Defence Science Journal, Vol 43, No 2, April 1993, pp 129-137

Parallel Processing for Supercomputing Speeds

K. Neelakantan, P.P. Ghosh, M.S. Ganagi, A.V.S.S. Prasad,
Anu Khendry and K. Mahesh

Advanced Numerical Research and Analysis Group, Hyderabad-500 258

ABSTRACT

The various approaches to improving the speeds of computers by exploiting parallelism is briefly described. The architecture of PACE which is a loosely coupled, message-passing MIMD machine developed by ANURAG is described. PACE is an open architecture which is independent of the specific hardware used for the computing nodes. Several versions of PACE have been configured and the performance figures for some applications are presented.

1. INTRODUCTION

There are several areas of science and technology which rely very heavily on high speed digital computers. Some of these include:

(a) *Predictive scientific modelling:* This requires extensive numerical simulation experiments on fast computers. Often, large scale numerical computations have to be performed on a tremendous volume of data in order to achieve the required accuracy and turn-around time. Examples are weather forecasting, oceanographic studies, astrophysical models,. world economics, and biological system simulations.

(b) *Engineering and design automation:* Structural analysis using finite element techniques involves the solution of large systems of algebraic equations. Such computations require supercomputing speeds when the number of elements is large. Computational fluid dynamics (CFD) is another area which requires supercomputing speeds. The use of CFD codes for aircraft design reduces the dependency on wind tunnel experiments and is therefore being used extensively. Supercomputers are also used in AI systems, VLSI design, etc.

(c) *Energy resource exploration:* Supercomputers find an important application in oil and natural gas exploration. They allow for rapid processing of seismic signals and oil reservoir modeling. Nuclear reactor design and safety analysis is another area where high speed computers are needed.

(d) *Medical, military and basic research:* Supercomputers are needed in modern medical diagnosis equipment such as computer aided tomography. High speed computers are also required in medical image processing and reconstruction of 3-d images from 2-d X-ray and MRI data. Genetic engineers demand fast computers for studying molecular biology, artificial synthesis of proteins, etc. Supercomputers are required for weapons research and the design of missiles, cartographic surveys, etc. Basic research, in almost all areas, requires supercomputers. Standard examples are quantum chromodynamics, molecular dynamics, crystal growth studies and chemical kinetics studies. Breakthroughs in many areas depend on the availability of reliable computing systems that can be used to suggest new theories, interpret experiments, model real processes and provide accurate calculations in a reasonable time.

Traditionally, CFD has been the most demanding of calculations performed on computers. The demand for computation speeds has always outstripped the

capability of available technology. This is because a typical calculation involves around 10^{12} computations. Even with a system capable of 10^8 to 10^9 floating point operations per second, each calculation would take 1000 to 10,000 seconds.

PACE is an acronym for the Processor for Aerodynamic Computations and Evaluation. PACE is a loosely coupled, message-passing, MIMD machine based on a cluster concept. ANURAG has developed PACE for use in CFD calculations. However, since CFD calculations involve the solution of partial differential equations, PACE finds a very wide variety of applications in scientific computing.

The purpose of this paper is to describe the architecture of PACE, its user's model and report its performance figures. A brief overview of parallel architectures is given in the next section. This is followed, in section 3, by a detailed description of the PACE architecture. The user's model of PACE and the programming environment is explained in section 4 while the performance figures are given in section 5. Section 6 is devoted to conclusions.

2. AN OVERVIEW OF PARALLEL PROCESSING

Ever since the first digital electronic computer (the ENIAC) was built in 1946, the need for higher and higher computing speeds has been felt. The earlier computers used vacuum tubes which offered switching speeds in the range of microseconds. Computer speeds have improved with technology in the area of semiconductors till the mid 1970s saw logic gates based on emitter-coupled-logic devices with switching speeds of around one nanosecond. At this stage, the speed of computation was limited by the amount of time it took for the information to be transmitted within the system rather than the gate delays. There were two options at this stage: (a) to look for newer architectures, or, (b) reduce the physical size of systems. Attempts have been made in both directions but the real breakthrough in speeds came with the evolution of newer architectures which exploited parallelism.

2.1 Levels of Parallelism

Parallelism is not a new concept and has been used to improve the effectiveness of computers since the earliest designs. It can be applied at various levels which can be classified[1] as (a) job level, (b) program level, (c) instruction level, and (d) arithmetic and bit level.

2.1.1 Job Level Parallelism

Job level parallelism is implemented in most computer installations. Viewed in a simplistic manner, every job is divided into several sequential phases each of which requires a different systems program and system resources. It is therefore natural to have several jobs running on the system to exploit system resources in parallel. For example, an I/O operation is very slow compared to the actual program execution and therefore, any reasonably large computer installation provides several I/O channels or peripheral processors which can perform the I/O in parallel with program execution.

2.1.2 Program Level Parallelism

Program level parallelism is most important in the context of supercomputers. Within a large program, there could be sections of code which are quite independent and could therefore be executed in parallel on different processors in a multiprocessor environment. This is the central idea in parallel processing. There are several methods of implementing this architecturally and we will discuss these presently.

Program level parallelism could arise in several ways, the most commonly encountered one being a 'DO' loop which can be replaced by one or more vector instructions. Essentially, the idea is to exploit the parallelism offered by the vector hardware and execute the DO loop instructions in a single vector operation. This has been exploited in the well-known vector machines such as the CRAY, CYBER, etc.

In certain programs, the successive executions of the body of a loop may be completely independent of each other. This arises, for example, in Monte Carlo analysis where the same calculations are repeated many times with different data chosen in a random fashion. In such cases, the full code can be loaded onto each processor in a multi-processor environment and the calculations for each sample done in parallel. In this case the various processors do not need to exchange data. They merely post the final results to a particular processor.

However, things are not usually so simple. Most often, the various segments of code running on different processors cannot run indefinitely without the need to exchange data. It would be nice if the onus of recognizing the parallelism could be left to the compiler. However, in practice, the programmer has to take care

of parallelizing the application. Presently, the hardware and architecture in general can only provide the necessary computing power and the ability to communicate. The actual parallelizing is left to the programmer at present. A lot of effort is being put into the development of automatic parallelizers but these are yet to mature.

2.1.3 Instruction Level Parallelism

Instruction level parallelism is important in enabling the unit processors in a multiprocessor environment to work faster. The instructions in a unit processor may be divided into several sub-operations, which may then be pipelined to speed up execution. Another approach which is popular in the newer microprocessors is instruction prefetching and the use of an instruction cache. Here, while a particular instruction is being executed and the processor bus is free, new instructions are fetched and loaded into an instruction cache. This overlap of instruction fetch cycles with the execution cycles enhances the processor speed.

2.1.4 Arithmetic and Bit Level Parallelism

Arithmetic and bit level parallelism is the lowest level of parallelism in computers. This is a self-evident concept. Obviously, using an 8-bit machine to do 64-bit arithmetic is going to be much slower than using a 64-bit machine in the single precision mode. This level of parallelism is dictated by considerations such as the requirements of the typical problem to be solved, available technology, the amount of hardware the system designer wants to use, etc.

2.2 Architectures

Having seen that program level parallelism offers a major increase in computation speed, we will now examine the various architectural concepts which allow for such parallelism. One can classify computers into four broad categories according to whether the instruction or the data streams are single or multiple[1]. (A stream is defined as a sequence of items – instructions or data – as executed or operation on by a processor):

SISD: Single instruction/single data stream. This is the conventional serial von Neumann computer in which there is one stream of instructions (and consequently only one processing unit) and each arithmetic instruction initiates one arithmetic operation.

SIMD: Single instruction/multiple data stream. This is a computer that retains a single stream of instructions but has vector instructions that initiate many operations.

MISD: Multiple instruction/single data stream. This is essentially a void class.

MIMD: Multiple instruction/multiple data stream Multiple instruction streams imply the existence of several instruction processing units and therefore, necessarily, several data streams.

One can see that only two of the categories described above are of interest as far as parallel processing is concerned i.e., the SIMD and MIMD machines. The SIMD architecture can be realised in three ways as shown in Fig.1. Of these, the vector pipelined

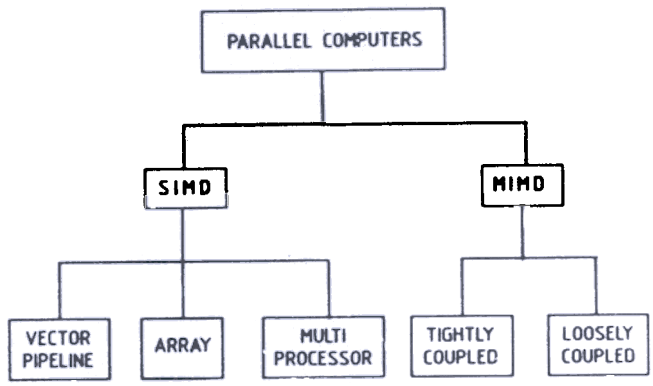

Figure 1 Classification of parallel computers.

technique, is the most popular one. We shall not go into the details of this architecture but detailed descriptions may be found elsewhere.

The MIMD machines have multiple processors (as opposed to processing elements such ALUs) and can therefore operate on multiple data streams. Unlike the SIMD machines, the processors in a MIMD machine are complete in themselves and can really operate independently if required (they are not subject to centralised control except for synchronisation).

The various processors in a MIMD machine are linked together. This link can be established in two ways: (a) tightly-coupled systems where the processors share a global memory (this does not preclude each processor from having its own local memory); and (b) loosely-coupled systems where there is no shared memory but the processors communicate with each other through I/O ports.

2.2.1 Tightly Coupled Systems

In general, these systems consist of m processors connecting n memory banks through a communication network[2,3] as shown in Fig. 2 (n and m are equal and, in order to reduce contentions for memory, n is usually a prime number). It may be noted that each processor P_i can act as a master so that this is a multi-master system. Problems arise in the interconnection network, these being: (a) two or more processors can request access to the same memory bank at the same time, and (b) two or more processors can require a particular communication path in the interconnection network to access different memory banks. In both these cases, some of the processors must wait. In turn this leads to the requirement of an arbitration mechanism to resolve contention (as is expected in any multimaster system). The commonly used communication networks structures are: (i) the shared bus, (ii) the cross-bar switched network, and (iii) the multiple shared bus structure.

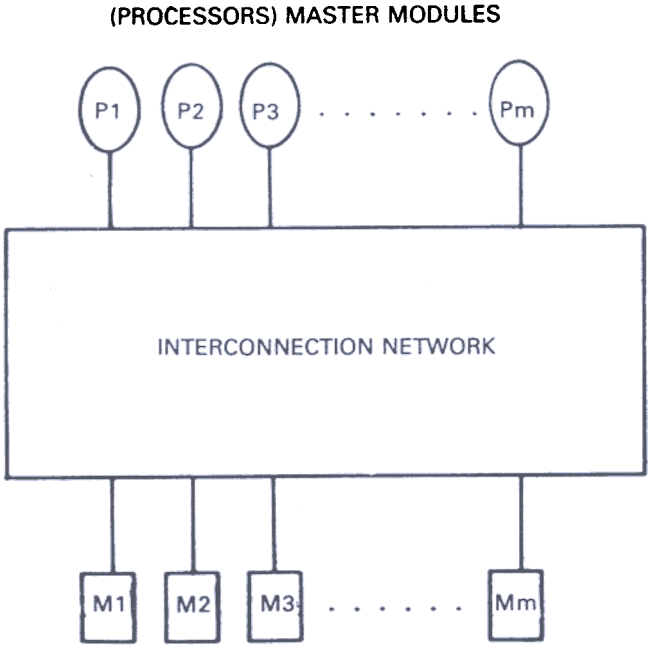

Figure 2. A generalized block diagram of a tightly coupled MIMD machine.

2.2.2 Loosely Coupled Systems

These systems do not have any shared global memory but consist of a network of N processors each with its own local memory. The processors communicate with each other via I/O ports. In the case of PACE, the processors do not use an explicit I/O port but write the data directly into the buffer memory space of their neighbours. The most convenient classification is in terms of the communication network topology. Again as in the case of tightly coupled systems, one would ideally require a network which is completely connected. However, this is expensive, and therefore the usual topologies[2] (Fig. 3) are (i) the linear

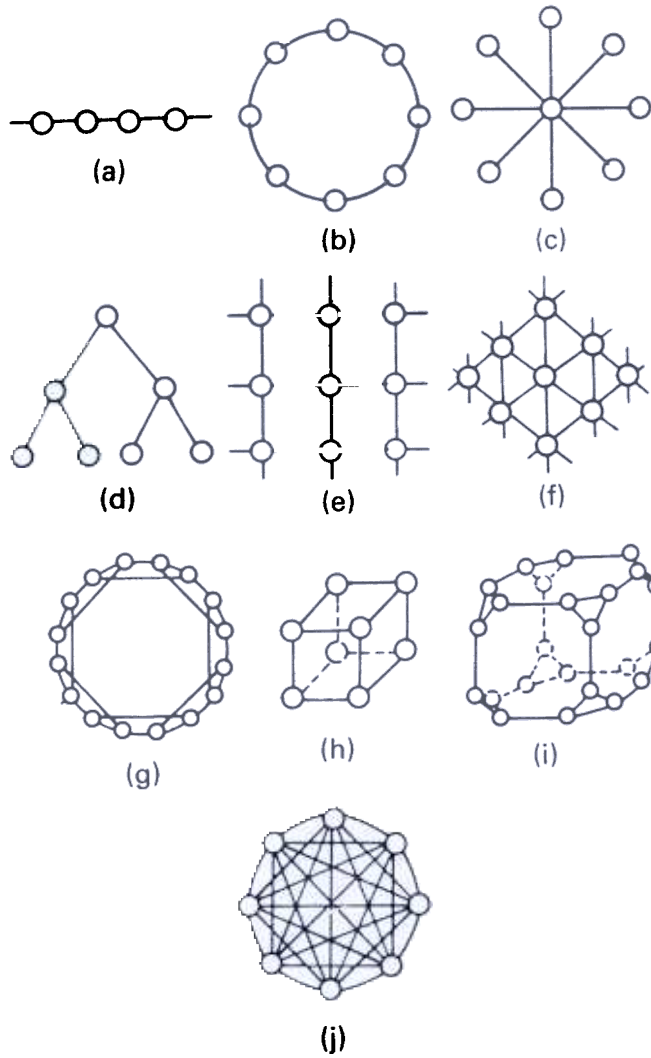

Figure 3. Some of the common topologies used in loosely coupled architectures (a) linear 1-dimensional array, (b) the ring, (c) the star, (d) the tree, (e) near-neighbour mesh, (f) the systolic array, (g) the chordal ring, (h) the cube, (i) the 3-cube connected cycle, and (j) the n-dimensinal hyper cube.

1-dimensional array, (ii) the ring, (iii) the star, (iv) the tree, (v) the nearest neighbour mesh, (vi) the systolic array, (vii) the chordal ring, (viii) the cube, (ix) the 3-cube connected cycle and finally (x) the n-dimensional hypercube[4].

In the case of the MIMD machines, the effective utilisation of the machines does not depend on the

vectorizability of the problem. This is obviously a more general architectural concept and therefore parallelism can be invoked in other ways as well. One could break up the problem into several sub-tasks which could be carried out independently and in parallel without any need for the various sub-tasks to communicate with each other. Obviously this cannot be carried too far and the sub-tasks would need to interact with each other at some stage. One can therefore define a granularity for the problem, i.e., the size of the sub-tasks that can be defined before there is a need for synchronization between them. Large granularity reduces the need for communication but some processors which finish their tasks earlier will have to wait (i.e., be idle) for long times while small granularity increases the communication overheads.

3. PACE ARCHITECTURE AND HARDWARE

PACE is a loosely coupled MIMD machine. PACE is a parallel processing system which is scaleable. This scaleability was planned along two fronts:

(a) The number of processors can be varied to suit the requirements of the applications. For example, certain applications such as spectral analysis through the FFT algorithm cannot be efficiently parallelised with a large number of processors. On the other hand, CFD codes can be run efficiently even on massively parallel machines.

(b) The power of each processing node can be scaled. With advances in processor technology, today one can get extremely fast processors which pack a lot of computing power. The PACE concept allows for upgrading the basic processor at each node.

Originally, it was planned to implement PACE as a 7-dimensional hypercube. For various reasons as reported elsewhere[5], the Motorola 68020 processors were selected for implementing the first prototype of PACE. However, during the detailed planning it was realised that, since each processor was based on the VME standard, the VME back-plane provides the most inexpensive medium of communication. Consequently, the approach was modified and the final architecture is based on a cluster approach. Though the common bus architecture is more common in a tightly coupled system, we based the PACE model on a loosely coupled architecture where messages are transmitted by merely writing data into a linear communication buffer. No explicit locking mechanism has been implemented within a cluster.

The architecture of PACE is shown in Fig. 4. PACE consists of a Front-End-Processor (FEP) which is connected to 4 super-clusters by means of VME-to-VME communication links. These VME-to-VME links provide high speed parallel (32-bit wide) communications between the two VME backplanes. As shown the super-clusters are completely connected to each other by VME-to-VME links. Each super-cluster has two CPUs exclusively devoted to communications. One CPU handles intra-super-cluster messages while the other handles inter-super-cluster messages.

Each super-cluster has four clusters connected to it. Each cluster has 8 CPUs connected on a VME backplane. The clusters are linked to the super-clusters by VME-to-VME links as shown in Fig. 4. Thus each super-cluster has 32 CPUs and the 4 super-clusters can accommodate 128 CPUs. The CPUs within the cluster are completely connected over the VME bus. They communicate with each other by directly writing the messages into the appropriate buffer space over the VME bus. For communication across clusters within the same super-cluster, the CPUs within a cluster pass the message on to the super-cluster which then passes on the message to the node in the destination cluster. Communication between nodes in different super-clusters, takes three hops which involves two super-clusters, the source cluster, and the destination cluster.

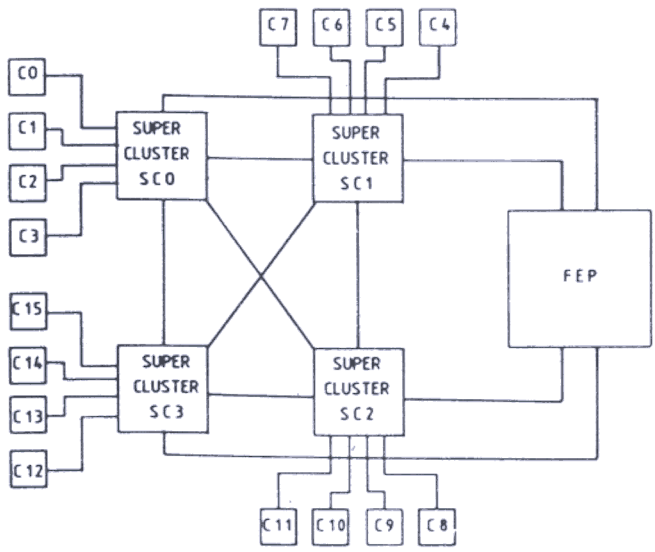

Figure 4. The architecure of PACE.

The initial prototypes of PACE were based on the MOTOROLA MC 68020 processor. A 4-node prototype based on the MC 68020 processor (working at 16.67 MHz) was first established. This used the VME bus for communication. The VME backplane is 'natural' to the MOTOROLA family of processor and was found to provide the necessary bandwidth and operational flexibility. Later a 8-node prototype based on the MC 68030 processor (working at 25 MHz) was developed[6]. This 8-node cluster forms the back-bone of the PACE architecture. The 128 node prototype is based on the MC 68030 processor working at 33 MHz. In order to enhance the floating point speed, ANURAG has developed its own custom floating point processor, ANUCO. The processor board has been specially designed to accommodate the MC 68881, MC 68882 or the ANUCO floating point accelerators.

The architecture of PACE depends only on the fact that each processor board should be able to access the memory of another CPU through a suitable bus. Necessary care has been taken in the systems software to ensure that there is no dependence on any hardware specific functions. Consequently, it was possible for us to configure models of PACE based on other processors.

ANURAG configured a 2-node version of PACE based on the Intel i860 processor. Since the specific CPU boards used were not configured on a standard bus, the communications were established through a common multiported memory. This was basically done as a demonstration to prove that the concept was portable.

Later, the PACE architecture was ported on the SPARC-II processor working at 40 MHz. The SPARC-II CPUs are based on the VME-bus and the porting was a relatively simple affair. The advantage of the SPARC-II version of PACE is that the front-end processor is compatible with the SUN workstations. In fact, the operating system is SOLARIS and even the graphics features of SUN are fully supported. The user will therefore have a wide choice of third party software which can be executed on the front-end. The PACE user model allows for easy parallelization of some of this software if it is available in source code form.

Some results obtained on the models based on the MC68020, MC68030 and SPARC-II are presented in section 5.

4. USER'S MODEL

The user's model of PACE has been designed to minimize the programming efforts on the part of the user. The user interacts with the FEP which is a standard UNIX engine. The FEP maintains all the system resources such as winchester space, floppy drives, cartridge tape drives, etc. All I/O devices and terminals are connected to the FEP.

The parallel processor is really treated as a resource of the FEP. The user writes his program in a sequential fashion (called the 'host' program). All computationally intensive portions of the programs are written as subroutines which are executed in parallel on the parallel processor. The user therefore needs to parallelise only the computationally intensive parts of the program which are treated as subroutines (called the 'node' program) to be called by the host program.

In order to enable the user to create, debug and execute his programs, ANURAG has written a parallel programming environment called ANUPAM (ANURAG's Parallel Applications Manager). ANUPAM runs under UNIX and consists of several modules and utilities which are explained below.

4. Preprocessor

The preprocessor allows the user to prepare his task for execution. It compiles the jobs for execution, links the appropriate run-time libraries, carries out the topology mapping and creates the executable modules. The preprocessor allows the user to specify various system options such as the amount of communication buffer space, the programming language, etc. An integral part of the preprocessor is the Topology Mapper which maps the logical topology conceived by the user onto the physical topology of the parallel machine. This makes the physical architecture of the parallel machine transparent to the user.

4.2 The Simulator

PACE is a number crunching machine, and, it is expected that jobs that run on the system are computation intensive tasks. It is presumed that the jobs that run on PACE are debugged, well tested codes as computer time is costly. In order to facilitate debugging a simulator has been provided on the FEP of PACE. The simulator provides exactly the same programming environment as the parallel machine.

Users can check the logic of their codes by running it under the simulator, thus obviating the use of parallel machine for debugging purpose. Besides offering a framework for parallel computing, it also provides the user with the communication statistics of the program which is very essential for evaluating the efficiency of the user algorithm.

4.3 The Queue Manager

The Queue Manager is a program which allows multiple jobs to be submitted in a multi-user environment on the FEP. It queues the jobs and allows them to be executed in order of submission.

4.4 Run-time Libraries

These actually form the core of ANUPAM. These contain the communication libraries as well as other utilities required for executing tasks on PACE. The communications routines have been optimized to allow for extracting the best performance from the hardware.

4.5 Communications Debugger

A comprehensive communications debugger has been provided with PACE. When invoked, this checks the communications at run-time for errors, such as parameter mismatch in the communication calls, deadlock due to buffer overflow, etc., and reports errors along with statistics of usage of the communications.

4.6 Source Level Debugger

A window based source level debugger for debugging the parallel application at the source level is being designed as part of ANUPAM. This software provides a user-friendly environment which facilitates the user to migrate across the nodes in PACE for debugging from the FEP by a sequential source level debugger. This also appraises the user of the communication status of the nodes in the system.

4.7 Parallel Library

A library of pre-parallelized subroutine is available The user can add other subroutines to this library.

4.8 Other Utilities

Other utilities to monitor the status of tasks, provide timing information etc., are also available.

The ANUPAM software only depends on the availability of UNIX at the front-end. The software is completely portable across machines with very few modifications (the modifications relate to the physical addresses of the CPU boards).

5. PERFORMANCE FIGURES

Several applications programmes were run on the various models of PACE. These include the Linpack benchmark[6], FFT[7], Simulation of neural networks[8], several CFD codes[9], Finite-element analysis codes[10], etc. Of these the LINPACK benchmark programme which solves a set of linear equations is often used as an index of the computation speeds of present day computers. The LINPACK rating of the various models of PACE is given in Table 1. While the LINPACK rating is only indicative, it gives a fair idea of the capability of the system.

It may be seen from Table 1 that PACE-128 (based on the Motorola 68030 processor with MC 68882 coprocessor) delivers over 30 MFLOPS of speed for large problems. The speed per node is 0.33 MFLOPS. The speed per node has been enhanced by incorporating ANUCO, ANURAG's custom floating point processor to 0.75 MFLOPS per node. With the SPARC-II processor, about 4 MFLOPS per node are obtained. A speed of 15 MFLOPS on Linpack with a 1000×1000 matrix size was obtained. This is around half the CRAY-1S speed (27 MFLOPS for 100×100 matrix size) and comparable to CRAY-XMS (17 MFLOPS for 100×100 matrix size).

From the preliminary experiments, it is believed that a 8 node version of PACE based on the SPARC-II would deliver around 30 MFLOPS while a 32-node version would deliver around 100 MFLOPS. Such performance figures put PACE in the true Supercomputer class.

Table 2 depicts the performance of PACE on a CFD code developed by ADA. This program is a generalized unsteady Euler equation solver. In order to validate the code, the computations have been carried out on standard fuselage. The results are reported in terms of the time taken per iteration per grid point. It may be seen from Table 2, that the Euler code is amenable to reasonably efficient parallelization on as many as 128 nodes provided the problem size is large enough. Details of the code and the algorithm is reported elsewhere[9].

Table 1 Single precision LINPACK performance of various versions of PACE

Version	Processor	Co-processor	Speed (MFLOPS)	Saturation speed (large matrix)
PACE-4	68020 @ 16.67 MHz	68881	0.175*	
PACE-8	68030 @ 25 MHz	68882	2.23	2.5
PACE-16	68030 @ 33 MHz	68882	4.6	5.0
PACE-32	68030 @ 33 MHz	68882	7.35	9.9
PACE-64	68030 @ 33 MHz	68882	9.16	19
PACE-128	68030 @ 33 MHz	68882	10.2	35
PACE-4 with ANUCO	68030 @ 33 MHz	ANUCO	2.993	
PACE-8 with ANUCO	68030 @ 33 MHz	ANUCO	5.4	
PACE-SPARC 4 Nodes	SPARC-II @ 40 MHz		15.2	

* 100 × 100 LINPACK

Table 2. Performance of PACE for aerodynamic code (SGUES)*

No. of nodes	Problem size (grid points)	Time/iteration/grid point (ms)
Uniprocessor	24576	5903
8	24576	635
	104448	945
16	24576	250
	104448	313
32	52224	140
	104448	132
64	52224	78.5
	104448	78.5
128	104448	48

* The processor is MC 68030 with MC 68882

Results for other applications have been reported elsewhere[6-10].

6. CONCLUSIONS

The architecture of PACE has been discribed. PACE is a scaleable architecture. We have configured 4, 8, 16, 32, 64 and 128-node models of PACE. This has been done using different processors such as the MC 68020, MC 68030, Intel i860 and the SPARC-II. In the future, one can upgrade the processor as technology develops.

In principle, the number of nodes can be increased to 1024 with only minor modifications in the architecture and in ANUPAM. In the future, ANURAG will support massively parallel computers based on the SPARC-II. It is also possible for ANURAG to upgrade

the processor from the present generation RISC processors to super scalar processors as and when they become commercially available. The objective has been to provide the users with a flexible hardware platform that can be scaled to meet their requirements.

REFERENCES

1. Hockney, R.W. & Jesshope, C.R. Parallel computers, Adam Hilger, Bristol, 1981.
2. Feng, Tse-Yung. A survey of interconnection networks. *IEEE Computer*, 1981, **4**(12), 12.
3. Yalamanchili, S. & Aggarwal, J.K. Reconfiguration strategies for parallel architectures. *IEEE Computer*, 1985, **18**(12), 44.
4. Fox, G. The caltech hypercube in scientific calculations: a preliminary analysis. *In* Supercomputers, edited by F.A. Matsen & T. Tajima. University of Texas Press, Austin, 1986.
5. Neelakantan, K. & Athithan, G. PACE user information. ANURAG Report No. ANU/ PACE/89/001, 1989.
6. Neelakantan, K.; Ghosh, P.P.; Ganagi, M.S.; Athithan, G.; Atre, M.V. & Venkataraman, G. Performance characteristics of a hypercube type parallel computer. *Current Science*, 1990, **59**(20), 982.
7. Ganagi, M.S. & Neelakantan, K. Implementation of the fast-fourier transform algorithm on a parallel processor. *Current Science*, 1991, **61**(2), 105.
8. Athithan, G. Speed-up factors for simulation of neural networks on a parallel computer. *Current Science*, 1992, **62**(8), 568.
9. Ganagi, M.S.; Singh, K.P.; Athithan, G. & Atre, M.V. A Fluid dynamics study on ANURAG's parallel computer PACE-8. *Current Science*, 1991, **60**(12), 694.
10. Prasad, M.D.R.; Ramanathan, R.K. & Ganagi, M.S. Performance of ANURAG PACE-16 parallel computer for finite element analysis. *In* Innovative applications in computing. Tata McGraw Hill, New Delhi, 1993.