Project 1: Matrix redistribution

02616

February 25, 2020

First project in course 02616. A hand-in of a report comprising teams composed of 2-3 members. Please consider these things for your report:

- Describe all variables discussed
- Use log-plots wisely
- Provide a sufficient text for your figure captions; figures + captions should be self-explanatory
- Ensure axes labels on figures
- Ensure one can interpret your figures (do not put too many lines in one plot)
- When using `MatLab` or `matplotlib` to generate figures, consider using ImageMagick to remove surrounding white-space:

  ```
  $> convert fig_in.png -trim fig_out.png
  ```

- For LaTeX *never* use `eqnarray`, use `equation` (single equation) or `align` (for aligned equations) instead)
- Use your co-students to proof read your reports

The project should be handed in (uploaded on CampusNet) by the 15 of March 2019 at 23:59. Both the report (in PDF) and the source code (in zip/tar.gz) file is required uploaded.

1 Matrix distribution for distributed workloads

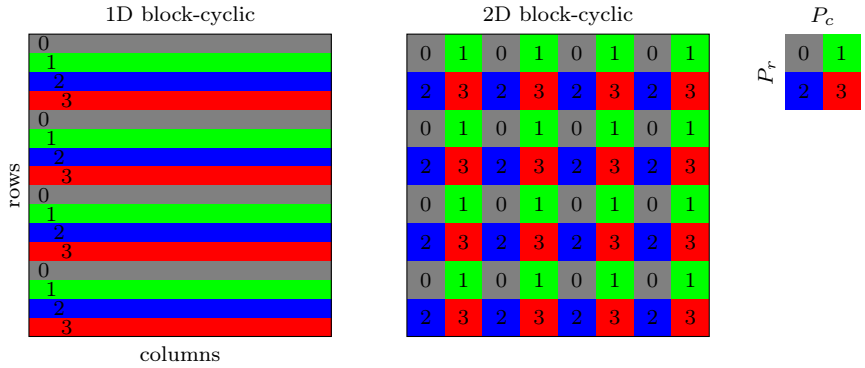
Parallel programming exceeds complexity of serial programs in regards of workload distribution. This forces the developer to think about distribution layout since workload imbalances may occur. This shows it self in non-obvious ways since there may be several *good* data distributions, few *optimal* data distributions and many *bad* data distributions. This is a constant field of investigation since this is also hardware dependent. Even though one may think that the added communication overhead of re-distributing data is a performance hit, it may be that the algorithm leverages the data layout much better.

The above is true for typical matrix operations, matrix-multiplications, decompositions etc.

In this project you will be asked to work on matrix distributions using a 2D block-cyclic scheme.

1.1 Block cyclic distributions

In this project we will focus on benchmarking two popular block-cyclic distributions, the 1D and 2D variants.



1D block-cyclic This distribution function is determined by a single number, the block-size NB. Every processor gets assigned a distributed index in a block-cyclic manner. The above figure shows $P = 4$ processors and $N = P \cdot NB$ matrix dimension. Thus the 1st processor gets the first NB elements, the 2nd processor gets elements from $NB + 1$ to $2NB$ and so on.

2D block-cyclic This distribution function is determined by three(four) numbers, the block-size[1] NB, and number of processors along columns and rows P_c/P_r, respectively. Such a 2D layout intrinsically lets each processor have a Cartesian coordinate p_r, p_c index.

2 Tasks

Important! When solving the below tasks please keep in mind a modular implementation. I.e. try to break up your codes into smaller code segments (functions!). Having a modular code is beneficial to maintain an overview of the code and also when doing benchmarks, since everything can be done in a single program call. It is especially important to implement your index functions in a simple and clean function (or C-preprocessor code). A rule-of-thumb:

a function does one thing, and ONE *thing only!*

We don't want to see everything hard-coded in a single `main`/`program`!

How you initialize the matrices is relatively un-important. Whether you initialize the matrix as a distributed matrix or create row-by-row and distribute is up to you.

- 2D block-cyclic
  - Implement a re-distribution function to transfer a given blocksize NB to another block-size NB', implement several variants, blocking send/recv vs. non-blocking send/recv vs. collective communications. Can you come up with other variants?
  - Benchmark the performance of re-distributions for various initial processor grids/NB and final processor grids/NB' and matrix size N. Comment on the results, it is important to explain *why* the results are as they are!
- **OPTIONAL!** ScaLAPACK provides all you need to implement (and more) through BLACS. You can benchmark against the BLACS implementation, can you beat it?

  Additionally you can use BLACS as a reference implementation to check your re-distribution functions actually does it correctly.

Things to note

When timing your program it is vital that you consider how to time a parallel execution, note that you get different timings per processor, why, and how to handle this?

[1] An additional number could be added so that each block is non-square, i.e. NB and MB determines the rows/column size of each block.

- This is *large scale modelling* so please do benchmarks on at least 60 processors!
- Investigate the first MPI calls. Do you see anything in particular? And if so, how to resolve this?
- Timing is individual per process, consider:
  - Maximum of individual timings
  - Minimum of individual timings

When you start to do testing here are some ideas to increase your throughput.

- Create two executables, one for testing your distribution functions and one for performance analysis (i.e. the latter should *assume* the algorithms are correct!)
- Use pre-processors to add/remove code for various tasks:
  - Only timing
  - **OPTIONAL!** When analysing the performance it may be beneficial to count send/recv to and from individual processors to get a figure of the communication imbalance (don't do this for timings since there is a substantial overhead of this)
  - The index calculating functions has a substantial overhead, it may be advantageous to time overhead as well.
- Limit the parameter space to but a few block-sizes, a small, medium and large (maximally 256)
- The processor grid is also of major relevance to the performance. Try and figure out what is "worst case" and "best case". Then do benchmarks on a few of these.