Research Group for Parallel Computing
Faculty of Informatics
TU Wien

Parallel Computing

2022S

Home exercises and programming projects.

Issue date:	2022-05-11
Group registration:	2022-06-07 (23:59)
Due date:	2022-06-15 (23:59)
submission opens	2022-06-08 (00:00)

- The exercises and programming projects can be done in groups of up to three. Mark hand-in clearly with name and matriculation number. Hand-in via TUWEL, there will be no deadline extension.
- Solutions should be short and concise, but do explain your solution. Solutions can be prepared in English or German.
- For legibility, handwritten solutions are not accepted.
- For the programming projects, working programs according to the specification have to be handed in. Benchmark results must be produced on the "`hydra`" system. Read carefully how to prepare the hand-in of the programs and outputs.
- Grade points will be given for correctness, clarity, explanations, code, and overall style. There are 90 points in total for the whole exercise sheet.
- Codes that do not compile on `hydra` will lead to 0 points in the respective exercise.

WHAT TO HAND IN

1. A PDF file with the solutions to all exercises and the plots (we provide an optional LaTeX template),
2. `parmerge-student-1.0.0-Source.tar.bz2`, and
3. `mpimv-student-1.0.0-Source.tar.bz2`.
   (Note: `-student-` is simply part of the tarballs' name. You do not have to rename the files.)

In order to create these tarballs, go the respective directory (either to parmerge-student-1.0.0-Source or to mpimv-student-1.0.0-Source) and type

```
cmake .
make package_source
```

Make sure that the tarball indeed contains your updated versions of the source code. DO NOT REUPLOAD the provided tarball without any modification! We cannot accept code changes after the deadline.

Hint: Please download your uploaded tarball from TUWEL and check whether this version is indeed the version that you want to submit.

Exercise 1 (9 [2+4+3] points)

The following sequential code performs matrix-vector multiplication of an $m \times n$ matrix `M[m][n]` and an n-element vector `V[n]` with the resulting vector stored as `W[n]`.

```
void mv(int m, int n, double M[m][n], double V[n], double W[m])
{
  int i, j;

  for (i=0; i<m; i++) {
    W[i] = 0.0;
    for (j=0; j<n; j++) {
      W[i] += M[i][j]*V[j];
    }
  }
}
```

The exercise is to parallelize this algorithmic idea on the PRAM, using the `par()` pseudo-code notation as used in the lectures for starting a number of PRAM processors.

1. Give a simple parallel implementation of the code on a PRAM that can run in $O(n)$ parallel time steps. State which PRAM model (EREW, CREW, CRCW) your solution requires?

2. Give a faster parallel implementation of the code on a PRAM that can run in $O(\log n)$ parallel time steps. State which PRAM model (EREW, CREW, CRCW) your solution requires? How many operations does your solution perform as a function of m and n? Is this work-optimal? You can use the ideas from the lecture slides for computing the maximum of n numbers in $O(\log n)$ time steps and $O(n)$ operations.

3. Sketch how to make the fast algorithm run on an EREW PRAM. The outcome should be an algorithm running on $O(\log n)$ time steps on an EREW PRAM with no more operations than the algorithm developed above. What are the essential modifications that are needed to achieve this?

Exercise 2 (7 [1+1+2+1+2] points)

Consider this program fragment, where `a` and `b` are arrays of integers:

```
for (i=0; i<n; i++) {
  int count = 0;
  for (j=0; j<i; j++) {
    if (a[j]<=a[i]) count++;
  }
  j++;
  for (; j<n; j++) {
    if (a[j]<a[i]) count++;
  }
  b[count] = a[i];
}
for (i=0; i<n; i++) a[i] = b[i];
```

1. What does the program do? What does the comparison with $\leq$ ensure? What is this property called?

2. What is the asymptotic, sequential work of the code fragment? Give a short explanation and calculation.

3. Parallelize the program with OpenMP by addressing single loops only (no nesting of parallel regions!). Modify the program by inserting pragmas, and give the final program with all necessary OpenMP directives.

4. Given p threads, what is the asymptotic, parallel running time of your parallelization as a function of n and p (assuming $n \geq p$)?

5. What is the relative and absolute speed-up of your implementation? For the absolute speedup, use the best known asymptotic running time for this type of algorithm.

Exercise 3 (6 [1+4+1] points)

In the following loop, which is parallelized with OpenMP, the concrete schedule is decided at runtime by the setting of the `OMP_SCHEDULE` environment variable.

```
1 #pragma omp parallel for schedule(runtime)
2 for (i=0; i<n; i++) {
3   a[i] = omp_get_thread_num();
4   t[omp_get_thread_num()]++;
5 }
```

The loop is run with 6 threads (by setting `OMP_NUM_THREADS=6`) with `n=19` iterations.

1. What do `a` and `t` count?
2. Give the values for all elements in `a` and `t` with
   - `OMP_SCHEDULE="static"`.
   - `OMP_SCHEDULE="static,2"`.
   - `OMP_SCHEDULE="static,5"`.
   - `OMP_SCHEDULE="static,6"`.

   Show possible values (one possibility is enough) for all elements in `a` and `t` with
   - `OMP_SCHEDULE="dynamic,1"`.
   - `OMP_SCHEDULE="dynamic,2"`.
   - `OMP_SCHEDULE="guided,3"`.

   To answer this question, fill the following tables:

case / i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
static			insert thread ids																
static,2																			
static,5																			
static,6																			
dynamic,1																			
dynamic,2																			
guided,3																			

case / t	$t[0]$	$t[1]$	$t[2]$	$t[3]$	$t[4]$	$t[5]$
static						
static,2						
static,5						
static,6						
dynamic,1						
dynamic,2						
guided,3						

3. What is a possible performance problem with the assignment to the `t` array (Line 4)? Explain your answer.

Exercise 4 (8 [4+4] points)

Consider the standard, sequential merging algorithm, as given in the lecture slides, to be run on a single core with a certain cache system.

```
i = 0; j = 0; k = 0;
while (i<n&&j<m) {
   C[k++] = (A[i]<=B[j]) ? A[i++] : B[j++];
}
while (i<n) C[k++] = A[i++];
while (j<m) C[k++] = B[j++];
```

The inputs are two large arrays of integers, A and B, with n elements each (both `n` and `m` set to n). The output array is called C and contains $2n$ elements. All arrays are much larger than can fit in the small cache. The block size of the cache can hold 16 integers à 4 Bytes each, 64 Bytes in total. The cache is directly mapped. The cache is write non-allocate, so that writes to C will not cause cache misses (this may not correspond to any existing cache system). It also means you can ignore array C when investigating the cache behavior. The input arrays have been allocated in such a way that each $A[i]$ goes to the same cache line as $B[i]$ (same index i).

In order to analyze the cache behavior of the merging algorithm under these conditions, we consider different possible inputs. We assume that once an array element (e.g, `A[i]`) has been accessed (loaded from cache, with or without a cache miss), it will be in a register. Thus, subsequent accesses to a specific element (e.g., `A[i]`) will not be counted as a cache access. It is now your task to

1. Construct a *best case* input for A and B leading to the smallest number of cache misses (hint: how should the values inside A and B look like to avoid cache misses?). Give the cache miss rate for the $2n$ iterations of the merging algorithm.

2. Construct a *worst case* input for A and B with the largest number of cache misses. Give the cache miss rate for the $2n$ iterations of the merging algorithm.

Exercise 5 (12 [4+6+2] points)

Two sorted arrays A and B of n and m distinct elements, respectively, can be merged into an array C by computing for all elements $A[i], 0 \leq i < n$, of A their `rank`$(A[i], B)$ in B (the number of elements in B that are smaller than $A[i]$) and putting these elements into $C[i + \texttt{rank}(A[i], B)]$; and similarly for the elements in B. This leads to a correctly merged output in array C. The algorithm can be easily parallelized, since each element of A and B can be treated independently of the other elements.

The programming task is to implement this parallel algorithm in C with OpenMP. The exposed interface should be as follows.

```
void merge(double A[], long n, double B[], long m, double C[]);
```

where `A` is a given, preallocated input of `n` floating point numbers (`double`), `B` the other preallocated input array of `m` elements, and `C` a preallocated array for the output of $n + m$ elements. A useful internal interface (that needs to be devised and implemented) is

```
int rank(double x, double X[], int n);
```

for computing the rank of element x in the n-element array X.

1. Implement the two ranking steps as parallel loops with an appropriate OpenMP loop parallelization directive (`merge1.c`). Use the fastest sequential algorithm for computing the rank that was explained in the lecture. Explain briefly the solution in your text and state the asymptotic runtime of your algorithm.

2. Benchmark your merge operation on the `hydra` system (see test cases below). Create and discuss plots for the benchmarks.

3. Show how to make the merging algorithm stable, meaning that the assumption that all elements in A and B are distinct can be dropped. Equal elements in A as well as B should preserve their relative order in C, and all elements in A that are equal to some element in B should be placed before the B element in C. Hint: you need to consider the specification of the `rank` operation, and possibly use two different rank operations, one for ranking A-elements in B and one for ranking B-elements in A. This strategy only has to be described but not implemented or benchmarked.

Test cases

Measure the running time of the current version of merge with 2, 4, 8, 11, 16, and 32 threads. Repeat the measurement 10 times for each test case and compute the average (mean) running time per case.

Also benchmark the sequential merge implementation (provided, use flag `-s` for each "merge tester"), and compute the absolute speed-up with respect to this "best possible" implementation.

The different input sizes are

1. input 1: $n = 1\,000\,000, m = 2\,000\,000$,
2. input 2: $n = 10\,000\,000, m = 20\,000\,000$, and
3. input 3: $n = 100\,000\,000, m = 200\,000\,000$.

You need to create and discuss the following plots:

1. input 1, x axis: number of threads, y axis: runtime
   input 1, x axis: number of threads, y axis: absolute speedup;

2. input 2, x axis: number of threads, y axis: runtime
   input 2, x axis: number of threads, y axis: absolute speedup; and

3. input 3, x axis: number of threads, y axis: runtime
   input 3, x axis: number of threads, y axis: absolute speedup.

Exercise 6 (14 [2+4+6+2] points)

Two arrays A and B of n and m elements, respectively, can be merged into an array C by socalled co-ranking as shown in the following code (and as discussed in the lecture) where a fixed number of blocks (corresponding to the number of threads or processes in a parallel implementation) `t` has been chosen.

```
void merge_corank(double A[], int n, double B[], int m, double C[])
{
  int t; // number of blocks (threads)
  int i;

  int coj[t+1];
  int cok[t+1];

  for (i=0; i<t; i++) {
    corank(i*(n+m)/t,A,n,&coj[i],B,m,&cok[i]);
  }
  coj[t] = n;
  cok[t] = m;

  for (i=0; i<t; i++) {
    merge(&A[coj[i]],coj[i+1]-coj[i],
          &B[cok[i]],cok[i+1]-cok[i],
          &C[i*(n+m)/t]);
  }
}
```

The programming task is to parallelize this code with OpenMP. The interface is

```
void merge(double A[], long n, double B[], long m, double C[]);
```

as explained in the previous exercise. The function

```
int corank(int i, double A[], int n, int *j, double B[], int m, int *k);
```

is the interface for the co-ranking operation that is described in the lecture slides/script. It should be straight-forward to adopt the algorithm.

1. What is the asymptotic complexity of the sequential code as a function of t, n, m? How must t be chosen to be comparable with the sequential complexity of merge?

2. Parallelize the code with OpenMP, such that each thread generates an output block in C, that is replace and/or parallelize the loops in the code by using appropriate OpenMP constructs (`merge2.c`).

3. Benchmark your merge operation on the `hydra` system (see test cases below). Create and discuss plots for the benchmarks. Test cases: Use the same test case as in Exercise 5.

4. The sequential code performs the co-ranking steps once per block. Where is synchronization necessary in your parallel OpenMP code? Explain how this is achieved in your program. Modify your program to perform instead the co-ranking twice per thread (see the discussion in the lecture slides), and explain where and how synchronization operations can be saved.

   You only need to explain your strategy, i.e., you do not have to benchmark this version.

Exercise 7 (12 [4+2+6] points)

A different use of the ranking idea was given with the divide-and-conquer merge algorithm (see OpenMP slides, the digression on cilk), and is shown as the following sequential, recursive code.

```
void merge_divconq(double A[], int n, double B[], int m, double C[])
{
  int i;

  if (n==0) { // task parallelize for large n
    for (i=0; i<m; i++) C[i] = B[i];
  } else if (m==0) { // task parallelize for large m
    for (i=0; i<n; i++) C[i] = A[i];
  } else if (n+m<CUTOFF) {
    merge(A,n,B,m,C); // sequential merge for small problems
  } else {
    int r = n/2;
    int s = rank(A[r],B,m);
    C[r+s] = A[r];
    merge_divconq(A,r,B,s,C);
    merge_divconq(&A[r+1],n-r-1,&B[s],m-s,&C[r+s+1]);
  }
}
```

The programming task is to implement this parallel algorithm in C with OpenMP using OpenMP tasks. The interface should be as follows.

```
void merge(double A[], int n, double B[], int m, double C[]);
```

as explained for Exercise 5. The `rank` operation can be taken from the previous exercises.

1. Give a parallel implementation of the divide-and-conquer algorithm in OpenMP (`merge3.c`).
2. How could a good CUTOFF to stop the recursion early be estimated (irrelevant? a constant? some function of input and environment?)? Explain and give your calculations.
3. Benchmark your divide-and conquer merge operation on the `hydra` system with your best CUTOFF value/function. Create and discuss plots for the benchmarks. Test cases: Use the same test case as in Exercise 5.

Exercise 8 (11 [5+2+4] points)

A parallel matrix-vector multiplication algorithm using MPI and the `MPI_Allgather` collective was discussed in the lecture. The algorithm assumes that the global input $m \times n$ matrix A has been distributed row-wise across the p MPI processes and the input vector x likewise, such that each process has the same number m/p of full rows and the same number n/p of input vector elements. Also the result $b = Ax$ is distributed across the processes, and each process has the task of computing the part of the output for its rows. Each process computes m/p of the output elements. It was assumed that p divides both m and n.

The programming task now is to make this algorithm work also for cases where p does not divide m nor divides n. This extension can easily be done by using instead the `MPI_Allgatherv` collective. Now, each MPI process has a number of rows m_i of the matrix each with the same, global number of n columns, that is, each process stores an $m_i \times n$ submatrix (in contrast to the $(m/p) \times n$ submatrix for the regular case). The total number of rows is $m = \sum_{i=0}^{p-1} m_i$ for p MPI processes. Each process also stores a part of the input vector, namely n_i elements with $n = \sum_{i=0}^{p-1} n_i$. The output will consist of m_i vector elements.

The interface for the generalized, irregular, matrix-vector multiplication function should be as follows.

```
void mv(base_t **A, int nrows, int ncols, int nrows_a_loc, int ncols_a_loc,
        base_t *x, int ncols_x_loc,
        base_t *b, int nrows_b_loc);
```

Matrices are dynamically allocated as an array of pointers to the rows, as is shown in the accompanying benchmarking code, therefore the function signature uses a double pointer for the matrix `A`. The matrix, the part of the input vector `x`, and the part of the result vector `b` must all have been allocated prior to calling the function (and be freed accordingly when no longer to be used), as is correctly done in the benchmarking code.

1. Implement the matrix-vector multiplication function in MPI as indicated. Your solution should be placed in file `mv1.c`.

2. When each process is generating its own part of the total input matrix, with each part being approximately the same size (that is, a small difference between the number of local rows), what is your expectation on the running time as a function of the number of MPI processes? Compute and state the theoretical complexity of the algorithm as a function of the total input size, the absolute speed-up that can be achieved compared to a best sequential algorithm, and the parallel efficiency.

3. Benchmark your implementation using the provided framework. See test cases below. Plot and discuss the following experimental cases.

Test cases

The number of repetitions should be 20. You will need to compute the average (mean) over the 20 runtimes obtained per case. For each input size, measure the sequential time of the algorithm (your code executed with 1 process only).

The input sizes are (`-n`)

1. input 1: 300,
2. input 2: 600,
3. input 3: 1200, and

4. input 4: 12 000.

Now, you will performance experiments using N compute nodes and ppn processes per node. Then, obtain the measurements and create the plots for

1. input 1, $N = 1, ppn = 1$; $N = 8, ppn = \{1, 16, 32\}$
   x axis: number of processes, y axis: mean runtime;
2. input 1, $N = 1, ppn = 1$; $N = 8, ppn = \{1, 16, 32\}$
   x axis: number of processes, y axis: mean runtime;
3. input 1, $N = 1, ppn = 1$; $N = 8, ppn = \{1, 16, 32\}$
   x axis: number of processes, y axis: mean runtime; and
4. input 1, $N = 1, ppn = 1$; $N = 8, ppn = \{1, 16, 32\}$
   x axis: number of processes, y axis: mean runtime.

Exercise 9 (11 [5+2+4] points)

Another parallel matrix-vector multiplication algorithm using MPI and the `MPI_Reduce_scatter_block` collective was discussed in the lecture. The algorithm assumes that the global input $m \times n$ matrix A has been distributed column-wise across the p MPI processes and the input vector x likewise, such that each process has the same number n/p of full columns and the same number n/p of input vector elements. Each process computes m/p of the output elements $b = Ax$. It was assumed that p divides both m and n.

The programming task now is to make this algorithm work also for cases where p does not divide m nor divides n. This extension can easily be done by using instead the `MPI_Reduce_scatter` collective. Each process now stores n_i columns of the input matrix, and generates an output of m_i result vector elements, where $n = \sum_{i=0}^{p-1} n_i$ and $m = \sum_{i=0}^{p-1} m_i$.

The interface for the generalized, irregular, matrix-vector multiplication function should be the same as in Exercise 8.

1. Implement the matrix-vector multiplication function in MPI as indicated. Your solution should be placed in file `mv2.c`.

2. When each process is generating its own part of the total input matrix, with each part being approximately the same size (that is, a small difference between the number of local rows), what is your expectation on the running time as a function of the number of MPI processes? Compute and state the theoretical complexity of the algorithm as a function of the total input size, the absolute speed-up that can be achieved compared to a best sequential algorithm, and the parallel efficiency.

3. Benchmark your implementation on `hydra`. Plot and discuss the experimental test cases.
   Test cases: Use the same test case as in Exercise 8.

A. Additional Information for OpenMP Exercise

Here, we provide some additional information to compile and run the merge programs on `hydra`. We assume that you have copied (`scp`) the tarball to `hydra`, and there in directory `project`.

For example, you can copy the tarball like this (`hydra` is the name given in the ssh config):

```
% scp parmerge-student-1.0.0-Source.tar.bz2 hydras:~/
```

A.1. Unpack the Tarball

```
stester@hydra:~/$ pwd
/home/student/stester/

stester@hydra:~/$ ls
parmerge-student-1.0.0-Source.tar.bz2

stester@hydra:~$ tar xfvj parmerge-student-1.0.0-Source.tar.bz2
parmerge-student-1.0.0-Source/docker/
parmerge-student-1.0.0-Source/docker/Dockerfile
parmerge-student-1.0.0-Source/jobs_files/
parmerge-student-1.0.0-Source/jobs_files/run_seqmerge_tests.job
parmerge-student-1.0.0-Source/jobs_files/run_parmerge_tests.job
parmerge-student-1.0.0-Source/CMakeLists.txt
parmerge-student-1.0.0-Source/src/
parmerge-student-1.0.0-Source/src/seq_common.h
parmerge-student-1.0.0-Source/src/parmerge_tester.c
parmerge-student-1.0.0-Source/src/merge3.c
parmerge-student-1.0.0-Source/src/merge.h
parmerge-student-1.0.0-Source/src/merge2.c
parmerge-student-1.0.0-Source/src/seq_common.c
parmerge-student-1.0.0-Source/src/merge1.c
```

A.2. Generating a Makefile

After the tarball is unpacked, you can run `cmake` to create the `Makefile`. Once you have the `Makefile`. you can compile it simply using `make`.

```
stester@hydra:~$ cd parmerge-student-1.0.0-Source/
stester@hydra:~/parmerge-student-1.0.0-Source$ cmake .
-- The C compiler identification is GNU 8.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Found OpenMP_C: -fopenmp (found version "4.5")
-- Found OpenMP: TRUE (found version "4.5")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/student/stester/parmerge-student-1.0.0-
    Source

stester@hydra:~/parmerge-student-1.0.0-Source$ make
Scanning dependencies of target merge2_tester
[  8%] Building C object CMakeFiles/merge2_tester.dir/src/merge2.c.o
[ 16%] Building C object CMakeFiles/merge2_tester.dir/src/seq_common.c.o
[ 25%] Building C object CMakeFiles/merge2_tester.dir/src/parmerge_tester.c.o
```

```
[ 33%] Linking C executable bin/merge2_tester
[ 33%] Built target merge2_tester
Scanning dependencies of target merge1_tester
[ 41%] Building C object CMakeFiles/merge1_tester.dir/src/merge1.c.o
[ 50%] Building C object CMakeFiles/merge1_tester.dir/src/seq_common.c.o
[ 58%] Building C object CMakeFiles/merge1_tester.dir/src/parmerge_tester.c.o
[ 66%] Linking C executable bin/merge1_tester
[ 66%] Built target merge1_tester
Scanning dependencies of target merge3_tester
[ 75%] Building C object CMakeFiles/merge3_tester.dir/src/merge3.c.o
[ 83%] Building C object CMakeFiles/merge3_tester.dir/src/seq_common.c.o
[ 91%] Building C object CMakeFiles/merge3_tester.dir/src/parmerge_tester.c.o
[100%] Linking C executable bin/merge3_tester
[100%] Built target merge3_tester
```

Note: On some operating systems (e.g., MacOS), the default C compiler is `clang` (possibly an older version), which may lack OpenMP support. In such cases, you can change the default compiler used by `cmake` like this:

```
cmake -DCMAKE_C_COMPILER=gcc-mp-11 ./
```

This works on MacOS with `gcc-mp-11` installed from macports.

A.3. Testing your Code

You can test your code as shown below. Make sure that you use the front-end node (`hydra`) only to test small problem instances! Otherwise, use `srun` or `sbatch`.

```
stester@hydra:~/parmerge-student-1.0.0-Source$ ./bin/merge1_tester -n 10 -m 8 -p 2
p,n,m,time
2,10,8,0.000001
```

The check whether the computed results are correct, you can execute:

```
stester@hydra:~/parmerge-student-1.0.0-Source$ ./bin/merge1_tester -n 10 -m 8 -p 2 -c
p,n,m,time
2,10,8,0.000001
PASSED
```

A.4. Packing your Code For Submission

Note: Only do this after you have added your solution to the respective files.

```
stester@hydra:~/parmerge-student-1.0.0-Source$ make package_source
Run CPack packaging tool for source...
CPack: Create package using TBZ2
CPack: Install projects
CPack: - Install directory: /home/student/stester/parmerge-student-1.0.0-Source
CPack: Create package
CPack: - package: /home/student/stester/parmerge-student-1.0.0-Source/parmerge-
    student-1.0.0-Source.tar.bz2 generated.
```

A.5. Benchmarking on Compute Nodes

A.5.1. With `srun`

Using `srun`, you will run on one of the 36 different compute nodes of hydra (the worker nodes with 32 cores). If many students are currently working on the machine, it may take minutes or hours for your command to be executed. In such a case, it will be better to submit a job to the job queue (use `sbatch`). The following command executes the binary on the compute node that becomes available first.

```
$ srun -t 1:00 -p q_student -N 1 -c 32 ./bin/merge1_tester -n 10 -m 8 -p 2 -c
p,n,m,time
2,10,8,0.000000
PASSED
```

A.5.2. With `sbatch`

We have provided a batch job template in the directory `job_files`. You can submit a job using `sbatch`. Edit the job file to your own needs.

```
$ cd job_files
$ sbatch run_parmerge_tests.job
Submitted batch job 814
```

After the job has finished, the output is written to the designated output file, and you can inspect the results like this (note, this is just a sample output):

```
$ cat output_merge1_100000_200000_8.out
p,n,m,time
8,100000,200000,0.001861
8,100000,200000,0.000540
8,100000,200000,0.000460
8,100000,200000,0.000460
8,100000,200000,0.000451
8,100000,200000,0.000452
8,100000,200000,0.000451
8,100000,200000,0.000451
8,100000,200000,0.000451
8,100000,200000,0.000452
```

A.6. Developing on your Private Machine

If you want to develop on your private machine first and if you do not have a working development environment with `gcc` or `cmake` in place, you can use `Docker`. Install `Docker` on your machine and then do the following (you may need to adapt settings for your OS). Build the `Docker` image, assuming that you are in the "docker" directory of our source code package (`Docker` needs to find the `Dockerfile`):

```
# change to the docker directory of the source code package
$ cd docker
$ docker build -t parco .
```

Now, start the container and mount your source code (assuming we are in the main directory of the tarball, where the `CMakeList.txt` resides). (Beware that copying this command sometimes becomes erroneous due to font encoding problems. The command `pwd` is surrounded by backticks. You also need to adapt this line on Windows; with the PowerShell, `${PWD}` should work.)

```
# I am here and see ./src
$ ls
bin CMakeLists.txt docker  jobs_files  Makefile  src

$ docker run -it --rm -v `pwd`:/home/parco parco
```

Now that you are inside the container, and you can build the `parmmm` binary like this:

```
root@d821a55d68a4:/# cd /home/parco/

root@d821a55d68a4:/home/parco# cmake .
-- The C compiler identification is GNU 7.5.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Found OpenMP_C: -fopenmp (found version "4.5")
-- Found OpenMP: TRUE (found version "4.5")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/parco

root@d821a55d68a4:/home/parco# make
Scanning dependencies of target merge2_tester
[  8%] Building C object CMakeFiles/merge2_tester.dir/src/merge2.c.o
[ 16%] Building C object CMakeFiles/merge2_tester.dir/src/seq_common.c.o
[ 25%] Building C object CMakeFiles/merge2_tester.dir/src/parmerge_tester.c.o
[ 33%] Linking C executable bin/merge2_tester
[ 33%] Built target merge2_tester
Scanning dependencies of target merge1_tester
[ 41%] Building C object CMakeFiles/merge1_tester.dir/src/merge1.c.o
[ 50%] Building C object CMakeFiles/merge1_tester.dir/src/seq_common.c.o
[ 58%] Building C object CMakeFiles/merge1_tester.dir/src/parmerge_tester.c.o
[ 66%] Linking C executable bin/merge1_tester
[ 66%] Built target merge1_tester
Scanning dependencies of target merge3_tester
[ 75%] Building C object CMakeFiles/merge3_tester.dir/src/merge3.c.o
[ 83%] Building C object CMakeFiles/merge3_tester.dir/src/seq_common.c.o
[ 91%] Building C object CMakeFiles/merge3_tester.dir/src/parmerge_tester.c.o
[100%] Linking C executable bin/merge3_tester
[100%] Built target merge3_tester
```

It should now be possible to run the algorithms like this:

```
root@d821a55d68a4:/home/parco# ./bin/merge1_tester -n 10 -m 100 -p 3 -c
p,n,m,time
3,10,100,0.000035
PASSED
```

B. Additional Information for MPI Exercise

Here, we provide some additional information to compile and run the matrix-vector multiplication programs on `hydra`. We assume that you have copied (`scp`) the tarball to `hydra`, and there in directory `project`.

B.1. Unpack the Tarball

```
% scp mpimv-student-1.0.0-Source.tar.bz2 hydras:~/
mpimv-student-1.0.0-Source.tar.bz2

stester@hydra:~$ pwd
/home/student/stester

stester@hydra:~$ tar xfvj mpimv-student-1.0.0-Source.tar.bz2
mpimv-student-1.0.0-Source/docker/
mpimv-student-1.0.0-Source/docker/Dockerfile
mpimv-student-1.0.0-Source/jobs_files/
mpimv-student-1.0.0-Source/jobs_files/run_mv.job
mpimv-student-1.0.0-Source/CMakeLists.txt
mpimv-student-1.0.0-Source/src/
mpimv-student-1.0.0-Source/src/mpimv_tester.c
mpimv-student-1.0.0-Source/src/utils.h
mpimv-student-1.0.0-Source/src/mv_instance.h
mpimv-student-1.0.0-Source/src/mv.h
mpimv-student-1.0.0-Source/src/mv2_instance.c
mpimv-student-1.0.0-Source/src/mv1_instance.c
mpimv-student-1.0.0-Source/src/utils.c
mpimv-student-1.0.0-Source/src/mv1.c
mpimv-student-1.0.0-Source/src/mv2.c
```

B.2. Generating a Makefile

IMPORTANT:

You need to load the MPI library into your environment before calling `cmake`

`$ module load mpi/openmpiS`

After the tarball is unpacked, you can run `cmake` to create the `Makefile`. Once you have the `Makefile`, you can compile it simply using `make`.

```
stester@hydra:~/mpimv-student-1.0.0-Source$ module load mpi/openmpiS

stester@hydra:~/mpimv-student-1.0.0-Source$ cmake .
-- The C compiler identification is GNU 8.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Found MPI_C: /opt/mpi/openmpi-4.0.5-slurm/lib/libmpi.so (found version "3.1")
-- Found MPI: TRUE (found version "3.1")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/student/stester/mpimv-student-1.0.0-
   Source
```

```
stester@hydra:~/mpimv-student-1.0.0-Source$ make
[ 10%] Building C object CMakeFiles/mv1.dir/src/mv1.c.o
[ 20%] Building C object CMakeFiles/mv1.dir/src/mpimv_tester.c.o
[ 30%] Building C object CMakeFiles/mv1.dir/src/mv1_instance.c.o
[ 40%] Building C object CMakeFiles/mv1.dir/src/utils.c.o
[ 50%] Linking C executable bin/mv1
[ 50%] Built target mv1
[ 60%] Building C object CMakeFiles/mv2.dir/src/mv2.c.o
[ 70%] Building C object CMakeFiles/mv2.dir/src/mpimv_tester.c.o
[ 80%] Building C object CMakeFiles/mv2.dir/src/mv2_instance.c.o
[ 90%] Building C object CMakeFiles/mv2.dir/src/utils.c.o
[100%] Linking C executable bin/mv2
[100%] Built target mv2
```

Note: If you work on your private machine, you will need an MPI compiler and an MPI library (there are plenty of options on how to install them on the various operating systems). Therefore, we show you how to use Docker below.

B.3. Running and testing your code on `hydra`

You can test your code as shown below. When executing MPI programs, you will need to use `srun` (note: this is a dummy test, that is why the runtime is 0).

```
$ srun -p q_student -t 1 -N 2 --ntasks-per-node=2 ./bin/mv1 -n 10 -r 10
rep,n,p,t
0,10,4,0.000000
1,10,4,0.000000
2,10,4,0.000000
3,10,4,0.000000
4,10,4,0.000000
5,10,4,0.000000
6,10,4,0.000000
7,10,4,0.000000
8,10,4,0.000000
9,10,4,0.000000
```

To check whether the computed results are correct, you can execute:

```
$ srun -p q_student -t 1 -N 2 --ntasks-per-node=2 ./bin/mv1 -n 4  -c -v
rep,n,p,t
0,4,4,0.000000
0: b[0]=0 != B[0]=14
A:
0.000  1.000  2.000  3.000
4.000  5.000  6.000  7.000
8.000  9.000  10.000  11.000
12.000  13.000  14.000  15.000

b (reference):
14.000  38.000  62.000  86.000
0: b (part): 0.000
NOT PASSED
1: b[0]=0 != B[1]=38
1: b (part): 0.000
2: b[0]=0 != B[2]=62
2: b (part): 0.000
3: b[0]=0 != B[3]=86
3: b (part): 0.000
```

B.4. Benchmarking on Compute Nodes

B.4.1. With `srun`

When using `srun`, you will need to select the number of compute nodes (`-N`) and the number of processes per compute node (`--ntasks-per-node`). Thus, in our notation `XxY`, `-N` corresponds to `X` and `--ntasks-per-node` corresponds to `Y`.

If you want to run with 4 processes, where 2 processes are running on 2 different compute nodes, then you would execute

```
$ srun -p q_student -t 1 -N 2 --ntasks-per-node=2 ./bin/mv1 -n 10
rep,n,p,t
0,10,4,0.000000
```

If you want to start 4 processes, where each process is mapped to a different compute node then you would execute:

```
$ srun -p q_student -t 1 -N 4 --ntasks-per-node=1 ./bin/mv1 -n 10
rep,n,p,t
0,10,4,0.000000
```

B.4.2. With `sbatch`

We have provided batch job templates in the directory `job_files`. You can submit a job using `sbatch`. Edit the job file to your own needs.

```
$ cd job_files
$ sbatch run_mv.job
Submitted batch job 1588346
```

After the job has finished, the output is written to the designated file, and you can inspect the results like this (note, this is just a sample output):

```
$ cat output_mv2_32_300.out
rep,n,p,t
0,300,256,0.000001
1,300,256,0.000000
2,300,256,0.000000
3,300,256,0.000000
4,300,256,0.000000
5,300,256,0.000000
```

B.5. Developing on your Private Machine

Build and start the docker container as shown in the previous section (for OpenMP).

Now that you are inside the container, and you can build the `parmmm` binary like this:

```
cd /home/parco

root@8dc32f5ff938:/home/parco# cmake .
-- The C compiler identification is GNU 7.5.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Found MPI_C: /usr/lib/x86_64-linux-gnu/openmpi/lib/libmpi.so (found version "3.1")
-- Found MPI: TRUE (found version "3.1")
```

```
-- Configuring done
-- Generating done
-- Build files have been written to: /home/parco

root@8dc32f5ff938:/home/parco# make
Scanning dependencies of target mv1
[ 10%] Building C object CMakeFiles/mv1.dir/src/mv1.c.o
[ 20%] Building C object CMakeFiles/mv1.dir/src/mpimv_tester.c.o
[ 30%] Building C object CMakeFiles/mv1.dir/src/mv1_instance.c.o
[ 40%] Building C object CMakeFiles/mv1.dir/src/utils.c.o
[ 50%] Linking C executable bin/mv1
[ 50%] Built target mv1
Scanning dependencies of target mv2
[ 60%] Building C object CMakeFiles/mv2.dir/src/mv2.c.o
[ 70%] Building C object CMakeFiles/mv2.dir/src/mpimv_tester.c.o
[ 80%] Building C object CMakeFiles/mv2.dir/src/mv2_instance.c.o
[ 90%] Building C object CMakeFiles/mv2.dir/src/utils.c.o
[100%] Linking C executable bin/mv2
[100%] Built target mv2
```

It should now be possible to run the algorithms like this:

```
root@8dc32f5ff938:/home/parco# mpirun --allow-run-as-root -np 2 ./bin/mv1 -n 6 -r 10
rep,n,p,t
0,6,2,0.000000
1,6,2,0.000000
2,6,2,0.000000
3,6,2,0.000000
4,6,2,0.000000
5,6,2,0.000000
6,6,2,0.000000
7,6,2,0.000000
8,6,2,0.000000
9,6,2,0.000000

root@8dc32f5ff938:/home/parco# mpirun --allow-run-as-root -np 2 ./bin/mv1 -n 4 -v -c
rep,n,p,t
1: b[0]=0 != B[2]=62
0,4,2,0.000000
0: b[0]=0 != B[0]=14
A:
0.000  1.000  2.000  3.000
4.000  5.000  6.000  7.000
8.000  9.000  10.000  11.000
12.000  13.000  14.000  15.000

b (reference):
14.000  38.000  62.000  86.000
0: b (part): 0.000  0.000
1: b (part): 0.000  0.000
NOT PASSED
```