Parallel Sorting in OpenMP

Daniel Langr, Ivan Šimeček, Pavel Tvrdík

Department of Computer Systems
Faculty of Information Technology
Czech Technical University in Prague

Parallel and Distributed Programming (MIE-PDP)
Summer Semester 2020/21, Lecture 05

https://courses.fit.cvut.cz/MIE-PDP

Lecture Outline

1. QuickSort and MergeSort
2. QuickSort
    - Sequential algorithm
    - Straightforward parallelization using task parallelism
    - Parallel algorithm without tail recursion
    - Parallel algorithm with task parallelism thresholding
    - Parallelization of partitioning
    - Final performance evaluation of QuickSort implementations
3. MergeSort
    - Sequential MergeSort
    - Straightforward parallelization using task parallelism
    - Basic optimizations
    - Parallel 2-way merge
    - Parallel multi-way merge

Comparison of MergeSort and QuickSort

- Both are **recursive Divide-and-Conquer**.

QuickSort

- **Data sensitive**: In „most“ cases the fastest seq. sorting algorithm.
- **In-place**: It requires only stack the recursive call structures:

  $O(\log_2 n)$ in case of efficient implementations.
- Can be implemented exclusively with the **C(ompare)&S(wap)** operation.
- **Unstable** (= the ordering of equal elements is not preserved).

MergeSort

- **Data oblivious**: In „most“ cases the fastest par. sorting algorithm.
- **Out-of-place**: It needs an aux. array of the same size as the input.
- Implication: it cannot be implemented with only the C&S operations.
- **Stable** (= the ordering of equal elements is preserved).

Sequential textbook (Wikipedia) version of QuickSort I

In-place C implementation on array A of size n (Lomuto's version).

```
void swap(int* A, long i, long j) { //exchange A[i] and A[j]
   int temp = A[i]; A[i] = A[j]; A[j] = temp;
}
void seq_quicksort(int* A, long lo, long hi) {
   if (lo < hi) {
      long r = seq_partition(A, lo, hi); //A[r] == pivot
      seq_quicksort(A, lo, r - 1); // recursive sorting of the left part
      seq_quicksort(A, r + 1, hi); // recursive sorting of the right part
}  }
long seq_partition(int* A, long lo, long hi) {// Lomuto's variant
   long pivot = A[hi]; //pivot becomes the last element
   long i = lo; //elements left from i are < pivot
   for (long j = lo; j < hi; j++)
      if (A[j] < pivot)
         swap(A, i++, j); // exchange A[i] and A[j], increment i
   swap(A, i, hi); // pivot placement at the correct position
   return i; // return the pivot index
}
```

Sequential textbook version of QuickSort II

- Basic version that does not handle special cases.
- Using 2 pointers `i` and `j` the array is traversed left-to-right so that the following **invariant** holds:
    - After iteration `j` all numbers in `A[lo,...,i-1]` are $<$ than pivot and all numbers in `A[i,...,j]` are $\geq$ to pivot.
- Once `j` approaches `A[hi]`, `A[i]` and `A[hi]` needs to be swapped.
- Then $\forall$ `lo` $\leq$ `k` $\leq$ `i-1`; `A[k]`$<$`A[i]` and $\forall$ `i+1` $\leq$ `k` $\leq$ `hi`; `A[k]`$\geq$`A[i]`.

An application example for 100 millions of random numbers

```
int main() {
   static const long n = 100000000;
   int* A = (int*)malloc(n * sizeof(int));
   for (long i = 0; i < n; i++)
      A[i] = rand();
   seq_quicksort(A, 0, n - 1);
   free(A);
}
```

Straightforward parallelization of QuickSort

- Similarly to the parallelization of the subtree size computation in Lecture 2 we will apply **functional parallelism**.
- Recursive calls will be parallel thanks to directive `#pragma omp task`.

```
void par_quicksort(int* A, long lo, long hi) {
   #pragma omp parallel // forking of threads
   {
      #pragma omp single // processing of the recursion call tree root ..
      par_quicksort_rec(A, lo, hi); // ... by a single thread
}  }
void par_quicksort_rec(int* A, long lo, long hi) {
   if (lo < hi) {
      long r = seq_partition(A, lo, hi);
      #pragma omp task // 1st task into the task pool
      par_quicksort_rec(A, lo, r - 1);
      #pragma omp task // 1st task into the task pool
      par_quicksort_rec(A, r + 1, hi);
   }
}
```

Evaluation of the `QuickSort` implementations I

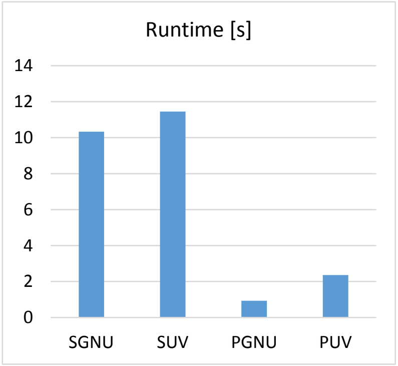
Compared implementations:

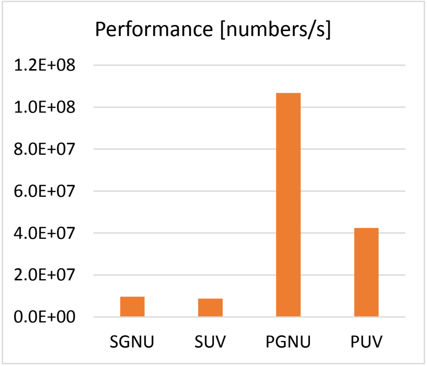
1. Reference implementation: highly optimized variants of the `QuickSort` algorithm from `Libstdc++` (= GNU implementation of the standard library C++). It includes the sequential version (SGNU) and parallel OpenMP version (PGNU).
2. Implementation of textbook versions: sequential (SUV) and parallel (PUV).

Experiment:

- Sorting of 100 millions of random numbers.
- Hardware: Intel Xeon E5-2680v3 (Haswell) with 12 cores.
- Software: GNU GCC version 5.4.

Evaluation of the `QuickSort` implementations II





Observation:

1. SUV exhibits only a bit worse performance than SGNU.
2. PGNU on 12 cores achieves speedup 11,03, …
3. … whereas PUV on 12 cores achieves speedup **only** 4,86!!!

PUV analysis

The main reasons of the small speedup of PUV:

1. Forking of a large number of OpenMP tasks ⇒ big overhead of the task pool management and control.
2. Function `seq_partition` is sequential ⇒ partitioning of the input array is performed only by one thread (see the Amdahl law in Lecture 1) and at several initial levels, some threads will remain idle.

Three possibilities of the solution:

1. **Removal of the tail recursion**, i.e., replacement of one recursion call with the iteration (*tail call optimization*).
2. **Thresholding** of the array size for forking of new OpenMP tasks.
3. **Parallelization** of the `seq_partition` algorithm.

Parallel QuickSort: Tail call optimization (TCO)

One recursion call replaced with iteration loop

```
void par_quicksort_rec(int* A, long lo, long hi) {
 if (lo < hi) {                         while (lo < hi) {
  long r = seq_partition(A, lo, hi);     long r = "seq_partition!(A, lo, hi);
  #pragma omp task //left part            #pragma omp task
  par_quicksort_rec(A, lo, r - 1);       par_quicksort_rec(A, lo, r - 1);
  #pragma omp task //right part           lo = r + 1 //2nd task will
  par_quicksort_rec(A, r + 1, hi);       //be executed by the calling thead
 } }                                     }
```

Analysis:

- The number of recursive calls drops to one half, since one half of them is replaced with iteration back.
- In OpenMP the number of created tasks drops therefore also to one half and the task pool overhead is decreased.
- Another optimization where we apply recursive calls only to smaller of the two parts and reiterate the greater part within the loop, enables to reduce the system stack depth to $O(\log n)$ for any input data.

Parallel `QuickSort`: Task parallelism thresholding (ST) I

- At the beginning, the number of threads is set up with `p=omp_get_num_threads()`.
- If the size of unsorted parts of the input array shrinks under a given **threshold**, e.g., `n/p`, we can sort these parts with sequential `QuickSort` within the current task.
- In dependence on input data threshold `n/p` can cause problems with thread load balancing.
  - For such cases it is better to use smaller threshold, e.g. `n/(k*p)` for some sufficiently big constant `k > 1`.
  - The total number of tasks in the task pool is several times greater the the number of threads `p` and by consuming tasks from and producing tasks to the task pool the threads **implement automatic load balancing**.

Parallel QuickSort: Task parallelism thresholding (ST) II

```
void par_quicksort(int* A, long lo, long hi) {
   static const long k = ...
   #pragma omp parallel
   { const long seq_thr = (hi - lo + 1) / omp_get_num_threads() / k;
     #pragma omp single
     par_quicksort_rec(A, lo, hi, seq_thr);
}  }

void par_quicksort_rec(int* A, long lo, long hi, const long seq_thr) {
   while (lo < hi) {
      if ((hi - lo) < seq_thr) {
         seq_quicksort(A, hi, lo); // sequential QuickSort variant
         return;
      }
      long r = seq_partition(A, lo, hi);
      #pragma omp task
      par_quicksort_rec(A, lo, r - 1, seq_thr);
      lo = r + 1
}  }
```

Parallel `QuickSort`: Parallelizable partitioning I

- We consider the following variant of sequential **2-way** array partitioning using a pivot (the original Hoare's variant).
- The array is traversed **simultaneously from left and right ends** and the corresponding pairs of elements will be compared with the pivot.
- The following 4 cases can happen:
  1. The left elem is less than the pivot and the right one is greater or equal to the pivot ⇒ both elems are on right places and we can move the pointers innerward.
  2. Both elems are less than the pivot ⇒ the left elem is on the right position, shift the left pointer rightward.
  3. Both elems are greater than or equal to the pivot ⇒ the right elem is on the right position, shift the right pointer leftward.
  4. The left elem is greater or equal to the pivot and the right elem is less than the pivot ⇒ swap both elems and move both pointers innerward.
- This is executed repeatedly until both pointers meet.
- This process is called **neutralization**.
- In each iteration, at least one array element is neutralized.

Parallel QuickSort: Parallelizable partitioning II

An example of the sequential algorithm of 2-way partitioning:

```
long seq_partition_2(int* A, long lo, long hi) { // Hoare's variant
  long pivot = A[hi];
  long i = lo;      // index of rightward traversal
  long j = hi - 1; // index of leftward traversal
  while (i < j) {
    if (A[i]<pivot) && (A[j]>=pivot) { i++; j--;}
    else if (A[i]>=pivot) && (A[j]<pivot) { swap(A,i,j); i++; j--;}
    else if (A[i]>=pivot) && (A[j]>=pivot) j--;
    else i++;    //if (A[i] < pivot) && (A[j]<pivot)
  }
  if (A[j] < pivot) j++; //correction only if index j overran too far
  swap(A,j,hi);  // place the pivot to the correct position
  return j;      // return the pivot's position
}
```

Parallel QuickSort: Parallelizable partitioning III

The previous algorithm can be simplified by optimizing the ordering of condition evaluation of those 4 cases:

```
long seq_partition_3(int* A, long lo, long hi) { // Hoare's variant
   long pivot = A[hi];
   long i = lo;     // index of rightward traversal
   long j = hi - 1; // index of leftward traversal
   while (i < j) {
     if (A[i] >= pivot) && (A[j] < pivot) {
        swap(A, i, j); i++; j--; // both elems are neutralized
     } else { //at least one of A[i], A[j] is on the correct side
       if (A[j] >= pivot) j--; // the right element is neutralized
       if (A[i] < pivot) i++;  // the left element is neutralized
      }
   }
  if (A[j] < pivot) j++; //correction only if index j overran too far
  swap(A,j,hi);  // place the pivot to the correct position
  return j;      // return the pivot's position
}
```

Parallel `QuickSort`: Parallel partitioning I

The previous algorithm `seq_partition_3` can easily be parallelized:

1. Indexes of rightward and leftward traversals `i` and `j` will be **shared** variables.
2. Every thread needs for itself **to capture unique** values of these variables so that all threads can contribute to the partitioning by working on **disjoint pairs** of array elements.
3. At the end of the traversal, each thread will be left with **at most one dirty (i.e., incorrectly placed)** element.
4. These at most p dirty elements can be neutralized either sequentially ($O(p)$) or using parallel reduction ($O(\log p)$).
5. Since $p \ll n$, the sequential epilogue can be not even simpler but also faster.
6. This algorithm requires to have **nested OpenMP parallelism** switched on.
   - This is activated using the `omp_set_nested(1)` call.
   - The nested parallelism needs to be activated before calling the PGNU algorithm.

Parallel QuickSort: Parallel partitioning II

The code sketch without controlling the number of threads for partitioning.

```
void par_quicksort(Int* A, long lo, long hi) {
   static const long k = ...
   omp_set_nested(1);  //1 = true
   #pragma omp parallel
   {  const long seq_thr = (hi - lo + 1) / omp_get_num_threads() / k;
      #pragma omp single
      par_quicksort_rec2(A, lo, hi, seq_thr);
}  }
void par_quicksort_rec2(int* A, long lo, long hi, const long seq_thr) {
   while (lo < hi) {
      if ((hi - lo) < seq_thr) {
        seq_quicksort(A, hi, lo); return;
      }
      long r = par_partition(A, lo, hi); // nested parallelism
      #pragma omp task
      par_quicksort_rec2(A, lo, r - 1, seq_thr);
      lo = r + 1
}  }
```

Parallel `QuickSort`: Why the nested parallelism is needed?

- Algorithm `par_quicksort` on Slide 17 leads to a recursive call tree (RCT), expanded in parallel using p threads.
- The RCT root is created by the 1st single thread that gradually thanks to the task parallelism with the TCO always creates and places into the task pool a new task with numbers $<$ pivot and the thread itself takes care of the right part with numbers $\geq$ pivot.
- These new tasks are step-by-step taken by other threads so that finally all p threads are running and picking up the tasks from the task pool.
- One node of the RCT is therefore executed by one thread.
- However, when `par_quicksort_rec2` performs multithreaded partitioning, then each node of the RCT becomes a parallel region inside the main parallel region.
- For example, in the root node, all p threads can perform the parallel partitioning, since the task pool is empty at that time.
- In general, individual parallel threads themselves get temporarily forked to child threads running in parallel.

Parallel QuickSort: Parallel partitioning III

1st attempt for a parallel solution provides an INCORRECT algorithm.

```
long par_partition_1(int* A, long lo, long hi) {
   long pivot = A[hi];
   long i = lo; // shared index for rightward traversal
   long j = hi - 1; // shared index for leftward traversal
   #pragma omp parallel shared(i,j)
   {  long my_i, my_j;
      my_i = i++; // capture of a unique value of i
      my_j = j--; // capture of a unique value of j
      while (my_i < my_j) {
        if (A[my_i] >= pivot) && (A[my_j] < pivot) {
          swap(A, my_i, my_j); my_i = i++; my_j = j--; // capture both
        } else {
          if (A[my_j] >= pivot) my_j = j--; // capture a new right
          if (A[my_i] < pivot) my_i = i++; // capture a new left
      } } } // end of the parallel region
   ... // seq. proc. of remaining max. p elems (indexes stored as shared)
   if (A[j] < pivot) j++;  swap(A,j,hi); // pivot to the right position
   return j; // return of the pivot's position
}
```

Parallel `QuickSort`: Parallel partitioning IV

Why the `par_partition_1` algorithm is incorrect?

- Variables `i` and `j` are shared and therefore they **cannot** be accessed within a parallel region without synchronization.
- There are 3 possible solutions in OpenMP: critical sections, atomic operations, mutexes.
- Atomic operations are the best choice wrt efficiency.
- Atomic operations of type „get the value and increment“ (`Fetch-and-Add`) $\Rightarrow$ directive `atomic capture`.

Parallel QuickSort: Parallel partitioning V

2nd attempt for a parallel solution is correct.

```
long par_partition_2(int* A, long lo, long hi) {
   long pivot = A[hi];
   long i = lo; // shared index for rightward traversal
   long j = hi - 1; // shared index for leftward traversal
   #pragma omp parallel
   {  long my_i, my_j;
      #pragma omp atomic capture
      my_i = i++; // capture of a unique value of i
      #pragma omp atomic capture
      my_j = j--; // capture of a unique value of j
      while (my_i < my_j) {
         if (A[my_i] >= pivot) && (A[my_j] < pivot) {
            swap(A, my_i, my_j);
            #pragma omp atomic capture
            my_i = i++; // capture of another unique value of i
            #pragma omp atomic capture
            my_j = j--; // capture of another unique value of j
         ... // etc.
```

Parallel QuickSort: Parallel partitioning VI

We will use in pseudocode [....] as an abbrv for #omp atomic capture {...}.

```
long par_partition_2(int* A, long lo, long hi) {
   long pivot = A[hi];
   long i = lo; long j = hi - 1;
   #pragma omp parallel
   {  long my_i, my_j;
      [my_i = i++]; [my_j = j--];
      while (my_i < my_j) {
         if (A[my_i] >= pivot) && (A[my_j] < pivot) {
            swap(A, my_i, my_j);
            [my_i = i++]; [my_j = j--]; // capture of two new elems
         } else {
            if (A[my_j] >= pivot) [my_j = j--]; // capture of new right
            if (A[my_i] < pivot) [my_i = i++]; // capture of new left
   }  }  } // end parallel region
   ... // seq. proc. of remaining max. p elems (indexes stored as shared)
   if (A[j] < pivot) j++;  swap(A,j,hi); // pivot to the right position
   return j; // return of the pivot's position
}
```

Parallel QuickSort: Parallel partitioning VII

Why is the `par_partition_2` algorithm INEFFICIENT?

- Partitioning by **individual array elements** leads to:
  1. frequent atomic accesses into the shared memory, and therefore:
  2. frequent hits of the same cachelines by different threads (false sharing).
- **Solution**: capturing and processing of **disjoint blocks of elements**.

```
#pragma omp atomic capture
{ my_i = i; i += K; } // block size K
```

- **Definition: Dirty** block: contains elements both less than and greater or equal to pivot. Otherwise, a block is **clean**.
- **Observation**: After seq. partitioning of elements within both blocks, **at least** one of them becomes **clean** and **correctly** placed.
- Equally as in the non-block variant, the thread then captures new block(s) and continues.
- After the loop ends, **each thread** is left with **at most 1 dirty** block.
- These must be **neutralized** so that at most 1 dirty block remains.
- In general, the number of input elements `hi-lo+1` is not divisible by the block size `K` $\Rightarrow$ the last block size will be $<$ `K`.

Parallel QuickSort: Parallel partitioning VIII

```
long par_partition_block(int* A, long lo, long hi) {
   long pivot = A[hi]; long i = lo; long j = hi - 1;
   #pragma omp parallel
   {  long my_i, my_j, init_my_i, init_my_j;
      [my_i = i; i += K]; [my_j = j; j -= K]; //capture pointers to
      init_my_i = my_i; init_my_j = my_j; // asigned block begins
      while (my_i < my_j) {
         neutralization(A, my_i, init_my_i, my_j, init_my_j, K, pivot);
           // seq_partitioning of two blocks of size K
         if (my_i >= init_my_i + K)
          { [my_i = i; i += K]; init_my_i=my_i; } // left is clean
         if (my_j =< init_my_j - K)
          { [my_j = j; j -= K]; init_my_j=my_j; } // right is clean
      }
   #pragma omp barrier
   ...... // parallel neutralization of max. p dirty blocks
   }      // par.region end, 1 dirty block may still remain
   ...... // placement of the pivot to the correct position
   return j; // return of the pivot's position
}
```

Paralelní `QuickSort`: Thread load balancing

- Paralel partitioning (`neutralization`) is the main computational task of the `QuickSort`, the rest is pointer handling and a short epilogue.
- `QuickSort` is data sensitive: The ratio of the left and right part sizes for further recursion level depends on the quality of pivot selection.
- And the depth of the RCT depends on that (not related to parallelization), but in parallelization it makes the thread load balancing complicated.
- There is no space to go into details, but in principle we need in algorithm `par_quicksort_rec2` on Slide 17 a parameter that reflects the ratio `r-lo`:`hi-r`.
- If for example the ratio of the left and right part sizes is 2:5, it would be ideal to partition the thread pool for execution of the nested parallel regions (using clause `num_threads` in directive `#pragma omp parallel` inside `par_partition_block`) roughly in the same ratio.
- At the same, at least one thread must always be assigned to each part, however small it is.

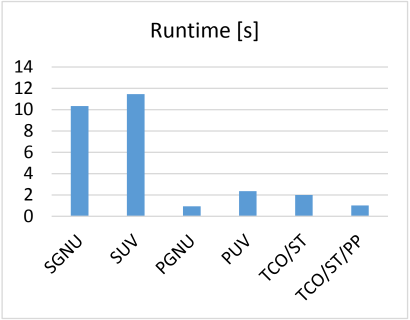
Parallel QuickSort with thread balancing

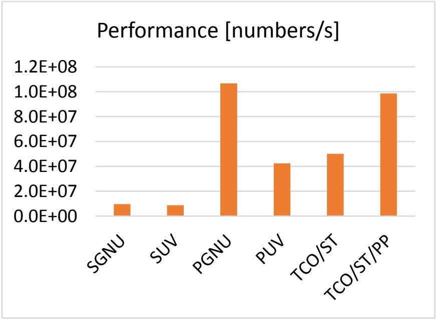
The idea of controlling the numbers of threads for parallel paritioning

```
void par_quicksort(int* A, long lo, long hi) {
  static const long k = ...
  omp_set_nested(1); //1 = true
  #pragma omp parallel
  { const long seq_thr = (hi - lo + 1) / omp_get_num_threads() / k;
    #pragma omp single
    par_quicksort_rec3(A, lo, hi, seq_thr, omp_get_num_threads());
}  }

par_quicksort_rec3(int* A, long lo, long hi, const long seq_thr, int nt) {
//nt je počet vláken které provádějí paralelní partitioning
  while (lo < hi) {
    if ((hi - lo) < seq_thr) { seq_quicksort(A, hi, lo); return; }
    long r = par_partition_block(A, lo, hi, nt); //nested parallelism
    //nt vláken nyní rozdělím v poměru r-lo:hi-r na lnt a rnt, obě >=1
    #pragma omp task
    par_quicksort_rec3(A, lo, r - 1, seq_thr, lnt);
    lo = r + 1; nt = rnt;
}  }
```

Parallel QuickSort: Final performance evaluation





1. TCO/ST = PUV + tail call optimization + thresholding of new tasks.
2. TCO/ST/PP = TCO/ST + `par_partition_block`.
3. PGNU is still a bit better, since other **optimizations** are applied (e.g., taking into account the cache memory effects, switching to `InsertionSort` for sufficiently small inputs, etc.).

Parallel `QuickSort`: Summary

- Parallel *partitioning* is an algorithmically complicated problem and its efficient integration into `QuickSort` is complicated as well (e.g., it is necessary to control the number of threads).
- What we have presented in this lecture is just a sketch. For more details, refer to source codes in `libstdc++`.
- For random data, the naive sequential variant SUV had almost the same performance as the optimized SGNU. Similarly, our parallel optimized code has almost the same performance as the best GNU implementation.
- However, this does not hold in general, since random numbers represent an ideal dataset for `QuickSort`.
- For different input data, GNU implementations have optimizations our codes do not have and we would get worse performance.
- Therefore, the implementation of PGNU is more complicated. For details, see again the source codes `libstdc++`.

Sequential textbook version of MergeSort I

C implementation on integer array A of size n from Wikipedia.

```
void seq_merge(int* A, long lo, long middle, long hi, int* B) {
   long i = lo; long j = middle;
   for (long k = lo; k < hi; k++) {
      if ((i < middle) && ((j >= hi) || (A[i] <= A[j]))) B[k] = A[i++];
      else B[k] = A[j++];
} }
void seq_mergesort_rec(int* B, long lo, long hi, int* A) {
   if ((hi - lo) < 2) return; // recursion hi
   long middle = (hi + lo) / 2;
   seq_mergesort_rec(A, lo, middle, B); //recur. sort of the left half
   seq_mergesort_rec(A, middle, hi, B); //recur. sort of the right half
   seq_merge(B, lo, middle, hi, A); //merge of both  halves
}
void seq_mergesort(int* A, long n) {
   int* B = (int*)malloc(n * sizeof(int)); // aux. array allocation
   for (long i = 0; i < n; i++) B[i] = A[i];
   seq_mergesort_rec(B, 0, n, A);
   free(B); }
```

Sequential textbook version of MergeSort II

An application example for 100 millions of random numbers

```
int main() {
   static const long n = 100000000;
   int* A = (int*)malloc(n * sizeof(int));

   for (long i = 0; i < n; i++)
      A[i] = rand();

   seq_mergesort(A, n);

   free(A);
}
```

Straightforward parallelization of MergeSort (PUV)

```
void par_mergesort(int* A, long n) {
   int* B = (int*)malloc(n * sizeof(int)); //aux.array allocation
   #pragma omp parallel
   {
      #pragma omp for
      for (long i = 0; i < n; i++) B[i] = A[i];
      #pragma omp single
      par_mergesort_rec(B, 0, n, A);
 }  free(B);
}
void par_mergesort_rec(int* B, long lo, long hi, int* A) {
   if ((hi - lo) < 2) return; // hi of the recursion
   long middle = (hi + lo) / 2;
   #pragma omp task
   par_mergesort_rec(A, lo, middle, B); //recur. sort of the left part
   #pragma omp task
   par_mergesort_rec(A, middle, hi, B); //recur. sort of the right part
   #pragma omp taskwait
   seq_merge(B, lo, middle, hi, A);
}
```

Performance evaluation of the MergeSort implementations

The same experiment, HW, and notation

	QuickSort	MergeSort
SGNU	10,33 [s]	10,77 [s]
PGNU	0,94 [s]	0,98 [s]
SUV	11,46 [s]	13,86 [s]
PUV	2,36 [s]	198,46 [s]

- The straightforward parallelization of MergeSort does not work as in case of QuickSort@!!!!
- Parallel MergeSort version PUV with 12 threads is 14 times slower than the sequential MergeSort algorithm SUV.
- Why?

Analysis and comparison of basic parallel versions

`QuickSort` PUV

- It performs the work first = function `partition`.
- Then it calls recursively itself on subarrays of various sizes.

`MergeSort` PUV

- The recursive calls for halves of input arrays are made first.
- After that, the work is done at return by performing the `merge` operation.

Corollary

- After the recursion in `MergeSort`, a lot of tiny OpenMP tasks that merge at most 2 elements are created at the same time.
- These tiny tasks are simultaneously executed by all threads that therefore have to access neighboring input array indexes $\Rightarrow$ false sharing.

Three possible improvements of the parallel `MergeSort`

1. **Thresholding** of the input size for creating parallel OpenMP tasks.
2. Instead of creating 2 new parallel recursive tasks of half size. A new task is created only for the left half, whereas the right half will be completed by the current thread sequentially.
   Thus, instead of the Divide-&-Conquer principle, the **Divide-&-KeepOneHalf** will be applied.
3. **Parallelization** of the `seq_merge` operation.

Parallel MergeSort: Threshold and Divide-&-KeepOneHalf

Let us denote by PUV+ST this version of MergeSort.

```
void par_mergesort_rec(int* B, long lo, long hi, int* A, long seq_thr) {
   if ((hi - lo) < seq_thr) {
      seq_mergesort_rec(B, lo, hi, A); // sequential version
      return;
   }
   long middle = (hi + lo) / 2;
   #pragma omp task
   par_mergesort_rec(A, lo, middle, B); //a new OpenMP task into the pool
   par_mergesort_rec(A, middle, hi, B); //current OpenMP task goes on
   #pragma omp taskwait
   seq_merge(B, lo, middle, hi, A);
}
```

Implementation and comparison of results

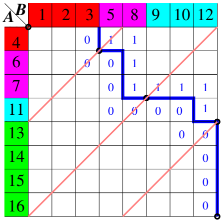
1. PUV+ST: 3,14 [s], instead of 198,46 [s] for PUV.
2. Speedup 4,41 wrt SUV, but still 4,5x slower than PGNU.

Parallel `MergeSort`: Parallelization of the `seq_merge` function

- The classical `seq_merge` is inherently sequential: The traversal of two sorted arrays left-to-right and comparisons of the first elements (heads).
- The smaller element is moved to the resulting merged array and the head pointer moves rightward.
- Such a merge is usually called the **2-way merge**.
- Let us discuss its parallelization possibilities.

Parallel MergeSort: Parallel 2-way merge I

- p parallel threads is to merge 2 sorted pole of size $k = n/2$.
- Using an example, we will explain the idea of a very elegant solution.
- Assume $n = 16$, $p = 4$, and 2 sorted arrays $A = 4, 6, 7, 11, 13, 14, 15, 16$ and $B = 1, 2, 3, 5, 8, 9, 10, 12$.
- Assume that A and B consist of unique numbers for simplicity.
- We will construct a binary matrix $M[k, k]$ such that A is mapped to its row indexes and B to its column indexes.
- If $A[i] > B[j]$, then we define $M[i, j] = 0$, else $M[i, j] = 1$ (in the figure only some 0s and 1s are listed).
- Since B is sorted, in each row a 0 never follows a 1.
- So each row is a k-bit sequence 0^*1^*,
- Similarly, since A is sorted, columns are k-bit sequences 1^*0^*.
- Therefore, the boundary between 0s and 1s is a rightwards-downwards-staircase-like curve from the left upper corner to the right bottom corner (see the thick blue line).

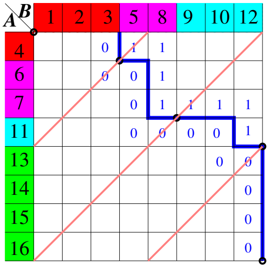


Parallel MergeSort: Parallel 2-way merge II

- In M, let us now lead $p-1$ equidistant antidiagonals in diagonal distance k/p and let us denote by $x_1, \ldots, x_{p-1}$ the intersections of antidiagonals with this boundary and x_0 and x_p the terminal points of the main diagonal.
- For $i = 1, \ldots, p$, let A_i be the part of A bounded by horizontal projection of intersections x_{i-1} and x_i let B_i be the part of B bounded by vertical projection of intersections x_{i-1} and x_i. (For example, $A_1 = \{4\}$ a $B_2 = \{5, 8\}$).
- Let $X_i = \texttt{merge}(A_i, B_i)$ for $i = 1, \ldots, p$.
- Then it holds that:

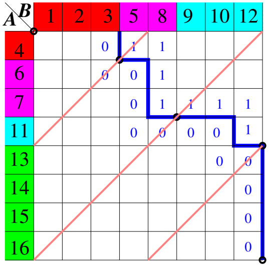
1. X_i is a sorted array of size $|X_i| = n/p$ for $i = 1, \ldots, p$.
2. For $i = 1, \ldots p-1$, the last element of X_i is less than the first element of X_{i+1}.

- Corollary: The array constructed by concatenating $X_1 \ldots X_p$ is sorted.



Parallel MergeSort: Parallel 2-way merge III

- The design of a parallel 2-way merge algorithm is therefore reduced to the question how compute intersections $x_1, \ldots, x_{p-1}$ using p threads.
- Since then the problem becomes trivial: each thread i just performs a sequential 2-way merge with A_i and B_i, produces $X_i = \texttt{merge}(A_i, B_i)$, and writes it into the preallocated disjoint part of the output array.
- It follows from the figure that thread i can independently on the others compute x_i in binary array representing the i-th antidiagonal an the unique index where the antidiagonal elements jump from 0 to 1.
- Using a classical binary search, it can make it in $O(\log n)$ time.
- Hence, $p-1$ threads simultaneously compute all $p-1$ intersections x_i, $i = 1, \ldots, p-1$, in time $O(\log n)$.



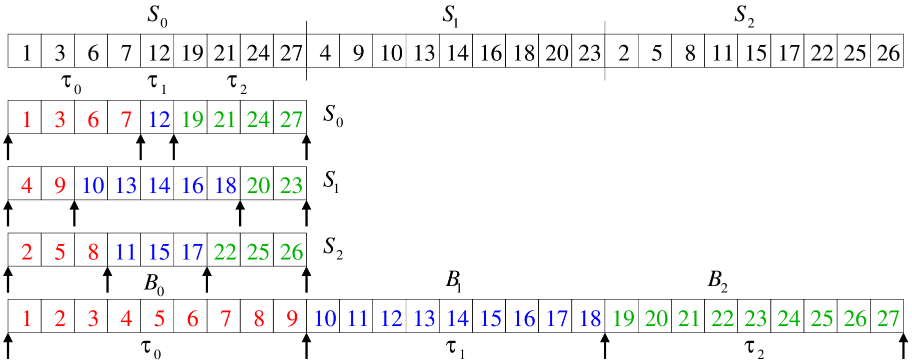
Parallel MergeSort: Parallel 2-way merge IV

- Let us denote by PM the variant with the parallel 2-way merge.
- The running time of PUV+ST+PM was 2,01 [s], which is speedup 6,91 wrt SUV. BUT it is still significantly worse than PGNU!
- 2-way merge implementation may suffer from false sharing.
- Simply, the parallel 2-way merge is still not as efficient as the parallel partitioning in QuickSort.
- The way to improve MergeSort for $p \ll n$ is to abandon the sequential 2-way merge concept and to make the merge operation p-way. Let us call this variant **Parallel Multi-Way MergeSort**, PMWMS.
- The basic idea of PMWMS: Each thread i sorts its part S_i of size n/p. Then each thread contributes to the resulting sorted array of size n so that it inserts into its respective slot of size n/p a sorted sequence B_i created by sequential p-way merge of apropriate shorter disjoint parts taken from all S_j, $j = 0, \dots, p-1$, of total size n/p.
- This is the basic component of the PGNU MergeSort implementation.

Parallel `MergeSort`: The principle of the p-way PMWMS

- p threads split the input unsorted array of size `n` into `p` parts `S[0],...,S[p-1]` of size `n/p`.
- Every thread `i` sorts sequentially its part `S[i]`.
- After a barrier, every thread `i>0` computes its vector of its `p` **splitters** `splitters_vec[i]`, which are indexes into `p` sorted sequences `S[p][n/p]` such that the sum of sizes of disjoint parts defined in `S[p][n/p]` by two neighboring `splitters_vec[i]` and `splitters_vec[i+1]`, `i=0,...,p-1`, has **the total size** `n/p`.
- After a barrier, every thread `i` takes its p disjoint sorted parts (cut-outs) from all S_j defined by its splitters of total size `n/p`.
- Then using **sequential** p**-way merge**, every thread i merges its cut-outs into its output contribution B_i of size `n/p`.

Parallel MergeSort: An example of a 3-way PMWMS



- Consider $p = 3$ threads τ_0, τ_1, τ_2 and let $n = 27$.
- Each thread sorts sequentially its $n/p = 9$ numbers.
- Then each thread (except for τ_0) computes its vector of splitters (2 inner arrows at each sorted sequence S_i).
- Then each thread i merges using a 3-way merge its 3 cut-outs (e.g., τ_0 has red numbers) into a sorted part B_i of size exactly 9.

Parallel MergeSort: p-way PMWMS I

```
int S[p][n/p], splitters_vec[p][p], my_tuple[p][n/p];
void ParMultiWayMergeSort(int* A, int n) {
  int* B = (int*)malloc(n * sizeof(int)); //auxiliary array allocation
  #pragma omp parallel
  { my_id = omp_get_thread_num();
    S[my_id] = A[my_id*n/p,..,(my_id+1)*n/p-1]; //assign A to threads
    seq-sort(S[my_id]);  //each thread sorts sequentially its part of S
    #pragma omp barrier
    splitters_vec[my_id] = Splitters_by_Rank([S,my_id*n/p);
    //each thread computes its vector of splitters_vec
    #pragma omp barrier
    my_tuple[my_id] = split(S,splitters_vec,my_id)
    //each thread gets a p-tuple of cut-outs from S[0],..,S[p-1]
    //of total size n/p based on its splitters
    B[my_id*n/p,..,(my_id+1)*n/p-1] = SeqMultiWayMerge(my_tuple[my_id],p)
    //each thread applies a p-way merge to its p-tuple of cut-outs
    //from S[0],..,S[p-1] and produces a sorted array of size n/p and
    //puts it into corresponding part B[i] of the output sorted array
} }
```

Parallel `MergeSort`: p-way PMWMS III

- The most important algorithmic component of this solution is function `Splitters_by_Rank`.
- Each thread (except 0) computes it using its rank `my_id*n/p` $=$ the index where its contribution in the output array starts.

Every thread takes the shared array of p sorted subarrays `S[0],..,S[p-1]` of sizes `n/p` and the `rank` of its first number in the output and generates array of p splitters of these subarrays such that the total number of elements to the left from the splitters summed over all `S[i]` equals exactly `rank`.

Asymptotic estimate of the parallel time:

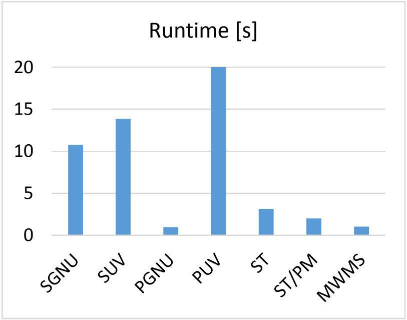
$$T(n,p) = O\left(\frac{n}{p}\log\frac{n}{p} + p\log\frac{n}{p}\log n + \frac{n}{p}\log p\right).$$

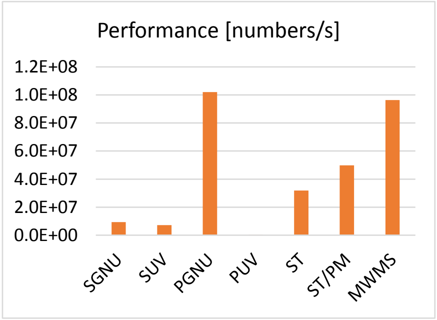
Sequential sort of n/p numbers $+$ $\log n$ executions of p instances of BinarySearch in arrays of size n/p $+$ sequential p-way merge of p cut-outs of total size n/p. For small p and big n, the 1st term will dominate.

Parallel MergeSort: p-way PMWMS IV

```
void Splitters_by_Rank(int* S, int rank) {//thread i has rank = my_id*n/p
   int L[p], R[p], m[p]; int v;
   for (i = 0; i < p; i++) {
     L[i] = 0; R[i] = n/p-1;  //each S[i] has n/p elements and these are
    //the left and right bounds within which my splitters lie
   while (there exists i such that L[i] < R[i]) {
    v = Pickup_Random_Pivot(S[0],..,S[p-1],L[0],R[0],..,L[p-1],R[p-1])
     //pick up a random  element  within the current bounds
   for (i = 0; i < p; i++)
     m[i] = binarySearch(v, S[i]); //compute rank of v in S[i]
   if (m[0] + .. + m[p-1] >= rank) //compare the global rank
     //of randomly chosen pivot with the requested global rank
     R[0],..,R[p-1] = m[0],..,m[p-1]; //vector assignment
     //to narrow the bounds for seeking a pivot from the right
   else L[0],..,L[p-1] = m[0],..,m[p-1];
   //narrow the bounds from the left
} //pivot v has the requested rank and left and right bounds match
return L[0],..,L[p-1] //vector of splitters for the local thread
```

Parallel `MergeSort`: Final performance analysis





1. ST = PUV + thresholding of new parallel task.
2. ST/PM = ST + parallel 2-way merge.
3. PMWMS = p-way `MergeSort`, sequential sort with SGNU.
4. PGNU is still a little better, since it includes other **optimizations** (see the source codes `libstdc++`).