Report

INF-9380

Mathias Bockwoldt

20.05.2016

The repository with all files, including this report, can be found here: https://github.com/mathiasbockwoldt/inf9380exam.git .

1 ChIP-Seq SLURM workflow

All files for this part are in the folder `chipseq-part`.

1.1 Bowtie and MPI (part 1)

All files for this part are in the folder `ex1_bowtie`.

I ran Bowtie five times in serial mode to get a baseline for the execution time. Afterwards I ran Bowtie using MPI as shown in the example script with one to 16 cores five times each. Everything was run on a single node with all 16 cores allocated to my script. Hence, there should not have been any delay by inter-node data transmission. The script for running Bowtie with and without MPI is given in `run.slurm` (calling the MPI script in a loop) and `mpirun_script.sh` (calling Bowtie with MPI). The runtimes were measured by the `time` command and saved in `times.ods`. As the average times varied quite heavily, I chose the minimum time of five calls for an estimate of the runtime. The speedup is shown in fig 1 using `real` times.

The speedup is as expected. The execution gets faster with each additional core, but the increase in speed declines with the number of cores due to the parallelizing overhead. When using more cores, the file must be split into more chunks, more processes must be spawned and more result files must be concatenated after execution. With few processes, this pays off, but the more processes are spawned, the less pronounced is the speedup.

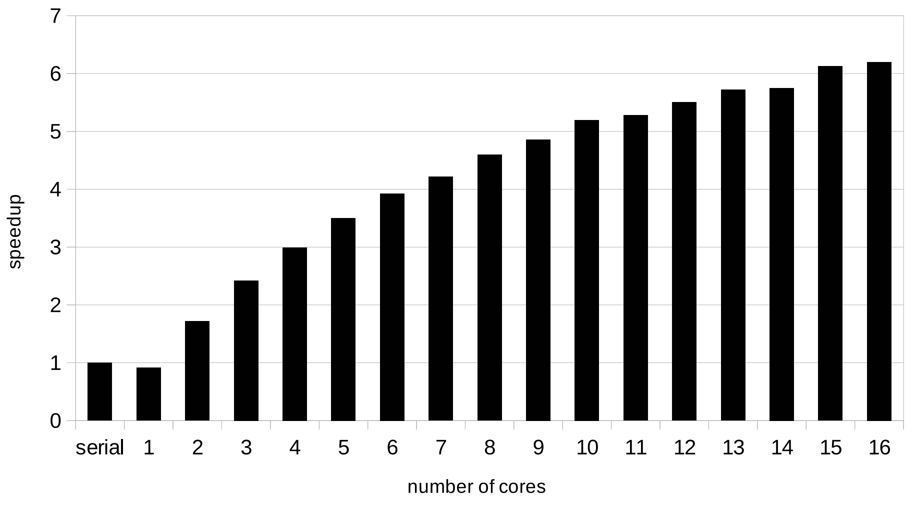

Figure 1 – The speedup of Bowtie with MPI. Serial is Bowtie without MPI whose speedup is set to 1. The MPI with only one processor has a speedup lower than 1 due to the overhead of parallelization.

A `diff` result of the output file of the serial Bowtie and Bowtie using MPI with 16 cores shows that the file contents themselves are identical. Only the header lines repeat (one header for each process). This is easily solvable for example using `awk` to remove the excessive headers[1].

1.2 SLURM scripts (parts 2 to 4)

All serial files are in the folder `ex2_serial` and the parallelized files (using `arrayrun`) in `ex3_parallel`.

SLURM scripts were written to follow the workflow given in the course. This resulted in eleven scripts plus one for preparation of the data and one R script. I did not use the SCRATCH folder in any of the scripts as the files were very small. If the files were bigger I would of course take the effort to copy all relevant files for a given script to SCRATCH and copy all relevant output files back to the original directory.

Firstly, all scripts were run ones with generous memory and time demands to find out the actual memory consumption and runtime that were used to adjust the `sbatch` directives accordingly. Runtimes were given by the SLURM log files. I wrote an SDAG file to start the workflow given in fig 2. The whole workflow was started three times with Bowtie using MPI (including the little `awk` script to remove excessive headers) and three times using serial

[1]`awk 'NR <= 3 || !/^@/'`

Bowtie. The times used for each script and the total times measured by getting the start time of the first script and the end time of the last script (usually `makeBigWig`). The results were saved in `runtimes.ods` and shown in fig 3.

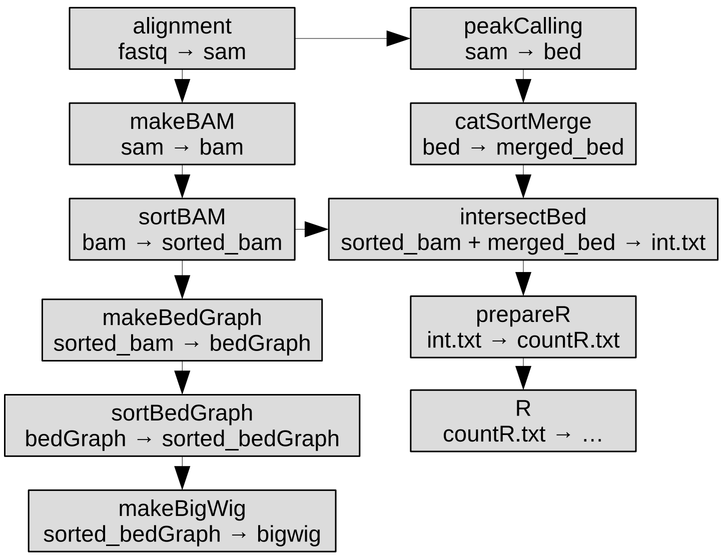

Figure 2 – The workflow used for the chipseq part. The first line shows the name of the SLURM script, the second line shows the file types that are used and produced.

Most scripts were rewritten to work with `arrayrun`. Each script that should be run in parallel was divided in a submit script (`sub_*.slurm`) and a worker script (`work_*.slurm`). The SDAG script was adapted accordingly. The new parallelized workflow was run three times and the results saved in `runtimes.ods` and shown in fig 3.

Interestingly, all arrayrun scripts took between 120 and 150 seconds (2 to 2.5 minutes), although the serial counterparts, except for the Bowtie and the bedGraph sorting step took considerably less time. As all arrayrun scripts took a very similar (and relatively long) time, I assume that arrayrun has either a huge overhead or only polls for results every few minutes. As such, arrayrun is not suitable for tasks that take less than about three minutes when run in serial. In the two cases where the serial SLURM scripts took more than 150 seconds, arrayrun resulted in a speedup but only to a very limited extend.

Both workflows without MPI (i.e. serial and parallel) coordinated with SDAG were faster than the sum of their parts (columns one vs. three and two vs. four, respectively, in fig 3), although the runtime for the whole workflow included the time waiting for allocation of resources. The speedup is much larger for the parallel workflow than for the serial one. This is due to the longer runtimes of the parallel scripts compared to the serial scripts, such that the overhead of running the workflow (especially the time for resource allocation) is relatively less for the parallel than for the serial workflows.

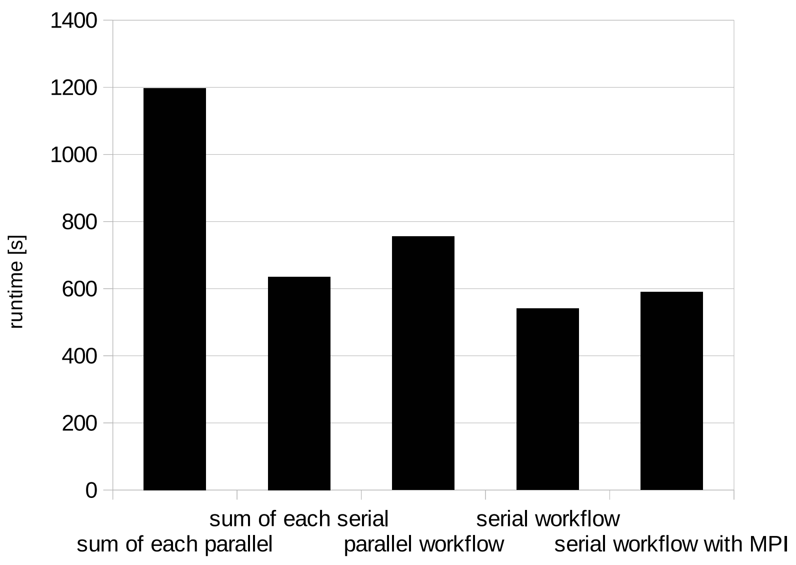

Figure 3 – Runtimes for different variations of the chipseq workflow. The first two bars show the sum of runtimes of each script using `arrayrun` (first bar) or not using any parallelization (second bar). The other three bars show the actual runtimes of whole workflows from the start of the first script until the end of the last script. Each value depicted here is the minimum of three executions of the respective workflows.

Running Bowtie with MPI in eight processes resulted in slightly longer total workflow runtimes than without any parallelization. This is probably again due to the larger overhead of MPI parallelization (splitting, distribution, and merging of files) compared to the small file sizes used in this exam.

In general, larger files would have supported the use of parallelization more than the relatively small files used in the course.

1.3 Setup manual (part 5)

This manual assumes that all relevant files are in the same directories[2] as they were during the course. Furthermore, it assumes execution on Stallo as well as all used programs being installed properly or in the course directory.

1. clone my git repository to Abel (`git clone https://github.com/mathiasbockwoldt/inf9380exam.git`)
2. navigate to the directory `chipseq-part/ex2_serial` or `chipseq-part/ex3_parallel` for the serial and parallel part, respectively

[2] `/work/projects/norbis/workflows/chip/`

3. prepare directory structure and index reference genome by running `sbatch preparation.slurm`

4. load the sdag module (`module load sdag`)

5. run the desired workflow (`chipseq.sdag` for both, parallel and serial, or `chipseq_withMPI.sdag` in the serial folder for execution without arrayrun but with MPI for the Bowtie part `sdag chipseq.sdag`)

2 Differences between two VCF files

All files for this part are in the folder `python-part`.

The Python 3 code I wrote to find differences between two VCF files is very crude and unflexible but it gets its job done.

First the control and subject files are read into Python lists. Afterwards, these are turned into dictionaries (key-value pairs). Each key is a tuple of the chromosome, the position on the chromosome and the alternative base. The value is the whole line of the respective SNP. For each dictionary, a Python set is generated of the dictionary keys. These two sets are then subtracted from each other to result in a new set with only the keys that are in the subject but not in the control. Based on this new set, a dictionary is returned with only the values whose keys are in the new set. All lines in that resulted dictionary are finally written to disk.

The parallel script does in principle the same, but splits the input files by chromosome and initializes a Python multiprocessing worker pool to send the data for each chromosome to a separate process. After processing the data, the results are written serially to a single file.

The fastest run of the serial program took 150 ms `real` and 124 ms `user` time, whereas the fastest run of the parallel version needed 233 ms and 184 ms `real` and `user` time, respectively. The average times were close to these values. Thus, the parallel execution took more time than the serial execution. This is due to the overhead of splitting and merging the working packages as well as the initialization of the worker pool. The parallel script would probably be faster than the serial script when used with larger files than the given test files.

3 Galaxy and Docker

All files for this part are in the folder `galaxy-part`.

[3]http://haystack.mocklerlab.org

I decided to write a wrapper for the software `Haystack`[3]. In my PhD project, I use none of the programs in the list given for this exam and I thought it would be better to use something, I actually work with. Haystack is a Perl script to identify oscillating genes in gene expression data. The program call is not as complicated as the example programs, but with seven parameters it fulfills the demand for more than five parameters and still kept me busy for quite some hours.

I got a basic xml file by using `planemo tool_init` with many parameters. This xml file was then filled with additional data and two tests. After some corrections, the Planemo linting tool revealed no obvious failures in the file. Also the tests with `planemo test` passed.

After creating a `.shed.yaml` file and filling it with necessary infos, I created the repository `haystack` in the Test Tool Shed[4] where I could then upload the tool with `planemo shed_upload`.

I wrote the necessary files for extending the Docker Galaxy flavour and built it with `docker build`, ran it and pushed it to DockerHub where it can be found under `mathiboc/galaxyhaystack`. I had a problem with a Perl dependency. Haystack needs the Perl module Statistics::Distributions which was installed on my Computer, so I did not get any failures while testing the Galaxy tool on my computer. However, the module is not installed on the basic Galaxy Docker image. I tried to define the dependecies such that the module is installed automatically (using a `tool_dependencies.xml` file that can still be found in the git repository), but when starting the Haystack-flavoured Galaxy in Docker, I always got the error message that the required module is not available. After a lot of unsuccessful reading and trying, I decided to resolve the dependecies by hand in the `Dockerfile` (lines 17 and 18). I'm aware of that this is not the best practice but the deadline forced me to compromise at some point.

[4]https://testtoolshed.g2.bx.psu.edu