Concepts from High-Performance Computing

Lecture A - *Overview of HPC paradigms*

OBJECTIVE:

The clock speeds of computer processors are topping out as the limits of traditional computer chip technology are being reached.

Increased performance is being achieved by increasing the number of compute cores on a chip.

In order to understand and take advantage of this disruptive technology, one must appreciate the paradigms of high-performance computing.

The economics of technology

Technology is governed by economics.

The rate at which a computer processor can execute instructions (e.g., floating-point operations) is proportional to the frequency of its clock.

However,

$$\begin{aligned} \text{power} &\sim (\text{voltage})^2 \times (\text{frequency}), \\ \text{voltage} &\sim \text{frequency}, \\ \implies \text{power} &\sim (\text{frequency})^3 \end{aligned}$$

e.g., a 50% increase in performance by increasing frequency requires **3.4 times more power**.

By going to 2 cores, this performance increase can be achieved with **16% less power**[1].

[1] This assumes you can make 100% use of each core.

Moreover, transistors are getting so small that (undesirable) quantum effects (such as electron tunnelling) are becoming non-negligible.

$\rightarrow$ Increasing frequency is no longer an option!

When increasing frequency was possible, software got faster because hardware got faster.

In the near future, the only software that will survive will be that which can take advantage of parallel processing.

It is already difficult to do leading-edge computational science without parallel computing; this situation can only get worse.

Computational scientists who ignore parallel computing do so at their own risk!

Most of numerical analysis (and software in general) was designed for the serial processor.

There is a lot of numerical analysis that needs to be revisited in this new world of computing.

HPC paradigms

The implementation of algorithms on parallel computers is to a large extent dependent on the target architecture.

So before we can investigate algorithms that can take effective advantage of parallelism, it is necessary to consider how a parallel computer operates.

It is easy to write parallel programs if you don't care about performance!

There are two main ways in which computers can be organized to perform computations in parallel:

- shared memory
- distributed memory (message passing)

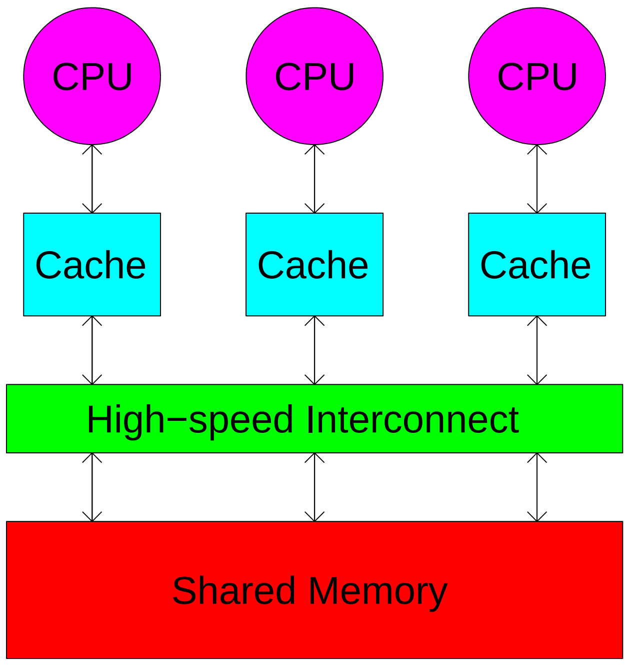
To make life interesting, lately these two ways have been combined to form a hybrid third way.

Shared-memory paradigm

Shared-memory computers have multiple processors that share access to a global memory space via a high-speed memory bus.

This global memory space allows the processors to efficiently exchange or share access to data.

The number of processors used in shared-memory architectures is usually limited because the amount of data that can be processed is limited by the bandwidth of the memory bus connecting the processors.

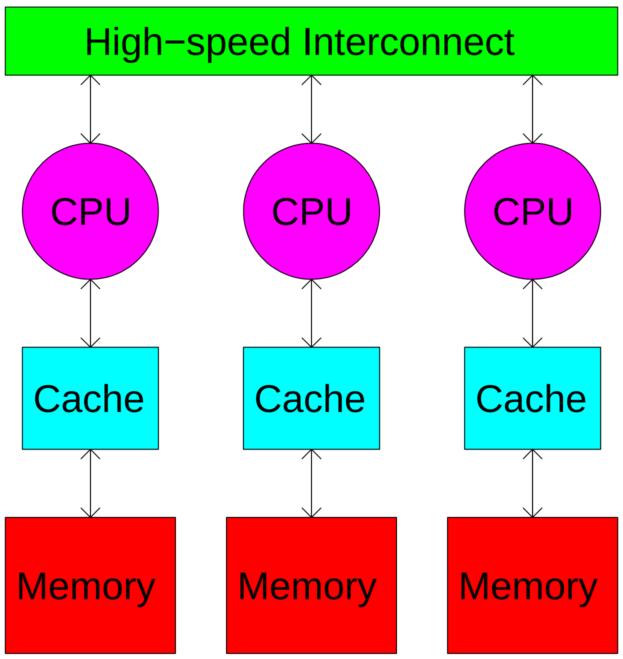


Distributed-memory paradigm

Distributed-memory parallel computers are essentially a collection of serial computers (nodes) working together to solve a problem.

Each node has rapid access to its own local memory and access to the memory of other nodes via some sort of communications network, usually a high-speed communications network (or "interconnect").

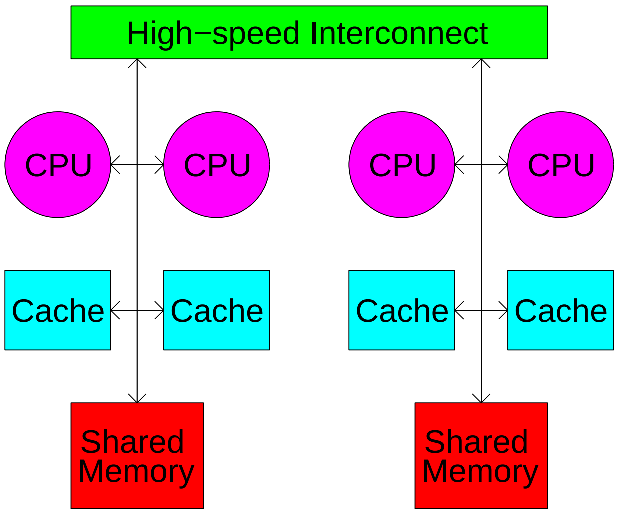
Data are exchanged between nodes as messages over the network.



Hybrid-memory paradigm

The latest generation of parallel computers now uses a mixed shared/distributed memory architecture.

Each node typically consists of a group of 2 to 16 processors connected via local shared memory, and the multiprocessor nodes are, in turn, connected via a high-speed interconnect.



Shared vs. Distributed

Some facts of life:

- Distributed-memory systems are more prevalent than shared-memory systems; hybrid-memory systems are becoming increasingly popular.

- Message-passing programs can execute on either distributed or shared-memory systems; shared-memory programs can only execute on shared-memory systems.

- Shared-memory programming is relatively easy; message passing is less so. For better or worse, this means the programmer has more fine-tune control on the program and can potentially handle more complicated tasks.

- Message-passing programs often outperform shared-memory programs even when run on a shared-memory system.

Message passing

One of the basic methods of programming for parallel computers is the use of *message passing libraries*.

These libraries manage data transfer between instances of a parallel program (usually) running on multiple processors by a simple send-and-receive mechanism.

Although this seems simple and intuitive, the details can become much more complicated. In particular,

- How are messages actually sent? (How are they buffered within the system?)
- Can a processor do other useful work while sending or receiving messages?
- How can sends and received be paired to ensure proper transfer of information?

An important aspect of message passing is that *sending a message is much more costly than a flop*.

Parallel computing with MATLAB

Many of the advantages of MATLAB can be used in conjunction with parallel computing.

There are a number of parallel MATLAB libraries including the Parallel Computing Toolbox and the MATLAB Distributed Computing Server as well as the freely available `pMatlab` and `MatlabMPI`.

In MATLAB, it is convenient to think of *distributed arrays*, i.e., data that are split up and owned by different processes[2], as opposed to messages.

Distributed arrays can be used to handle the vast majority of parallel programming tasks.

They are easy to understand, require little code to use, and fit well with the array-centric nature of MATLAB.

In many ways, distributed arrays form an optimal core parallel programming model for working in MATLAB.

[2]A *process* can be thought of as an independent CPU.

Types of parallel decomposition

Typically the first step in designing a parallel algorithm is to decompose the problem into smaller problems that can be assigned to different processors to work on simultaneously.

There are two main kinds of decompositions for the purposes of parallelization:

- domain decomposition (data parallelism)
- functional decomposition (task parallelism)

In a shared-memory paradigm, a lot of the work is done by the compiler.

In message passing, these decompositions must be programmed explicitly.

Domain decomposition (or data parallelism)

A data parallel algorithm is *a sequence of elementary instructions applied to (different) data*.

In other words, a new instruction is initiated only after the previous instruction terminates.

The advantage is that there is a single flow of control.

Data are divided into pieces of *approximately the same size* and *mapped to different processors*.

Each processor works only on the portion of the data that it is assigned.

Of course, the processes may need to communicate periodically in order to exchange data.

Special cases of this include Single-Program-Multiple-Data (SPMD) or Single-Instruction-Multiple-Data (SIMD) techniques.

SPMD strategies are commonly employed in finite-difference / finite-element algorithms where processors can operate independently on large blocks of data and exchange only the much smaller shared border data at each iteration.

Beware that there may be parts of an algorithm that are simply **not parallelizable**; e.g., get user input or add two scalars.

These tasks are often handled by the server in the client-server paradigm (see below).

Fortunately, for many applications the non-parallelizable tasks consume a relatively small proportion of the overall effort.

Functional decomposition (or task parallelism)

When the data assigned to the different processors require greatly different lengths of time to process, domain decomposition will not be effective.

The performance of the code will be limited by the speed of the slowest process; i.e., this becomes a *bottleneck* for the computation.

Also, the remaining idle processes do no useful work.

In this case, functional decomposition (or task parallelism) may make more sense.

In task parallelism, the problem is decomposed into smaller tasks, and the tasks are assigned to the processors as they become available.

This way processors that finish quickly can be assigned more tasks.

Client-server paradigm

Task parallelism is typically implemented in a *client-server paradigm*.

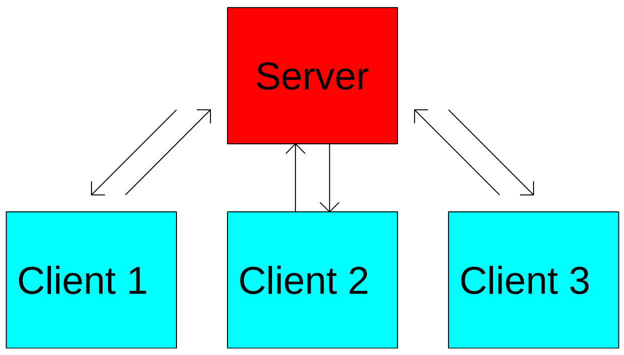


Figure 1: The client-server paradigm

A master process (the server) is in charge of allocating tasks to a number of slave processes (the clients).

The master process may also assign tasks to itself.

The client-server paradigm can be implemented at virtually any level in a program.

For example, to run a program with multiple inputs, a parallel client-server implementation might simply run multiple copies of the code serially with the server assigning the different inputs to each client process.

Once a processor finishes its task, it can be assigned a new one.

Task parallelism can also be implemented at a lower level within code.