Programming of Parallel Computers

Assignment 2. PThreads.

Christos Sakalis
Christos.Sakalis.3822@student.uu.se

Aliaksandr Ivanou
Aliaksandr.Ivanou.1364@student.uu.se

February 23, 2014

1 Introduction

In this report we present the results that we got by running pthread implementations.

2 Results

There were several experiments. The figure 2 show the dependence of execution time on the amount of threads. The size of input data is 100000000. The figure 1 shows the dependence of the size. The number of threads is 20.

As we can see the builtin sort produces the expected output. The time almost the same for different number of cores, also there is linear dependence on size.

The Devide and Conquer algorithm showed quite good results. Until approximately 10 threads, this algorithm shows good scalability. But after it has almost the same execution time. The DNC algorithm has a tree structure. At each iteration current thread creates two new threads with the smaller arrays and waits their execution. With a large input array and a large number of threads we will have a lot of threads that are just waiting and doing nothing. This can be the obstacle for DNC algorithm.

The Peer sorting algorithm behaved strangely. As can be seen, it scales well until 12 threads, but then the execution time raises. Each iteration there only halve of threads running. It can be one of performance obstacles. This algorithm is similar to odd-even sort. Here, we have the same problem: load imbalance. But, using pthreads we are eliminating unnecessary communication and data exchange.

The parallel quicksort has a best performance. We can see, that the execution time decreases with the amount of threads.

Also, during tests there were other programms running and possibly could affect results.

3 The code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>
```

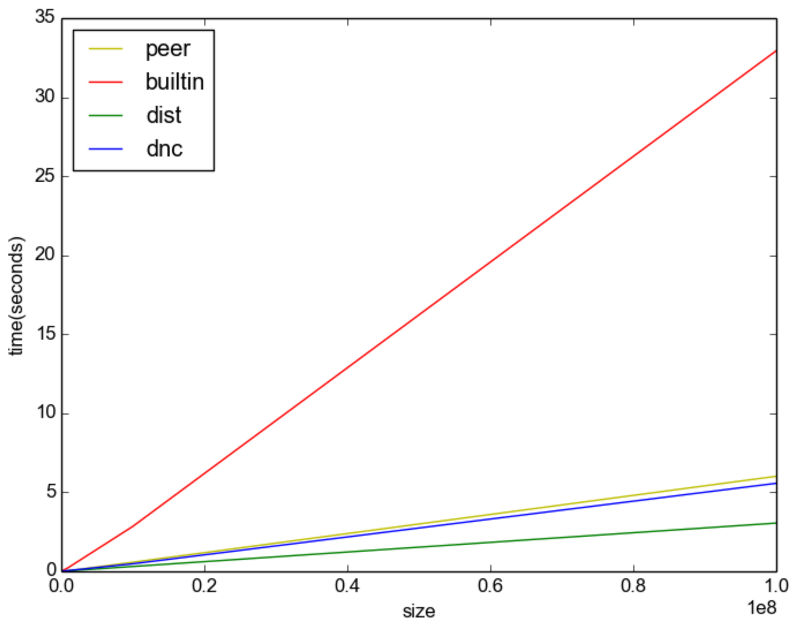

Figure 1: Size - time dependence

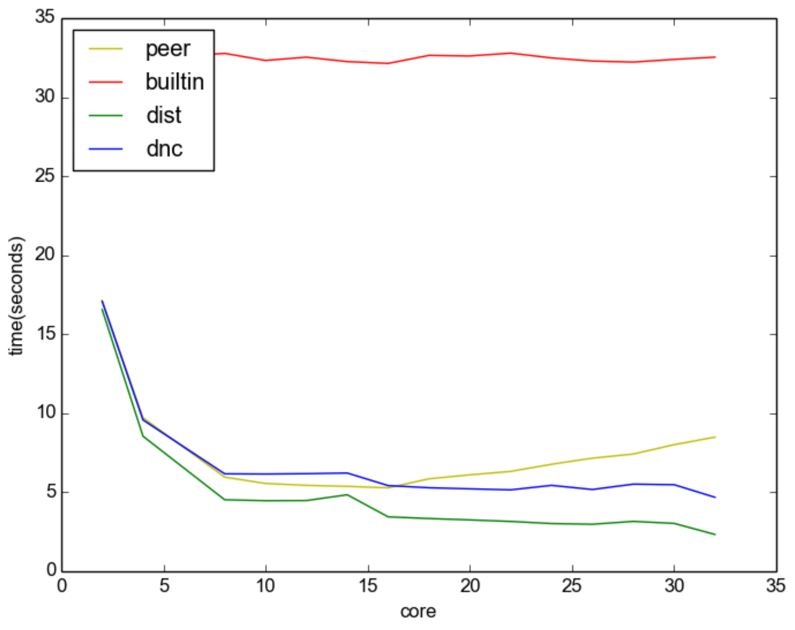

Figure 2: Core - time dependence

```
#include <sys/time.h>
#include <time.h>

#include <unistd.h>
#include <pthread.h>

#define BUFFER_SIZE 32

// Exchange the values of two variables
void swap(double * a, double * b);
// Partition the elements of the array in-place
size_t partition(double * array, size_t size, size_t pivot_inx);
// Divide and conquer quicksort
void * qsort_dnc_pth(void * arg);
void qsort_dnc(double * array, size_t size, int threads);
// Peer odd-even quicksort
void * qsort_peer_pth(void * arg);
void qsort_peer(double * array, size_t size, int threads);
// Distributed quicksort
void * qsort_dist_pth(void * arg);
void * qsort_dist_init(void * arg);
void qsort_dist(double * array, size_t size, int threads);
// Fill an array with random numbers
void randomize(double * array, size_t size, unsigned int seed);
// Check if the input array is sorted correctly
int verify(double * array, size_t size, unsigned int seed);
// Compare two doubles
int double_cmpr(const void * a, const void * b);
// Print usage
void print_usage(const char * progname);
// Find the position of the element closest to the average element
size_t find_avg(const double * array, size_t size);
// The buildin qsort implementation
void qsort_builtin(double * array, size_t size, int threads);
// Math functions
int is_power2(int num);
int is_even(int num);
// Merge two sorted arrays into the third one
void merge(double * s1, size_t size1, double * s2, size_t size2, double * dest);
// Split a array into two
void split(double * src, double * d1, size_t size1, double * d2, size_t size2);
// Find the largest element in the sorted array that is less than the bound
size_t lower_bound(const double * arr, size_t size, double e);

// Arguments for the threaded sorts
typedef struct qdnc_args qdnc_args;
typedef struct qpeer_args qpeer_args;
typedef struct qdist_args qdist_args;
```

```
int main(int argc, char *argv[])
{
        size_t size;
        int v;
        int nthreads;
        char fun[BUFFER_SIZE];
        void (*sort)(double *, size_t, int);
        double * array;
        int c;

        // Read the command line arguments
        size = 1000;
        v = 0;
        nthreads = 1;
        // Test size and verification
        while ( (c = getopt(argc, argv, "s:vt:")) != -1) {
                if (c == 's')
                        size = atoll(optarg);
                else if (c == 'v')
                        v += 1;
                else if (c == 't')
                        nthreads = atoi(optarg);
                else {
                        print_usage(argv[0]);
                        return 1;
                }
        }
        // Sorting algorithm to use
        if (optind < argc) {
                strncpy(fun, argv[optind], BUFFER_SIZE);
        } else {
                print_usage(argv[0]);
                return 1;
        }
        if (strncmp(fun, "dnc", BUFFER_SIZE) == 0) {
                sort = qsort_dnc;
                // We will only use a floor(log2(nthreads))
                if (!is_power2(nthreads))
                        fprintf(stderr, "WARNING: '-t' should be set to a power
        } else if (strncmp(fun, "builtin", BUFFER_SIZE) == 0) {
                sort = qsort_builtin;
        } else if (strncmp(fun, "peer", BUFFER_SIZE) == 0) {
                sort = qsort_peer;
        } else if (strncmp(fun, "dist", BUFFER_SIZE) == 0) {
                sort = qsort_dist;
                // We will only use a floor(log2(nthreads))
                if (!is_power2(nthreads)) {
                        fprintf(stderr, "WARNING: '-t' should be set to a power
                        nthreads = 1 << (int)floor(log2((double)nthreads));
                }
        } else {
                print_usage(argv[0]);
                return 1;
        }

        // Initialize
        array = (double *) malloc(sizeof(double)*size);
        randomize(array, size, 43);

        int ttime=timer();
        // Sort
        sort(array, size, nthreads);
        ttime=timer()-ttime;
        printf("took time: %f\n",ttime/1000000.0);
        // Verify the results
        if (v) {
                if (verify(array, size, 43))
                        printf("Verified\n");
                else
                        printf("FAILED!!\n");
        } else {
                printf("Done\n");
        }

        free(array);
        return 0;
}

// Exchange the values of two variables
void swap(double * a, double * b)
{
        double tmp;

        tmp = *a;
        *a = *b;
        *b = tmp;
}

// Partition the elements of the array in-place
size_t partition(double * array, size_t size, size_t pivot_inx)
{
        double pivot;
        size_t index, i;
        // Get the pivot
        pivot = array[pivot_inx];
        // Move the pivot to the end
        swap(&array[pivot_inx], &array[size -1]);
        // Move all the elements that are bigger smaller than the pivot
        index = 0;
        for (i = 0; i < size-1; ++i) {
                if (array[i] <= pivot) {
                        swap(&array[i], &array[index]);
                        ++index;
                }
        }
        // Move the pivot to it's final place
        swap(&array[index], &array[size -1]);

        return index;
}

// Arguments for the divide and conquer sort
struct qdnc_args
{
        double * array;
        size_t size;
        int threads;
};

// Divide and conquer quicksort
void * qsort_dnc_pth(void * arg)
{
        size_t pivot_inx;
        qdnc_args * qargs = (qdnc_args *) arg;

        double * array = qargs->array;
        size_t size = qargs->size;
        int threads = qargs->threads;

        // Are we done?
        if (size <= 1)
                return NULL;

        qdnc_args args[2];
        // Partition the elements
        // We will only use the average to improve the load balancing of the thr
        if (threads > 0)
                pivot_inx = find_avg(array, size);
        else
                pivot_inx = 0;
        pivot_inx = partition(array, size, pivot_inx);
        // Set the arguments for each recursive sort
        args[0].array = array;
        args[0].size = pivot_inx;
        args[1].array = &array[pivot_inx+1];
        args[1].size = size-pivot_inx-1;
        // Do we want more threads?
        if (threads > 1) {
                pthread_t thread_handle[2];
                args[0].threads = threads / 2;
                args[1].threads = threads / 2;

                // Start the threads
                pthread_create(&thread_handle[0], NULL, qsort_dnc_pth, (void*)&a
                pthread_create(&thread_handle[1], NULL, qsort_dnc_pth, (void*)&a

                // Wait for them to finish
                pthread_join(thread_handle[0], NULL);
                pthread_join(thread_handle[1], NULL);
        } else {
                // We will use the builtin qsort, no need for all the extra over
                qsort_builtin(args[0].array, args[0].size, 0);
                qsort_builtin(args[1].array, args[1].size, 0);
        }

        return NULL;
}

void qsort_dnc(double * array, size_t size, int threads)
{
        qdnc_args args;

        args.array = array;
        args.size = size;
        args.threads = threads;

        qsort_dnc_pth((void *)&args);
}

// Fill an array with random numbers
void randomize(double * array, size_t size, unsigned int seed)
{
        size_t i;

        srand48(seed);
        for (i = 0; i < size; ++i)
                array[i] = drand48() * size * 8.0;
}
```

```
// Check if the input array is sorted correctly
int verify(double * array, size_t size, unsigned int seed)
{
        size_t i;
        double * test_array = (double *) malloc(sizeof(double) * size);

        // Generate the same random array
        randomize(test_array, size, seed);
        // Sort using the qsort from stdlib
        qsort_builtin(test_array, size, 0);

        for (i = 0; i < size; ++i) {
                if (abs(test_array[i] - array[i]) > 0.1) {
                        return 0;
                }
        }

        free(test_array);
        return 1;
}

// Compare two doubles
int double_cmpr(const void * a, const void * b)
{
        double av = *(double *)a;
        double bv = *(double *)b;

        if (av < bv)
                return -1;
        else if (av > bv)
                return 1;
        else
                return 0;
}

// Print usage
void print_usage(const char * progname)
{
        fprintf(stderr, "Usage: %s [-svt] FUNCTION\n\n", progname);
        fprintf(stderr, "  -sNUM\t- test size\n");
        fprintf(stderr, "  -tNUM\t- number of threads\n");
        fprintf(stderr, "  -v\t- verify the result (also verbose output)\n\n");
        fprintf(stderr, "FUNCTION is one of\n");
        fprintf(stderr, "  builtin - qsort from stdlib\n");
        fprintf(stderr, "  dnc     - divide and conquer algorithm\n");
        fprintf(stderr, "  peer    - odd-even peer algorithm\n");
        fprintf(stderr, "  dist    - distributed memory algorithm\n");
}
```

```
// Find the position of the element closest to the average element
size_t find_avg(const double * array, size_t size)
{
        size_t i;
        double sum = 0.0;
        double avg;
        size_t inx;
        double diff;

        // Calculate the average
        for (i = 0; i < size; ++i)
                sum += array[i];
        avg = sum / size;

        // Find the element closest to the average
        diff = abs(avg);
        inx = size;
        for (i = 0; i < size; ++i) {
                if (abs(array[i]-avg) <= diff) {
                        diff = abs(array[i]-avg);
                        inx = i;
                }
        }

        return inx;
}

// The buildin qsort implementation
void qsort_builtin(double * array, size_t size, int threads)
{
        qsort(array, size, sizeof(double), double_cmpr);
}

// Arguments for the peer qsort
struct qpeer_args
{
        double * local_array;
        qpeer_args * next;
        pthread_barrier_t * barrier;
        size_t size;
        int threads;
        int tid;
};

// Peer odd-even quicksort
void qsort_peer(double * array, size_t size, int threads)
{
        size_t tsize, offset;
        int rem;
        int i;
        qpeer_args * args;
        pthread_t * thread_handles;
        pthread_barrier_t barrier;

        args = (qpeer_args *) malloc(sizeof(qpeer_args)*threads);
        thread_handles = (pthread_t *) malloc(sizeof(pthread_t)*threads);
        tsize = size / threads;
        rem = size % threads;
        pthread_barrier_init(&barrier, NULL, threads);

        // Allocate the parts of the array to the threads.
        offset = 0;
        for (i = 0; i < threads; ++i) {
                args[i].local_array = &array[offset];
                args[i].tid = i;
                args[i].threads = threads;
                args[i].barrier = &barrier;
                offset += args[i].size = tsize + (rem-- > 0);
                if (i < threads-1) {
                        args[i].next = &args[i+1];
                } else {
                        args[i].next = NULL;
                }
        }

        // Start the threads
        for (i = 0; i < threads; ++i) {
                pthread_create(&thread_handles[i], NULL, qsort_peer_pth, &args[i
        };

        // Wait for them to finish
        for (i = 0; i < threads; ++i) {
                pthread_join(thread_handles[i], NULL);
        };

        free(args);
        free(thread_handles);
        // Cleanup
        pthread_barrier_destroy(&barrier);
}

void * qsort_peer_pth(void * arg)
{
        int i;
        int merge_size;
        double * merge_buffer;
        qpeer_args * args = (qpeer_args *) arg;

        // malloc is not good for multithreading, let's allocate any space we ne
        // here and keep it until the end
        if (args->next) {
                merge_size = args->size + args->next->size;
                merge_buffer = (double *) malloc(sizeof(double) * merge_size);
        } else {
                merge_size = 0;
                merge_buffer = NULL;
        }
        // Sort out local data
        qsort_builtin(args->local_array, args->size, 0);
        // Repeat 'threads' times
        for (i = 0; i < args->threads; ++i) {
                // Wait for the previous phase to finish (also for the initial s
                pthread_barrier_wait(args->barrier);
                // Are we on a even or an odd round? Also, only one CPU from one
                // should do any work. Also the thread the far next edge should
                // careful, in case it has no next.
                if (args->next &&
                        ((is_even(i) && is_even(args->tid)) ||
                        (!is_even(i) && !is_even(args->tid))))
                {
                        //merge and exchange with tid+1
                        merge(args->local_array, args->size,
                              args->next->local_array, args->next->size,
                              merge_buffer);
                        split(merge_buffer,
                              args->local_array, args->size,
                              args->next->local_array, args->next->size);
                }
        }

        if (merge_buffer)
                free(merge_buffer);
        return NULL;
}

// Math functions
int is_power2(int num)
{
        // Powers of two contain only one '1'
        return !(num & (num - 1));
}

int is_even(int num)
```

```
{
        // Even numbers do not end in '1'
        return !(num & 1);
}

// Merge two sorted arrays into the third one
void merge(double * s1, size_t size1, double * s2, size_t size2, double * dest)
{
        int i1, i2, j;

        // Start merging
        i1 = i2 = j = 0;
        while (i1 < size1 && i2 < size2) {
                if (s1[i1] < s2[i2])
                        dest[j] = s1[i1++];
                else
                        dest[j] = s2[i2++];
                ++j;
        }
        // Copy what's left
        while (i1 < size1)
                dest[j++] = s1[i1++];
        while (i2 < size2)
                dest[j++] = s2[i2++];
}

// Split a array into two
void split(double * src, double * d1, size_t size1, double * d2, size_t size2)
{
        int i, j;

        j = 0;
        // Copy to the first destination
        for (i = 0; i < size1; ++i)
                d1[i] = src[j++];
        // Copy to the second destination
        for (i = 0; i < size2; ++i)
                d2[i] = src[j++];
}

struct qdist_args
{
        double * global_array;
        double * local_array;
        qdist_args * others;
        pthread_barrier_t * barrier;
        double average;
        size_t local_pivot;
        size_t size;
        int threads;
        int tid;
};
pthread_mutex_t mm = PTHREAD_MUTEX_INITIALIZER;
// Distributed quicksort
void * qsort_dist_pth(void * arg)
{
        double * merge_buffer;
        qdist_args * args = (qdist_args *) arg;
        qdist_args * other;
        double average;
        size_t pivot, middle;
        int i;

        qdist_args * others;
        pthread_barrier_t * barrier;
        int threads;
        int tid;

        others = args->others;
        barrier = args->barrier;
        threads = args->threads;
        tid = args->tid;
        while (threads > 1) {
                middle = threads / 2;

                // Find the pivot point and separate the local data
                average = args->local_array[find_avg(args->local_array, args->si
                args->average = average;
                pthread_barrier_wait(barrier);
                average = 0.0;
                for (i = 0; i < threads; ++i)
                        average += others[i].average;
                average = average / threads;
                args->local_pivot = pivot =
                    lower_bound(args->local_array, args->size, average);
                pthread_barrier_wait(barrier);

                // Exchange data
                size_t new_size;
                // Do we want the smallest or the largest part of the data?
                if (tid < middle) {
                        other = &others[tid + (threads / 2)];
                        new_size = pivot + other->local_pivot;
                        merge_buffer = (double *)malloc(sizeof(double) * new_siz
                        merge(args->local_array, pivot, other->local_array,
                              other->local_pivot, merge_buffer);
                } else {
                        other = &others[tid - (threads / 2)];
                        new_size = args->size - pivot + other->size - other->loc
                        merge_buffer = (double *)malloc(sizeof(double) * new_siz
                        merge(&args->local_array[pivot], args->size - pivot,
                              &other->local_array[other->local_pivot],
                              other->size - other->local_pivot, merge_buffer);
                }
                pthread_barrier_wait(barrier);

                // Prepare for the next round
                if (args->size)
                        free(args->local_array);
                args->local_array = merge_buffer;
                args->size = new_size;
                threads = threads / 2;
                if (tid >= middle) {
                        tid -= middle;
                        others = &others[middle];
                }
                pthread_barrier_wait(barrier);
        }

        // Gather the data
        size_t offset = 0;
        for (i = 0; i < args->tid; ++i)
                offset += args->others[i].size;
        memcpy(&args->global_array[offset], args->local_array,
               args->size * (sizeof(double)));

        // That's all folks!
        pthread_barrier_wait(args->barrier);
        return NULL;
}

void * qsort_dist_init(void * arg)
{
        qdist_args * args = (qdist_args *) arg;
        double * local_array;

        // We need to keep our data separate, otherwise it becomes kind of mess
        // later on
        local_array = (double *) malloc(sizeof(double) * args->size);
        memcpy(local_array, args->local_array, args->size * sizeof(double));
        args->local_array = local_array;

        // Sort locally and start the main sorting routine
        qsort_builtin(args->local_array, args->size, 0);
        qsort_dist_pth(arg);

        // Cleanup
        if (args->size)
                free(args->local_array);
        return NULL;
}

void qsort_dist(double * array, size_t size, int threads)
{
        size_t tsize, offset;
        int rem;
        int i;
        qdist_args * args;
        pthread_t * thread_handles;
        pthread_barrier_t barrier;

        args = (qdist_args *) malloc(sizeof(qdist_args)*threads);
        thread_handles = (pthread_t *) malloc(sizeof(pthread_t)*threads);
        tsize = size / threads;
        rem = size % threads;
        pthread_barrier_init(&barrier, NULL, threads);

        // Allocate the parts of the array to the threads.
        offset = 0;
        for (i = 0; i < threads; ++i) {
                args[i].global_array = array;
                args[i].local_array = &array[offset];
                args[i].others = args;
                args[i].tid = i;
                args[i].threads = threads;
                args[i].barrier = &barrier;
                args[i].size = tsize + (rem-- > 0);
                offset += args[i].size;
        }

        // Start the threads
        for (i = 0; i < threads; ++i) {
                pthread_create(&thread_handles[i], NULL, qsort_dist_init, &args[
        };

        // Wait for them to finish
        for (i = 0; i < threads; ++i) {
                pthread_join(thread_handles[i], NULL);
        };

        // Cleanup
        free(args);
        free(thread_handles);
        pthread_barrier_destroy(&barrier);
}

// Find the largest element in the sorted array that is less than the bound
size_t lower_bound(const double * arr, size_t size, double e)
{
        const double * p;
        size_t count, step;

        count = size;
        p = arr;
        while (count > 0) {
                step = count / 2;
                if (p[step] < e) {
                        p += step + 1;
                        count -= step + 1;
                } else {
                        count = step;
                }
        }

        return p - arr;
}

int timer(void)
{
  struct timeval tv;
  gettimeofday(&tv, (struct timezone*)0);
  return (tv.tv_sec*1000000+tv.tv_usec);
}
```