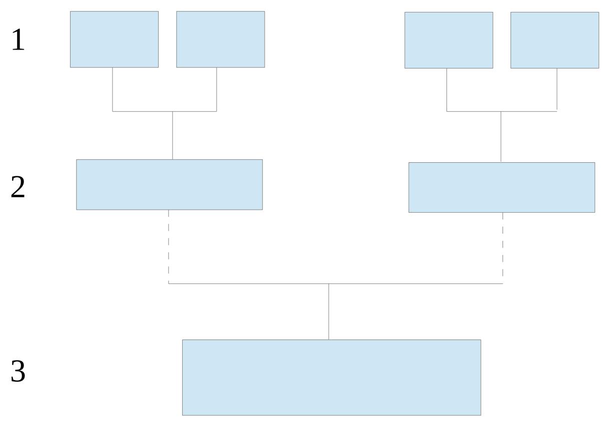
Simple Parallel Sort: pthread vs. OpenMP

There are many ways to implement a parallel sort. For this comparison we chose a POSIX pthreads implementation we call pthread sort. In order to compare performance an OpenMP version was implemented as well. We call this version OpenMP sort. Code for both is included. As a baseline version we use the sequential sorting algorithm `algorithm::sort.`

Parallel sort design.

The list of numbers to be sorted is broken up into as many chunks as number of threads used in a given run. If the list is not divisible by the number of threads, the remainder of the division is assigned to the last thread. Each chunk is then sorted by `algorithm::sort`. Once the sorting is finished we use `std::inplace_merge` to merge the lists. This is most likely the easiest way to produce a parallel sort. We chose it precisely for its simplicity. Both sorting and merging happens in parallel if the number of lists remaining is greater than one. For added simplicity, the number of threads is chosen as power of two, as illustrated in Figure 1.



1. The list is broken up into number_of_threads chunks. Each chunk is then sorted
2. Every pair of lists is merged into one list
3. Merging continues until only one sorted list is left.

Fig 1. Illustration of the parallel sort and merge algorithm

Results

The hardware used in this experiment is an AMD A10-6800K 4-Core CPU running at 4.1Ghz with 16GB DDR3 RAM at 1866 Mhz. All results are based on averages of five runs. The actual time to sort the lists using pthread sort is presented in Figure 2. This is presented for comparison purposes, in case a reader wants to compare their hardware to ours. As expected the higher the parallelism the better the result, i.e. higher parallelism resulted in shorter sort time.

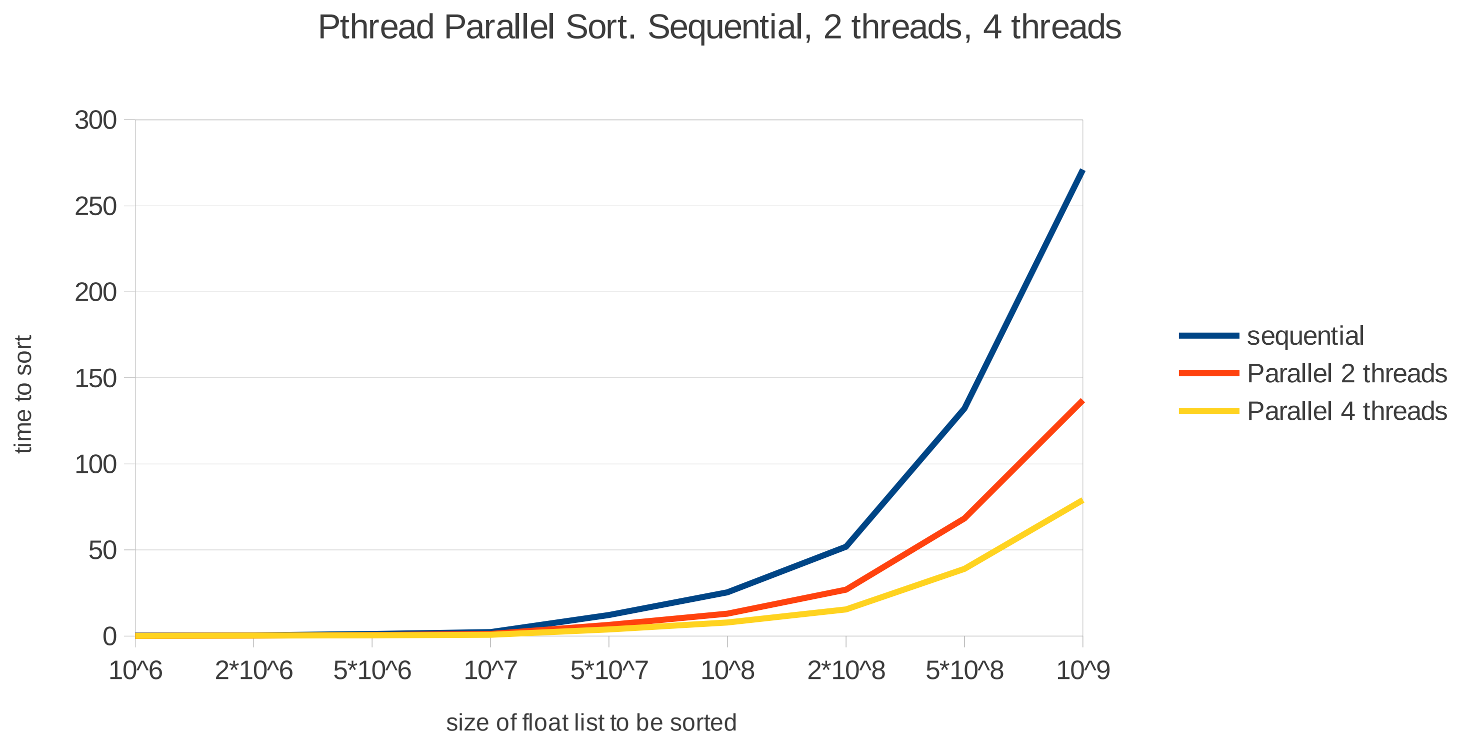


Fig 2. Time (sec.) to sort a float list of given size, sequentially, 2 threads, 4 threads. Smaller is better

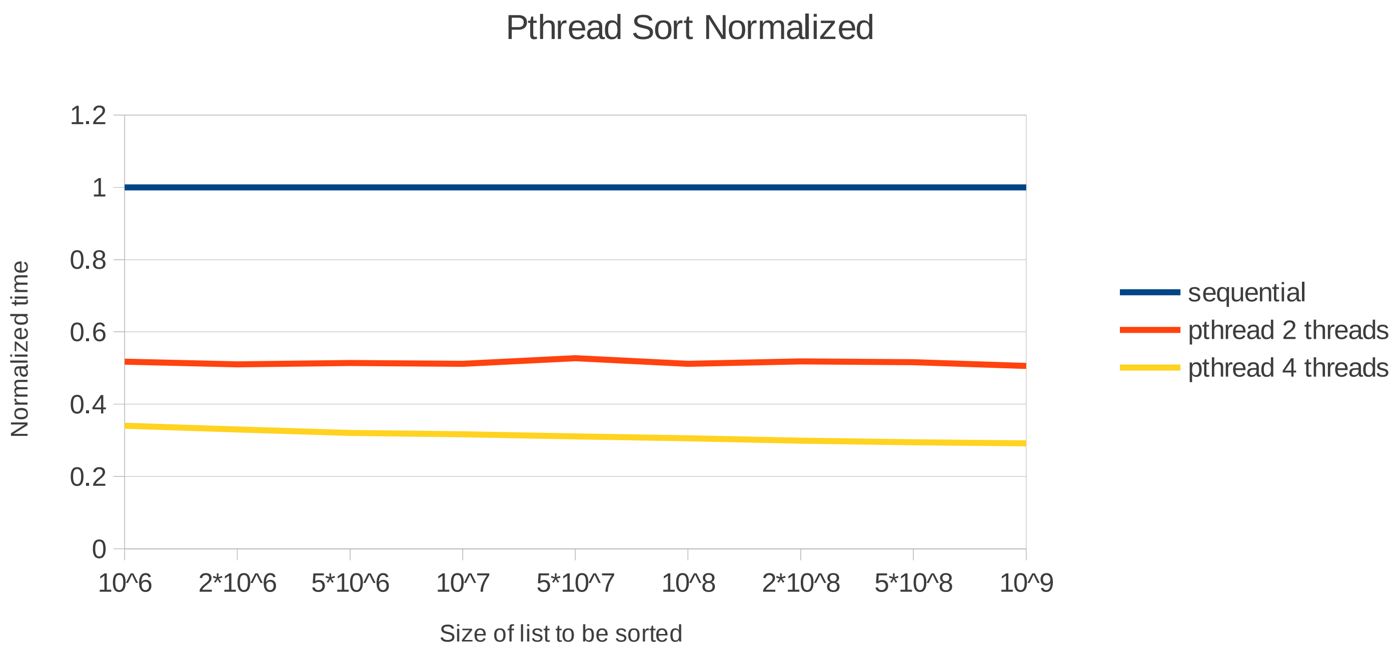


Fig 3. Normalized time to sort a float list of given size, sequentially, 2 threads, 4 threads. Smaller is better

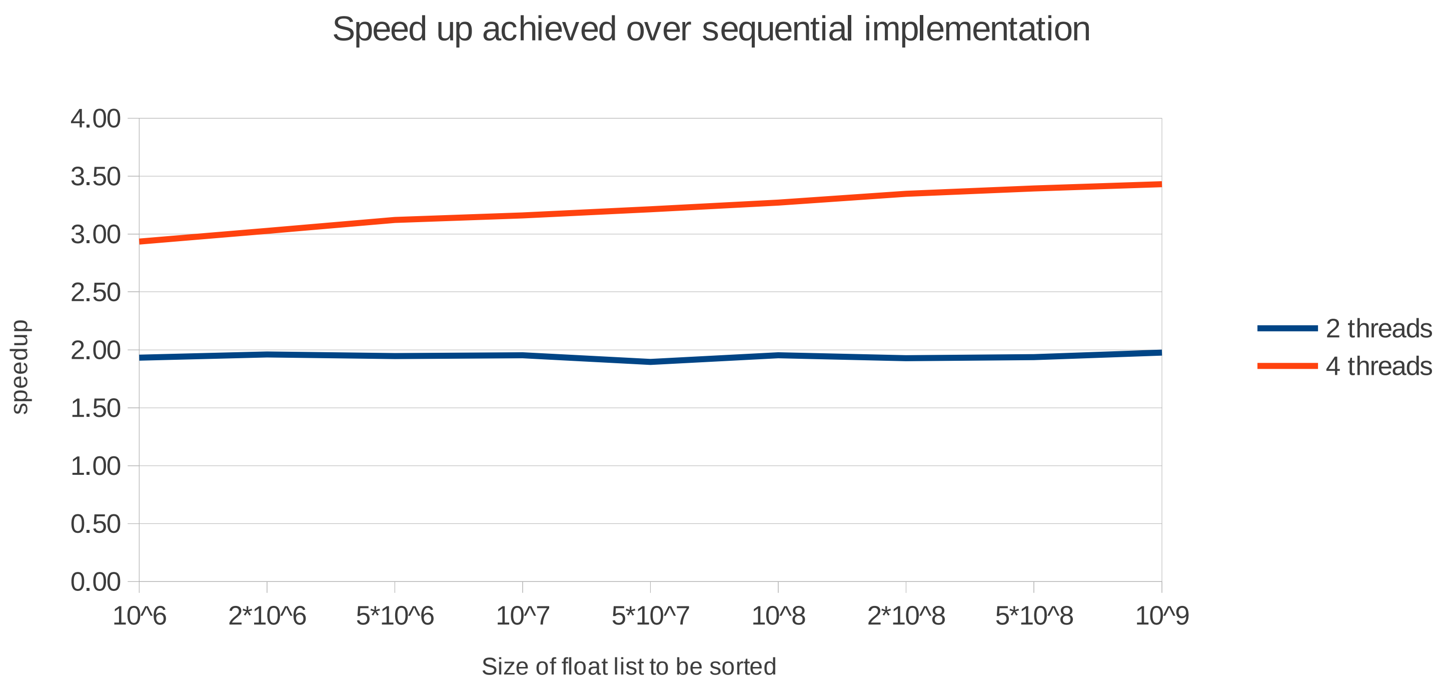


Fig 4. Pthread parallel sort speedup achieved over sequential sort. Higher is better.

OpenMP

The absolute time to sort the lists using our OpenMP sort is presented in Figure 5. Figure 6 presents the time normalized with respect to sequential (baseline) sort. Figure 7 presents the speedup. All results are as expected. The higher the parallelism the better the time. Maximum speedup achieved of 3.59 times was in the case of the largest list, 1 billion floating point numbers.

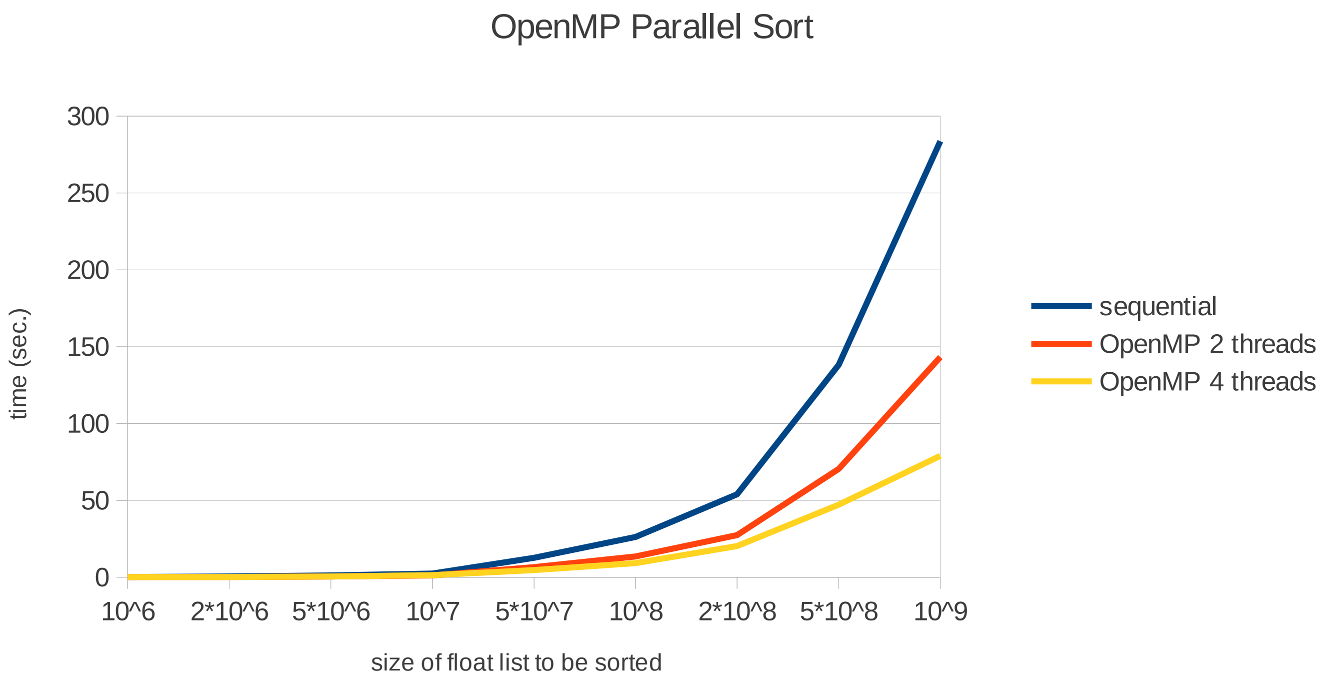


Fig. 5 Absolute time to sort a list using OpenMP sort. Smaller is better

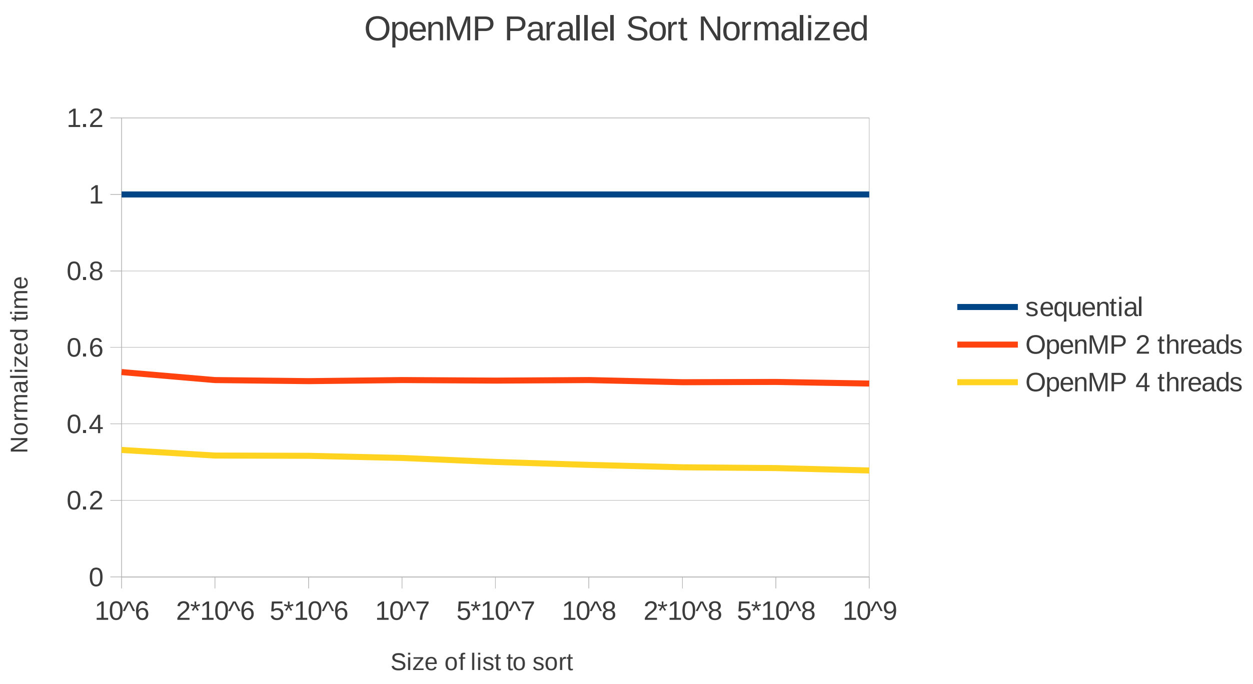


Fig. 6 Normalized time to sort a list using OpenMP sort. Smaller is better

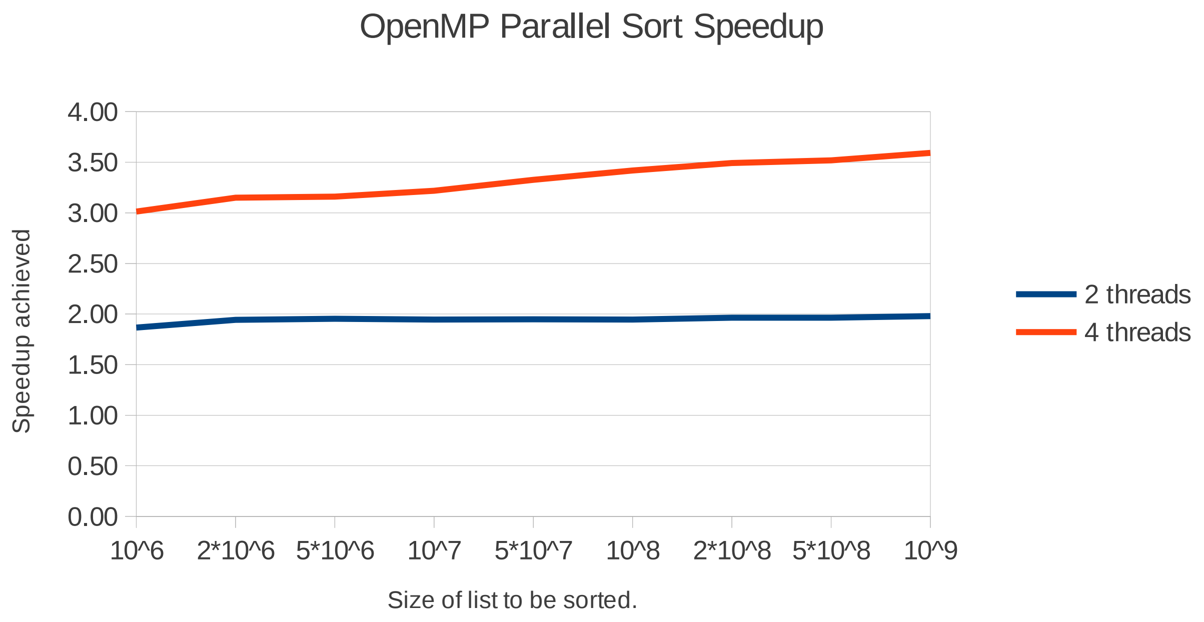


Fig. 7 OpenMP sort speedup achieved over sequential sort. Higher is better.

OpenMP vs. pthread

Figure 8 presents a comparison of speedups achieved using pthread and OpenMP sorts. OpenMP sort achieves a higher speedup with less coding complexity. The OpenMp implementation took roughly 100 lines of code versus the pthread implementation's roughly 200 lines of code.

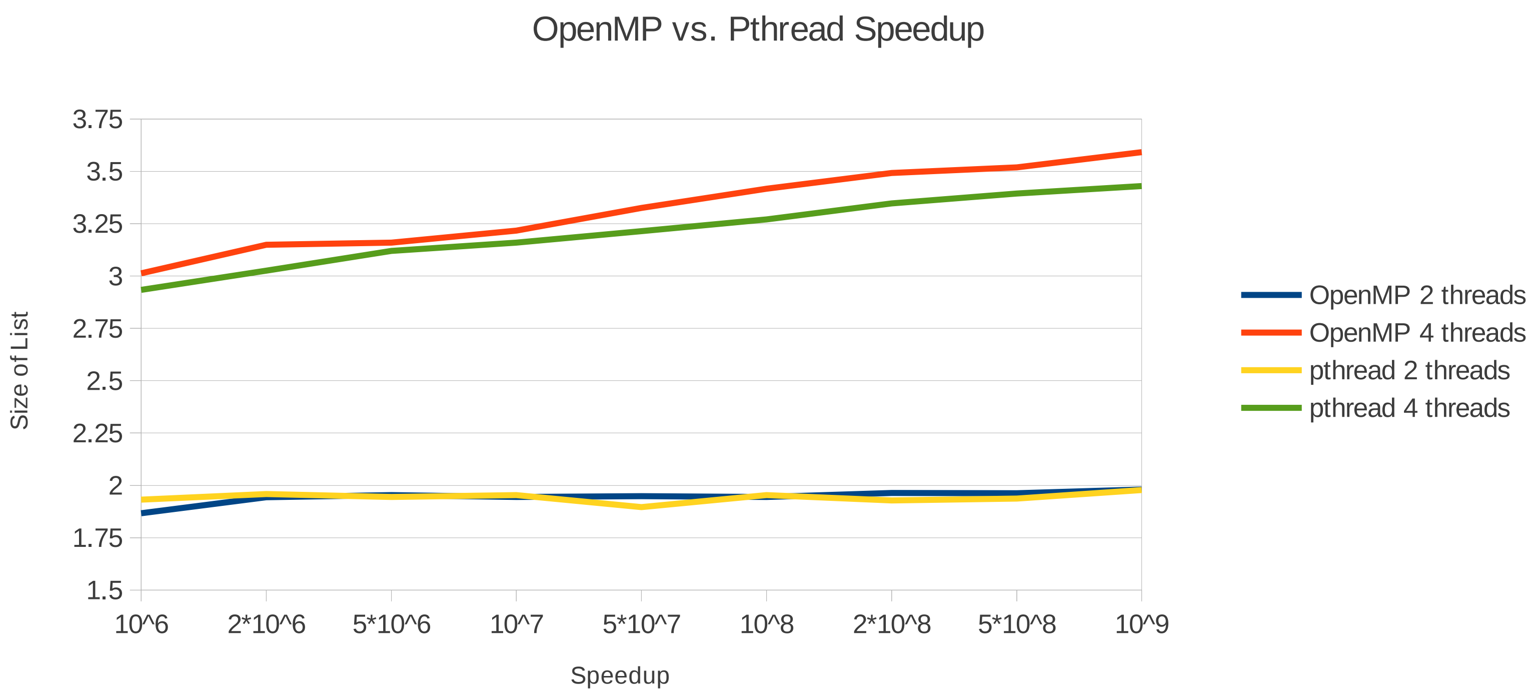


Fig. 8 OpenMP sort vs. pthread sort speedup comparison. Higher is better.

Conclusion

We have compared a simple implementation of parallel sorting using two different parallel libraries: POSIX threads (pthread) and OpenMP. In this experiment the OpenMP implementation was both simpler in terms of code complexity and it offered higher measured performance.

Code

Pthread sort:

```
/*
 Pthread sort
 Albert Szmigielski
       aszmigie@sfu.ca
 to compile: g++ -g -lrt -D_REENTRANT par_pthreads_sort.cpp -o par_pthreads_sort
-fopenmp -pthread
 usage: par_pthreads_sort <number of threads> <size>
 <number of threads> must be power of two


*/
#include <algorithm>
#include <math.h>
#include <iostream>
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <sys/time.h>
#include <omp.h>

using namespace std;


int Num_Threads=4;  //default is 4 can be changed via argument
int size =1000000;  //default is 1M can be changed via argument
float  * Array, * Array1;
```

```
int interval = floor(size/Num_Threads);

// errexit *******************************************************
void
errexit (const char *err_str)
{
    fprintf (stderr, "%s", err_str);
    exit (1);
}
// BARRIER *******************************************************
// from cmpt431
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
void
barrier (int expect)
{
    static int arrived = 0;

    pthread_mutex_lock (&mut);   //lock

    arrived++;
    if (arrived < expect)
        pthread_cond_wait (&cond, &mut);
    else {
        arrived = 0;             // reset the barrier before broadcast is important
        pthread_cond_broadcast (&cond);
    }

    pthread_mutex_unlock (&mut); //unlock
}

// END OF BARRIER

/***************************    PAR MERGE    ******************************/
void par_merge(int thread_id, int num_lists, int mul, int interval){
             //merge until 1 list left
             //merge list 1&2, 3&4, ... (n-1 & n)

       mul = mul*2;
       int par_merge_threads = num_lists/2;
       if (num_lists > 1){
             if (thread_id < par_merge_threads)
                    {
                          int start = interval * thread_id *mul;
                          int end = interval * (thread_id+1) *mul ;
                          int middle = floor(start+ (end - start)/2);
                          if (thread_id == par_merge_threads-1)
                                 { end=size; }
                          inplace_merge(&Array[start], &Array[middle], &Array[end]);

                    }
                    num_lists = ceil(num_lists/2);
                    barrier(Num_Threads);
                    par_merge (thread_id, num_lists, mul, interval);
       }
}
/***************************    PTHREAD SORT    ******************************/
void * pthreadSort(void * id){
       //cout << "in pthreadsort \n";
       int thread_id= *((long *) id);
       int start = interval * thread_id;
       int end = interval * (thread_id+1) ;
       //cout << "id: " <<  thread_id << " start: " << start << " end: "<< end <<"\n";
       if (thread_id == Num_Threads-1)
             {end=size;}
       sort (&Array[start], &Array[end]);
       //cout << "Thread " << thread_id << " par sort finished \n";
       // BARRIER
       barrier(Num_Threads);
       //Now merge
       par_merge(thread_id, Num_Threads, 1, interval);
```

```
}

void * pthreadSortTEST(void * id){
       int thread_id= *((long *) id);
       cout << " Hello from " << thread_id << endl;
}
/*************************    PAR SORT    ******************************/
void par_sort() {
       interval = floor(size/Num_Threads);
       // break into num_threads chunks, sort each chunk
       pthread_attr_t attr;
    pthread_t *tid;
       int *id;
       long i;

       // create threads
       id = (int *) malloc (sizeof (int) * Num_Threads);

       tid = (pthread_t *) malloc (sizeof (pthread_t) * Num_Threads);
       if (!id || !tid)
           errexit ("out of shared memory");

       pthread_attr_init (&attr);
       pthread_attr_setscope (&attr, PTHREAD_SCOPE_SYSTEM);
       //cout << "about to create threads \n";

       for (i = 1; i < Num_Threads; i++) {
           id[i] = i;
           pthread_create (&tid[i], &attr, pthreadSort,  &id[i]);
       }

       id[0]=0;
             pthreadSort(&id[0]);

       // wait for all threads to finish
       for (i = 1; i < Num_Threads; i++)
           pthread_join (tid[i], NULL);

}

/***************************   MAIN   *********************************/
int main (int argc, char** argv) {
       double start_s, stop_s, start_p, stop_p;
       double time_s, time_p;
       int range = 100, error=0;


       if (argc>1)  Num_Threads= atoi(argv[1]);
       if (argc>2)  size = atoi(argv[2]);

       //cout << "size: " << size << endl;
       //cout << "num_threads: " << Num_Threads << endl;

       Array = new float [size];
       Array1 = new float [size];

       for (int i=0; i<size; i++){
             Array[i]  = (rand()/(RAND_MAX +1.0));
             Array1[i]  = Array[i];
       }

       //time sequential sort
       //gettimeofday(&start_s, 0);
       start_s=omp_get_wtime();
             sort(&Array1[0], &Array1[size]);
       stop_s=omp_get_wtime();
       //gettimeofday(&end_s, 0);
       //cout << "seq. sort done \n";

       //time parallel sort
       start_p=omp_get_wtime();
```

```
		par_sort();
	stop_p=omp_get_wtime();
	//gettimeofday(&end_p, 0);

	//check correctness
	for (int i=0; i<size;i++){
		if (Array[i] != Array1[i]){
			//cout << " Error " << "position "<< i ;
			error = 1;
		}
	}

	if (error) cout << "ERROR \n ";
	else {
		cout << "OK \n";
		cout << "Sorted " << size << " numbers with seq. sort in "<< stop_s-start_s
<< " seconds\n" ;
		cout << "Sorted " << size << " numbers with par. sort in "<< stop_p-start_p
<< " seconds\n" ;
		//int print_once=1;
		//if(print_once) cout << "size, sequential, parallel\n";
		//cout << Num_Threads << ", " << size << ", " << (stop_s-start_s) << ", "
<< (stop_p-start_p) <<"\n";
		}

	delete (Array);
	delete (Array1);

	return 0;
}
```

OpenMP sort:

```
/*
 OpenMP sort
 Albert Szmigielski
	aszmigie@sfu.ca
 to compile: g++ -g  par_omp_sort.cpp -o par_omp_sort -fopenmp
 usage: ./par_omp_sort <number of threads> <size of list>
 **<number of threads> must be power of two


*/
#include <algorithm>
#include <math.h>
#include <iostream>
#include <omp.h>

using namespace std;


int Num_Threads=4;  //default is 4 can be changed via argument
int size =1000000;  //default is 1M can be changed via argument

/***************************    PAR MERGE    ***************************/
void par_merge(float * Array, int num_lists, int mul, int interval){
		//merge until 1 list left
		//merge list 1&2, 3&4, ... (n-1 & n)

	mul = mul*2;
	int par_merge_threads = num_lists/2;
	if (num_lists > 1){
		#pragma omp parallel num_threads(par_merge_threads)
		{
			int start = interval * omp_get_thread_num() *mul;
			int end = interval * (omp_get_thread_num()+1) *mul ;
			int middle = floor(start+ (end - start)/2);
			if (omp_get_thread_num() == par_merge_threads-1)
```

```
                        { end=size; }
                    inplace_merge(&Array[start], &Array[middle], &Array[end]);

                }
                num_lists = ceil(num_lists/2);
                par_merge (Array, num_lists, mul, interval);
        }
}

/***************************    PAR SORT    ***************************/
void par_sort (float * Array) {
        // break into num_threads chunks, sort each chunk
        int interval = floor(size/Num_Threads);
        #pragma omp parallel num_threads(Num_Threads)
        {
                int start = interval * omp_get_thread_num();
                int end = interval * (omp_get_thread_num()+1) ;
                if (omp_get_thread_num() == Num_Threads-1)
                        {end=size;}
                sort (&Array[start], &Array[end]);
        }
        par_merge(Array, Num_Threads, 1, interval);
}


/*****************************   MAIN   *******************************/
int main (int argc, char** argv) {
double start_s, start_p, end_s, end_p;
float  * Array, * Array1;

if (argc>1)  Num_Threads= atoi(argv[1]);
if (argc>2)  size = atoi(argv[2]);


int range = 100, error=0;
Array = new float [size];
Array1 = new float [size];

#pragma omp parallel for
        for (int i=0; i<size; i++){
                Array[i]  = (rand()/(RAND_MAX +1.0));
                Array1[i]  = Array[i];
        }


//time sequential sort
start_s =  omp_get_wtime( );
        sort(&Array1[0], &Array1[size]);
end_s =  omp_get_wtime( );

//time parallel sort
start_p =  omp_get_wtime( );
        par_sort(Array);
end_p =  omp_get_wtime( );

//check correctness
#pragma omp parallel for
        for (int i=0; i<size;i++){
                if (Array[i] != Array1[i]){
                        cout << " Error " << "position "<< i ;
                        error = 1;
                }
        }


if (error) cout << "ERROR \n ";
else cout << "OK \n";
cout << "Sorted " << size << " numbers with seq. sort in "<< end_s - start_s << "
seconds\n" ;
cout << "Sorted " << size << " numbers with par. sort in "<< end_p - start_p << "
seconds\n" ;
```

```
//cout << Num_Threads << ", " << size << ", " << (end_s - start_s) << ", " << (end_p -
start_p) <<"\n";
return 0;
}
```

Further Reading:

Proceedings of the 6th European Workshop on OpenMP

Süß, Michael, and Claudia Leopold. "A user’s experience with parallel sorting and openmp." In *Proceedings of Sixth European Workshop on OpenMP-EWOMP*, pp. 23-28. 2004.