School of Computing & Mathematical Sciences
Oxford Brookes University

Dissertation for Computing Science (BSc)
(Apr, 2001)

Graph theory with distributed programming

Supervisor : Dr. Fraser Mitchell
Student : Zoltan Papp / 98063617

Dissertation for Computing Science (BSc)
Graph theory with distributed programming

Who : Supervisor : Dr. Fraser Mitchell
Student : Zoltan Papp / 98063617

What : A program that uses basic graph theory and linear algebra to carry out analysis on a graph. The analysis is decomposed into subtasks and can run on several hosts in a heterogeneous unbalanced network environment.

Why : To study the principles of computer simulation with distributed programming.

How : Using C as the development language, PVM as the task distribution tool and UNIX and Windows hosts on the internet to perform calculation.

Contents :

Preface :

Graph theory is an increasingly useful mathematical area which is widely used to solve real world problems through abstraction. As computers have become powerful enough to carry out practically useful calculations, a number of areas emerged in computer science to model and solve problems that would take unacceptably long time with other methods. Today, these areas seem to cover every aspect of engineering and science from toy-design to protein-folding simulation. One such technique being used to model the real world is graph theory. Graph theory is strongly related with linear algebra as every graph can be represented as a matrix. And in turn, matrices in general are easy to implement on any major programming language.

Though, it is fairly tempting to implement three-dimensional heterogeneous matrices, I kept the overall design of data structures as structured and simple as possible and still ended up with a challenging complexity of data structures.

Hence, data and task decomposition is a key issue in computer simulation or in more general. Every area of life has its dominant factor and I think that computer simulation's primary factor is speed. We could argue if assembly or binary code would be the optimal language to write such applications but the implementation of these codes would be highly prone to have bugs because of their implementation complexity. C is just a little bit slower than assembly language and C is often argued to be a high level assembly language because of its simplicity and speed.

I choose efficiency, simplicity and clarity as the main criterias when choosing the development platform, language and distribution tool.

Chapter 1 Overview of Technologies

- FreeBSD became the choice of development platform,
- C the language
- PVM was advised by Dr. Fraser Mitchell, my supervisor to decompose the calculation and distribute it across a heterogeneous imbalanced network.

1.1 FreeBSD:

Though, Linux was the main platform to be considered because of the high number of systems installed, FreeBSD became the choice because of its stability and speed. FreeBSD is the free version of the BSD Unix operating system. Ken Thompson originally wrote Unix in 1969 at Bell Labs at Murray Hill, New Jersey in the United States. Bell Labs is a division of AT&T and they initially proposed to write an operating system that could share a single computer over many people and programs. The name of the project was Multics, and the reason Ken Thompson wrote his own operating system was, to be able to play his favorite game on a disposed PDP-7 computer. Ironically, Unix is a derivative of Multics. It is because Multics was a large project run by a range of highly educated individuals. Multics failed to be completed because it couldn't even handle three users at the same time to share the same resources. Ken Thompson spent one week each on the kernel, file system etc., and finished Unix in one month along with developing his game. I think this is fairly remarkable if we consider the billion-dollar business of the Unix market along with its applications.
Today, after 30 years of the implementation, Unix is still rock solid and undoubtedly the simplest and most secure OS around.
Netcraft, an independent information technology analysis portal publishes statistics on the web for free. One such survey examines the web servers with the longest uptime.
Surprisingly, 46 out of 50 on the top servers with longest uptime use FreeBSD as their operating system. If we want to develop secure and safe systems we have to consider these facts. Especially that this technology is free and has an abundant number of support on the internet and in the form of books.
Ironically, people pay fortunes for other operating systems. Notably, other well advertised and hyped Unix systems, but it is been a fact that most expensive equipment is not always the best.

Some of the properties of FreeBSD that are worth mentioning are:

- requires ISA, EISA, VESA, PCI based computer with an INTEL 80386SX to Pentium CPU (or compatible) with 4MB of RAM and 60MB of hard disk space.
  picoBSD, a small version of BSD requires 1.44MB hard disk space and 4MB RAM as well, but no hard drive required.
- preemptive multitasking with dynamic priority adjustment

- multi-user access of any peripheral and device attached to the system, for example printer, cdrom, tape drives can be shared across the network.
- complete TCP/IP networking including SLIP, PP, NFS, and NIS protocolls
- memory protection, demand paged virtual memory with VM/buffer cache design.
- binary compatibility with many other UNIX versions, including LINUX, SCO and Solaris.
- source code compatible with many other UNIX versions and hence, most applications require few if any changes to compile. The latter property is also due to the fact that UNIX was written in C.

1.2 The C language

The C language was derived from B by Dennis Ritchie at Bell Labs in 1972 for systems programming on the PDP-11 and was used to immediately re-implement Unix instead of assembly language.
C is not an object-oriented language nor has its multiple inheritance capability. Although, both properties can be easily replaced I think.
As source code grows, we inevitably confront the necessity to modularize our code through decomposition of variables and functions.
C allows compiling separate modules into object files and linking them into a single loadable binary object.
We can achieve this by using header files and the make utility.
I have recently been involved in programming Java and though Java and C have been written for different purposes, I found the use of header files very similar to the technique of inheritance. I structured the dependency relations into a tree form, and using a tree is, I think the only way to represent a hierarchy structure most efficiently.
Hence, global variables and those functions that have been used throughout the program, were placed in the bottom module.
The dependency structure didn't get complex enough to reach the make Unix tool's capabilities but it is been widely known that make have got to be improved or replaced because it have overgrown its own limitations over the years. Make will be introduced in more depth in Chapter 4.

Some of C's properties are :
- portable language, standardized by both ANSI and ISO.
- terse, low level and permissive
- has a macro preprocessor, cpp
- the dominant language in systems and microcomputer applications programming
- it is simple, efficient and flexible
- fast and has a series of mature, well-developed compilers compilers for CISC platforms are in wider use and are also more advanced as there are more CISC processors in use than RISC and it is easier to write a compiler for CISC architecture because of the wide choice of processor instructions.
- easy to adapt to new environments

- standardized by ANSI and ISO
- supports dynamic module loading during execution time
- a large number of libraries support C

Hence, C looked very appropriate to program distributed applications. One such library that supports distributed application is PVM.

1.3 PVM

PVM stands for Parallel Virtual Machine and is an industry standard in parallel network computing. PVM tasks can run simulatenously across a variety of host types linked by the internet.
Every PVM task has an identifier called TID, which is a 32-bit integer.

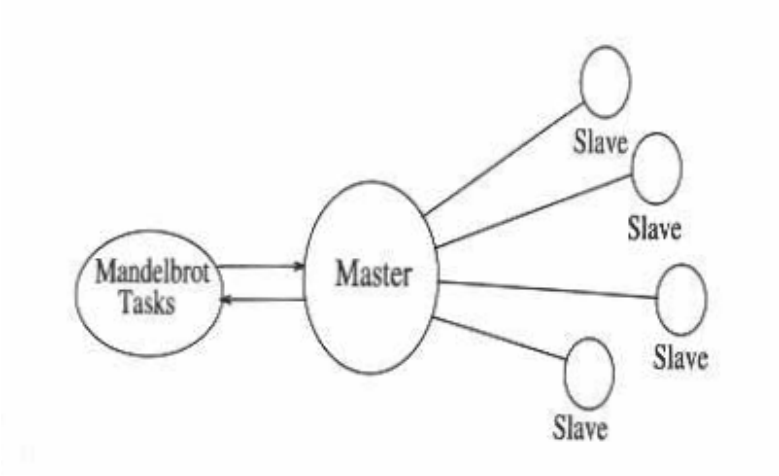


Figure 1.1 PVM computational model

TID is similar to the purpose of PID's on the UNIX system. TID is a global identifier in the virtual computer and is chosen by the master pvm process to be unique. It encodes several information about the task itself but neither the programmer nor any external application should rely on the information encoded in TID.
PVM uses message passing to communicate among simultaneously running tasks in a heterogeneous environment. Message passing, in general, also became the de facto standard in inter task and process communication in modular kernels as well.

The PVM project began in 1989 at Oak Ridge National Laboratory in the United States. The first viable version of PVM, 1.0 was written by Vaidy Sunderman and AL Geist. Both of them are still involved in the development. The University of Tennessee wrote version 2.
In the forthcoming years it became increasingly popular and many science and engineering applications use the current version of PVM, version 3. Its popularity can partly be contributed to its free cost and that is comes with source code.

With necessity of completeness, three descriptions of PVM exist to my best:

1: PVM is a software system that permits a heterogeneous collection of UNIX computers networked together to be viewed by a user's program as a single parallel computer.

2: A software system designed to allow a network of heterogeneous machines to be used as a single distributed parallel processor

3: The intermediate language used by the Gambit for Scheme.

1.3.1 Other distributed programming tools

PVM is not the only networked parallel computing tool. Other notable efforts include P4, Express, MPI and Linda.

- P4 was developed at Argonne National Laboratory at the University of Chicago but its main problem is dynamic process creation.
  Hence, process forking is either pre-determined or very intricate to perform.
- Express was developed by members of the California Institute of Technology and commercialized by ParaSoft corporation. Express uses a development cycle to transfer a sequential C program to a parallel C program by using analytical tools. It determines how a sequential code can be transferred. It provides a smooth transfer from sequential programming to distributed programming for those people whom are new to this area. And consequently, it includes almost everyone including myself. Though, the question remain open if it is worth writing a sequential code first and than start changing it to parallel rather than understaning parallel programming first and than writing a parallel program without modifying it.
  Hence, I think, Express can be used as a viable tool to port sequential C programs to parallel ones but not necessarily for writing new applications. Also, existing applications running sequentially will not be necessarily ported to their parallel counterparts because modifying existing, and well working complicated systems might require more work than writing new applications. But PVM_POV is an exception, what I will introduce later in Chapter 5.
- MPI (Message Passing Interface) is an interface and hence cannot be used on its own to implement a parallel programming environment. It is intended to standardize message passing and thereby, increase portability of those applications and tools that use this technique.
  The use of MPI varies from MPPs (Massively Parallel Processors) to modular kernels. Hence, PVM can also use MPI to exploit the benefits of standard technology and thereby, increasing its portability.

- Linda was written at Yale university and uses an abstract computational space called “tuple-space” . Processes are co-operating via communication in the tuple space. Linda has been proposed as an alternative paradigm to the two traditional methods of parallel processing :

  1: based on shared memory
  2: based on message passing

  From the application point of view, Linda is a set of programming language extensions for facilitating parallel programming.
  It provides a shared memory abstraction for process communication without requiring the underlying hardware to physically share memory. Depending on the environment (shared-memory multiprocessors, message passing parallel computers, networks of workstations, etc.) the tuple space mechanism is implemented using different techniques and with varying degrees of efficiency.

  This creates a bottleneck because it makes load balancing more difficult as the programmer has to deal with a series of special cases whether certain techniques are being used or not. Though a newer system is now available which proposes proactive approach to concurrent computing :
  + computational resources (viewed as active agents)
  + seize computational tasks from a well-known location based on availability and suitability.

  This scheme may be implemented on multiple platforms.

1.3.2 Conclusion and the possible future

Hence, we can see that networked parallel computing is still a rapidly evolving area where many researchers are still not sure what methodology to use to have a delicate balance of usability and performance.
It is notable, that only two of these technologies have a newsgroup on the internet. These are MPI and PVM, hence, clearly proving which technologies have succeeded outside the research laboratories.

The reason why distributed programming and PVM in particular became very popular in the past several years is because computer science witnessed and ever increasing acceptance and adaptation of parallel processing. The acceptance has been facilitated by two major developments:
1: massively parallel processor (MPP)
2: distributed programming

MPPs are the most powerful computers in the world but also the most expensive. What distributed computing thrives to achieve is computational performance comparable to MPPs but at significantly lower costs.
SETI (Search for Extraterrestrial Intelligence) and Intel’s explained example just made for doing that.

Even if not as fast, both SETI and Intel achieved something that did not cost money and still gained a considerable factor of computational power increase rather than spending millions of dollars.
That is a huge difference to MPPs. MPPs combine a few hundred to a few thousand CPUs in a single large cabinet connected to hundreds of gigabytes of memory.

As mentioned earlier, the most important benefit of distributed programming is cost. Common technology to both distributed computing and MPPs is the Message Passing Interface. In all parallel processing, data must be exchanged between corresponding tasks.
Several paradigms have been tried including shared memory, paralellizing compilers and message passing. In contrast to PVM, in a MPP, every processor is exactly like every other in capability, resources, software and communication speed.

However, large MPP systems have drawbacks. Among these are:

- Custom hardware components are quickly superseded by commodity components.
- Volume vendors are not the best organizations to create niche products.
- Large system scalability requires specialized knowledge and research.
- Large-scale systems must grow in size and capability over their lifetime.
- Applications that require high levels of compute performance will continue to grow in size, variety, and complexity.

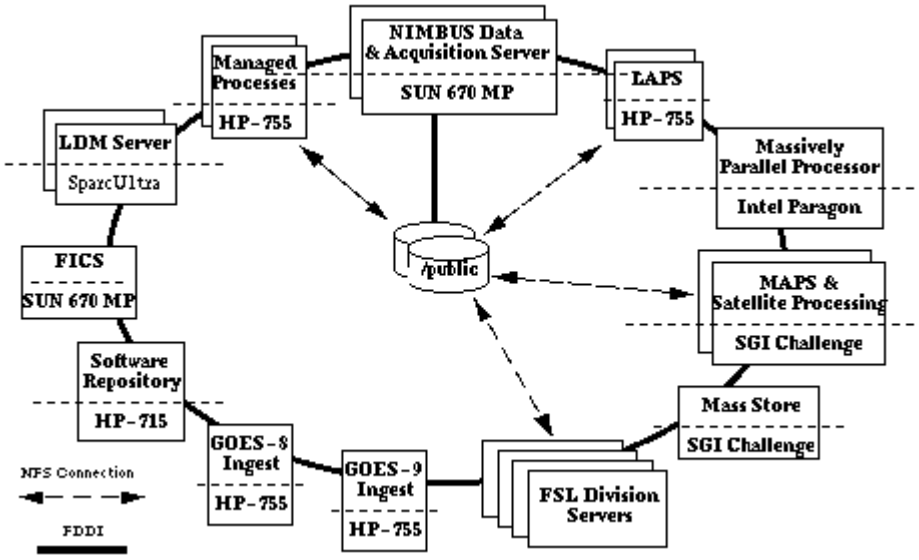


Figure 1.2 Example of a heterogeneous distributed environment

But a heterogeneous network is a completely different issue.
The computers available on the network may be made by different vendors or have different platforms with different compilers.

Indeed, when a programmer wishes to exploit a collection of networked computers, he or she may have to contend with several different types of heterogeneity such as:

- architecture (Intel 386, Sun sparc, Alpha AXP)
- data format (protocol type, variable type)
- computational speed (ALU, FPU, CISC, RISC)
- machine load (multi-user-multitasking environment)
- network load (publicly shared internet)

Despite the fact that the programmer has to take these properties into consideration when proposing to optimally exploit the available resources, distributed programming can be turned into a profitable technology in the following ways:

- Performance can be optimized by assigning each individual task the most appropriate architecture.
- One can exploit the heterogeneous nature of computation.
  Heterogeneous network is not just a local area network connecting workstations together. For example, it provides access to different databases or to a special processor for those parts of an application that can run only on a certain platform.
  Also, the parallel virtual computer can be internet wide like the SETI project. It would also be possible that some company is a fan of fancy equipment and because they have enough money they buy Alpha AXP platforms or SGI machines for word processing. Also, let's imagine that we work for this company because let's assume that we are well paid. We are told to set up a parallel virtual machine to solve some large computational problem.
  We would than be able to exploit heterogeneity because we would pre-determine the process of task distribution based on task type and architecture. Hence, grouping and dedicating certain tasks to certain machines will help maximizing the use of resources.
- The virtual computer resources can grow in stages and take advantage of the latest computational and network technologies.
- Program development can be enhanced by using a familiar environment.
  Programmers can use editors, compilers and debuggers that are available on individual machines.
  PVM has libraries for C, C++ and Fortran languages and can run on basically any UNIX platform and MS-Windows environment as well. Hence, covering basically any kind of computer platform from Intel Paragon to Intel 386.
- The individual computers and workstations are usually stable and substantial expertise in their use is readily available.
  User level or program-level fault tolerance can be implemented with little effort either in the application or in the underlying operating system.
  Security software can be well integrated into an existing PVM system because security tools usually do not interfere with existing applications. Such tools include secure shells; secure IP network, firewall, and other increasingly emerging middle tier tools.

Though I haven't yet heard about the following possible technology but I think that wireless networks can also be used for computation. Currently, satellites, laser beams and microwave transmitters also provide a partly wireless internet.

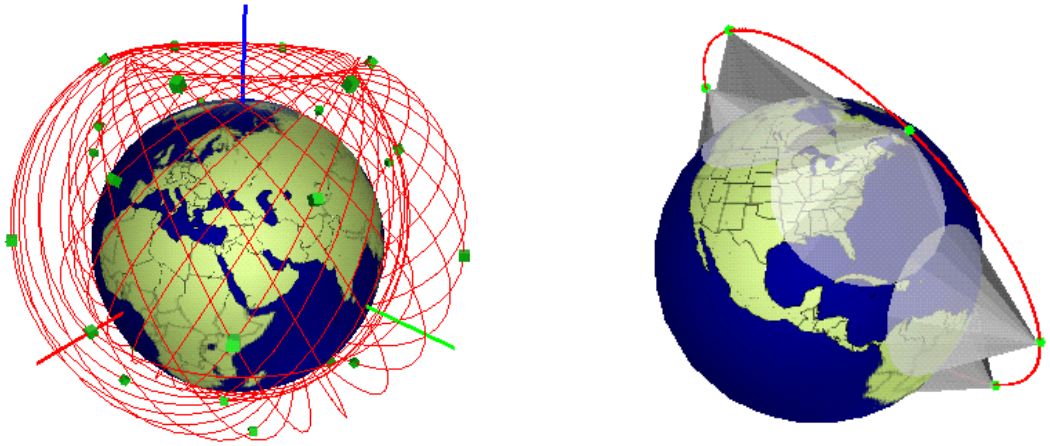


Figure 1.3 Wireless satellite network with high bandwith

But in common sense, a series of hand-held devices such as mobile phones and PDAs can be considered in the future as being part of a co-operating tasks on the wireless network. The introduction of Ipv6 made it possible to every single device connected to the internet to have a unique ip address. This fact changes the picture of the net radically. Even though a mobile phone cannot be an equal computational resource of a workstation but still in network terms, these devices hold computational power because of their following usages:

- voice recognition
- mp3/mpeg de/en coding
- supporting simple mini operating systems such as
    - o Windows CE, Psion EPOC, Linux, and others.
    - o mobile computing is an emerging trend
    - o wireless bandwidth is increasing

A survey have concluded the following trends in high performance computing and the success of massively parallel systems in commercial environments is not bound to any special architecture. Maturity of systems and availability of key application software in a standard Unix system environment are much more important than details of the system architecture. The use of standard workstation technology for single nodes is one key factor. This eases the task of building reliable systems with portable application software.

From the present eight releases of the we see the following trends:

- The number of industrial customers in the has risen steadily since June 1995.
- The most successful companies (IBM and SGI) are selling disproportionately well in the industrial market.
- The average system size at industrial sites is increasing strongly.
- Database applications is the most important and most successful new application area for supercomputers.
- Distributed-memory systems are being installed at industrial sites in reasonable numbers and have outnumbered shared memory vector systems in the meantime.
- Only in the automotive industry vector processing is still dominant.
- IBM is leading in the industrial market place ahead of SGI/Cray.
- The United States is the world leader in the industrial usage of HPC systems.

Heterogeneous computing hence can be very efficient by studying the environment in which we aim to use it. The exploitations of different platforms and architectures can be primarily contributed to their connection. The network and its evolution, hence, played an integral part in the improvement of the performance of distributed computing.
Hence, I think it is worth outlining the evolution of network protocols and the appearance of the internet in particular.

1.4 IP and the network

In 1961, at the height of the Cold War, an engineer named Paul Baran at the Rand institute, sold the US Department of Defense on the idea of failure-resistant communications method called packet switching. But because of roadblocks at AT&T and the Pentagon, it wasn't until the 1970s that the technology was finally adopted as the foundation architecture of the Arpanet – the precussor to the Internet.
In April, Paul Baran will receive the Franklin Institute's 2001 Bower Award and Prize for Achievement in Science, his latest in a string of prestigious honors from professional organizations including the Institute of Electrical and Electronic Engineers, the Association for Computing Machinery and NEC corporation. Over a lifetime of quietly sustained achievement as investor and entrepeneur, Baran cofounded the Institute for the Future and created a series of successful companies based on technologies he developed. Baran's inventions went mainstream: His discrete multitone technology is the heart of DSL (Digital Subsriber Line) and his developments in spread spectrum transmission are essential to the ongoing wireless explosion. Yet, Paul Baran is little known outside his field.

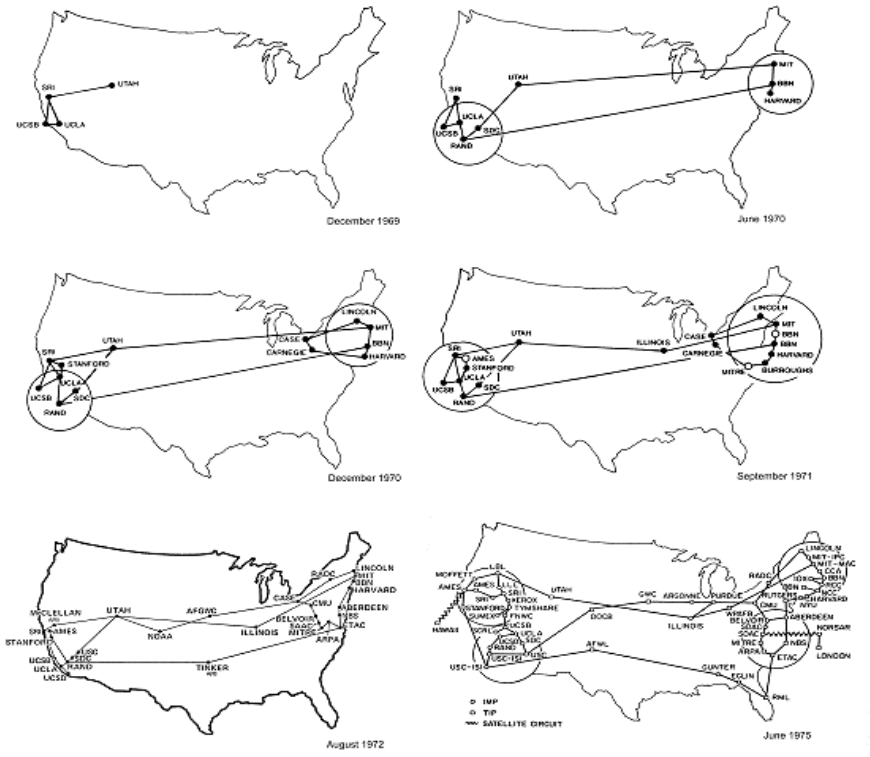


Figure 1.3 Evolution of the Internet between 1969 nd 1975 (USA)

A few facts about the Internet are, in April 2001 :

- Total domains registered worldwide : 35396877
- Total .COM registered : 22373097

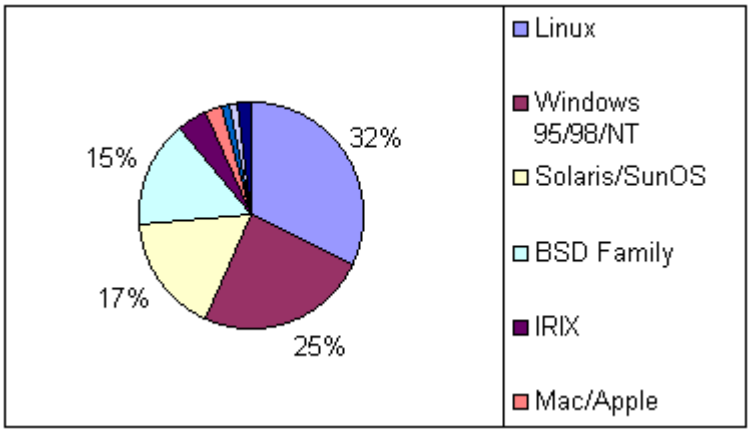


Figure 1.4 Percentage of host platforms on the Internet

Later on, further technologies were developed, all of them based on Baran's ideas

- Ethernet, the name given to the popular local area network by Xerox PARC. The Ethernet is a 10MB/s broadcast bus technology with distributed access control.

- FDDI, Fiber Optic Data Interface is a 100Mbit/s token –ring that was optical fiber for transmission between stations and has dual counter-rotating rings to provide redundant data paths for reliability.

- HiPPI, High Performance Parallel Interface is a copper-based data communications standard capable of transferring data at 800MB/s over 32 parallel lines or 1.6 Gbit/s over 64 lines. Most commercially available high-performance computers offer HiPPI interface. It is a point-to-point channel that does not support multidrop configurations.
- SONET, Synchronous Optical Network is a series of optical signals that are multiples of basic signal rate of 51.48 Mbit/sec called OC-1. OC-3 has 155.52 MB /sec transfer rate. OC-192 has 9.952 Gbit/sec and is currently the fastest fiber optic technology used commercially, made by Siemens electronics.
- ATM, Asynchronous Transfer Mode is the technique for transport, multiplexing and switching that provides a high degree of flexibility required by B-ISDN. ATM is a connection oriented protocol employing fixed size packet with a P5-byte header and 48 bytes of information.

So far, we haven't considered hardware and it is actually less relevant because it is highly probable that a PVM program will run as the second most important application on a certain host. For example, Intel uses distributed simulation on its inside office PCs to carry out integrated circuit optimization. The program uses the office PCs idle time and hence users should not be able to notice that an extra program is running on their PC. In this case, the key point is that we have an existing environment and we would like to extract the remaining resources of the system. Hence, optimal running of a distributed program on this system is not a primary concern, though should run as good as possible.
We should also note that Intel did not invest any serious money, only restructured its network software, self organizes itself what reduced in spending. The other case is when a dedicated hardware is available to perform the task. This requires money and hence the purpose of the system must be serious enough to invest in. Such examples can include, nuclear weapon testing, protein folding, climate modeling, financial analysis and so on. These applications came from science areas.
A series of science areas have reached a level of complexity and research cost that computer simulation became the essential tool for the experiment test bed of research areas. In particular, biology and physics fosters the demand of strong computational power. Currently, Sandia National Laboratory, Compaq, Celera Genomics and the Department of Energy of the United States have signed a contract to start develop cheaper and faster solutions for the high demand of computational power. This type of research is of national and public interest and usually sponsored by governments and/or academia just like in the case of Sandia National Laboratory. Compaq will use a computer called ASCI-RED, which probably cost a lot of money. It will be designed in every aspect to maximize the execution of a specific type of application. It uses Alpha AXP RISC processors.

As it is known, Intel manufactures RISC processors as well.
The argument of Alpha or Intel is based on the following properties :

Alpha:

- Floating Point Unit is faster than Intel CISC platforms and simulation usually heavily uses floating-point arithmetic.
- Simulations require a narrow set of CPU instructions and RISC is exactly what is about.
- Unix is believed to run faster on Alpha platforms

Intel:

- Cheaper than Alpha because it is a commercial product used by millions of people
- More support, compilers, applications and tools optimized for the Intel architecture simply because of Pentium being a standard in processor terms.
- Intel is just as fast as Alpha for the same amount of money and work.

In the forthcoming years, I think, Intel type networked wide distributed simulations will be more common, as networked bandwidth is increasing and local area networks will be fast enough if not already. It is because normal people and organizations cannot afford expensive and special equipment and they use commonly available widely used technologies like Intel.

Chapter 2 Theoretical and philosophical issues of simulation

We, homo sapiens have always been depending on our environment and probably will in the future as well. Planet Earth protects us from the extraordinary conditions of outer space. The extent to which we depend on our environment can be changed, and the ability to change this level plays a key role in our survival. It is notable, that we are in turn, the environments of ourselves because of our human mates. Change of dependency from the environment if necessary, can be done in two ways:
1: changing location to places were it is better to live (moderate climate, economy, etc)
2: staying where we are and changing the environment, including ourselves through

a: the modification of DNA naturally (immigration)
b: the modification of DNA artificially (genetical engineering)

It is quite clear that from our early civilization, we developed our civilization because we observed, studied, understood and saved the information about our environment by using logical abstraction.
I would like to point out that we also developed beliefs about our environment in the form of superstition, religion, and other primitive forms of knowledge. These beliefs were/are unsure and difficult to apply in practice, creating self-destruction in the form of religious wars, fear and stupidity.

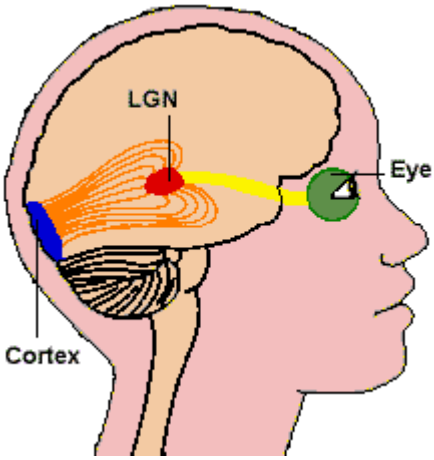


Figure 2.1 Overall structure of visual signal processing

Hence, one of the key issues to depend less on our environment is to replace belief with logic and human understanding.
If we think about it more, belief is a kind of passive unstructured knowledge, a heuristical rule, that used in Artificial intelligence.
Intelligence, something we all claim to have.
People in western technological civilizations are 24% more intelligent than traditional religious eastern and other nations. It is partly because of the plasticity of mind, plasticity is the ability of the brain to develop or modify existing connections and structures in the brain. We all know that brain is a connection oriented biological computational system.
Plasticity allows higher functionality for the brain and is fostered by the high demand of functionality to be able to cope with technology from cars to mobile phones or anything that requires this capability, diverse structure of environment for example.
It is worth to note that every form of life has the capability of acting intelligently, including viruses at the bottom of the pyramid of the ecosystem. It is not too important if an intelligent process happens consciously or not because what we call consciousness is only a tiny proportion of processes going on in our brain. Even worse, we are not always conscious during our daily routines as well. Have you ever found yourself wondering about something while you were doing something else. Or have you ever applied your reflexes with consciousness ?
So life is intelligence itself because intelligence is the capability of manipulating matter. This applies also to the way we think. If someone hits you very hard, the first thing you would do is to hit back.
But isn't that the same as hitting the ball to the wall and the ball bounces back ? Simple physics rule, effect and anti-effect law.
It is obvious that there is no redirection of forces at all, all you apply is reflection and hence hitting back. But would the other person be surprised if you were going and buy a newspaper and read daily news after he hit you ? Performing the first and second reaction requires radically different ways of thinking.
What is the difference and what does it require?
Surprisingly, the brain implements thinking and working as two distinct processes.

We feel that thinking is a creative process in what at least 2 different ideas merge and create a more sophisticated third one. While working is a more or less fixed task, just like a pre-programmed washing machine.
What computer simulation has more to do with, is thinking.
Over 90% of information we sense is visual, if we mean someone who is not blind or has any damage on its brain what would change this otherwise. The brain, hence, has a considerable size devoted to visual processing which is also very sophisticated. Pattern matching for example. On average, everybody is capable of recognizing a close relative or friend by performing 100 operations in parallel.

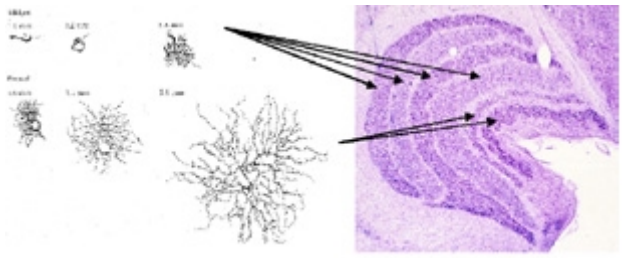


Figure 2.2 Parallel pathways in the visual cortex

The eye first sends the image as a signal broadcasted to many areas in parallel as shown on Figure 2.1. These signals spread like a wave on the surface of a lake, called ripple. These ripples are filtered out and some might intersect or converge with one another in a way that it works as a process of pattern recognition. So, this computational method is a kind of harmonical computational process where natural processes and physics is heavily used along with abstraction of raw information. For example, a person can be handled as an object. It is necessary for the brain to use abstraction to be able to model and relate objects and places to one another just like a computer simulation or a game uses objects.
It doesn't need to be explained, all we have to do is analyze the way we think. You can think of anything and try to understand how you represent it in your idea. Hence, we partly think by modeling and simulating things. Engineers, architectures, programmers are especially using this technique when they plan their system.

Chapter 3 Overview of the program

The functionality of the program comprise of two parts:

1: Input/output of information
2: Processing of information

Input/output mainly concerns with opening a file, saving a file and creating a new file. Processing the information is mainly about analyzing the graph through task distribution.

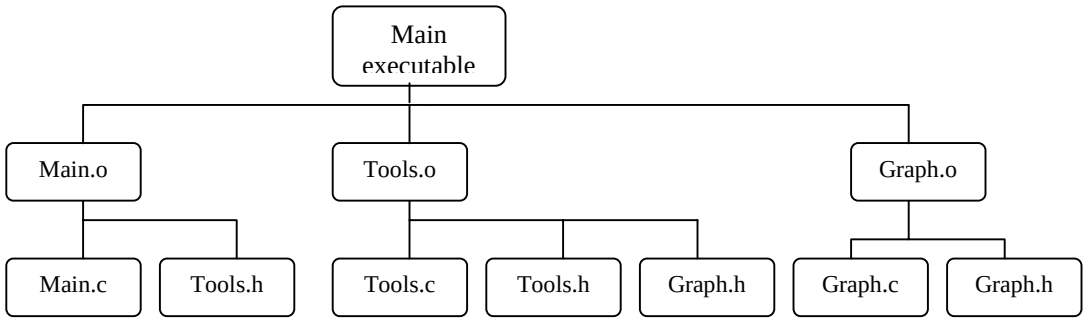


Figure 3.1 Module dependencies of the master pvm task

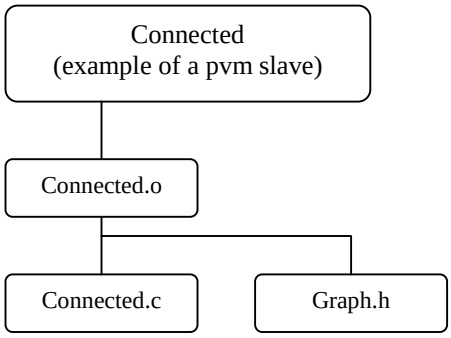


Figure 3.2 Module dependencies of slave pvm tasks

The source code of the program comprise of several parts and these functions and variables are grouped together in modules according to their function shown.
The relation and dependency between modules is achieved via header files as shown on Figure 3.1 and 3.2.
These modules than will be linked together and coompiled into two set of executables.
1: *main* (Figure 3.1), the main executable target of make. Main holds the menu, tools, and graph modules. Hence, main controlls the process of the program, including the master pvm task. a single executable binary that has simple character oriented menu interface. It is partly because portability was a primary concern and it had to conform to the ANSI/ISO

C standard. Hence, Motif and Microsoft MFC libraries were excluded from the development cycle, though it should not take long to implement a graphical user interface to the system. Under X windows, it would be possible to run the interface on a different host from other sub functions running on a dedicated host by using inter process communication.

2: sub-tasks (Figure 3.2) that perform analysis on the graph independently and return the obtained information to *`main`*. These pvm slave tasks are as follows :

- **`regular`**
- **`connected`**
- **`eulerian`**
- **`degree_sequence`**
- **`chromatic`**

The executable files can be grouped in two. The first one is the main executable on its own which has to run on the current host we start executing it from. The other group is the slave tasks that can run on any host in the virtual machine, depending on the arguments of the pvm_spawn system call. We can spefify for tasks to run on a dedicated virtual machine host or let the system load manager choose the most optimal host for network transmission and computational speed.
I let the load manager decide the host.

Hence, there is a possibility that all the executables will run on the same host or will run on unique hosts, depending on the number of available hosts.

Data is stored as a file and can be temorarily loaded or created in memory. Data is decomposed into two parts:

1: Adjacency matrix that represents the graph
2: Information about the graph that can be obtained by selecting the graph properties.

Both information group have a dedicated global variable, and indeed these are the only global variables that can be accessed globally in any function.

3.1 Development cycle

The first thing I had to decide was how to represent graphs.
Many textbooks recommended edge-adjacency matrices because standard adjacency matrices can create sparse matrices.
It is because a sparse matrix represents the minimal information required and hence maximizing space. On the other hand adjacency matrices are easier to perform calculations on, because they have a fixed NxN geometry where n is a positive integer. Hence, the number of calculations required to traverse an adjacency matrix is n^2 which is always going to be more than or equal to that required for on edge-adjacency matrix.
Irrespective of storage space, I choose adjacency matrices because it was easier to follow the already complicated graph algorithms. I think, this little sacrifice in storage space paid off in development time and source code clarity. And in turn, because of program clarity, it made less possible that I will make a mistake, and it increased robustness.
Data structures:
matrix_t : to store adjacency matrix
It is declared as a list of linked lists. **Row_t** is a linked list and **matrix_t** is a list of **row_t** type lists.

```
typedef struct row {
     int weight;
  struct row *x;
} row_t;

typedef struct matrix {
  struct matrix *next_row;
  struct row *y;
} matrix_t;
```

Figure 3.1 Implementation of **matrix_t**

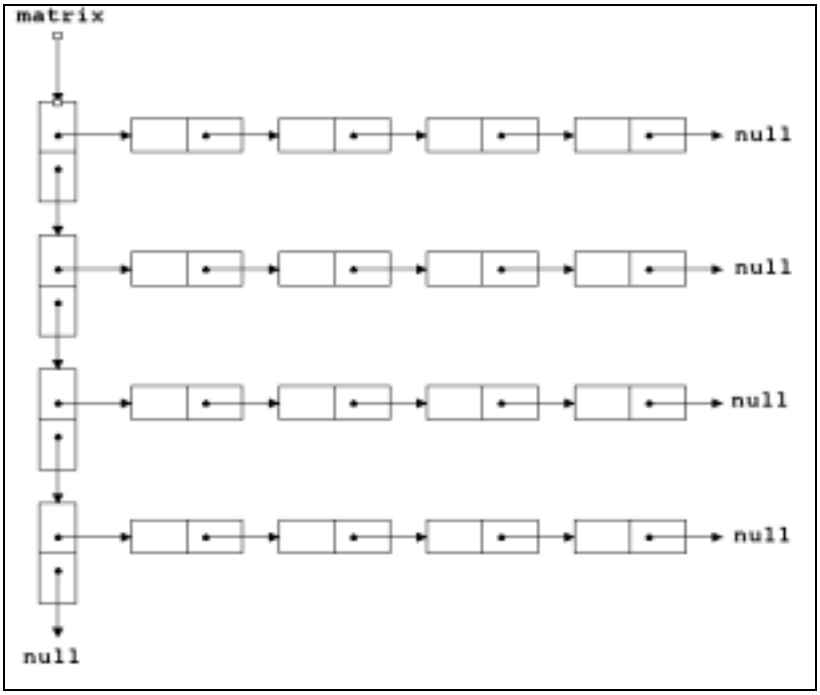


Figure 3.2 Structure diagram of a 4x4 matrix

info_t : to store information about loaded file and analyzed graph

```
typedef struct info {
       char  filename[NAME_SIZE],
             matrix_name[NAME_SIZE];
        int  size,
             chromatic_index,
             chromatic_number;
  boolean_t  connected,
             tree,
             complete_graph,
             eulerian,
             hamiltonian,
  bool_int_t regular;
  row_t      *degree_sequence;
}info_t;
```

Figure 3.2 Implementation of **info_t**

boolean_t : boolean type

```
typedef enum {
  FALSE,
  TRUE
} boolean_t;
```

Figure 3.3 Implementation of **boolean_t**

Basic functions

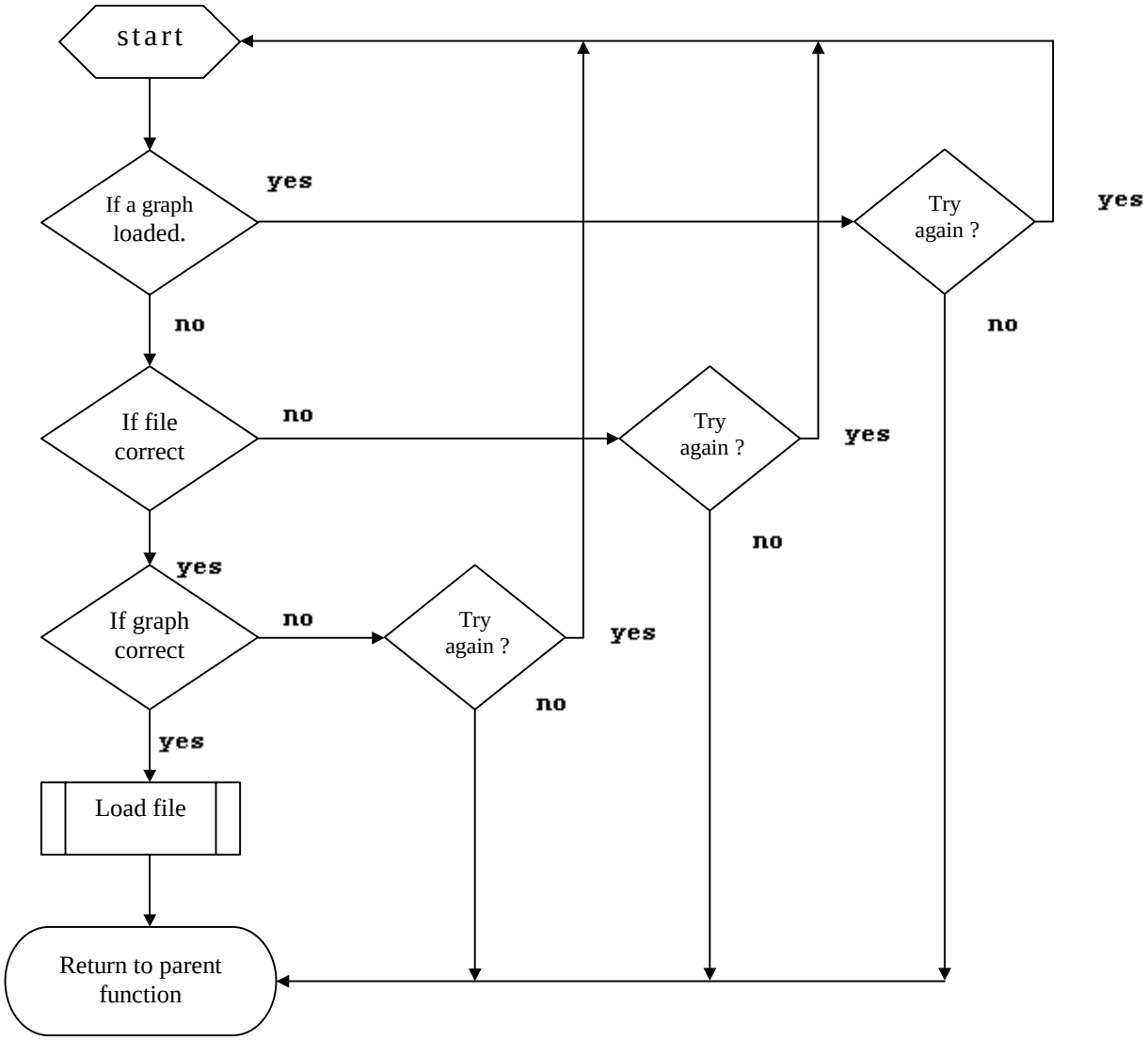
The first function being implemented was *load().*
Load() checks if a file is already been loaded or not. If the file is loadable and the global *matrix* variable is set to null, *load()* will examine the following properties of the graph and the matrix itself :

- file exists and loadable
- file is not empty
- first string of file is graph_name
- the adjacency matrix has NxN geometry, where N is a non-negative integer.

If any of these conditions doesn't hold the program returns on error message and asks if the user wants to try loading again.
If the file found to be correct, the program proceeds be reading file to memory. *Load()* also saves the following information about the graph to the **info** variable:

- name of the graph
- size of the graph

The flow diagram of *load()* is as follows :



The loading part was the first slightly complicated part to implement. It basically has two loops, one for the rows, and one for the entries. Every line in the file corresponds to one outer loop and every number in the row corresponds to one inner loop. There are two pointers keep track of the current entry and node. One pointer, *current_row* holds the pointer to the current row and the other pointer *current_node* holds the pointer to the current entry.
So the adjacency matrix is stored as a list of linked lists, where every linked list corresponds to a row in the matrix as figure 3.2 shows.
To make sure that the matrix's entries are easily accessible I declared an arbitrary function which can be used to absolutely access any entry of the matrix. The function is called *access_xy()* and uses two loops to hop from the root node to the appropriate rows by hopping along the rows and than the columns to set the *current_node* pointer to the appropriate value. Alternatively, I could have used pointer offsetting but it would have required special treatment for matrices of size 1xN, Nx1 where N is a non-negative integer.

Once the graph is loaded to the memory, and I had a function to access any entry, it came obvoius to see the loaded matrix in memory. Hence, *display()* function is called to display the graph and its name.
Just like all the functions , display operates on the global variables, *matrix* and *info*. For example, *display()* uses the *matrix* variable to obtain the values of the matrix entries saving cost and time. It was very convenient to use NULL as a guard in both loops as the structure of the matrix is stored as a list of linked lists and both lists are terminated with a NULL.

Menu system and decomposing the program

At this stage, even if the complexity of the code was not high.
I had two distinct functionality of the program, loading file and displaing it. I thought it is better to begin structuring it as early as possible to make it easy to follow the overall structure. I believe that this investigation paid off, even though I had a hard time with PVM, the code itself remained clear and logically structured throughout the development process.

Also, I had to leave the menu, character oriented and I won't use some other functionality like clearing the screen properly. This bulk made my main interface somewhat unprofessional, and hence, I decided to put an extra comfort for the user and implemented a prioritized menu what I explain soon. The main function should hold only the high level operations of the program and try to hide the underlying complexity and size.

With this goal in mind, I hide all the unnecessary functions in the sub modules by using the header files. The main function only comprise of the menu declaration and argument processing making it the smallest module among all.

It also holds the most flexible and highest level code. I mean this because in other modules, functions and procedures deal with sequential processing, graph analysis for example. In other words, sub modules deal with predetermined tasks while the main module deals with process control. It is a typical issue of master-slave programming.

Implementation of main function is as follows:
Menu text is stored in an array of string pointers.

```
const     char *menu_text[FUN_MAX]={
                        "Open file",
                        "Save",
                        "New graph",
                        "Display adjacency matrix",
                        "Properties of graph",
                        "PVM options",
                        "Exit"};
```

Figure 3.3 Declaration of **menu_text**

The corresponding high-level functions to the menu options are stored in array of function pointers.

```
typedef struct {
  void   (*function)();        //pointer to function
   int    priority,            //priority of functions
          cases;               //number of use in 17 cases
} function_t;

function_t functions[FUN_MAX] //variable declaration
```

Figure 3. 4 Implementation of **function_t** and declaration of **functions** variable

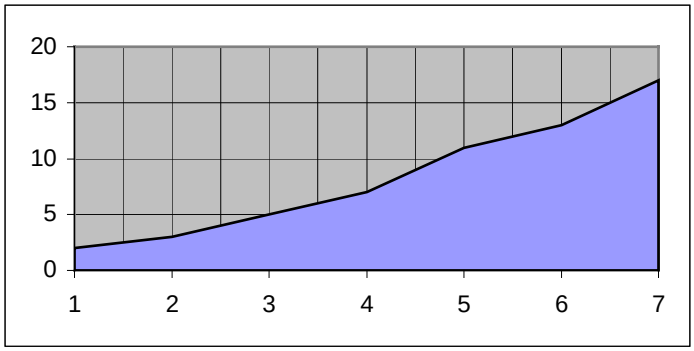
There is also on array of integers to store the limits of the priorities.

```
const int limits[FUN_MAX]={0,11,7,5,3,2,1};
```

Figure 3.5 Declaration of **limits**

Hence, if a priority of a function exceeds the limit of its current menu position, it will be shifted up one level in the menu. I found it very handy because the more I use a certain function, the higher it will appear in the menu option. And everyone's eyes starts looking at menu options from top to bottom.

The priority list starts at o and uses the next consecutive prime number as the next limit. I have long been thinking about what numbers to choose as limits and I decided to use prime numbers because prime numbers occur naturally on the x-axis. Hence, they have the closest relationship to real nature and human behavior.



Hence, every menu option has a corresponding variable to maintain the number of cases it is been called. The number of cases for each function is reset after a period of the longest prime number in the array of limits. It had to be done that way because we have to measure the frequency of use and not the total where

f = freqency
n = number of cases
T = period

$$f = \frac{n}{T}$$

This is very similar to the calculation of frequency in terms of number of cases and period time in physics.
At this stage, the program had the final frame on which I could build as many further functions as I wanted to. It is because extending the menu requires 5 changes in case we want to add one more function. These steps are as follows :
Step 1: Insert one more prime number to limits variable.
Step 2: Insert new string entry to menu_text variable.
Step 3: Set function name to appropriate function array cell in functions variable.
Step 4: Increase **FUN_MAX** constant with one.
Step 5: Change **PERIOD** constant to largest number in limits array.

Also, tools.c and graph.c were created where tools.c holds all the high level functions and support functions. Graph.c holds all the graph algorithm related functions and data type declarations. Hence, every further development was possible with this structure. Tools.c inherits necessary low level functions and types from graph and main.c inherits necessary High level functions and types from tools.h partly via graph. Hence, tools.h acts like an interface between graph.o and main.o .
The next step was to implement the graph algorithms. I originally proposed to implement the travelling salesman problem but decided to implement a set of graph analysis functions. It is because the travelling salesman problem would have rewuired only 2 functions to determine the lower and upper bound of the problem, while little bit more tasks running in parallel could demonstrate and use the capabilities of PVM more.
The following graph algorithms were implemented :
Regular.c : A graph is regular of degree n if all of its vertices have the same degree.
The degree of the graph is the number of incident edges to it, but in our case, I always assume that a graph is undirected, hence there are no incoming and outgoing vertices, only connectivity is examined to increase generality.
Hence, regular takes a graph and examines all of its vertices if they are all of the same degree.
If yes, returns true and the value of the degree. Otherwise, returns false and 0 as the degree.

Connected.c : A graph is connected of there is a path between any nodes of the graph.Hence, connected examines if every node is accessible from any node by building up an accessibility list and then checking if all of them are true.

A series of functions had to be implemented next to support the general functionality of the program. These were
end() : to exit from the program and free up memory space
cls() : to clear the screen in a primitive way
save(): to save a graph from memory to file
new() : to input a new graph from screen to memory
init(): to initialize global variables

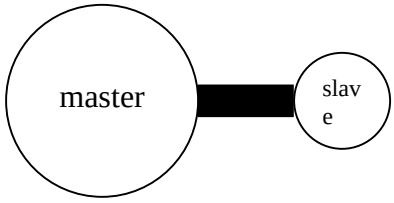
3.2 Run-time analysis

As the main data is a dynamic data structure, it can shrink and grow over execution. The size of the data varies, computational time will vary as well. But because the calculation performed on the data is distributed, it is not sure how the computaion time will vary. Computational time can depend on many factors, including :

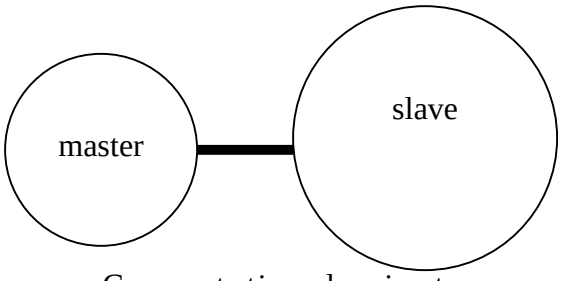
- network speed
- master and slave host(s) computational speed
- architecture type of hosts
- configuration of acrhitecture of hosts
- platform and platform configuration
- complexity of calculation
- number of tasks

All of the slave pvm tasks have computational complexity if $O(n^2)$ because all of them traverse through the matrix at least once. This requires a double loop and hence n^2 calculations at least, where n is the size of the matrix because the matrix has to be square.

As the size of matrix varies, load balance varies as well. To determine The calculation performance, I implemented an analysis function that generates 5 matrices of different size and measures the time taken to perform the calculation in seconds and milliseconds.
From the analysis it seems that the graph algorihms are not complex enough to perform them on a remote host is the network is slow.
It is because data transfer takes more time than calculating the result.
Also, some of the graph algorithms are complex and would be better if they on a remote, but fast host because the computational time becames more dominant than data transfer. Hence, the conclusion is that tasks that require a lot of computation can be distributed to a remote but fast host with slow network connection. Taks that require little computational time should be run on a host to where the bandwith is good.
Also, there is the third possibility that neither computation nor data transfer will dominate the distribution, in this case a compromise solution Is to use host with similar levels of bandwith and computaional resource.



Bandwith dominates.



Computation dominates.

Chapter 4 Technical know-how of make and PVM

Both the use of make and PVM was new to me, and hence, I decided to explain the skills I learned independently on my own. I decided to decompose the program by using make instead of imake or aimk because make was good enough to do that.

1. Portability issue was no concern because I wrote the code in ANSI/ISO C and the PVM libraries were already compiled with platform independent aimk.
2. The program didn't require special treatment that would exceed the limitations of make.

PVM was advised by my superior Dr. Fraser Mitchell. I have never heard of PVM before Dr. Fraser told me about it. Now, I appreciate being introduced to PVM since I found out the many versatile but also some disadvantages.

1. Industry standard in distributed programming
2. Applications ported to PVM, POV(Perception Of Vision)

4.1 Make

In its simplest form, a make file contains make rules of the form:

target list: dependency list
command list

Where target list is a list of target files and depencylist is a list of files that the files in target list depend on. For example, let's think about the file inter-dependencies related to the main executable file. This file is built out of three object modules:

- main.o
- tools.o
- graph.o

If either file is changed, then the main may be reconstructed by compiling the modified module and linking it again with ld. Therefore, one rule in my make file is :

```
main: main.o tools.o graph.o
cc –L./lib/LINUX main.o tools.o graph.o –lpvm3 –o main
```

Please note that the extra arguments were required because graph.c uses the pvm3 library but the pvm3 library only has to be loaded and linked when *make* builds the main executable.
Hence, **-L./lib/LINUX** specifies the optional search library for the *ld* tool to look for archive library files.
In this case, the pvm3 library file is called libpvm3.a .
-lpvm3 specifies for the *ld* command to link and use the **pvm3.h** header file and any required object file of its functions.

The rule for specifying the main executable is usually declared at the top of the make file. It is because the order of the rules declared in the make file is evaluated from top to bottom.
The make system than visits each rule associated with each file in the dependency list and performs the same action.

4.2 PVM

I didn't find the MIT PVM book comprehensive enough to install and configure PVM and then write and use PVM applications without spending a considerable time on trial and error techniques and studying:
: the UNIX shell
: UNIX compiling, loading and linking mechanism

I turned to use a few other resources such as
- UNIX for programmers and users (Prentice Hall)
- The design of the UNIX operating system (Prentice Hall).
- a series of web pages on how to install, configure and use PVM.

Step 1: Obtaining PVM

PVM can be obtained from the University of Tennessee's ftp web server. Hence, at the UNIX prompt, in our home directory with enough disk quota, we should type:

>ftp ftp.netlib.org	connect to ftp server
>bin	change file transfer type to binary
>cd pvm3	change remote directory
>ls	list available files
>get pvm3.4.3.tgz	get the compressed pvm3 source code
>by	disconnect from ftp server
>gzip –d pvm3.4.3.tgz	decompress gzipped file
>tar –xfv pvm3.4.3.gz	decompress tar file

Step 2 Configuring PVM
edit **.cshrc** file in home directory and insert the following parameters:

setenv PVM_ROOT "your absolute path of home directory concatenated with the pvm3 directory"

We can now copy this path to the edited .cshrc window.
Insert the next line:

setenv PVM_ARCH "Architecture name"
where architecture name is a variable and specifies your current hosts architecture. It is necessary to specify the architecture because aimk will compile the PVM source code in terms of this variable to perform platform dependent modifications during compilation.

PVM_ARCH can be either determined by using **setenv PVM_ARCH '$PVM_ROOT/lib/pvmgetarch'** or issuing the following UNIX command which also determines the platform as well but then we will have to look up the corresponding **PVM_ARCH** variable from a table.
The command is as follows:

```
>uname –a
```

It is also recommended to set up a path for our compiled PVM binary executables held in **'$PVM_ROOT/bin/$PVM_ARCH'**. It is because every time we execute a PVM application, the **pvmd** daemon will look for slave task executables in **'$PVM_ROOT/bin/$PVM_ARCH'** and hence it is useful to include this directory in the **path** environment variable as well. Hence, the third line to be inserted is:

```
setenv MY_PVM_BINARIES '$HOME/sources'
```

and consequently the fourth line will be:

```
set path = ($path $MY_PVM_BINARIES/$PVM_ARCH
          $PVM_ROOT/bin/$PVM_ARCH $PVM_ROOT
```

We now should be able to compile PVM and reveice a message from **aimk** about the result
PVM uses **rsh** to execute programs remotely. Therefore, we have to make sure that the hosts that we would like to add to the parallel virtual machine are capable of receving **rsh** requests from the master host.
It can be done by creating **.rhosts** file in our home directory and specifying the hosts and users to which we want to grant access through an **rsh** request. It happens very similarly as well under MS-Windows platform.

Adding hosts can be achieved by either setting up a hostfile in the **PVM_ROOT** directory or adding it manually by using the **add** pvm console command. Other important pvm console commands include :

spawn : starts a pvm task
conf : lists the configuration of the virtual machine
jobs : lists currently running jobs
delete : can be used to delete hosts from the virtual machine
quit : exit from pvm, but let the pvmd daemon running
halt : exit from pvm and kill the pvmd daemon

Chapter 5 Conclusion and the future through an example

Even if distributed programming is been around for a while, it seems like the real emergence of distributed programming and applications is only having its early days but boosted with a series of applications. I would like to introduce a few of these applications through an example of chemical molecular simulation. Such programs are Biodesigner and PVM_POV.

5.1 POV-Ray

POV stands for (Perception Of Vision) and is a ray-tracer that creates photo-realistic images. It reads in a text file containing information describing

- objects
- lighting
- camera

and generates an image of that scene from the viewpoint of a camera. POV is a very high quality software and used in many areas of computer graphics. POV has the following main advantages:

- free program
- comes with source code
- uses state of the art rendering algorithms
- runs on basically every platform and architecture

About ray-tracing:

Ray tracing is a rendering technique that calculates an image of a scene by simulating the way rays of light travel in the real world. However, it does its job backwards. In the real world, the ray comes from the light emitting object, sun for example and Than the photons are being reflected, absorbed by objects or passes through on objects.
This light than hits a camera lens or our eyes. But because the vast majority of rays never hit an observer, it would take forever to trace a scene. Ray tracing, hence, starts with their illuminated camera and trace rays backwards out into the scene. The user specifies the location of the camera, light sources and objects as well as the surface texture properties of objects, their interiors (if transparent) and any atmospheric media such as fog, haze, or fire. For every pixel in the final image, one or more viewing rays are shot from the camera, into the scene to see if it intersects with any of the objects in the scene. These viewing rays originate from the viewer, represented by the camera and pass through the viewing window (representing the final image).
Every time an object is hit, the color of the surface at that point is calculated. For this purpose rays are sent backwards to each list source to determine the amount of light coming from the source. These “shadow rays” are rested to tell whether the surface point lies in shadow or not. If the surface is reflective or transparent new rays are set up and traced in order to determine the contribution of the reflected and refracted light to the final surface colour.

Also, special features like inter-diffuse reflection (radiosity), atmospheric effects and area lights make it necessary to shoot a lot of additional rays into the scene for every pixel.

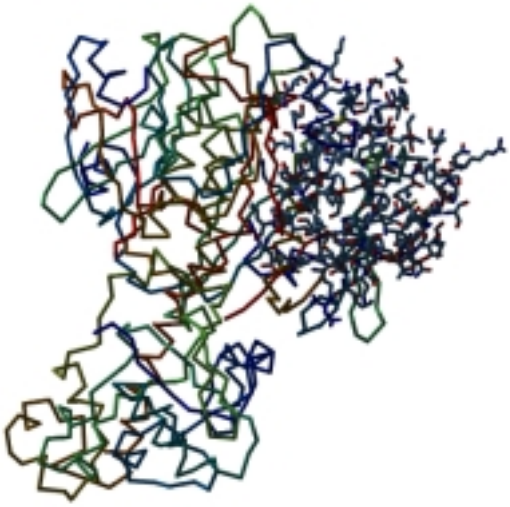
5.2 PVM_POV

PVM works as usual on POV as well, there is a master task and several other slaves. The master process has the responsibility of dividing the image up into small blocks, which are assigned to slaves. When the slaves have finished rendering the blocks, they are sent back to the master, this combines them to form the final image. The master does not render anything by itself, although there is usually a slave process running on the same machine as the master, since the master doesn't use very much CPU power. PVMPOV starts the slaves at a reduced priority to avoid annoying the users on the other machines. The slave tasks will also automatically time out if the master fails, to avoid having lots of lingering slave tasks in case we kill the master. PVMPOV uses grids to perform rendering. The size of the grid represents the resolution of the scene to which it is been decomposed. Hence, setting it to a considerable fine value (64x64) it will increase the performance of network balance because of the large chunks of the scene to be rendered, a higher granularity of tasks helps smoothly distribute the rendering tasks among the hosts.
I tried to represent this through a flame simulation in 3D Studio.
The major problem in choping off the 3d environment is that the environment is heterogeneous, hence some parts are complex some parts doesn't need rendering time at all. So, even if increase granularity, the tasks will still be radically different in computational terms. This problem is very similar to the compression of jpeg images or mpeg technology.
A rendering of a 1024x768 scence on PVM_POV produced the following results with 6 hosts and granularity of a 64x64 grid.
PVM Task Distribution Statistics:

host name	[done]	[late]	host name	[done]	[late]
cs1	[6.09%]	[0.00%]	cs1	[4.70%]	[0.19%]
cs1	[4.85%]	[0.29%]	cs1	[4.73%]	[0.15%]
cs5	[5.61%]	[0.19%]	cs5	[4.30%]	[0.00%]
cs5	[5.54%]	[0.26%]	cs5	[5.77%]	[0.14%]
cs6	[4.81%]	[0.00%]	cs6	[5.32%]	[0.13%]
cs6	[5.40%]	[0.06%]	cs6	[4.89%]	[0.13%]
cs7	[3.79%]	[0.00%]	cs7	[4.81%]	[0.13%]
cs7	[3.32%]	[0.00%]	cs7	[3.90%]	[0.00%]
cs2	[5.54%]	[0.06%]	cs2	[6.89%]	[0.13%]
cs2	[4.59%]	[0.45%]	cs2	[5.14%]	[0.26%]

5.3 Biodesigner

There are many areas of computer science i mentioned in the text and one of them is molecular chemistry. One such program is Biodesigner. Biodesigner is a free general purpose molecular chemistry modeling and visualization program for personal computers. It can use POV as an external visualisation tool and hence PVM_POV. Unfortunatelly, Biodesigner is not a distributed application on its own.
I rendered a protein molecule using Biodesigner and used POV as the visual output as the image shows on the right.



5.4 Conclusion

Distributed programing and PVM in particular have a lot to offer, the question is, that how can we achieve high throughput by varying the configuration of the PVM program. I say program, because we rarely have the opportunity to change the speed of the network or the CPU. Hence, we can only achieve better results by writing better programs that can exploit resources as much as it is possible. In the early days, complexity was low, programming was about binary arithmetic and boolean algebra. In contrast, today, a programmer has to contend with a vast number of properties and high complexiy.
How can we achieve performance and generality without compromise in such a world ?
There are a few issues. First, in PVM, we can achieve better computational performance by changing the parameters manually after observing the computational statistics. Or, by trying to pre-determine the required parameters by analyzing the task and the environment.
This idea can be the key to automated parameter setting which runs the calculation with optimal parameters. The next question is than, what do we have to measure and than what rules apply to obtain the optimal parameters. Also, if possible, what is the optimal distributed computaional structure for the current task or is there a general structure.

I have been recently reading an interview with Paul Baran, the founding father of the internet.
The article begins with the following image :

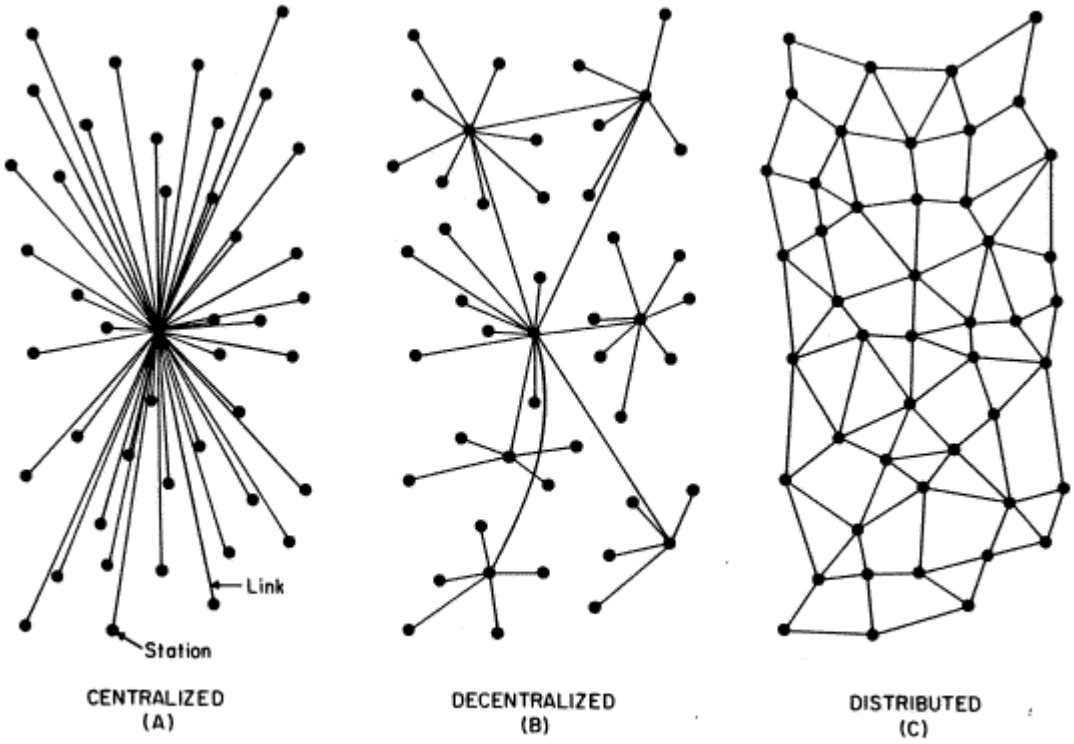


FIG. 1 — Centralized, Decentralized and Distributed Networks

This image represents the evolution of the internet, and I think it also represents the evolution of computational models as well. If we think about it, in the early days, only one task was running at a time and was using other resources. Which, indeed looks like the (A) part of the image. Then Unix came and multi-tasking multi-user systems appeared where several tasks running simulatenously, most of them using the master-slave computational model. And indeed, that looks like the (B) part of the image. But there is a third computational. And that doesn't have a master process nor a slave. The key in this computation is being equal. Being anything central has to be eliminated, and rather, words like groups and rings should be used for describing simultaniously running tasks.
This is my very personal opinion and I think that ideas like round-robin solution already apply this technology. Will this be the end of computational method evolution ? Definitely not, but I am not sure if hardware and sofware will ever merge at all. But new technologies are on their way to come. Some of them are holographic storage devices, biological, plasma and optical processors.

Also, in Japan, in Kamioka observatory, people have sent a single neutrino to a distance of 250kms underground, through earth. Neutrino is a particle and it travels with the speed of light, but it can go through any material, hence, very difficult to detect. In Japan, people are working hard to improve these transmission technology and some people use it to study the inner processes of our Sun.

It could be used as a communication device at a speed of light also wireless and noiseless. Maybe, this and other emerging communication technologies will soon expand our information exchange system, the network. The network is what matters most, it fosters convergence in all terms and decentralises power, through what life will benefit and increase to ever higher levels.

Appendix A Source code

A.1 makefile

```
A_DIR = ../lib/LINUX

all : main regular eulerian degree_s complete chromatic_n chromatic_i

#building main
main: main.o tools.o graph.o
      cc -L${A_DIR} main.o tools.o graph.o -lpvm3 -o main

#building regular
regular:regular.o graph.o
      cc -L${A_DIR} regular.o graph.o -lpvm3 -o regular

#building eulerian
eulerian:eulerian.o graph.o
      cc -L${A_DIR} eulerian.o graph.o -lpvm3 -o eulerian

#building degree sequence
degree_s:degree_s.o graph.o
      cc -L${A_DIR} degree_s.o graph.o -lpvm3 -o degree_s

#building complete
complete:complete.o graph.o
      cc -L${A_DIR} complete.o graph.o -lpvm3 -o complete

#building chromatic number
chromatic_n:chromatic_n.o graph.o
      cc -L${A_DIR} chromatic_n.o graph.o -lpvm3 -o chromatic_n

#building chromatic index
chromatic_i:chromatic_i.o graph.o
      cc -L${A_DIR} chromatic_i.o graph.o -lpvm3 -o chromatic_i

#rules are as follows

main.o:     main.c tools.h graph.h

tools.o:tools.c graph.h

regular.o:regular.c graph.h

chromatic_n.o:chromatic_n.c graph.h

chromatic_i.o:chromatic_i.c graph.h

eulerian.o:eulerian.c graph.h

degree_s.o:degree_s.c graph.h

complete.o:complete.c graph.h

graph.o:graph.c
```

A.2 main.c

```
//main block of graph
//create sub menu

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "tools.h"
#define  BUFFER_SIZE 1024     //size of read file buffer
#define  ROW_SIZE    2048     //maximum number of characters,single row
#define FUN_MAX 7             //number of functions in menu
#define PERIOD limits[1]      //case period size, interval
                              //in which the program examines
                              //the selections frequency

typedef struct {
  void   (*function)();       //pointer to function
  int        priority,        //priority of functions
       cases;                 //number of use in 17 cases
} function_t;

function_t functions[FUN_MAX];

main (int argc, char *argv[])
{
        FILE   *file_ptr;           //file pointer

        char   buffer[BUFFER_SIZE],   //file readin buffer
               current_row_[ROW_SIZE],//single row buffer
               *token_ptr,            //to tokenise a single row
               option_c;              //user option

    matrix_t   *current_row,        //pointer to current row
               *temp_row;           //when creating new (row node)!

       row_t   *current_node = NULL,//pointer to current node
               *temp_node    = NULL;//when creating new node

   boolean_t   first_trial,         //first try of loading a file
               correct_file,        //check file before loading
               first_load,          //first load of file as argument
               first_row;           //for counting entries in row
                                    //to determine dimension of matrix

      struct   stat statbuffer;     //getting file information

    unsigned   dimension;           //dimension of matrix

         int   i,                   //loop guard
               option;              //user choice
  function_t   temp;                //for swapping functions in array

const char *menu_text[FUN_MAX]={"Open file",
                                "Save",
                                "New graph",
                                "Display adjacency matrix",
                                "Properties of graph",
                                "PVM options",
                                "Exit"};
```

```
const int limits[FUN_MAX]={0,11,7,5,3,2,1};

      for(i=0;i<FUN_MAX;i++) functions[i].priority=i;
      functions[0].function=load;
      functions[1].function=save;
      functions[2].function=new;
      functions[3].function=display;
      functions[4].function=display_info;
      functions[5].function=pvm_options;
      functions[6].function=end;

      init();
      if (argc>1) {
        memcpy(info.filename,argv[1],strlen(argv[1]));
        do{
          first_load    =TRUE;
          first_trial   =TRUE;
          correct_file  =TRUE;
          irst_row      =TRUE;
          dimension     =0;
          info.size     =0;
          do{
            if (!first_trial && !correct_file) {
              printf("'%s' may not exist or not a valid file. ",
                    info.filename);
              printf("\nWould you like to try again ? (y/n): ");
              getchar();
              scanf("%c", &option_c);
              option=toupper(option_c);
            }
            correct_file      =TRUE;
            if (!first_load) {
              printf("Please enter the name of the graph file : ");
              scanf("%s", &info.filename);
            }
            first_load =FALSE;
            first_trial=FALSE;
            if ((stat(info.filename, &statbuffer)==-1) &&
                (statbuffer.st_size==0))   {
              correct_file=FALSE;
              printf("'%s' is empty. ", info.filename);
            }
            if ((file_ptr=fopen(info.filename, "r"))== NULL)
              correct_file=FALSE;
          }while (!correct_file);
          fscanf(file_ptr,"%s",&buffer);
          if (memcmp(buffer,"graph_name", strlen("graph_name"))!=0)
            correct_file=FALSE;
            fscanf(file_ptr,"%s%s", &buffer, &buffer);
            memcpy(info.matrix_name,buffer,NAME_SIZE);
            fgets(current_row_, ROW_SIZE, file_ptr); //ignoring next line
            fgets(current_row_, ROW_SIZE, file_ptr); //ignoring next line
          while (!feof(file_ptr)) {
          info.size++;
          fgets(current_row_, ROW_SIZE, file_ptr); //parsing current row
          temp_row=malloc(sizeof(matrix_t));       //allocate new node
          temp_row->y=NULL;
          temp_row->next_row=NULL;
          if (matrix==NULL) {                    //matrix is empty
            matrix=temp_row;
            current_row=temp_row;
```

```
        }
        else {
          current_row->next_row=temp_row;
          current_row=temp_row;
        }
        current_node=current_row->y;
        token_ptr=strtok(current_row_, " ");
        while(token_ptr!=NULL) {         //parsing single entries
          if (first_row) dimension++;
            temp_node=malloc(sizeof(row_t));
          temp_node->weight=atoi(token_ptr);
          temp_node->x=NULL;
          if (current_row->y==NULL)
            current_row->y=temp_node;
          else
            current_node->x=temp_node;
          current_node=temp_node;
          token_ptr=strtok(NULL," ");
        }
        first_row=FALSE;
      }
      fclose(file_ptr);
      if (correct_file) {
        if (dimension!=info.size)
          correct_file=FALSE;  //if matrix is not square (NxN)
      }
      if (correct_file)
        display();
      else {
        if (dimension!=info.size) {
          printf("\nPlease declare an NxN matrix");
          printf(", the current one is %dx%d.\n", info.size,
                 dimension);
        }
        else
          printf("'%s' doesn't seem to be correct, please check it.\n",
                   info.filename);
        printf("Would you like to try again ? (y/n): ");
        getchar();
        scanf("%c", &option_c);
        if (toupper(option_c)=='Y') {
          correct_file=FALSE;
          first_trial=TRUE;
        }
        else
          correct_file=TRUE;
      }
    }while(!correct_file);
  }
  do {
    for (i=0; i<FUN_MAX; i++)
      printf("\t\t%d-%s\n", i, menu_text[functions[i].priority]);
      scanf("%d", &option);
      if ((option < 0) || (option > FUN_MAX-1)) {
        printf("Please select a value between 0 and %d.\n", FUN_MAX-1);
        printf("Thank you.\n");
      }
      else {
        functions[option].function();
        printf("\n");
        functions[0].cases++;
        if (functions[0].cases==PERIOD)
          for (i=0;i<FUN_MAX;i++)
```

```
            functions[i].cases=0;
          if (option!=0) {
            functions[option].cases++;
            if (functions[option].cases>=limits[option]) {
              if (option==FUN_MAX-1)
                functions[option].cases=0;
              functions[0].cases=0;
              if (functions[option].cases > functions[option-1].cases) {
                temp=functions[option];
                functions[option]=functions[option-1];
                functions[option-1]=temp;
              }
            }
            functions[option].cases=
            (functions[option].cases % limits[option]);
          }
        }
      }while(option!=FUN_MAX-1);
}
```

A.3 tools.c

```
// tools.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "graph.h"
#include "../include/pvm3.h"
#define  BUFFER_SIZE 1024          //size of read file buffer
#define  ROW_SIZE    2048          //maximum number of characters for a
single row

//maximum >lenght< of a single number can not be more then BUFFER_SIZE-2
//because of string concatenation in save()

void load();                       //load file to memory
void display();                    //displays a matrix
void display_path(row_t *path);    //displays a path
void cls();                        //clears screen
void end();                        //exit from program
void display_info();               //displays info of graph
void save();                       //save matrix to file
void new();                        //new matrix
void init();                       //initialize global variables
void pvm_options();                //pvm options


void cls()                         //clears screen
{
      int i;

      for(i=0; i<=80; i++)
        printf("\n");
}
```

```
//load matrix to memory from file
void load()
{
      FILE   *file_ptr;             //file pointer

      char   buffer[BUFFER_SIZE],   //file readin buffer
             current_row_[ROW_SIZE],//single row buffer
             *token_ptr,            //to tokenise a single row
             option;                //user option

    matrix_t *current_row,          //pointer to current row
             *temp_row;             //when creating new (row node)!

       row_t *current_node = NULL,  //pointer to current node
             *temp_node    = NULL;  //when creating new node

   boolean_t first_trial,           //first try of loading a file
             correct_file,          //check file before loading
             first_row;             //for counting numberof entries
                                    //to determine dimension of matrix

     struct  stat statbuffer;       //getting file information

   unsigned  dimension;             //dimension of matrix

      if (matrix==NULL) {
        do{
          first_trial   =TRUE;
          correct_fie   =TRUE;
          first_row     =TRUE;
          option        ='Y';
          dimension     =0;
          info.size     =0;
            do{
              if (!first_trial && !correct_file) {
                printf("'%s' may not exist or not a valid file. ",
                     info.filename);
                printf("\nWould you like to try again ? (y/n): ");
                getchar();
                scanf("%c", &option);
                option=toupper(option);
              }
              if (option=='Y') {
                init();
                correct_file  =TRUE;
                printf("Please enter the name of the graph file : ");
                scanf("%s", &info.filename);
                first_trial=FALSE;
                if ((stat(info.filename, &statbuffer)==-1) &&
                    (statbuffer.st_size==0))     {
                  correct_file=FALSE;
                  printf("'%s' is empty. ", info.filename);
                }
                if ((file_ptr=fopen(info.filename, "r"))== NULL)
                  correct_file=FALSE;
              } //end of  if (option=='Y')
              if (option!='Y') correct_file=TRUE;
            }while (!correct_file);
            if (option=='Y') {
              fscanf(file_ptr,"%s",&buffer);
              if (memcmp(buffer,"graph_name", strlen("graph_name"))!=0)
                correct_file=FALSE;
```

```
  fscanf(file_ptr,"%s%s", &buffer, &buffer);

  memcpy(info.matrix_name,buffer,NAME_SIZE);
  fgets(current_row_, ROW_SIZE, file_ptr);//ignoring line
  fgets(current_row_, ROW_SIZE, file_ptr);//ignoring line
  while (!feof(file_ptr)) {
    info.size++;
    fgets(current_row_, ROW_SIZE, file_ptr);//parsing row
    temp_row=malloc(sizeof(matrix_t));     //allocate node
    temp_row->y=NULL;
    temp_row->next_row=NULL;
    if (matrix==NULL) {                     //matrix is empty
      matrix=temp_row;
      current_row=temp_row;
    }
    else {
      current_row->next_row=temp_row;
      current_row=temp_row;
    }
    current_node=current_row->y;
    token_ptr=strtok(current_row_, " ");
    while(token_ptr!=NULL) {         //parsing single entries
      if (first_row) dimension++;
      temp_node=malloc(sizeof(row_t));
      temp_node->weight=atoi(token_ptr);
      temp_node->x=NULL;
      if (current_row->y==NULL)
        current_row->y=temp_node;
      else
        current_node->x=temp_node;
      current_node=temp_node;
      token_ptr=strtok(NULL," ");
    }
    first_row=FALSE;
  }
  fclose(file_ptr);
  if (correct_file) {
    if (dimension!=info.size) //if matrix is not square (NxN)
     correct_file=FALSE;
  }
  if (correct_file)
    display();
  else {
    if (dimension!=info.size) {
      printf("\nPlease declare an NxN matrix");
      printf(", the current one is %dx%d.\n", info.size,
             dimension);
    }
    else
      printf("'%s' doesn't seem to be correct,
             please check it.\n", info.filename);
    printf("Would you like to try again ? (y/n): ");
    getchar();
    scanf("%c", &option);
    if (toupper(option)=='Y') {
      correct_file=FALSE;
      first_trial=TRUE;
    }
    else
      correct_file=TRUE;
    }
  } //end of  if (option=='Y')
}while(!correct_file);
```

```
          } // if (matrix==NULL)


        else {
          printf("'%s' graph is already loaded, ", info.matrix_name);
          printf("\nWould you like to close it and load a new file ?
                 y/n): ");
          getchar();
          scanf("%c", &option);
          if (toupper(option)=='Y') {
            printf("'%s' closed.\n",info.matrix_name);
            init();
          }
          else {
            printf("'%s' remains loaded\n",info.matrix_name);
          }
        }
      }

void display()                      //displays a matrix
{
      matrix_t *current_row;
      row_t    *current_node;

      if (matrix==NULL) {
        printf("There were no graphs loaded yet,\n");
        printf("Please load or create a graph first.\n");
      }
      else {
        current_row=matrix;
        current_node=current_row->y;
        printf("'%s' graph has the corresponding adjacency matrix :
               \n",info.matrix_name);
        while (current_row != NULL) {     //outer loop for rows
          while (current_node != NULL) {  //inner loop for columns
            printf(" %d", current_node->weight);
            current_node=current_node->x;
          }
          current_row=current_row->next_row;
          if (!current_row=='\0') {
            current_node=current_row->y;
            printf("\n");
          }
        }
        printf("\n");
       }
}

void end()              //exit from program
{
      cls();
      free(matrix);
      printf("by.\n");
      exit(0);
}

void display_info()     //display properties of the current graph
{
      int x;

      info=get_info(matrix);
      if (matrix==NULL) {
        printf("\nNo graph have been loaded yet.\n");
```

```
      }
      else {
        cls();
        for (x=0;x<70;x++) printf("_");
        printf("\n%s graph has the following properties
               :\n",info.matrix_name);
        printf("\ngeometry        :%dx%d", info.size, info.size);
        printf("\nregular         :");
        if (info.regular.is)
          printf("yes, degree is : %d", info.regular.value);
        else
          printf("no");
        printf("\nchromatic number:");
        if (info.chromatic_number!=-1)
          printf("X(G) < %d", info.chromatic_number);
        else
          printf("X(G) undetermined because graph contains self loops.");
        printf("\nchromatic index :");
        if (info.chromatic_index!=-1)
          printf("%d < X'(G) < %d", info.chromatic_number,
                                    info.chromatic_number+1);
        else
          printf("X'(G) undetermined because graph contains self
                 loops.");
        printf("\neulerian        :");
        if (info.eulerian)
          printf("yes");
        else
          printf("no");
        printf("\ndegree sequence :");
        display_path(info.degree_sequence);
        printf("\ncomplete        :");
        if (info.complete)
          printf("yes");
        else
          printf("no");
      }
      printf("\n");
      for (x=0;x<70;x++) printf("_");
      printf("\n\n");
}

//saves the current matrix while using info file for names
void save()
{
      FILE *file_ptr;               //file pointer
      char buffer[BUFFER_SIZE],     //file write buffer
           current_row_[ROW_SIZE];  //single row buffer
       int x, y;                    //matrix co-ordinates
 boolean_t filename_ok;             //if filename is correct or not

      filename_ok=TRUE;
      do {
        if (memcmp(info.filename, "", strlen(info.filename))==0) {
           printf("Please enter the name of the file :");
           scanf("%s", &info.filename);
        }
        if ((file_ptr=fopen(info.filename, "w"))==NULL) {
           printf("%s can not be saved, please give it a new name.\n",
           info.filename);
           filename_ok=FALSE;
        }
      }while((file_ptr=fopen(info.filename, "w"))==NULL);
```

```
      if (filename_ok) {
          strcpy(buffer, "graph_name = ");
          strcat(buffer, info.matrix_name);
          strcat(buffer, "\n\n\0");
          fwrite(buffer, sizeof(char), strlen(buffer), file_ptr);
          for(y=0;y<info.size;y++) {
            strcpy(buffer, "");
            strcpy(current_row_, "");
            for(x=0;x<info.size;x++) {
              if (x!=info.size-1)             //if not last row
              //convert char -> int
                sprintf(buffer,"%d \0", access_xy(matrix, x, y));
              else
                //convert char -> int
                sprintf(buffer,"%d", access_xy(matrix, x, y));
                strcat(current_row_, buffer);
              }
              if (y!=info.size-1)             //if not last column
                strcat(current_row_, "\n");
              fwrite(current_row_, sizeof(char), strlen(current_row_),
                     file_ptr);                }
            }
            printf("Successfully wrote %s.\n", info.filename);
            fclose(file_ptr);
          }

//new matrix
void new()
{
      char  option='N';             //user option
       int  i, j;                   //loop guard
  matrix_t  *current_row,           //pointer to current row
            *temp_row;              //when creating new (row node)!

     row_t  *current_node = NULL,   //pointer to current node
            *temp_node    = NULL;   //when creating new node

      if (matrix!=NULL) {
        printf("'%s' is already loaded.\n",info.matrix_name);
        printf("\nWould you like to close it and enter a new graph ?
               (y/n): ");
        getchar();
        scanf("%c", &option);
        if (toupper(option)=='Y') {
          init();
          }
      }
      else
        option='Y';
        if (toupper(option)=='Y') {
          printf("\nPlese enter the name of the graph :");
          scanf("%s", &info.matrix_name);
          printf("Please enter the size of the NxN matrix :");
          scanf("%d", &info.size);
          for (i=0;i<info.size;i++) {           //outer loop begins here
            temp_row=malloc(sizeof(matrix_t));
            temp_row->y=NULL;
            temp_row->next_row=NULL;
            if (matrix==NULL) {           //matrix is empty
              matrix=temp_row;
              current_row=temp_row;
            }
```

```
            else {
              current_row->next_row=temp_row;
              current_row=temp_row;
            }
            //set current node to first entry of row
            current_node=current_row->y;
            //parsing single entries, inner loop
            for (j=0;j<info.size;j++) {
              temp_node=malloc(sizeof(row_t));  //allocate node
              printf("Please enter %s[%d][%d]: ", info.matrix_name, i+1,
                                                                  j+1);
              scanf("%d", &temp_node->weight);
              temp_node->x=NULL;      //set temp to null for security
              if (current_row->y==NULL)
                current_row->y=temp_node;
              else
                current_node->x=temp_node;
              current_node=temp_node;
            }
          }
        }
        else {
          printf("'%s' remains loaded\n",info.filename);
        }
}

//pvm options
void pvm_options()
{
      struct pvmhostinfo *pvm_info;       //to obtain system information
                     int n_hosts,         //number of hosts
                         n_arch,          //number of architectures
                         i, j, x,         //loop guard
                         option,          //user option
                         matrix_size;     //temporary matrix size
         struct timeval time_value_a,     //time before calculation
                         time_value_b,    //time after  calculation
                         times[5];        //intervalls of calculations
          struct timezone time_zone;      //unused, for timezone

                matrix_t *temp_matrix=NULL,//temporary matrix
                         *current_row,     //pointer to current row
                         *temp_row;        //when creating new node

                   row_t *current_node = NULL,//pointer to current node
                         *temp_node    = NULL;//when creating new node

      cls();
      do{
        printf("\n\t0-PVM configuration");
        printf("\n\t1-System analysis");
        printf("\n\t2-Return to main menu\n");
        scanf("%d", &option);
        switch (option) {
          case 0:
          if (pvm_config(&n_hosts, &n_arch, &pvm_info)<0)
            printf("Could not receive information about current vistual
                   machine.\n");
            else {
              printf("\nVirtual machine configuration :\n");
            for (i=0;i<70;i++) printf("_");
```

```
      printf("\nTID\thost name\t\tarchitecture\trelative speed");


      for(i=0;i<n_hosts;i++) {
        printf("\n%d\t%s\t%s\t\t%d",pvm_info[i].hi_tid
                                   ,pvm_info[i].hi_name
                                   ,pvm_info[i].hi_arch
                                   ,pvm_info[i].hi_speed);
      }
    }
    printf("\n");
    for (i=0;i<70;i++) printf("_");
    printf("\n");
    break;
  case 1: printf("System analysis:\n");
       for (i=0;i<70;i++) printf("_");
         for(x=0;x<5;x++) {
           if (temp_matrix!=NULL)
             free(temp_matrix);
           matrix_size=(5+(5*x*x)); //precompute test matrix size
           for (i=0;i<matrix_size;i++) {  //outer loop begins here
             temp_row=malloc(sizeof(matrix_t));
             temp_row->y=NULL;
             temp_row->next_row=NULL;
             if (temp_matrix==NULL) {           //matrix is empty
               temp_matrix=temp_row;
               current_row=temp_row;
             }
             else {
               current_row->next_row=temp_row;
               current_row=temp_row;
             }
             current_node=current_row->y;
             for (j=0;j<matrix_size;j++) {
             temp_node=malloc(sizeof(row_t));//allocate node
               temp_node->weight = (rand() % 10);
               temp_node->x=NULL;
               if (current_row->y==NULL)
                 current_row->y=temp_node;
               else
                 current_node->x=temp_node;
               current_node=temp_node;
             }
             }
       //  Linux doesn't support this call
       //  gettimeofday(&time_value_a, &time_zone
           get_info(temp_matrix);
       //  gettimeofday(&time_value_b, &time_zone);
           times[x].tv_sec = time_value_b.tv_sec –
                             time_value_a.tv_sec;
           times[x].tv_usec =time_value_b.tv_usec –
                             time_value_a.tv_usec;
         } //outer for loop_5
         printf("\nsize\tseconds\tmilliseconds");
         for(i=0;i<5;i++) {
           printf("\n%d\t%d\t%d", (5+(5*i*i)),
                                  times[i].tv_sec,
                                  times[i].tv_usec);
           }
         printf("\n");
         for (i=0;i<70;i++) printf("_");
           break;
```

```
         case 2 : cls(); break;
         default: printf("\nPlease enter a number between 0 and 2,
                          thank you.\n");
       } //switch
     }while(option!=2);

}

//initialize global variables
void init()
{
      if (matrix!=NULL)
        free(matrix);

      matrix=NULL;
      strcpy(info.filename,"");
        strcpy(info.matrix_name,"");
        info.size       =0;
        info.chromatic_index  =0;
        info.chromatic_number =0;
      info.complete           =FALSE;
      info.eulerian           =FALSE;
        info.regular.is       =FALSE;
        info.regular.value    =0;
        info.degree_sequence  =NULL;
}

void display_path(row_t *path)      //displays a path
{
       long sum=0;
      row_t *current_node;

      current_node=path;
      printf("(");
      while (current_node != NULL) {
          printf("%d", current_node->weight);
        sum += current_node->weight;
        current_node=current_node->x;
        if (current_node != NULL)
          printf("-->");
        }
      printf(") = %d.", sum);
}
```

A.4 tools.h

```
// tools.h
  #include "graph.h"                //inherit from graph_algorithms.h

  void load();                      //load file to memory
  void display();                   //displays a matrix
  void display_info();              //displays info of graph
  void display_path(row_t *);       //displays a path
  void cls();                       //clears screen
  void end();                       //exit from program
  void save();                      //save graph
  void new();                       //new graph
  void init();                      //initialize global variables
  void pvm_options();               //pvm options


  matrix_t *matrix;                 //global variable to hold graph
  info_t  info;                     //global variable to hold information
```

A.5 graph.c

```
//graph algorithms

#include <stdio.h>
#include <stdlib.h>
#include "../include/pvm3.h"
#define  NAME_SIZE   11   //maximum length of matrix name

typedef struct row {
     int weight;
  struct row *x;
} row_t;

typedef struct matrix {
  struct matrix *next_row;
  struct row *y;
} matrix_t;

typedef enum {
  FALSE,
  TRUE
} boolean_t;

typedef struct bool_int {
 boolean_t is;
  unsigned value;
} bool_int_t;

typedef struct info {
      char filename[NAME_SIZE],
           matrix_name[NAME_SIZE];
       int size,
           chromatic_index,
           chromatic_number;
 boolean_t complete,
           eulerian;
bool_int_t regular;
     row_t *degree_sequence;
} info_t;
```

```
void send_matrix(const int pvm_tid);      //send matrix to pvm slave

      matrix_t *matrix=NULL;              //stores adjacency matrix
        info_t info;                      //information about graph

info_t get_info() //obtain information about graph
{
      row_t *current,
          *temp;

      int rec_buf,            //receive buffer
          i,                  //loop guard
          regular_tid,        //pvm tid of task: regular
          chromatic_n_tid,    //pvm tid of task: chromatic number
          chromatic_i_tid,    //pvm tid of task: chromatic index
             eulerian_tid,    //pvm tid of task: eulerian
             degree_s_tid,    //pvm tid of task: degree_s
            complete_tid;     //pvm tid of task: complete

//spawning regular
      if (pvm_spawn("regular", (char**)0, 0, "", 1, &regular_tid)==1) {
        send_matrix(regular_tid);   //sending matrix to regular
      }
      else {
        printf("can't start the following slave task: regular.\n");
        pvm_exit();
      }
//spawning chromatic_n
      if (pvm_spawn("chromatic_n", (char**)0, 0, "", 1,
          &chromatic_n_tid)==1) {
        send_matrix(chromatic_n_tid);//sending matrix to chromatic number
      }
      else {
        printf("can't start the following slave task: chromatic_n.\n");
        pvm_exit();
      }
//spawning chromatic_i
      if (pvm_spawn("chromatic_i", (char**)0, 0, "", 1,
          &chromatic_i_tid)==1) {
        send_matrix(chromatic_i_tid);//sending matrix to chromatic index
      }
      else {
        printf("can't start the following slave task: chromatic_i.\n");
        pvm_exit();
      }
//spawning eulerian
      if (pvm_spawn("eulerian", (char**)0, 0, "", 1, &eulerian_tid)==1) {
        send_matrix(eulerian_tid);  //sending matrix to eulerian
      }
      else {
        printf("can't start the following slave task: eulerian.\n");
        pvm_exit();
      }

//spawning degree sequence
      if (pvm_spawn("degree_s", (char**)0, 0, "", 1, &degree_s_tid)==1) {
        send_matrix(degree_s_tid);  //sending matrix to degree_s
      }
      else {
        printf("can't start the following slave task: degree_s.\n");
        pvm_exit();
```

```
      }
//spawning complete
      if (pvm_spawn("complete", (char**)0, 0, "", 1, &complete_tid)==1) {
        send_matrix(complete_tid);  //sending matrix to degree_s
      }
      else {
        printf("can't start the following slave task: complete.\n");
        pvm_exit();
      }


//receiving results from
here_____________________________________________________________



//start receiving results from regular
      pvm_recv(regular_tid, 1);
      pvm_upkint(&rec_buf, 1, 1);
      if (rec_buf==1)
        info.regular.is=TRUE;
      else
        info.regular.is=FALSE;
      pvm_recv(regular_tid, 1);
      pvm_upkint(&rec_buf, 1, 1);
      info.regular.value=rec_buf;
//end of receiving result from regular

//start receiving results from chromatic_n
      pvm_recv(chromatic_n_tid, 1);
      pvm_upkint(&info.chromatic_number, 1, 1);
//end of receiving results from chromatic_n

//start receiving results from chromatic_i
      pvm_recv(chromatic_i_tid, 1);
      pvm_upkint(&info.chromatic_index, 1, 1);
//end of receiving results from chromatic_i

//start receiving results from eulerian
      pvm_recv(eulerian_tid, 1);
      pvm_upkint(&rec_buf, 1, 1);
      if (rec_buf==1)
        info.eulerian=TRUE;
      else
        info.eulerian=FALSE;
//end of receiving results from eulerian

//start receiving results from degree sequence

      for (i=0;i<info.size;i++) {   //building up sequence from here
        temp=malloc(sizeof(row_t));
        pvm_recv(degree_s_tid, 1);
        pvm_upkint(&rec_buf, 1, 1);
        temp->weight=rec_buf;
        if (info.degree_sequence==NULL) {
          info.degree_sequence=temp;
          current=info.degree_sequence;
        }
        else {
          current->x=temp;
          current=current->x;
        }
      }
```

```
      current->x=NULL;        //end of building up degree sequence
//end of receiving results from degree sequence
//start receiving results from complete
      pvm_recv(complete_tid, 1);
      pvm_upkint(&rec_buf, 1, 1);
      if (rec_buf==1)
        info.complete=TRUE;
      else
        info.complete=FALSE;
//end of receiving results from eulerian
//end of receiving
results_____________________________________________________
pvm_exit();
return info;      //return info to display_info() in tools.c
}


//accesses a matrix's entry absolutely
int access_xy(matrix_t *matrix, const int x, const int y)
{
      matrix_t *current_row;
         row_t *current_node;
           int index_x, index_y;

      current_row=matrix;
      for(index_y=0;index_y<y;index_y++)
        current_row=current_row->next_row;
      current_node=current_row->y;
      for(index_x=0;index_x<x;index_x++)
        current_node=current_node->x;

      return current_node->weight;
}

//send matrix to pvm task with given tid
void send_matrix(const int pvm_tid)
{
      int i, j,         //loop control
          send_buf;     //send buffer

        pvm_initsend(PvmDataDefault);//sending the size of the matrix
        pvm_pkint(&info.size, 1, 1);
        pvm_send(pvm_tid, 1);

        for(i=0;i<info.size;i++) {              //sending matrix
          for(j=0;j<info.size;j++) {
            send_buf=access_xy(matrix, j, i);
            pvm_initsend(PvmDataDefault);
            pvm_pkint(&send_buf, 1, 1);
            pvm_send(pvm_tid, 1);
         }
        }
}
```

A.6 graph.h

```
//graph.h
#define  NAME_SIZE   11   //maximum length of matrix name

typedef enum {
  FALSE,
  TRUE
} boolean_t;

typedef struct row {
  int weight;
  struct row *x;
} row_t;

typedef struct matrix {
  struct matrix *next_row;
  struct row *y;
} matrix_t;

typedef struct bool_int {
  boolean_t  is;
  unsigned value;
} bool_int_t;

typedef struct info {
        char filename[NAME_SIZE],
             matrix_name[NAME_SIZE];
         int size,
             chromatic_index,
             chromatic_number;
   boolean_t complete,
             eulerian;
  bool_int_t regular;
  row_t      *degree_sequence;
} info_t;

      matrix_t *matrix;
        info_t info;

   info_t get_info();               //for display_info() in tools.c

  //for save() in tools.c
  int access_xy(matrix_t *, const int, const int);
  //send matrix to pvm task with given tid
  void send_matrix(const int);
```

A.6 regular.c

```
#include <stdio.h>
#include <stdlib.h>
#include "graph.h"
#include "../include/pvm3.h"

#define  NAME_SIZE   11             //maximum length of matrix name

bool_int_t regular();

        matrix_t *temp_matrix;      //to store received matrix

main()
{
      bool_int_t temp;              //to store result of regular()

        matrix_t *current_row,      //pointer to current row
                 *temp_row;         //when creating new (row node)

           row_t *current_node,     //pointer to current node
                 *temp_node;        //when creating new node

             int ptid,              //parent tid
                 msgtag,            //message tag
                 send_buf,          //send buffer
                 rec_buf,           //receive buffer
                 size,              //size of matrix
                 i, j,              //loop control
                 total;             //delete it later

      ptid = pvm_parent();
      msgtag=1;
      pvm_recv(ptid, msgtag);
      pvm_upkint(&size, 1, 1);
//end of receiving the size of the matrix


//receive matrix

      for (i=0;i<size;i++) {              //outer loop begins here
        temp_row=malloc(sizeof(matrix_t));
        temp_row->y=NULL;
        temp_row->next_row=NULL;
        if (temp_matrix==NULL) {          //matrix is empty
          temp_matrix=temp_row;
          current_row=temp_row;
        }
        else {
          current_row->next_row=temp_row;
          current_row=temp_row;
        }
        //set current node to the first entry of row
        current_node=current_row->y;
        for (j=0;j<size;j++) {      //parsing single entries, inner loop
          pvm_recv(ptid, msgtag);   //initialize reveiving
          //receive single entry from parent
          pvm_upkint(&rec_buf, 1, 1);
          temp_node=malloc(sizeof(row_t));//allocate node for new entry
          temp_node->weight=rec_buf;//assign received value to new node
          temp_node->x=NULL;              //set temp to null for security
          if (current_row->y==NULL)
```

```
            current_row->y=temp_node;
          else
            current_node->x=temp_node;
          current_node=temp_node;
        }
      }
      total=0;
      for(i=0;i<size;i++) {
        for(j=0;j<size;j++) {
           total += access_xy(temp_matrix, j, i);
        }
        }
      pvm_initsend(PvmDataDefault);
      pvm_pkint(&total, 1, 1);            //set value to send buffer
      pvm_send(ptid, msgtag);


      temp=regular(temp_matrix);          //perform graph analysis
      if (temp.is==TRUE)                  //check result, stage 1
         send_buf=1;
      else
         send_buf=0;
      send_buf=total;
      pvm_initsend(PvmDataDefault);
      pvm_pkint(&send_buf, 1, 1);         //set value to send buffer
      pvm_send(ptid, msgtag);             //send it bask tp parent task
      send_buf=temp.value;                //send back other info, stage 2
      pvm_initsend(PvmDataDefault);
      pvm_pkint(&send_buf, 1, 1);         //set value to send buffer
      pvm_send(ptid, msgtag);             //send it bask tp parent task
      pvm_exit();
}

//determines if graph is regular or not
//if yes, returns degree of graph
//if not, returns largest degree of graph
//in info.value and info.is
bool_int_t regular()
{

 bool_int_t info_temp;
      int x, y,
          current_degree;
      row_t *sequence=NULL,
          *current,
          *temp;

      info_temp.is=TRUE;
      current=sequence;
      for (x=0;x<info.size;x++) {
        temp=malloc(sizeof(row_t));
        if (sequence==NULL)
          sequence=temp;
        else
          current->x=temp;
        current=temp;
      }
      current->x=NULL;

      for(y=0;y<info.size;y++)
          for(x=0;x<info.size;x++)
          if (access_xy(temp_matrix, x, y)!=0) {
             sequence[x].weight++;
```

```
            sequence[y].weight++;
           }
      info_temp.value=sequence[0].weight;

      for(x=1;x<info.size;x++)
           if (info_temp.value!=sequence[x].weight)
           info_temp.is=FALSE;

      return info_temp;
}
```

A.7 complete.c

```
#include <stdio.h>
#include <stdlib.h>
#include "graph.h"
#include "../include/pvm3.h"

boolean_t complete();              //determines if graph is complete or
not


        matrix_t *temp_matrix;     //to store received matrix
             int size;             //size of matrix


main()
{
        matrix_t *current_row,     //pointer to current row
                 *temp_row;        //when creating new (row node)

           row_t *current_node,    //pointer to current node
                 *temp_node;       //when creating new node

             int ptid,             //parent tid
                 msgtag,           //message tag
                 send_buf,         //send buffer
                 rec_buf,          //receive buffer
                 i, j;             //loop control

      ptid = pvm_parent();    //obtaining parent tid for communication
      msgtag=1;               //there is only one copy of task running
      pvm_recv(ptid, msgtag);
      pvm_upkint(&size, 1, 1);      //receiving the size of matirx

                                    //begin to receive matrix

      for (i=0;i<size;i++) {              //outer loop begins here
        temp_row=malloc(sizeof(matrix_t));      //allocate new node
        temp_row->y=NULL;
        temp_row->next_row=NULL;
        if (temp_matrix==NULL) {          //matrix is empty
          temp_matrix=temp_row;
          current_row=temp_row;
        }
        else {                //if matrix is not empty, not first row
          current_row->next_row=temp_row;
          current_row=temp_row;
        }
        current_node=current_row->y;//set current node to the first entry
        for (j=0;j<size;j++) {      //parsing single entries, inner loop
          pvm_recv(ptid, msgtag);         //initialize reveiving
          pvm_upkint(&rec_buf, 1, 1);//receive single entry from parent
```

```
          temp_node=malloc(sizeof(row_t));//allocate node for new entry
          temp_node->weight=rec_buf;//assign received value to new node
          temp_node->x=NULL;              //set temp to null for security
          if (current_row->y==NULL)
            current_row->y=temp_node;
          else
            current_node->x=temp_node;
          current_node=temp_node;
        }
      }
      if (complete())                     //set result to send buffer
        send_buf=1;
      else
        send_buf=0;
      pvm_initsend(PvmDataDefault);       //initialize send buffer
      pvm_pkint(&send_buf, 1, 1);
      pvm_send(ptid, msgtag);             //send it back to parent task

      pvm_exit();
}




//determines if graph is complete or not
boolean_t complete()
{
      int i, j,         //loop guard
          degree;       //current vertex degree
boolean_t valid=TRUE;   //if graph is complete or not

      for(i=0;i<size;i++) {
        for(j=0;j<size;j++) {
          if (i!=j && access_xy(temp_matrix, i, j)!=0)
            degree++;
          }
        valid = (valid && (degree==size-1));
          degree=0;
        }
      return valid;
}
```

A.8 eulerian.c

```
#include <stdio.h>
#include <stdlib.h>
#include "graph.h"
#include "../include/pvm3.h"

boolean_t eulerian();         //determines if graph is connected or not


        matrix_t *temp_matrix;      //to store received matrix
             int size;              //size of matrix

main()
{
        matrix_t *current_row,      //pointer to current row
                 *temp_row;         //when creating new (row node)

           row_t *current_node,     //pointer to current node
                 *temp_node;        //when creating new node

             int ptid,              //parent tid
                 msgtag,            //message tag
                 send_buf,          //send buffer
                 rec_buf,           //receive buffer
                 i, j;              //loop control

      ptid = pvm_parent();    //obtaining parent tid for communication
      msgtag=1;               //there is only one copy of task running
      pvm_recv(ptid, msgtag);
      pvm_upkint(&size, 1, 1);      //receiving the size of matirx

                                    //begin to receive matrix

      for (i=0;i<size;i++) {              //outer loop begins here
        temp_row=malloc(sizeof(matrix_t));      //allocate new node
        temp_row->y=NULL;
        temp_row->next_row=NULL;
        if (temp_matrix==NULL) {          //matrix is empty
          temp_matrix=temp_row;
          current_row=temp_row;
        }

        else {                //if matrix is not empty, not first row
          current_row->next_row=temp_row;
          current_row=temp_row;
        }
        current_node=current_row->y;//set current node to the first entry
        for (j=0;j<size;j++) {      //parsing single entries, inner loop
          pvm_recv(ptid, msgtag);         //initialize reveiving
          pvm_upkint(&rec_buf, 1, 1);//receive single entry from parent
          temp_node=malloc(sizeof(row_t));//allocate node for new entry
          temp_node->weight=rec_buf;//assign received value to new node
          temp_node->x=NULL;              //set temp to null for security
          if (current_row->y==NULL)
            current_row->y=temp_node;
          else
            current_node->x=temp_node;
          current_node=temp_node;
        }
        }
```

```
      if (eulerian(temp_matrix))          //set result to send buffer
        send_buf=1;
      else
        send_buf=0;
      pvm_initsend(PvmDataDefault);       //initialize send buffer
      pvm_pkint(&send_buf, 1, 1);
      pvm_send(ptid, msgtag);             //send it back to parent task

      pvm_exit();

}

//determines if graph is eulerian or not
boolean_t eulerian(matrix_t *temp_matrix)
{
      int i, j,               //loop guard
          degree;             //current vertex degree
boolean_t valid=TRUE;         //if it is eulerian or not

      for(i=0;i<size;i++) {
        for(j=0;j<size;j++) {
           if (access_xy(temp_matrix, i, j)!=0)
            degree++;
          }
        valid = (valid && ((degree > 0) && (div(degree,2).rem==0)));
        degree=0;
      }
      return valid;
}
```

A.9 degree_s.c

```
#include <stdio.h>
#include <stdlib.h>
#include "graph.h"
#include "../include/pvm3.h"

void degree_sequence();       //determines if graph is connected or not


        matrix_t *temp_matrix;      //to store received matrix
             int size;              //size of matrix
             row_t *sequence=NULL;  //to store degree sequence

main()
{
        matrix_t *current_row,      //pointer to current row
                 *temp_row;         //when creating new (row node)

           row_t *current_node,     //pointer to current node
                 *temp_node;        //when creating new node

            int ptid,               //parent tid
                msgtag,             //message tag
                send_buf,           //send buffer
                rec_buf,            //receive buffer
                i, j;               //loop control

      ptid = pvm_parent();    //obtaining parent tid for communication
      msgtag=1;               //there is only one copy of task running
      pvm_recv(ptid, msgtag);
      pvm_upkint(&size, 1, 1);      //receiving the size of matirx
```

```
                                    //begin to receive matrix

      for (i=0;i<size;i++) {              //outer loop begins here
        temp_row=malloc(sizeof(matrix_t));//allocate new node
        temp_row->y=NULL;
        temp_row->next_row=NULL;
        if (temp_matrix==NULL) {          //matrix is empty
          temp_matrix=temp_row;
          current_row=temp_row;
        }
        else {                //if matrix is not empty, not first row
          current_row->next_row=temp_row;
          current_row=temp_row;
        }
        current_node=current_row->y;//set current node to the first entry
        for (j=0;j<size;j++) {      //parsing single entries, inner loop
          pvm_recv(ptid, msgtag);         //initialize reveiving
          pvm_upkint(&rec_buf, 1, 1);//receive single entry from parent
          temp_node=malloc(sizeof(row_t));//allocate node for new entry
          temp_node->weight=rec_buf;//assign received value to new node
          temp_node->x=NULL;              //set temp to null for security
          if (current_row->y==NULL)
            current_row->y=temp_node;
          else
            current_node->x=temp_node;
          current_node=temp_node;
        }
        }

      degree_sequence();            //performing analysis

      for(i=0;i<size;i++) {
        pvm_initsend(PvmDataDefault);           //initialize send buffer
        pvm_pkint(&sequence[i].weight, 1, 1);
        pvm_send(ptid, msgtag);           //send it back to parent task
      }

      pvm_exit();
}

//returns the degree sequence of the graph
void degree_sequence()
{
      row_t *current,               //current node
            *temp;                  //for allocating new node
        int i, j,                   //loop guards
            temp_swap;              //for swapping nodes


      for (i=0;i<size;i++) {        //creating sequence
        temp=malloc(sizeof(row_t));
        if (sequence==NULL) {
          sequence=temp;
          current=sequence;
        }
        else {
          current->x=temp;
          current=current->x;
        }
      }
      current->x=NULL;

      for(j=0;j<size;j++) {         //checking vertex degrees
```

```
          for(i=0;i<size;i++) {
          if (access_xy(temp_matrix, i, j)!=0) {
             sequence[i].weight++;
          }
        }
      }

      //simple bubble sort on linked list to create degree sequence
      for(i=0;i<size;i++) {
        for(j=i;j<size;j++) {
          if(sequence[j].weight < sequence[i].weight) {
            temp_swap=sequence[i].weight;
            sequence[i].weight=sequence[j].weight;
            sequence[j].weight=temp_swap;
          }
        }
      }

}
```

A.10 chromatic_i.c

```
#include <stdio.h>
#include <stdlib.h>
#include "graph.h"
#include "../include/pvm3.h"

int chromatic_i();                  //determines chromatic index of graph


        matrix_t *temp_matrix;      //to store received matrix
             int size;              //size of matrix
           row_t *sequence=NULL;    //to store degree sequence

main()
{
        matrix_t *current_row,      //pointer to current row
                 *temp_row;         //when creating new (row node)

             row_t *current_node,   //pointer to current node
                   *temp_node;      //when creating new node


               int ptid,            //parent tid
                   msgtag,          //message tag
                   send_buf,        //send buffer
                   rec_buf,         //receive buffer
                   i, j;            //loop control

      ptid = pvm_parent();    //obtaining parent tid for communication
      msgtag=1;               //there is only one copy of task running
      pvm_recv(ptid, msgtag);
      pvm_upkint(&size, 1, 1);      //receiving the size of matirx

                                    //begin to receive matrix

      for (i=0;i<size;i++) {              //outer loop begins here
        temp_row=malloc(sizeof(matrix_t));      //allocate new node
        temp_row->y=NULL;
        temp_row->next_row=NULL;
        if (temp_matrix==NULL) {          //matrix is empty
          temp_matrix=temp_row;
          current_row=temp_row;
```

```
        }
        else {          //if matrix is not empty, not first row
          current_row->next_row=temp_row;
          current_row=temp_row;
        }
        current_node=current_row->y;//set current node to the first entry
        for (j=0;j<size;j++) {      //parsing single entries, inner loop
          pvm_recv(ptid, msgtag);         //initialize reveiving
          pvm_upkint(&rec_buf, 1, 1);//receive single entry from parent
          temp_node=malloc(sizeof(row_t));//allocate node for new entry
          temp_node->weight=rec_buf;//assign received value to new node
          temp_node->x=NULL;              //set temp to null for security
          if (current_row->y==NULL)
            current_row->y=temp_node;
          else
            current_node->x=temp_node;
          current_node=temp_node;
        }
      }
      send_buf=chromatic_i();             //performing analysis
      pvm_initsend(PvmDataDefault);       //initialize send buffer
      pvm_pkint(&send_buf, 1, 1);
      pvm_send(ptid, msgtag);             //send it back to parent task

      pvm_exit();
}

int chromatic_i()       //determines chromatic index of graph
{
      int i, j,         //loop guard
          degree,       //degree of current vertex
          degree_max;   //largest degree so far
boolean_t loops=FALSE;  //checking for self loops in graph

      for(i=0;i<size;i++) {
        for(j=0;j<size;j++) {
          if (access_xy(temp_matrix, i, j)!=0)
            degree++;
          if (i==j && access_xy(temp_matrix, i, j)!=0)
            loops=TRUE;
        }
        if (degree > degree_max);
          degree_max=degree;
        degree=0;
      }
      if (loops)
        return -1;
      else
          return degree_max;
}
```

A.11 chromatic_n.c

```
#include <stdio.h>
#include <stdlib.h>
#include "graph.h"
#include "../include/pvm3.h"

int chromatic_n();                  //determines chromatic number of
graph


        matrix_t *temp_matrix;      //to store received matrix
             int size;              //size of matrix
             row_t *sequence=NULL;  //to store degree sequence

main()
{
        matrix_t *current_row,      //pointer to current row
                 *temp_row;         //when creating new (row node)

           row_t *current_node,     //pointer to current node
                 *temp_node;        //when creating new node

            int ptid,               //parent tid
                msgtag,             //message tag
                send_buf,           //send buffer
                rec_buf,            //receive buffer
                i, j;               //loop control

      ptid = pvm_parent();    //obtaining parent tid for communication
      msgtag=1;               //there is only one copy of task running
      pvm_recv(ptid, msgtag);
      pvm_upkint(&size, 1, 1);      //receiving the size of matirx

                                    //begin to receive matrix

      for (i=0;i<size;i++) {              //outer loop begins here
        temp_row=malloc(sizeof(matrix_t));      //allocate new node
        temp_row->y=NULL;
        temp_row->next_row=NULL;
        if (temp_matrix==NULL) {          //matrix is empty
          temp_matrix=temp_row;


          current_row=temp_row;
        }
        else {                //if matrix is not empty, not first row
          current_row->next_row=temp_row;
          current_row=temp_row;
        }
        current_node=current_row->y;//set current node to the first entry
        for (j=0;j<size;j++) {      //parsing single entries, inner loop
          pvm_recv(ptid, msgtag);         //initialize reveiving
          pvm_upkint(&rec_buf, 1, 1);//receive single entry from parent
          temp_node=malloc(sizeof(row_t));//allocate node for new entry
          temp_node->weight=rec_buf;//assign received value to new node
          temp_node->x=NULL;              //set temp to null for security
          if (current_row->y==NULL)
            current_row->y=temp_node;
          else
            current_node->x=temp_node;
          current_node=temp_node;
```

```
        }
      }
      send_buf=chromatic_n();             //performing analysis
      pvm_initsend(PvmDataDefault);       //initialize send buffer
      pvm_pkint(&send_buf, 1, 1);
      pvm_send(ptid, msgtag);             //send it back to parent task

      pvm_exit();
}

int chromatic_n()
{
      int i, j,               //loop guard
          degree,             //degree of current vertex
          degree_max;         //largest degree so far
boolean_t loops=FALSE;        //checking for self loops in graph

      for(i=0;i<size;i++) {
        for(j=0;j<size;j++) {
          if (access_xy(temp_matrix, i, j)!=0)
            degree++;
          if (i==j && access_xy(temp_matrix, i, j)!=0)
            loops=TRUE;
        }
        if (degree > degree_max);
          degree_max=degree;
        degree=0;
      }
      if (loops)
        return -1;
      else
          return degree_max;
}
```

A.12 degree_s.c

```
#include <stdio.h>
#include <stdlib.h>
#include "graph.h"
#include "../include/pvm3.h"

void degree_sequence();             //determines if graph is connected or
not


        matrix_t *temp_matrix;      //to store received matrix
             int size;        //size of matrix
             row_t *sequence=NULL;  //to store degree sequence
main()
{
        matrix_t *current_row,      //pointer to current row
                 *temp_row;         //when creating new (row node)

             row_t *current_node,   //pointer to current node
                 *temp_node;        //when creating new node


               int ptid,            //parent tid
                 msgtag,            //message tag
               send_buf,            //send buffer
                 rec_buf,           //receive buffer
               i, j;          //loop control
```

```
      ptid = pvm_parent();               //obtaining parent tid for
communication
      msgtag=1;                     //there is only one copy of task
running
      pvm_recv(ptid, msgtag);
      pvm_upkint(&size, 1, 1);           //receiving the size of matirx

                                    //begin to receive matrix

      for (i=0;i<size;i++) {             //outer loop begins here
        temp_row=malloc(sizeof(matrix_t));      //allocate new node
        temp_row->y=NULL;
        temp_row->next_row=NULL;
        if (temp_matrix==NULL) {         //matrix is empty
          temp_matrix=temp_row;
          current_row=temp_row;
        }
        else {                      //if matrix is not empty, not first
row
          current_row->next_row=temp_row;
          current_row=temp_row;
        }
        current_node=current_row->y;           //set current node to the
first entry of row
        for (j=0;j<size;j++) {           //parsing single entries, inner
loop
          pvm_recv(ptid, msgtag);        //initialize reveiving
          pvm_upkint(&rec_buf, 1, 1);          //receive single entry
from parent
          temp_node=malloc(sizeof(row_t));     //allocate node for new
entry
          temp_node->weight=rec_buf;           //assign received value
to new node
          temp_node->x=NULL;             //set temp to null for security
          if (current_row->y==NULL)
            current_row->y=temp_node;
          else
            current_node->x=temp_node;
          current_node=temp_node;
        }
        }

      degree_sequence();            //performing analysis

      for(i=0;i<size;i++) {
        pvm_initsend(PvmDataDefault);          //initialize send buffer
        pvm_pkint(&sequence[i].weight, 1, 1);
        pvm_send(ptid, msgtag);          //send it back to parent task
        }

      pvm_exit();

}

//returns the degree sequence of the graph
void degree_sequence()
{
      row_t *current,               //current node
            *temp;                  //for allocating new node
        int i, j,                   //loop guards
            temp_swap;              //for swapping nodes
```

```
      for (i=0;i<size;i++) {        //creating sequence
        temp=malloc(sizeof(row_t));
        if (sequence==NULL) {
          sequence=temp;
          current=sequence;
          }
        else {
          current->x=temp;
          current=current->x;
          }
      }
      current->x=NULL;

      for(j=0;j<size;j++) {         //checking vertex degrees
          for(i=0;i<size;i++) {
          if (access_xy(temp_matrix, i, j)!=0) {
             sequence[i].weight++;
          }
        }
      }

      //simple bubble sort on linked list to create degree sequence
      for(i=0;i<size;i++) {
        for(j=i;j<size;j++) {
          if(sequence[j].weight < sequence[i].weight) {
            temp_swap=sequence[i].weight;
            sequence[i].weight=sequence[j].weight;
            sequence[j].weight=temp_swap;
          }
        }
      }
}
```

A.13 Example of a graph file

```
graph_name = just_a_graph

1 0 0 4 3
5 4 0 2 3
8 7 6 0 3
1 0 0 0 3
9 6 2 5 3
```

Appendix B System calls in PVM

NAME

pvm_mytid - Returns the tid of the calling process.

SYNOPSIS

C int tid = pvm_mytid(void)

Fortran call pvmfmytid(tid)

PARAMETERS

tid Integer returning the task identifier of the calling PVM process. Values less than zero indicate an error.

DESCRIPTION

The routine pvm_mytid enrolls this process into PVM on its first call. It also generates a unique tid if this process was not created by pvm_spawn. pvm_mytid returns the tid of the calling process and can be called multiple times in an application.

Any PVM system call (not just pvm_mytid) will enroll a task in PVM if the task is not enrolled before the call.

The tid is a 32 bit positive integer created by the local pvmd. The 32 bits are divided into fields that encode various information about this process such as its location in the virtual machine (i.e. local pvmd address), the CPU number in the case where the process is on a multiprocessor, and a process ID field. This information is used by PVM and is not expected to be used by applications. Applications should not attempt to predict or interpret the tid with the exception of calling tidtohost()
If PVM has not been started before an application calls pvm_mytid the returned tid will be < 0.

NAME

pvm_spawn - Starts new PVM processes.

SYNOPSIS

C int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)

PARAMETERS

task Character string which is the executable file name of the PVM process to be started. The executable must already reside on the host on which it is to be started. The name may be a file in the PVM search path or an absolute path. The default PVM search path is $HOME/pvm3/bin/$PVM_ARCH/ .

argv Pointer to an array of arguments to the executable (if supported on the target machine), not including the executable name, with the end of the array specified by NULL. argv[0] of the spawned task is set to the executable path relative to the PVM working directory (or absolute if an absolute filename was specified). If the executable needs no arguments, then the second argument to pvm_spawn is NULL.

flag Integer specifying spawn options.

In C, flag should be the sum of:

Option value		MEANING
PvmTaskDefault	0	PVM can choose any machine to start task
PvmTaskHost	1	where specifies a particular host
PvmTaskArch	2	where specifies a type of architecture
PvmTaskDebug	4	Start up processes under debugger
PvmTaskTrace	8	Processes will generate PVM trace data. *
PvmMppFront	16	Start process on MPP front-end.
PvmHostCompl	32	Use complement host set

where Character string specifying where to start the PVM process. Depending on the value of flag, where can be a host name such as ``ibm1.epm.ornl.gov'' or a PVM architecture class such as ``SUN4''. Also, the host name "." is taken as the localhost. If flag is 0, then where is ignored and PVM will select the most appropriate host.

ntask Integer specifying the number of copies of the executable to start.

tids Integer array of length ntask returning the tids of the PVM processes started by this pvm_spawn call.

numt Integer returning the actual number of tasks started. Values less than zero indicate a system error. A positive value less than ntask indicates a partial failure. In this case the user should check the tids array for the error code(s).

DESCRIPTION

The routine pvm_spawn starts ntask copies of the executable named task. On systems that support environment, spawn passes selected variables from parent environment to children tasks. If set, the envar PVM_EXPORT is passed. If PVM_EXPORT contains other names (separated by ':') they will be passed too.

This is useful for e.g.:

```
setenv DISPLAY myworkstation:0.0
setenv MYSTERYVAR 13
setenv PVM_EXPORT DISPLAY:MYSTERYVAR
```

The hosts on which the PVM processes are started are determined by the flag and where arguments. On return the array tids contains the PVM task identifiers for each process started.

If pvm_spawn starts one or more tasks, nuof tasks started. If a system error occurs then numt will be < 0. If numt is less than ntask then some executables have failed to start and the user should check the last ntask - numt locations in the tids array which will contain error codes (see below for meaning). The first numt tids in the array are always valid. When flag is set to 0 and where is set to NULL (or ``*'' in Fortran) a heuristic (round-robin assignment) is used to distribute the ntask processes across the virtual machine.

If the PvmHostCompl flag is set, the resulting host set gets comple mented. Given that the TaskHost host name "." is taken as localhost, these can be used together, for example, to request n - 1 tasks on host "." but with flags TaskHost|HostCompl to spawn n - 1 tasks on hosts other than the localhost.

In the special case where a multiprocessor is specified by where, pvm_spawn will start all ntask copies on this single machine using the vendor's underlying routines.

The spawned task will have argv[0] set to the the executable path relative to its inherited working directory (or possibly an absolute path), so the base filename can be got by using:

```
char *p;
p = (p = rindex(argv[0], '/')) ? p + 1 : argv[0];
```

If PvmTaskDebug is set, then the pvmd will start the task(s) under debugger(s). In this case, instead of executing pvm3/bin/ARCH/task args it executes pvm3/lib/debugger pvm3/bin/ARCH/task args. debugger is a shell script that the users can modify to their individual tastes. Presently the script starts an xterm with dbx or comparable debugger inmt will be the actual number ERRORS These error conditions can be returned by pvm_spawn either in numt or in the tids array.

PvmBadParam
giving an invalid argument value.

PvmNoHost
Specified host is not in the virtual machine.

PvmNoFile
Specified executable cannot be found. The default location PVM looks in is ~/pvm3/bin/ARCH, where ARCH is a PVM architecture name.

PvmNoMem
Malloc failed. Not enough memory on host.

PvmSysErr
pvmd not responding.

PvmOutOfRes
out of resources.

NAME

pvm_initsend - Clear default send buffer and specify message encoding.

SYNOPSIS

C int bufid = pvm_initsend(int encoding)

Fortran call pvmfinitsend(encoding, bufid)

PARAMETERS

encoding

Integer specifying the next message's encoding scheme. Options in C are:

Encoding value		MEANING
PvmDataDefault	0	XDR
PvmDataRaw	1	no encoding
PvmDataInPlace	2	data left in place

Option names are in Fortran are:

Encoding value		MEANING
PVMDEFAULT	0	XDR
PVMRAW	1	no encoding
PVMINPLACE	2	data left in place

bufid Integer returned containing the message buffer identifier. Values less than zero indicate an error.

DESCRIPTION

The routine pvm_initsend clears the send buffer and prepares it for packing a new message. The encoding scheme used for the packing is set by encoding. XDR encoding is used by default because PVM can not know if the user is going to add a heterogeneous machine before this message is sent. If the user knows that the next message will only be sent to a machine that understands the native format, then he can use Pvm-DataRaw encoding and save on encoding costs.

PvmDataInPlace encoding specifies that data be left in place during packing. The message buffer only contains the sizes and pointers to the items to be sent. When pvm_send is called the items are copied directly out of the user's memory. This option decreases the number of times a message is copied at the expense of requiring the user to not modify the items between the time they are packed and the time they are sent.

If pvm_initsend is successful, then bufid will contain the message buffer identifier. If some error occurs then bufid will be < 0.

RESTRICTIONS

PvmDataInPlace allows only dense (stride = 1) data in version 3.3. It cannot be used on shared memory (*MP) architectures; a PvmNotImpl error will occur at send time.

EXAMPLES

C:

```
bufid = pvm_initsend( PvmDataDefault );
info = pvm_pkint( array, 10, 1 );
msgtag = 3 ;
info = pvm_send( tid, msgtag );
```

ERRORS

These error conditions can be returned by pvm_initsend

PvmBadParam
giving an invalid encoding value

PvmNoMem
Malloc has failed. There is not enough memory to create the Buffer

NAME

pvm_recv - Receive a message.

SYNOPSIS

C int bufid = pvm_recv(int tid, int msgtag)

Fortran call pvmfrecv(tid, msgtag, bufid)

PARAMETERS

tid Integer task identifier of sending process supplied by the user.

msgtag Integer message tag supplied by the user. msgtag should be >= 0.

bufid Integer returns the value of the new active receive buffer identifier. Values less than zero indicate an error.

DESCRIPTION

The routine pvm_recv blocks the process until a message with label msgtag has arrived from tid. pvm_recv then places the message in a new active receive buffer, which also clears the current receive buffer.

A -1 in msgtag or tid matches anything. This allows the user the following options. If tid = -1 and msgtag is defined by the user, then pvm_recv will accept a message from any process which has a matching msgtag. If msgtag = -1 and tid is defined by the user, then pvm_recv will accept any message that is sent from process tid. If tid = -1 and msgtag = -1, then pvm_recv will accept any message from any process.

The PVM model guarantees the following about message order. If task 1 sends message A to task 2, then task 1 sends message B to task 2, message A will arrive at task 2 before message B. Moreover, if both messages arrive before task 2 does a receive, then a wildcard receive will always return message A.

If pvm_recv is successful, bufid will be the value of the new active receive buffer identifier. If some error occurs then bufid will be < 0.

pvm_recv is blocking which means the routine waits until a message matching the user specified tid and msgtag values arrives at the local pvmd. If the message has already arrived then pvm_recv returns immediately with the message.

Once pvm_recv returns, the data in the message can be unpacked into the user's memory using the unpack routines.

EXAMPLES

C:

```
tid = pvm_parent();
msgtag = 4 ;
bufid = pvm_recv( tid, msgtag );
info = pvm_upkint( tid_array, 10, 1 );
info = pvm_upkint( problem_size, 1, 1 );
info = pvm_upkfloat( input_array, 100, 1 );
```

ERRORS

These error conditions can be returned by pvm_recv

PvmBadParam
: giving an invalid tid value, or msgtag < -1.

PvmSysErr
: pvmd not responding.

NAME

pvm_unpack Unpack the active message buffer into arrays of prescribed data type.

SYNOPSIS

C

```
int info = pvm_unpackf( const char *fmt, ... )
int info = pvm_upkbyte( char *xp, int nitem, int stride)
int info = pvm_upkcplx( float *cp, int nitem, int stride)
int info = pvm_upkdcplx( double *zp, int nitem, int stride)
int info = pvm_upkdouble( double *dp, int nitem, int stride)
int info = pvm_upkfloat( float *fp, int nitem, int stride)

int info = pvm_upkint( int *ip, int nitem, int stride)
int info = pvm_upkuint( unsigned int *ip, int nitem, int stride )
int info = pvm_upkushort( unsigned short *ip, int nitem, int stride )
int info = pvm_upkulong( unsigned long *ip, int nitem, int stride )
int info = pvm_upklong( long *ip, int nitem, int stride)
int info = pvm_upkshort( short *jp, int nitem, int stride)
int info = pvm_upkstr( char *sp )
```

PARAMETERS

fmt
Printf-like format expression specifying what to pack. (See discussion)

nitem
The total number of items to be unpacked (not the number of bytes).

stride
The stride to be used when packing the items. For example, if stride = 2 in pvm_upkcplx, then every other complex number will be unpacked.

xp
Pointer to the beginning of a block of bytes. Can be any data type, but must match the corresponding pack data type.

cp
Complex array at least nitem*stride items long.

zp
Double precision complex array at least nitem*stride items long.

dp
Double precision real array at least nitem*stride items long.

fp
Real array at least nitem*stride items long.

ip
Integer array at least nitem*stride items long.

jp
Integer*2 array at least nitem*stride items long.

sp
Pointer to a null terminated character string.

what
Integer specifying the type of data being unpacked.

what options

STRING	0	REAL4	4
BYTE1	1	COMPLEX8	5
INTEGER2	2	REAL8	6
INTEGER4	3	COMPLEX16	7

info
Integer status code returned by the routine. Values less than zero indicate an error.

DESCRIPTION

Each of the pvm_upk* routines unpacks an array of the given data type from the active receive buffer. The arguments for each of the routines are a pointer to the array to be unpacked into, nitem which is the total number of items to unpack, and stride which is the stride to use when unpacking.

An exception is pvm_upkstr() which by definition unpacks a NULL terminated character string and thus does not need nitem or stride arguments. The Fortran routine pvmfunpack(STRING, ...) expects nitme to be the number of characters in the string and stride to be 1.

If the unpacking is successful, info will be 0. If some error occurs then info will be < 0.
A single variable (not an array) can be unpacked by setting nitem = 1 and stride = 1.
The routine pvm_unpackf() uses a printf-like format expression to specify what and how to unpack data from the receive buffer. All variables are passed as addresses. A BNF-like description of the format syntax is:

```
format : null | init | format fmt
init : null | '%' '+'
fmt : '%' count stride modifiers fchar
fchar : 'c' | 'd' | 'f' | 'x' | 's'
count : null | [0-9]+ | '*'
stride : null | '.' ( [0-9]+ | '*' )
modifiers : null | modifiers mchar
mchar : 'h' | 'l' | 'u'
```

Modifiers:

- h short (int)
- l long (int, float, complex float)
- u unsigned (int)

Future extensions to the what argument in pvmfunpack will include 64 bit types when XDR encoding of these types is available. Meanwhile users should be aware that precision can be lost when passing data from a 64 bit machine like a Cray to a 32 bit machine like a SPARCstation. As a mnemonic the what argument name includes the number of bytes of precision to expect. By setting encoding to PVMRAW (see pvmfinitsend) data can be transferred between two 64 bit machines with full precision even if the PVM configuration is heterogeneous.

Messages should be unpacked exactly like they were packed to insure data integrity.
Packing integers and unpacking them as floats will often fail because a type encoding will have occurred transferring the data between heterogeneous hosts. Packing 10 integers and 100 floats then trying to unpack only 3 integers and the 100 floats will also fail.

EXAMPLES

C:

```
info = pvm_recv( tid, msgtag );
info = pvm_upkstr( string );
info = pvm_upkint( &size, 1, 1 );
info = pvm_upkint( array, size, 1 );
info = pvm_upkdouble( matrix, size*size, 1 );
```

ERRORS

PvmNoData: Reading beyond the end of the receive buffer. Most likely cause is trying to unpack more items than were originally packed into the buffer.

PvmBadMsg: The received message can not be decoded. Most likely because the hosts are heterogeneous and the user specified an incompatible encoding. Try setting the encoding to PvmDataDefault (see pvm_mkbuf).

PvmNoBuf: There is no active receive buffer to unpack.

NAME

pvm_exit - Tells the local pvmd that this process is leaving PVM.

SYNOPSIS

C int info = pvm_exit(void)

Fortran call pvmfexit(info)

PARAMETERS

info Integer status code returned by the routine. Values less than zero indicate an error.

DESCRIPTION

The routine pvm_exit tells the local pvmd that this process is leaving PVM. This routine does not kill the process, which can continue to perform tasks just like any other serial process.

pvm_exit should be called by all PVM processes before they stop or exit for good. It must be called by processes that were not started with pvm_spawn.

EXAMPLES

C:

```
/* Program done */
pvm_exit();
exit();
```

ERRORS

PvmSysErr

pvmd not responding

NAME

pvm_send - Immediately sends the data in the active message buffer.

SYNOPSIS

C int info = pvm_send(int tid, int msgtag)

Fortran call pvmfsend(tid, msgtag, info)

PARAMETERS

tid Integer task identifier of destination process.

msgtag Integer message tag supplied by the user. msgtag should be >=0.

info Integer status code returned by the routine.

DESCRIPTION

The routine pvm_send sends a message stored in the active send buffer to the PVM process identified by tid. msgtag is used to label the content of the message. If pvm_send is successful, info will be 0. If some error occurs then info will be < 0.

The pvm_send routine is asynchronous. Computation on the sending processor resumes as soon as the message is safely on its way to the receiving processor. This is in contrast to synchronous communication, during which computation on the sending processor halts until the matching receive is executed by the receiving processor.
The PVM model guarantees the following about message order.
If task 1 sends message A to task 2, then task 1 sends message B to task 2, message A will arrive at task 2 before message B. Moreover, if both messages arrive before task 2 does a receive, then a wildcard receive will always return message A.

Terminating a PVM task immediately after sending a message or messages from it may result in those messages being lost. To be sure, always call pvm_exit() before stopping.

EXAMPLES

C:

```
info = pvm_initsend( PvmDataDefault );
info = pvm_pkint( array, 10, 1 );
msgtag = 3 ;
info = pvm_send( tid, msgtag );
```

ERRORS

These error conditions can be returned by pvm_send

PvmBadParam
: giving an invalid tid or a msgtag.

PvmSysErr
: pvmd not responding.

PvmNoBuf
: no active send buffer. Try pvm_initsend() before send.

NAME

pvm_pack Pack the active message buffer with arrays of prescribed data type.

SYNOPSIS

C

```
int info = pvm_packf( const char *fmt, ... )
int info = pvm_pkbyte( char *xp, int nitem, int stride )
int info = pvm_pkcplx( float *cp, int nitem, int stride )
int info = pvm_pkdcplx( double *zp, int nitem, int stride )
int info = pvm_pkdouble( double *dp, int nitem, int stride )
int info = pvm_pkfloat( float *fp, int nitem, int stride )

int info = pvm_pkint( int *ip, int nitem, int stride )
int info = pvm_pkuint( unsigned int *ip, int nitem, int stride )
int info = pvm_pkushort( unsigned short *ip, int nitem, int stride )
int info = pvm_pkulong( unsigned long *ip, int nitem, int stride )
int info = pvm_pklong( long *ip, int nitem, int stride )
int info = pvm_pkshort( short *jp, int nitem, int stride )
int info = pvm_pkstr( char *sp )
```

PARAMETERS

fmt
Printf-like format expression specifying what to pack. (See discussion).

nitem
The total number of items to be packed (not the number of bytes).

stride
The stride to be used when packing the items. For example, if stride = 2 in pvm_pkcplx, then every other complex number will be packed.

xp
Pointer to the beginning of a block of bytes. Can be any data type, but must match the corresponding unpack data type.

cp
Complex array at least nitem*stride items long.

NAME

pvm_addhosts - Add hosts to the virtual machine.

SYNOPSIS

C int info = pvm_addhosts(char **hosts, int nhost, int *infos)

Fortran call pvmfaddhost(host, info)

PARAMETERS

hosts An array of strings naming the hosts to be added. Pvmd must already be installed and the user must have an account on the specified hosts.

nhost Integer specifying the length of array hosts.

infos Integer array of length nhost which returns the status for each host. Values less than zero indicate an error, while positive values are TIDs of the new hosts.

host Character string naming the host to be added.

info Integer status code returned by the routine. Values less than nhost indicate partial failure, and values less than 1 indicate total failure.

DESCRIPTION

The routine pvm_addhosts adds the computers named in hosts to the configuration of computers making up the virtual machine. The names should have the same syntax as lines of a pvmd hostfile (see man page for pvmd3): A hostname followed by options of the form xx=y.

If pvm_addhosts is successful, info will be equal to nhost. Partial success is indicated by 0 < info < nhost, and total failure by info < 1. The array infos can be checked to determine which host caused the error.

The Fortran routine pvmfaddhost adds a single host to the configuration with each call. info will be 1 if successful or < 0 if error.

The status of hosts can be requested by the application using pvm_mstat and pvm_config. If a host fails it will be automatically deleted from the configuration. Using pvm_addhosts a replacement host can be added by the application, however it is the responsibility of the application developer to make his application tolerant of host failure. Another use of this feature would be to add more hosts as they become available, for example on a weekend, or if the application dynamically determines it could use more computational power.

EXAMPLES

C:

```
static char *hosts[] = {
   "sparky",
   "thud.cs.utk.edu",
};
info = pvm_addhosts( hosts, 2, infos );
```

ERRORS

The following error conditions can be returned by pvm_addhosts:

PvmBadParam
: giving an invalid argument value.

PvmAlready
: another task is currently adding hosts.

PvmSysErr
: local pvmd is not responding.

and in the infos vector:

PvmBadParam
: bad hostname syntax.

PvmNoHost
no such host.

PvmCantStart
failed to start pvmd on host.

PvmDupHost
host already configured.

PvmBadVersion
pvmd protocol versions don't match.

PvmOutOfRes
PVM has run out of system resources.

NAME

pvm_delhosts - Deletes hosts from the virtual machine.

SYNOPSIS

C int info = pvm_delhosts(char **hosts, int nhost, int *infos)

Fortran call pvmfdelhost(host, info)

PARAMETERS

hosts An array of pointers to character strings containing the names of the machines to be deleted.

nhost Integer specifying the number of hosts to be deleted.

infos Integer array of length nhost which contains the status code returned by the routine for the individual hosts. Values less than zero indicate an error.

host Character string containing the name of the machine to be deleted.

info Integer status code returned by the routine. Values less than nhost indicate partial failure, values less than 1 indicate total failure.

DESCRIPTION

The routine pvm_delhosts deletes the computers pointed to in hosts from the existing configuration of computers making up the virtual machine. All PVM processes and the pvmd running on these computers are killed as the computer is deleted. If pvm_delhosts is successful, info will be nhost. Partial success is indicated by 1<= info < nhost, and total failure by info < 1. The array infos can be checked to determine which host caused the error.

The Fortran routine pvmfdelhost deletes a single host from the configuration with each call. If a host fails, the PVM system will continue to function and will automatically delete this host from the virtual machine. An application can be notified of a host failure by calling pvm_notify. It is still the responsibility of the application developer to make his application tolerant of host failure.

EXAMPLES

C:

```
static char *hosts[] = {
    "sparky",
    "thud.cs.utk.edu",
};
int status[2];
info = pvm_delhosts( hosts, 2, status );
```

Fortran:

```
CALL PVMFDELHOST( 'azure', INFO )
```

ERRORS

These error conditions can be returned by pvm_delhosts

PvmBadParam
: giving an invalid argument value.

PvmSysErr
: local pvmd not responding.

NAME

pvm_kill - Terminates a specified PVM process.

SYNOPSIS

C `int info = pvm_kill( int tid )`

Fortran `call pvmfkill( tid, info )`

PARAMETERS

tid Integer task identifier of the PVM process to be killed (not yourself).

info Integer status code returned by the routine. Values less than zero indicate an error.

DESCRIPTION

The routine pvm_kill sends a terminate (SIGTERM) signal to the PVM process identified by tid. In the case of multiprocessors the terminate signal is replaced with a host dependent method for killing a process. If pvm_kill is successful, info will be 0. If some error occurs then info will be < 0.

pvm_kill is not designed to kill the calling process. To kill yourself in C call pvm_exit() followed by exit(). To kill yourself in Fortran call pvmfexit followed by stop.

EXAMPLES

C:

```
info = pvm_kill( tid );
```

ERRORS

These error conditions can be returned by pvm_kill

PvmBadParam
: giving an invalid tid value.

PvmSysErr
: pvmd not responding.

Appendix C Explanation of used terms

ALU : Arithmetic and Logic Unit

<processor> (ALU or "mill") The part of the central processing unit which performs operations such as addition, subtraction and multiplication of integers and bit-wise AND, OR, NOT, XOR and other Boolean operations. The CPU's instruction decode logic determines which particular operation the ALU should perform, the source of the operands and the destination of the result.
The width in bits of the words which the ALU handles is usually the same as that quoted for the processor as a whole whereas its external busses may be narrower. Floating-point operations are usually done by a separate "floating-point unit". Some processors use the ALU for address calculations (e.g. incrementing the program counter), others have separate logic for this.

ANSI : American National Standards Institute

<body, standard> (ANSI) The United States government body responsible for approving US standards in many areas, including computers and communications. ANSI is a member of ISO. ANSI sells ANSI and ISO (international) standards.

B : The B language

<language> A systems language written by Ken Thompson in 1970 mostly for his own use under Unix on the PDP-11. B was later improved by Kerninghan(?) and Ritchie to produce C. B was used as the systems language on Honeywell's GCOS-3.
B was, according to Ken, greatly influenced by BCPL, but the name B had nothing to do with BCPL. B was in fact a revision of an earlier language, bon, named after Ken Thompson's wife, Bonnie.

BSD : Berkeley System Distribution, a Unix variant.

<operating system> (BSD) A family of Unix versions for the DEC VAX and PDP-11, developed by Bill Joy and others at the University of California at Berkeley. BSD Unix incorporates paged virtual memory, TCP/IP networking enhancements, and many other features.

CISC : Complex Instruction Set Computer

CISC) A processor where each instruction can perform several low-level operations such as memory access, arithmetic operations or address calculations. The term was coined in contrast to Reduced Instruction Set Computer.

Before the first RISC processors were designed, many computer architects were trying to bridge the "semantic gap" - to design instruction sets to support high-level languages by providing "high-level" instructions such as procedure call and return, loop instructions such as "decrement and branch if non-zero" and complex addressing modes to allow data structure and array accesses to be compiled into single instructions.
While these architectures achieved their aim of allowing high-level language constructs to be expressed in fewer instructions, it was observed that they did not always result in improved performance. For example, on one processor it was discovered that it was possible to improve the performance by NOT using the procedure call instruction but using a sequence of simpler instructions instead. Furthermore, the more complex the instruction set, the greater the overhead of decoding an instruction, both in execution time and silicon area. This is particularly true for processors which used microcode to decode the (macro) instruction. It is easier to debug a complex instruction set implemented in microcode than one whose decoding is "hard-wired" in silicon.
Examples of CISC processors are the Motorola 680x0 family and the Intel 80186 through Intel 486 and Pentium.

CPU : Central Processing Unit
<architecture, processor> (CPU, processor) The part of a computer which controls all the other parts. Designs vary widely but, in general, the CPU consists of the control unit, the arithmetic and logic unit (ALU) and memory (registers, cache, RAM and ROM) as well as various temporary buffers and other logic.

DSL : Digital Subscriber Line
<communications, protocol> (DSL, or Digital Subscriber Loop, xDSL - see below) A family of digital telecommunications protocols designed to allow high speed data communication over the existing copper telephone lines between end-users and telephone companies.
When two conventional modems are connected through the telephone system (PSTN), it treats the communication the same as voice conversations. This has the advantage that there is no investment required from the telephone company (telco) but the disadvantage is that the bandwidth available for the communication is the same as that available for voice conversations, usually 64 kb/s (DS0) at most. The twisted-pair copper cables into individual homes or offices can usually carry significantly more than 64 kb/s but the telco needs to handle the signal as digital rather than analog.

There are many implementation of the basic scheme, differing in the communication protocol used and providing varying service levels. The throughput of the communication can be anything from about 128 kb/s to over 8 Mb/s, the communication can be either symmetric or asymmetric (i.e. the available bandwidth may or may not be the same upstream and downstream). Equipment prices and service fees also vary considerably.
The first technology based on DSL was ISDN, although ISDN is not often recognised as such nowadays. Since then a large number of other protocols have been developed, collectively referred to as xDSL, including HDSL, SDSL, ADSL, and VDSL. As yet none of these have reached very wide deployment but wider deployment is expected for 1998-1999.

EISA: Extended Industry-Standard Architecture
<architecture, standard> (EISA) /eesa/ A bus standard for IBM compatibles that extends the ISA bus architecture to 32 bits and allows more than one CPU to share the bus. The bus mastering support is also enhanced to provide access to 4 GB of memory. Unlike MCA, EISA can accept older XT bus architecture and ISA boards.

FPU : Floating Point Unit

(FPU) A floating-point accelerator, usually in a single integrated circuit, possible on the same IC as the central processing unit.

Gambit :
<language> A variant of Scheme R3.99 supporting the future construct of Multilisp by Marc Feeley <feeley@iro.umontreal.ca>. Implementation includes optimising compilers for Macintosh (with Toolbox and built-in editor) and Motorola 680x0 Unix systems and HP300, BBN GP100 and NeXT. Version 2.0 conforms to the IEEE Scheme standard. Gambit used PVM as its intermediate language.

HPCC : High Performance Computing and Communications

(HPCC) High performance computing includes scientific workstations, supercomputer systems, high speed networks, special purpose and experimental systems, the new generation of large scale parallel systems, and application and systems software with all components well integrated and linked over a high speed network.

ISA : Industry Standard Architecture
<architecture, standard> (ISA) A bus standard for IBM compatibles that extends the XT bus architecture to 16 bits. It also allows for bus mastering although only the first 16 MB of main memory is available for direct access. In reference to the XT bus architecture it is sometimes referred to as "AT bus architecture".

ISO : International Organization for Standardization

<standard, body> (ISO) A voluntary, nontreaty organisation founded in 1946, responsible for creating international standards in many areas, including computers and communications. Its members are the national standards organisations of 89 countries, including the American National Standards Institute. ISO produced the OSI seven layer model for network architecture. The term "ISO" is not actually an acronym for anything. It is a pun on the Greek prefix "iso-", meaning "same". Some ISO documents say ISO is not an acronym even though it is an anagram of the initials of the organisation's name.

NIS : Network Information Service

<networking, protocol> (NIS) Sun Microsystems' Yellow Pages (yp) client-server protocol for distributing system configuration data such as user and host names between computers on a network. Sun licenses the technology to virtually all other Unix vendors. The name "Yellow Pages" is a registered trademark in the United Kingdom of British Telecommunications plc for their (paper) commercial telephone directory. Sun changed the name of their system to NIS, though all the commands and functions still start with "yp", e.g. ypcat, ypmatch, ypwhich.

NFS : Network File System

<networking, operating system> (NFS) A protocol developed by Sun Microsystems, and defined in RFC 1094, which allows a computer to access files over a network as if they were on its local disks. This protocol has been incorporated in products by more than two hundred companies, and is now a de facto standard. NFS is implemented using a connectionless protocol (UDP) in order to make it stateless.

OS : Operating system
<operating system> (OS) The low-level software which handles the interface to peripheral hardware, schedules tasks, allocates storage, and presents a default interface to the user when no application program is running.

PCI : Peripheral Component Interconnect, an interface.
<hardware> (PCI) A standard for connecting peripherals to a personal computer, designed by Intel and released around Autumn 1993. PCI is supported by most major manufacturers including Apple Computer. It is technically far superior to VESA's local bus. It runs at 20 - 33 MHz and carries 32 bits at a time over a 124-pin connector or 64 bits over a 188-pin connector. An address is sent in one cycle followed by one word of data (or several in burst mode).
PCI is used in systems based on Pentium, Pentium Pro, AMD 5x86, AMD K5 and AMD K6 processors, in some DEC Alpha and PowerPC systems, and probably Cyrix 586 and Cyrix 686 systems. However, it is processor independent and so can work with other processor architectures as well.
Technically, PCI is not a bus but a bridge or mezzanine. It includes buffers to decouple the CPU from relatively slow peripherals and allow them to operate asynchronously.

PPP : Point-to-Point Protocol

<communications, protocol> (PPP) The protocol defined in RFC 1661, the Internet standard for transmitting network layer datagrams (e.g. IP packets) over serial point-to-point links. PPP has a number of advantages over SLIP; it is designed to operate both over asynchronous connections and bit-oriented synchronous systems, it can configure connections to a remote network dynamically, and test that the link is usable.
PPP can be configured to encapsulate different network layer protocols (such as IP, IPX, or AppleTalk) by using the appropriate Network Control Protocol (NCP).

RAM : Random Access Memory
<storage> (RAM) (Previously "direct-access memory"). A data storage device for which the order of access to different locations does not affect the speed of access. This is in contrast to, say, a magnetic disk, magnetic tape or a mercury delay line where it is very much quicker to access data sequentially because accessing a non-sequential location requires physical movement of the storage medium rather than just electronic switching.
The most common form of RAM in use today is built from semiconductor integrated circuits, which can be either static (SRAM) or dynamic (DRAM). In the 1970s magnetic core memory was used. RAM is still referred to as core by some old-timers. The term "RAM" has gained the additional meaning of read-write.
Most kinds of semiconductor read-only memory (ROM) are actually "random access" in the above sense but are never referred to as RAM. Furthermore, memory referred to as RAM can usually be read and written equally quickly (approximately), in contrast to the various kinds of programmable read-only memory.

Finally, RAM is usually volatile though non-volatile random-access memory is also used. Interestingly, some DRAM devices are not truly random access because various kinds of "page mode" or "column mode" mean that sequential access is faster than random access.

RISC : Reduced Instruction Set Computer
<processor> (RISC) A processor whose design is based on the rapid execution of a sequence of simple instructions rather than on the provision of a large variety of complex instructions (as in a Complex Instruction Set Computer). Features which are generally found in RISC designs are uniform instruction encoding (e.g. the op-code is always in the same bit positions in each instruction which is always one word long), which allows faster decoding; a homogenous register set, allowing any register to be used in any context and simplifying compiler design; and simple addressing modes with more complex modes replaced by sequences of simple arithmetic instructions.
Examples of (more or less) RISC processors are the Berkeley RISC, HP-PA, Clipper, i960, AMD 29000, MIPS R2000 and DEC Alpha. IBM's first RISC computer was the RT/PC (IBM 801), they now produce the RISC-based RISC System/6000 and SP/2 lines. Despite Apple Computer's bogus claims for their PowerPC-based Macintoshes, the first RISC processor used in a personal computer was the Advanced RISC Machine (ARM) used in the Acorn Archimedes.

Scheme :
(Originally "Schemer", by analogy with Planner and Conniver). A small, uniform Lisp dialect with clean semantics, developed initially by Guy Steele and Gerald Sussman in 1975. Scheme uses applicative order reduction and is lexically scoped. It treats both functions and continuations as first-class objects.
One of the most used implementations is DrScheme, others include include Bigloo, Elk, Liar, Orbit, Scheme86 (Indiana U), SCM, MacScheme (Semantic Microsystems), PC Scheme (TI), MIT Scheme, and T. See also Kamin's interpreters, PSD, PseudoScheme, Schematik, Scheme Repository, STk, syntax-case, Tiny Clos, Paradigms of AI Programming.
There have been a series of revisions of the report defining Scheme, known as RRS (Revised Report on Scheme), R2RS (Revised Revised Report ..), R3RS, R3.899RS, R4RS.

SLIP : Serial Line Internet Protocol
<communications, protocol> (SLIP) Software allowing the Internet Protocol (IP), normally used on Ethernet, to be used over a serial line, e.g. an EIA-232 serial port connected to a modem. It is defined in RFC 1055.

SLIP modifies a standard Internet datagram by appending a special SLIP END character to it, which allows datagrams to be distinguished as separate. SLIP requires a port configuration of 8 data bits, no parity, and EIA or hardware flow control. SLIP does not provide error detection, being reliant on other high-layer protocols for this. Over a particularly error-prone dial-up link therefore, SLIP on its own would not be satisfactory.
A SLIP connection needs to have its IP address configuration set each time before it is established whereas Point-to-Point Protocol (PPP) can determine it automatically once it has started.

TCP/IP : Transmission Control Protocol over Internet Protocol.

The de facto standard Ethernet protocols incorporated into 4.2BSD Unix. TCP/IP was developed by DARPA for internetworking and encompasses both network layer and transport layer protocols. While TCP and IP specify two protocols at specific protocol layers, TCP/IP is often used to refer to the entire DoD protocol suite based upon these, including telnet, FTP, UDP and RDP.

VESA: Video Electronics Standards Association

<body, standard> (VESA) An industry standards organisation created in 1989 or 1990 mostly(?) concerned with IBM compatible personal computers. The first standard it created was the 800 x 600 pixel Super VGA (SVGA) display and its software interface. It also defined the VESA Local Bus (VLB).

Appendix D Bibliography

Books :

- **PVM Parallel Virtual Machine**
  MIT Press, ISBN: 0-262-57108-0
- **Managing projects with make**
  O'Reilly, ISBN: 0-937175-90-0
- **Graph theory,**
  Springer Verlag, ISBN: 185233259X
- **C How to program**
  Prentice Hall, ISBN: 0132261197
- **Unix for programmers and users**
  Prentice Hall, ISBN: 0-13-061771-7
- **The design of the Unix operating system**
  Prentice Hall, ISBN: 0-13-201757-1 025
- **Computer system architecture**
  Prentice Hall ISBN: 0131755633
- **Biology of Mind,**
  Fitzgerald Science, ISBN: 1891786075

Newspapers :

- **IEEE, Distributed systems online**
  http://computer.org/dsonline
- **IEEE, Computer**
  http://computer.org
- **Fortune, business**
  http://www.fortune.com
- **Wired, hype and interesting**
  http://www.wire.com/wired

Internet portals :

- **Online dictionary**
  http://www.foldoc.org
- **Netcraft, market analysis**
  http://www.netcraft.com
- **Internet society, statistics**
  http://www.isoc.org
- **Sofware panorama, independent technology review**
  http://www.softpanorama.org
- **Cybergeography, network technologies**
  http://www.cybergeography.org
- **Top 500 supercomputers**
  http://www.top500.org
- **Reuters, news agency**
  http://www.reuters.com
- **Protein databank**
  http://www.rcsb.org

Index