Question 1: Manual Vectorization and Reduction Operator

The vectorization with a 128 bit register provided by Intel SSE should yield a performance increase by a factor of 4 for values of data type float and by a factor of 2 for values of data type double. As a division of 128 bits (= 16 bytes) by 4 bytes for 'float' and by 8 bytes for 'double' results in 4 and 2 values being stored and computed simultaneously with every CPU cycle.

The plot in Figure 1 a) shows a 4 times speed-up for increasing number of threads and performs as expected. It is not 100 % clear why such a high variance was caused for 16 and 24 threads as a full euler node (24 cores) was reserved for the computation. Increasing the number of samples could reduce the variance in this case. (Clarified this issue with Michail and he mentioned that I was not necessary to do the computation again with more samples.)

The plot also reveals a performance difference for small (N = 32768) and large (N = 1048576) data sets. The reason for the perfect scaling of the small data set is that it fits in the L1d cache entirely. Thus all the elements can be fetched from this cache. However, the large data set does not fit into the L1d cache entirely. Therefore, the different threads need to load blocks from higher levels of memory when the data is not cached in L1d resulting in a major performance decrease.

The same behaviour is also expected for the 2 way vectorization, but with a 2 times speed-up. As can be seen from Figure 1 b) the speed-up for the large data set is as expected, whereas the results for the small data set are not as expected – perfect scaling should be observed here as well. As function sse_red_2() is only a slight adaptation from function sse_red_4() – the float operations offered by SSE are replaced with double operations – it should be correct so that no further inspections were made at this point.

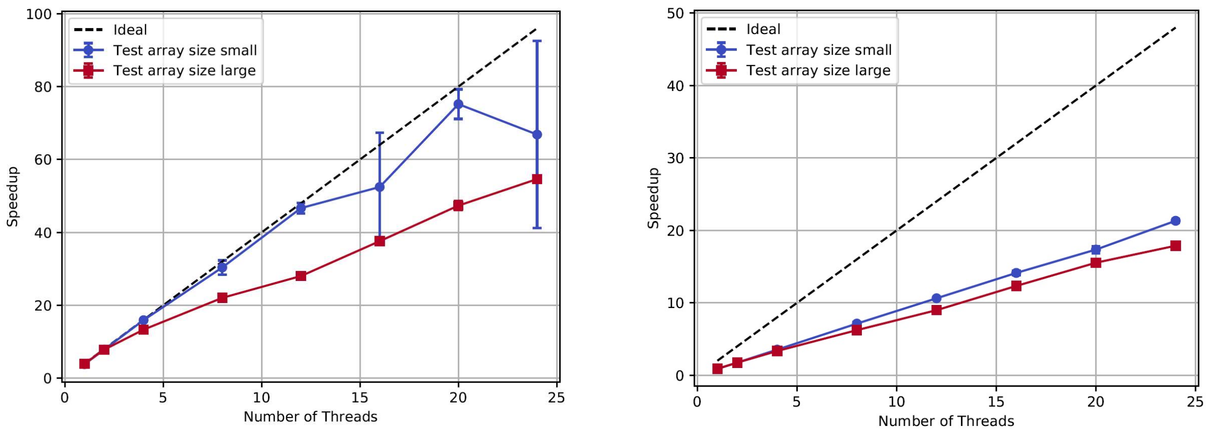

Figure 1. Performance analysis of SIMD vectorized code in combination with thread level performance exploitation. Speedup is mapped against the number of threads used for the computation. **(a)** 4 way SIMD vectorization (32 bit precision data) and **(b)** 2 way SIMD vectorization (64 bit precision data).

```c
static inline float sse_red_4(const float* const ary, const size_t N)
{
    ////////////////////////////////////////////////////////////////////////
    // TODO: Write your vectorized reduction kernel here using intrinsics from
    // the xmmintrin.h header above.  The Intel intrinsics guide is a helpful
    // reference: https://software.intel.com/sites/landingpage/IntrinsicsGuide
    ////////////////////////////////////////////////////////////////////////

    // Iteration variables.
    unsigned int i;
    // Return variable, sum of the array.
    float sum = 0;
    // Iteration size.
    const int simd_width = 16/sizeof(float);

    #pragma omp parallel reduction(+:sum)
    {
        // Create storage for the result.
        float *r;
        posix_memalign((void **) &r, 16, 4*sizeof(float));
        // Initialize allocated memory to zero.
        for (int i = 0; i < 4; i++)
            r[i] = 0;

        // Loop over the vector and sum 4 elements with every CPU cycle.
        #pragma omp for nowait
        for (i = 0; i < N; i+=simd_width) {
            // Save elements for addition with regards to SSE.
            const __m128 a = _mm_load_ps(ary + i);
            //const __m128 b = _mm_load_ps(ary + i + simd_width);
            const __m128 b = _mm_load_ps(r);
            // Add elements together.
            // Store the elements in the 4 slots provided by r.
            _mm_store_ps(r, _mm_add_ps(a, b));
        }

        // Add the final 4 elements.
        for (int i = 0; i < 4; i++)
            sum += r[i];

        free(r);
    }

    return sum; // the function returns the summation of all elemnts in ary
}
```

Figure 2. Function to vectorize the summation of values of data type ‘float’. 4 SIMD lanes can be used with __m128.

```
static inline double sse_red_2(const double* const ary, const size_t N)
{
    ////////////////////////////////////////////////////////////////////////////
    // TODO: Write your vectorized reduction kernel for doubles (2-way SIMD)
    // here.  Note this code is very similar to what you do in sse_red_4 for
    // the 4-way SIMD case (floats)
    ////////////////////////////////////////////////////////////////////////////

    // Iteration variable.
    unsigned int i;
    // Return variable, sum of the array.
    double sum = 0;
    // Iteration size.
    const int simd_width = 16/sizeof(double);

    // Create storage for the simd result.
    double *r;
    posix_memalign((void **) &r, 16, 2*sizeof(double));
    // Set values of r to zero.
    for (int i = 0; i < 2; i++)
        r[i] = 0;

    // Loop over the vector and sum 2 elements with every CPU cycle.

    for (i = 0; i < N; i+=simd_width) {
        // Save the elements for addition with regards to SSE.
        const __m128d a = _mm_load_pd(ary + i);
        const __m128d b = _mm_load_pd(r);
        // Add elements together.
        // Store the elemnts in the 2 slots provided by r.
        _mm_store_pd(r, _mm_add_pd(a, b));
    }

    // Add the final two elements.
    sum = r[0] + r[1];

    free(r);

    return sum; // the function returns the summation of all elemnts in ary
}
```

Figure 3. Function to vectorize the summation of values of data type 'double'. 2 SIMD lanes can be used with __m128.

```
template <typename T>
void benchmark_omp(const size_t N, T(*func)(const T* const, const size_t),\
                   const size_t nthreads, const string test_name)
{
    T * ary;
    // total length of test array is nthreads*N
    posix_memalign((void**)&ary, 16, nthreads*N*sizeof(T));
    typedef chrono::steady_clock Clock;

    // initialize data
    //////////////////////////////////////////////////////////////////////////
    // TODO: Initialize the array 'ary' using the 'initialize' function defined
    // above.
    //////////////////////////////////////////////////////////////////////////
    #pragma omp parallel
    {
        // Init array parts with different threads.
        const size_t tid = omp_get_thread_num();
        initialize(ary+N*tid, N, tid);
    }

    // reference (sequential)
    T res_gold = gold_red(ary, nthreads*N); // warm-up
    auto t1 = Clock::now();
    // collect 10 samples.  Note: we are primarily interested in the
    // performance here, not the result
    for (int i = 0; i < 10; ++i)
        res_gold += gold_red(ary, nthreads*N);
    auto t2 = Clock::now();
    const double t_gold = chrono::duration_cast<chrono::nanoseconds>(t2 - t1).count();
    //printf("Serial - t = %f\n", t_gold);

    // test
    T gres = 0.0;    // result
    double gt = 0.0; // time

    //////////////////////////////////////////////////////////////////////////
    // TODO: Write the benchmarking code for the test kernel 'func' here.  See
    // the serial implementation 'benchmark_serial' above to get an idea.
    //////////////////////////////////////////////////////////////////////////
```

Figure 4. Benchmark OMP first part of the code.

```cpp
    #pragma omp parallel num_threads(nthreads)
    {
        const size_t tid = omp_get_thread_num();

        T res = (*func)(ary+tid*N, N); // warm-up (per thread)

        auto tt1 = Clock::now();
        // Work on 10 samples.
        for (int i = 0; i < 10; i++)
        res += (*func)(ary+tid*N, N);
        auto tt2 = Clock::now();
        const double t = chrono::duration_cast<chrono::nanoseconds>(tt2-tt1).count();
        //printf("tid = %lu - t = %f\n", tid, t);
        #pragma omp critical
        {
            // Compute the total sum (sum of all the thread results).
            gres += res;
            // Compute used time (take the time of the thread that took longest to finish).
            if (t > gt)
                gt = t;
        }
    }

    // Report
    cout << test_name << ": got " << nthreads << " threads" << endl;
    cout << "  Data type size:      " << sizeof(T) << " byte" << endl;
    cout << "  Number of elements: " << nthreads*N << endl;
    cout << "  Absolute error:      " << abs(gres-res_gold)  << endl;
    cout << "  Relative error:      " << abs((gres-res_gold)/res_gold)  << endl;
    cout << "  Speedup:             " << t_gold/gt << endl;

    // clean up
    free(ary);
}
```

Figure 5. Benchmark OMP second part of the code.

```make
# File       : Makefile
# Description: Compile targets
# Copyright 2020 ETH Zurich. All Rights Reserved.
CXX ?= g++
########################################################################
# TODO: Fill in the compiler flags.  Make sure you compile SSE code
CXXFLAGS = -fopenmp -msse -msse2 -O3 -Wall
########################################################################

# debug=true --> use assertions for debugging
debug ?= false

.PHONY: clean measurement

ifeq ("$(debug)", "false")
########################################################################
# TODO: You may want to add flags here?
CXXFLAGS += -DNDEBUG
########################################################################
else
CXXFLAGS += -O0 -g
endif

########################################################################
# TODO: Write the main make target 'vec_red'.  This target will compile your
# code using the flags you set in CXXFLAGS.  Be sure that the generated
# executable is named 'vec_red' in order to use the BASH script
# measure_speedup.sh later on.
vec_red: vectorized_reduction.cpp
	$(CXX) -std=c++11 $(CXXFLAGS) vectorized_reduction.cpp -o vec_red
########################################################################

# You can submit a job on euler using this target.  It depends on the 'vec_red'
# target, that is, it will compile the code if there are changes prior to
# submitting the job.  It will then submit the contents of the script
# measure_speedup.sh
measurement: vec_red
	bsub -W 00:15 \
		-n 24 -R fullnode -R 'rusage[scratch=16]' < measure_speedup.sh

clean:
	rm -f vec_red
```

Figure 6. Makefile to compile the code.

Question 3: Implementing a distributed reduction

The task of this question was to implement an analytic computation of the sum of the given vector. The analytic solution was obtained using the Gauss trick.

For the second task, the implementation of the sum using MPI functionality, the MPI_Reduce() function was used with the additional MPI_IN_PLACE functionality.

Lastly, the summation was also computed manually using the tree based structure displayed in the figure below. The idea is to split the vector elements into two halves and add the upper half elements to the lower half elements (e.g. the value of the element at index m is summed with the value of the element at index m + N/2). This process is carried out until only 1 element is left, which then holds the final result.

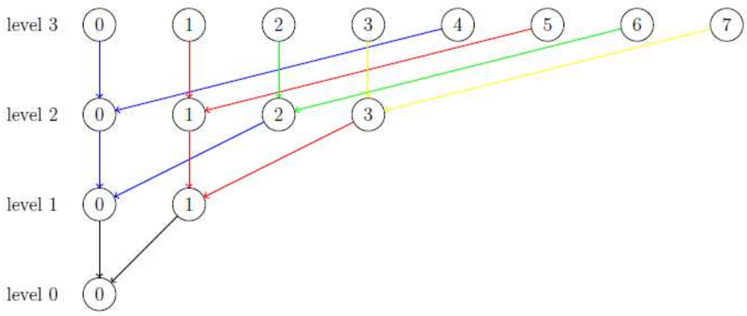

Figure 7. Tree based implementation of the manual function shown in the code snippet below.

```
Your job looked like:

------------------------------------------------------------
# LSBATCH: User input
mpirun ./main
------------------------------------------------------------

Successfully completed.

Resource usage summary:

    CPU time :                                   0.25 sec.
    Max Memory :                                 22 MB
    Average Memory :                             -
    Total Requested Memory :                     4096.00 MB
    Delta Memory :                               4074.00 MB
    Max Swap :                                   -
    Max Processes :                              -
    Max Threads :                                -
    Run time :                                   17 sec.
    Turnaround time :                            11 sec.

The output (if any) follows:

--------------------------------------------------------------------------
No OpenFabrics connection schemes reported that they were able to be
used on a specific port.  As such, the openib BTL (OpenFabrics
support) will be disabled for this port.

  Local host:           eu-ms-019-42
  Local device:         mlx4_0
  Local port:           1
  CPCs attempted:       oob, rdmacm
--------------------------------------------------------------------------
Final result (exact):     499999500000
Final result (MPI):       499999500000
[eu-ms-019-42:22752] 3 more processes have sent help message help-mpi-btl-openib-cpc-base.txt / no cpcs for port
[eu-ms-019-42:22752] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
```

Figure 8. Results of the exact and manual implementation functions. The same result was produced by the reduce_mpi() function that makes use of the a built in MPI reduction function.

Question f) Name two advantages of the tree based approach compared to the naive approach.

1. The tree based approach is faster as not just the root rank is doing all the summation work, but different ranks are working on it. Therefore, the amount of work is shared and this makes this approach faster. Considering my manual implementation, every loop iteration of a rank reduces the number of 'add' operations by a factor of 2! The implementation is thus O(log(n)) instead of O(n).
2. General concept: Faster computation means more results per time and more results per time result in more money for the service provider.

The following two code snippets show the C++ implementation. The code should be relatively self-explanatory so that no further details are provided here.

```cpp
// Copyright 2020 ETH Zurich. All Rights Reserved.

#include <iostream>
#include <iomanip>
#include <random>
#include <cmath>
#include <mpi.h>

inline long exact(const long N){
    // TODO b): Implement the analytical solution.
    // Gauss sum formula.
    long r = (N*(N-1))/2;
    return r;
}

void reduce_mpi(const int rank, long& sum){
    // TODO e): Perform the reduction using blocking collectives.
    MPI_Reduce(rank ? &sum : MPI_IN_PLACE, &sum, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
}

// PRE: size is a power of 2 for simplicity
void reduce_manual(int rank, int size, long& sum){
    // TODO f): Implement a tree based reduction using blocking point-to-point communication.
    long recvSum = 0;
    for (int i = size; i > 1; i/=2) {
        // Ranks above I have already been computed.
        if (rank < i) {
            // Divide ranks in lower and upper half.
            if (rank >= i/2) {
                // Upper half ranks sends their sums to the lower half ranks
                MPI_Ssend(&sum, 1, MPI_LONG, rank-i/2, 420, MPI_COMM_WORLD);
            }
            else {
                // Lower half ranks receive the sums of the upper half ranks.
                MPI_Recv(&recvSum, 1, MPI_LONG, rank+i/2, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                sum += recvSum;
                // Set temporary sum to zero again, results stored must not add up.
                recvSum = 0;
            }
        }
    }
}
```

Figure 9. Implementation of the exact and manual tree reduction functions.

```cpp
int main(int argc, char** argv){
    const long N = 1000000;

    // Initialize MPI
    int rank, size;
    // TODO c): Initialize MPI and obtain the rank and the number of processes (size)
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // -------------------------
    // Perform the local sum:
    // -------------------------
    long sum = 0;

    // Determine work load per rank
    long N_per_rank = N / size;

    // TODO d): Determine the range of the subsum that should be calculated by this rank.
    long N_start = N_per_rank*rank;
    long N_end = N_start + N_per_rank;

    // N_start + (N_start+1) + ... + (N_start+N_per_rank-1)
    for(long i = N_start; i < N_end; ++i){
        sum += i;
    }

    // -------------------------
    // Reduction
    // -------------------------
    //reduce_mpi(rank, sum);
    reduce_manual(rank, size, sum);

    // -------------------------
    // Print the result
    // -------------------------
    if(rank == 0){
        std::cout << std::left << std::setw(25) << "Final result (exact): " << exact(N) << std::endl;
        std::cout << std::left << std::setw(25) << "Final result (MPI): " << sum << std::endl;
    }
    // Finalize MPI
    // TODO c): Finalize MPI
    MPI_Finalize();

    return 0;
}
```

Figure 10. Implementation of the main function. The sections marked with 'TODO' had to be implemented.

Question 4: MPI Bug Hunt

Question a)

The code of this question has 3 different bugs:

- The loop iteration has to be '**< N**' and not '<= N' as this would cause one element being considered that is no longer part of the array.
- When a stream of data is sent to an opened file, this file should be closed after the whole data has been sent to the file. The command 'file.close()' can achieve this and should be included in the code.
- Furthermore, the code, as it is written, does not write the data computed by each rank to the file correctly. Only N numbers of entries will be featured in the file compared to the expected N*size numbers of lines intended in the output file. When one rank has written its data to the file and the next rank starts writing its data to the file it starts again at line 0 and thus overwrites what has been written by the prior rank. The writing process can also be carried out at a random sequence, i.e. the results of the different ranks get mixed up in the output file. In order to solve this problem the results of all ranks are gathered by the root rank in a special data buffer dedicated to this purpose. Once all the results have been gathered, they are written to the file by the root rank

a)

```
const int N = 10000;
double* result = new double[N];
// do a very computationally expensive calculation
// ...

// write the result to a file
std::ofstream file("result.txt");

for(int i = 0; i <= N; ++i){
    file << result[i] << std::endl;
}

delete[] result;
```

Figure 11. Bug hunt problem 4a.

```cpp
#include <fstream>
#include <mpi.h>

int main(int argc, char *argv[])
{
    // Initialize MPI ranks..
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int tag = 420;
    const int N = 20;
    double *result = new double[N];
    // Declarer buffer for writing data to file.
    double *printBuf;

    // Write data to array.
    for (int i = 0; i < N; i++)
        result[i] = (double)i*rank;

    // Write results to file.
    if (rank == 0)
    printBuf = new double[size*N];
    MPI_Gather(result, N, MPI_DOUBLE, printBuf, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    if (rank == 0) {
    std::ofstream file("bug_a_results.txt");
    int lim = size*N;
    for (int i = 0; i < lim; i++)
        file << printBuf[i] << std::endl;
    file.close();
        delete[] printBuf;
    }

    // Free allocated memory.
    delete[] result;

    // Finalize MPI processes.
    MPI_Finalize();

    return 0;
}
```

Figure 12. Proposed solution for problem 4a. The results of the different ranks are gathered in a special buffer ‘printBuf’ by root zero. Root zero writes all the gathered results to the output file.

Question b)

The only thing that was needed to be resolved here was the data type of the MPI_Recv() argument. It has to be changed from MPI_INT to MPI_DOUBLE.

b)
```c
// only 2 ranks: 0, 1
double important_value;

// obtain the important value
// ...

// exchange the value
if(rank == 0)
    MPI_Send(&important_value, 1, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD);
else
    MPI_Send(&important_value, 1, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD);

MPI_Recv(
    &important_value, 1, MPI_INT, MPI_ANY_SOURCE,
    MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE
);

// do other work
```

Figure 13. Bug hunt problem 4b.

```c
#include <mpi.h>

int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    double important_value;

    // Obtain the important_value.
    if (rank == 0)
    important_value = 0;
    else
    important_value = 1.1;

    // Send message over network.
    if (rank == 1)
        MPI_Send(&important_value, 1, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD);
    else
    MPI_Send(&important_value, 1, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD);

    // Receive the message.
    MPI_Recv(&important_value, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // Print received result.
    printf("Rank %d received the double: %f\n", rank, important_value);

    MPI_Finalize();

    return 0;
}
```

Figure 14. Proposed solution for bug hunt problem 4b. The relevant MPI functions are added. The data type of the received data is changed from MPI_INT to MPI_DOUBLE.

Question 4c)

To solve this broadcast related problem the if/else statements have been deleted and replaced by a single line MPI line, the broadcast function MPI_Bcast(). This function is automatically called by all ranks. The sending rank sends the data to all other ranks that retrieve the data by calling this function.

c) What is the output of the following program when run with 1 rank? What if there are 2 ranks? Will the program complete for any number of ranks?

```
MPI_Init(&argc, &argv);

int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int bval;
if (0 == rank)
{
    bval = rank;
    MPI_Bcast(&bval, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
else
{
    MPI_Status stat;
    MPI_Recv(&bval, 1, MPI_INT, 0, rank, MPI_COMM_WORLD, &stat);
}

cout << "[" << rank << "] " << bval << endl;

MPI_Finalize();
return 0;
```

Figure 15. Bug hunt problem 4c.

```
#include <mpi.h>

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int bval;

    // Broadcast a value to ranks.
    if (rank == 0)
    bval = 10;

    MPI_Bcast(&bval, 1, MPI_INT, 0, MPI_COMM_WORLD);

    std::cout << "[" << rank << "] " << bval << std::endl;

    MPI_Finalize();

    return 0;
}
```

Figure 16. Proposed solution for bug hunt problem 4c. The if/else statements were eliminated and replaced by a single line MPI_Bcast() function.