Software for MIMD Message Passing Machines

- Old languages with additions for concurrent programming.
  - Parallel versions of C
  - Parallel versions of Fortran
- Routines are added for access to communication links.
- One or a few processes are placed on each processor.
- Mechanism of inter-process communication depends on process location.
- The hardware changes but the languages remain the same.
  - Important for market acceptance.

Software for MIMD Message Passing Machines

Portability Problems

- Code written in C for an NCUBE hypercube won't run on an Intel hypercube.
- Code written for an 8 node machine will use only 8 nodes of a 16 node machine.
- Code written for an 8 node machine will not run at all on a 4 node machine.

Excess Parallelism & Virtual Concurrency

- Sufficient Parallelism

  With the *Parallel C/Fortran* approach we extract as much concurrency from the problem as we need. We can then write a program for each processor.

- Excess Parallelism

  If instead we extract as much concurrency from the problem as possible, we find that we will often have more parallelism than we have processors. We have *excess parallelism*.

- Virtual Concurrency

  In order to support excess parallelism, we run multiple processes on a single processor. This multi-tasking we call *virtual concurrency* because the time-sliced processes must appear to run concurrently.

The Benefits of Excess Parallelism

- Masking of Message Latency

  In MIMD message passing systems the latency of message passing is often a limiting factor.

  In order to mask this latency such that it doesn't effect the execution time of the program, we can *deschedule* a process which is waiting for communication such that it no longer gets any CPU time.

  The greater the level of *excess parallelism* the greater the masking effect.

- Abstraction

  The number of processors is no longer important. The real concurrency will expand to fit any number of processors until there is only one process on each processor.

  Programs are more portable and easier to write.

Portable Programs

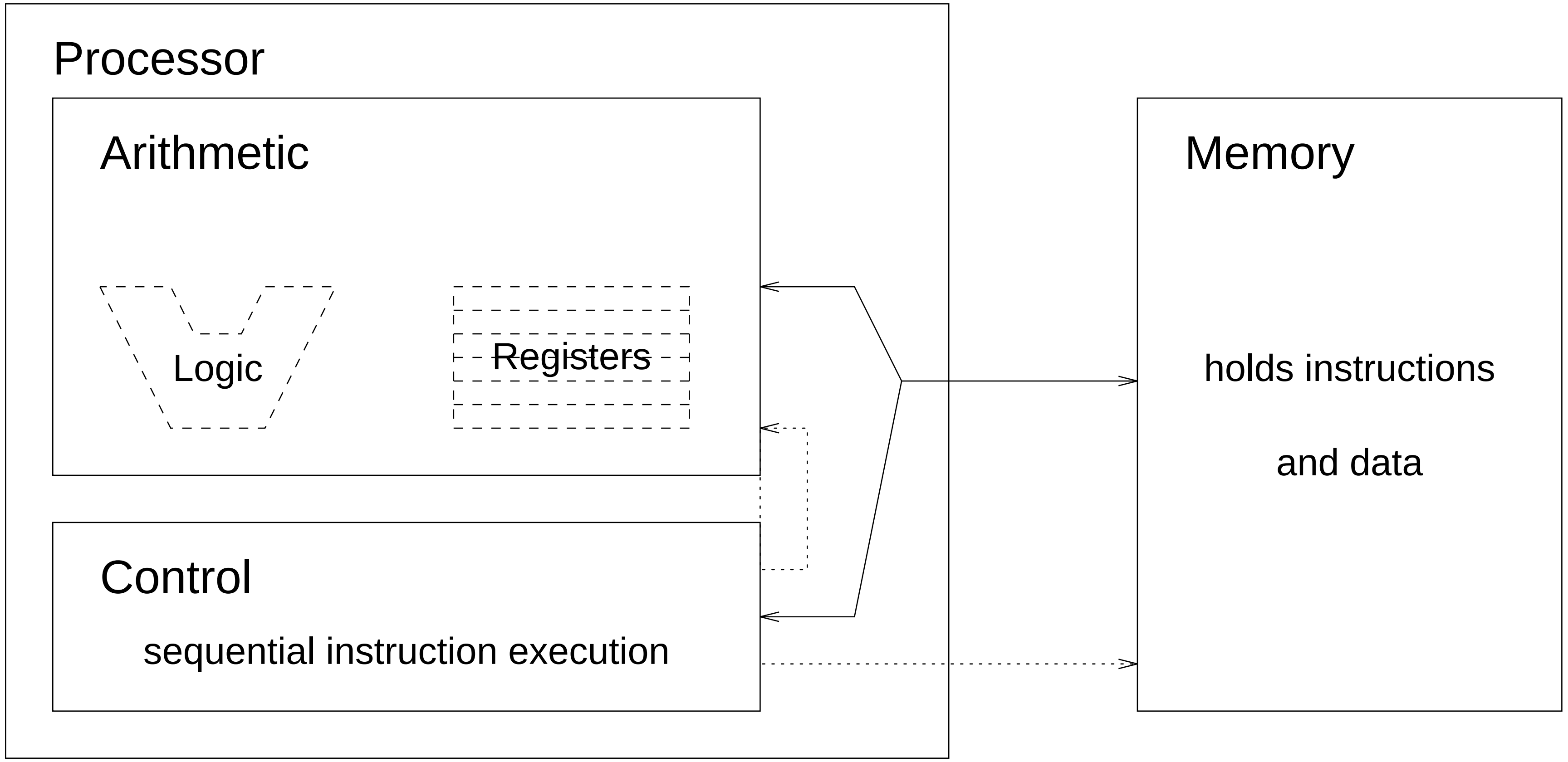
Consider Portability for sequential machines:[1]

- All sequential machines are assumed to be von Neumann machines.

Programmers seldom need to know more about the architecture than the approximations contained in the von Neumann model, since exact processor details are hidden by the compiler.

[1]Most code for sequential computers is portable, there is no market for a book on programming in C for the Sun workstation, since C is the same on all machines. Some problems still remain with the portability of i/o code, since this relies on a number of emerging standards such as operating systems and windowing systems.

Von Neumann Machine



Von Neumann Machine

The von Neumann model is a loose description of a real machine.

- Processor

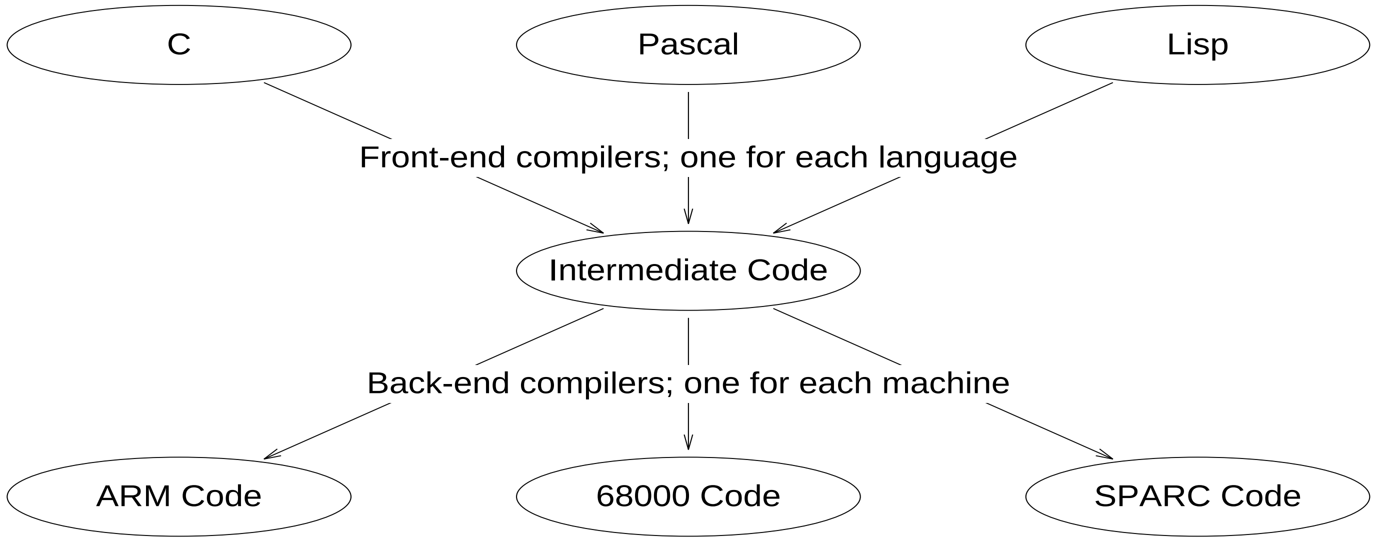
  A processor that performs instructions such as "add the contents of these registers and put the result in that register."
- Control

  A control scheme that fetches one instruction after another from the memory for execution by the processor, and shuttles data between memory and processor one word at a time.
- Memory

  A memory that stores both the instructions and data of a program in cells having unique addresses.

Portable Programs – Virtual Machines

For portable compilers we introduce an intermediate code.

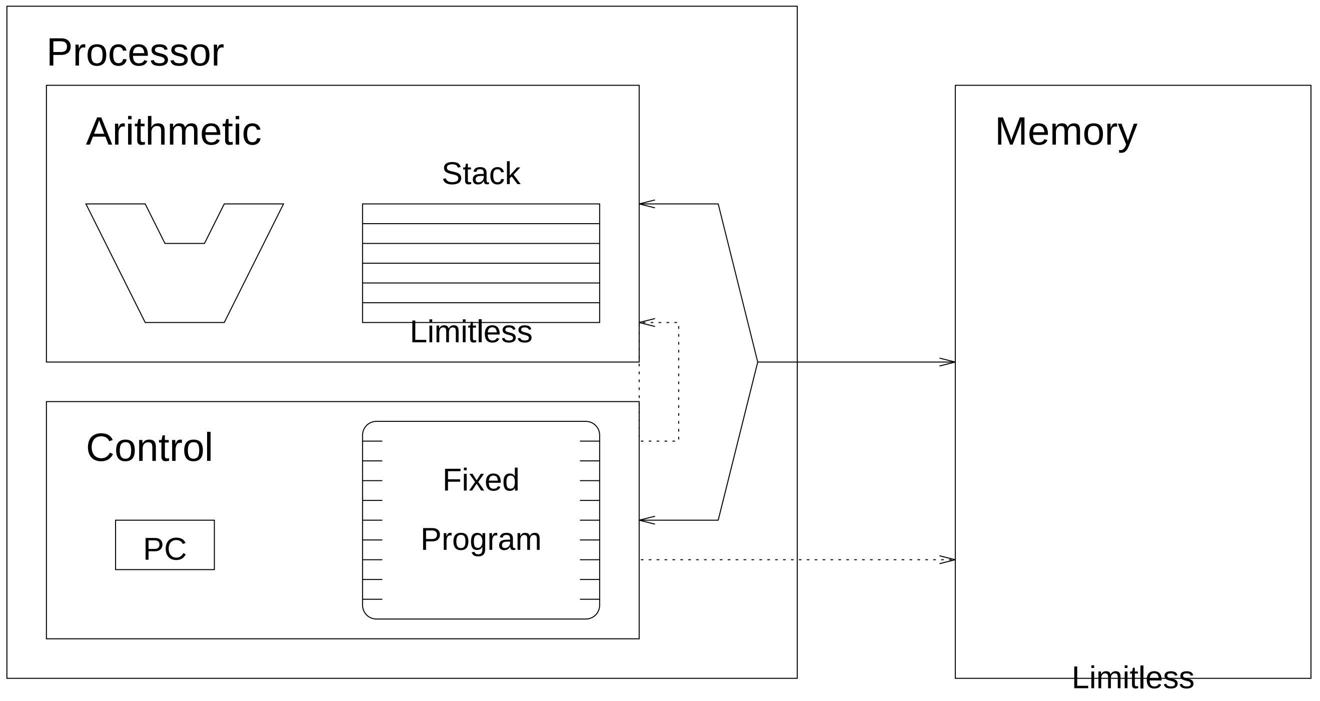

C
Pascal
Lisp
Front-end compilers; one for each language
Intermediate Code
Back-end compilers; one for each machine
ARM Code
68000 Code
SPARC Code

- All front-end compilers are written in this code and compile to this code.
- All back-end compilers and run-time systems support this code.

This is the machine code for a *virtual machine*. The virtual machine must be carefully chosen to allow efficient code generation for all real machines.

Portable Programs – Virtual Machines

A typical virtual machine will perform a defined set of arithmetic operations on a stack, with variable storage available in addressed memory locations.



Both the stack and the addressed memory are considered as unlimited in size.

Portable Programs – Virtual Machines

Major Problems

- A real machine cannot have unlimited memory or stack space.
- A real machine may have a number of fast access registers.

Solutions

- Virtual Memory

  A system of virtual memory with swap space on disc allows a small amount of real memory to appear as a very large (if not unlimited) amount of memory.

  This system must be supported by the run-time system and/or additional hardware.

- Code Optimization

  The back-end compiler will re-code stack arithmetic as register – register arithmetic so as to reduce off-chip data accesses.

  In general it is the task of the back-end compiler to produce the most efficient code possible for the target architecture.

Portable Programs for Parallel Machines

We have seen that portability relies on the concept of a generic virtual machine or processing model[2].

Having defined the functionality of the model the system designers must support the model for any particular machine.

We have introduced three different processing models

- Data Parallel Model
- Shared Memory Model
- Communicating Processes Model

[2]A processing model may be considered as a loosely specified virtual machine which has no machine code.

Data Parallel Model

- This model assumes that we have a *single process* which performs *calculations over whole data structures in parallel.*

The data parallel model has led to Fortran 90 as a portable language for vector processors and other SIMD machines. Some MIMD machines adopt the data parallel model due to its ease of use, although the model (and hence the language) has to be extended in order to achieve process parallelism.

For portable languages the user is unable to control the mapping of data to processors. A poor mapping may result in unnecessary communications overheads on a real processor.

Shared Memory Model

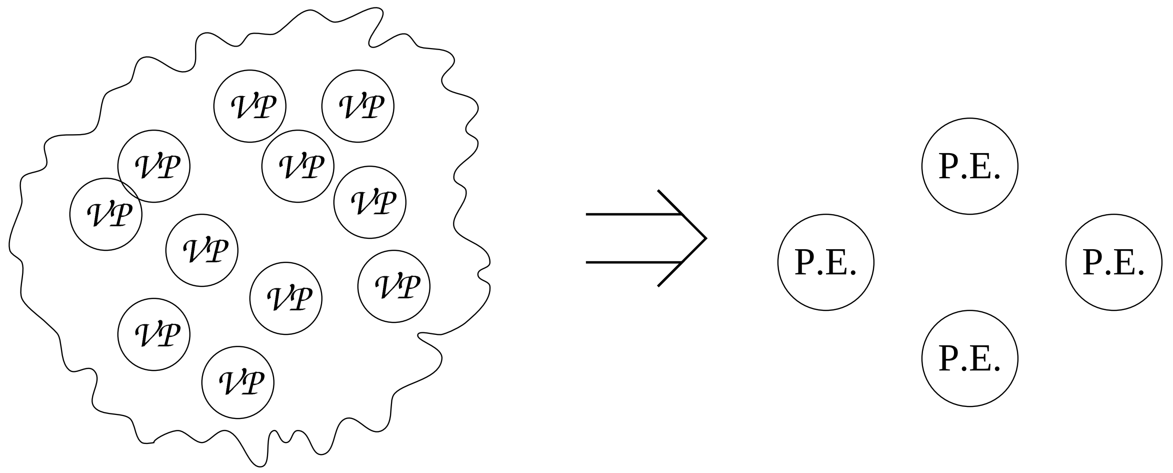
- Assumes that we have *multiple processes* all of which *share the same memory map*.

The standard multi-tasking facilities of UNIX convert C into a portable language for shared memory machines.

In order to create portable code we must exploit the *excess parallelism* contained within a problem since we no longer know how many processors the real machine will have. For many machines this is wasteful where the overheads of virtual concurrency are heavy.

Virtual Processors

We can consider this programming style as programming for a set of *virtual processors*[3].



We program as if for an arbitrarily large number of *virtual processors*, one per concurrent process, and then map the *virtual processors* onto the available real processors.

[3]c.f. virtual memory

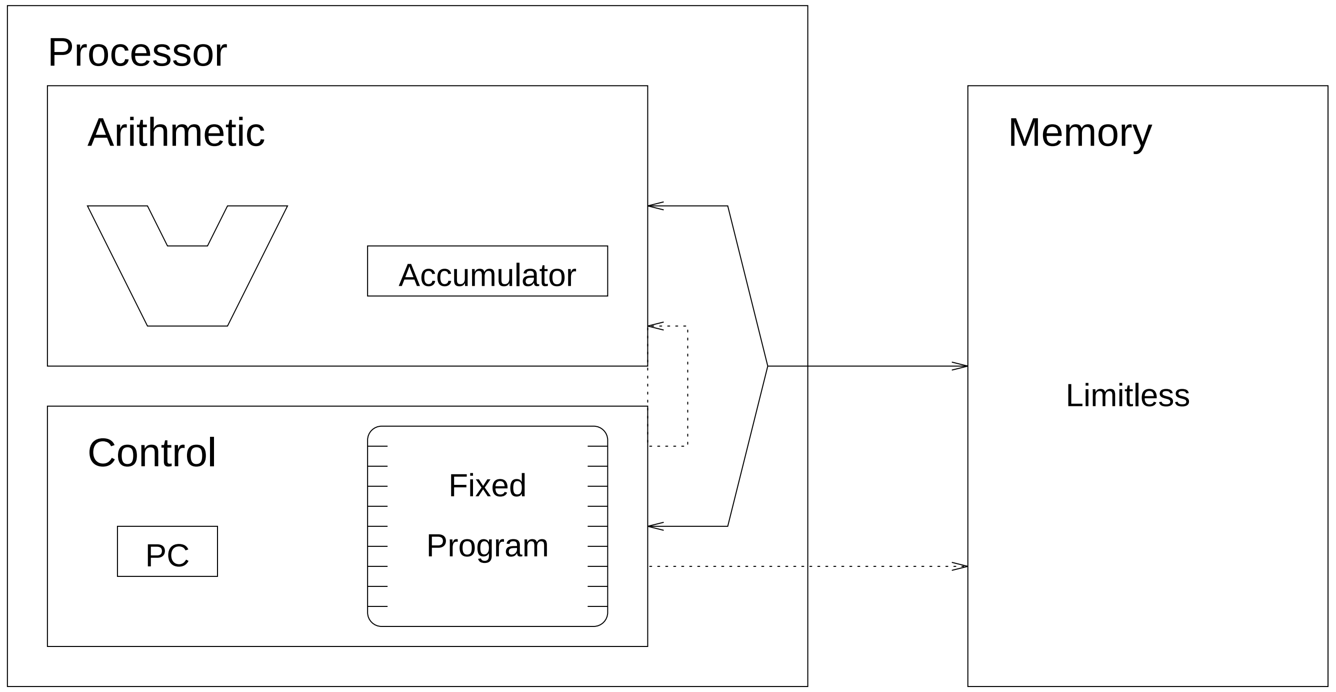
Shared Memory Model

The shared memory model is typified by the much cited PRAM machine, an ideal processor on the same lines as the Paracomputer.

Real shared memory machines are far too small to be considered as general purpose machines.

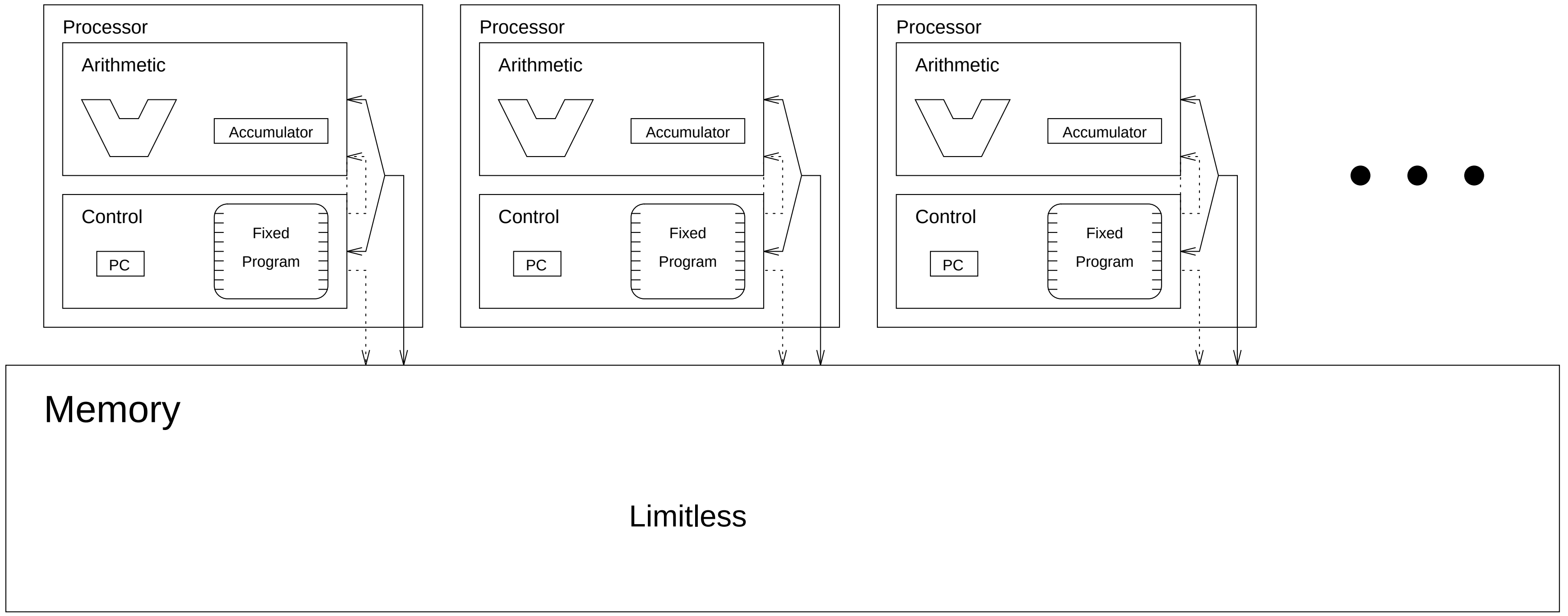
Although a distributed memory machine may emulate shared memory in software, the assumptions made in the model about uniform access times for all memory locations mean that code written for this model will run very badly on these machines.

Random Access Machine



The Random Access Machine (RAM) is a model used in algorithm comparison (complexity theorem). Each instruction in its fixed instruction set is assumed to take a single cycle to complete.

Parallel Random Access Machine



The P-processor Parallel Random Access Machine (PRAM) completes P instructions, one on each processor, in every cycle.

Communicating Processes Model

- Assumes that we have *multiple independent processes* which *communicate and co-ordinate via the sending of messages*. Each process can communicate with any other process, but can only access its own local memory.

A truly portable language for this model should be able to express fine-grain concurrency to allow maximum excess parallelism in order to hide communications latency.

The language should not contain references to process placement or the placement of communication channels since no knowledge of the underlying architecture can be assumed.

Occam attempts to fulfil this requirement with built-in support for fine grain parallelism and inter-process communication.

General Purpose Parallel Computers

Much of the research into Parallel Computer architecture is now aimed at building general purpose parallel computers. This research is driven by the requirement of usable power.

Parallel computers have always been difficult to program, the situation becomes worse as the machines get more powerful.
In order to achieve acceptance of parallel machines as cheaper desktop workstations than equivalent sequential machines, programming the parallel machines must become a task of similar complexity.

Here we consider a general purpose machine as one designed to support one or more portable languages based on one or more of the processing models.

General Purpose Parallel Computers

- Functionally
  - The programmer will be able to program without regard to or knowledge of:
    - - - process or data location
    - - - processor interconnection
  - A single program should run without modification on any machine.

- Performance
  - The performance of portable code should be comparable with that of hand-crafted code.

General Purpose Parallel Computers

- Implication
  - System software (compilers & operating systems) must contain the knowledge of how to get the best from the machine - this may call for simple architectures since complex functionality may be wasted[4].
  - Hardware must be tailored towards the programming model since there is now much less ability to tailor the programming model to the hardware.
- Pragmatism

  This is a major task, most systems are approaching the goal one step at a time.

[4] c.f. CISC vs RISC