1 Introduction

NTS – Network Topology Simulator is used to simulate a network topology in a parallel processing perspective. This introduction will present parallel processing, compared with other types of networking. The theory concerning parallelization overheads, and network topologies used in parallel processing will be introduced to provide an explanation of why NTS is a useful tool.

1.1 Parallel Processing ([1])

Parallel processing, the method of having many small tasks solve a single problem has emerged into modern computing as a key enabler to computing large problems. Each small task can be executed simultaneously on a separate processor. The last few years have witnessed an increase in use and acceptance of parallel processing, both for high-performance scientific computing and for more general-purpose applications. This is the result of a demand for higher performance, lower cost, and sustained productivity in the computing world. Two developments in technologies have facilitated this acceptance: **massively parallel processors** (MPPs) and the widespread use of **distributed computing**.

1.1.1 Massively Parallel Processors

Massively Parallel Processors are now the most powerful computers in the world. These monster machines combine hundreds to thousands of CPUs in a single large cabinet connected to hundreds gigabytes of memory. MPPs are used to solve complex problems with their enormous computational power, such as global climate modeling and drug development. As simulations run on computers become more realistic, even more computational power is required to produce results within acceptable limits. This is why researchers are turning to MPPs and parallel processing in order to get the most computational power possible.

1.1.2 Distributed Computing

The other major development, distributed computing, is a process whereby a set of computers, connected by a network, are collectively used to solve a single large

problem. More and more organizations have high-speed local area networks connecting many general-purpose computers or workstations. The computers connected to these local area networks may have a combined computational power that may exceed the power of a single high-performance computer. These companies can use this combined power instead of investing in such a high-performance computer.

The main attraction to distributed computing is cost, while mimicking the power of MPPs. A single MPP computer is extremely expensive. By contrast, users see very little cost executing their problems on an already existing network of local computers. Distributed computing users cannot usually achieve the raw computational power of a MPP, but are able to solve bigger problems than would otherwise be impossible on a single workstation within realistic time periods. These networks used to mimic supercomputers are known as a **cluster**.

1.1.3 Communicating Processes

Each of these two technologies, MPPs and distributed computing, take advantage of executing multiple operations at the same time, rather then being limited by the speed of a single "sequential" computer. Due to this parallel execution, and common to all parallel processing, is the notion of communicating processes. Data must be exchanged between the cooperating tasks. For this purpose, several paradigms can be used, including shared memory, parallelizing compilers, and **message passing**. The latter has become the paradigm of choice, in view of its wide use over the range of multiprocessors, applications, languages, and software systems that implement it.

Message passing, in between various threads running on different processors, if not different computers, brings its own set of problems: sequencing, storage, and deadlock management, just to name a few. Techniques exist to counteract each problem. Once these problems are resolved using the appropriate network technology, a parallel program can be run on the network of processors.

1.2 Parallel Study ([3])

1.2.1 Scalability

In a sequential program, the efficiency of the program can be defined in terms of the time taken to execute it compared with the size of the task being performed. This simplistic view can be used for parallel processing theory. With parallel processing, the task is divided in between a number of processors. The **scalability** of a program is defined as the measure of its ability to achieve proportional performance increase compared to the number of processors employed.

1.2.2 Speedup

The **speedup** (S) of a parallel program is defined as:

$$S = \frac{T_{seq}}{T_{par}}$$

where T_{seq} is the sequential runtime or the time taken to execute the program sequentially, and T_{par} is the parallel runtime or the time taken to execute the program once it has been parallelized. Ideally:

$$T_{par} = \frac{T_{seq}}{N}$$

where N is the number of processors the task is separated on. To see the scalability of a program, the speedup of a program is plotted against the increasing number of processors used in running the parallel algorithm. As can be seen from the Figure 1, the speedup does not increase linearly with the number of processors.

1.2.3 Efficiency

The **efficiency** of a parallel system is defined as:

$$E = \frac{S}{N} = \frac{N \cdot T_{seq}}{T_{par}}$$

Each processor involved in the calculation works for E% of the total time, the rest of the time is lost due to parallel overhead. The efficiency represents how well each processor is used. Initially adding extra processors or more power to solve the problem will increase the efficiency as the speedup will increase. However a theoretical threshold

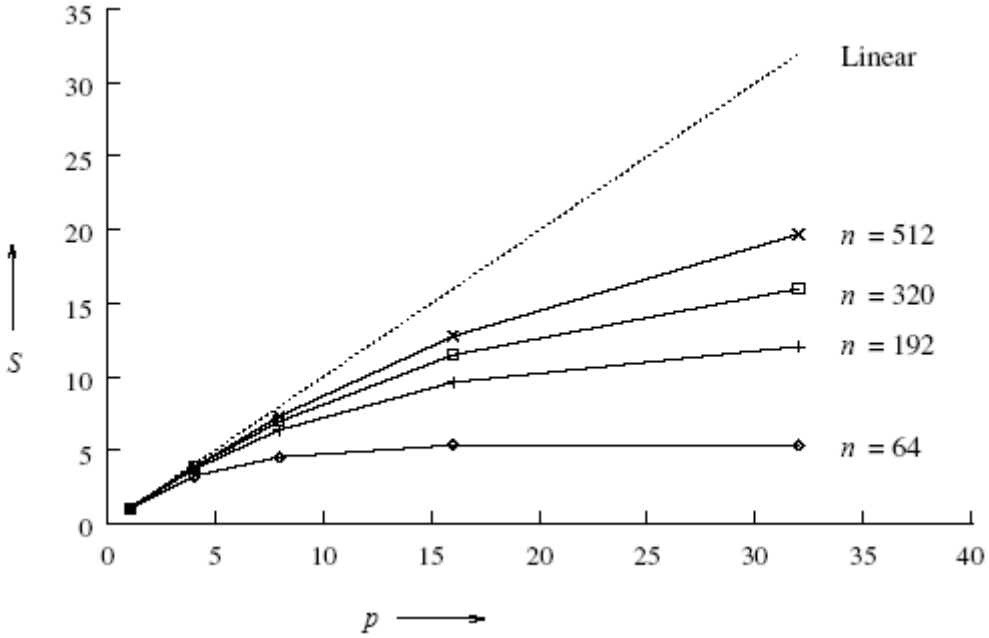

Figure 1: Speedup versus the number of processing elements for adding a list of numbers [2]

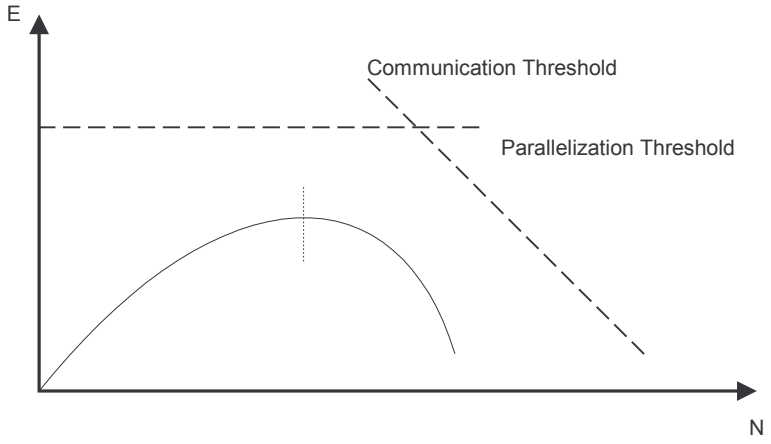

Figure 2: Efficiency vs. number of processors

exists where once this limit is surpassed, the overheads created by the extra processors will have increased to such an extent resulting in a decrease in efficiency (Figure 2).

1.2.4 Coefficient of Parallelization

An application cannot be completely parallelized, or partitioned exactly so that it runs best on multiple processors. A **coefficient of parallelization** (p) is therefore introduced to dictate how well a certain application can be parallelized, resulting in the following equation:

$$T_{seq} = p \cdot T_{seq} + (1 - p) \cdot T_{seq}$$

for a sequential application, or the total time to execute the application on a single processor. This equation is displayed in such a form to better compare with the parallel equation:

$$T_{par} = p \cdot T_{seq} + \frac{(1 - p) \cdot T_{seq}}{N}$$

where N is equal to the number of processors the application is divided over. In fact in all parallel algorithms there is a time of setting up, which happens on a sequential basis, and only part of the algorithm can properly be divided among the processors. The parallel equation is erroneous though, as it implies that communication in between the various processes is immediate:

$$T_t = p \cdot T_{setup} + \frac{(1 - p) \cdot T_{seq}}{N} + parallel\ overhead$$

This overhead is due to the communication laps. The communication required between threads, is different for each parallel algorithm, but the communication overhead is also dependent on the communication hardware, software, the load imbalance and the arrangement of the layout of the network being used. The speed of the hardware is directly proportional to the investment made in it, the communication software is usually tailor made for the hardware, and it is assumed that the load is properly balanced most of the time. These will be judged either optimal, or as factors decided outside the scope of current investigation and therefore will be overlooked for the current study. This leaves the network layout or topology, therefore:

$$T_t = p \cdot T_{setup} + \frac{(1 - p) \cdot T_{seq}}{N} + \mathrm{f}(topology)$$

1.3 Topology

A **store and forward** network is one where each node has to receive the full message before it can be forwarded towards its final destination. **Pass through** networks exist where a message is forwarded on immediately the moment its next destination is determined without the whole message having been received. These, while being faster, are less common and more difficult to implement.

1.3.1 Full Interconnect

There are many ways of setting up the interconnections between computers on a network. Mainly, the number of other computers each computer or node can be physically connected to defines the layout or topology of the network. The most efficient topology is the **full interconnection**, where every node is connected to every other. Usually a crossbar implements this (Figure 3).

1.3.2 Static Networks

Connections in between computers or network nodes are a significant cost in a network. When a network starts having many nodes it becomes expensive or physically impossible to setup a full interconnection network. Other factors may limit the full interconnection of all nodes, such as distances in between nodes. The solution is to remove some of the interconnections and create a **static** network, where connected nodes will forward messages destined to nodes, which are not physically connected immediately to each other (Figure 4).

Various topologies or formats of interconnections for networks may be used. Which of these is used mostly depends on the total number of nodes present. A **hypercube** (Figure 5), with enough dimensions, removes the least number of interconnections and brings interesting properties for programmers to take advantage of. The **diameter**, the maximum number of hops to reach a possible destination on the

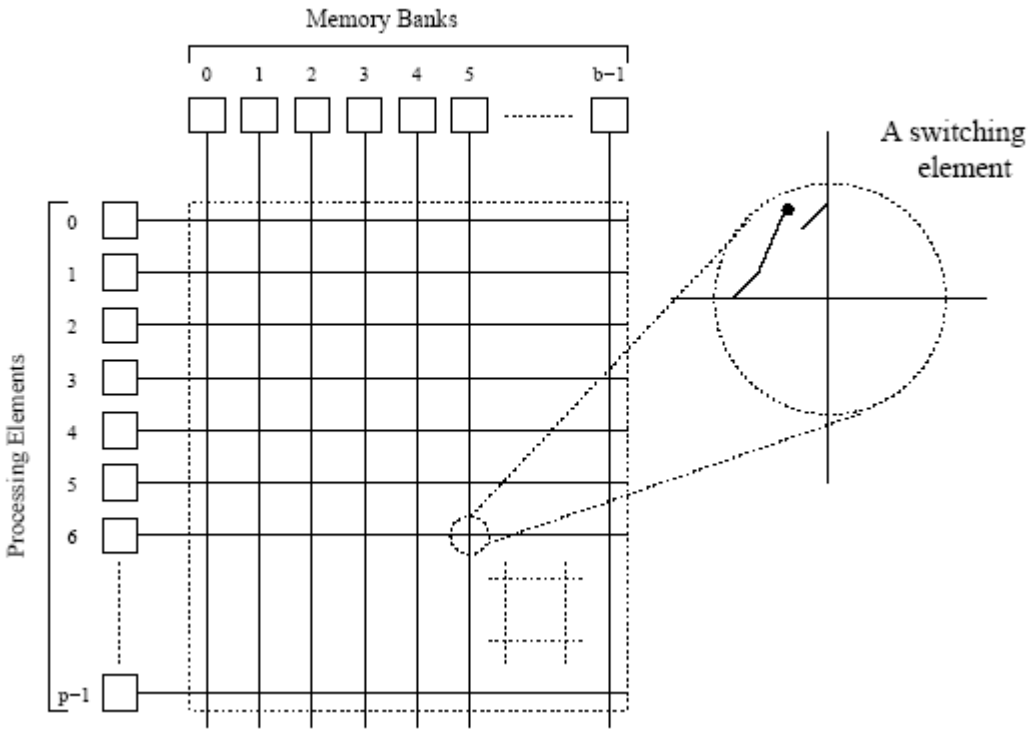

Figure 3: A completely non-blocking crossbar network connecting p processors to b memory banks [2]

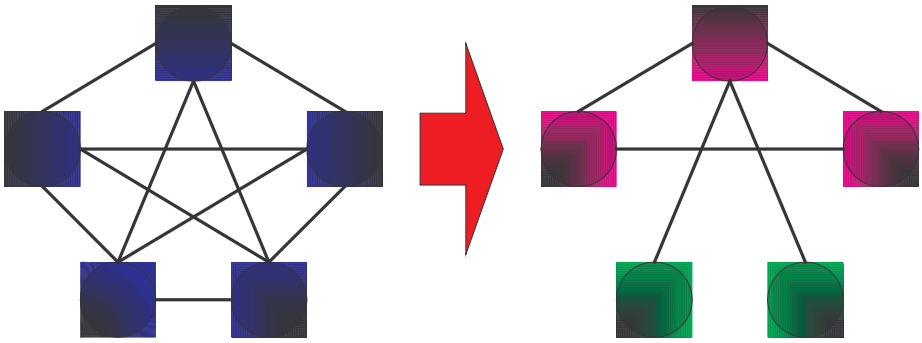

Figure 4: Removing links from a full interconnect network to make a static forward network

network, is the smallest on these types of networks. The **ring** topology is at the other extreme, taking away the most interconnections, so that communications have to be forwarded on average the most, i.e., to a node on the other side of the circle. The **mesh** (2D or 3D, with wraparound or not) (Figure 6) is usually the most frequently used. It combines the advantages of the ring, i.e., not too many interconnections, but retains enough of them, that the diameter is the square root of half the number of nodes squared rather then half the number of nodes as in the simple ring. Also with wraparound links,

the diameter is vastly cut down, but wraparounds do introduce some complex routing issues.

As can be seen, the topology of the network directly affects the number of hops that have to be taken on average to get a message from one node to another on the network. This, of course, defines the average speed of delivery of a message and therefore affects the overall efficiency of the parallel algorithm.

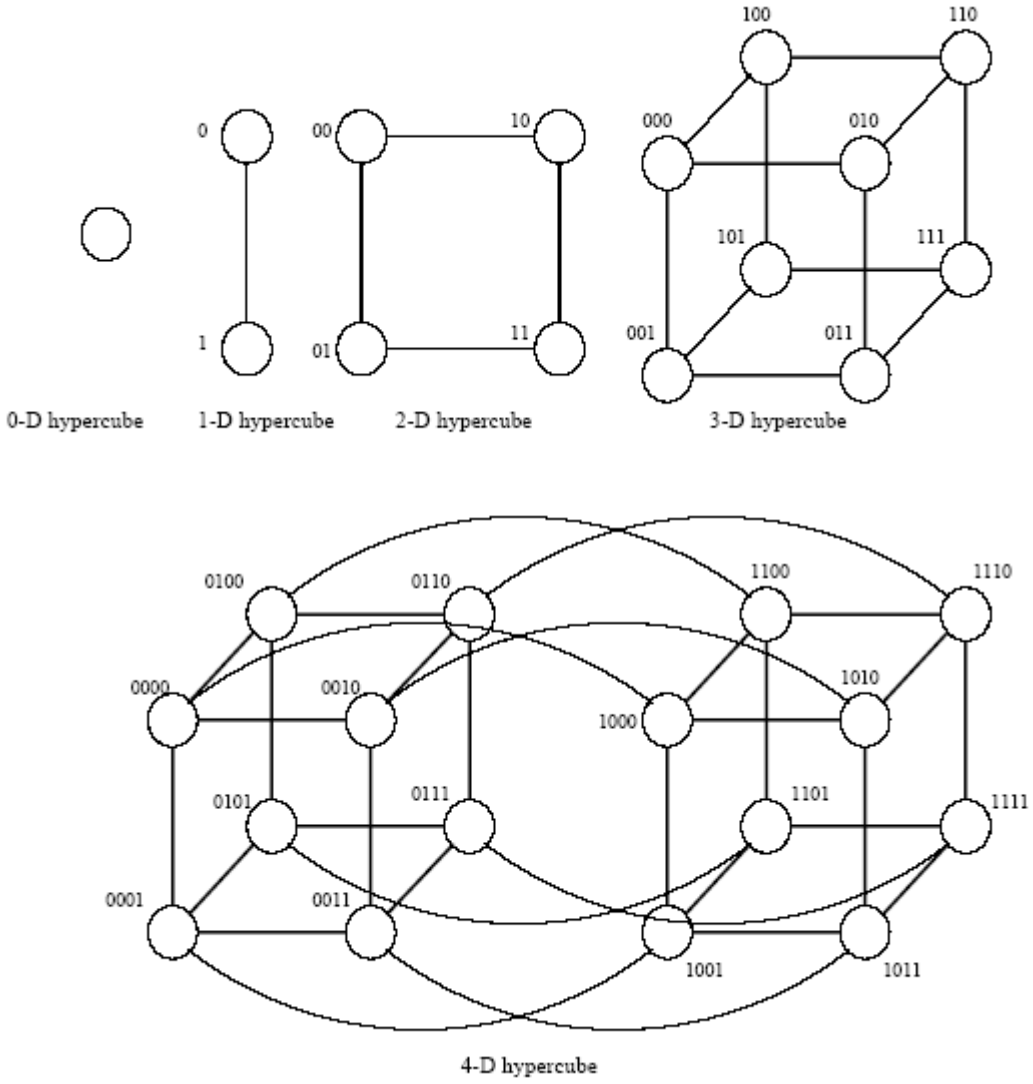

Figure 5: Construction of hypercubes from hypercubes of lower dimension.[2]

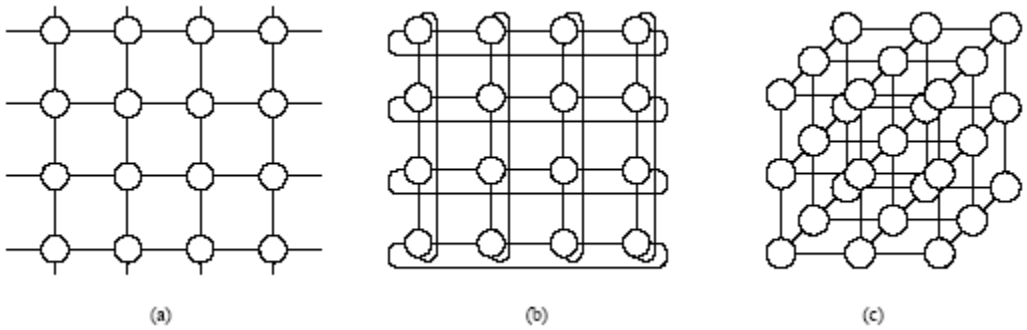

Figure 6: Two and three-dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.[2]

1.4 Why NTS

In quasi-all parallel algorithms, processes need to exchange data with other processes. This communication procedure between processes can significantly influence the efficiency of the parallel algorithm due to the introduction of overheads. The actual structure of the network connecting the processes has a great impact on these interactions, as shown before, and therefore on the overall efficiency.

The main purpose of this simulation is to implement a software solution for simulating different kinds of network topologies over a static network or a Network Topology Simulator (NTS). The basic idea is to install a **router** program on each computer that will behave as a switching node. Hence, the construction of the interconnection network will happen just by adding interconnection **links** between the routers. Every router will keep track of the possible routes in a **routing table**, which will be updated with dynamic data exchange between routers. The communication process between routers will be implemented with **PVM**[1].

The routing process will be based on a **vector-distance routing protocol** (like **RIP**[2]), in this case a hop-by-hop routing protocol. On every router, only the routes with the lowest number of hops will be held, in the routing table. The network will be dynamic in the sense that the actual topology of the network can be changed during execution, i.e., some links can be deleted or additional links can be configured after the

[1] PVM: Parallel Virtual Machine
[2] RIP: Routing Information Protocol

initial configuration, and the network will update itself dynamically. All changes to the network topology will happen through a **Master** program.

Finally, the interface between this "dynamic network cloud" (also sometimes called "indirect network") and the **client** code (i.e., the processing process which is a piece of a parallel program) will happen through an overloaded PVM library. NTS will be an extra layer to redirect messages in between the PVM message communication and the client (Figure 7).

To summarize, this work has been divided into three blocks:

- The dynamic routing program,
- The communication library (interface for the client), based on an overloaded PVM,
- The Master program to configure and manage the routers.

These three parts and their use are detailed in the following chapters. With NTS it is possible to simulate any imaginable store and forward topology: from the simple circular design, to the most complicated mesh, to letting your fantasy run wild with figurative network designs. All that is needed is for PVM to be able to run on the underlying network. Figure 8 represents the concepts schematically on a 3 by 3, 2-Dimensional mesh with wraparound links layout.

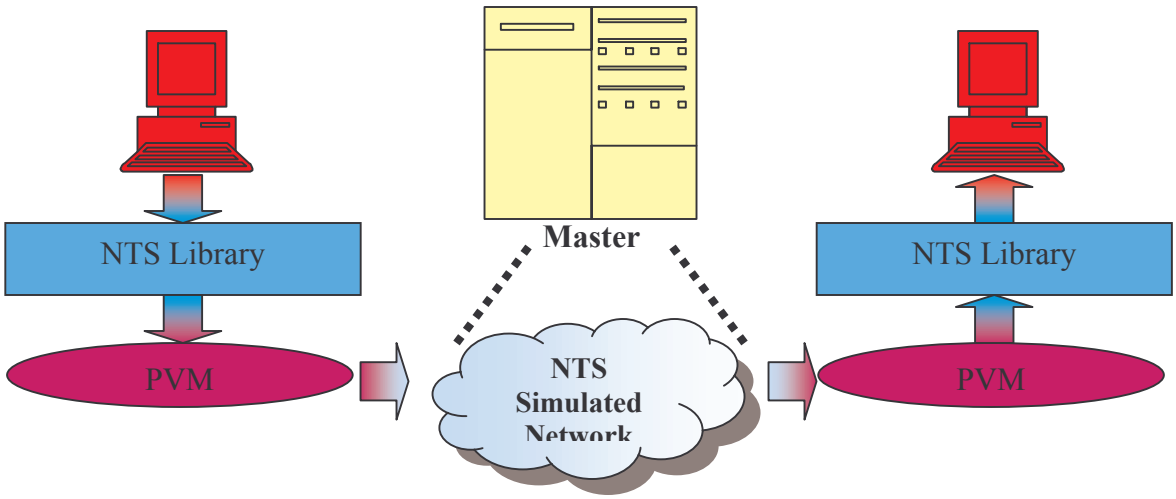

Figure 7: The layers of NTS

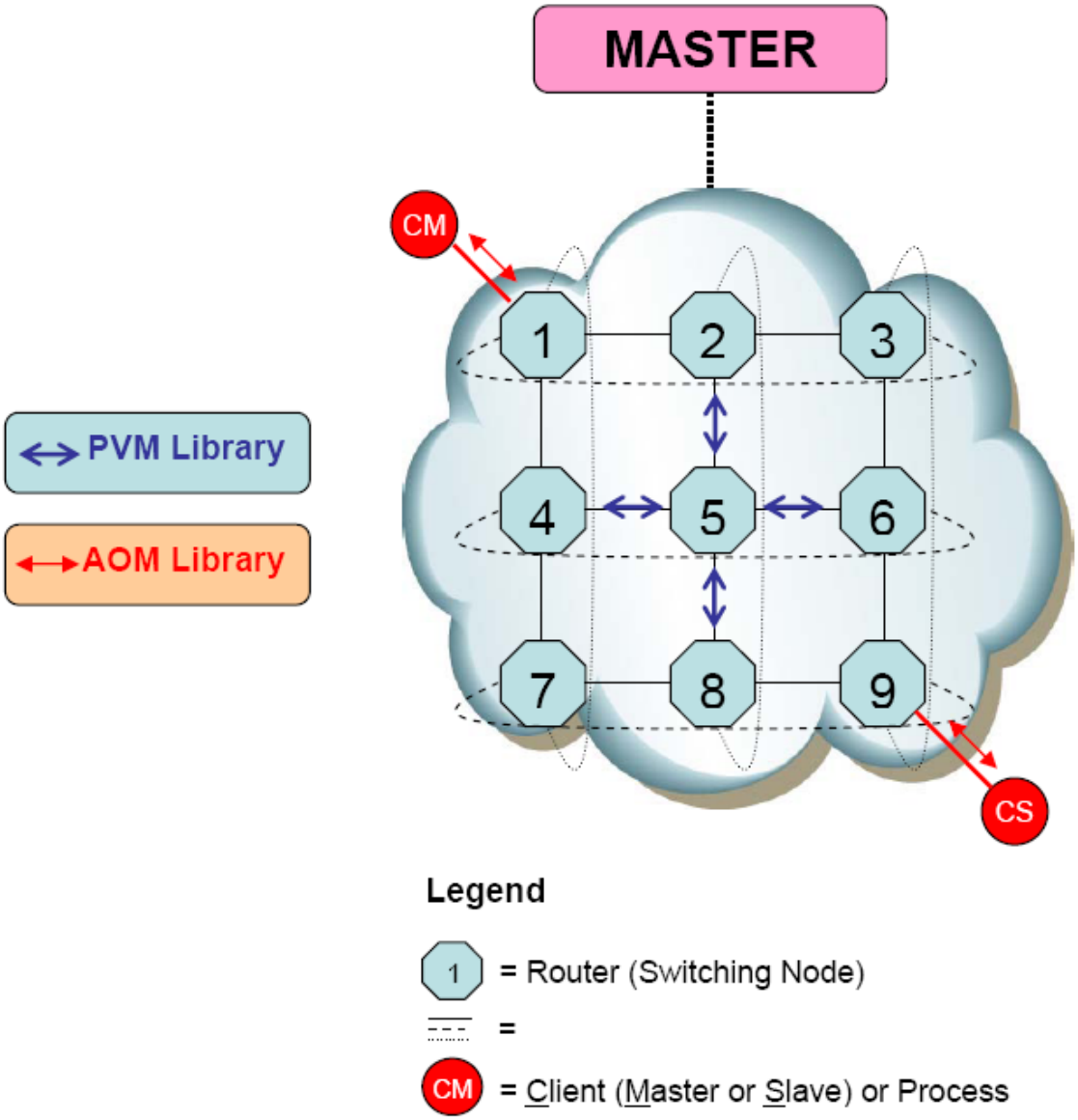

Figure 8: Representation of a (dynamic) 3-by-3, 2-D mesh with wraparound ([6])