Building a Fast Parallel Cluster Computer Using Slow PCs

Jovan Andjelich, Sean Dockery, Meghan Allen, Nick Hushebeck, Dr. Akhtar Mahmood and Dr. Faiz Ahmad (Bellarmine University)

Abstract

We have built two fast cluster computing systems (called Sphinx-1 and Sphinx-2) using slow PCs using the Fedora Core-3 Linux operating system, and the LAM-MPI (Message Passing Interface) parallel processing middleware for carrying out the various scientific calculations. The MPI middleware is suitable for data-parallel, task-parallel, and coarse-grain data flow programming and supports random or sequential access to any integral set of data items. The cluster is based on a multi-computer architecture which can be used for parallel computations. The cluster consists of one server node, and several client nodes connected together via a 100Mbps Ethernet network switch. The server node controls the whole cluster and serves files to the client nodes. Cluster computers are one of the new platforms that are at the forefront in the computational arena. A Parallel Cluster computing concept is at the boundary between the notion of a tightly coupled parallel computer and a distributed cluster system. The idea is to speed up the execution of a program by aggregating computing power across nodes to solve a problem faster (by spreading the computation across the nodes of the cluster using message passing to communicate between the cluster's nodes). Theoretically speaking, a program being executed across *n* processors might execute *n* times faster than it would by using a single processor. Our goal was to build a single computer system, a cluster, with multi-tasking capabilities having the manageability of one computer with the performance of the sum of its components. We will present the computational results obtained from our Sphinx-1 Beowulf Cluster.

Introduction and Methodology

High performance computing is undergoing a fundamental transition. Parallel Cluster computers are increasingly being used not only in the scientific research arenas, but also in commerce and industry. Additionally, Parallel Cluster computers are not only being used for high-performance computation, but increasingly as a platform to provide services for applications such as web, grid and cloud servers. One important applications of high performance cluster computers is the handling of large datasets. Data-intensive research fields such as High Energy Physics has become increasingly dependent on cluster computers for various monte-carlo simulation and data analysis tasks for extracting "rare" signals from enormous backgrounds involving large datasets.

Clusters can be of multiple flavors; one that is of interest to us is called the Beowulf cluster. Beowulf is a class of cluster computing systems built with cost-effective hardware commodity items, an operating system (e.g. LINUX), parallel processing libraries and middleware (e.g. MPI), and other utilities. The Beowulf cluster can provide a highly flexible distributed memory-based computing environment. The goal is to speed up the execution of a program by aggregating computing power across nodes to solve problems faster (by using message passing to communicate between the nodes).

In a Beowulf cluster, all the nodes are dedicated to the cluster, and function as one system to process a job. This helps ease load-balancing problems, since the performance of individual nodes are not subject to external factors. The Beowulf cluster-system usually consists of one server node, and several client (compute) nodes connected together via fast network switch. Since parallel clusters must pass information between nodes, high-speed inter-connectivity is a critical component of the cluster. In a Beowulf cluster, one server node controls the whole cluster. This server node is the cluster's console for compiling the source code, starting parallel jobs and is the only access point to and from the outside world, and is basically the "brains of the operation".

The main server is configured to serve file systems to the client nodes. Each client node has a private IP (Internet Protocol) address, thus the cluster interconnections are not "visible" to the outside world. All access to the client nodes is done via the server node. Server node consists of two network cards. One of them has a "global" IP address which connects the server to the internet and the other network switch is connected to the local network switch to communicate with the client nodes. Usually the server node has a monitor, video card, keyboard, mouse, and CD-ROM; the client nodes do not have these peripherals. It can get very hectic to switch keyboard, mouse and monitor between each client in order to install the operating system or doing some of the early troubleshooting. So, keeping this in mind, we decided to implement a KVM switch (abbreviation for Keyboard, Visual display, and Mouse) for our cluster which allows us to use a single keyboard, mouse and monitor across all the nodes including the server node.

Beowulf clusters are highly scalable, and their "parameters" can be tuned to improve the system's overall "parallel" performance, by evaluating the ratio of communication time/processing time. We ensured that the memory performance is optimized so that each client node is able to exchange data rapidly with the other client nodes. A critical aspect of a Beowulf cluster, which determines its performance, is the underlying switch fabric connecting the nodes. In a Beowulf Cluster, a process on one client node can send signals to a process on another client node, all within the user domain. The nodes must then be tuned to provide better throughput for coarser-grain jobs (where processes may perform millions of operations between communication events), because they are not interacting directly with the users.

A word of Caution: Not all applications will run effectively on a Beowulf Cluster. If a segment of a code runs for less time than it takes to transmit its result value (i.e. latency), executing that code segment serially on one node would be faster than using multiple nodes, where serial execution would avoid the communication overhead. In a cluster, the time required to move data between nodes is critical to the system's performance and parallel processing. Achieving low latency requires efficient communication protocols, message size, and host network interfaces that minimize the communication overhead. A key factor in predicting the code performance is the amount of inter-processor communication involved. We can calculate the average Message-Passing Time and Message Size for the Beowulf cluster using the following equation:

Message Communication Time = (Latency Time) + (Size/Bandwidth)

Message Size = (Bandwidth) x (Latency Time)

Cluster Middleware: The Message Passing Interface (MPI) is a set of middleware functions that enable programs to pass messages between processes of a parallel job. It has become common to use PC clusters as a single parallel computing resource running MPI programs. The MPI was designed to support portability and platform independence. MPI is a collection of library routines that consists of a header file, a library of routines and a runtime environment that provides interaction/message-passing and related operations between processors with a parallel system to enable parallel computation. MPI can be used in the Fortran, C and C++ programming languages.

Currently at least two implementations of MPI exists. Both allow MPI programs to be executed across a cluster of Linux systems using UDP/TCP socket communication:

LAM-MPI (Local Area Multicomputer) – MPI 1.1 / 2.0 Standard - Developed by Ohio State University

MPICH (MPI CHameleon) – MPI 1.1 / 2.0 Standard – Developed by Argonne National Laboratory.

We have chosen to use LAM-MPI as the Message Passing Interface. LAM is a MPI programming environment and development system for heterogeneous computers on a network. With LAM, a dedicated cluster or an existing network computing infrastructure can act as one parallel computer solving one problem. LAM features extensive debugging support in the application development cycle and peak performance for production applications. LAM features a full implementation of the MPI communication standard.

LAM-MPI is a high-performance, freely available, open source implementation of the MPI standard that is maintained at the Open Systems Lab at Indiana University. LAM-MPI is the middleware that implements the LAM run-time environment that provides many of the services required by MPI programs.

Bash Scripting to Automate the Job Processes in a Beowulf Cluster: We also wrote a *bash* script for all our parallel jobs. Bash is a command processor that allows the user to type commands for performing tasks that are frequently done by the user. Bash can also read commands from a file, called a script. Scripts can be organized to efficiently run the programs and to generate the data in such a way as to facilitate analysis of the data using tools such as Excel. Without a script, recording individual data sets for each calculation can be a very tedious process. Taking these data sets and finding ways to analyze this data in an organized fashion can be even more tedious. Scripts can automatically run a job that could take days to run. Scripts can be adjusted to different situations with minor changes in coding.

Hardware Components: PC Nodes, Network and KVM Switch

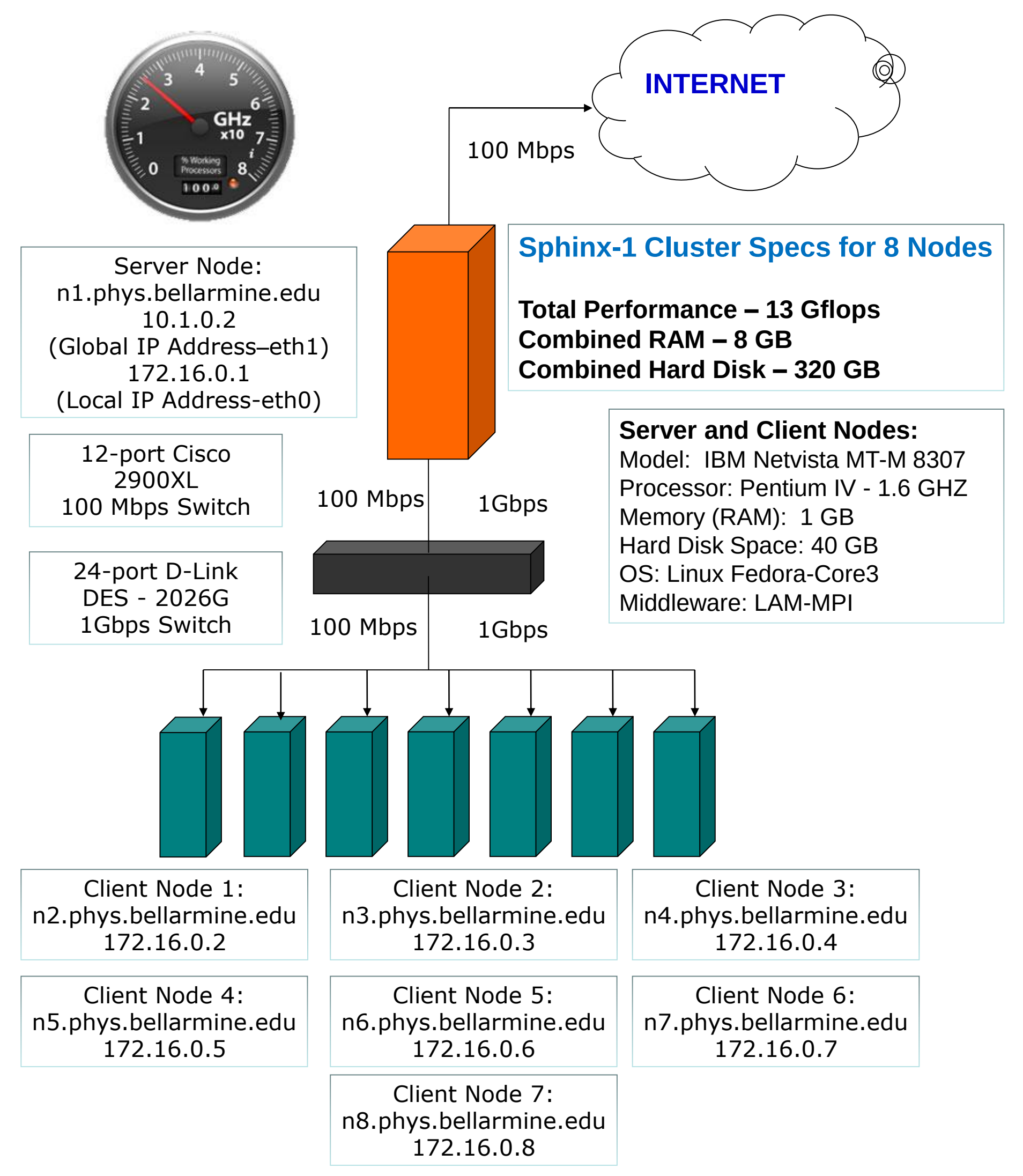
INTERNET
100 Mbps
GHz x10
Server Node: n1.phys.bellarmine.edu 10.1.0.2 (Global IP Address–eth1) 172.16.0.1 (Local IP Address-eth0)
12-port Cisco 2900XL 100 Mbps Switch
24-port D-Link DES - 2026G 1Gbps Switch
100 Mbps
1Gbps
100 Mbps
1Gbps
Sphinx-1 Cluster Specs for 8 Nodes
Total Performance – 13 Gflops
Combined RAM – 8 GB
Combined Hard Disk – 320 GB
Server and Client Nodes:
Model: IBM Netvista MT-M 8307
Processor: Pentium IV - 1.6 GHZ
Memory (RAM): 1 GB
Hard Disk Space: 40 GB
OS: Linux Fedora-Core3
Middleware: LAM-MPI
Client Node 1: n2.phys.bellarmine.edu 172.16.0.2
Client Node 2: n3.phys.bellarmine.edu 172.16.0.3
Client Node 3: n4.phys.bellarmine.edu 172.16.0.4
Client Node 4: n5.phys.bellarmine.edu 172.16.0.5
Client Node 5: n6.phys.bellarmine.edu 172.16.0.6
Client Node 6: n7.phys.bellarmine.edu 172.16.0.7
Client Node 7: n8.phys.bellarmine.edu 172.16.0.8

Figure 1. Schematics of the Sphinx-1 Beowulf Cluster.

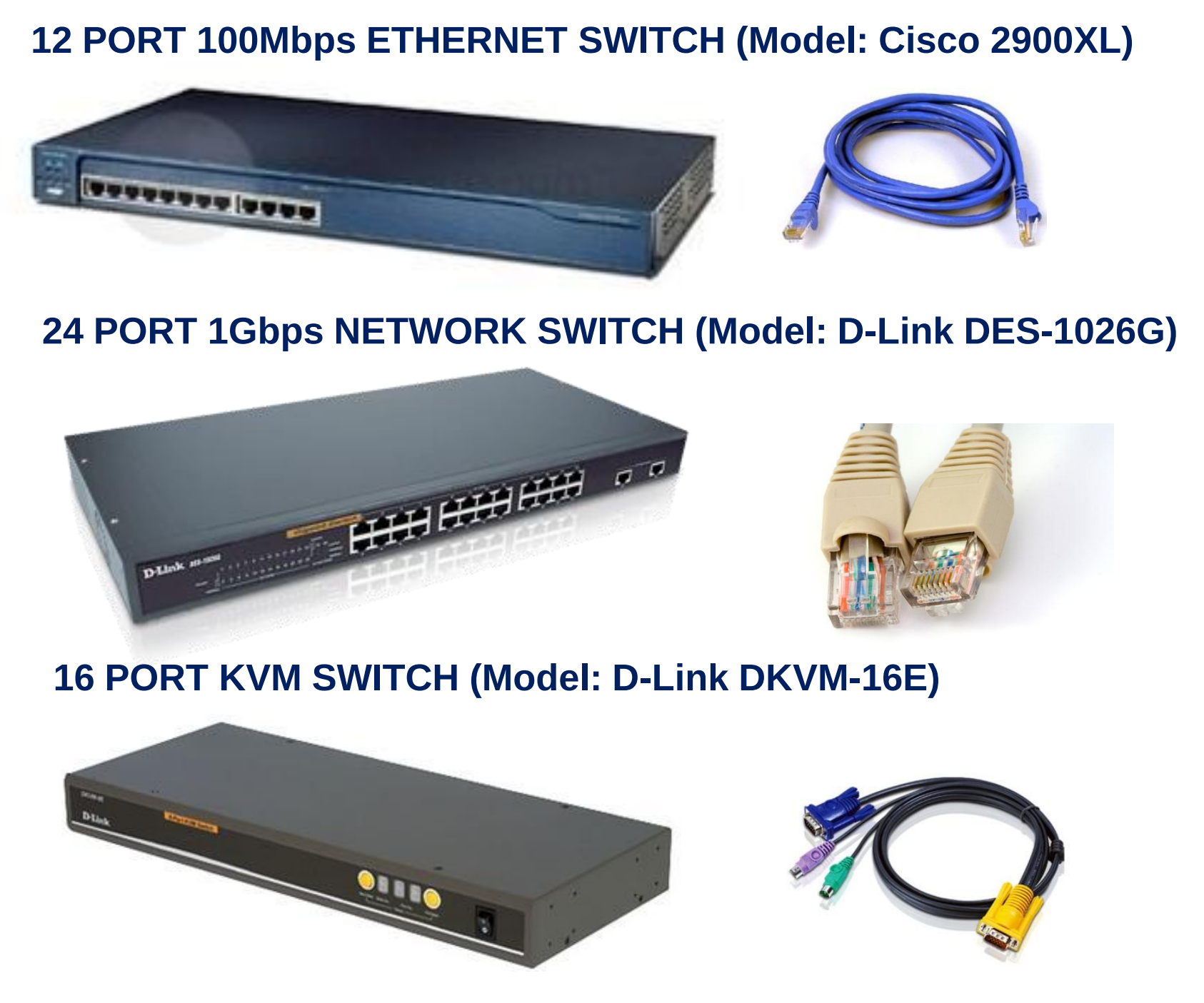
12 PORT 100Mbps ETHERNET SWITCH (Model: Cisco 2900XL)
24 PORT 1Gbps NETWORK SWITCH (Model: D-Link DES-1026G)
16 PORT KVM SWITCH (Model: D-Link DKVM-16E)

Figure 2. Shows pictures of the 100 Mbps and 1Gbps network switches and the 16-port KVM switch used to build the Sphinx-1 cluster.

Figure 3. Picture of the Sphinx-1 Beowulf Cluster in Pasteur 209.

Figure 4. Picture of the Sphinx-2 Beowulf Cluster in Pasteur 209.

Figure 5. That's Jovan posing with both the Sphinx-1 and Sphinx-2 Clusters.

Parallel Programs Studied in This Research Project

- **First Program (Particle Simulation): This program simulates up to 4000 particles using the Monte-Carlo Method.**
- **Second Program (Calculating Pi): This program calculates the value of Pi (π), by integrating the function $f(x) = 4.0/(1.0 + x^2)$ using *n* partitions. It then compares the result up to 25 decimal place with the known value of Pi and determines the error of the calculated value of π.**
- **Third Program (Parallel Trapezoidal Method): This Trapezoid Method program integrates the function $f(x) = 1/x$ from *a* (100) to *b* (1000000) using *n* partitions.**

Results

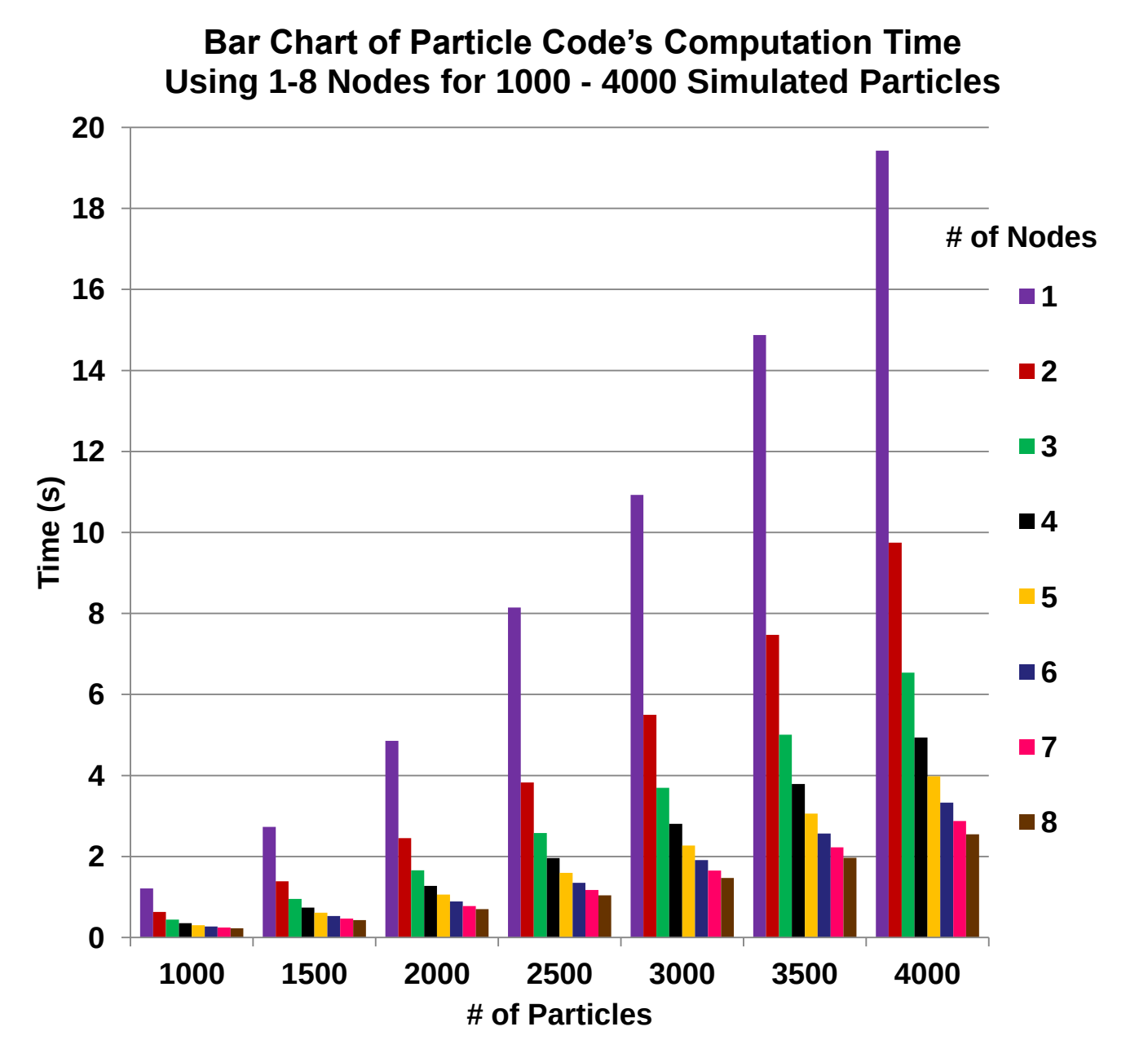

Figure 6. shows a bar chart of the Particle code's computation time vs. the number of nodes for 1000 - 4000 Monte-Carlo generated particles.

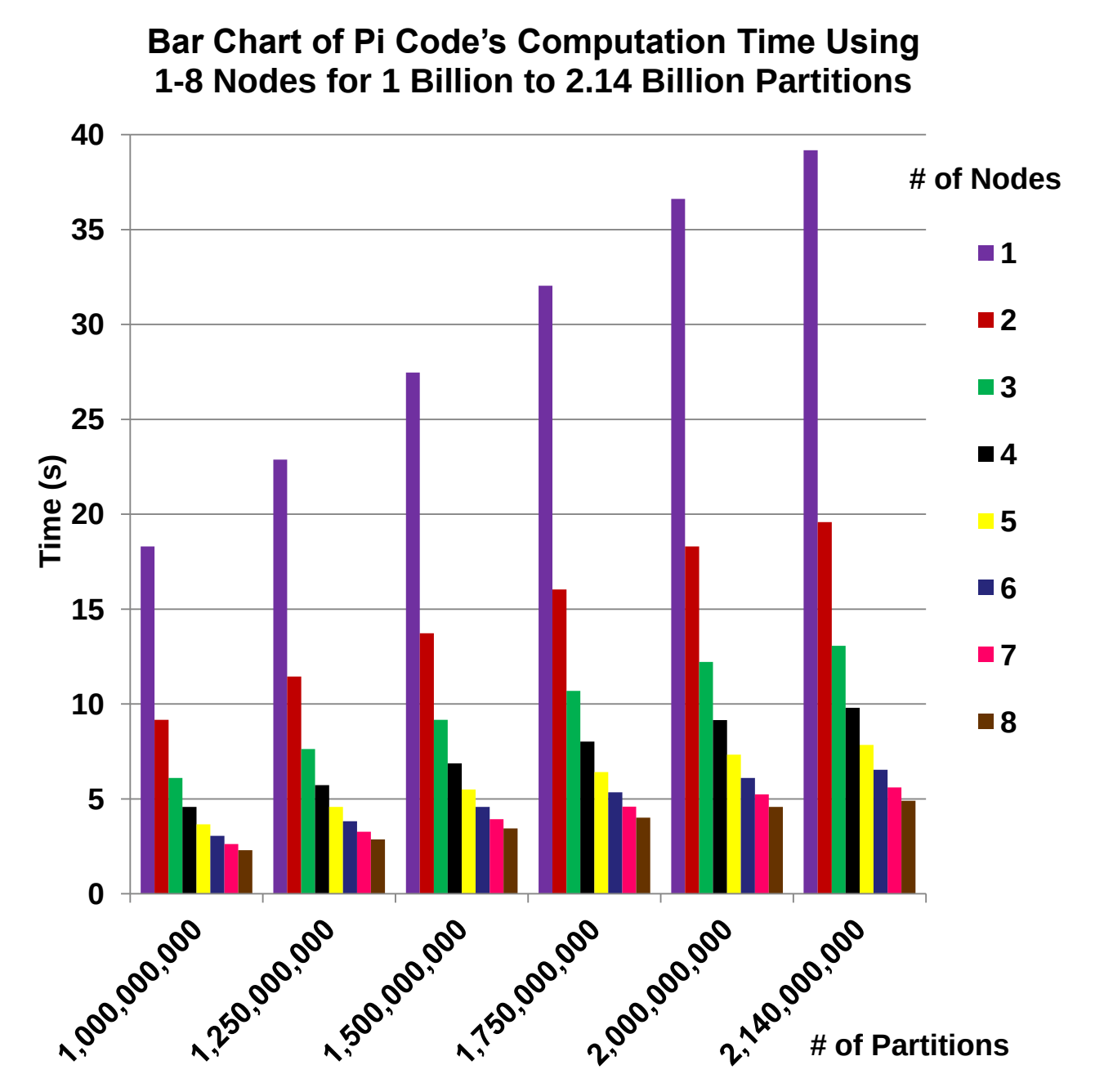

Figure 7. shows a bar chart of the Pi code's computation time vs. the number of nodes for 1 billion to 2.14 billion partitions.

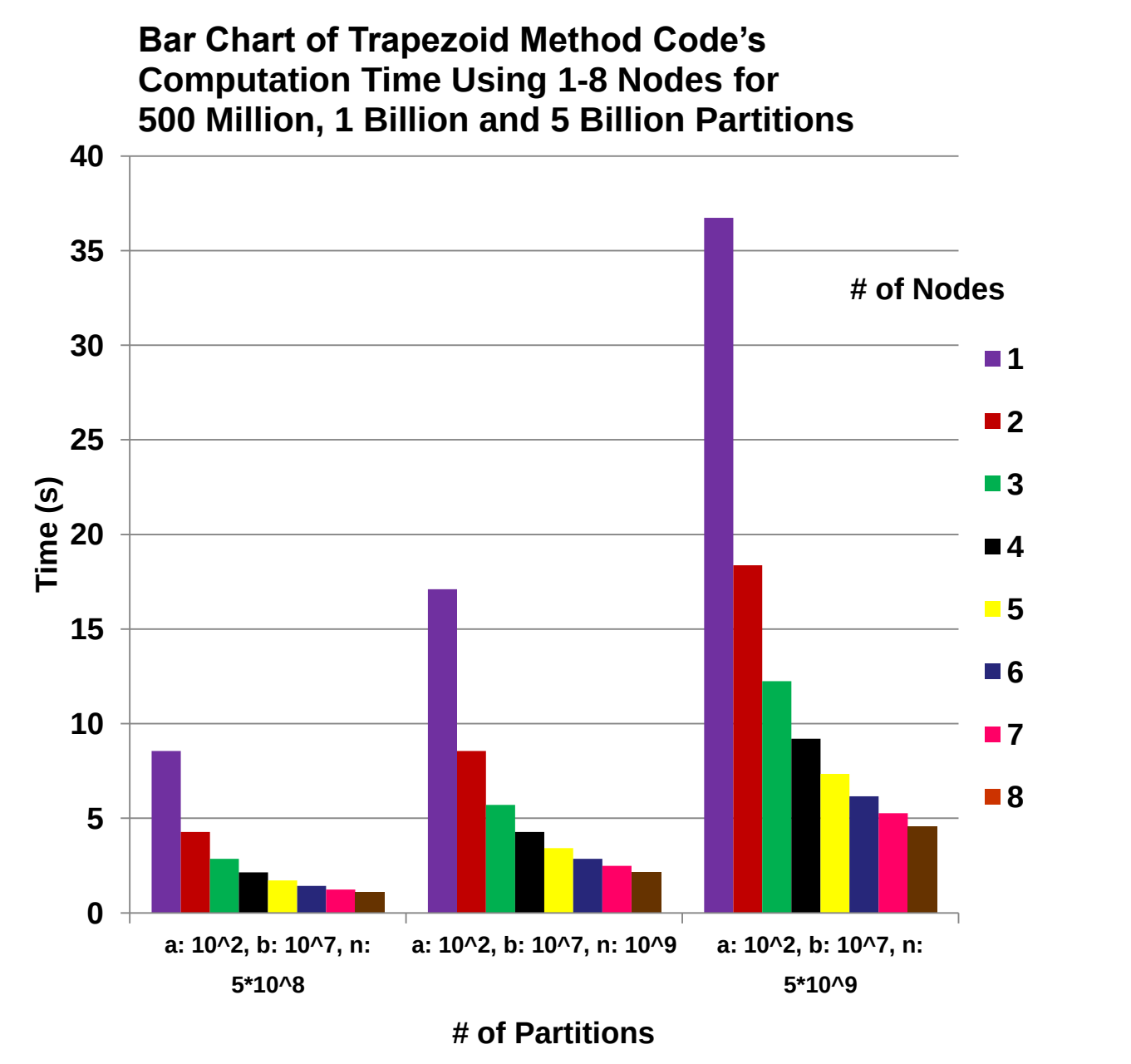

Figure 8. shows a bar chart of the Trapezoid Method code's computation time vs. the number of nodes for 500 million, 1 billion, and 5 billion partitions, between the intervals 100 and 10,000,000.

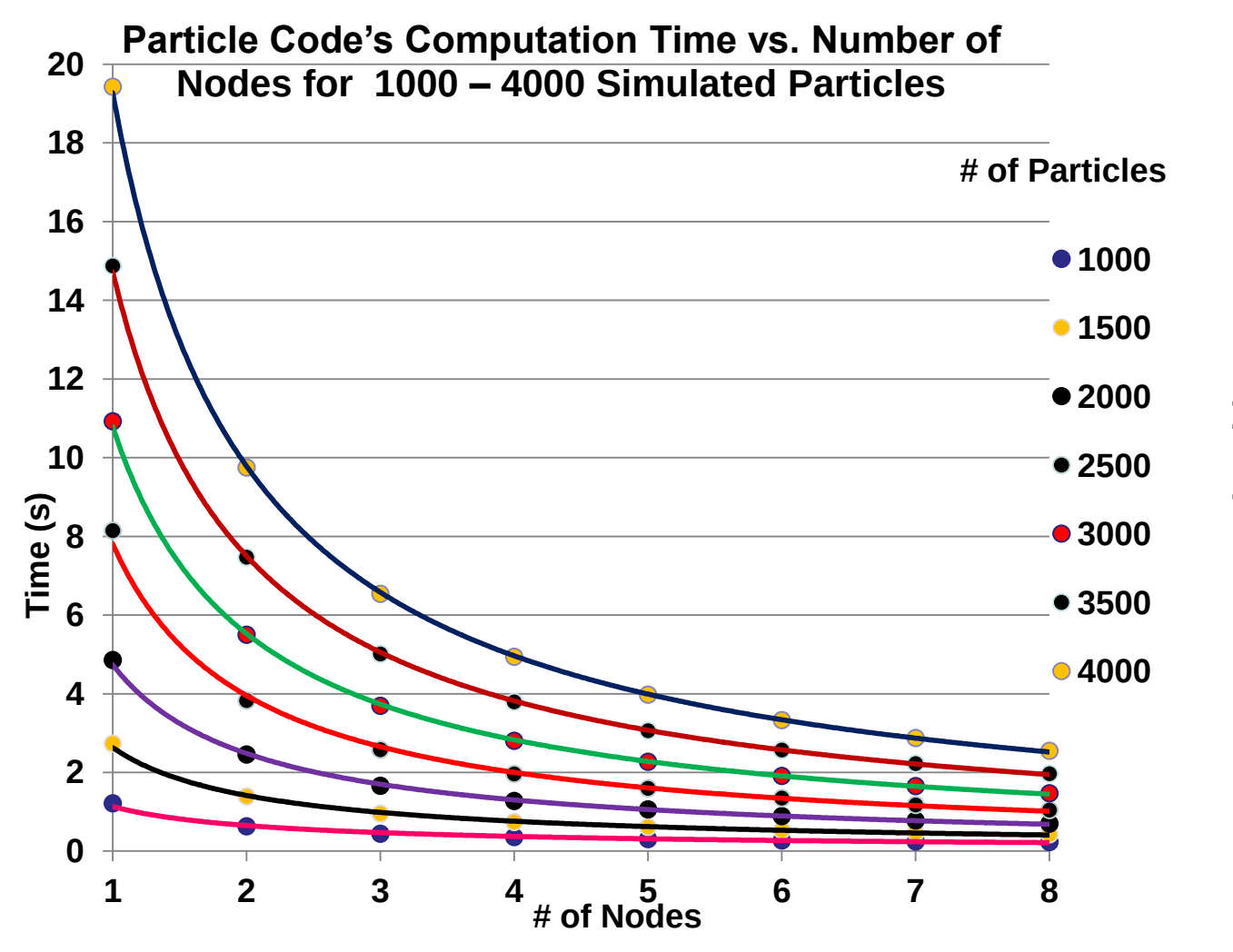

Figure 9. shows a graph of the Particle code's computation time vs. the number of nodes for 1000 - 4000 particles.

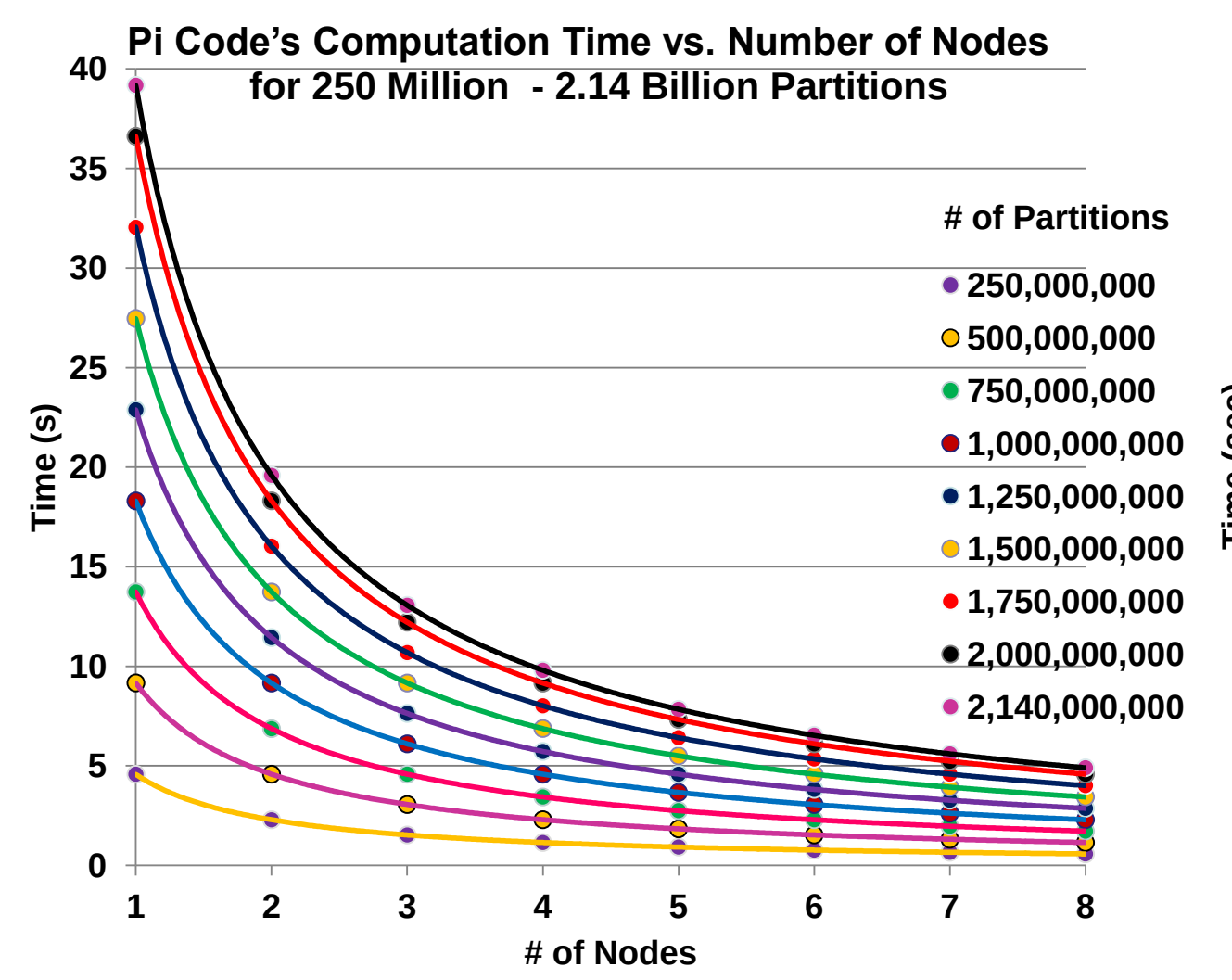

Figure 10. shows a graph of the Pi code's computation time vs. the number of nodes for 250 million – 2.14 billion partitions.

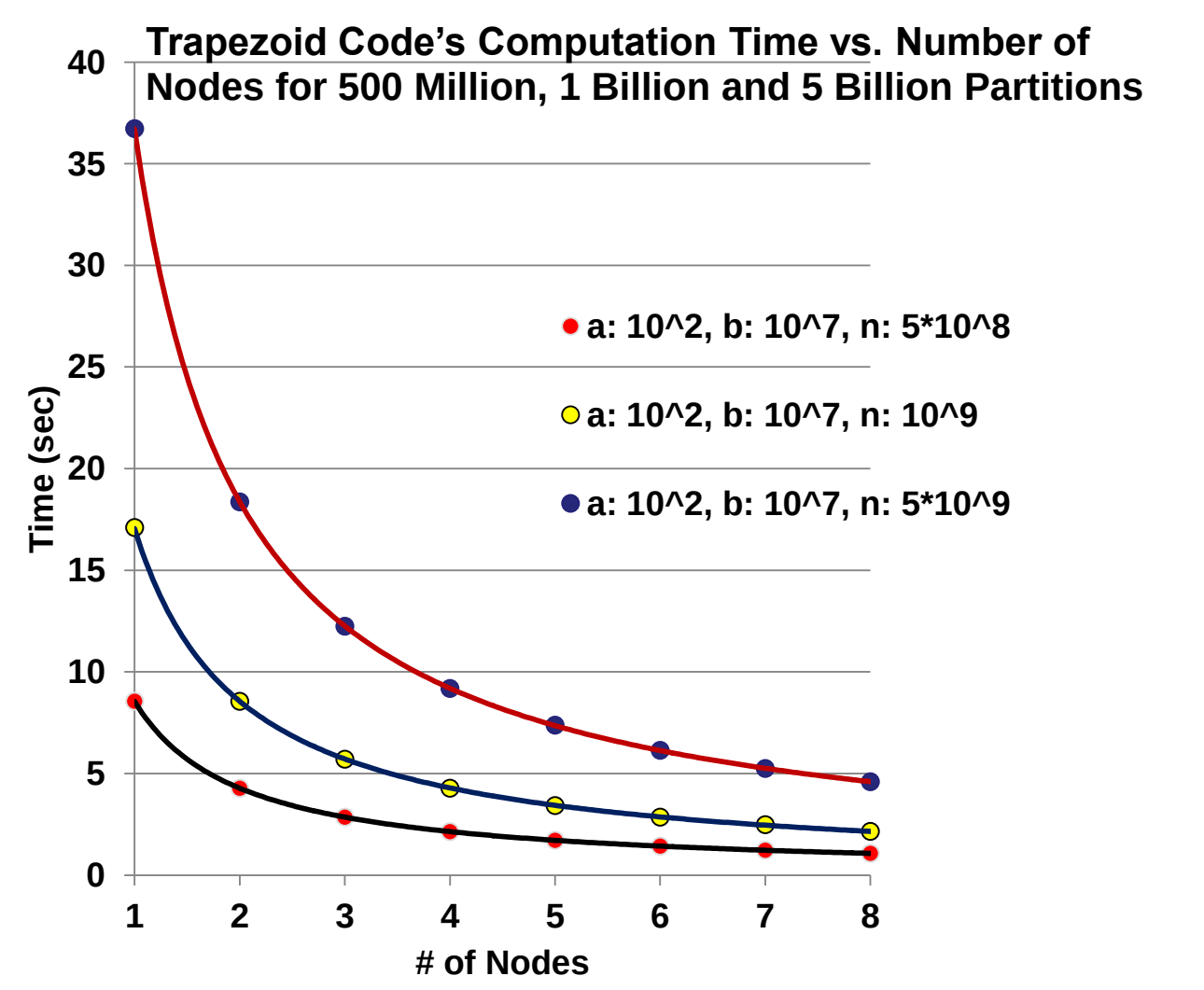

Figure 11. shows a graph of the Trapezoid Method code's computation time vs. the number of nodes for 500 million, 1 billion, and 5 billion partitions.

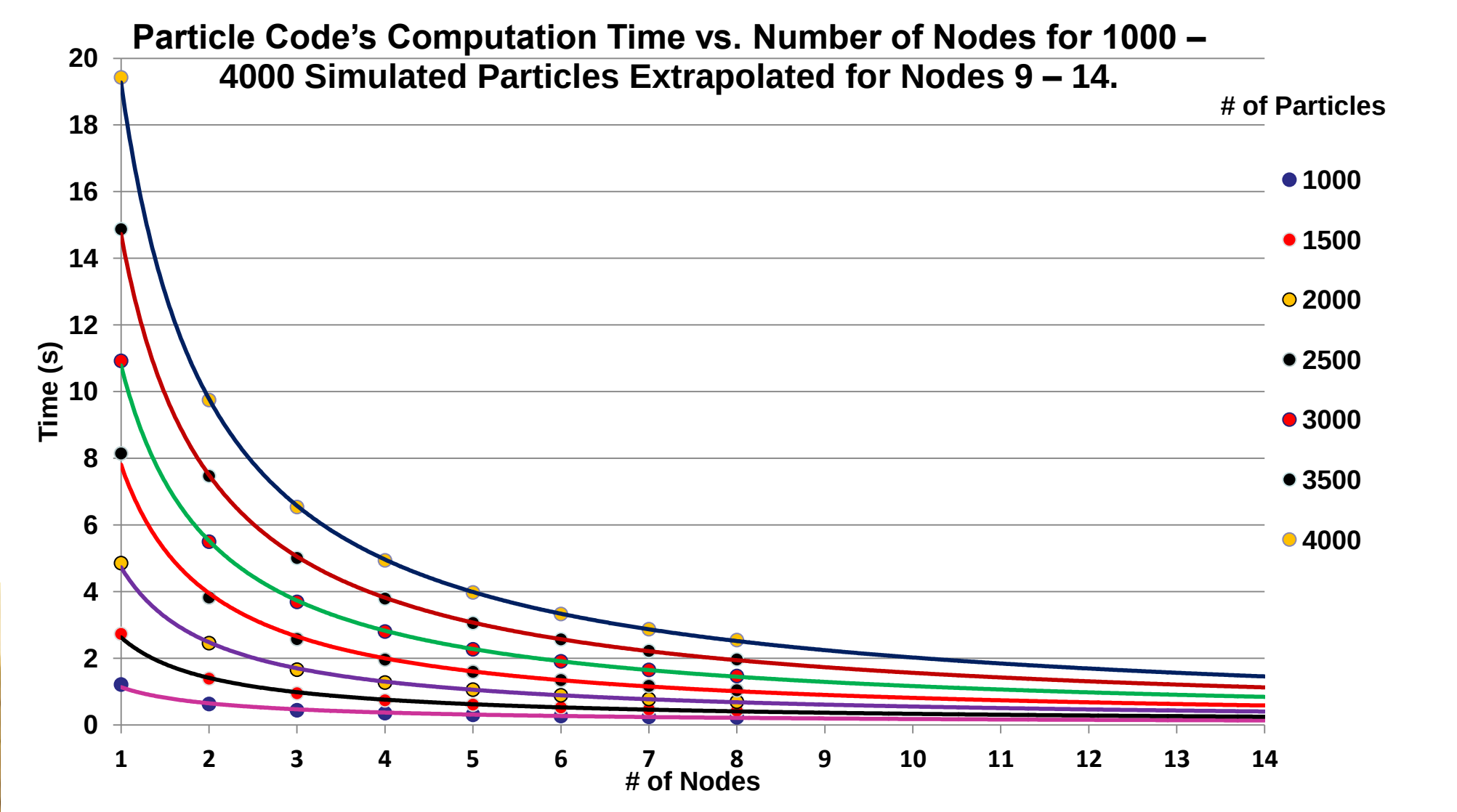

Figure 12. shows a graph of the Particle code's extrapolated computation time from node 9 to 14 for 1000 - 4000 particles.

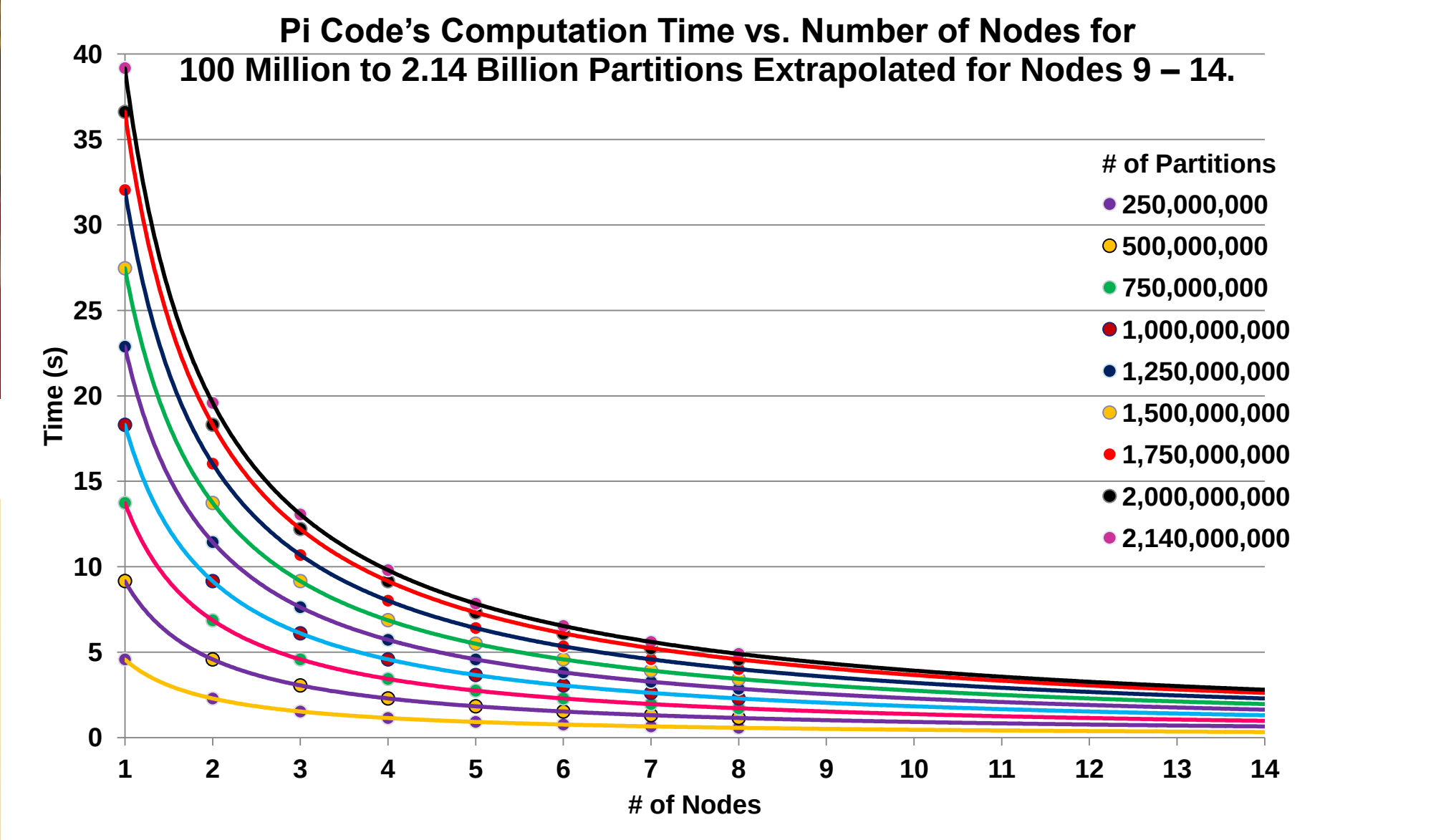

Figure 13. shows a graph of the Pi code's extrapolated computation time from node 9 to 14 for 250 million to 2.14 billion partitions.

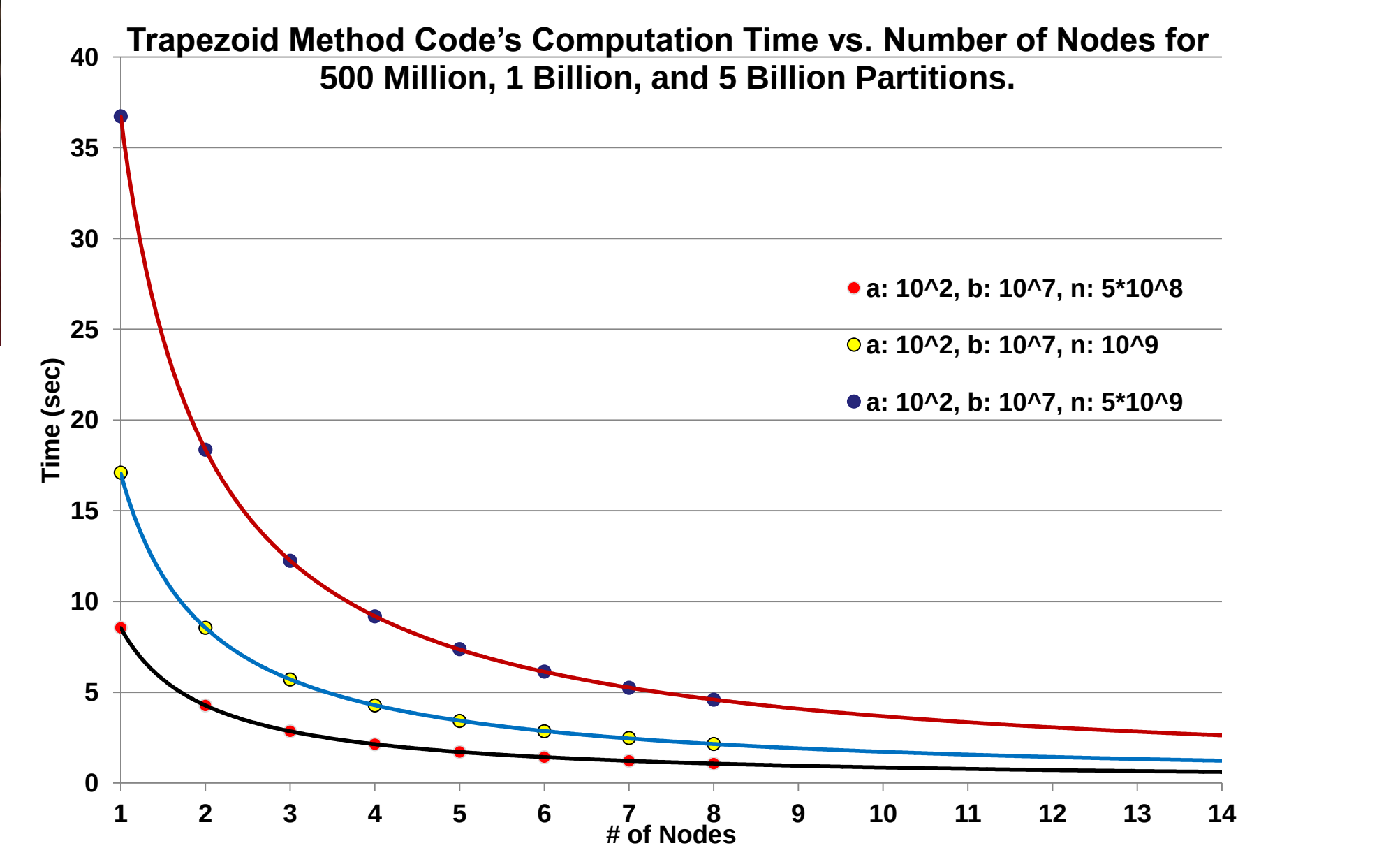

Figure 14. shows a graph of the Trapezoid Method code's extrapolated computation time from node 9 to 14 for 500 million, 1 billion, and 5 billion partitions, between the intervals 100 and 10,000,000.

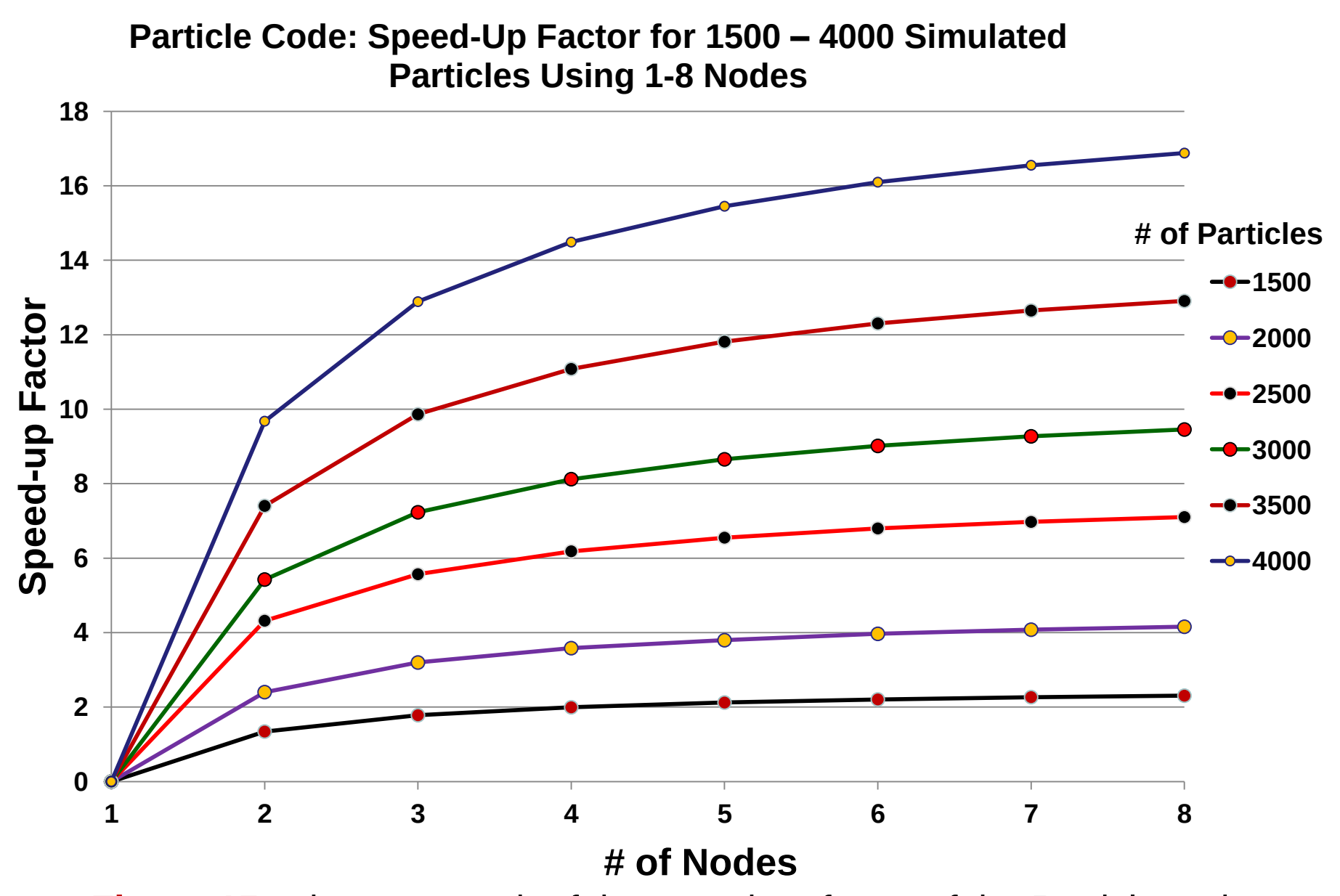

Figure 15. shows a graph of the speed-up factor of the Particle code using 1-8 nodes for 1000 to 4000 simulated particles.

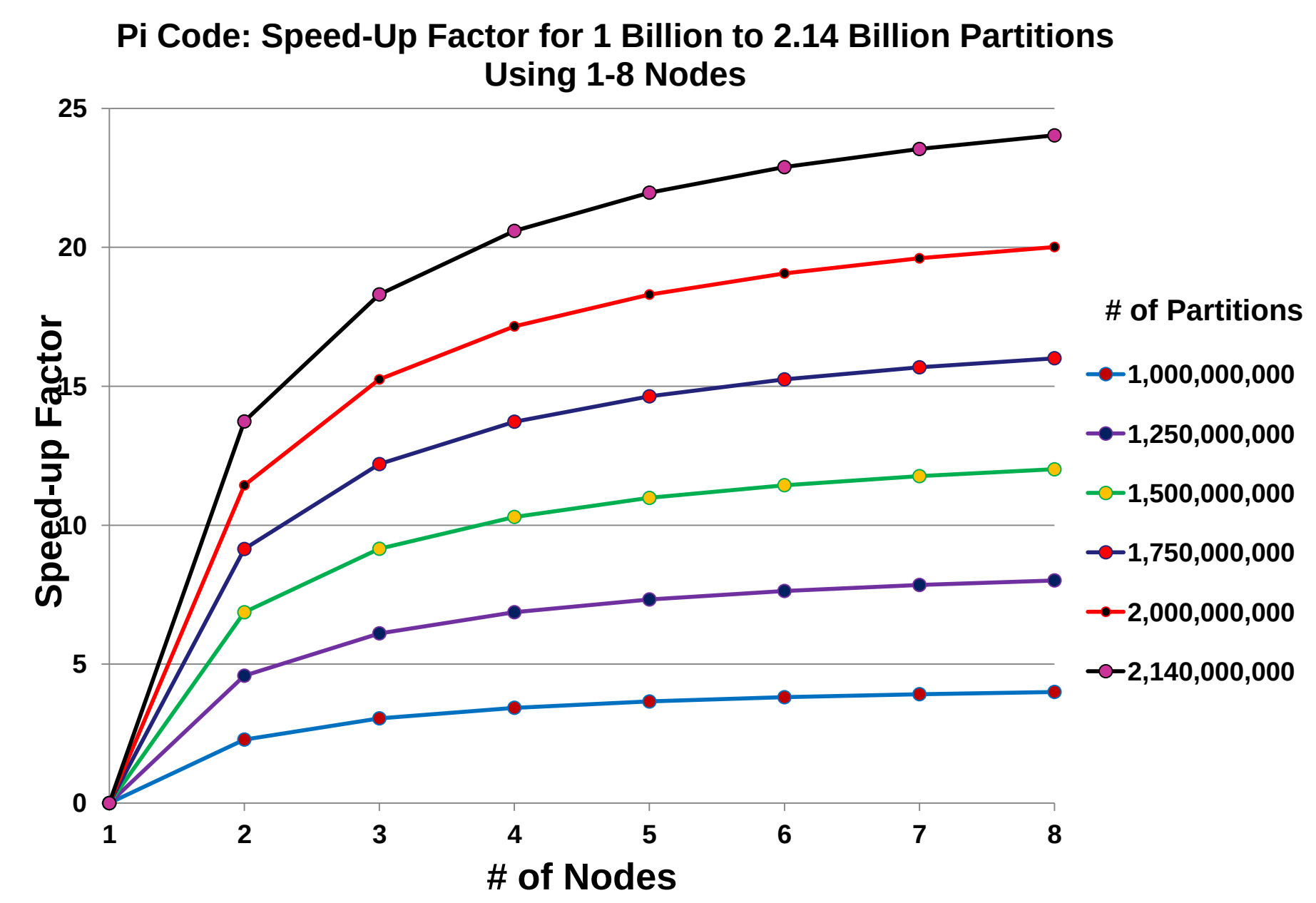

Figure 16. shows a graph of the speed-up factor of the Pi code using 1-8 nodes for 1 billion to 2.14 billion partitions.

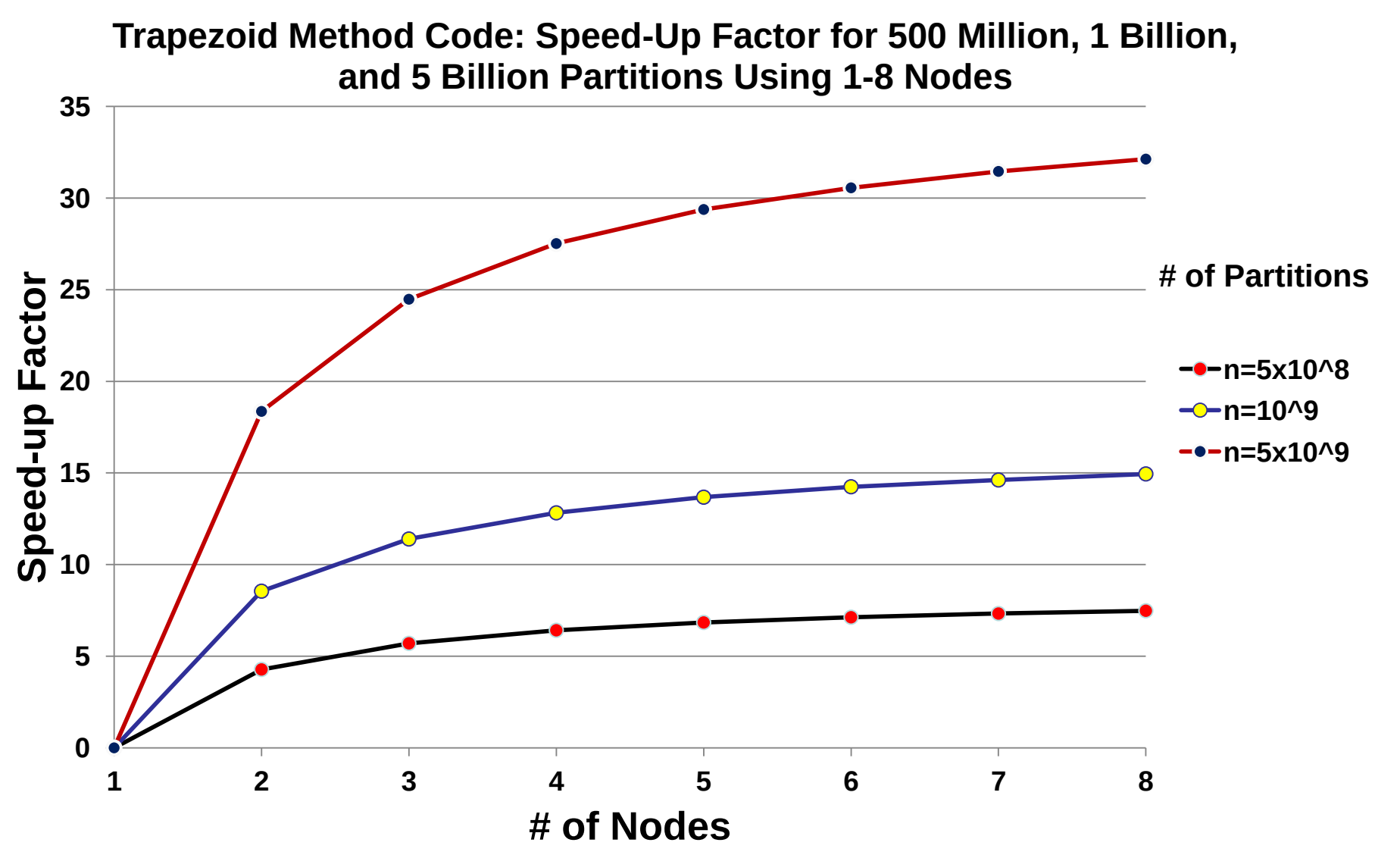

Figure 17. shows a graph of the speed-up factor of the Trapezoid Method code using 1-8 nodes for 500 million, 1 billion, and 5 billion partitions.

Conclusion and Future Plans

As evident from our results, using the Sphinx-1 cluster, we were able to reduce the computation time of our parallel MPI codes by spreading the computation across the nodes of the cluster using the message passing middleware MPI to communicate between all the nodes. As expected, we have demonstrated that using the MPI middleware, all the three parallel codes ran faster as we added more nodes. Also, as evident from graphs 15 – 17, the speed-up factor graphs (i) for the Particle parallel code, we were able to speed up the computation time with 8 nodes (compared to 1 node) anywhere from 2 to 17 times (depending on the number of simulated particles); (ii) for the Pi parallel code, we were able to speed up the computation time with 8 nodes (compared to 1 node) anywhere from 8 to 24 times (depending on the number of partitions); and (iii) for the parallel Trapezoid Method code, we were able to speed up the computation time with 8 nodes (compared to 1 node) anywhere from 7.5 to 32 times (depending on the number of partitions). We noticed that latency (inter-node communication time) played a critical role in our parallel computations as we added more nodes.

In this research project, we implemented data mining techniques using bash scripting and developed the capability of extracting useful information from the computational data. We carried out performance measurements and latency studies using different network switches (100Mbps vs. Gigabit Ethernet). We did not see any gains when we used a 1Gbps switch over of the 100Mbps switch since we were using the same 100Mbps Network Interface Card (NIC) in all our nodes. Therefore, we would not expect any gains in the speed-up factor without a 1Gbps network card inside all the compute nodes. For all three parallel codes, we have also extrapolated the computation time for nodes 9 to 14 (see graphs 12 - 14) in order to predict the trend in computation time, and to determine the best possible computation time without adding additional nodes. We determined the optimal number of nodes necessary to run parallel jobs efficiently is about 14, since the computation time tends to level off after about 14 nodes (as evident from graphs 12 - 14). So, adding more nodes beyond 14 will not further reduce the computation time, hence no gains is expected in the speed-up factor. Therefore, based on our studies, we have determined that the number of nodes that maximizes the efficiency of the Sphinx-1 cluster is 14. We are currently running the same three parallel codes on nodes 9 to 14 of the Sphinx-1 cluster in order to determine how well our expected extrapolated computation time matches with the actual computation time for nodes 9 to 14. Our next goal is to conduct all our computational studies with the Sphinx-2 cluster with the three parallel codes and then compare the Sphinx-2 cluster studies with the Sphinx-1 cluster results.

Acknowledgement

We would like to thank Earle Foskett, our Systems/Network Manager for assisting us in this project. We would also like to thank BU's former IT Director, Fred Lassiter for donating the old PCs that was used for our studies in this research project.