

Quick Performance

Project Report

cpe 631

Kyle Ray

April 30, 2018

Contents

[Abstract 2](#_Toc500448362)

[Introduction 3](#_Toc500448363)

[Environment 3](#_Toc500448364)

[Test Layout 4](#_Toc500448365)

[Results 5](#_Toc500448366)

[MPI Results 6](#_Toc500448367)

[OpenMP Results 8](#_Toc500448368)

[Conclusion 10](#_Toc500448369)

[Future Work 10](#_Toc500448370)

[Appendix 11](#_Toc500448371)

[MPI Source Code 11](#_Toc500448372)

[OpenMP Source Code 17](#_Toc500448373)

# Abstract

A simple performance analysis tool named Quick Performance has been developed to allow software developers an easy way to gather performance information about their applications. The tool utilizes common open source analysis tools such as LIKWID (Like-I-Know-What-I’m-Doing) developed by the HPC group of the computing center at the university of Erlanger-Nuremburg (<https://github.com/RRZE-HPC/likwid>) and PERF which is a common Linux performance analysis tool.

The Quick Performance tool for this experiment is being used to analyze some simple yet effective software optimization techniques. This experiment will hopefully demonstrate Quick Performance ability to analyze and present meaningful data in a rapid manner. This experiment is a proof of concept for the development process and the role that Quick Performance could play in helping optimize algorithms during or after the development process.

# Introduction

In this day and age, we have witnessed the end of Moore’s Law. It is no longer the heavenly dream of the software developer to wait on the next great hardware update to provide their applications with out of the box speed ups. It is now up to the software developer to harness/exploit the power of current hardware via software optimization techniques. While there exists a plethora of software profiling and analysis tools to provide the developer with useful information on how to optimize their code, most of these tools have a steep learning curve.

For this project, a simple user interface has been created named Quick Performance that provides the user with some initial analysis information on their application to hopefully facilitate the optimization process. This interface utilizes some popular open source tools, LIKWID and PERF, to gather information about the platform architecture as well as performance information for the user application under test.

The following report contains the information and results where the tool was used to analyze some of the more simple software optimization techniques such as loop interchanging, blocking optimization, as well as loop unrolling; Please refer to the User’s Manual for Quick Performance on how the tool can be used as this report does not present this detail.

# Environment

The machine setup for this experiment consists of the following hardware:

CPU: Intel core i7-7500U

CPU Clock: 2.90 GHz

CPU Type: Quad Core Kaby Lake Processor

Cache:

L1: 32 KB 8 way set-associative data cache

L2: 256 KB 4 way set-associative unified cache

L3: 4 MB 16 way set-associative unified cache

Memory : 16 GB DDR4

# Results

**This portion of the report contains the data collected when running the experiment. The table below shows the serial run time of each test case.**

**Table 1: Serial Run Times for Each Test Scenario**

****

**The algorithms are ordered by complexity with SHA256 being the most complex hashing algorithm tested.**

## **MPI Results**

**This section houses the data collected from the MPI test runs.**

**Table 2: MPI Run Times for Each Test Scenario**

****

**Comparing the results from Table 2 with the results from Table 1 we can see that the MPI version performed poorly in the cases where less than eight processes were used and that the word was found near the middle or end of the file. In all other cases the serial version performed better. The values in Table 2 DO take into account the scattering of the dictionary to the other processes, so this incurs quite a bit of communication overhead as well as the overhead of creating the processes. Refer to the next two tables for the speed up and efficiency of this implementation.**

**Table 3: MPI Speed Up for Each Test Scenario**

****

**Although the MPI implementation reduced the execution in a few cases the speed up achieved was not desirable compared to the number of resources used and this can be seen by viewing the efficiency table below.**

**Table 4: MPI Efficiency for Each Test Scenario**

****

**Looking at the table above we can see that in most cases where more than one process is used that the efficiency declines significantly.**

**It seems that, from this experiment, the process of hashing a string and doing string compares doesn’t really suit the type of parallelism offered by using MPI.**

## **OpenMP Results**

**Not only was the OpenMP version much easier to implement, the results achieve were vastly better than the MPI version implemented. This implementation achieved a reasonable decrease in execution time every time more threads were utilized.**

**Table 5: OpenMP Run Times for Each Test Scenario**

****

**Comparing Table 5 with the serial run times of Table 1 we can clearly see that this implementation is much faster. It’s interesting, but makes sense, that the execution time for the case that the word is near the front and the application utilizes four and eight threads is near the same for each implementation. It seems that there is a limit for this case and adding more threads will not be as efficient.**

**Table 6: OpenMP Speed Up for Each Test Scenario**

****

**As far as speed up, looking at the table above, the cases using two and four threads have close to ideal speed up compared to the other scenarios.**

**Table 7: OpenMP Efficiency for Each Test Scenario**

****

**Overall OpenMP is the most efficient implementation as well as very easy to implement compared to the MPI version.**

# **Conclusion**

**After the experiment it is clear to see that a hacker could benefit from using parallel processing to perform the brute force dictionary attack. The experiment also demonstrated that some tools are better for the job than others in the case where MPI was very tedious to use because of all the synchronization and message passing and OpenMP involved changing a few lines of code in the serial version to achieve a substantial performance increase.**

# **Future Work**

Quick Performance has more potential to grow and there are many features of the underlying analysis tools, LIKWID and PERF, that were not incorporated into the final product. Below are a few additional areas where the tool could be improved.

1. Modifying the tool to work with multiple cores at once
2. Modifying the tool to work with multi-threaded applications
3. Integrating LIKWID support for MPI applications
4. Add a batching mode for users to queue up runs

# Appendix