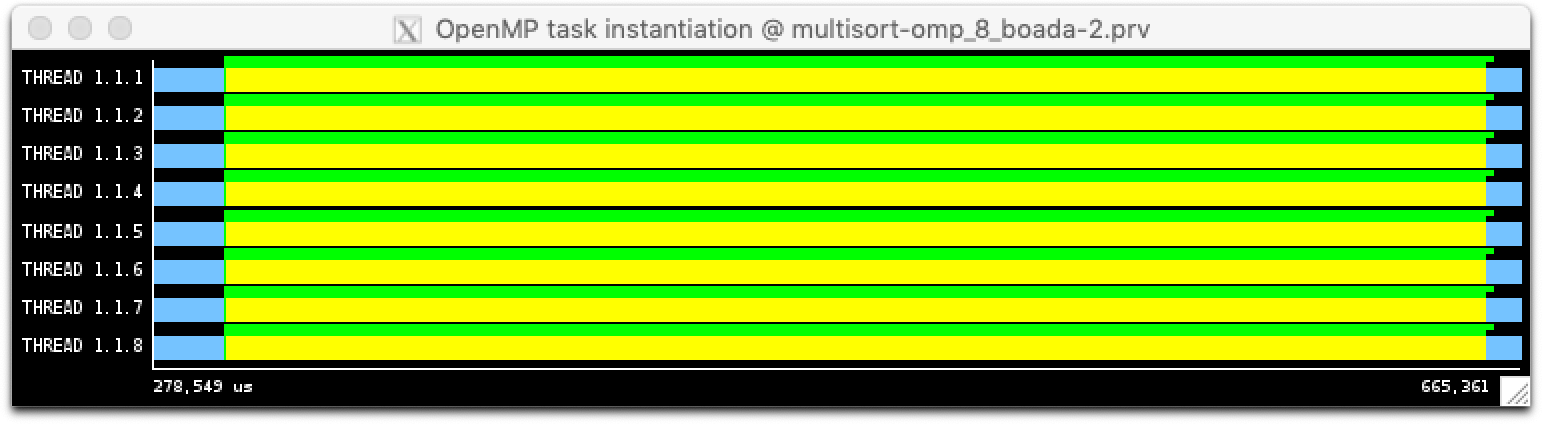
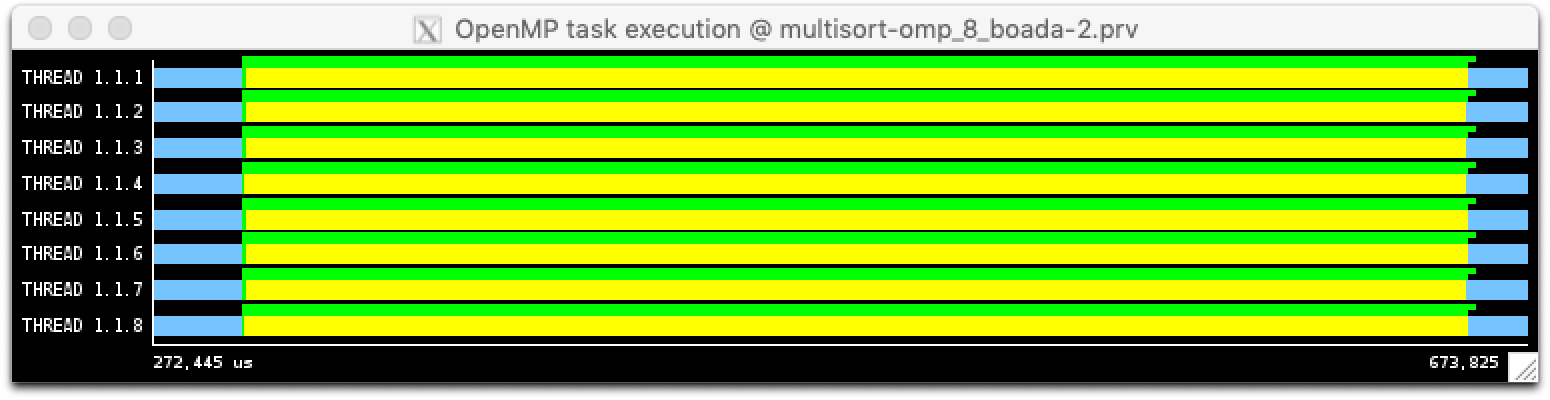
### Code

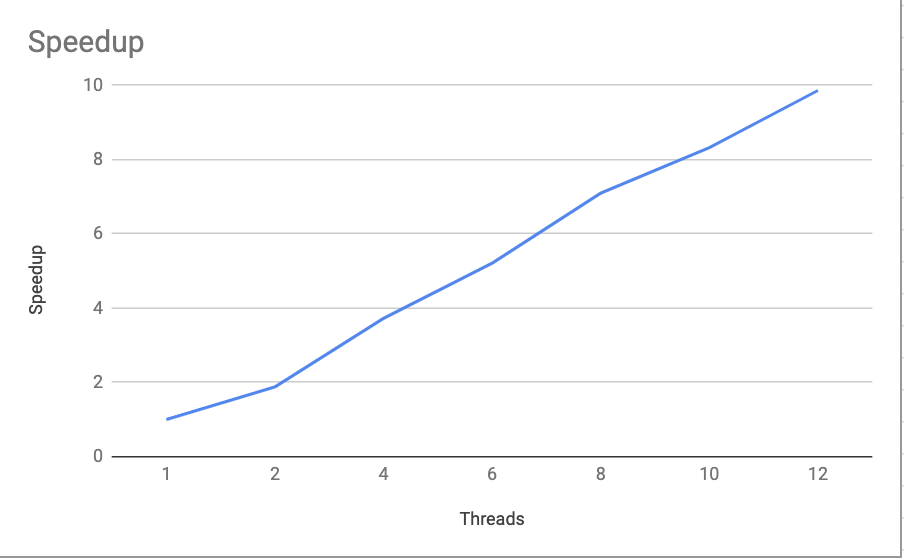
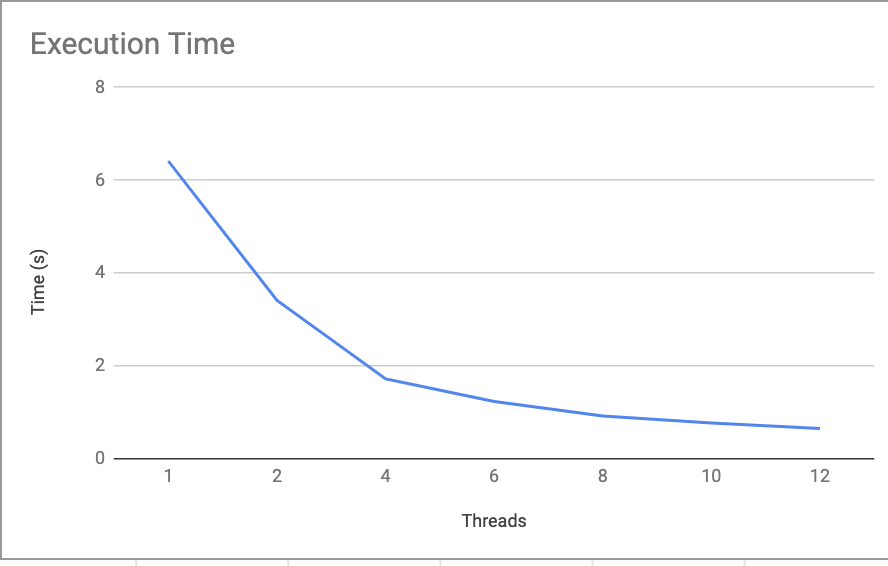
|  |
| --- |
| **void** **merge**(**long** n, T left[n], T right[n], T result[n\*2], **long** start, **long** length) {  **if** (length < MIN\_MERGE\_SIZE\*2L) {  *// Base case*  basicmerge(n, left, right, result, start, length);  } **else** {  *// Recursive decomposition*  **#pragma omp task**  merge(n, left, right, result, start, length/2, depth+1);  **#pragma omp task**  merge(n, left, right, result, start + length/2, length/2, depth+1);  } }  **void** **multisort**(**long** n, T data[n], T tmp[n]) {  **if** (n >= MIN\_SORT\_SIZE\*4L) {  *// Recursive decomposition*  **#pragma omp taskgroup**   {  **#pragma omp task**  multisort(n/4L, &data[0], &tmp[0], depth+1);  **#pragma omp task**  multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);  **#pragma omp task**  multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);  **#pragma omp task**  multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L], depth+1);  }   **#pragma omp taskgroup**   {  **#pragma omp task**  merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);  **#pragma omp task**  merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L, depth+1);  }   **#pragma omp task**  merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);  } **else** {  *// Base case*  basicsort(n, data);  } }  ...  **int** **main**(**int** argc, **char** \*\*argv) {  ...  **#pragma omp parallel**  **#pragma omp single**  multisort(N, data, tmp);  ...  } |

### Conclusion

Initialization time in seconds: 0.820284  
Multisort execution time: 0.960214  
Check sorted data execution time: 0.015376

This strategy is not ideal because it creates too many tasks, so we need a cut-off.





## Tree with cutoff strategy

### Code

|  |
| --- |
| **void** **merge**(**long** n, T left[n], T right[n], T result[n\*2], **long** start, **long** length, **int** depth) {  **if** (length < MIN\_MERGE\_SIZE\*2L) {  *// Base case*  basicmerge(n, left, right, result, start, length);  } **else** {  if (!omp\_in\_final()) {  *// Recursive decomposition*  **#pragma omp task final(depth >= CUTOFF)**  merge(n, left, right, result, start, length/2, depth+1);  **#pragma omp task final(depth >= CUTOFF)**  merge(n, left, right, result, start + length/2, length/2, depth+1);  }else{  merge(n, left, right, result, start, length/2, depth+1);  merge(n, left, right, result, start + length/2, length/2, depth+1);  }  } }  **void** **multisort**(**long** n, T data[n], T tmp[n], **int** depth) {  **if** (n >= MIN\_SORT\_SIZE\*4L) {  if (!omp\_in\_final()) {  *// Recursive decomposition*  **#pragma omp taskgroup**   {  **#pragma omp task final(depbasicsort(n, data)th >= CUTOFF)**  multisort(n/4L, &data[0], &tmp[0], depth+1);  **#pragma omp task final(depth >= CUTOFF)**  multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);  **#pragma omp task final(depth >= CUTOFF)**  multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);  **#pragma omp task final(depth >= CUTOFF)**  multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L], depth+1);  }   **#pragma omp taskgroup**   {  **#pragma omp task final(depth >= CUTOFF)**  merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);  **#pragma omp task final(depth >= CUTOFF)**  merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L, depth+1);  }   **#pragma omp task final(depth >= CUTOFF)**  merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);  } else {  multisort(n/4L, &data[0], &tmp[0], depth+1);  multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);  multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);  multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L], depth+1);    merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);  merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L, depth+1);    merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);  }   } **else** {  *// Base case*  basicsort(n, data);  } } ... **int** **main**(**int** argc, **char** \*\*argv) {  ...  **#pragma omp parallel**  **#pragma omp single**  ...  ...  } |
|  |

### Conclusion

Initialization time in seconds: 0.824397s  
Multisort execution time: 0.866247s  
Check sorted data execution time: 0.015769s

This is the best version of the three.

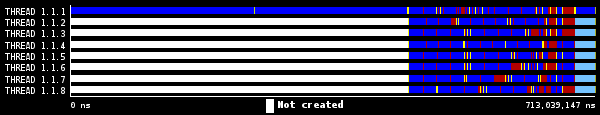
### Traces



*Trace with CUTOFF=0*



*Trace with CUTOFF=1*



*Trace with CUTOFF=2*



*Trace with CUTOFF=3*



*Trace with CUTOFF=4*

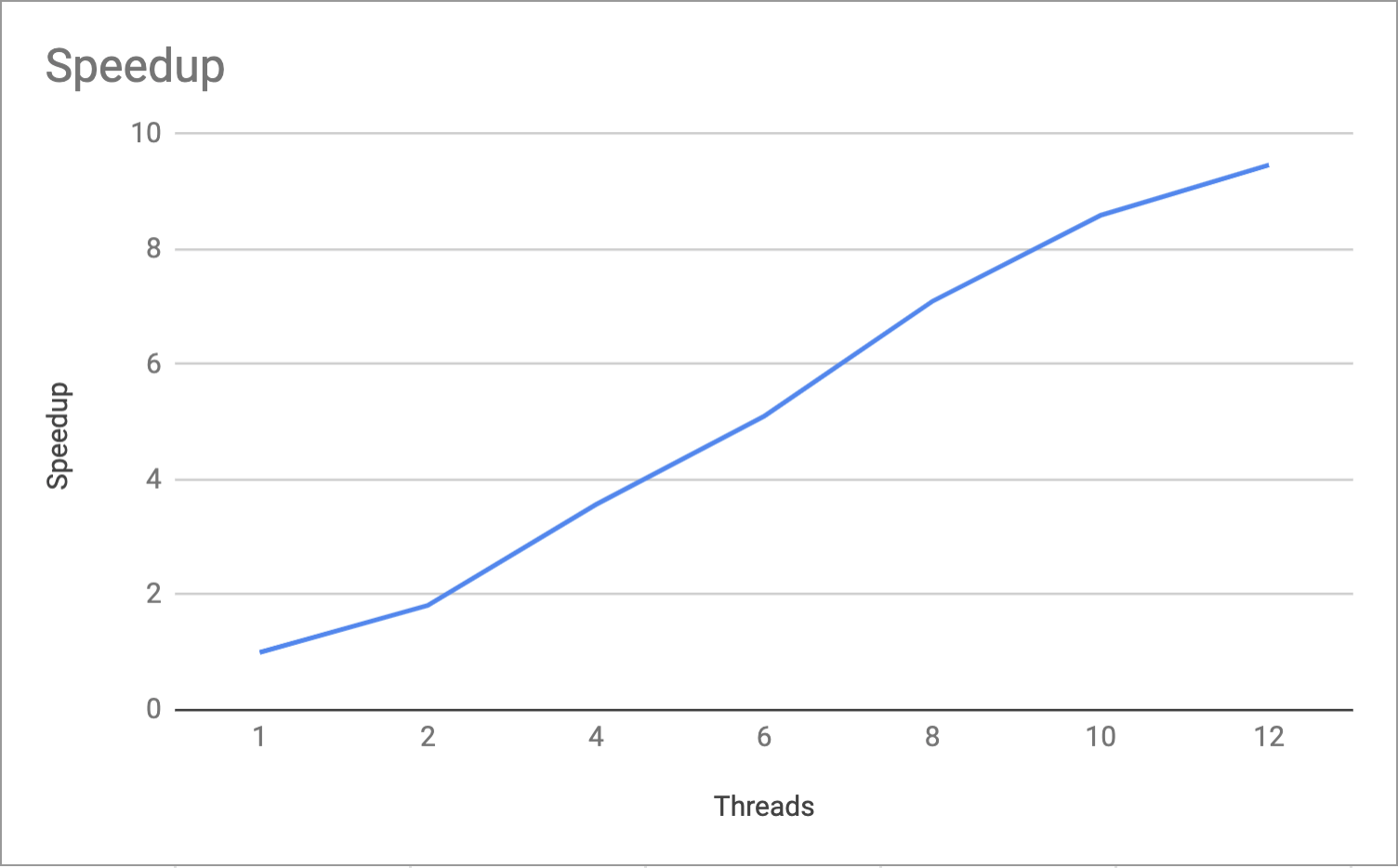
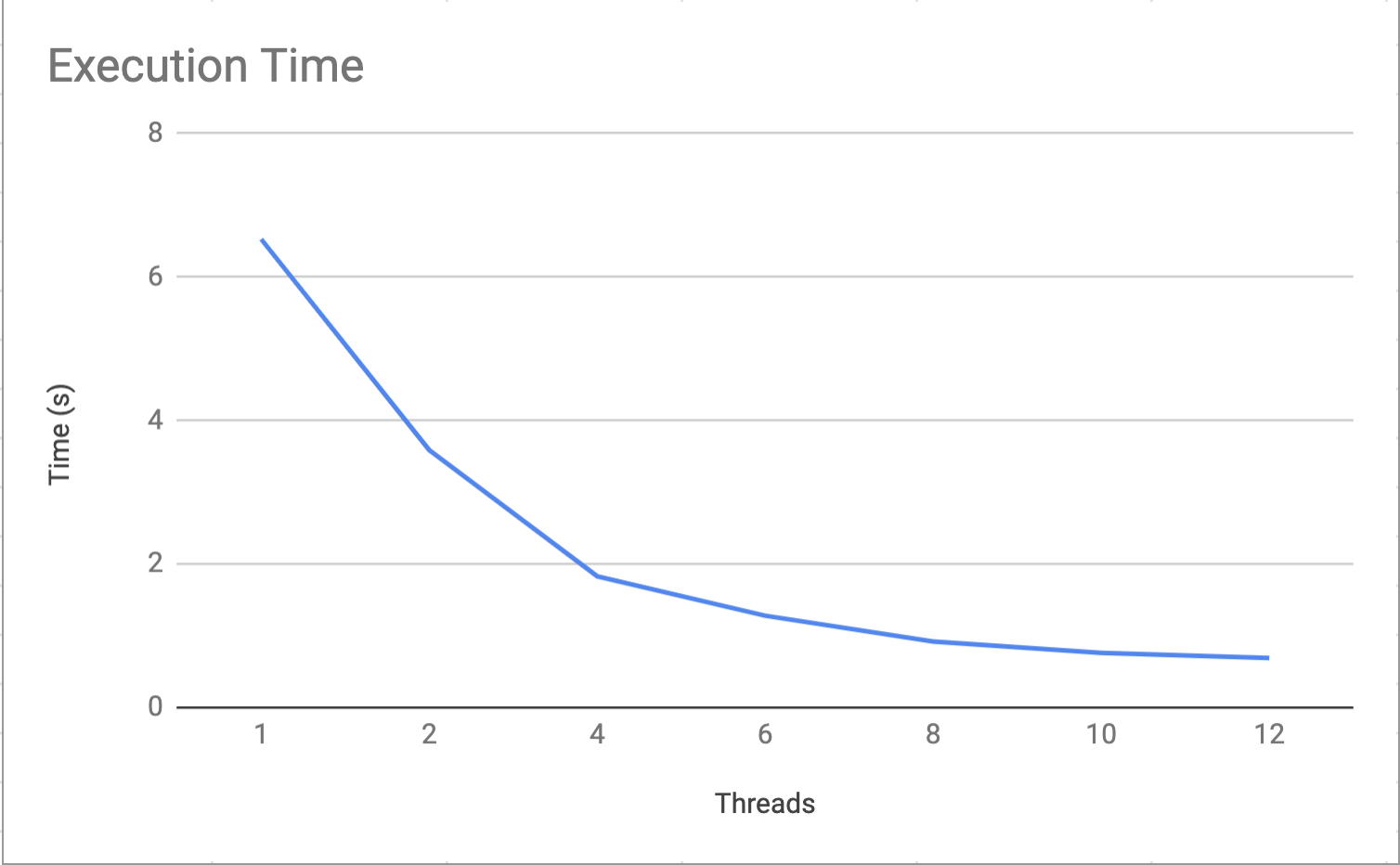
## Tree depend strategy

### Code

|  |
| --- |
| **void** **merge**(**long** n, T left[n], T right[n], T result[n\*2], **long** start, **long** length) {  **if** (length < MIN\_MERGE\_SIZE\*2L) {  *// Base case*  basicmerge(n, left, right, result, start, length);  } **else** {  *// Recursive decomposition*  #pragma omp task  merge(n, left, right, result, start, length/2, depth+1);  **#pragma omp task**  merge(n, left, right, result, start + length/2, length/2, depth+1);  **#pragma omp taskwait**  } } **void** **multisort**(**long** n, T data[n], T tmp[n]) {  **if** (n >= MIN\_SORT\_SIZE\*4L) {  *// Recursive decomposition*  **#pragma omp task depend(out: data[0])**  multisort(n/4L, &data[0], &tmp[0], depth+1);  **#pragma omp task depend(out: data[n/4L])**  multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);  **#pragma omp task depend(out: data[n/2L])**  multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);  **#pragma omp task depend(out: data[3L\*n/4L])**  multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L], depth+1);   **#pragma omp task depend(in:data[0], data[n/4L]) depend(out: tmp[0])**  merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);  **#pragma omp task depend(in:data[n/2L], data[3L\*n/4L]) depend(out: tmp[n/2L])**  merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L, depth+1);   **#pragma omp task depend(in: tmp[0], tmp[n/2L])**  merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);   **#pragma omp taskwait**  } **else** {  *// Base case*  basicsort(n, data);  } }  ...  **int** **main**(**int** argc, **char** \*\*argv) {  ...  **#pragma omp parallel**  **#pragma omp single**  multisort(N, data, tmp);  ... } |

### Conclusion

Initialization time in seconds: 0.818794s  
Multisort execution time: 0.919293s  
Check sorted data execution time: 0.015405s



## Optional optimizations

### Code

|  |
| --- |
| **static** **void** **initialize**(**long** length, T data[length]) {  **long** i;  **int** first = 1;  **#pragma omp parallel for schedule(static,1) firstprivate(first)**  **for** (i = 0; i < length; i++) {  **if** (first) {  data[i] = rand();  first = 0;  } **else** {  data[i] = ((data[i-1]+1) \* i \* 104723L) % N;  }  } }  **static** **void** **clear**(**long** length, T data[length]) {  **long** i;  **#pragma omp parallel for schedule(static,1)**  **for** (i = 0; i < length; i++) {  data[i] = 0;  } } |

### Conclusion

Initialization time in seconds: 0.386720  
Multisort execution time: 0.931427  
Check sorted data execution time: 0.017691

**As we see, the initialization is 2.1 times faster.**

### Plots

