Individual Project
Parallelized Merge Sort with OpenMP

Finn Joel Bjervig[*]

1TD062 - High Performance Programming,
Department of Information Technology,
Uppsala university, Sweden

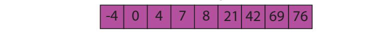
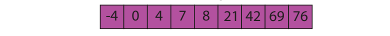
August 31, 2022

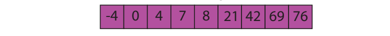
[*]Electronic address: `joelfbjervig@gmail.com`

Introduction

The Merge sort algorithm is designed with a divide and conquer paradigm: dividing the problem recursively into sub problems. For this reason it is natural to parallelize the problem using some available API. Two thread API's are presented in the course, two which are very different, namely the high level OpenMP and the low level POSIX Threads.

To introduce the merge sort algorithm, consider an array of length L containing integers ordered in a random fashion (which is really not an order, since its random). The merge sort algorithm recursively divides the array into sub-arrays until it reaches sub arrays only containing only two elements . Thereafter the two elements are compared to each other, placed in order and compared once again with the neighboring array using linear search to determine how to merge the two into an ordered array. If a sub array has three elements, the division will result in one of the arrays containing only one element. This poses no problem as it too will be compared with the neighboring arrays. The merging is also done recursively, and stops when there is no longer two sub arrays to merge, but only one final array, which should be sorted if all has gone well. In figure 1 the serial (one thread) algorithm is visualized for arrays of even and uneven lengths. The consequence is not serious, but illustrates how a single element can still be sorted in case on an uneven array length.

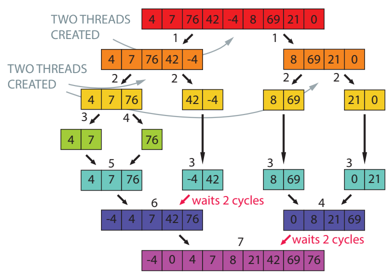





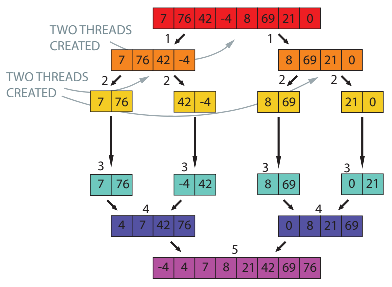
(a) Array of uneven length. (b) Array of even length.

Figure 1: Serial version of the merge sort algorithm. The values next to the arrows marks the time step (no particular unit) at which every operation is executed.

It is apparent that the computations of each sub array is independent, so it makes perfect sense to parallelize the algorithm. By assigning each new sub arrays to a thread, the computations will be parallel and simultaneously. Figure 2 illustrates how a parallelized merge sort algorithm operates. Depending on how many threads the machine has, "thread splits" can be done several times. The Mac computer used to write this report and to build the program has four threads at its disposal. This allows the program to assign new threads at two levels as shown in Figure 3 (no need to focus on other numbers than the depth in the figure. They are explained later). If the algorithm is perfectly parallelized, using four threads should be four times faster than just running in serial (one thread).



(a) Array of uneven length



(b) Array of even length

Figure 2: Merge sort algorithm with four thread.

Problem description

The problem is as follows: Implement and parallelize the sorting algorithm merge sort for an array of length N

Solution method

As mentioned, there are two avalible threading API's available in this project, namely POSIX threads (Pthreads) and OpenMP. The choice of API depends on the problem, like how much control you want to have over the threading process. Pthreads works on a low level and while it is more cumbersome to handle, it does allow for more manual control of the parallelization. On the other hand OpenMP is what one may say more user friendly on the cost that it is not as "flexible" as pthreads. Since merge sort is a fairly straight forward algorithm, the high level API OpenMp was deemed fit for parallelizing the algorithm.
But how does one write with OpenMP? See appendix for the complete code and the makefile. To begin with, we need to include the library `omp.h`. Apple Clang does not have a built-in OpenMP library. Therefore Mac users need a couple of additional flags to access the OpenMP that comes with Xcode. Assuming `usr/local` contains OpenMP, the flags needed are "`-Xpreprocessor -fopenmp -lomp`". `-Xpreprocessor` will specify system specific preprocessor options that GCC dont recognize, in this case the unrecognized option is to use `-fopenmp`, which enables for linking `-lomp`.[Sch] [GCC]

Concerning how to use the OpenMP API.

- `#Pragma` defines the block of code t be run in parallel, and if OpenMP is not supported on the machines compiler, it will simply be ignored and the program will be run in serial.

- `num_threads(N)` is used to set how many threads to create within the block.

- `omp_get_thread_num` returns the thread ID, which makes it possible to keep track of each thread such that they can all do different things in parallel.

- `omp_get_num_threads()` returns the total number of threads.

Since the parallelized merge sort divides the array in two sub arrays, we need only initialize two threads for each recursive step. Thereafter, if needed, the two sub arrays within these threads will be split and two new threads will compute them. Here we have threads initialized within another thread. This is called nested parallelism, and it is necessary to specify that such is the case in the code.

This is done by writing `omp_set_max_active_levels(l)`, where l is the max allowed depth of nesting. In the context of this algorithm, four threads would produce two levels of active threads, as seen in figure 3. Depending on how many threads the computer has available, the algorithm will be split between pairs of threads multiple times.

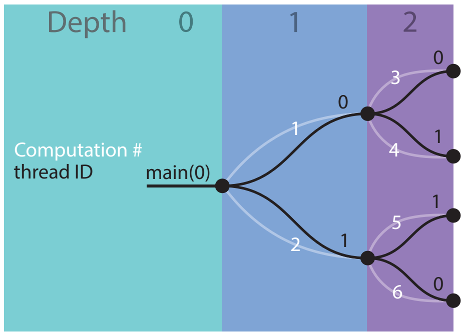


Figure 3: A visual of how threads are created. See table 2 to correlate the parents ID and the current ID

# 1	Calls made: 1	Parent thread ID: 0	Current thread ID: 0
# 2	Calls made: 1	Parent thread ID: 0	Current thread ID: 1
# 3	Calls made: 2	Parent thread ID: 0	Current thread ID: 0
# 4	Calls made: 2	Parent thread ID: 0	Current thread ID: 1
# 5	Calls made: 2	Parent thread ID: 1	Current thread ID: 1
# 6	Calls made: 2	Parent thread ID: 1	Current thread ID: 0

Table 2: Example of how threads are created and what their ID will be. See figure 3 for a visual

Figure 3 and table 2 shows how the threads will be called, and how the nested parallelism works. To grasp this schematic, look how the first row in the table 2 correlates to the white line labeled computation #1 in figure 3. Here the current thread of $ID = 0$ was made by the parent thread with $ID = 0$, (which happens to be the main thread $main(0)$). The second row corresponds to the white line labeled 2. Here we have the same parent thread ID $main(0)$ but the current thread ID is now $ID = 1$. The other four rows follow the same logic. Figure 3 also reveals why four threads corresponds to a max depth of two in the nested parallelism.

Experiments

Examining different optimizationflags, the best over all result for was obtained when the flags `-Ofast -march=native -funroll-loops}` were used. The `-funroll-loop` seem to have some effect as there are loops that can be unrolled in the code. `-march=native` does not produce any significant

Compiler flags	$N = 10^6$	$N = 10^7$
No flags	0.209 s	2.093 s
-O1	0.154 s	1.517 s
-O2	0.145 s	1.488 s
-O3	0.150 s	1.489 s
-Ofast	0.146 s	1.478 s
-Ofast -march=native	0.152 s	1.492 s
-Ofast -funroll-loops	0.148 s	1.484 s
-Ofast -march=native -funroll-loops	0.148 s	1.463 s

Table 3: Execution times for different compiler flags for $N = 10^6$ and $N = 10^7$. The execution times are the best out of 5 simulations with four threads

Going back to figure 2. If the array is divisible by two as in figure 2b, there is an even load balance such that no threads have to wait on others to reach that merging stage. However if the program is called with multiple threads and an array of *uneven* length, there will be some threads waiting. In table 5 are the runtimes for merge sort with slightly different lengths of the array. The significant difference lies in the fact that the even array only differs by ± 1 element from the uneven arrays, which is only 0.00000375% of the entire length of the even array. None the less, going from one of the uneven (Longer of shorter length) arrays to the even array decreases the the runtime by 6.5 second (on average). This corresponds to a speedup of approximately 1.138!. This seemed to good to be true, so a new test was run with a smaller array length as seen in table 4. Here the uneven array length exhibit no decrease in performance. So it may be safe to say that the uneven load balance rarely is a problem in this code and the previous test run was just some improbable analomy .

Merge sort with 4 threads

Array length N	Time (s)
266666665	53.525
266666666	47.193
266666667	53.870

Table 4: Runtimes for the merge sort algorithm, displaying the significance of choosing an uneven or an even array.

Another measure of performance is how much of the CPU's capacity the program is using. Looking at the CPU usage through the terminal command `top`, one can see how large the CPU usage is in percent for all processes running on the machine. When a program is using all N threads the CPU usage will show as $N \cdot 100\%$ (it will always be a bit lower than that). Looking at the merge

Merge sort with 4 threads	
Array length N	Time (s)
2666665	0.398356
2666666	0.406141
2666667	0.406595

Table 5: Runtimes for the merge sort algorithm, displaying the significance of choosing an uneven or an even array.

sort process in top when its handling the uneven arrays, the CPU usage jumps around between $100, 150, 200, 250, 300, 350$ and 400 percent (plus minus a few percent). This is a sign that some threads are just waiting and arenot doing any computations, which is inefficient because the full power of the CPU is not utilized. When running the merge sort with the same four threads and with an even array length as in table 5, the CPU usage more or less stays the same, in the $300 - 400$ percent regime, showing that *all* threads are working.

A key property of an algorithm is how well they scale with the problem size, which in this case is the size of the array to be sorted. In figure 4 the scaling is displayed for different number of threads, and as expected the scaling is better with increasing threads. But the most significant leap in performance is in the jump from one to two threads. An interesting measure of a codes efficiency is

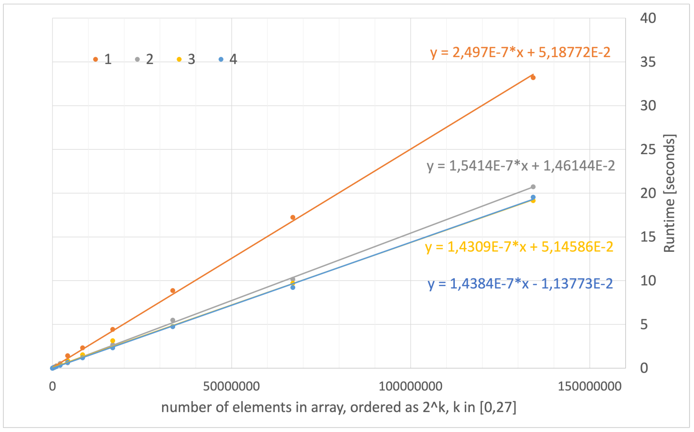


Figure 4: Measured runtime for increasing length of array, ordered as 2^k, where $k \in [0, 27]$. Each line represents the runtime utilizing a number of threads from 1 to 4

how the speedup scales with the number of threads used in the program. Figure 5 shows just this. Through this graph one may draw the conclusion that the code does not half in runtime for a doubling in number of threads. This may be due to how OpenMP handles the threads, or to the parts in the code that are not parallelized, outside of the Pragma block. The curve flattens out after four threads as expected since the computer used only has four threads. When the program is called with more than four threads, the threads simply compute in serial.

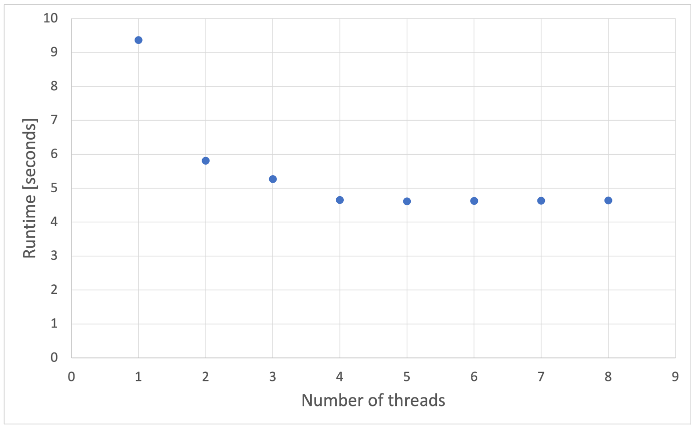


Figure 5: Measured runtime for constant array size of $N = 2^{25} = 33554432$ for increasing number of threads from 1 to 8

References

[GCC] GNU GCC. *Options Controlling the Preprocessor*. URL: `https://gcc.gnu.org/onlinedocs/gcc/Preprocessor-Options.html`. (accessed: 12.03.2021).

[Sch] Henry Schreiner. *OpenMP on High Sierra*. URL: `https://iscinumpy.gitlab.io/post/omp-on-high-sierra/`. (accessed: 12.03.2021).

Appendix

The makefile

```
CC = gcc
LD = gcc
#CFLAGS = -Xpreprocessor -fopenmp -Ofast -funroll-loops -Wall
CFLAGS = -Xpreprocessor -fopenmp -O3 -Wall
LDFLAGS= -lm -lomp
RM = /bin/rm -f
EXEC = mergeSortOmp


mergeSortOmp: mergeSortOmp.c
	$(LD) $(CFLAGS) -o $(EXEC) mergeSortOmp.c $(LDFLAGS)
clean:
	$(RM) $(EXEC)
```

The Program galsim.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <omp.h>

void merge_sort(int* array_to_sort, int N, int nThreads, int nCalls
    ) {
  if(N == 1) {
    // Only one element, no sorting needed. Just return directly in
     this case.
    return;
  }

  //int threadID;

  // given N is divisible by two
  int n1 = N / 2;
  int n2 = N - n1;

  // Allocate new lists
  int *buffer  = (int*)malloc((n1+n2)*sizeof(int));

  // partition buffer into two other lists
  int *list1 = buffer;
  int *list2 = buffer + n1;


  // load list to sort into the two buffer lists
  int i;
  for(i = 0; i < n1; i++)
    list1[i] = array_to_sort[i];
  for(i = 0; i < n2; i++)
    list2[i] = array_to_sort[n1+i];

  // Sort list1 and list2
  /*
    Divide number of threads by two each time, until it reaches one
    , then its serial computing.
  */
  // creates two threads recursively, and divides nThreads by two,
    until nThreads < 1
  // if nThreads = 4, then two threads will in turn create two
    other threads. In total six threads
  // when nThreads<1, then merge operates on that single thread
  int threadID = omp_get_thread_num();
  if (nThreads>1){
  #pragma omp parallel num_threads(2)
    {
      printf("calls made: %d \t | Parent thread ID: %d \t | Current
     thread ID: %d \n", nCalls, threadID, omp_get_thread_num());
      if(omp_get_thread_num()==0){
        merge_sort(list1, n1, nThreads/2,nCalls+1);
      }
      if (omp_get_thread_num()==1 )
      {
        merge_sort(list2, n2, nThreads/2,nCalls+1);
      }
    }
  }
  else{ // if nthreads<=1, then stop the division of threads and
    finish the mergsort
        merge_sort(list1, n1, nThreads,nCalls);
        merge_sort(list2, n2, nThreads,nCalls);
  }

  // sort and merging
  int i1 = 0;
  int i2 = 0;

  i = 0;
  while(i1 < n1 && i2 < n2) { // while we're not out of bounds
    if(list1[i1] < list2[i2]) { // sort
      array_to_sort[i] = list1[i1];  //store
      i1++;
    }
    else {
      array_to_sort[i] = list2[i2];
      i2++;
    }
    i++;
  }
  while(i1 < n1)
    array_to_sort[i++] = list1[i1++];
  while(i2 < n2)
    array_to_sort[i++] = list2[i2++];
  free(buffer);
  //free(list2);
}

static double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
```

```c
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
}

static int count_values(const int* list, int n, int x) {
  int count = 0;
  int i;
  for(i = 0; i < n; i++) {
    if(list[i] == x)
      count++;
  }
  return count;
}

int main(int argc, char* argv[]) {
  if(argc != 3) {
    printf("Please give 1 argument: N (number of elements to sort)
    and number of threads.\n");
    return -1;
  }

  // length of array
  int N = atoi(argv[1]);
  //printf("Length of array to be sorted is N = %d\n", N);

  if(N < 1) {
    printf("Error: N < 1 \n");
    return -1;
  }

  int* array_to_sort = (int*)malloc(N*sizeof(int));

  // Fill array with random numbers
  int i;
  for(i = 0; i < N; i++)
    array_to_sort[i] = rand() % 100;

  /*// Count how many times the number 7 exists in the list.
  int count7 = count_values(array_to_sort, N, 7);
  printf("Before sort: the number 7 occurs %d times in the list.\n
    ", count7);
  */

  // Enable nested parallelism
  omp_set_max_active_levels(3); // wat dis tho
  int nThreads = atoi(argv[2]);


  // Sort list
  double time1 = get_wall_seconds();
  {merge_sort(array_to_sort, N, nThreads,1);}
  //printf("Sorting list with length %d took %7.3f wall seconds.\n
    ", N, get_wall_seconds()-time1);
  printf("%7.6f\n", get_wall_seconds()-time1);

  //int count7_again = count_values(array_to_sort, N, 7);
  //printf("After sort : the number 7 occurs %d times in the list.\
    n", count7_again);

  // Check that list is really sorted
  for(i = 0; i < N-1; i++) {
    if(array_to_sort[i] > array_to_sort[i+1]) {
      printf("Error! List not sorted!\n");
      return -1;
    }
  }
  //printf("OK, list is sorted!\n");

  free(array_to_sort);

  return 0;
}
```