International Journal of Computing Academic Research (IJCAR)
ISSN 2305-9184, Volume 5, Number 4 (August 2016), pp.233-238

http://www.meacse.org/ijcar

Performance Test of MPI on Raspberry Pi 2 Beowulf Cluster

Erdem Ağbahca and Adem Alpaslan Altun

Computer Engineering Department, Faculty of Engineering, Selçuk University, Turkey

Abstract

When a problem can be divided into subtasks and these non-sequential tasks were to run on different processes at the same time on multiple CPUs over a cluster using Message Passing Interface (MPI), performance of the application which solves the problem will be greatly enhanced. This kind of cluster can be built with any type of computer, yet single board computers (SBC) are amongst the lowest power consuming, the most mobile and easier to build with their small volumes and passive cooling. Raspberry Pi is a SBC used in a wide variety of projects from small servers to home automation, image processing, auto-navigating robots to wearable gadgets. Excluding these projects, Raspberry Pi can still be used as a computer and can even be part of a computer cluster. In this study, we present timing records of test results for finding all of the prime numbers in a range, over various process counts on a nine node Raspberry Pi 2 cluster.

Keywords: Message Passing Interface, Raspberry Pi, Parallel Programming, Cluster Computing

1. Introduction

MPI is a message-passing library addressing primarily parallel programming model, which helps in dynamic process spawning, parallel I/O, data movement between processes, collective operations and remote-memory access operations [1]. MPI is not an IEEE or ISO standard, yet it has become a de-facto standard over the years it has been used and supported by many organizations and universities.

Raspberry Pi is a SBC [2] which is used in various types of projects for example; small non-CPU intensive computing projects, home automation, TV boxes and jukeboxes, autonomous robots, home-type less traffic demanding servers and alike. However, with the study conducted at Southampton University [3], Boise State University [4] and Texas A&M University [5], Raspberry Pi's proved to be useful for cluster computing for educational, non-CPU intensive problems and most importantly where the budged matters since these SBCs cost 35$ each. Raspberry Pi 2 Model B is equipped with 900MHz (stock overclocked) quad-core ARM Cortex-A7 CPU, 1 GB of ram and 16 GB class 10 micro SD cards are used for this study. However, Pi CPUs are overclocked to 1000MHz in this study.

In computing, performance is measured in floating-point operations per second (flops) and after a Beowulf cluster is built, there are several ways to benchmark its performance, LINPACK [6] and High-Performance LINPACK is the most used ones. Since these benchmarks output the cluster performance in terms of flops as it can be seen in Iridis-Pi study [3], however the wall clock time performance of the cluster heavily depends on algorithm used to solve a problem so that there will be many flops of computing power lost in the process due to non-optimized algorithms. Because of that, we propose our test results in terms of seconds to actually depict how long a problem took to be solved on to cluster. Yet, to show that the results scale with the rise of process and node count, a problem which is suitable to divide in subtasks is needed, so first thing comes in the mind is finding prime numbers in a range which gives nearly linear result when parallelized.

2. Related Work

Building a cluster is fairly easy when the size of computers and the power demanded by them are small. SBCs can be stacked like a rack to ease carriage which can be seen in Fig 1. Raspberry Pi's needs to be powered from 5V micro USB port and can drain up to 2A. In this study, a dc power supply of 5V and 40A is used to cover further expanding the cluster size for oncoming studies.

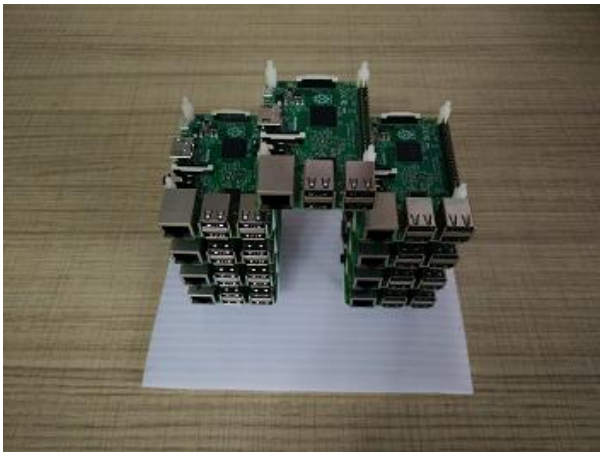

Fig.1 System Rack

Raspberry Pi's have their own OS called Raspbian [7] a derivative of Debian Linux, distributed as image files that needs to be written on SD card. After booting to the OS, an implementation of MPI should be installed and MPICH (3.2 version) [8] is used for this purpose in this study. Instead of building MPICH for each node, it can be built for the first node then the image of that SD card can be taken and written to other SD cards. Hostnames should be changed to prevent further problems from happening. After preparing and naming, each node should be connected through network to form the cluster and to achieve this a 100Mbit 3Com Switch (4400) is used to connect nodes to university network. Then one of the Pi's was set as master which is connected to a monitor, keyboard and mouse which contains a file of IP addresses for every Pi. This architecture of the said system and final system build can be seen as whole in Fig 2.

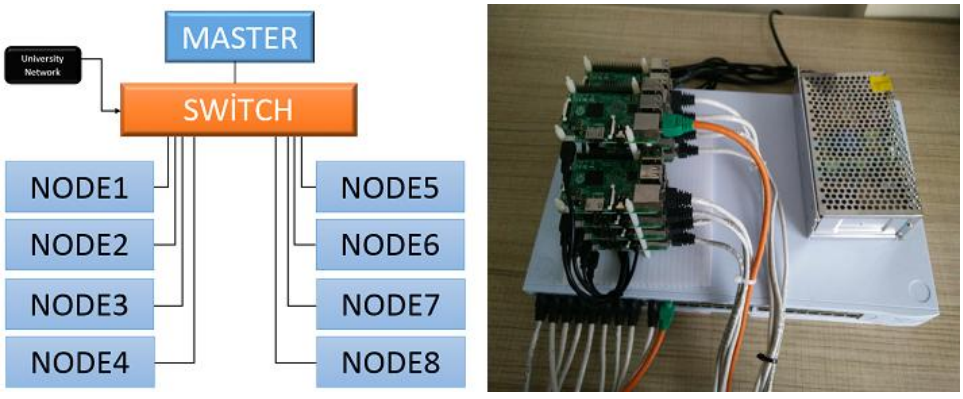

Fig.2 System Architectural Diagram (Left) Actual System Image (Right)

Each Pi contains a quad-core CPU (4 threads total) which means 4 processes can run simultaneously and that leads 9 Pi's to run 36 processes at the same time. A program using MPI is coded in C and copied to every node in the cluster using scp command. The program takes the maximum boundary as an input argument which then calculates margin for each processes depending on the process count. If total process count equals to 1 program acts serially but when it exceeds that, 1 process on the master which is called root process is only responsible for the distribution of margins and collection of the count of the prime numbers found. Rest of the available processes only finds the prime numbers after taking lower and upper bound of their margin and returns the total count of prime numbers to root process. Timing measurement starts when MPI initializes and ends when all processes returns the counts to root process. Since MPI runs the same executable in all nodes and every node executes its partition of code which are divided by if clauses depending of the rank (index of process). Writing code for just one executable eases the programming but dividing the code for several ranks makes is harder to develop an algorithm. Algorithm for the program can be seen in Fig 3 for root process and other processes.

Root	Other

Begin
 	
Read Input Arg (Maximum Boundary)
If Arg is NOT Divisible with Process Count
-Exit Program
End If
 	
Initialize MPI
Wait All Processes
Start Timer
 	
If Process Count Equals 1
-For X; 3 to Maximum Boundary
--If X is Prime (Iterative Function)
---Increment local Count
--End If
-End For
Else
-If Rank Equals 0 (Root Process)
--For Each Process (Excluding Root)
---Calculate Margins (lower, upper bound)
---MPI Send Boundaries (Blocking)
--End For
--For Each Process (Excluding Root)
---MPI Receive Found Counts (Blocking)
---Sum Counts
--End For
-Else
-….
-End If
End If
 	
Wait All Processes
IF Rank Equals 0 (Root Process)
-Stop Timer
-Write Elapsed Time & Prime Count toText File
End If
 	
MPI Finalize
End
	Begin
	
	Read Input Arg (Maximum Boundary)
	If Arg is NOT Divisible with Process Count
	-Exit Program
	End If
	
	Initialize MPI
	Wait All Processes
	Start Timer
	
	If Process Count Equals 1
	…
	Else
	-If Rank Equals 0 (Root Process)
	-….
	-Else (Not Root Process)
	--MPI Receive Boundaries (Blocking)
	--For X; lowerbound to upperbound
	---If X is Prime (Iterative Function)
	----Increment local Count
	---End If
	--End For
	--MPI Send local Count
	-End If
	End If
	
	Wait All Processes
	If Rank Equals 0 (Root Process)
	….
	End If
	
	MPI Finalize
	End

Fig.3 Program Pseudo Code

3. Results

Timing values gathered for ***1,4,8, …, 32, 36*** processes when 1 process means totally serial program, until 5 processes computation is done only on the master node. Assuming 13 process is used; 4 process runs on master node, 4 process runs on node1, 4 process runs on node2, and 1 process runs on node3. Tests are repeated 10 times (D 1, …, D 10) for each process count with a waiting period of 5 minutes in between every test to minimize ambient effects on cluster (ex. Temperature) and the average of these 10 timing results are used as final result. Timing results are in seconds and rounded to nearest integer. These result can be seen at Table 1.

Table.1 Timing Results of Tests

Process Count	1	4	8	12	16	20	24	28	32	36
D 1	8046	3561	1605	1038	766	612	506	433	379	333
D 2	7985	3550	1605	1038	769	609	506	431	376	333
D 3	8009	3540	1604	1039	769	609	508	433	378	335
D 4	8028	3544	1598	1038	765	612	506	431	377	335
D 5	7983	3543	1597	1038	769	612	507	433	376	333
D 6	7980	3562	1597	1033	769	609	506	434	378	333
D 7	8026	3557	1601	1038	766	613	508	431	376	335
D 8	7983	3557	1605	1033	769	609	506	433	378	333
D 9	8035	3540	1604	1033	765	609	506	432	376	333
D 10	8026	3556	1597	1033	769	609	506	434	376	335
Average Time(s)	**8010**	**3551**	**1601**	**1036**	**767**	**610**	**506**	**432**	**377**	**333**

When serially executed (1 process only) it takes 8010 seconds in average to find all of the prime numbers in-between 3 and maximum boundary which is approximately 133 minutes but when run as 4 processes which means 1 process distributes and 3 process computes this period drops down to 3551 seconds which is approximately 59 minutes. This is closely 2.25 times speedup which cuts the wall clock time of execution nearly %44. When it run as 8 process a speedup of 2.21 can be seen compared to previous 4 process. If we were to calculate the timing theoretically for 8 process only 7 of them does computation and for 12 process it is 11. Knowing that this algorithm is suitable for linear scaling if 7 computing processes takes 1601 seconds to finish then it should take 1018 seconds for 11 computing processes. However, timing results shows that it's slightly higher with 1036 seconds in average. This is caused by communication overhead between processes which makes one of the processes wait another to get or send the results because the previous rank hasn't done it yet. Even with the system overhead there is still 1.54 times speedup when 8 and 12 processes compared. The reason of decrease in speedup when the process size gets bigger is that process size increases 4 process for every node for every new test but the percentage of currently available resource to these new processes is getting smaller. For example, when run as 16 processes it takes 610 seconds to complete but when the process count is risen to 20 it drops 506 which means ~1.21 times speedup. This is because that there were 15 computing processes and it has risen to 19 which brings ~1.29 times speedup theoretically. So that considering system overhead for sending and receiving operation attained speedup is completely normal and can still be considered linear. When run as 36 processes it takes 333 seconds to complete in comparison to 8010 seconds of serial execution it is nearly %2405 performance gain in terms of time. In order to provide better visual depiction, speedup for every process count is shown in Fig 4.

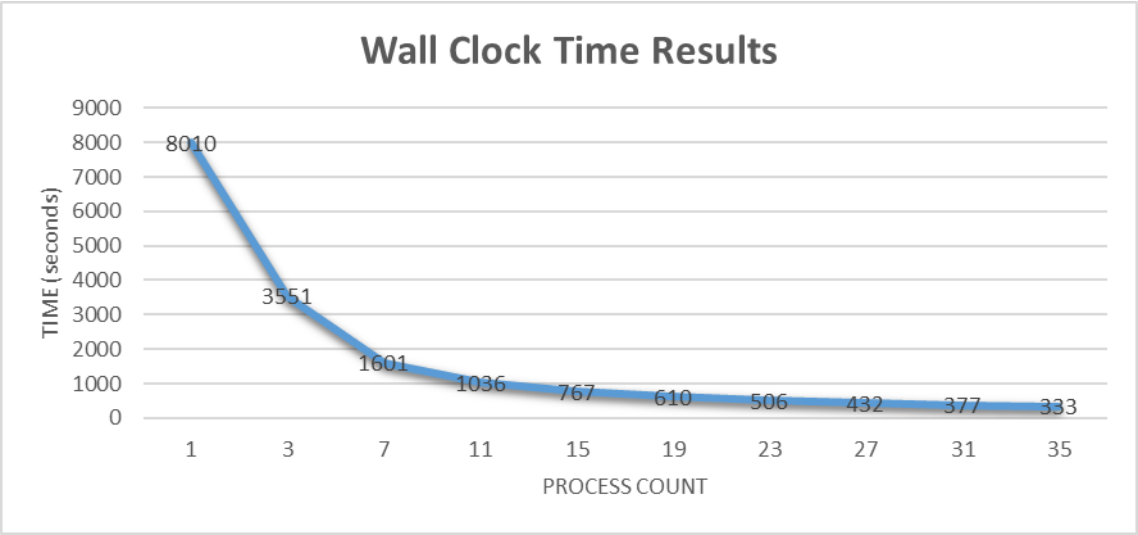

Fig.4 Timing Graph of Tests (Root Process Excluded)

If Fig 4 is reviewed as a summary, even the highly parallelizable problems can lose the immense grip they have in advantage of speedup when the needed computational power is reached. Although, the speedup will continue until a break point where the overhead is bigger than the computational time, yet it wouldn't be as advantageous as it is shown in the Fig 4. Considering 36 processes takes 333 seconds to complete the whole computation, if the process count were risen to 72 processes, it would be taking 167 seconds theoretically (ignoring communication overheads). Which would lead to 166 seconds of advantage over 36 processes with twice the cost. However, it is not to be paid attention when compared to the 4 – 8 process difference of 1950 seconds. These results can be interpreted as when the barrier of the needed computational power is reach, cluster performance loses the grip it has on performance advantage.

4. Conclusion

It is observed that Raspberry Pi's can be used for complex computational problems by clustering them together even though they are mostly used by hobbyist for different types of projects. With their small volumes and weights which provides better mobility over conventional clusters and their low power consumption and low cost, they can be used in specific areas which can benefit from these properties.

In addition, communication overhead can be lowered by using MPI only for inter-node communication and doing rest of the multiprocessing work inside on a node using OpenMP. This will be studied for several problems as future work.

5. References

[1] University of Tennessee, "MPI: A Message-Passing Interface Standard Version 3.1," 2015.
[2] Anonymus. "Raspberry," 21/03/2016; https://www.raspberrypi.org/.
[3] S. J. Cox, J. T. Cox, R. P. Boardman, S. J. Johnston, M. Scott, and N. S. O'Brien, "Iridis-pi: a low-cost, compact demonstration cluster," Cluster Computing-the Journal of Networks Software Tools and Applications, vol. 17, no. 2, pp. 349-358, Jun, 2014.
[4] J. Kiepert. "Creating a Raspberry Pi-Based Beowulf Cluster," http://coen.boisestate.edu/ece/files/2013/05/Creating.a.Raspberry.Pi-Based.Beowulf.Cluster_v2.pdf.
[5] E. Wilcox, P. Jhunjhunwala, K. Gopavaram, and J. Herrera, "Pi-Crust: A Raspberry Pi Cluster Implementation."

[6] J. J. Dongarra, P. Luszczek, and A. Petitet, “The LINPACK benchmark: past, present and future,” Concurrency and Computation-Practice & Experience, vol. 15, no. 9, pp. 803-820, Aug, 2003.
[7] Anonymus. "Raspberry Downloads," 21/03/2016; https://www.raspberrypi.org/downloads/.
[8] Anonymus. "MPICH | High-Performance portable MPI," 21/03/2016; https://www.mpich.org/.