Concepts and Models for Parallel and Data-centric Programming, Summer 2021

Exercise 3: Distributed Memory

Task 1. General Questions

Task 1.1. We discussed the BSP model with the Bulk implementation and the PGAS model with the DASH implementation. What are the most important conceptual differences between them?

Task 1.2. Using distributed memory computers also means partitioning data. There are different data distribution schemes available in Bulk and DASH. Explain with examples for which kinds of problems a cyclic distribution is better suited than a block distribution and vice versa.

Task 1.3. Beside cyclic and block distributions, a combination of both patterns, a *blockcyclic* distribution is possible (see 29-DASH-Data-Structures slide 10). How would a distribution function look like for a 1-dimensional array? Give a function definition that matches an array of size n to p processes in a blockcyclic distribution with the following signature:

$$distr : \{0, \ldots, n-1\} \rightarrow \{0, \ldots, p-1\}$$

Task 2. Inner Product Revisited
In the lecture we discussed an inner product BSP algorithm where each processor computes and broadcasts its local intermediate result to *all* other processors such that each processor locally computes the final result after receiving all remote values. Another solution would be to have each processor send its local intermediate result to a designated *root process* which computes the final inner product and broadcasts it to all other processes.
Assuming that you are using a BSP computer with p processors and a vector size of n, compare this modification in terms of BSP costs (see 26-BSP-Cost-Model slides 8 – 12 for reference) by solving the following tasks:

Task 2.1. Adapt the pseudocode algorithm on 25-BSP-Programming slide 12 such that the intermediate results are sent to a *single* root process which computes the final inner product and broadcasts the result back to all other processes. You may use `if` statements to differentiate between root and non-root processes.

Task 2.2. Provide the number of FLOPS $w_i^{(s)}$ of processor s in superstep i in your algorithm. Assume that process 0 has been chosen to be the root process.

Task 2.3. Provide the number of data elements $r_i^{(s)}$ / $t_i^{(s)}$ received / transmitted by process s in superstep i in your adapted algorithm. In addition, determine the h-relation for each superstep. Assume that process 0 has been chosen to be the root process.

Task 2.4. Use the BSP cost formula (26-BSP-Cost-Model slide 10) to calculate the final costs of your adapted algorithm. Compare it to the costs of the lecture algorithm (26-BSP-Cost-Model slide 12). What is remarkable? Explain the result.

Task 2.5. Provide the *total* number of FLOPS W and the total number of data elements received / transmitted R / T for both algorithms and explain the results.

Task 2.6. After solving the previous tasks, think about the expressiveness of the BSP cost model and write down at least one resulting weakness.

parallelism reduces cost -> no parallelism increases cost

Task 3. Stencil Codes in BSP
Stencil codes are a typical class of iterative kernels used in various scenarios (image processing, partial differential equations, ...). The general idea is to perform a seqeuence of timesteps on a given n-dimensional data set. In one timestep, for each data element called "cell" a new value is computed using the data of the neighboring cells.
A typical kernel is a 2D von Neumann stencil working on a matrix, where the value of each element is computed using the top, right, left, and bottom neighbor (see figure).

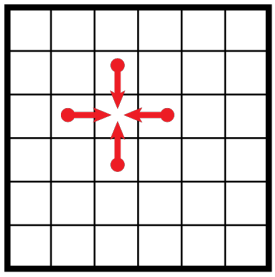

Figure 3.1: 2D stencil, the red arrows indicate the data dependencies of a cell (source: Wikipedia, `https://en.wikipedia.org/wiki/Stencil_code`)

Another simpler case is a 1D stencil working on a plain array, where each cell is recomputed using the neighboring left and right cell: Let $x^0 \in \mathbb{R}^n = (x_0^0, x_1^0, \ldots, x_{n-1}^0)^T$ be an array in the initial timestep 0. In order to compute the k+1-th step, you have to perform the computation

$$x_i^{k+1} = 0.5 \cdot (x_{i-1}^k + x_{i+1}^k) \quad \forall i \in \{1, \ldots, n-2\}.$$

In other words, you take the left and right neighbor value of the previous timestep to compute the new value of a given cell. Note that the outer elements x_0^k and x_{n-1}^k keep their initial value due to a missing left resp. right neighbor. You can assume that the data has already been initialized with some values in the initial timestep.
For a shared memory system, such a stencil code is easy to implement. In a distributed memory system, you have to take care that elements at the borders of the distribution become available at the neighboring processes ("halo" exchange).
In the following, you will design a BSP algorithm that computes the given 1D stenci generalize (optional task) your solution to a 2D stencil.

Block distribution because there occur many data exchange

Task 3.1. Start with the 1D stencil. Decide between a block and cyclic data distribution for the array x. Give the distribution function *distr*. Justify your answer.

Task 3.2. Taking your chosen distribution function into account, write down a pseudocode algorithm that calculates the 1D stencil for k timesteps on p processes with the formula depicted above.

Hint: The easiest way is to let every rank s put its leftmost element to the left neighbor $s-1 \bmod p$ and its rightmost element to the right neighbor $s+1 \bmod p$. After communication is finished, the actual calculation of the next timestep can start. Similar to the inner product algorithm pseudocode, use *global indexing* for simplicity (like the global view of coarrays).

Task 3.3. Calculate the BSP costs of your 1D stencil algorithm depending on the number of iteration steps k by giving the final cost formula.

Task 3.4. *(Optional)* Generalize your algorithm to the 2D von Neumann stencil that works on a square matrix $A = (a_{ij})_{i=0,\ldots,n-1,j=0,\ldots,n-1} \in \mathbb{R}^{n \times n}$. The computation should be

$$a_{ij}^{k+1} = 0.25 \cdot (a_{i-1j}^{k} + a_{ij+1}^{k} + a_{i+1j}^{k} + a_{ij-1}^{k}) \quad \forall\, i,j \in \{1,\ldots,n-2\}$$

as depicted in Figure 3.1. Note again that the outer elements of the matrix will keep their initial value.
The data distribution has been chosen to be as follows: The matrix of size $n \times n$ is split up into p equal-sized submatrices such that each process gets its own block of the matrix on which it calculates the new cell values. Each submatrix contains $\frac{n^2}{p}$ elements and therefore has $n/\sqrt{p}$ rows and columns. For simplicity, assume that n and p are chosen such that $n^2/p \in \mathbb{N}$ and $\sqrt{p} \in \mathbb{N}$. The data distribution function is

$$\begin{aligned} &distr : \{0,\ldots,n-1\} \times \{0,\ldots,n-1\} \to \{0,\ldots,p-1\} \\ &distr(i,j) = \left\lfloor \frac{i}{b} \right\rfloor \cdot \sqrt{p} + \left\lfloor \frac{j}{b} \right\rfloor \text{ with } b = \frac{n}{\sqrt{p}} \quad \forall\, i,j \in \{0,\ldots,n-1\} \end{aligned}$$

You can use this function in your algorithm to determine the processor which owns the matrix element (i,j).
As a last step, determine the BSP costs of your 2D stencil algorithm.

Task 4. K-Means in Bulk
If you have not already set up an account for "Hochleistungsrechnen" in the RWTH Selfservice, please do so using the link `https://www.rwth-aachen.de/selfservice`. Then, send your TIM-ID to `contact@hpc.rwth-aachen.de`.
There are two versions of this exericse: The recommended one is the **Jupyter notebook**, because it solves the exercise in an incremental way and gives you feedback to your code. If you want to use the Jupyter notebook version of this task, then go to Task 4.1. Otherise, go on with Task 4.2.
In the second exercise sheet you implemented k-means clustering using GPUs. We now want to investigate the challenges and achievable performance when running k-means on a distributed memory computer using Bulk. For a detailed description of the k-means algorithm, please have a look at exercise sheet 2. The basic, iterative algorithm was given as follows:

- Initialize the k means randomly.
- Assignment step: each data point is assigned to the nearest mean.
- Update step: each mean is recalculated based on the new assignments.
- Repeat assignment and update steps until algorithm converges.

Task 4.1. Doing the exercise as a Jupyter notebook

1. Follow the documentation to start a desktop environment on a frontend node: `https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/25f576374f984c888bb2a01487fef193` (Remember that you have to be connected to the RWTH network (via VPN) to be able to connect to the cluster.)

2. Copy the exercise tarball from the lecture group directory by running the following command in a shell:

   ```
   $ cp /home/lect0053/exercise3_distributed_memory.tar.gz $HOME
   ```

3. Change into your home directory and unpack the tarball with the command

   ```
   $ cd $HOME && tar xvf exercise3_distributed_memory.tar.gz
   ```

   in the shell and change into the directory

   ```
   $ cd exercise3_distributed_memory
   ```

4. Run the shell script `jupyter-run.sh` to start a Jupyter server:

   ```
   $ ./jupyter-run.sh
   ```

   You might have to wait a few minutes until a node is assigned.

   If successful, you will see a message with a link to the Jupyter server. The link has the format `http://<...>.itc.rwth-aachen.de:<port>/lab`. Copy that link and paste it into a browser **in the graphical cluster session**, e.g., in Firefox. The link will **not** work in your local browser, you have to open the link in the graphical cluster session.

5. You should see a list of files, in particular four different Jupyter notebooks. Click on `01-DataDistribution.ipynb` to start the first Jupyter notebook.

Now, everything is set up and you can go on with solving the task in JupyterHub.

Task 4.2. Doing the exercise on the cluster
You first have to set up the cluster environment. If you have problems following this guide, please post your questions in the Moodle discussion board or write us a mail.

Follow the documentation to set up a shell environment: `https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/25f576374f984c888bb2a01487fef193`

You can also directly login to one of the CLAIX 2018 frontend nodes (login18-[1-4]) via SSH:
`$ ssh -Y login18-2.hpc.itc.rwth-aachen.de`

Copy the exercise tarball from the lecture group directory by running the following command in a shell:
`$ cp /home/lect0053/exercise3_distributed_memory.tar.gz $HOME`

Change into your home directory and unpack the tarball with the command
`$ cd $HOME && tar xvf exercise3_distributed_memory.tar.gz`

and change into the directory
`$ cd exercise3_distributed_memory`

The archive contains:

- `prepare_env.sh`: See below for further explanation.
- `Makefile`: Build file to compile and run the exectuable.
- `kmeans.cpp`: The C++ Bulk code skeleton for Task 4.
- `kmeansjob.sh`: A Slurm batch file allowing to submit a job to the cluster.
- `input`: Contains a small (1000 elements), a large (10^6 elements), and a larger (10^7 elements) 2D input data set.
- `utils`: Contains a visualization script

The file `prepare_env.sh` loads the right compilers and sets up your environment. Before compiling and running your programs, you have to source this file in your shell with the command
`$ source prepare_env.sh`

If you have sent your TIM ID to `contact@hpc.rwth-aachen.de`, you have the possibility to submit to a special lecture project (lect0053) that reserves some compute resources for you for the purpose of completing your exercises. Note that these compute resources are for all students in this lecture. Please use them responsibly!
Open the `kmeans.cpp` source file in an editor. Have look at the source code and the TODOs in the `main` and `kmeans` function. Solve the TODOs in the code in order to implement k-means in Bulk. Some additional hints and notes:

- The data distribution of the points is already implemented. A block distribution has been chosen that evenly distributes the points over the processes.
- The centroids are also already initialized and available as coarrays. Each processor has a local (identical) copy at the beginning.
- Remember that you access the i-th local element of a partitioned array (`points` is a partitioned array) with `points.local(i)`.
- You might want to use the already implemented function `reduce_coarray(coarray, f, size, root)`. This function takes a coarray with local size `size` (at each processor) and performs an element-wise reduction using the specified lambda function f. The reduction results are available at the given `root` rank after execution (see 29-Message-Passing slide 28 for an illustration of an element-wise reduction).
  As an example, `reduce_coarray(coarray, [](int& lhs, int rhs) {lhs += rhs;}, k, 0, world)` performs an element-wise sum reduction on a coarray of size k. The resulting reduction vector of size k is available at the coarray of process 0 when the function returns. Note that the function overrides the local elements of the coarray at process 0 with the results.

- The code is executed on the cluster frontend using `mpirun` (see `Makefile` for the full command). Note that the number of processes (`-np`) you can choose is limited on the frontend. Furthermore, running the program on the frontend is not suited for performance measurement, but only useful for debugging your code.
- Use the slurm job script `kmeansjob.sh` to submit a job on the cluster:

  ```
  $ sbatch kmeansjob.sh
  ```

  By default, the job runs on a single node (with 48 cores). You can change `--nodes` in the script to another value (2, 4, 8, 16) to increase the number of nodes.
- Do some scalability measurements of your solution.

Discussion of this exercise on June 10th, 2021.

2.1 Adapt the pseudocode such that intermediate results are sent to the root processor and the root processor broadcasts to every other processors

######
Superstep 0
######
a_s = 0

for(i = s; i <n; i+=p) do

a_s = a_s + x_i * y_i // compute the local product

2 * (n/p)

for(t:=0;t<p;t++)do

if(t = root processor) do
put α" in P(t);
barrier();

h_0 = r_0 = t_0 = p-1

T_0 = 1/r * 2 * (n/p) + l + g(p-1)

######
Superstep 1
######

if rank == root processor do
a_result = 0
for(t:=0;t<p;t++)do

a_result = a_result + a_t

p

for(t:=0;t<p;t++)do

if(t != root processor) do
put α" in P(t);

h_1 = r_1 = t_1 = p

barrier();

T_1 = 1/r * (p) + l + g(p-1)

T = 2*(1/r * (n/p) + l + g(p-1))+ 1/r * (p)

-> cost of communication and latency is doubbled

Total # Flop, r, t
1. from the slide = 2*n/p + p,
r = t = p-1
2. from the algorithm = 2*n/p + p
r = t = p-1

for(i : 1; i < n-1; i++) do

Idea : every processor has its own rank,
thus, by this number, to limit the extend to
send message or access the global array is possible