University Göttingen
GWDG/AG-C
Zoya Masih

Exercise 1 / 2026-04-09
Practical Course on High-Performance Computing / SoSe 26
80 Minutes Total

Practical Performance Analysis with Score-P/Vampir

Perform all the following tasks in your study group.

Contents

Task 1: Build and trace a small MPI C program (15 min)

Build, trace, and inspect a small MPI program with Score-P.

Steps

1. Connect with X11 forwarding if you want to use graphical tools remotely:

```
ssh -X user@glogin-p2.hpc.gwdg.de
```

   `-X` enables X11 forwarding, so graphical applications running on the remote system can be displayed locally.

2. Load the required modules:

```
module spider scorep/8.4
module load gcc/14.2.0 openmpi/4.1.7 scorep/8.4
```

3. Save the following MPI C code as `code.c` (You can find the code in the website page):

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void heavy_work(long long n)
{
    volatile double x = 0.0;
    for (long long i = 0; i < n; i++) {
        x += (i % 7) * 0.000001;
    }
```

```
}

int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    long long scale = 1;
    if (argc > 1) {
        scale = atoll(argv[1]);
        if (scale < 1) {
            if (rank == 0) {
                fprintf(stderr, Scale factor must be >= 1\n);
            }
            MPI_Finalize();
            return 1;
        }
    }

    long long base_work_slow = 400000000LL;
    long long base_work_fast = 100000000LL;
    long long work = (rank == 0) ? base_work_slow * scale : base_work_fast * scale;

    heavy_work(work);

    /* Everybody waits here for the slow rank */
    MPI_Barrier(MPI_COMM_WORLD);

    int value = rank;

    if (rank == 0) {
        int* recvbuf = malloc(size * sizeof(int));
        if (!recvbuf) {
            perror(malloc);
            MPI_Abort(MPI_COMM_WORLD, 1);
        }

        recvbuf[0] = value;

        for (int src = 1; src < size; src++) {
            MPI_Recv(&recvbuf[src], 1, MPI_INT, src, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
        }

        FILE* f = fopen(mpi_bad_output.txt, w);
        if (!f) {
            perror(fopen);
            free(recvbuf);
            MPI_Finalize();
            return 1;
        }

        long long repeats = 2000LL * scale;
        for (long long repeat = 0; repeat < repeats; repeat++) {
            for (int i = 0; i < size; i++) {
                fprintf(f, repeat=%lld rank=%d value=%d\n, repeat, i, recvbuf[i]);
                fflush(f);
            }
        }

        fclose(f);
        free(recvbuf);
    } else {
```

```
        MPI_Send(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Finalize();
    return 0;
}
```

4. Instrument and compile the code with Score-P:

```
scorep --mpp=mpi --thread=none --nomemory mpicc -O2 -o scorep_mpi code.c
```

5. Run the program (you can change the scale factor 50 to run the code for longer:

```
export SCOREP_ENABLE_TRACING=true
export SCOREP_TOTAL_MEMORY=4GB
srun -p <partition> -N 1 -n 4 ./scorep_mpi 50
```

6. You will see a Score-P output directory in the execution location. Explore its contents:

```
scorep-20260409_xxxx_xxxxxxxxxxxxxxxx
```

7. Optionally inspect the generated files, for example:

```
ls scorep-*/
scorep-score scorep-*/profile.cubex
```

Task 2: Analyze the output traces of Task 1 in Vampir (20 min)

Open the generated trace in Vampir and identify the main performance issues (It may take a few seconds, please be patient).

Steps

1. Load Vampir (You must have connected with -X option):

```
module load vampir
```

2. Open the trace:

```
vampir /path/to/output/traces.otf2
```

3. In Vampir, inspect:
   - load imbalance across ranks
   - waiting time at the barrier
   - communication concentration at rank 0
4. As discussed during the session, you can also install a limited Vampir version locally, copy the full output directory to your local machine, and analyze it there.

Task 3: Repeat the steps for an optimized code (10 min)

Save the following code as fixed_code.c (You can find the code in the website page), do the exact same steps in task 1, for the following optimized code, and compare the results

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

static void heavy_work(long long n)
{
    volatile double x = 0.0;
    for (long long i = 0; i < n; i++) {
        x += (i % 7) * 0.000001;
    }
}

int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /* Optional scale factor */
    long long scale = 1;
    if (argc > 1) {
        scale = atoll(argv[1]);
        if (scale < 1) {
            if (rank == 0) {
                fprintf(stderr, Scale factor must be >= 1\n);
            }
            MPI_Finalize();
            return 1;
        }
    }

    /* Fixed version:
       all ranks do the same amount of work */
    long long base_work = 100000000LL;
    long long work = base_work * scale;
    heavy_work(work);

    /* Improved communication:
       use collective communication instead of rank 0 receiving one by one */
    int value = rank;
    int* all_values = NULL;

    if (rank == 0) {
        all_values = malloc(size * sizeof(int));
        if (!all_values) {
            perror(malloc);
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
    }

    MPI_Gather(&value, 1, MPI_INT,
               all_values, 1, MPI_INT,
               0, MPI_COMM_WORLD);

    /* Improved I/O:
       rank 0 writes once in larger buffered form */
    if (rank == 0) {
        FILE* f = fopen(mpi_fixed_output.txt, w);
        if (!f) {
```

```
            perror(fopen);
            free(all_values);
            MPI_Finalize();
            return 1;
        }

        long long repeats = 2000LL * scale;
        for (long long repeat = 0; repeat < repeats; repeat++) {
            for (int i = 0; i < size; i++) {
                fprintf(f, repeat=%lld rank=%d value=%d\n,
                        repeat, i, all_values[i]);
            }
        }

        fclose(f);
        free(all_values);
    }

    MPI_Finalize();
    return 0;
}
```

Task 4: Build and trace a small Python MPI program (20 min)

Create a Python environment, install the required packages, and trace a small MPI Python program with Score-P.

Steps

1. Load Miniforge and create a local Python environment:

```
module load miniforge3
conda create --prefix ./myenv python=3.12
source activate ./myenv
```

2. Load the needed modules:

```
module load gcc/14.2.0 openmpi/4.1.7 scorep/8.4
module load openmpi
```

3. Install `mpi4py`:

```
pip install mpi4py
pip install numpy
```

4. Check that `mpi4py` is installed correctly:

```
python -c import mpi4py
```

5. Clone and install the Score-P Python bindings:

```
git clone https://github.com/score-p/scorep_binding_python
cd scorep_binding_python
pip install .
cd ..
```

6. Save the following code as `script.py`

```
from mpi4py import MPI
import numpy as np
amode = MPI.MODE_WRONLY|MPI.MODE_CREATE
comm = MPI.COMM_WORLD
fh = MPI.File.Open(comm, ./datafile.contig, amode)
buffer = np.empty(10, dtype=np.int32)
buffer[:] = comm.Get_rank()
offset = comm.Get_rank()*buffer.nbytes
fh.Write_at_all(offset, buffer)
fh.Close()
```

7. Enable tracing and set Score-P memory:

```
export SCOREP_ENABLE_TRACING=true
export SCOREP_TOTAL_MEMORY=1837MB
```

8. Run the Python program with Score-P:

```
srun -p medium -n 3 python -m scorep --io=runtime:posix --mpp=mpi script.py
```

9. Inspect the generated Score-P output directory:

```
scorep-20260409_xxxx_xxxxxxxxxxxxxxxx
```

Task 5: Analyze the output traces of Task 4 in Vampir (15 min)

Steps

1. Open the generated trace in Vampir, and analyze it (follow the steps in task 2).