Questions:
In order to save you precious time please follow the next questions policy:
- Questions regarding the assignment please address to Itay and Anat
- Technical questions regarding the servers you should run your programs on please address Ido - the TA in charge of the course.

In this exercise you will implement the parallel version of Walsh-Hadamard transform (WHT). This transform is an example of a generalized class of Fourier transforms. It performs an orthogonal, symmetric, involutary, linear operation on \(2^m\) numbers (we will assume that they are integers).

WHT is performed by multiplying a given vector of length \(2^m\) by Hadamard matrix of size \(2^m \times 2^m\), which is generated as follows:

\[
H_0 = +1 \\
H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \\
H_m = \begin{bmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{bmatrix}
\]

An element in a Hadamard matrix is also given by:

\[(H_m)_{k,n} = (-1) \sum_{j} k_j n_j\]

where \(k_j\) and \(n_j\) are binary digits of the binary representations of \(k\) and \(j\) respectively. Simply put, to get the value of \((H_m)_{k,n}\) you should AND the binary representations of \(k\) and \(n\). If the result has an even number of 1s, then \((H_m)_{k,n} = 1\), otherwise \((H_m)_{k,n} = -1\). You can find more details in Wikipedia: [http://en.wikipedia.org/wiki/Hadamard_transform](http://en.wikipedia.org/wiki/Hadamard_transform) (Note that in this assignment we omit the normalization constant.)

In this assignment you will implement two versions of this transform: slow version (through simple matrix product - which takes \(O(2^{2m})\) time) and fast version (by employing the Cooley-Tukey algorithm, as in FFT - which takes \(O(m2^m)\) time). The fast version is one of the fundamental algorithms used in numerous applications and fast execution is of interest to many.
**Part 1 (20 points): Implement simple_parallel_walsh(int* vector, int size)**

This procedure performs naive WHT by generating the columns of WHT matrix and multiplying them by the input vector. Both the WHT matrix generation and the multiplication should be parallelized via OpenMP. For multiplication, the granularity of the parallelization can be as low as one value of the output vector per one thread. Note that the implementation should not generate Hadamard matrix first and then multiply the vector, since the matrix will not fit the memory.

Hints (use is not mandatory):

- You can use the expression `i & j` to get the Bitwise AND between `i` and `j`.
- You can use the expression `x &= x - 1` to set the first 1-bit in `x` to zero.

![Figure 1: Fast Walsh Transform (from Wikipedia)](image)

**Part 2 (50 points): Implement fast_parallel_walsh(int* vector, int size)**

This procedure performs fast WHT by using the Cooley-Tukey divide-and-conquer algorithm as in the example in Figure 1. In this example the vector of size 8 is transformed in 3=log(8) stages. To better understand the way this transform is performed please read the chapter on FFT in CLR (Cormen, Leiserson, Rivest) book, and/or check the Wikipedia pages on FFT and WHT explaining the concept.

You may also find this link useful: [http://fourier.eng.hmc.edu/e161/lectures/wht/node2.html](http://fourier.eng.hmc.edu/e161/lectures/wht/node2.html)

Performance grading

Your implementation should be scalable and efficient. Therefore, your grade will rely on the performance of your code:

1. Comparing to your own implementation using 1 thread, the speedup for 4 or 8 threads (on a $2^{24}$ sized vector and 8 cpu cores) should be at least 2.4. If it's lower, you will lose 1 point for every 0.1 difference (up to 10 points).  
   Namely, if you got a speedup of 1.9 for 8 threads and 2.1 for 16, you will lose $(3.2-2.9) \times 10 + (3.2-2.7) \times 10 = 8$ points

2. Bonus: we will hold a competition, in which the speedups of all implementations will be compared to our serial implementation. Bonuses of 3, 2 and 1 points (to the final grade) will be awarded to the 3 implementations that obtained the highest speedup for 16 threads that are run on 32 cores on $2^{26}$ sized vector.

3. Additional Bonus: Anyone who reaches a speed up of over 16 will gain 25 bonus points for this assignment.

4. The usage of the default "#pragma omp parallel for" is forbidden since its scheduling is implementation dependent. Use scheduling options and chunks’ sizes wisely.

Assumptions (applicable to both parts)

1. The size of the input vector is some power of 2 (2, 4, 8, 16, ...)
2. There is no upper bound on the number of threads. Namely, the program should be written while keeping in mind that the number of threads can be larger than the number of cores in contemporary multicore systems.
3. In Part 2 you can assume the number of threads to be a power of 2 (2, 4, 8, 16, ...)

Hint: Think of the data dependencies between the stages of WHT. Consider how you would best parallelize WHT for 2 threads. What should be changed in order to parallelize for 4 threads? Now generalize it to $2^n$ threads. Is there any price you pay for that generalization?

General implementation hints

1. Your code will do a lot of simple arithmetic operations, so operations that are usually regarded harmless (such as operator +) will affect its performance. Move as many as possible out of the loops.
4. Performance can be significantly affected by compiler optimizations. We will test your code with no optimizations at all. Make sure you do the same (use the Makefile we provide, and if you use Visual Studio on your own machine, use Debug build configuration).
5. Having more than one core use a certain cache line (with at least one core writing to it) adds a significant overhead to memory operations, as the caches must be synchronized. Try to minimize such events.
Performance is significantly affected by the hardware, OS and compiler, especially in multi-threaded programs. If you develop on your own machine, expect different results when running your program on the server.

Part 3 (30 points): Analysis (dry part)

Experiment with the scalability of your solution. Run the fast implementation (part 2), and gather the speedup from the output. All runs for the dry part should be on 16 cores.

**Hint: You might tackle many kinds of anomalies (specifically with 8 threads).**

1. **(8 points)** Draw the graph of speedup as a function of the number of threads. Use 1, 2, 4, 8, 16, 32 and 64 threads, on a vector of size $2^{24}$.
   a. Explain the graph.
   b. Does it correspond with Amdahl's law?
   c. Any reason for the implementation not to scale linearly for multiple threads vs. 1 thread?
   d. Specify 3 reasons why for the given experiment, 32 and 64 threads might actually perform worse than 16. Specify one reason why, for the given experiment, the difference might be insignificant.

2. **(4 points)** Draw the graph of speedup as a function of the vector size. Use 64 threads on vector sizes of $2^{18}$, $2^{20}$, $2^{22}$, $2^{24}$ and $2^{26}$.
   a. Explain the graph.
   b. Does it correspond with Gustafson's law?
   c. On what conditions, would increasing the input size yield higher speedup, and on what conditions would increasing the input size yield lower speedup?

3. **(12 points)** Calculate how many arithmetic operations does your program have in the parallel part, and how many operations it has in the serial part.
   **Note:** arithmetic operations are $+,-,\ast,/$, also $+=\text{ and other similar operations should be referred as arithmetic operations. Assignment (=}\text{ is not an arithmetic operation!}$
   a. How many operations are there in each part? Find an expression depending on $N$=size of vector.
   b. Using the previous calculations, calculate the upper bound of the speedup of your program, using Amdahl’s law and then Gustafson’s law, using 64 CPUs and vector sizes: 218,220,222. Write down your calculations.
   c. Run your program using 8 threads on vectors of sizes 218,220,222 and write down your speedup results. Compare these results to the speedups you calculated in the previous section. Did you get similar results to those you calculated on section b? Are there any major differences?
   d. Try to explain the differences you observed.

4. **(6 points)** Now assume you can modify the hardware to get better results. In this section, we will assume that the parallel portion $A = 0.9$, and that we have a 32 BCE chip.
   a. Why is it more reasonable to consider the area of each core in the course server to be larger than the area of a single a BCE?
   b. Calculate the optimal configuration for our chip, if it were an asymmetric multicore chip.
   c. Calculate the optimal configuration for our chip, if it were a dynamic multicore chip.
Additional Reading

As you see, a main part of this exercise is performance of your code. You are more than welcome to read the next post discussing performances of different operations you most likely use in your code.


**Not all CPU operations are created equal**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost in CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Simple” register-register op (ADD, OR, etc.)</td>
<td>~1</td>
</tr>
<tr>
<td>Memory write</td>
<td>~2</td>
</tr>
<tr>
<td>Bypass delay: switch between integer and floating-point units</td>
<td>~3</td>
</tr>
<tr>
<td>“Right” branch of “if”</td>
<td>~4</td>
</tr>
<tr>
<td>Floating-point/vector addition</td>
<td>~5</td>
</tr>
<tr>
<td>Multiplication (integer/floating)</td>
<td>~6</td>
</tr>
<tr>
<td>Return error and check</td>
<td>~7</td>
</tr>
<tr>
<td>L1 read</td>
<td>~8</td>
</tr>
<tr>
<td>L2 read</td>
<td>~9</td>
</tr>
<tr>
<td>“Wrong” branch of “if” (branch misprediction)</td>
<td>~10</td>
</tr>
<tr>
<td>Floating-point division</td>
<td>~11</td>
</tr>
<tr>
<td>128-bit vector division</td>
<td>~12</td>
</tr>
<tr>
<td>C function direct call</td>
<td>~13</td>
</tr>
<tr>
<td>Integer division</td>
<td>~14</td>
</tr>
<tr>
<td>C function indirect call</td>
<td>~15</td>
</tr>
<tr>
<td>C++ virtual function call</td>
<td>~16</td>
</tr>
<tr>
<td>L3 read</td>
<td>~17</td>
</tr>
<tr>
<td>Main RAM read</td>
<td>~18</td>
</tr>
<tr>
<td>NUMA: different-socket L3 read</td>
<td>~19</td>
</tr>
<tr>
<td>Allocation+deallocation pair (small objects)</td>
<td>~20</td>
</tr>
<tr>
<td>NUMA: different-socket main RAM read</td>
<td>~21</td>
</tr>
<tr>
<td>Kernel call</td>
<td>~22</td>
</tr>
<tr>
<td>Thread context switch (direct costs)</td>
<td>~23</td>
</tr>
<tr>
<td>C++ Exception thrown+caught</td>
<td>~24</td>
</tr>
<tr>
<td>Thread context switch (total costs, including cache invalidation)</td>
<td>~25</td>
</tr>
</tbody>
</table>

**NOTE:** This is not part of the course material and you are not obligated to read it. This is only for your own knowledge. It contains useful material that any programmer should know.
Technical details
You should implement the two functions above in a separate file, called `parallel-walsh.c` in **C99** (not C++). Use the provided `Makefile` and `main.o` file to compile your code on the server (on other environments you’ll have to use your own main). The result will be an executable named `hw2`, which should be run as following:

```
  srun -c <num-of-cores> ./hw2 <vector size> <implementation: 1 for simple, 2 for fast>
```

To externally set the number of threads OpenMP will use, run the following command:

```
  export OMP_NUM_THREADS=<number of threads>
```

Run your code specifying different input sizes. If you fail to obtain "!!!Comparison successful!!!!" results on both implementations, your code has a bug. Fix it and try again. **Do not submit your implementation if it fails to pass this test for inputs up to $2^{24}$ (simple version it's enough to test up to $2^{15}$).** In the case of failure the respective part will be graded with 0 – there is no point in evaluating performance if the code is wrong.

**Note:** Do not experiment with the input sizes resulting in large memory footprint. Your application is allowed to allocate at most 150MB and consume at most 10 CPU seconds continuously. This limitation will be enforced in your account to allow others to work on the same machine. We suggest you avoid starting a new run as long as you see that others also have their tasks running.

Other useful commands:

- `ps -ef | grep hw2` will show you if other students are currently running their programs. Try to do the measurements when no other students are doing theirs.
- `cat /proc/cpuinfo` will show you useful information about the server’s CPU.

Submission
You should submit a zip file that should contain:

1. The `parallel-walsh.c` file with your implementation.
2. A single PDF file named `parallel-walsh.pdf` with the following content –
   - Your names, IDs and email addresses
   - A description of your solutions
   - The answers to the dry part

**Note:** No need to submit printed version of dry part.

The feedback for the dry part and the code will be returned after the submission is evaluated via GR++.