3: Parallel Programming Concepts

John Burkardt
Information Technology Department
Virginia Tech

.....

FDI Summer Track V:
Parallel Programming

.....

https://people.sc.fsu.edu/~jburkardt/presentations/...
parallel_2008_vt.pdf

10-12 June 2008

Parallel Programming Concepts

Parallel Programming Concepts

The difference between 1,000 workers working on 1,000 projects, and 1,000 workers working on 1 project is **organization** and **communication**.

The key idea of parallel programming:

> *Independent agents, properly organized and able to communicate, can cooperate on one task.*

Grand Challenge Problems

Why do we need parallel programming?

One answer is Paris Hilton's motto:
Too much is never enough!

A more dignified answer is our pursuit of **Grand Challenge Problems**, fundamental problems in science and engineering whose solution is just becoming imaginable or practical with the highest performance computing resources.

Grand Challenge Problems

- Weather prediction
- Climate Modelling
- Design of pharmaceutical drugs
- Protein folding
- Human genome
- Oil recovery
- Subsurface water flow
- Simulation of turbulent flow
- Superconductivity
- Quantum Chromodynamics
- Astronomical simulation
- Analysis of combustion
- Nuclear fusion

Grand Challenge Problems

Grand Challenge Hardware

- memory - massive, with levels and multiple paths
- processors - more of them, and more power on each one
- communication - faster

Grand Challenge Problems

Grand Challenge Software

- new languages or extensions to old languages
- smarter compilers or smarter software
- numerical libraries
- user codes
- algorithms

Types of Parallel Processing

There are many classifications of parallel processing.
The useful distinction for us:

- **shared memory**, embodied in OpenMP.
- **distributed memory**, embodied in MPI.

Shared Memory - Multiple Processors

The latest CPU's are called *dual core* and *quad core*, with rapid increases to 8 and 64 cores to be expected.

The cores share the memory on the chip.

A single program can use all the cores for a computation.

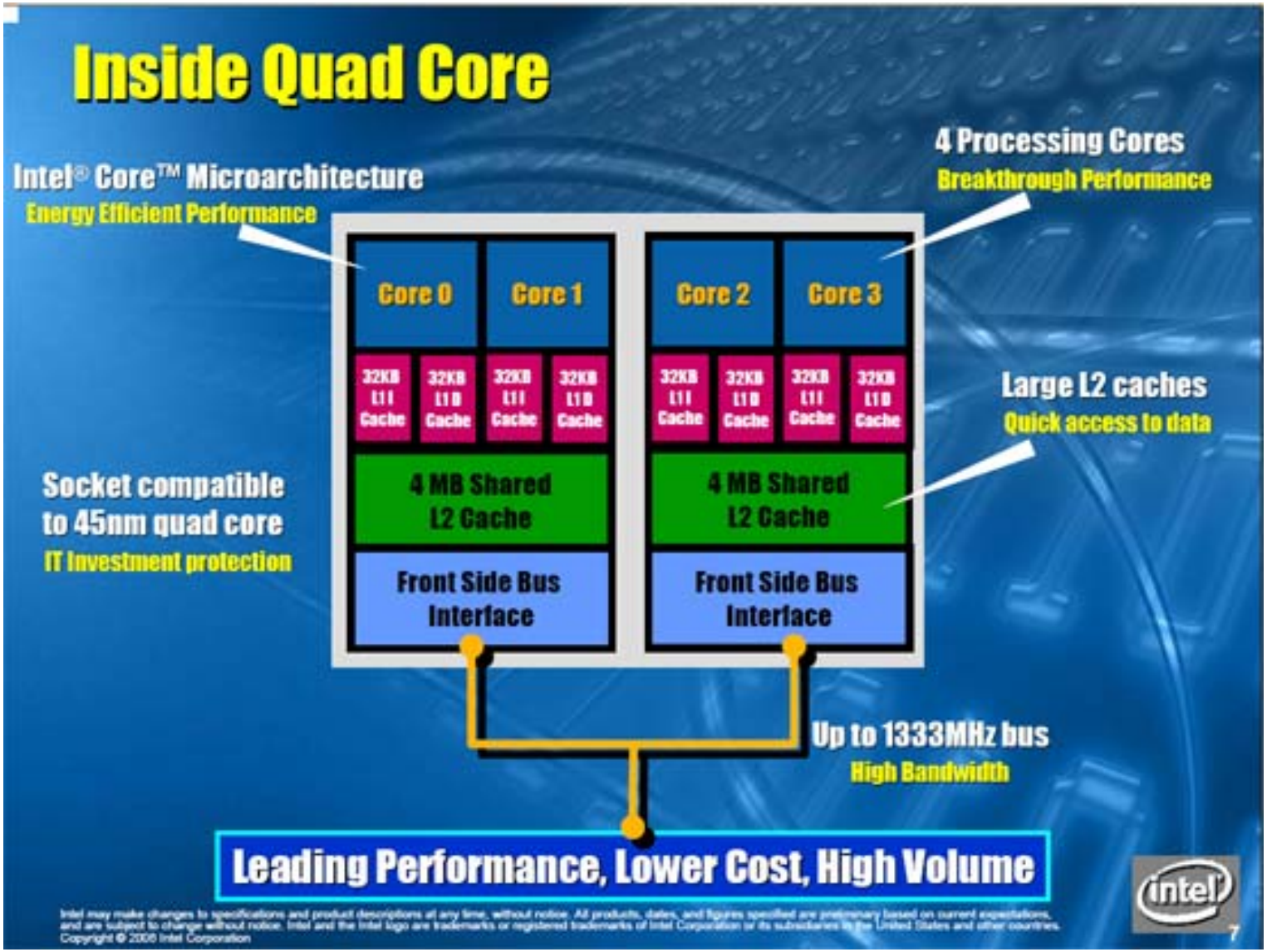
It may be confusing, but we'll refer to a core as a *processor*.

Shared Memory - Multiple Processors

If your laptop has a dual core or quad core processor, then you will see a speedup on many things, simply because the operating system can run different tasks on each core.

When you write a program to run on your laptop, though, it probably will *not automatically* benefit from multiple cores.

Shared Memory - Multiple Processors



Shared Memory - Multiple Local Memories

The diagram of the Intel quad core chip includes several layers of memory associated with each core.

The full memory of the chip is relatively "far away" from the cores. The cache contains selected copies of memory data that is expected to be needed by the core.

Anticipating the core's memory needs is vital for good performance. (There are some ways in which a programmer can take advantage of this.)

A headache for the operating system: *cache coherence*, that is, making sure the original data is not changed by another processor, which invalidates the cached copy.

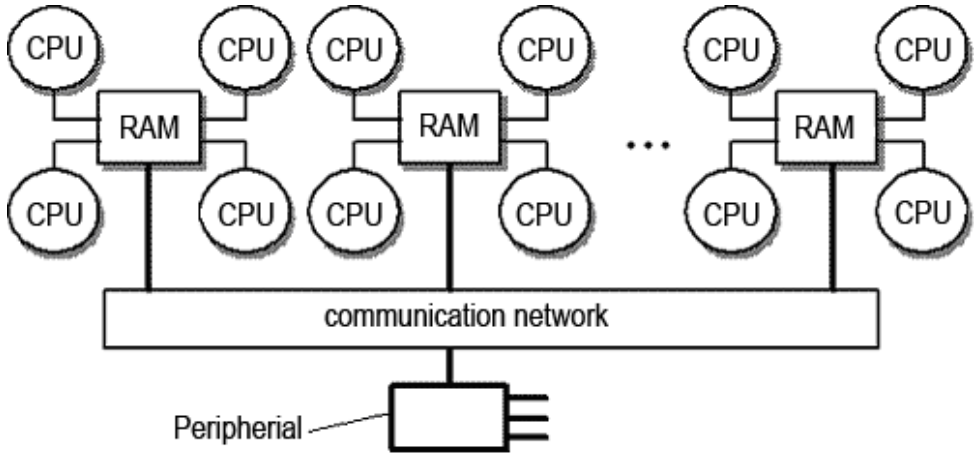
Shared Memory - NUMA Model

It's easy for cores to share memory on a chip. And each core can reach any memory item in the same time, known as **UMA** or "Uniform Access to Memory".

Until we get 8 or 16 core processors, we can still extend the shared memory model, if we are willing to live with **NUMA** or "Non-Uniform Access to Memory".

We arrange several multiprocessor chips on a very high speed connection. It will now take longer for a core on one chip to access memory on another chip, but not too much longer, and the operating system makes everything look like one big memory space.

Shared Memory - NUMA Model



Chips with four cores share local RAM, and have access to RAM on other chips.

VT's SGI ALTIX systems use the NUMA model.

Shared Memory - Implications for Programmers

On a shared memory system, the programmer does not have to worry about distributing the data. It's all in one place *or at least it looks that way!*

A value updated by one core must get back to shared memory before another core needs it.

Some parallel operations may have parts that only one core at a time should do (searching for maximum entry in vector).

Parallelism is limited to the number of cores on a chip (2, 4, 8?), or perhaps the number of cores on a chip *multiplied* by the number chips in a very fast local network (4, 8, 16, 64, ...?).

Shared Memory - Implications for Programmers

The standard way of using a shared memory system in parallel involves **OpenMP**.

OpenMP allows a user to write a program in C/C++ or Fortran, and then to mark individual loops or special code sections that can be executed in parallel.

The user can also indicate that certain variables (especially "temporary" variables) must be treated in a special way to avoid problems during parallel execution.

The compiler splits the work among the available processors.

This workshop will include an introduction to **OpenMP** in a separate talk.

Shared Memory - Data Conflicts

Here is one example of the problems that can occur when working in shared memory.

Data conflicts occur when data access by one process interferes with that of another.

Data conflicts are also called *data races* or *memory contention*. In part, these problems occur because there may be several copies of a single data item.

If we allow the "master" value of this data to change, the old copies are out of date, or *stale data*.

Shared Memory - Data Conflicts

A mild problem occurs when many processes want to **read** the same item at the same time. This might cause a slight delay.

A bigger problem occurs when many processes want to bf write or modify the same item. This can happen when computing the sum of a vector, for instance. But it's not hard to tell the processes to cooperate here.

A serious problem occurs when a process does not realize that a data value has been changed, and therefore makes an incorrect computation.

Data Conflicts: The VECTOR MAX Code

```
program main

integer i, n
double precision x(1000), x_max

n = 1000

do i = 1, n
  x(i) = rand ( )
end do

x_max = -1000.0

do i = 1, n
  if ( x_max < x(i) ) then
    x_max = x(i)
  end if
end do

stop
end
```

Shared Memory - Data Conflicts - VECTOR MAX

```
      program main

      include 'omp_lib.h'

      integer i, n
      double precision x(1000), x_max

      n = 1000

      do i = 1, n
        x(i) = rand ( )
      end do

      x_max = -1000.0

!$omp parallel do
      do i = 1, n
        if ( x_max < x(i) ) then
          x_max = x(i)
        end if
      end do
!$omp end parallel

      stop
      end
```

Shared Memory - Data Conflicts - VECTOR MAX

It's hard to believe, but the parallel version of the code is incorrect. In this version of an OpenMP program, the variable **X_MAX** is *shared*, that is, every process has access to it.

Each process will be checking some entries of **X** independently.

Suppose process P0 is checking entries 1 to 50, and process P1 is checking entries 51 to 100.

Suppose X(10) is 2, and X(60) is 10000, and all other entries are 1.

Shared Memory - Data Conflicts - VECTOR MAX

Since the two processes are working simultaneously, the following is a possible history of the computation:

1. **X_MAX** is currently 1.
2. P1 notes that **X_MAX** (=1) is less than **X**(60) (=10,000).
3. P0 notes that **X_MAX** (=1) is less than **X**(10) (=2).
4. P1 updates **X_MAX** (=1) to 10,000.
5. P0 updates **X_MAX** (=10,000) to 2.

and of course, the final result **X_MAX**=2 will be wrong!

Shared Memory - Data Conflicts - VECTOR MAX

This simple example has a simple correction, but we'll wait until the **OpenMP** lecture to go into that.

The reason for looking at this problem now is to illustrate the job of the programmer in a shared memory system.

The programmer must notice points in the program where the processors could interfere with each other.

The programmer must coordinate calculations to avoid such interference.

Distributed Memory - Multiple Processors

Distributed Memory - Explicit Communication

For several servers to cooperate in a distributed memory system,

- the program to run must be copied to each server
- certain synchronizations are needed ("all start", "wait for me", "all stop"
- some problem data must be divided, some duplicated
- programs must be able to "talk" to each other (pass results)

Because communication between servers is much slower than between a core and its memory, distributed memory computations must limit communication.

A Simple Approach

1. Divide a big computation into smaller tasks.
2. Give separate tasks to each processor.
3. Gather results (perhaps a single number from each processor) at the end.

Computations this easy are called *embarassingly parallel*.

They are characterized by requiring little or no intermediate communication between processors.

"Embarassingly Parallel" Applications

Estimate the integral of a function by evaluating it at regularly spaced or random points.

Estimate the evolutionary relationship among several species by considering all possible evolutionary trees, weighted by their probability.

Search an online database of EMAIL messages for all occurrences of the phrase "Sell my Enron shares!"

Does some binary string of length N causes a given circuit to produce the output value of 1? *(The circuit satisfiability problem)*.

Most Applications are NOT Embarrassingly Parallel

Most parallel applications are not easily broken up into almost independent subtasks.

In domain decomposition problems, a physical region is divided into subregions, each associated with a processor.

If the program models the flow of heat throughout the region, then at each boundary between subregions, the associated processors will need to communicate the updated boundary values.

To sort N numbers between 0 and 1, processor I could sort the I-th part of the data, then keep the values between I/P and I+1/P, sending the rest to the others.

Distributed Memory - Programmer Implications

The programmer must distribute data among the processors. What must everyone have? What big vectors can be broken into separate pieces on separate processors?

The programmer must arrange for data to move between the processors.

The programmer must distribute the work among the processors.

The programmer must synchronize the execution of the work.

In particular, if one program sends data, the recipient program must be ready to receive it.

If a program needs data that is not ready, it must be told to wait.

Distributed Memory - Programmer Implications

Parallel programming on distributed memory machines can be implemented using **MPI**.

MPI allows the user to write a single program, in C/C++ or Fortran.

The same program will run on all the cooperating processors. Each processor is given a distinct ID number; this allows it to decide what parts of the program it must execute.

Separate processors can communicate using their ID numbers.

MPI includes methods of sending and receiving data, synchronizing execution, and combining results.

This workshop will include an introduction to **MPI** in a separate talk.

Distributed Memory - Deadlock

As an example, we solve a problem with 100 values x along a line, We use domain decomposition, and two processes, so process P0 has x(1) through x(50) and process P1 has x(51) through x(100).

P0 always needs a current copy of x(51), and P1 a copy of x(50).

In **MPI**, processors communicate using *messages*.

A user may require that messages are acknowledged to have been received, before the next command is executed.

A message can only be received if the receiving process is executing the command "Receive". Otherwise, the message waits in some temporary location.

This can be a recipe for deadlock.

Distributed Memory - Deadlock

```
===The P0 side==============+====The P1 side============
                           |
Initialize X(1)...X(50)    |  Initialize X(51)...X(100).
                           |
Begin loop                 |  Begin loop
                           |
  Send X(50) to P1      ==>|<== Send X(51) to P0
                           |
  When receipt confirmed,<==|==> When receipt confirmed,
                           |
  Receive X(51) from P1 <==|==> Receive X(50) from P0
                           |
  Update X(1)...X(50).     |    Update X(51)...X(100).
                           |
Repeat                     |  Repeat
```

Conclusion - Don't Panic!

A mental image of how shared memory systems work is to imagine that a pair of twins are working on your calculation.

All of the data is written on a whiteboard that both can see.

Whenever the computation involves a loop, one twin does the even iterations, and one the odd.

During these calculations, each twin can make some private calculations on a notepad that the other twin doesn't see.

Each twin can go up to the whiteboard at any time and change numbers there as the result of a calculation.

Conclusion - Don't Panic!

An appropriate idea of how distributed memory systems work is to imagine that you have a program running on a single computer, working on part of a problem.

Every now and then, instead of printing out the value of some number it has computed, it "sends" that value to another computer.

Every now and then, instead of reading data from the user, it "receives" information from another computer.

Except for these occasional chats with other computers, the computation proceeds as usual.

Conclusion - Don't Panic!

Parallel programming is a very deep pool, but we can take our time getting into it!

If you've already done some programming, then you can understand the important fundamentals of shared and distributed memory systems by simple extensions of what you already know.

Once we see some examples of parallel programs, I hope you will agree that they are based on simple ideas.

There is really nothing magic going on!

Conclusion - Don't Panic