Guide: Performance Optimization and OpenMP on AWS

Ignacio M. Llorente

v1.0 - 8 January 2018

Abstract

This is a guideline document to show the necessary actions to set up and use `gcc` to evaluate its performance optimization and OpenMP support on Ubuntu (16.04).

Requirements

- **First you should have followed the Guide "First Access to AWS"**. It is assumed you already have an AWS account and a key pair, and you are familiar with the AWS EC2 environment.
- We strongly recommend an instance with at least 4 vCPUs to be able to evaluate parallel implementation. The results in this guide have been obtained on a t2.2xlarge instance with 8 vCPUs.

Acknowledgments

The author is grateful for constructive comments and suggestions from David Sondak, Charles Liu, Matthew Holman and Keshavamurthy Indireshkumar.

1. Install gcc

- Install `gcc` via the toolchain PPA

```
$ sudo apt-get install python-software-properties
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
$ sudo apt-get update
$ sudo apt-get install gcc
```

- To check the `gcc` installation is successful run following command in the terminal

```
$ gcc -v
```

2. Evaluate Performance Flags

This section includes a simple optimization session aimed at verifying the correct installation of the gcc compiler.

- Upload to the VM the `seq_mm.c` code and compile with several optimization flags (you also need `timing.c` and `timing.h`). This simple code performs a 1,500 by 1,500 matrix multiplication. See that by default the matrices are created in the stack of the process (8MB), you should use `ulimit -s 64000` to increase the stack to < 64MB, which is the hard limit for the stack size.

```
$ gcc -DUSE_CLOCK seq_mm.c timing.c  -o seq_mm
$ gcc -O3 -DUSE_CLOCK seq_mm.c timing.c  -o seq_mm_o3
$ time ./seq_mm > output

real  0m28.533s
user  0m28.380s
sys   0m0.052s

$ time ./seq_mm_O3 > output

real  0m3.964s
user  0m3.836s
sys   0m0.032s
```

3. Verify OpenMP Support

This section includes a simple session aimed at verifying the OpenMP support provided by the gcc compiler.

- Use `lscpu` to visualize the number of CPUs and cores of the system.

```
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
```

```
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    1
Core(s) per socket:    8
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
Stepping:              2
CPU MHz:               2400.072
…
```

- Upload to the VM the `omp_sc.c`, compile it with the `-fopenmp` flag, and run the code with different numbers of cores.

```
$ gcc -fopenmp omp_sc.c -o omp_sc
$ export OMP_NUM_THREADS=8
$ time ./omp_sc
```

- Upload to the VM the `omp_mm.c` code with the OpenMP parallelization of `seq_mm.c`, compile it with the `-fopenmp` flag, and run the code with a growing number of cores.

```
$ gcc -O3 -fopenmp omp_mm.c -o omp_mm_O3
$ export OMP_NUM_THREADS=1
$ time ./omp_mm_O3 > output

real  0m6.023s
user  0m5.708s
sys   0m0.064s


$ export OMP_NUM_THREADS=2
$ time ./omp_mm_O3 > output

real  0m3.920s
user  0m5.884s
sys   0m0.040s


$ export OMP_NUM_THREADS=4
$ time ./omp_mm_O3 > output

real  0m2.945s
user  0m6.124s
sys   0m0.040s
```

```
$ export OMP_NUM_THREADS=8
$ time ./omp_mm_O3 > output

real   0m2.357s
user   0m6.140s
sys    0m0.076s
```

There are two important considerations from previous results:

- An OpenMP program in one thread runs slower than its sequential version, because the parallelized version introduces an overhead associated with the setup of the runtime environment and the creation of the thread. Moreover the compiler may not be able to as aggressively optimise the parallel code as the serial code.
- In order to measure times we must use real time and not cpu time, which adds the time consumed by the process in all CPUs. See that CPU times are the same for any number of threads.
- This code ends with a write to file part that limits the speedup (Amdahl law). In our particular case this sequential part takes 1.8 seconds approximately. If we only consider the parallel part we achieve a linear speedup.

4. Automatic Parallelization

gcc brings a simple automatic parallelization ….

- Use the automatic parallelization flag `-ftree-parallelize-loops=8` to generate a parallel version of `seq_mm.c`

```
$ gcc -O3 -ftree-parallelize-loops=2 seq_mm.c -o seq_mm_ap
$ time ./omp_mm_ap > output

real   0m2.367s
user   0m5.972s
sys    0m0.044s
```

Stop your instances when are done for the day to avoid incurring charges
Terminate them when you are sure you are done with your instance