CDP 2009

OpenMP implementation of parallel Walsh-Hadamard transform

Due 14 May 2009, 23.55

This assignment is expected to require 4 student-days for the implementation, 2 student-days for the experiments

May 2, 2009

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, involutory, 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:

$$H_0 = +1$$

$$H_1 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$
\[ H_m = \begin{pmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{pmatrix} \]

The element of Hadamard matrix is 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. You can find more details in Wikipedia: 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 \(O(2^m)\) 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 void simple_parallel_walsh(int n, int* a)**

This procedure performs naive WHT by generating the columns of WHT matrix and multiplying them by the input vector \(a\) of size \(n\). 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.

**Part 2 (50 points) Implement void fast_parallel_walsh(int n, int* a)**

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.

**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 one 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?

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

Experiment with the scalability of your solution.

1. **(10 points - only for Part 2)** Draw the graph of speedup as a function of the number of threads (for 1, 2, 4, 8, 16 threads) when invoked on 2 Mega-integers. In this experiment only make sure that the scheduling policy of OpenMP loop is static (no chunk size specified). Explain the graph. Does it contradict Amdahl’s law? Any reason for the implementation not to scale linearly for 2 threads vs. 1
thread? Can you improve that if that's the case? Note, there will be a competition held between all the participants. The first 3 places will be awarded with bonuses.

2. **(10 points - for both Parts 1 and 2)** Write the analytical formula of the EX-ACT number of integer operations performed by the algorithm to obtain the final result as a function of the input size. We call this number *computation complexity*. Computation complexity should include only the operations required to produce the final result, and ignoring all the implementation-related operations, like loop iterations and alike. For example, we will now develop the formula for the computation complexity of computing dot product of two vectors of size $k$. There are $k$ multiplications and $k - 1$ summations. Thus, the computation complexity is $2k - 1$.

3. **(10 points)** Here we are going to compute the efficiency of the parallel implementation (as opposed to the efficiency of the algorithm). The efficiency of the implementation is the measure of the ability of the program to utilize the hardware doing useful things. It is measured in operations/second. On the other hand, the efficiency of the algorithm is the asymptotic measure of the computational complexity as a function of the input/output size. It is measured in operations. In our case we know for sure that the fast version of the algorithm is much more efficient than the simple version. However the question we want to compare how well this more efficient version utilizes the hardware as compared to the hardware utilization of the simple version. This is important since the simple version is typically easier to implement and it may work better for smaller inputs despite the higher asymptotic complexity.

So, compute the ratio between the computation complexity (as computed in 2.) and the time it takes to produce the results for Part 1 and for Part 2. To compute the running time for Part 1 invoke the implementation using two threads on **32 Kilointegers**. For Part 2 invoke it using two threads on **8 Megaintegers**. Compute the respective ratios. Explain the differences in the results for each implementation? Do the same while using only single thread. It may be that you
would have to reduce the input to meet the time constraints, as explained below.

Technical details

You should implement the two functions above in a separate file, called `parallel-walsh.c` in C (not C++). Login via SSH (all the details on the website) to the remote server (all the details provided on the site), upload the ex2.zip archive from the course website to your home directory, and substitute the parallel-walsh.c in ex2 directory with your own file. To test your implementation, use the provided `Makefile` and `run.sh` script. Run it by specifying different input sizes. If you fail to obtain "!!!Comparison successful!!!" result in both cases - your implementation has bug. Fix it and try again. **Do not submit your implementation if it fails to pass this test for inputs up to 16 Megaintegers. In the case of failure the respective part will be graded with 0.**

**Note:** Do not experiment with the input sizes resulting in large memory footprint. Your application is allowed to allocate **at most 150 MBytes and consume at most 100 CPU seconds continuously.** This limitation will be enforced in your account to allow others to work on the same machine. We suggest to avoid starting a new run as long as you see that others also have their tasks running.

Grading policy

Your **implementation** will be graded according to the correctness and performance. As we said, implementations that will not pass the electronic test will get 0 for the respective part. The implementations will not get any grade reduction due to performance if fast parallel version runs at least as fast as our serial version (as provided in the test) in 3 out of 5 invocations of **test-walsh** program on 16Megaintegers of data. Otherwise, the penalty depends on how slow your run is for that input. The penalty is 10/(speedup). Namely, if your parallel version is as twice as slow, you’ll loose 20 points. In general, you are not doing well if you don’t get ANY speedup...

Your **algorithm** will be graded as follows:
1. The parallel algorithm of the fast version will get +5 bonus if cache optimizations are implemented correctly.

2. The parallel algorithm will lose 20 points if written assuming a certain number of threads.

You are encouraged to search the Internet to find the solutions. However, let us make it clear - this homework assignment is for you to understand the concepts. There are no EXACT same implementations since the problem is defined differently from any others. If you did use the external sources, specify them explicitly in your dry submission.

Submission

Please submit to the course box:

- Code (all contained in parallel-walsh.c file). MUST be single file written in C.
- Short description of the algorithms
- Answers to the dry part

Electronic submission should include parallel-walsh.c file with both functions implemented, and the doc/ps/pdf file with the algorithm description and the answers.

FAQ

1. Every time I run I see “used 1 threads” running. The reason

2. Is recursion allowed in OpenMP? Recursion (coupled with nested parallelism) does not work well with OpenMP, and it MUST NOT be used in this homework

3. Is it OK if simple parallel version is not faster than serial or fails on larger inputs because of the running time exceeding 100 seconds? Yes, it is fine. The serial version in the test program implements the fast algorithm, and for slightly larger inputs the parallel version neither will be able to complete within 100
seconds, nor to compete with the serial version because of the high computational complexity of the simple algorithm.

4. Do we have to set the number of threads by omp_set_num_threads or it will be set up externally? You should not set number of threads from within the program. It should be set externally via OMP_NUM_THREADS.

5. What is the expected input size of simple_parallel_walsh to test scalability? Whatever input that takes longer than 10-20 seconds.

6. In all the testing we’re required to work on X Megaintegers. Is it X*4 MB of memory, or it’s X MB of memory? By saying 16 Megaintegers I mean 4*16 Mega-bytes of memory (if sizeof(int) = 4)

7. Our slow version doesn’t make it in 100 seconds for 2 Mega-integers of input. Is it normal? Yes, it is normal. You should be able to process small inputs with this version, and test the scalability as it appears in the FAQ. As long as you consistently pass the correctness tests for different small input sizes, you’re doing fine.

8. Do we have to test speedup versus our parallel version with 1 thread or versus the provided serial version in walsh-transform.o? Versus the provided serial version. For example, if in order to implement the parallel version you had to change the algorithm to allow parallelism, but by that you killed the performance, i.e. your parallel algorithm is inherently slower due to some overheads, comparison with your own version would be too optimistic.

9. In part 3.3 64k is still too slow. Then test for the largest input you can.