HPC - MPI - optimizations of supercomputer code

Table of Contents

1 Files

Files for lab 4: mpi-lab-04.zip

- `lapace-seq.c`: sequential implementation of Laplace method (stencil with checkerboard algorithm, lab 3). We will benchmark it here.
- `blas-dmmmult.cpp`: a C++ implementation of matrix-matrix multiplication that invokes BLAS. You will implement your own matrix-matrix multiplication to compete with BLAS.
- `floyd-warshall/`: parallel implementation of Floyd-Warshall algorithm from lab 2 (here, we will benchmark and later break it)

2 Compiler optimizations

We assume the Cray Compilation Environment (CCE) in this part of the lab. Other compilers have similar options, but usually with different command-line switches.

2.1 Compiler feedback

The compiler is usually smart enough to significantly optimize our code, unless we disturb it. You can check the annotated source code with compiler feedback using:

```
CC -c -fsave-loopmark laplace-seq.cpp
```

Exercise: Look for loops that cannot be unrolled or functions that cannot be inlined. Is there anything you can change to permit these optimizations?

2.2 Floating point optimizations

Try the following compiler options `-O3 -ffp=4`. `-O3` is enabled by default. `ffp` enables optimizations of some floating point operations that may result in decreased precision.

Benchmark `laplace-seq.c` compiled in both variants. (hint: if you use a reservation with `salloc`, use `srun -n 1 ./laplace-seq` to run just a single process on a node).

3 Using standard numerical libraries

Standard numerical libraries provide optimized implementations of common math functions. Supercomputer vendors usually optimize these libraries further for their systems. The three most commonly-used libraries are:

- BLAS (Basic Linear Algebra Subprograms): operations such as:
  - vector-vector operations ($\bar{y} = \alpha\bar{x} + \bar{y}$) (level 1)
  - matrix-vector operations ($\bar{y} = \alpha A\bar{x} + \beta\bar{y}$) (level 2)
  - matrix-matrix operations ($C = \alpha AB + \beta C$) (level 3)
- LAPACK (Linear Algebra Package): matrix factorizations, solving linear equations, etc. LAPACK uses BLAS internally.
- MKL (Math Kernel Library): BLAS, LAPACK, FFT, statistics, data fitting, numerical optimization (e.g., the steepest gradient descend), etc.

`blas-dmmmult.cpp` shows how to invoke BLAS matrix-matrix multiplication from C++ code.

Exercise: implement your own matrix-matrix multiplication as a function in `blas-dmmmult.cpp`. Benchmark both versions (use a single node). Try to optimize your implementation using compiler hints.

4 Adjusting Slurm job/process/thread allocation

4.1 Hyperthreading

Okeanos nodes have 2 processors, each having 12 cores, each core supporting up to 2 threads (hyperthreading). When a core hyperthreads, it can switch between two threads if one of them stalls (e.g.: waits for a data to be transferred from the main memory).

Exercise: benchmark a distributed program (e.g., your homework from the previous lab) using `-tasks-per-node=24` (no hyperthreading) and `-tasks-per-node=48` (hyperthreading).

4.2 Hybrid MPI/OpenMP

If your code uses hybrid MPI/OpenMP, try adjusting simultaneously `-tasks-per-node` (the number of MPI processes per node) and the number of OpenMP threads (`export OMP_NUM_THREADS=NNN`). In Slurm you need to add to the job description `-c NNN`, the number of CPUs (=cores) per task.

For instance, on okeanos, we can use the following:

One MPI process per node; OpenMP will use all 24 hardware cores:

```
export OMP_NUM_THREADS=24
srun -tasks-per-node=1 -c 24
```

12 MPI processes per node; each process, through OpenMP, will use 4 cores (2 physical, 2 hyperthreaded):

```
export OMP_NUM_THREADS=4
srun -tasks-per-node=12 -c 4
```

5 Profiling of distributed applications

We will use visual tools. To check that the X connection is working:

```
ssh -Y hpc.icm.edu.pl
ssh -Y okeanos
xterm # should display a window on your machine
```

To enable profiling, you have to start by loading a common module, `perftools-base`.

```
module load perftools-base
```

There are two basic modes of profiling: sampling and tracing. When sampling, the binary is periodically interrupted to check which functions are currently executing. When tracing, the binary is instrumented with additional code that logs all function calls.

We will use the Floyd-Warshall parallel implementation in the examples below - but you can use your code instead.

5.1 Profiling through sampling

```
module load perftools-lite
make clean; make
# some logging from CrayPat/X should be printed as a result of make
# grab allocation
salloc --nodes 2 --tasks-per-node 24 --account g93-1577 --time 00:05:00 --reservation=rzadcapt
# run the instrumented binary inside the allocation
srun ./floyd-warshall-par.exe 4000
# end the allocation
exit
```

The instructions above should generate a detailed profiling log in a new sub-directory `floyd-warshall-par.exe+SOME_NUMBER`.

To generate a report:

```
pat_report -o fw-trace-report.txt floyd-warshall-par.exe+SOME_NUMBER
```

To see the report in a (bad) graphical interface:

```
app2 floyd-warshall-par-exe+SOME_NUMER/index.ap2
```

(if you don't see a GUI, but an error, make sure you did `ssh -Y` to connect to okeanos).

What to look for:

- mosaic: which ranks communicate
- activity: which ranks spend time computing / transferring / syncing

Exercise: Introduce some performance bugs to the Floyd-Warshall code, e.g. a rank that computes more than other ranks. See how it influences the `app2` profiling.

5.2 Profiling through tracing:

```
module unload perftools-lite
module load perftools
make clean; make
pat_build ./floyd-warshall-par.exe
# grab allocation
salloc --nodes 2 --tasks-per-node 24 --account g93-1577 --time 00:05:00 --reservation=rzadcapt
# run the instrumented library inside the allocation
srun ./floyd-warshall-par.exe+pat 4000 # a binary instrumented for sampling
# end allocation
exit
pat_report -o fw-report.txt floyd-warshall-par.exe+pat+SOME_NUMBER
```

Now, a new file - `build-options.apa` - is generated under `floyd-warshall-par.exe+pat+SOME_NUMBER`. Uncomment the lines starting with -T that specify functions to trace:

```
# 89.89%    floydWarshallPar

#    580 bytes

         -T floydWarshallPar$$CFE_id_5a05476a_main

#    580 bytes

         -T floydWarshallPar$$CFE_id_5a05476a_main
```

If there are no such functions, find the correct symbol names by:

```
nm floyd-warshall-par.exe+pat | grep runF
```

Example output:

```
0000000000466a60 T _Z24runFloydWarshallParallelP5Graphii
```

And then put into `build-options.apa`:

```
-T _Z24runFloydWarshallParallelP5Graphii
```

Rebuild the binary according to `build-options.apa`:

```
pat_build -O floyd-warshall-par.exe+pat+SOME_NUMBER/build-options.apa # creates an +apa binary
```

run the apa binary:

```
srun ./floyd-warshall-par.exe+apa 4000 # runs this new binary
```

create a report:

```
pat_report -o fw-report-apa floyd-warshall-par.exe+apa+SOME_NUMER/
```

To see the report in a (bad) graphical interface:

```
app2 floyd-warshall-par.exe+apa+SOME_NUMER/index.ap2
```

Compare the report with the sampling-based report. Check the HW counters analysis.

6 Debugging

There is a debugger accessible in an allocation. You start with asking for an allocation:

```
salloc --nodes 2 --tasks-per-node 24 --account g93-1577 --time 00:05:00
```

Then, load the module containing the debuger and launch the debugger:

```
module load gdb4hpc
gdb4hpc
```

Once the debugger is ready, start an MPI application (here, with 4 MPI processes) (remember to compile the binary with -g flag):

```
dbg all> launch $pset{4}  -a "10" ./floyd-warshall-par.exe
```

Create a breakpoint at a function:

```
dbg all> break runFloydWarshallParallel
```

Continue executing until the breakpoint:

```
dbg all> continue
```

Print variables:

```
dbg all> list
dbg all> print $pset::myRank
dbg all> print $pset{0}::myRank
dbg all> print $pset{2}::myRank
```

Switch focus to a particular process:

```
dbg all> focus $pset{0}
```

7 Bibliography

- Cray Compiling Environment: https://pubs.cray.com/content/S-2179/8.5/cray-c-and-c++-reference-manual-85/the-cray-compiling-environment
- BLAS: http://www.netlib.org/blas/
- LAPACK: http://www.netlib.org/lapack/
- MKL: https://software.intel.com/en-us/mkl
- CRAY PAT: https://pubs.cray.com/content/S-2529/17.05/xctm-series-programming-environment-user-guide-1705-s-2529/using-the-cray-performance-measurement-and-analysis-tools

Date: 2023/05/26

Author: Krzysztof Rządca

Created: 2023-06-02 Fri 16:49

Emacs 25.3.50.1 (Org mode 8.2.10)

Validate