



F. Wermelinger
Office: Pierce Hall 211

Homework 4

Scaling analysis, Amdahl's Law, processor pipelineing, vectorization
and hybrid MPI+OpenMP

Issued: March 28, 2023 Due: April 18, 2023 11:59pm

Note this is a 3 week exercise. *Do not procrastinate the work.*

Please keep in mind that when the cluster is under high load with many queued jobs, your submitted job(s) will not execute right away. Depending on the cluster load, your job may execute only in a couple of hours (resources are limited). *Make sure you plan ahead.*

IMPORTANT: please do not include any data files, images or other binary objects from Problem 5 in your Gradescope submission zip!

Submission instructions: please see the homework submission section in the syllabus.^a

^a<https://harvard-iacs.github.io/2023-CS205/pages/syllabus.html#homework-submission>

Problem 1: Weak Scaling Analysis (20 points)

An n -body solver describes the evolution of n inertialess particles governed by

$$\frac{dx_i}{dt} = \sum_{j=1}^n F(x_i, x_j) \quad i = 1, \dots, n \quad (1)$$

such that the number of operations at one time step is proportional to n^2 .

a) 5 points

Given the execution time depending on n and the number of processors p , compute four values for the *weak efficiency* $E_p = E_p(n, p)$ following a weak scaling analysis. Write your results in a table with four rows and four columns, one column for n , one for p , one for the time $T_p(n, p)$ and one for your result E_p . Create a plot with p on the abscissa and E_p on the ordinate. Use the data for $n = 10$ and $p = 1$ as reference for T_s . Do not forget to label the axes.

p	runtime [s]				
	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
1	12.0	45.0	118.0	198.0	300.0
2	6.5	30.0	65.0	105.0	155.0
3	4.5	15.0	45.0	84.0	115.0
4	3.5	13.0	32.0	52.0	70.0
5	3.0	8.0	25.0	42.0	60.0
9	1.5	6.0	15.0	25.0	40.0
16	1.0	4.0	10.0	18.0	28.0

b) 15 points

Suppose the solver is parallelized with MPI. Consider two algorithms. Their execution time depends on a constant β .

Algorithm 1 gathers all particles on all ranks with `MPI_Allgather` spending time $\beta n \log_2 p$ and computes the forces spending time $\frac{n^2}{p}$.

Algorithm 2 consists of p passes. All passes involve computation and the first $p - 1$ passes involve communication. The computation takes time $\frac{n^2}{p^2}$ per pass. The communication is non-blocking (overlapped with computation) and takes time βn per pass. Assume the passes are separated by `MPI_Barrier` with negligible overhead.

For each algorithm,

- compute the total execution time $T_p = T(n, p)$,
- compute the weak scaling efficiency $E_p = E(p)$ with $T_p(1, 1)$ as reference,
- for $\beta = \frac{1}{8}$, plot E_p as a function of p and report all p for which $E_p = 1$. Do not forget to label the axes. Use one plot for both algorithms.

Which algorithm has higher weak efficiency E_p for $p \rightarrow \infty$?

Hint: Identify the relationship between n and p and use it to substitute n in your expressions.

Problem 2: Amdahl's Law (20 points)

Amdahl's Law expresses the effectiveness of improving the performance of a particular part of a system for which the size of the system (associated work) *does not change*. We denote the *serial fraction* of the system as f and $1 - f$ relates to the *improvable* (parallel) fraction of the code. Our goal is to improve the performance of the parallel fraction by a factor of p . If our system without optimization takes time T_s to run, we expect the improved time T_p to be in the order of $\frac{1-f}{p}T_s$. Using Amdahl's Law, the improved time can be written as

$$T_p = \left(f + \frac{1-f}{p} \right) T_s. \quad (2)$$

Because the *size of the system does not change*, we can equate the expected *speedup* for p processors as $S_p = T_s/T_p$, or

$$S_p = \frac{1}{f + \frac{1-f}{p}}. \quad (3)$$

Note that the formulation is very general. We can map this formalism to a computer code by denoting that $1 - f$ corresponds to a parallel fraction of the code that can be improved using p parallel processors. For example, if $1 - f$ corresponds to 70 % for a particular code and we want to improve this part by using $p = 4$ cores, Amdahl's Law tells us that we can expect a net speedup for the whole system of $2.1\times$. Even though we have improved a rather large fraction of the code, we only get a 50 % efficiency for our improvement.

a) 6 points

Suppose you live in Worcester, MA, and you are studying Computational Science at Harvard. The distance between the two cities is 55 miles and you are required to take a 90 minutes bus ride every day (in any direction). While on the bus today, you read in the news that a new 36 mile train route from Framingham (which is one of the bus stops) to Allston is about to open with trains that can average 70 mph.

i) 4 points

Assume that you can switch from the bus to the train (and vice versa) without waiting time. What is the net speedup you will get from this announcement?

ii) 2 points

How much faster could you get to Allston if the new trains were traveling at the speed of light?

b) 8 points

You are working on a science project and you are responsible for the simulation code. Your advisor asks you to work on the code performance to run your simulations $8\times$ faster. From a previous analysis you know that you can potentially improve 90 % of the code and that it does not benefit much from multiple hardware threads on the compute cores (hyper threading). Your group currently has access to a single node on a compute cluster, where each node is a NUMA architecture with *two sockets* and a 12 core CPU on each socket.

i) 3 points

How many cores do you need to reach your performance goal?

ii) **4 points**

If you need to buy new nodes, you can only buy them in the configuration described above. How many nodes do you need and what is the maximum speedup you can get from your groups investment?

iii) **1 points**

You have informed your supervisor about the amount of resources you need to achieve the performance goal. Your supervisor is asking you whether you could use more still. Would it make sense to buy more nodes?

Hint: Assume compute nodes are priced at \$12'000 per node.

c) **6 points**

You wrote a program which is composed of *three* parts executed consecutively:

1. A part which you *cannot parallelize*, responsible for a fraction $f_1 = 0.01$ of the total running time.
2. A part which you *can parallelize with only 2 processors*. This part is responsible for $f_2 = 0.04$ of the time.
3. A part that can be parallelized with all processors, occupying the remaining time of the program execution.

According to the above information, what maximum speedup can you achieve, if you had no limitations on the number of processors p ? How many processors do you need to obtain a speedup of *at least* 8?

Problem 3: Processor Pipelining (20 points)

a) 8 points

In the lecture we have looked at the pipeline block diagram of the Intel Broadwell CPU. You can find this diagram at this link [https://en.wikichip.org/wiki/intel/microarchitectures/broadwell_\(client\)](https://en.wikichip.org/wiki/intel/microarchitectures/broadwell_(client))

i) 2 points

Use this diagram to determine the number of data loads and data stores you can perform in the same cycle.

ii) 6 points

With the help of the lecture slides, explain why the architecture designers have chosen the particular ratio of data load and data store instructions.

b) 12 points

Suppose we have the following functional units to process instructions with latencies:

IF	(Instruction Fetch)	2 ns
ID	(Instruction Decode)	2 ns
EX	(Execute Instruction)	3 ns
MEM	(Physical Memory Access)	6 ns
WB	(Write Back Result)	2 ns

i) 4 points

If we use these units to build a *non-pipelined* processor, how long does it take to execute a single instruction?

ii) 2 points

If we use these units to build the simple 5-stage pipelined processor discussed in the lecture, what is the shortest possible cycle time for the processor, i. e., the time for executing a single stage?

Hint: You are the hardware architect and you prefer simplicity over complexity in your design.

iii) 4 points

How long does it take to execute at least N instructions in a *full* pipeline? Assume the pipeline is empty at the beginning.

iv) 2 points

What is the speedup of this pipelined processor over the non-pipelined implementation? Assume a large number of instructions that do not cause pipeline stalls.

Problem 4: Manual Vectorization (20 points)

We are interested in optimizing the performance of the following compute kernel

$$r = \sum_{i=1}^n a_i, \quad (4)$$

where a_i are the elements of a vector $\mathbf{a} \in \mathbb{R}^n$ and $r \in \mathbb{R}$ is the result of reducing $\mathbf{a}$ by summing up its elements. We aim at improving the performance of this kernel by exploiting the data-level parallelism (DLP) of $\mathbf{a}$ using the SIMD capabilities available on the CPU. We will utilize the streaming SIMD extensions (SSE) to *manually* vectorize the kernel shown in equation 4. We want to test our implementation for single precision data (32-bit) and double precision data (64-bit). You can assume that n is an integer multiple of the largest number of SIMD lanes supported by SSE for the given data type.

Work in directory: `./code/p4`

a) 14 points

Implement vectorized code for the reduction kernel in equation 4. A baseline version `gold_red` has already been implemented and can be used as a reference. You are asked to work on the items marked with “TODO” in the source file `reduction_kernels.h`. A guide with a possible workflow can be found in the `README.md` file within the source directory. Furthermore, the Intel intrinsics guide is a useful reference for this task, see <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>.

b) 6 points

You can measure the performance of your vectorized code by running

```
$ make measurement
```

The job will run on one Intel Xeon E5-2683 v4 CPU on the compute cluster using OpenMP to run on all of the 16 cores. It will create two png files, one for 32-bit precision and another for 64-bit precision results, each with speedup plots for a small n and another with a large n .

i) 2 points

What is the maximum speedup you expect for 32-bit precision and 64-bit precision data?

ii) 4 points

Study the speedup plots in the generated png files and clearly explain the reason for differences you may observe for a small vector size n and a large vector size n (if there are any).

Problem 5: Hybrid Gray-Scott system with MPI and OpenMP (20 points)

Nature reveals many beautiful patterns. Examples where you may find such patterns are the skin of exotic fish, the stripes on zebras, your finger prints, the spots on butterflies or the ripples on a sand dune. In the following problem we are going to study such pattern formation which is governed by reaction and diffusion processes.

The formation of a pattern requires at least two species that interact with each other by means of *reaction* and *diffusion*. We are restricting ourselves to two species U and V in a 2D square domain. The concentration q of a quantity Q describes the amount of that quantity *per volume of mixture*. In the case of our two species, a concentration $u(x, y, t) = 1$ and $v(x, y, t) = 0$ would mean that only species U is present at the particular coordinates x, y and t . Differences in chemical and concentration potentials between the two species will cause them to react and diffuse with each other. The dynamics of this process is governed by the Gray-Scott¹ equations

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u - uv^2 + F(1 - u), \quad (5)$$

$$\frac{\partial v}{\partial t} = D_v \nabla^2 v + uv^2 - (F + \kappa)v. \quad (6)$$

These are two *coupled* PDEs that describe the spatial and temporal evolution of our unknown species u and v . F and κ are two constant parameters called feed and death rate, respectively. Note that ∇^2 is a common notation for the Laplacian operator in coordinate free form. For our 2D problem in Cartesian coordinates it simply is $(\nabla^2) = (\partial^2/\partial x^2 + \partial^2/\partial y^2)$. The terms on the left side of equations 5 to 6 describe the *rate of change with respect to time* for each species. The first term on the right side describes the *diffusion* of the species, the second term uv^2 is a non-linear *reaction* term that couples the two equations together and the last terms on the right side describe the *growth and decay* of the species given the feed rate F and death rate κ .

Work in directory: `./code/p5`

a) 6 points

The handout includes an MPI code that implements the Gray-Scott system in equation 5 and equation 6 using second order centered finite differences and an explicit first order time integrator. Unfortunately, this code only works correctly when run with a single rank. You will write code to fix this in task 5b. Before you do that, you want to enable OpenMP in your current MPI code. Extend the handout with the necessary code to support threads in MPI and parallelize the main work loop using OpenMP. You need to introduce the main modifications in the file `GrayScott.h`, study the private method `update_()` at the bottom of the file. The logic of integrating the Gray-Scott system is implemented in the public method `integrate()` and the system is initialized with the private method `initialize_()`. You may need to modify other files as well. You should observe a noticeable wall time improvement after enabling multiple cores.

¹P. Gray and S. K. Scott, "Chemical Oscillations and Instabilities: Non-linear Chemical Kinetics", v. 21 of International Series of Monographs on Chemistry, 1994.

Hint: Do not forget about thread support in MPI.

Hint: Your code must be NUMA aware.

b) **14 points**

As mentioned above, the current MPI implementation only supports one rank. In this task you must implement *asynchronous* MPI communication that exchanges the ghost cells at the four faces of each MPI subdomain. The numerical scheme requires one ghost cell on the subdomain boundaries with periodic physical domain boundaries. You can find the implementation of the Gray-Scott constructor at the top in the file `GrayScott.cpp` where a Cartesian MPI communicator with *periodic boundaries* is used. See the example of the 1D diffusion equation in the lecture 13 slides as well.

You will need to work in `GrayScott.h` exclusively for this task. In particular the private methods `update_async_()`, `pack_()` and `unpack_()` at the bottom. The main algorithm you need to realize in `update_async_()` is

1. Pack the ghost cells into a *data structure of your choice*. The requirement is that you pack the ghosts for both species u and v into the same message buffer and one message is sent and received from each of the neighboring ranks. Implement this in the `pack_()` method.
2. Initiate *asynchronous* communication. Based on the requirement above, you need four send/receive pairs. You can get the rank IDs of your neighbors using the `neighbor_` array together with the Neighbor enum (see the MPI section of the private data in `GrayScott.h`). For example, to get the rank ID of the neighbor at the top you can use `neighbor_[Top]`.
3. Update the interior domain while communication is in progress to enable compute/transfer overlap. The loop should again be parallelized with OpenMP.
4. Wait for MPI operations to complete.
5. Unpack the received message data (ghosts) into the fields for species u and v . Implement this in the method `unpack_()`.
6. Update the nodes on the boundary.

You should implement a hybrid OpenMP/MPI code here as well, similar to the previous task 5a. The handout implementation provides convenience access methods for the species u and v with the methods `u(i, j)` and `v(i, j)` respectively. The indices i and j are in the interval $[0, N)$, where N is the total number of nodes in the square subdomain of each rank. Indices can be negative as well to access ghost cells. For example, `u(-1, j)` would index the j -th node in the top row of ghosts. See also the `update_()` method for more examples.

You can use the provided `submit_check.sh` job script to submit an MPI job with 9 ranks (use 4 cores with 4 ranks when you test and implement, think of others too). Please submit this script from a compute node or a VSCode instance and *submit only the generated *.out file, no data files or other binary objects*. The script will generate a contour image of species v and also compute the L_2 -error norm between your implementation and reference data. A correct implementation should result in zero error up to machine precision, based on the provided settings in `main.cpp` (do not change the parameters for this test to be successful).

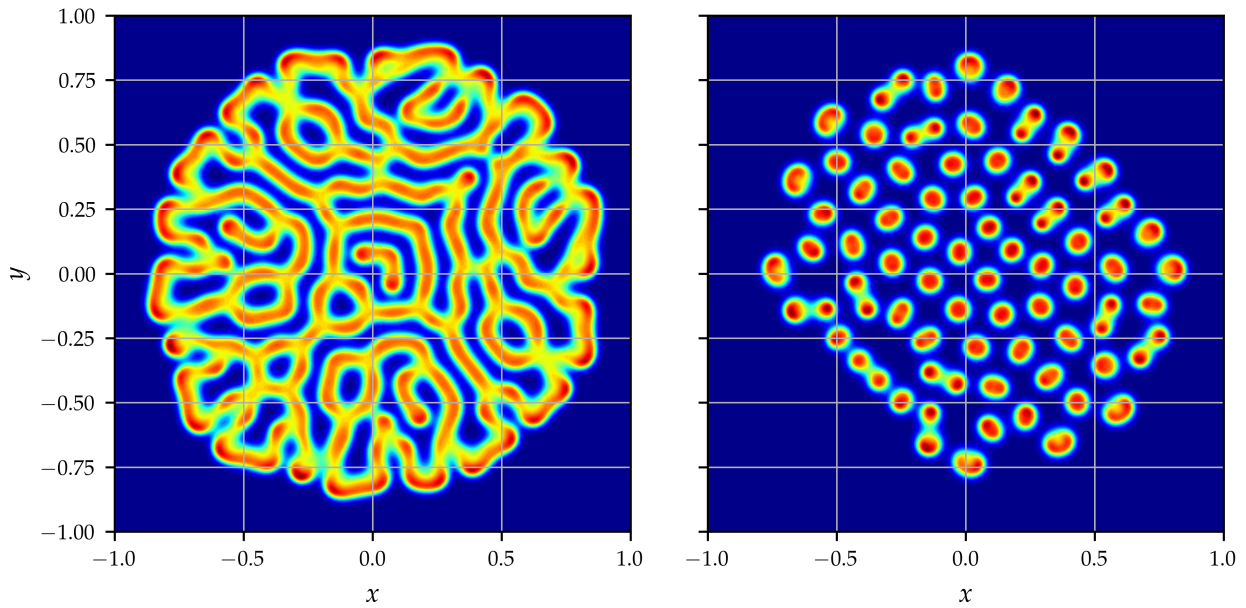


FIGURE 1: Different patterns computed with the Gray-Scott system of equations. The left image shows a “rolls” pattern ($F = 0.04$, $\kappa = 0.06$) and the right image shows a “spots” pattern ($F = 0.025$, $\kappa = 0.06$).

The Gray-Scott system is sensitive to the choice of parameters F and κ and will generate different patterns depending on them. The default parameter in the handout are $F = 0.04$ and $\kappa = 0.06$. This will generate a “rolls” pattern as seen in figure 1. If you change the feed rate to $F = 0.025$ you will observe a “spots” pattern (see [main.cpp](#)). You should recover these patterns with your implementation and they must be *symmetric* with respect to the diagonal as shown in figure 1. The L_2 -error check mentioned above is based on the default “rolls” setting. You can download movies for these two patterns to see how they evolve over time from the compute cluster, see

- `~/shared_data/hw4/p5/movies/rolls.mp4`
- `~/shared_data/hw4/p5/movies/spots.mp4`