

## 2 History

### 2.1 Performance Growth for Practical Parallel Computing

Before we begin our discussion of the DAISy project we should first show the aspects of how we got here. It took over 20 years for parallel computers to move from the laboratory to the marketplace. As mentioned before recent development in hardware and software have allowed for MPP type computation to be performed on PCs. Figure 1 shows that the performance-growth rate for minicomputers, mainframes, and traditional Supercomputers has been just under 20% per year, while the performance-growth rate for microcomputers has averaged 35% per year.

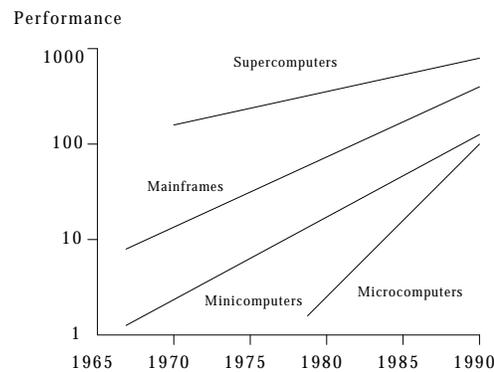


Figure 1. Performance growth of four classes of computers since 1965 (Courtesy of Hennessy and Patterson 1990). From M. J. Quinn's "Parallel Computing, Theory and Practice" [1994][5].

Technological advances in microprocessors [5] is solely due to their initial lack of sophistication. By the mid-1970's the fundamental architectural advances (bit-parallel memory, bit-parallel arithmetic, cache memory, channels, interleaved memory, instruction lookahead, instruction pipelining, multiple functional units, pipelined functional units, and data pipelining) had already been incorporated into supercomputer designs, and thus incorporating them into microprocessors was inevitable.. Architectural advances, coupled with reduced instruction-cycle times, have led to impressive performance gains for microprocessor class computers.

The convergence in relative performance between microcomputers and traditional Supercomputers has led to the development of commercially viable parallel computers consisting of tens, hundreds, or even thousands of microprocessors. At peak efficiency, microprocessor-based parallel computers such as Intel's Paragon XP/S™, MasPar's MP-2™, and Thinking Machines' CM-5™, exceed the speed of traditional single-processor Supercomputers, such as the Cray Y/MP™ and the NEC SX-3™.

There are many design features associated with harnessing the power of MPPs. What is important to note is how specific MPP design features virtually map onto the architecture of a networked cluster of workstations. Since a networked cluster of workstations is a distributed system it is reasonable to discuss such systems at this time. Later we will discuss the virtual overlap of design features common to both MPP systems and cluster systems.

## 2.2 Distributed System

A distributed system [6] consists of a collection of autonomous computers linked by a computer network to achieve a goal. Figure 2 shows an example of a simple distributed system. Existence of multiple autonomous computers are transparent to the user. In other words, the user of a distributed system is not aware that there are multiple processors; it looks like a virtual uniprocessor. The DAISy project defines the present DAISy cluster distributed system as a collection of commercially available workstation compatible personal computers working together on a local area network, an arrangement sometimes called a processing *farm*, whose added processing units are used to increase the number of jobs that can be handled concurrently.

Due to its connectivity, communication is much slower in a LAN-connected cluster than in massively parallel processing systems. This type of system tends to attract applications that have only modest communication requirements among the parallel components (in other words, improve *throughput* rather than *turnaround time*). To recall, *throughput* is the number of results produced per time unit, and *turnaround time* is the amount of time it takes to receive a response (result) after initiating a request (process). Some parallelism researchers regard such applications as bordering on being “embarrassingly parallel.” Due to the recent development of fast networking backbones (i.e. FDDI, Fast Ethernet, HPPI, ATM) other types of applications that only MPPs have been able to run are coming forth. As we will see later in this paper with the recent release of the NAS Parallel Benchmarks 2.0 [7] running on top of MPI (Message Passing Interface) [8].

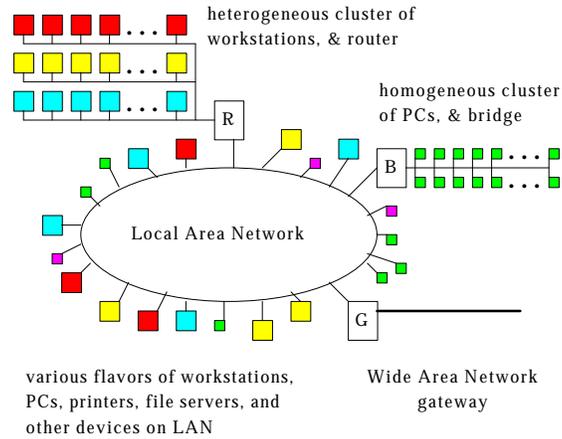


Figure 2. A simple distribute system

The question that arises is, is DAISy a parallel processing system or a distributed system? In the text, “*Highly Parallel Computing*” [1994], Almasi & Gottlieb [6] define a parallel processor as “A large collection of processing elements that can communicate and cooperate to solve large problems fast.” The downfall in the current DAISy design is its network. This downfall limits the type of parallel processing technique to those which again look at improving *throughput* rather than *turnaround time*. Because of this limitation the DAISy cluster will be viewed as a distributed system. Though with fast ethernet and new workstation speed CPUs that view may change.

### 2.3 Distributed System Key Characteristics

Basically, there are six key characteristics [9] that are responsible for the usefulness of a distributed system. These are *resource sharing*, *openness*, *concurrency*, *scalability*, *fault tolerance* and *transparency*. These key characteristics will be reviewed along with a description as to how they correspond to the DAISy cluster.

- 1) *Resource Sharing* - Resources in terms of hardware and software and the potential benefits from sharing these resources in a distributed computer system are significant. For sharing hardware (local disk space, local main memory, local cache memory, and CPU time) with other nodes on a distributed system network software must be configured and running. In DAISy user directories are shared resources to all nodes because all user directories are NFS (network file system) mounted from DAISy’s master node.

In distributed computing the client/server model and the message passing model are methods used in sharing resources and will be examined later in this paper.

- 2) *Openness* - Open distributed systems are based on the provision of a uniform interprocess communication mechanism and published interfaces for access to shared resources.

The UNIX operating system was an early example of a more open system design. Reason:

1. *for application developers*: they have access to the entire range of facilities that the system offers;
2. *for hardware vendors and systems managers*: the operating system can be extended relatively easily to add new peripheral devices or network controllers;
3. *for software vendors and users*: it is hardware-independent. Software can be written to be run without modification (theoretically) on many different manufacturer's computers.

The DAISy operating system, FreeBSD, is a UNIX compatible OS and is discussed in the system hardware/software configuration section of this paper.

- 3) *Concurrency* - Concurrency and parallel execution arise naturally in distributed systems. In a distributed system, such as DAISy, that is based on the resource-sharing model, opportunities for parallel execution occur for two reasons.

1. Many users simultaneously invoke commands or interact with application programs.
2. Many server processes run concurrently, each responding to different requests from client processes.

- 4) *Scalability* - The key in scalability is that the system and application software of a distributed system should not need to change when the scale of the system increases. An algorithm is scaleable if the level of parallelism increases at least linearly with the problem size. An architecture is scaleable if it continues to yield the same performance per processor. Algorithmic and architectural scalability are important, because they allow a user to solve larger problems in the same amount of time.

The NAS Parallel Benchmarks 1.0 [10] Kernel EP (embarrassingly parallel) is a great example of an application that is scaleable, and therefore, demonstrates the scalability of the DAISy cluster.

The results of this application are shown in the benchmark results section later in the paper.

- 5) *Fault Tolerance* - Computer systems sometimes fail. The design of fault-tolerant computer systems is based on two approaches:

1. *hardware redundancy*: the use of redundant components;
2. *software recovery*: the design of programs to recover from faults.

The allocation of redundant hardware required for fault tolerance can be designed so that the hardware is exploited for non-critical activities when no faults are present. The presence of both 10BASE-2 Ethernet and 100BASE-TX Fast Ethernet backbones on the DAISy cluster is a good example of redundant hardware. With the loss of one of the backbones, access to all nodes is still available.

Software recovery involves the design of software so that the state of permanent data can be recovered or 'rolled back' when a fault is detected. This is the responsibility of the applications programmer and should always be taken into consideration when writing parallel code.

- 6) *Transparency* - Transparency is the concealment of the separation of components in a distributed system from the user and the application programmer so that the system is perceived as a whole rather than as a collection of independent components.

Due to the nature of the DAISy cluster, *resource sharing, openness, concurrency, scalability, fault tolerance, and transparency* are relatively inherent. The reason being is that the **DAISy** cluster is a networked cluster of 16 Intel Pentium 90MHz workstations running very inexpensive UNIX compatible software.

## 2.4 Generalized Parallel Processor

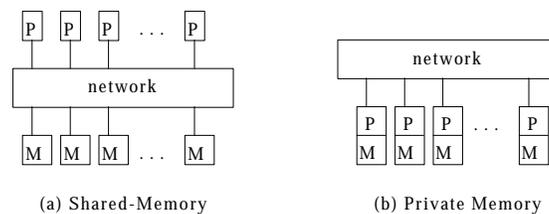


Figure 3. Generalized Parallel Processor Models (P = processor, M = memory)  
 From G. S. Almasi's and A. Gottlieb's "Highly Parallel Computing, Second Edition" [1994]. [6]

Since we determined that cluster systems and MPPs overlapped on certain viewpoints we can begin discussing them here. To first order, Figure 3 shows the two ways of arranging the processing elements (P) and memory modules (M) in a generalized parallel processor [6]. The model on the left is used for designs with a shared-memory model of computation in which every P has equal access to every M. The model on the right is used primarily for message-passing designs in which the processing elements

each have their own memory and communicate by exchanging messages, but can be used in shared-memory designs as well. Shared memory is a more powerful communication mechanism, but costs more to implement. The DAISy maps onto the message-passing or private memory model.

## 2.5 Control vs. Data Parallelism

The two types of parallelism discussed in parallel processing are control parallelism and data parallelism [6]. Control parallelism, which includes functional parallelism and pipelining, is achieved by applying different operations to different data elements simultaneously. In other words, different functions are performed concurrently.

The contrast to control parallelism is data parallelism. Data parallelism is the use of multiple functional units to apply the same operation simultaneously to elements of a data set. In other words, more or less the same operation is performed on many data elements by many processors simultaneously. A  $k$ -fold increase in the number of functional units leads to a  $k$ -fold increase in the throughput of the system, if there is no overhead associated with the increase in parallelism.

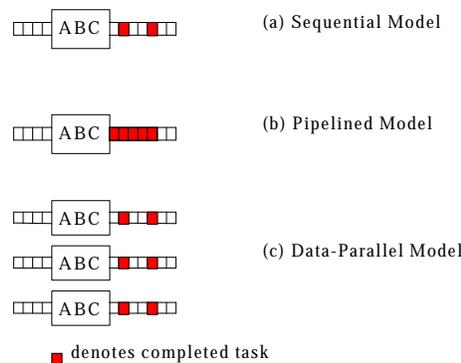


Figure 4. Parallelism  
From M. J. Quinn's "Parallel Computing, Theory and Practice" [1994] [5].

The following is from [5], Figure 4 demonstrates the difference between sequential execution, pipelining (control parallelism), and data parallelism. Also shown is a practical demonstration of speedup. Assume that it takes three units of time to perform a specific task. Also assume that it takes three steps to complete each task (A, B, & C), with each step taking one unit of time. As shown in Figure 4 it takes three units of time for a task to be performed with a sequential model. In the pipelining model, after the initial three units of time have been taken to complete the first task, each task thereafter is completed within the

next unit of time. In the data-parallel model the sequential model is replicated two more times. Throughput is increased by replicating models. Note that it still takes three units of time to complete a task, but now the tasks are scaleable with respect to the amount of processors.

The DAISy cluster can be a true demonstration of the data-parallel model. Acting as a *farm*, the added processing units can be used to increase the number of jobs that can be handled concurrently.

## 2.6 SPMD (Single Program, Multiple Data)

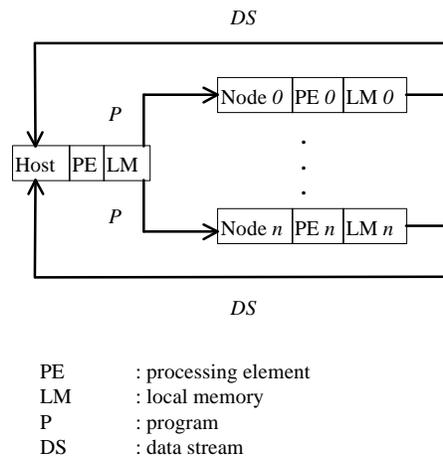


Figure 5. SPMD (Single Program, Multiple Data Stream) w/distributed memory

To implement a data-parallel application, the programmer assigns each processor responsibility for storing and manipulating its share of the data structure. This is called programming in the SPMD style [5]. It is a special case of MIMD in which all processors execute the same program, though at any one time the instruction can be different in each processor. Processors work on their own local data until they reach a point in the computation when they need to interact with other processors to: swap data items, communicate results, perform a combining operation on partial results, etc. If a processor initiates a communication with a partner that has not finished its own computation, then the initiator must wait for its partner to catch up. Once the processors have performed the necessary communications, they can resume work on local data. Figure 5 shows an SPMD model with distributed memory.

Programs written in this style are loosely synchronous. Between synchronization points (usually barrier synchronization's on a multiprocessor and communication functions on a multicomputer) processors asynchronously execute the same program but manipulate their own portion of the data.

This type of architecture can exploit “embarrassingly parallel” type of applications in which interprocess communication between nodes is kept to a minimum. An application of this type run on the DAISy cluster can prove to be very efficient.

## 2.7 Distributed Computing Models/IPC Sockets Method

In this section we will discuss two communication models and the interprocess communication method *sockets* that are and can be used in the use of DAISy as a distributed system.

The models discussed include the *client/server model* and the *message-passing model*. Both models are implemented on **DAISy** and used in the benchmark applications.

### 2.7.1 Client/Server Model

The *client/server model* [11] is the standard for distributed systems. A *server* is a process that is waiting to be contacted by a *client* process so that the server can do something for the client. Figure 6 shows a client/server example showing various methods of IPC. The dashed lines indicate some form of IPC.

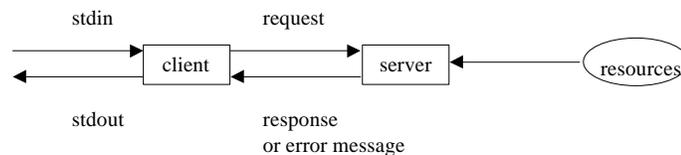


Figure 6. *Client/server* example.

The server process is started on some computer system. It is initialized, then sleeps, waiting for a client process to request a service.

The client process is started on either the same computer or a networked computer. The client process requests a service from the server process by sending the request across the network to the server via an IPC channel. The server then accepts and executes the request (if it can otherwise an error message is sent back) and replies to the IPC channel. The client then reads from the IPC channel and processes the information accordingly.

### 2.7.2 Message-Passing Model

The following discussion of the *message-passing* model will be based on the API (Application Program Interface) PVM [12] (Parallel Virtual Machine) System. Another set of API tools available is the MPI [8] (Message Passing Interface) System. Since there is no message-passing model standard there is an on going discussion about which will be the eventual successor. Of course, PVM users say that PVM will

prevail, but MPI users argue that MPI will win. In either case, there are some benchmarks that were run that required PVM and others that required MPI.

Both PVM [12] and MPI [8] provide a set of tools used to incorporate the SPMD taxonomy discussed earlier. They both are an integrated set of software tools and libraries that emulates a general-purpose, flexible, heterogeneous concurrent computing framework on interconnected computers of varied architectures. The principles included in PVM include the following:

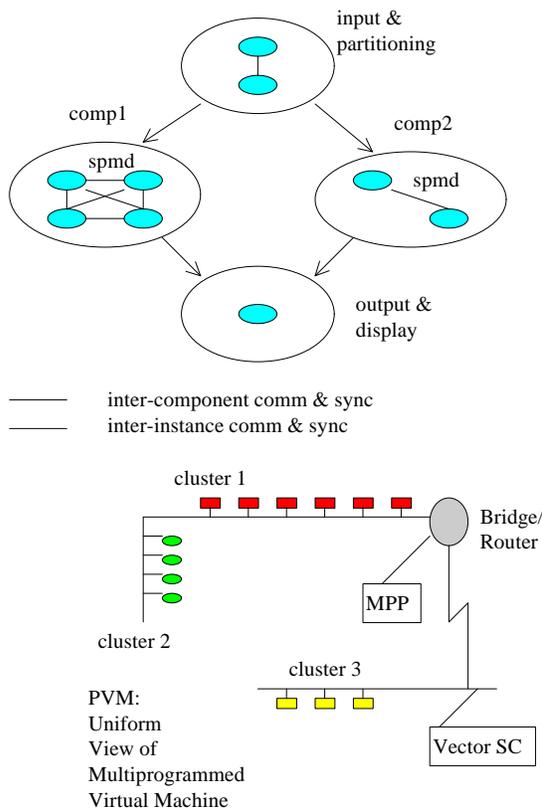
- User-configured host pool: The application's computational tasks execute on a set of machines that are selected by the user for a given run of the API program
- Translucent access to hardware: Application programs either may view the hardware environment as an attributeless collection of virtual processing elements or may choose to exploit the capabilities of specific machines in the host pool by positioning certain computational tasks on the most appropriate computers.
- Process-based computation: The unit of parallelism in PVM is a task (often but not always a UNIX process), an independent sequential thread of control that alternates between communication and computation. No process-to-process mapping is implied or enforced by PVM; in particular, multiple tasks may execute on a single processor.
- Explicit message-passing model: Collections of computational tasks, each performing a part of an application's workload using data-, functional-, or hybrid decomposition, cooperate by explicitly sending and receiving messages to one another. Message size is limited only by the amount of available memory.
- Heterogeneity support: The PVM system supports heterogeneity in terms of machines, networks, and applications. With regard to message passing, PVM permits messages containing more than one datatype to be exchanged between machines having different data representations.
- Multiprocessor support: PVM uses the native message-passing facilities on multiprocessors to take advantage of the underlying hardware. Vendors often supply their own optimized PVM for their systems, which can still communicate with the public PVM version.

The PVM system is composed of two parts. The first is the *daemon*, called *pvmd*, that resides on all the computers making up the virtual machine. When a user wishes to run a PVM application, he must first create a virtual machine by starting up PVM. The PVM application can then be started from a UNIX prompt on any of the hosts. Multiple users can configure overlapping virtual machines, and each user can execute several PVM applications simultaneously.

The second part of the system is a library of PVM interface routines. It contains a functionally complete repertoire of primitives that are needed for cooperation between tasks of an application. This library contains user-callable routines for message passing, spawning processes, coordinating tasks, and modifying the virtual machine.

The PVM computing model is based on the notion that an application consists of several tasks. With each task being responsible for a part of the application's computational workload. There are two

types of parallelism; functional parallelism and data parallelism. A functionally parallel application divides its tasks into functions, for example, input problem setup, solution, output, and display. For data parallelism all the tasks are the same, but each one only knows and solves a small part of the data. This is also referred to as the SPMD (single-program multiple-data) model of computing. PVM supports either or a mixture of these methods. Depending on their functions, tasks may execute in parallel and may need to synchronize or exchange data, although this is not always the case. Figure 7a shows the PVM computing model. An architectural view of the PVM system, highlighting the heterogeneity of the computing platforms supported by PVM, is shown in Figure 7b.



(a) PVM Computation Model

(b) PVM System Overview

Figure 7. PVM Computational Model and System Overview

A. Geist, A. Beguelin, J. Dongarra, R. Manchek, and V. Sunderam's, "PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Network Parallel Computing" [1994]. [12]

### 2.7.3 IPC Sockets Method

There are many variations of interprocess communication APIs. Since we are interested in distributed systems, discussed will only be a Internet IPC, the Berkeley Socket. The discussion is a brief overview of the user-level aspects of *sockets*.

To provide a common method for interprocess communication and to allow the use of network protocols, the BSD system provided a mechanism known as *sockets* [13].

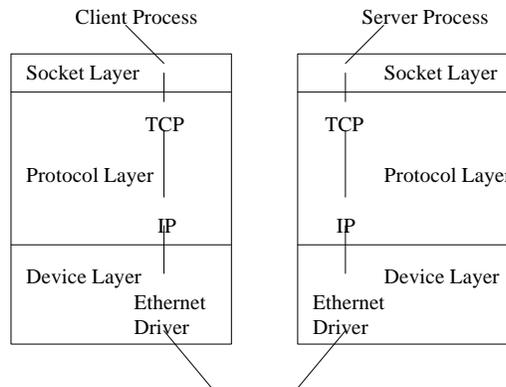


Figure 8. Sockets Model  
From M. J. Bach's "The Design of the UNIX operating System" [1986] [13].

Figure 8 shows the socket model consisting of three parts: the socket layer, the protocol layer, and the device layer. The socket layer provides the interface between the system calls and the lower layers, the protocol layer contains the protocol modules used for communication (TCP/IP in Figure 8), and the Device layer contains the device drivers that control the network devices. Processes communicate using the *client/server* model as mentioned earlier. A server process listens to a socket, one end point of a two-way communication path, and client processes communicate to the server process over another socket, the other end point of the communications path, which may or may not be on another machine.

*Sockets* support two communication domains: the UNIX system domain (in which process communicate on one machine), and the Internet domain (for processes communicating across a network).

## 2.8 System Performance

In all computer systems, system performance is limited by the slowest part of the path between CPU and I/O devices. In a distributed system, network performance plays a large role in the overall system performance. Figure 9 shows a typical collection of I/O devices which are of interest when determining system performance.

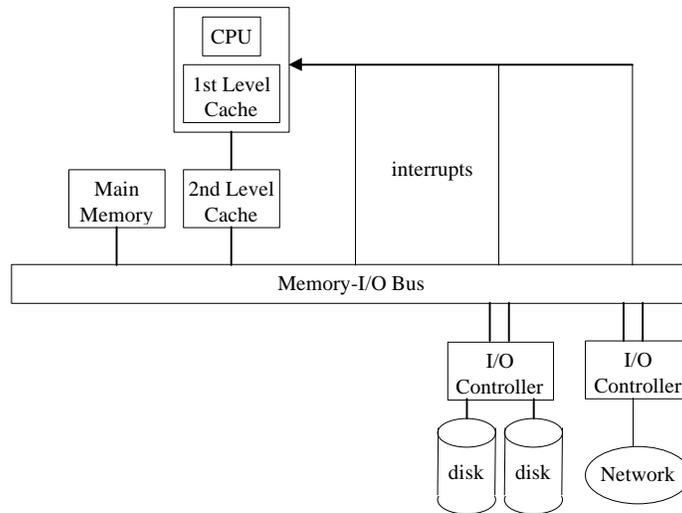


Figure 9. Typical collection of I/O devices on a computer.

In analyzing system performance both bandwidth and latency must be measured. To recall, *bandwidth* is the total amount of work done in a given time, and, *latency* is the time between the start and the completion of an event. Evaluating a distributed systems performance includes analyzing a single nodes subsystem performance, network performance, and overall system performance. Single node performance includes evaluation between CPU and: the theoretical CPU performance, 1st and 2nd level cache, 1st and 2nd level cache and main memory, and disk I/O. Included in evaluation of a distributed systems performance is network performance. This can be broken down into four categories: (1) low level latency and bandwidth tests which test various protocols (i.e. tcp, udp, and sockets), (2) API level latency and bandwidth tests (i.e. PVM and MPI), (3) Network File System (NFS) performance (needed for distributed systems), and (4) application level system performance tests (i.e. parallel applications).