

Set 11 - Exam collections

Issued: December 5, 2014

Hand in: December 12, 2014, 8:00am

In this exercise sheet we propose some exercises that were asked in previous exams. Try to solve them using only the material that we allow during the exam (see lecture homepage) to train yourself.

For this week you will not receive any hint, but you are welcome to ask all your questions when we discuss the solutions.

Disclaimer: Some of the questions have been simplified to fit the topic of this year lecture.

Question 1: Performance of Cell Lists (HS 2013)

- a) Assume you are working in a 2D computational domain with $(x, y) \in [0, 1]^2$ and your pairwise interactions have a cutoff of 0.08. What is the maximum number M of cells that you could use?

The cell side's length has to be big enough to fit the cutoff radius: $1.0/0.08 = 12.5$. In this case, 12 boxes fit in one direction. Thus, the answer is $M = 12 \times 12 = 144$ cells.

- b) Assume you have N particles and that the cost (or time) to compute a single pairwise interaction between them is c_{pw} . This means that the time required to compute x pairwise interactions is xc_{pw} . What is the cost t_{naive} to compute the pairwise interactions between all particles? Consider that each particle does not interact with itself and that the interactions are not symmetrical.

Each of the N particles interacts with $N - 1$ other particles. So we have $N(N - 1)$ interactions. The total cost is $t_{naive} = c_{pw}N(N - 1)$.

As a sidenote: If interactions in this system are symmetric (or antisymmetric) we have to do $N(N - 1)/2$ interactions. The cost would be $t_{naive} = c_{pw}N(N - 1)/2$.

- c) If we were to use a cell list with periodic boundary conditions, we would first need to go through all particles to create the cell list and then we would only compute interactions of a particle with particles in its own cell and neighboring ones. Assume that the N particles are nicely distributed over all cells so that each cell contains $N_c = N/M$ particles. Furthermore assume that the cost to put a single particle in a cell when creating the cell list is c_{cl} and the cost to compute a single pairwise interaction between 2 particles is again c_{pw} . What is the cost t_{cl} of creating the cell list and of computing pairwise interactions between all particles within the interaction range using a cell list, considering no self-interactions and no symmetry?

We have to spend N times c_{cl} to setup the cell lists and then we go through N particles and for each we interact with all particles in the same cell plus the eight neighboring cells. The total cost is $t_{cl} = c_{cl}N + c_{pw}N(9N/M - 1)$.

- d) Derive for what number of particles N it is better not (!) to use a cell list using your results from questions 1b and 1c. You can assume $M > 9$ and that we create the cell list at each time-step.

We have to solve for N in the inequality

$$c_{pw}N(N-1) < c_{cl}N + c_{pw}N(9N/M - 1). \quad (1)$$

The results would tell us that is better not to use a cell list when dealing with $N < c_{cl}M/(c_{pw}(M-9))$ particles.

- e) Cell list can be used to effectively handle a cutoff in the Lennard-Jones potential for the N-body problem, and similarly to handle the cutoff in the kernel of the Particle Strength Exchange (PSE) method. The benefit of using a cell list in the PSE method is however larger than in the N-body problem. Why?

In the PSE method, particles do not move. It means that it is sufficient to compute the cell list just once, whereas in the N-body problem the cell list has to be recomputed every few steps since particles are moving.

Question 2: Roofline Model (HS 2012)

Given the following serial GEMM code snippet:

```
1  for (int k=0; k<N; k++)
2      for (int i=0; i<N; i++)
3          for (int j=0; j<N; j++)
4              C[i+j*N] += A[i+k*N]*B[k+j*N];
```

- a) Can you identify potential problems and/or difficulties in the loop structure of the code, in particular regarding parallelization and locality?
- The loop ordering is not ideal for data locality as j (the fastest running index) corresponds to large strides in C and B .
 - Parallelization of the computation with OpenMP cannot be easily done on the outermost loop as it would create race conditions in updating the result of C .

To improve the performance of the GEMM we need to improve locality by making a better use of the caches through blocking of the matrix.

- b) What is the operational intensity (single precision) for a matrix of size N , assuming there is no caching? Count floating point operations and memory accesses. (Tip: You may write on the code or draw a picture.)

The algorithm requires 3 read, 1 write operation and 2 floating point operations for each of the N^3 iterations. Therefore the operational intensity for single precision is

$$\frac{2N^3}{4N^3 \cdot 4} \text{ FLOP/B} = 0.125 \text{ FLOP/B} \quad (2)$$

- c) What is the operational intensity (single precision) for a matrix of size N , assuming an infinite cache size? Count floating point operations and memory accesses. (Tip: You may write on the code or draw a picture.)

The minimum amount of memory accesses is $3N^2$ to read the three matrices, A, B and C, and N^2 to write the result C, while the number of floating point operations remains unvaried.

$$\frac{2N^3}{4N^2 \cdot 4} \text{ FLOP/B} = \frac{N}{8} \text{ FLOP/B} \quad (3)$$

We observe that the larger the matrix order, the higher the operational intensity is and therefore the higher the potential peak performance is.

- d) Complete Figure 1 by adding vertical lines corresponding to the minimum and maximum operational intensities computed in parts b) and c) for $N = 80$.

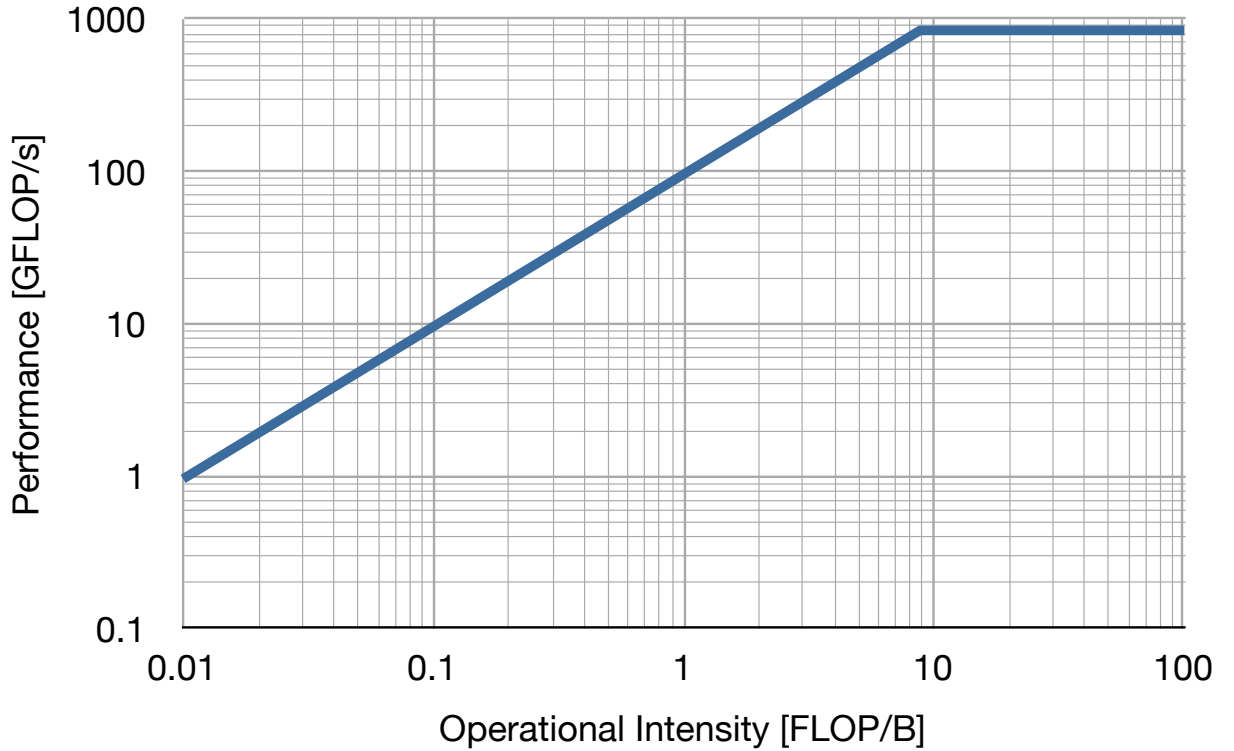


Figure 1: Roofline for Brutus.

- e) What is the maximum performance you can reach with the two operational intensities computed for $N = 80$ given that a Brutus node has a bandwidth of 96 GB/s and a peak single precision floating point performance of 844.8 GFLOP/s? Show your calculations.

Performance estimated from minimum operational intensity:

$$\min(0.125 \text{ FLOP/B} \cdot 96 \text{ GB/s}, 844.8 \text{ GFLOP/s}) = 12 \text{ GFLOP/s} \quad (4)$$

Performance estimated from maximum operational intensity with $N = 80$:

$$\min(100 \text{ FLOP/B} \cdot 96 \text{ GB/s}, 844.8 \text{ GFLOP/s}) = 844.8 \text{ GFLOP/s} \quad (5)$$

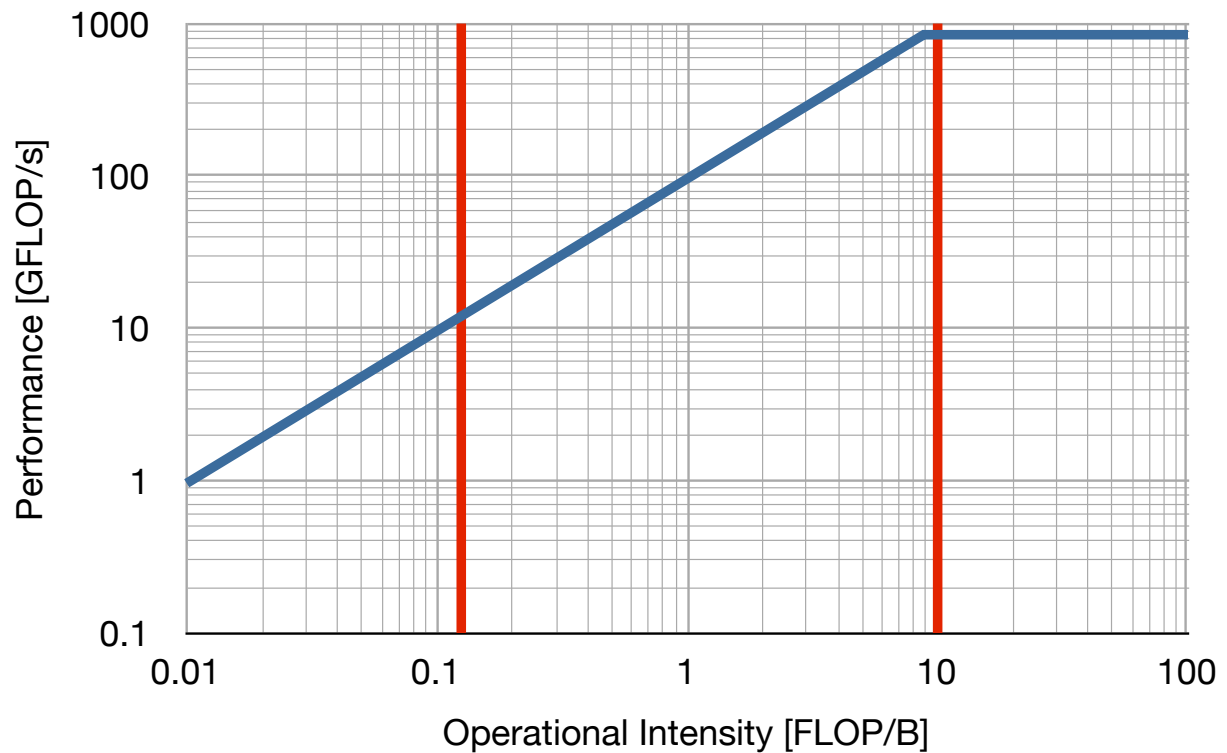


Figure 2: Roofline for Brutus with addition of minimum and maximum operational intensities for GEMM with $N = 80$ (red vertical lines).

Question 3: Parallel sum (HS 2012)

Parallelize the following vector addition:

```

1  for (int i=0; i<N; i++)
2      z[i] = x[i] + y[i];

```

The directory `skeleton_codes/HPCSE12_ParallelSum/` contains a skeleton code for each of the following questions. The solutions should be put directly into these files.

- a) using OpenMP. You can use the following code as a starting point and compile it by calling
- ```
$ make omp
```

---

```

1 // Skeleton code for HPCSE Exam, 18.12.2012
2 // Profs. P. Koumoutsakos and M. Troyer
3 // Question 5a)
4
5 #include <vector>
6 #include <numeric>
7 #include <iostream>
8
9
10 int main(int argc, char** argv)
11 {
12 // vector size

```

```

13 const int N = 1600000;
14
15 // initialize vectors
16 std::vector<float> x(N,-1.2), y(N,3.4), z(N);
17
18
19 // DO THE SUM z = x + y
20 ...
21
22
23 // print result checksum
24 std::cout << std::accumulate(z.begin(), z.end(), 0.)
25 << std::endl;
26 }

```

---

Listing 1: Skeleton code HPCSE12\_ParallelSum/omp.cpp

This part is solved by prefixing the loop with a single omp pragma:

---

```

14 #pragma omp parallel for
15 for(int i = 0; i < N; i++)
16 z[i] = x[i] + y[i];

```

---

- b) using C++11 threads. Create 4 threads such that N is a multiple of the number of threads. You can use the following code as a starting point and compile it by calling

\$ make threads

---

```

1 // Skeleton code for HPCSE Exam, 18.12.2012
2 // Profs. P. Koumoutsakos and M. Troyer
3 // Question 5b)
4
5 #include <vector>
6 #include <numeric>
7 #include <iostream>
8 #include <thread>
9
10
11 int main(int argc, char** argv)
12 {
13 // vector size
14 const int N = 1600000;
15
16 // initialize vectors
17 std::vector<float> x(N,-1.2), y(N,3.4), z(N);
18
19
20 // DO THE SUM z = x + y using 4 threads
21 ...

```

```

22
23
24 // print result checksum
25 std::cout << std::accumulate(z.begin(), z.end(), 0.)
 << std::endl;
26 }

```

---

Listing 2: Skeleton code HPCSE12\_ParallelSum/threads.cpp

Here we need to create 4 threads, let each of them run over one quarter of the vector and join them in the end:

---

```

15 int num_threads = 4;
16 int chunk = N / num_threads;
17 std::vector<std::thread> threads;
18 for(int t = 0; t < num_threads; ++t)
19 {
20 threads.emplace_back([&,t] {
21 for(int i = t*chunk; i < (t+1)*chunk; i++)
22 z[i] = x[i] + y[i];
23 });
24 }
25
26 for(std::thread& t : threads)
27 t.join();

```

---

- c) using MPI. You may assume that N is a multiple of the number of processes. In order to compile and run MPI code, you will first need to load the corresponding module:

```
$ module load mpi/mpich-x86_64
```

You can use the following code as a starting point, compile and run it by calling

```
$ make mpi
```

```
$ mpirun -np 4 ./mpi
```

---

```

1 // Skeleton code for HPCSE Exam, 18.12.2012
2 // Profs. P. Koumoutsakos and M. Troyer
3 // Question 5d)
4
5 #include <vector>
6 #include <numeric>
7 #include <cassert>
8 #include <mpi.h>
9 #include <iostream>
10
11 int main(int argc, char** argv)
12 {
13 // vector size
14 const int N = 1600000;
15 int num_processes, rank;

```

```

16
17 // SET UP COMMUNICATION.
18 // DETERMINE num_processes AND OUR rank
19 ...
20
21
22 // initialize local parts of the vectors and do the
 sum z = x + y
23 assert(N % num_processes == 0);
24 int nlocal = N / num_processes;
25 std::vector<float> x(nlocal,-1.2), y(nlocal,3.4), z(
 nlocal);
26 for(int i = 0; i < nlocal; i++)
27 z[i] = x[i] + y[i];
28
29 if(rank == 0)
30 {
31 std::vector<float> fullz(N);
32
33
34 // COLLECT ALL PARTS INTO fullz
35 ...
36
37
38 // print result checksum
39 std::cout << std::accumulate(fullz.begin(), fullz
 .end(), 0.) << std::endl;
40 }
41 else
42 {
43 // SEND LOCAL PART z TO ROOT PROCESS
44 ...
45 }
46
47 // CLEAN UP
48 ...
49 }

```

---

Listing 3: Skeleton code HPCSE12\_ParallelSum/mpi.cpp

The first missing piece is the standard MPI initialization sequence

---

```

15 MPI_Init(&argc, &argv);
16 MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
17 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

```

---

next is an MPI\_Gather collecting the full result vector on the root process

---

```

29 MPI_Gather(&z[0], nlocal, MPI_FLOAT, &fullz[0],
 nlocal, MPI_FLOAT, 0, MPI_COMM_WORLD);

```

---

---

with the corresponding call on the other processes

---

```
37 MPI_Gather(&z[0], nlocal, MPI_FLOAT, NULL, 0,
 MPI_FLOAT, 0, MPI_COMM_WORLD);
```

---

and finally the usual

---

```
41 MPI_Finalize();
```

---

## Summary

Summarize your answers, results and plots into a PDF document. Furthermore, elucidate the main structure of the code and report possible code details that are relevant in terms of accuracy or performance. Send the PDF document and source code to your assigned teaching assistant.