CPS311 Lecture: Hardware Support for Parallelism

Last revised November 20, 2019

Objectives:

1. To introduce the basic idea of parallelis
2. To introduce Flynn's taxonomy
3. To introduce various SIMD approaches (Vector processors, MMX)
4. To introduce various MIMD approaches using multiple CPUs (Shared memory - UMA and NUMA - and message passing)
5. To introduce multicore CPU's
6. To introduce the use of hardware-supported threads
7. To introduce Hyper-Threading
8. To introduce the use of GPU's as general purpose processors, including CUDA

Materials:

1. Projectable of Moore's Law (as of 2015)
2. Projectable of Moore's Law and other trends (as of 2011)
3. Projectable of Chip Real Estate Tradeoff
4. Projectable of Amdahl's law for various values of P
5. Projectable of Null Figure 9.5 (Crossbar switch)
6. Projectable of Null Figures 9.6, 9.7 (Interchange switch)
7. Projectable of Intel Quick Path Architecture
8. Projectable of Null Figure 9.3 (Message passing structure alternatives)
9. Projectable of CUDA Flow
10. Example of parallelizing several ways- handout + ability to demo on Linux

I. Introduction

A. Today's computers are orders of magnitude faster than those of a generation ago. Nonetheless, there is a continuing demand for ever faster computers.

Why? ASK

1. Reasons fall into two broad categories:

    a. The need to handle a greater volume of task per unit of time.

        Example: Various kinds of server systems.

    b. The need for greater speed to allow performing computations that otherwise would not be feasible, or could not be done as well as would be desirable.

        Example: Weather forecasting relies on computer simulations that model a region (including the atmosphere above it) by breaking it into a grid of smaller gridboxes, which are initialized to observed data at the start of the period being modeled. Then the process moves forward by calculating new values for each gridbox in terms of its own values and those in its neighbors at discrete time intervals.

        1. The smaller the gridboxes and the shorter the time steps, the more accurate are the predictions derived from the model.

        2. Of course, such a model is useless if the time needed to perform the computation is too long. (Of what value is an accurate forecast of tomorrow's weather if it is not available until the day day after!)

        3. Speed of computation effectively limits the precision of the model in terms of the side of the gridboxes and the size of the time intervals. (For example, current models are forced to use gridboxes that are actually larger than features like individual clouds)

        4. There are numerous potential applications for simulation in the sciences that are just becoming or are not yet feasible due to the need for faster hardware.

2. Broadly speaking, there are two general approaches to this issue:

    a. Build machines that can perform a single task more quickly (faster computers).

i. For much of the history of computing, a trend has held that was first described by Gordon Moore (a co-founder of Intel) in 1965: "The number of transistors that can be inexpensively placed on an integrated circuit doubles approximately every two years". [Moore's original statement had the doubling occurring every year, but doubling every two years fits the historic data better]

   (This is called Moore's Law)

ii. Though the rate of doubling has been closer to once every two years, the technological advances described by Moore's law have yielded increased complexity for CPU chips for five decades

   PROJECT: Moore's Law as of 2015 (http://education.mrsec.wisc.edu/SlideShow/images/computer/Moores_Law.png)

iii. However, clock speed, that was increasing dramatically along with transistor count has been flattened in recent years

   PROJECT: Other trends graph (source: Article "The Free Lunch is Over" - http://www.gotw.ca/publications/concurrency-ddj.htm accessed 6/3/11)

iv. We discussed another approach when we talked about such strategies as pipelining/superpipelining and the use of superscalar architectures.

   (a) This approach is called Instruction-Level-Parallelism. The idea is to make the <u>average</u> time for executing an instruction smaller by doing portions or all of the same instruction in parallel.

   (b) This is shown as the "performance per clock (ILP)" line on the graph just projected. Note that it too is flattening out - these strategies are limited by inherent dependencies between successive instructions.

b. The growingly-important alternative is to build systems that do multiple things at the same time: full parallelism - which is the topic we are looking at now.

   i. In this series of lectures we are focussing on larger-scale forms of parallelism, through such strategies as building multicore computers or using multiple computers.

   ii. One very important difference between what we are looking at today and ILP is that the various ILP strategies are designed to be invisible to the programmer - i.e. the hardware or the compiler ensures that it appears to the programmer that the instructions in the program are being executed in order, even if they are not. But with full parallelism, it becomes necessary for the programmer to be consciously aware that code is being executed in parallel.

   This leads to a need to learn to think differently about programming - to use "parallel thinking".

c. There is necessarily a tradeoff between these two approaches.

   i. For example, for a system with a single CPU chip (most systems today), the choice is whether to use the limited "real estate" on the chip to build a faster processor or to build multiple processors (or portions of a processor) that are slower but get more total work accomplished than a single, faster processor could do - or perhaps to build fewer but faster or more but slower cores.

   PROJECT: Chip Real Estate Tradeoff

   (a) The figure on the left shows a superscalar CPU with four sophisticated ALU's, a large cache memory, and a fairly complex control unit.

   (b) The figure on the right shows a Graphics Processor (GPU) with 8 much simpler processors - each with its own small

cache and 16 much simpler ALU's. (We will see shortly that GPU's, though originally designed for graphics operations, are now proving to be very powerful devices for handling certain kinds of general-purpose computing problem.)

ii. It is also possible to build systems using multiple CPU's. In this case, the tradeoff may be between using one (or a few) very fast CPU's or a larger number of slower but cheaper CPU's. In many cases, it turns out the latter approach actually yields the most economical solution.

B. When we look at parallelism, we need to realize that tasks differ widely in terms of how useful parallelism is for speeding them up.

1. Recall definitions that we discussed in CPS222:

a. The speedup of a parallel computation is the ratio of the time taken for doing the computation sequentially to the time taken for doing it in parallel.

b. Linear speedup means that we can achieve a speedup of n when using n processors - presumably the best speedup possible.

2. Suppose we have a task that takes a certain amount of time on a single processor. What will the speedup be if we perform the same task on a parallel system with n processors operating at the same time?

a. If the task is perfectly parallelizable, the speedup would be n.

Example: if one person takes four days to paint a house (one day on each side), then potentially the same task could be done by four people in just one day (a 4:1 speedup.)

b. On the other hand, there are tasks for which nothing would be gained by the use of parallelism

Example: consider the task of producing a depth-first ordering of the nodes of a graph, given a representation such as an adjacency matrix

or adjacency list. It can be proved that such a task is inherently sequential - i.e. there is no way way to improve the time complexity to better than O(n) by the use of parallelism.

c. Of course, most tasks like somewhere between these two extremes - i.e. part of the task is parallelizable and part of the task must be done sequentially.

   Example: even the task of painting a house requires some preliminary work such as buying the paint.

3. Recall that the potential speedup for a given task using parallelism is strongly dependent on how much of the task is parellelizable and how much is not. A formula known Amdahl's law says that in a task where S is portion of the total time needed for the task that is inherently sequential and P is the portion of the total time that is parallelizable (so S + P = 1), then the speedup resulting from using n processors will be

$$\frac{1}{(1 - P) + P/n}$$

   Example: Suppose 95% of the time needed for a given task is spent doing work that can be parallelized but 5% is spent on inherently sequential work. If the parallelizable part is speeded up by a factor of 10 (say by using 10 processors), then the overall speedup is

$$\frac{1}{(1 - .95) + (.95)/10} = \frac{1}{.05 + .095} = 6.9$$

   But now suppose 50% of the time needed is spent doing work that can be done using parallelism, and 50% of the time is spent on inherently sequential work. If the parallel part is speeded up by a factor of 10, the overall speedup is only

$$\frac{1}{(1 - .50) + (.50)/10} = \frac{1}{.50 + .05} = 1.82$$

a. A corollary is that the maximum possible speedup for the task (obtained if an infinite speedup were applied to the parallelizable portion) is

$$\frac{1}{(1 - P)}$$

Example: for the first case cited above, the maximum possible speedup is 20:1, which would be achieved if the time for the non-parallelizable portion remained the same, while the time for the parallelizable part dropped to 0. But in the second case, the maximum possible speedup is only 2:1.

b. The implications of Amdahl's law can be seen graphically.

PROJECT: Graph showing Amdahl's law results for various values of P

c. A further issue to bear in mind is that even the parallelizable part of a task may inherently limit the degree of speedup.

It is not always the case that a task consists of a perfectly parallelizable part and a perfectly sequential part - sometimes there is a portion that is partially but not fully parallelizable.

Example: We have argued that the task of painting a house could be speeded up by a factor of four by using 4 painters. But what would happen if we tried to use 1000 painters?

3. Parallelism is most obviously usable in two cases:

a. Where we have a task that consists of numerous independent parts that can be done in parallel.

Example: various sorts of server tasks, where the work to be done for each client is independent of the work being done for other clients

b. Where we have a task where the bulk of the time is spent on an easily parallelizable portion, with some purely sequential steps at the beginning and/or end.

Example: simulations such as the weather forecasting model we discussed earlier, where

i. We initially set up the data - a sequential task

ii. The bulk of the time is spent on the computations for each gridbox, which can be assigned to separate processors with minimal interaction only with immediate neighbors, and therefore can be done in parallel.

iii. A final sequential process "harvests" the results.

c. A major challenge is discovering the potential for parallelism in a task that at first looks to be sequential in nature, but does have some part(s) that could be speeded up by the use of parallelism.

4. Because we are rapidly moving toward reliance on parallelism for speedup, it is vital to learn to "think parallel" as we look at problems. (Recall our discussion of parallel algorithms in CPS222)

5. Sometimes, applications are classified based on the possibility of breaking them into parallelizable subtasks

a. An application may be inherently sequential - it has no parallelizable subtasks.

b. An application may allow fine-grained parallelism - which means that part of it can be broken into subtasks, but those subtasks need to coordinate/communicate frequently (many times per second)

c. An application may exhibit coarse-grained parallelism- which means that part of it can be broken into subtasks, but those subtasks need to coordinate/communicate less frequently.

d. An application may be classified as embarrasingly parallel - which means that part of it can be broken into subtasks that rarely or never need to coordinate/communicate with each other.

C. When we talk about parallel systems, there are actually two different kinds of parallel processing.

1. In data parallelism, the same operation is done on different data at the same time.

   Example: The bitwise and, or, xor, and nor instructions found on MIPS - and similar instructions found on most other computers - perform a logical operation (and, or ...) on 32 or 64 pairs of bits at the same time. The computation for each bit is independent of the others, but all the bits participate in the same computation at the same time.

   (We will look at more sophisticated examples of this kind of thing - that work on full numbers instead of individual bits in parallel - below)

2. In task parallelism, different operations are done on the same data at the same time.

   Example: Pipelining is a form of this. At any given time, the different portions of the pipeline are performing different operations on the same data stream.

D. Before looking at various ways hardware can be designed to support parallelism, we want to look at a system for classifying parallel systems, known as FLYNN'S TAXONOMY. This system goes back to the 1960's, but still has considerable relevance today.

   Flynn's taxonomy classifies systems by the number of instruction streams and data streams being processed at the same time.

1. Computers of the sort we have discussed up until now - i.e. single processor computers - are classified as SISD. A SISD machine has a Single Instruction Stream and a Single Data Stream - it executes one sequence of instructions, and each instruction works on one one set of operands.

2. A SIMD machine has a Single Instruction Stream and Multiple Data Stream - it executes one sequence of instructions, but each instruction can work on several sets of operands in parallel. (We will look at several approaches that fall into this category shortly).

3. A possibility that doesn't really exist except in very special cases is MISD (multiple instruction stream, single data stream - i.e. one stream of data passes through the system, but each item of data is operated on by several instructions.

4. A MIMD machine has Multiple Instruction Streams, each operating on its own Data Stream.

   a. One variant of this approach is the use of a multi-core CPU

   b. Another variant is to use multiple separate CPU's. This is commonly known as a multiprocessor system.

      There are a number subcategories under this general heading that we will consider later.

      (Note: sometimes the term MIMD is reserved for this latter approach)

   c. The most frequently used approach to MIMD is given the special name SPMD - Single Program Multiple Data. This is a multiprocessor in which all the processors execute the same program but may do different computations because of the presence of conditional branches in the program.

      One place this is increasingly seen is the use of a (possibly multi-core CPU) in conjunction with an array of graphics processors utilized for general-purpose computation.

II. **SIMD Approaches**

A. One class of SIMD machines is the class of Vector processors. Vector processing is combined with other strategies in the largest supercomputers; but most desktop/laptop CPU's include vector processing instructions that incorporate the basic idea into the ISA of a conventional CPU; and some video game systems utilize vector processing.

1. A vector may be a one-dimensional sequence of values, or a one-dimensional mapping of a higher-dimensionality structure.

   Example: the 2-dimensional array

   1 2 3
   4 5 6

   might be represented by the vector 1 2 3 4 5 6

2. Common operations on vectors include things like adding, subtracting, multiplying, or dividing corresponding elements of two vectors to produce a third vector result

   Example 1 2 3 4 5 6 + 7 8 9 10 11 12 = 8 10 12 14 16 18

3. Vector processors are provide support for performing operations like these on vectors. Most vector processors have two distinct hardware features:

   a. a set of vector registers, each of which can hold some number of values comprising a vector.

      (Actually, some vector processors have been built that allow vector operations to be performed directly on operands in memory, avoiding the need for these)

b. a pipelined ALU that can start a new element of a vector through the ALU on each clock. Thus, to add two 1000 element vectors, the processor needs 1000 + a few startup clock cycles.

4. A vector processor has two different kinds of instruction in its instruction set:

   a. Ordinary scalar instructions that operate on a single set of scalar operands to produce a single scalar result.

   b. Vector instructions that carry out an operation on a set of vector operands to produce a vector result.

      1. Special load store instructions support copying a vector between memory and a vector register.

         (a) Of course, if the vector is very large, it might need to be processed due to limitations on the number of elements a vector register can store.

         (b) If the processor allows vector operations to access memory directly, these instructions may not be needed.

      2. Vector computations like vector addition, which pipe entire vectors through the ALU, so that the operation in question is applied to corresponding elements of each vector. Once the pipeline is filled, it is usually possible to produce one result per clock period.

         Example: Suppose a is a vector of n elements and we want to double each element

      3. On a conventional CPU, this might be done by a loop like the following

```
for (int i = 0; i < n; i ++)
  a[i] *= 2;
```

         Which would require execution time n x the time for one loop iteration.

4. On a vector processor, this could be done by as little as one instruction, which would need to specify (among other things) the starting address in memory of the vector and the number of elements in the vector

   • On the first clock cycle, the vector processor would start `a[0]` through the pipe.
   • On the second clock cycle, the vector processor would start `a[1]` through the pipe, while `a[0]` would move on to stage 2 of the pipe.
   • On the third clock cycle, the vector processor would start `a[2]` through the pipe, while `a[0]` would move on to stage 3 of the pipe and `a[1]` would move on to stage 2.
   ....
   • Eventually, after as many clocks as there are stages in the pipe, `2*a[0]` would emerge from the pipe and be stored back into memory.
   • On each subsequent clock, one doubled value would emerge from the pipe while another would be started, until, on the (n + # of stages in the pipe -1) th cycle the final doubled value would emerge from the pipe and be stored back into memory.

   Thus, the total execution time would be n + s - 1 clocks (where s is the number of stages in the pipe) - which would be significantly less than n * number of clocks needed for a loop iteration.

5. Actually, this example assumes that the vector processor can operate directly on values stored in memory. Historically, that was true of some processors, but most required that the vector first be loaded into a vector register, and that the result be placed in a vector register from which it is stored back into memory.

   a. The instructions for loading and storing the vector registers might themselves be pipelined, or it might be possible to load/store an entire register from/to adjacent locations in memory with a single instruction.

   b. In this case, though, a loop would still be needed if the size of the vector exceeded the size of the vector registers - as would

often be the case. But the number of iterations needed would only be (# of elements in vector) / (vector register size), rather than (# of elements in the vector).

B. As an example of vector processing instructions that are part of the ISA of a conventional CPU, we will look at the Streaming SIMD Extensions (SSE instructions) that have been part of the x86 architecture since 1999. (Actually, we will look at SSE4, which is the most recent update of this instruction set - though it itself has slight updates called SSE4.1, SSE4.2)

1. x86 chips include 8 registers (xmm0 through xmm7) to support SSE. Each is a 128 bit register that can be used to hold any one of the following
   a. A vector of 16 8-bit bytes
   b. A vector of 8 16-bit shorts
   c. A vector of 4 32-bit integers or 32-bit floats
   d. A vector of 2 64-bit doubles
2. x86 chips support the following instructions (among others)
   a. Load an mmx register from a series of locations in memory
   b. Store an mmx register into a series of locations in memory
   c. Copy values between mmx registers
   d. Perform an arithmetic operation (+, -, *, /) between values in two mmx registers. These instructions can operate on any kind of vector that can be stored in an mmx register - e.g. 16 one-byte integers or 8 two-byte integers or 4 four-byte integers or floats ..
   e. Perform an arithmetic operation (+, -, *, /) between the values in an mmx registers and a scalar.

f. Compare the values in two mmx registers, producing a vector of booleans

g. ....

3. An earlier - and more limited - part of the x86 ISA is something called MMX (multi-media extension). The name comes from the fact that media processing often calls for doing the same operation on 1000‘s of values - e.g. processing all the samples in a sound or all the pixels in a picture.

   Though still supported in current implementations of x86, in practice SSE is used instead.

C.SIMD is most useful for embarrassingly parallel tasks

1. This is why specialized vector processors were most common in support of applications like weather forecasting or scientific simulations.

2. Facilities like SSE and MMX are mostly used for certain specialized tasks like those related to graphics / media manipulation.

III. **Multiprocessors**

A. Further parallelism can (beyond what we have already discussed) can be achieved by using multiple complete CPU's.

1. Sometimes the Flynn classification MIMD is reserved for such systems, though - as we shall see - multicore CPUs and the use of GPGPU's can also be considered instances of this classification. (The Flynn taxonomy is dated.)

2. True MIMD machines are distinguished from computer networks (which they resemble in having multiple CPU's) by the fact that in a network the cooperation of the CPU's is occasional, while in a MIMD machine all the CPU's work together on a common task.

B. MIMD systems are further classified by how the CPU's cooperate with one another. MIMD systems can be either based on SHARED MEMORY or on MESSAGE PASSING.

1. In a shared memory system, all the CPU's share physical memory (or at least some portion of it) in common.

   a. They communicate with one another by writing/reading variables contained in shared memory.

   b. Actually, it is not necessary for all of memory to be shared. Sometimes, each CPU has a certain amount of private local memory, but each also has access to shared global memory.

   c. A key feature of such a system is the use of a single address space for shared memory - i.e. if address 1000 lies in shared memory, then it refers to the same item, regardless of which processor generates the address.

d. Contention for access to shared memory limits the number of processors that can be connected in such a configuration. (Generally just a few processors).

e. We will say more about shared memory systems shortly.

2. In a message passing based system, each CPU has its own memory, and CPU's cooperate by explicitly sending messages to one another.

a. Thus, the different CPU's are connected to one another by some form of network. (But they are considered a single system because they are working cooperatively on a common task.)

b. One kind of system in this category is called MPP - massively parallel processing. Such systems may have 100's or 1000's of computers connected by a network and working cooperatively on a common task.

C. Further comments about shared memory systems.

1. Three further variations are possible in a shared memory system:

a. In a Symmetric Multi-Processing Systems (SMP), all the processors are identical, and they run under the control of a single operating system. (A multicore computer might be considered an instance of this.)

b. In a Uniform Memory Access system (UMA), the processors may or may not be identical, and each runs its own operating system, but the time needed to access a given location in memory is the same, regardless of which processor accesses it.

c. In a Non-Uniform Memory Access system (NUMA), a given processor may have faster access to some regions of memory than others. This is usually a consequence of different processors "owning" different parts of the overall memory, so that some accesses are more efficient than others.

2. In any case, some mechanism is needed for SYNCHRONIZING accesses to shared memory.

   a. Synchronization mechanisms were discussed in detail in CPS221. Recall that, at the heart of most such mechanisms is the idea of LOCKING a data item so that one processor has exclusive access to it during the performance of an operation.

   b. We won't repeat the discussion here.

3. Another key issue is how processors and shared memory are CONNECTED.

   a. Conceptually, the simplest connection structured would be a common bus. However, the number of modules that can be connected to one bus is limited in practice by problems of BUS CONTENTION, since only one processor can be using the bus at a time.

   b. It may therefore be desirable to break up the shared memory into a collection of independent modules, such that two processors may access two different modules at the same time. In this case, we must consider how the various CPU modules and memory modules are connected.

   c. Two alternative connection structures that might be used.

      1. A structure based on the use of a crossbar switch, which allows any CPU to connect directly to any memory module.

         PROJECT - Null Figure 9.5

      2. A structure based on the use of interchange switches.

         PROJECT - Null Figures 9.6, 9.7

   d. In a NUMA system, different processors may "own" different portions of memory - though there is still a single address space.

      PROJECT: Intel Quick Path Architecture

4. Of course, each processor will still have its own cache (or caches). Thus, the number of times a given processor actually accesses the shared memory is minimized. But now a new problem arises, because when one processor updates a shared variable, copies of the old value may still reside in the caches of other processors.

    a. This problem is called the CACHE COHERENCY problem.

    b. One possible solution is to use SNOOPY CACHEs, in conjunction with write-through caching.

        1. In addition to responding to requests for data from its own processor, a snoopy cache also monitors memory connection traffic and listens for any memory write being done by another process to a location it contains.

        2. When it hears such a write taking place, it either updates or invalidates its own copy.

5. Given the issues of synchronization and cache coherence, one might ask why one would use a shared memory instead of a message passing system, The basic answer is that it minimizes overhead for inter-processor communication: the time needed to read or write a shared variable is much less than the time needed to send or receive a message.

D.Further Comments About Message Passing Systems

1. In a shared memory system, a program does not need to be conscious of where data it accesses actually resides, since there is a single address space shared by all processors. (Though accessing some locations may take more time than others in a NUMA system - something which the hardware takes care of)

2. But in a message passing system, a program accesses data held by another processor by sending a message to it and awaiting a returned result. Thus, the programmer is quite conscious of accessing data belonging to another system.

3. One key issue in message-passing architectures is how the processors are connected. Recall that we discussed some possibilities in CPS222.

   PROJECT: Null figure 9.3

4. The most widely-used software infrastructure for such systems is MPI - Message Passing Interface.

   a. MPI is the defacto standard for scientific computing.

   b. It supports programming in Fortran or C.

   c. MPI provides abstractions that handle network communication issues, so the programmer can focus on the problem to be solved, not the mechanics of solving it.

   d. MPI is discussed much more fully in our Parallel Processing elective.

E. Another way to classify parallel systems is in terms of hardware infrastructure.

1. Historically, parallel systems were specially designed to be parallel systems, often making use of the fastest (and therefore most expensive) hardware available.

2. However, it is also possible to build parallel systems using relatively expensive, off the shelf computers - using quantity to compensate for lack of individual speed.

   Example: the BEOWULF project is a software infrastructure that can turn a heterogenous network of commodity devices (PCs, workstations, etc.) into a parallel cluster.

3. It is also possible to build parallel systems using workstations that are used for other purposes as well as serving as part of a parallel system - so called COW (Cluster of Workstation Systems).

4. Finally, there are projects which utilize donated cycles from huge numbers of systems during time when the system is otherwise idle. Such a system might make use of a special screen saver that performs computation on data accessed over the network when the machine on which it is running is otherwise idle, performing tasks ranging from SETI to number-crunching in support of cancer research.

F. As we noted earlier, while it is possible in principle for each of the computers in a MIMD system to be running a different program, it is often the case that all the computers actually run copies of the same program, but working on different data. This approach is sometimes called SPMD (single program multiple data) and resembles SIMD.

G. This area is explored in much greater depth in CPS343 - Parallel and High-Performance Computing.

IV. **Multicore CPUs**

A. A multicore chip consists of two or more complete CPU's (called cores) on a single chip (either on a single die, or two or more dies in the same package.)

1. Each core has its own ALU and control units, plus its own L1 cache and possibly L2 cache as well. This means that all cores on a chip can be executing simultaneously as long as the instructions and data each core needs are present in its cache(s).

2. Multiple cores on the same chip share a single bus interface. This means that if two cores need to access code or data that is not in their own cache, one will have to wait for the other to finish making use of the shared resources.

B. The first commercial multicore processors were 2-core chips released in 2001. Since then, chips with up to 6 cores have become commercially available, while chips with as many as 100 physical cores have been produced in research laboratories (and both numbers will grow in the future.)

C. But how do we make effective use of a multicore CPU in, say, a desktop or laptop computer?

   It turns out that the operating system needs to be designed to make use of the multiple cores.

   1. The simplest way is to assign different applications that are running at the same time to different cores - e.g. a music player might be running on one core at the same time a word processor is running on another.

      a. This is similar to the concept of time-sharing we discussed in operating systems, except now the different task are not competing for processor time.

         (In some cases, this could be quite beneficial - e..g I used to use a single core computer at home that became quite unresponsive when Time Machine backup is running.)

      b. But this does nothing to improve the performance of the system for an individual application.

   2. A better approach is possible if an individual application is multithreaded, provided threads are managed by the operating system (rather than by a user thread library that runs in a single kernel thread). In this case, the operating system can assign different threads to different cores.

      Example: suppose one is playing a movie as it is being downloaded. Typically, at least two threads are involved - a "producer" that is downloading the movie, and a "consumer" that is playing it. Each thread could be assigned to a different core.

D. In order for an individual program to make use effective use of a multicore CPU, it needs to partition its work into smaller units that can potentially be scheduled to run concurrently on multiple cores.

1. One approach is to break the program up into threads (assuming threading is managed by the operating system), with different threads running on different cores.

2. Other approaches are possible - e.g. Mac OS Snow Leopard (10.6) and later versions, and iOS4, utilize a mechanism called "Grand Central Dispatch" (GCD) - currently supported for programs written in C, C++, or Objective -C.

    a. To write a program that uses GCD, the programmer writes it as a collection of "blocks".

    b. To operating system maintains a one or more queues for each application that hold blocks that are available to run, and schedules a block to run when the resources to run it become available.

    c. This means that two blocks may be running at the same time on a dual core machine, or up to four on a quad core machine ...

        1. But the programmer does not need to know how many cores will be available on the machine the program will run on - the programmer just places runnable blocks in the queue and the system runs each block when it becomes possible to do so.

        2. It is also possible to designate a block as a handler for a given type of event (e.g. in a GUI); the block only becomes runnable when the event it handles occurs.

        3. It is also possible to create sequences of blocks that need to be run in a certain order.

E. For embarrassingly parallel tasks, though, a much higher degree of parallelism than that afforded by multicore chips alone is often desirable.

V. Hardware-Supported Threading

A. We have discussed the use of multiple threads managed by the operating system as a strategy for better exploiting the capabilities of a multicore system. Now we look at another use of threads.

B. Ideally, a program would run at the maximum speed allowed by the processor - e.g. completing one instruction per clock on a pipelined CPU, or perhaps several per clock on a superscalar machine. But reality is often far different from this.

1. Recall that memory accesses can take many clock cycles.

   a. L1 cache is designed to run at the same speed as the CPU - so an item that is needed from memory is in L1 cache, it can be accessed without any loss of time.

   b. But L2 cache is about 10 times slower than the CPU, and main memory can be over 100 times slower.

   c. Thus, when a program needs to access something that is not in L1 cache, it will be delayed by perhaps the time needed for hundreds of instructions.

      Example: On a system with a 3 GHz clock with a main memory access time of 60 ns, a main memory access requires 180 clocks plus whatever time is needed for bus overhead.

2. Actually, at fairly low cost in terms of increased complexity, it is possible to design hardware that allows more than one thread to be running on a given core at a given time. (Threading needs to be managed by the hardware, not software.) For example, one possibility is to design a processor so that it can be working on two or more threads at the same time.

   a. At any one time, only one thread will actually be running, of course.

b. But each thread will have its own set of registers and its own stack, so context switching from one thread to another can be done with no additional accesses to memory needed, and switching between threads can be managed by hardware, rather than software.

c. If the currently-running thread performs an operation that requires a delay - e.g. access to a memory location not in L1 cache, or an IO operation, then the processor can instantly switch to running the other thread, so as not to waste any cycles.

d. Since it is possible that both threads might need time-consuming operations, it may be desirable to design the processor to support more than 2 threads - perhaps 4 or 8 or more

3. Intel refers to its implementation of this approach as Hyper-Threading (HT). When Hyperthreading is turned on for a chip, it appears to the operating system to have twice as many virtual processors as there are physical cores, and the operating system is allowed to schedule as many threads as there are virtual processors. (However, Intel recommends turning HT off via the Bios when using an HT-enabled chip with an operating system that has not been optimized to make the best use of this technology, lest overall performance actually suffer.)

4. In practice, in most implementations this is limited to 2 threads per core, but up to 8 threads per core have been supported for this strategy with some other chips.

VI. General-Purpose Use of Graphics Processors

A. Graphics operations such as translation, rotation, and texturing are good candidates for use of parallel processing because they typically involve performing the same basic computation for each of the thousands of pixels that make up a screen image.

To support fast graphics. video cards typically incorporate a graphics processor chip that contains scores or hundreds of simple processing elements.

PROJECT: Chip Real Estate Tradeoff (again)

B. While GP'U's were initially designed to support the computations needed for sophisticated graphics, there has recently been a movement toward using them for other sorts of computation as well.

1. This model is sometimes called GPGPU - General Purpose Computing on a GPU.

2. It is well-suited to applications that require a large number of relatively simple computational operations - i.e. "number-crunching" sorts of applications that occur in the natural sciences, weather forecasting, mathematical modeling, and game physics.

C. One GPGPU architecture is CUDA (Compute Unified Device Architecture) that was developed by NVidia to support general-purpose computation using their graphics processor chips.

1. The first version was released in 2007.

2. It is currently available for Windows, Linux, and MacOS.

3. The CUDA software and documentation is free - but it is designed to support using chips manufactured by NVidia.

4. The basic architecture of CUDA looks like this

   PROJECT: CUDA Flow Diagram

5. Our Linux workstations have sophisticated graphics cards; but even ordinary laptops frequently have GPUs that can be used for parallel computation.

   SHOW on any workstation (except amos) lshw - look for data on display, then lookup Video Card on NVidia web site: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/DS_NV_Quadro_K2000_OCT13_NV_US_LR.pdf - note number of CUDA cores. (We will demonstrate this shortly)

VII. **Parallel Programming**

A. Making good use of parallel hardware requires appropriate programming, of course - and this highly hardware dependent.

B. However, we can look at a few simple parallelization strategies. Two can be used with multicore CPUs to allow parts of a parallelizable task to be partitioned among different cores.

HANDOUT: Discuss sequential program
Demo on a workstation - note time to find factors of demo large prime

C. One strategy is to explicitly parallelize using threads that run on differnet cores.

HANDOUT: Discuss pthreads program
Demo and compare time to sequential

Note that this strategy requires explicitly specifying the number of threads. Since the workstation has 4 cores, we could have broken the task down into 4 parts - but since we only broke it into 2, we only used 2 cores and achieved a speedup close to 2:1. (It's less than 2:1 because there are some parts of the program that are inherently sequential and starting the thread injects some overhead.)

D. Another strategy is to use Open MP (omp)**.**

HANDOUT: Discuss omp program
Demo and compare time to sequential

Note two things: this is much simpler, and omp discovers the number of cores at runtime, so we get close to 4:1 speedup on a 4 core machine. (It's less than 4:1 because there are some parts of the program that are inherently sequential and starting the thread injects some overhead.)

E. Because the workstations have graphics cards with NVidia GPU's, we can use CUDA.

HANDOUT: Discuss cuda program

1. Explain notion of parallel execution of a kernel

2. Note that the program has deliberately been made to use a very inefficient approach, in order to more dramatically show the difference between sequential version and parallel versions.

3. DEMO - be sure to do module load cuda first
   Note that, though the parallel program is much faster than the sequential one, it is nowhere near 384 times as fast as one might expect from the number of GPU cores we saw earlier. Why?

   ASK

   a. The clock rate for the GPUs is less than half that of the CPU.

   b. There is considerable overhead involved in moving data between the regular memory of the system (used by the CPU) and the dedicated memory used by the GPUs. That is particularly an issue in a program like this, where the amount of actual computation is quite moderate.

   c. The program was not at all structured to take into account the hardware architecture of the GPUs.