Introduction to Parallel Computing
Homework 1: Implicit parallelism techniques and performance models.
Results report

Alessandro Iepure, 228023
alessandro.iepure@studenti.unitn.it

25 October 2023

Abstract

This document analyzes the results and methodologies used while working on the assignment in the title. All work is original and done without input from outside sources except where explicitly noted.

1 Array addition and vectorization

Task1 and *Task2* ask for two functions, `routine1()` and `routine2()` respectively, to compute the element-wise sum of two float arrays, namely a and b, into another array c. Both functions should also measure the time it takes the `for`-loop to complete.
The only difference between the two functions is that `routine2()` uses implicit parallelism. The actual logic remains the same. To avoid code duplication, I reused `routine1()` for both tasks and played with the compiler flags to achieve implicit parallelism.
Source code 1 shows the trivial implementation of the requested logic.

Source code 1: Implemented algorithm

```c
#include <time.h>

float *routine1(const float *a, const float *b, float *c, int dim) {

    clock_t t0, t1;

    t0 = clock();
    for (int i = 0; i < dim; i++) {
        c[i] = a[i] + b[i];
    }
    t1 = clock();

    printf("%i,%12.4f\n", dim, (t1 - t0) / 1000000.0);
    return c;
}
```

After reading GCC's documentation[1, 2, 3, 4], I decided to use the following commands to compile the code:

- *Task1* (Sequential version): `gcc -std=c99 -o bin/array-seq.out src/array.c`
  Arguments:
  - `-std=c99`: specifies the *C* dialect used, *C99* in this case. Needed to allow index declarations inside `for`-loops. It does not impact speed and binary size whatsoever.
  - `-o bin/array-seq.out`: specifies the generated binary file path.
- *Task2* (Parallel version): `gcc -std=c99 -O2 -ftree-vectorize -funroll-loops -fprefetch-loop-arrays -march=native -o bin/array-par.out src/array.c`

Arguments:

- `-std=c99`: specifies the *C* dialect used, *C99* in this case. Needed to allow index declarations inside `for`-loops. It does not impact speed and binary size whatsoever.
- `-O2`: applies general optimizations that do not involve a space-speed tradeoff[3]
- `-ftree-vectorize`: enables vectorization[3]
- `-funroll-loops`: enables loop unrolling[3]
- `-fprefetch-loop-arrays`: enables memory prefetching to aid looping on large arrays[3]
- `-march=native`: enables all supported extensions by the CPU in use[1]. It accelerates mathematical operations, specifically floating-point ones, in hardware[4].

The compilation flags, specifically those for the second version, were chosen after testing with multiple combinations. After weighting the obtained results, I decided to use the one that yielded a result correctly parallelized and had a visible increase in execution speed.
Alongside the plain *C* code, I wrote a *bash* script that wraps the compilation commands and binary execution into a single file to ease the testing phase. Each binary is called with array sizes between 2^4 and 2^{22} (both inclusive) three times. Both sequential and parallelized runs dump their results into a `CSV` file so that I can examine them in external programs.

Results analysis

As per indication in the assignment, I executed the script on the University's cluster (GCC 4.8.5). Table 1 and Table 2 report the obtained results.

Table 1: Run times by array size - Sequential (times in seconds)

Size	Run 1	Run 2	Run 3	Average
2^4	0.0000	0.0000	0.0000	0.0000
2^5	0.0000	0.0000	0.0000	0.0000
2^6	0.0000	0.0000	0.0000	0.0000
2^7	0.0000	0.0000	0.0000	0.0000
2^8	0.0000	0.0000	0.0000	0.0000
2^9	0.0000	0.0000	0.0000	0.0000
2^{10}	0.0000	0.0000	0.0000	0.0000
2^{11}	0.0000	0.0000	0.0000	0.0000
2^{12}	0.0000	0.0000	0.0000	0.0000
2^{13}	0.0000	0.0000	0.0000	0.0000
2^{14}	0.0000	0.0000	0.0000	0.0000
2^{15}	0.0000	0.0000	0.0000	0.0000
2^{16}	0.0000	0.0000	0.0000	0.0000
2^{17}	0.0000	0.0000	0.0000	0.0000
2^{18}	0.0000	0.0000	0.0000	0.0000
2^{19}	0.0000	0.0000	0.0000	0.0000
2^{20}	0.0200	0.0000	0.0200	0.0133
2^{21}	0.0200	0.0000	0.0200	0.0133
2^{22}	0.0200	0.0300	0.0300	0.0267

Table 2: Run times by array size - Parallelised (times in seconds)

Size	Run 1	Run 2	Run 3	Average
2^4	0.0000	0.0000	0.0000	0.0000
2^5	0.0000	0.0000	0.0000	0.0000
2^6	0.0000	0.0000	0.0000	0.0000
2^7	0.0000	0.0000	0.0000	0.0000
2^8	0.0000	0.0000	0.0000	0.0000
2^9	0.0000	0.0000	0.0000	0.0000
2^{10}	0.0000	0.0000	0.0000	0.0000
2^{11}	0.0000	0.0000	0.0000	0.0000
2^{12}	0.0000	0.0000	0.0000	0.0000
2^{13}	0.0000	0.0000	0.0000	0.0000
2^{14}	0.0000	0.0000	0.0000	0.0000
2^{15}	0.0000	0.0000	0.0000	0.0000
2^{16}	0.0000	0.0000	0.0000	0.0000
2^{17}	0.0000	0.0000	0.0000	0.0000
2^{18}	0.0000	0.0000	0.0000	0.0000
2^{19}	0.0000	0.0000	0.0000	0.0000
2^{20}	0.0100	0.0000	0.0100	0.0067
2^{21}	0.0100	0.0100	0.0100	0.0100
2^{22}	0.0100	0.0200	0.0100	0.0133

As is visible, the results are mostly instantaneous. Because zero values do not help in weighting the benefits of one version over the other, I opted to run the same script on my local machine, a laptop equipped with an

[1] Useful only if the compiling machine is the same as the executing one[4].

Intel® Core™ i5-8300H at 2.30GHz and 16GB of RAM running Fedora 38 (GCC 13.2.1). Table 3 and Table 4 gather the obtained results.

Table 3: Run times by array size - Local machine - Sequential (times in seconds)

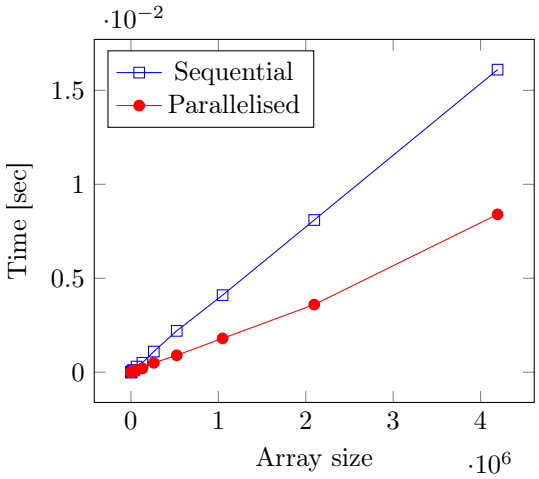
Size	Run 1	Run 2	Run 3	Average
2^4	0.0000	0.0000	0.0000	0.0000
2^5	0.0000	0.0000	0.0000	0.0000
2^6	0.0000	0.0000	0.0000	0.0000
2^7	0.0000	0.0000	0.0000	0.0000
2^8	0.0000	0.0000	0.0000	0.0000
2^9	0.0000	0.0000	0.0000	0.0000
2^{10}	0.0000	0.0000	0.0000	0.0000
2^{11}	0.0000	0.0000	0.0000	0.0000
2^{12}	0.0000	0.0000	0.0000	0.0000
2^{13}	0.0000	0.0000	0.0000	0.0000
2^{14}	0.0001	0.0001	0.0001	0.0001
2^{15}	0.0001	0.0001	0.0001	0.0001
2^{16}	0.0003	0.0003	0.0002	0.0003
2^{17}	0.0005	0.0005	0.0006	0.0005
2^{18}	0.0010	0.0012	0.0010	0.0011
2^{19}	0.0021	0.0021	0.0023	0.0022
2^{20}	0.0041	0.0041	0.0041	0.0041
2^{21}	0.0082	0.0081	0.0081	0.0081
2^{22}	0.0163	0.0160	0.0161	0.0161

Table 4: Run times by array size - Local machine - Parallelised (times in seconds)

Size	Run 1	Run 2	Run 3	Average
2^4	0.0000	0.0000	0.0000	0.0000
2^5	0.0000	0.0000	0.0000	0.0000
2^6	0.0000	0.0000	0.0000	0.0000
2^7	0.0000	0.0000	0.0000	0.0000
2^8	0.0000	0.0000	0.0000	0.0000
2^9	0.0000	0.0000	0.0000	0.0000
2^{10}	0.0000	0.0000	0.0000	0.0000
2^{11}	0.0000	0.0000	0.0000	0.0000
2^{12}	0.0000	0.0000	0.0000	0.0000
2^{13}	0.0000	0.0000	0.0000	0.0000
2^{14}	0.0000	0.0000	0.0000	0.0000
2^{15}	0.0001	0.0001	0.0001	0.0001
2^{16}	0.0001	0.0001	0.0001	0.0001
2^{17}	0.0003	0.0002	0.0002	0.0002
2^{18}	0.0005	0.0006	0.0005	0.0005
2^{19}	0.0009	0.0009	0.0009	0.0009
2^{20}	0.0018	0.0018	0.0017	0.0018
2^{21}	0.0036	0.0036	0.0036	0.0036
2^{22}	0.0073	0.0082	0.0096	0.0084

Results from different machines with different architectures and components cannot be compared to each other, especially if the code is compiled for a particular machine. From now on, I am going to consider only the results obtained from my laptop.
The average run times are plotted in Plot 1. The obtained graph is the initial part of the so-called "Roofline model". If the program were to be run with higher values and the machine was able to compute the result, the graph would rise to a certain point called "peak" and from then on would continue as a flat line having reached the maximum performance.
Overall, by implicitly parallelizing the code, we gained around 57% (average) more execution speed.

Plot 1: Run times by array size



2 Matrix copy via block reverse ordering

Task1 and *Task2* ask for two functions, `routine1()` and `routine2()` respectively, that take a matrix M of size $N \times N$ and reverse the order in blocks $b \times b$ in another matrix O.
As the previous exercise, the only difference between the two functions is whether implicit parallelism is used. I chose again to avoid code duplication and accomplish what was requested via compilation flags, the same used in the previous exercise. In addition, the *bash* script was integrated with the new commands and, as before, each version was executed three times with b values ranging from 2^2 to 2^8 (inclusive).
Source code 2 shows the implemented algorithm. It loops around each $b \times b$ block in the original matrix M and, for each, copies the values of the current block and the opposite one in the reverse order. Doing both copies in the same iteration allows the main matrix to be looped only for half of the rows.

Source code 2: Implemented algorithm

```
#include <time.h>

float **routine1(float **m, int dim, int b, float **o) {
    clock_t t0, t1;

    t0 = clock();
    for (int i = 0; i < dim / 2; i += b) {
        for (int j = 0; j < dim; j += b) {
            for (int k = 0; k < b; k++) {
                for (int l = 0; l < b; l++) {
                    o[i + k][j + l] = m[dim - i - (b - k)][dim - j - (b - l)];
                    o[dim - i - (b - k)][dim - j - (b - l)] = m[i + k][j + l];
                }
            }
        }
    }
    t1 = clock();

    printf("%i,%12.4f\n", b, (t1 - t0) / 1000000.0);
    return (float **) o;
}
```

Results analysis

Table 5 and Table 6 show the run times obtained for each execution. The parallel version is 49% (average) quicker than the same algorithm executed sequentialy.

Table 5: Run times by b size - Sequential (times in seconds)

Size	Run 1	Run 2	Run 3	Average
2^2	0.1200	0.1300	0.1300	0.1267
2^3	0.1400	0.1400	0.1200	0.1333
2^4	0.2000	0.2000	0.2000	0.2000
2^5	0.1900	0.1700	0.1800	0.1800
2^6	0.1600	0.1600	0.1600	0.1600
2^7	0.1500	0.1500	0.1500	0.1500
2^8	0.1400	0.1300	0.1300	0.1333

Table 6: Run times by b size - Parallelised (times in seconds)

Size	Run 1	Run 2	Run 3	Average
2^2	0.0700	0.0800	0.0800	0.0767
2^3	0.0700	0.0600	0.0700	0.0667
2^4	0.0800	0.1000	0.0900	0.0900
2^5	0.0800	0.0700	0.0700	0.0733
2^6	0.0800	0.0900	0.0800	0.0833
2^7	0.0700	0.0900	0.0800	0.0800
2^8	0.0700	0.0700	0.0700	0.0700

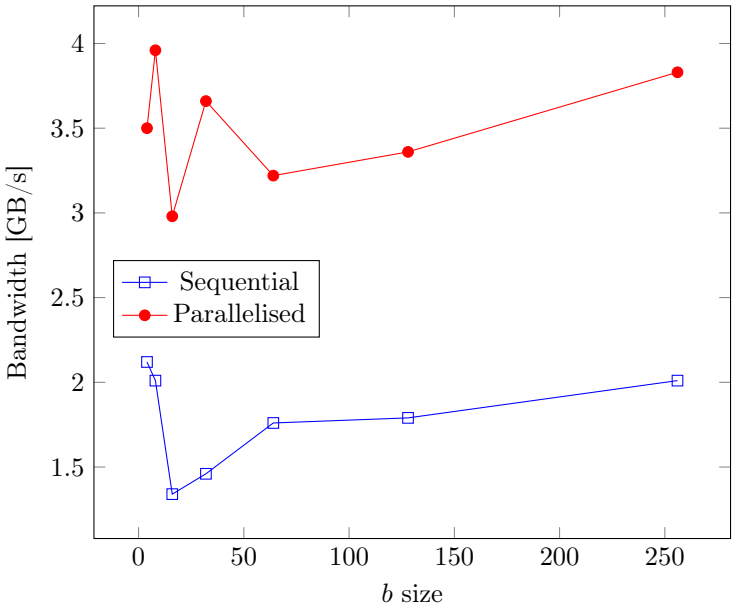
To compute the effective bandwidth, it is necessary to know the number of bytes read B_r and written B_w. In our case, both values are equal and calculated as $(N \times N) * S_f$, where $N \times N$ is the size of the matrix M and S_f is the size of a *float* type (4 bytes). The algorithm reads two values and stores the same quantity, totaling four operations for each matrix position. Substituting this information in equation (1), we get the results in Table 7. Plot 2 shows a visual representation of the effective bandwidth.

$$b = \frac{(B_r + B_w)/10^9}{t} = \frac{(4 * 4096^2 * 4)/10^9}{t} \qquad \text{[GB/s]} \tag{1}$$

Table 7: Effective bandwidths in GB/s

Size	Sequential	Parallelised
2^2	2.12	3.50
2^3	2.01	3.96
2^4	1.34	2.98
2^5	1.49	3.66
2^6	1.76	3.22
2^7	1.79	3.36
2^8	2.01	3.83
Average	1.79	3.50

Plot 2: Effective bandwidth by b size



References

[1] GCC team. *Auto-vectorization in GCC*. URL: https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[2] GCC team. *GCC Developer Options*. URL: https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html.

[3] GCC team. *Options That Control Optimization*. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[4] GCC team. *x86 Options*. URL: https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html.