ECE445 Parallel and Network Computing
Winter Semester 2022-2023
Task 2

Student group:
Ioannis Roubos - 2980
Gerasimos Agoras - 2947

Exercise 1

a) ____________ Serial Algorithm____________

X: Input array
Y: Output array
n: length of array
ω: nth root of unity

```
procedurefft(x, y, n, ω)
ifn=1then
        y[0] = x[0]
else
        fork=0that –/2 –1
                p[k] = x[2k]
                s[k] = x[2k+1]
        end
        fft(p, q, ,–/2 oh2)
        fft(s, t, , –/2 oh2)
        fork=0thatn-1
                y[k] = q[k mod (–/2)] + okt[k mod ( /2)]
        end
end
```

Cost calculation:

T(n) = ∑ ∑ –=∑ →* – =∑ =O(nlogn)

b) __________ Parallel Algorithm__________

X: Input array
Y: Output array
n: length of array
ω: nth root of unity

```
procedurefft(x_myID, y_myID, n) r
= log(n)
MPI_COMM_RANK(&myID)
R_myID = x_myID
```

```
for m=0 that r-1
        S_myID = R_myID
        j = (b0... bm-1, 0, bm+1... br-1) k
        = (b0... bm-1, 1, bm+1... br-1) if
        myID = j
                MPI_SEND(SJ, k)
                MPI_RECV(SK, k)
        end
        if myID = k
                MPI_SEND(SJ, j)
                MPI_RECV(SK, j)
        end
        R_myID = SJ+SKX o(bm, bm-1 ... b0, 0 ... 0)
        MPI_BARRIER()
end
y = R_myID
otherID = (br-1... b2, b1, b0)
if myID ≠ otherID
        if myID < otherID
                MPI_SEND(y, otherID)
                MPI_SEND(y_myID, otherID)
        end else
                MPI_SEND(y_myID, otherID)
                MPI_SEND(y, otherID)
        end
end
```

Cost calculation:

$T(n) = (t_c + t_s + t_w) \log(n)$

Where:

t_c: cost of multiplication and addition t_s: start-up cost of processes t_w: transfer cost per word For the parallel execution time we have:

$T(p) = t_c * n\log(n)/p + t_s * \log(p) + t_w * n\log(p)/p = O(n\log(n))$ for $p \leq n$

As in more detail we have:

- nlog(n) operations which are divided into p operations, hence the factor of t_c
- thesetup time is log(p) from the original formula, due to the Divide And Conquer nature of the algorithm
- its factortw are n/p redefinings of the components for each process for log(p) stages

c)

<u>Time improvement</u> : S = t_c*n*log(n)/T $_{(b)}$(p) = $\frac{p*n*log(n)}{n*log(n)+ * *log( )+ * *log( )}$

<u>Performance</u> : E = $T_{(b)}$(1)/$T_{(b)}$(p) = $\frac{1}{1+ \frac{* *log( )}{* *log( )} + \frac{*log( )}{*log( )}}$

d)

dS/dp = $\frac{\left(\frac{p*n*log(n)}{n*log(n)+ \; * *log( )+ \; * *log ( )}\right)}{}$ =

$\frac{n*log(n)(n*log(n)+ * *log( )+ * *log( )) - * *log ( )(*log( )+ + *)}{(n*log(n)+ * *log( )+ * *log( ))^2}$ = 0

n ∗ log(n) (n ∗ log(n) + ∗ ∗ log( ) + ∗ ∗ log( )) − ∗ ∗ log ( )(∗ log( ) + + ∗) =0 -

(even if n=256) 2048(2048 + ∗ ∗ log( ) + ∗ 256 ∗ log( )) − ∗ 2048(∗ log( ) + + ∗ 256) =0 -

256 ∗ ∗ ( ) = −2048 + ∗ + 256- ∗ ( ) = −1792 + ∗ $\frac{1}{256}$ ∗

e)

dE/dp = $\frac{\left(\frac{1}{1+\frac{* *log}{* *log( )}( )+ \frac{*log}{*log( )})( )}\right)}{}$ = $\frac{\frac{*log( )}{* *log( )} + \frac{}{* \; *log( )} + \frac{1*}{*log( )}}{(1+ \frac{* *log( )}{* *log( )} + \frac{*log( )}{*log( )})^2}$ = 0

$\frac{*log( )}{* *log( )}$ + $\frac{}{* *log( )}$ + $\frac{1*}{}$ $\frac{}{*log( )}$ = 0 - (even if n=256) $\frac{*log( )+}{*2048}$ $\frac{}{*2048}$ + $\frac{1*}{*8}$ =0

Exercise 2

We run the serial FFT algorithm, which is based on the Cooley-Tukey algorithm shown in the file fft_serial.c with sizes 2^{10}... 2^{20} and the times in seconds we get are:

K = 10	0.000604976	**K = 16**	0.0286929
K = 11	0.00123422	**K = 17**	0.0574807
K = 12	0.00292676	**K = 18**	0.121656
K = 13	0.00593098	**K = 19**	0.260288
K = 14	0.00644857	**K = 20**	0.584439
K = 15	0.0134876		

Exercise 3

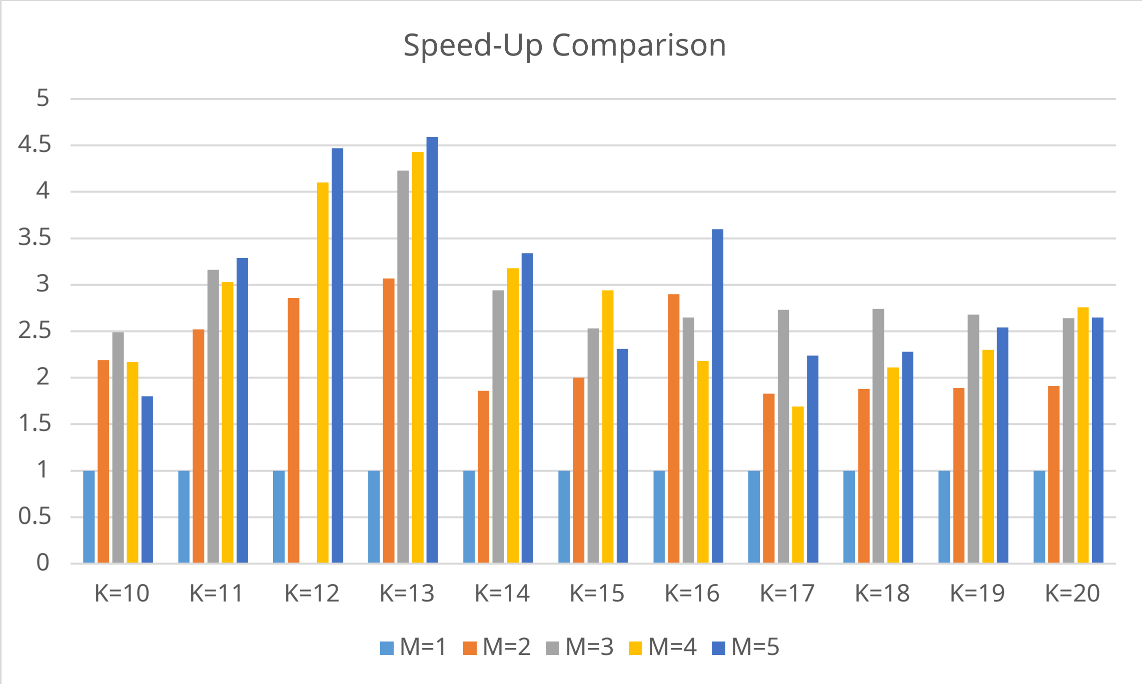
Adding OpenMP to the algorithm we used in Exercise 2 (fft_parallel.c), and with the same sizes that we used above, we notice that for the number of threads m = 6, an overhead of size k=10 (size = 2^{10}). This leads us to get counts up to a thread count of 5.

Exercise 4

The measurements we get for various exponents of size k and various thread counts m are shown in the table below. The speedup shown in each case is compared to the corresponding size for 1-threaded execution.

K vs M	1	2	3	4	5
10	0.000604976	0.000275948 (x2.19)	0.000242511 (x2.49)	0.00027853 (x2.17)	0.000335888 (x1.8)
11	0.00123422	0.00049004 (x2.52)	0.000390815 (x3.16)	0.000407605 (x3.03)	0.000375046 (x3.29)
12	0.00292676	0.00102307 (x2.86)	0.000719739 (x4.07)	0.000712662 (x4.1)	0.000654538 (x4.47)
13	0.00593098	0.00193173 (x3.07)	0.00140042 (x4.23)	0.00133922 (x4.43)	0.00129183 (x4.59)
14	0.00644857	0.00346731 (x1.86)	0.00218747 (x2.94)	0.00202571 (x3.18)	0.00192914 (x3.34)
15	0.0134876	0.00673193 (x2)	0.00532629 (x2.53)	0.00458149 (x2.94)	0.00582581 (x2.31)
16	0.0286929	0.0150378 (x1.9)	0.0108064 (x2.65)	0.0131791 (x2.18)	0.00795992 (x3.6)
17	0.0574807	0.0313205 (x1.83)	0.0210725 (x2.73)	0.0340494 (x1.69)	0.0256495 (x2.24)
18	0.121656	0.0646525 (x1.88)	0.0443356 (x2.74)	0.0575643 (x2.11)	0.0533176 (x2.28)
19	0.260288	0.137451 (x1.89)	0.0970999 (x2.68)	0.112899 (x2.3)	0.10246 (x2.54)
20	0.584439	0.305693 (x1.91)	0.221649 (x2.64)	0.211489 (x2.76)	0.220866 (x2.65)

The best speedup we observe is the one for size = $2^{13}$and threads = 5, with an improvement of its order**459%** .

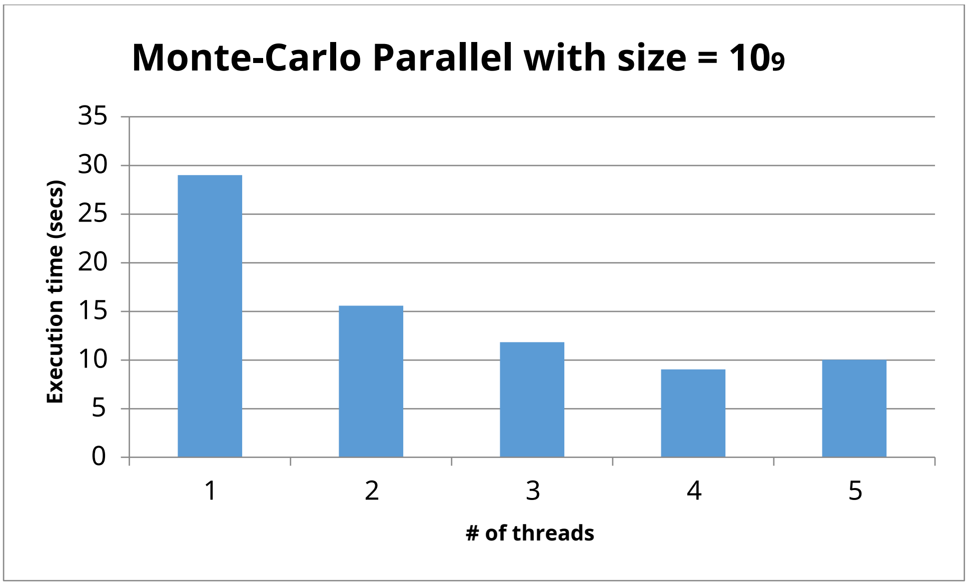


(These measurements can be found in detail in excel within the zip, sheet “ex4”)

We run the serial FFTW for size = $2^{13}$and the time we get is equal to 0.000721484, which is better than our best implementation (0.00129183). This shows that the parallel FFT algorithm needs more improvement.
(For his compilefftw.c, we use the flags:**gcc -Wall -g fftw.c -o fftw -lftw3 -lm**)

Exercise 5

Below we have the execution time for various numbers of threads in the Monte-Carlo algorithm. We notice that for the indicative size = 10^9we have speed-up up to a number of threads equal to 4, while when we go to 5 overhead appears and the execution time gets worse.



In the excel file included in the zip there are times for other sizes as well (sheet "ex5").

ANNEX

Machine description

- Number of cores: 8
- Cache sizes:
  L1d cache: 128KiB
  L1i cache: 128KiB
  L2 cache: 1 MiB
  L3 cache: 6MiB

Code of Practice 2 (eg)

Here is the code for query 2. File Name “fft_serial.c”

```
/********************************
 * Ergasia 2 – Askhsh 2
 * Roumpos Ioannis - 2980
 * Agoras Gerasimos - 2947
 *********************************/
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
# include <complex.h>
# include <string.h>
# include <time.h>
# include <omp.h>

# define pi 3.14159265358979323846

unsigned int bitReverse(unsigned int x, int size) {

	int n = 0;
	for (int i = 0; i < log2(size); i++) {

		n <<= 1;
		n |= (x & 1);
		x >>= 1;
	}
	return n;
}

void fft(complex double* input, complex double* output, int size)

{
	int k,j,i;
```

```
    complex double expTable[size/2];

    // bit reversal of the given array for (i = 0; i <
    size; ++i) {
        int rev = bitReverse(i, size); output[i] =
        input[rev];
    }

    // Trigonometric Table
    for(k = 0; k < size / 2; k++){
        expTable[k] = cexp(-2*pi*k/size*I);
    }

    int n,halfsize,tablestep; double
    complex temp?

    // Cooley-Tukey decimation-in-time radix-2 FFT for(n = 2; n <=
    size; n *= 2){
        halfsize = n/2;
        tablestep = size / n;

        for(i = 0; i < size; i += n){
            for(j = i, k = 0; j < i + halfsize; j++, k +=
tablestep){
                temp = output[j + halfsize] * expTable[k]; output[j +
                halfsize] = output[j] - temp; output[j] += temp;

            }
        }

        if(n == size)           // Prevent overflow in size *= 2
            breaks?
    }
}



int main(int argc,char* argv[]){
    struct timespec tv1, tv2;

    if(argc != 2){
        printf("Give correct number of arguments!!"); return -1;

    }

    int n = atoi(argv[1]); if((ceil(log2(n)) != floor(log2(n))))
    {
        printf("Size must be a power of 2!!"); return -1;

    }

    double complex* input = malloc(n*sizeof(double complex)); if (input ==
    NULL){
        printf("Error in malloc!"); return -1;

    }
```

```
        double complex* output = malloc(n*sizeof(double complex)); if (output ==
        NULL){
                printf("Error in malloc!"); return -1;

        }

        printf("Input array\n"); for(int
        i=0; i<n; i++){
                input[i] = (i+1) + 0*I; printf("%lf
                +
%lf*i\n",creal(input[i]),cimag(input[i]));
        }

        clock_gettime(CLOCK_MONOTONIC_RAW, &tv1);
        fft(input,output,n);
        clock_gettime(CLOCK_MONOTONIC_RAW, &tv2);


        printf("\nOutput array\n"); for(int
        i=0; i<n; i++){
                printf("%lf +
%lf*i\n",creal(output[i]),cimag(output[i]));
        }

        printf ("Total time = %10g seconds\n",
                              (double) (tv2.tv_nsec - tv1.tv_nsec) /
1000000000.0 +
                              (double) (tv2.tv_sec - tv1.tv_sec));

        free(input);
        free(output);
        return 0;
}
```

Code of Practice 3 (eg)

Here is the code for query 3. File Name “fft_parallel.c”

```
/*********************************
 * Ergasia 2 – Askhsh 3
 * Roumpos Ioannis - 2980
 * Agoras Gerasimos - 2947
 *********************************/
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
# include <complex.h>
# include <string.h>
# include <time.h>
# include <omp.h>
```

```
# define pi 3.14159265358979323846

unsigned int bitReverse(unsigned int x, int size) {

	int n = 0;
	for (int i = 0; i < log2(size); i++) {

		n <<= 1;
		n |= (x & 1);
		x >>= 1;
	}
	return n;
}

void fft(complex double* input, complex double* output, int size)

{
	int k,j,i;
	complex double *expTable = malloc(size/2*sizeof(double complex));

	if(expTable == NULL)
		return?

	// bit reversal of the given array
	# pragma omp parallel for shared(output) private(i) for (i = 0; i <
	size; ++i) {
		int rev = bitReverse(i, size); output[i] =
		input[rev];
	}

	# pragma omp parallel for shared(expTable,size) private(k) for(k = 0; k <
	size / 2; k++){
		expTable[k] = cexp(-2*pi*k/size*I);
	}

	int n,halfsize,tablestep; double
	complex temp?

	# pragma omp parallel
private(n,halfsize,tablestep,i,j,k,temp) shared(output)
	{
	for(n = 2; n <= size; n *= 2){
		halfsize = n/2;
		tablestep = size / n;

	# pragma omp for
		for(i = 0; i < size; i += n){
			for(j = i, k = 0; j < i + halfsize; j++, k +=
tablestep){
				temp = output[j + halfsize] * expTable[k]; output[j +
				halfsize] = output[j] - temp; output[j] += temp;

			}
		}

		if(n == size)
```

```
                    breaks?
        }
        }
        free(expTable);
}


int main(int argc,char* argv[]){
        struct timespec tv1, tv2;

        if(argc != 2){
                printf("Give correct number of arguments!!"); return -1;

        }

        int n = atoi(argv[1]); n =
        pow(2,n);

        double complex* input = malloc(n*sizeof(double complex)); if (input ==
        NULL){
                printf("Error in malloc!"); return -1;

        }
        double complex* output = malloc(n*sizeof(double complex)); if (output ==
        NULL){
                printf("Error in malloc!"); return -1;

        }

        /*printf("Input array\n"); for(int i=0;
        i<n; i++){
                input[i] = (i+1) + 0*I; printf("%lf
                +
%lf*i\n",creal(input[i]),cimag(input[i]));
        }*/

        clock_gettime(CLOCK_MONOTONIC_RAW, &tv1);
        fft(input,output,n);
        clock_gettime(CLOCK_MONOTONIC_RAW, &tv2);


        /*printf("\nOutput array\n"); for(int i=0;
        i<n; i++){
                printf("%lf +
%lf*i\n",creal(output[i]),cimag(output[i]));
        }*/
        printf("N = %.2lf ",log2(n)); printf ("Total time =
        %10g seconds\n",
                        (double) (tv2.tv_nsec - tv1.tv_nsec) /
1000000000.0 +
                        (double) (tv2.tv_sec - tv1.tv_sec));

        free(input);
        free(output);
        return 0;
}
```

Code of Practice 4 (eg)

Here is the code for query 4. File Name “fftw.c”

```
/********************************
 * Ergasia 2 – Askhsh 4
 * Roumpos Ioannis - 2980
 * Agoras Gerasimos - 2947
 ********************************/
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
# include <time.h>
# include <fftw3.h>

int main(int argc,char* argv[]){
	struct timespec tv1,tv2; int N =
	atoi(argv[1]); N = pow(2,N);


	fftw_complex *in,*out;
	fftw_plan p;

	in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) *
N);
	for(int i=0; i < N; i++){
		in[i][0] = i+1;
		in[i][1] = 0;
	}

	out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) *
N);
	clock_gettime(CLOCK_MONOTONIC_RAW, &tv1); p =
	fftw_plan_dft_1d(N, in, out, FFTW_FORWARD,
	FFTW_ESTIMATE);

	fftw_execute(p); /* repeat as needed */
	clock_gettime(CLOCK_MONOTONIC_RAW, &tv2);
	fftw_destroy_plan(p);

	printf ("Total time = %10g seconds\n",
				(double) (tv2.tv_nsec - tv1.tv_nsec) /
1000000000.0 +
				(double) (tv2.tv_sec - tv1.tv_sec));

	fftw_free(in); fftw_free(out); return 0;

}
```

Code of Practice 5 (eg)

Here is the code for query 5. File Name “mc_parallel.c”

```
/********************************
 * Ergasia 2 – Askhsh 5
 * Roumpos Ioannis - 2980
 * Agoras Gerasimos - 2947
 *********************************/
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
# include <omp.h>
# include <time.h>
inline double f(double x) {

        return sin(cos(x));
}

// WolframAlpha: integral sin(cos(x)) from 0 to 1 //                    =     0.738643
                                                                   0.73864299803689018
//
        0.7386429980368901838000902905852160417480209422447648518
        714116299

int main(int argc, char *argv[]) {

        double a = 0.0;
        double b = 1.0;
        unsigned long n = 24e8; long
        tseed = time(0);

        if (argc == 2) {
                tseed = atol(argv[1]);
        }
        else if (argc == 3) {
                n = atol(argv[1]);
                tseed = atol(argv[2]);
        }

        const double h = (ba)/n;
        const double ref = 0.73864299803689018; double
        res = 0;
        double t0, t1;
        unsigned long i?

      t0 = omp_get_wtime();

# pragma omp parallel {

        double local_res = 0; double
        xi?
```

```
        unsigned short buffer[3];
        buffer[0] = 0;
        buffer[1] = 0;
        buffer[2] = tseed+omp_get_thread_num();

# pragma omp for
        for (i = 0; i < n; i++) {
                xi = errand48(buffer);
            local_res += f(xi);
          }

# pragma omp atomic res +=
local_res;

}

        res *= h;
        t1 = omp_get_wtime();

          printf("Result=%.16f Error=%e Rel.Error=%e Time=%lf seconds\n", res,
fabs(res-ref), fabs(res-ref)/ref, t1-t0);

          return 0;
}
```