CS311 Lecture: Parallelism Last revised 11/26/07

Materials:

1. Projectable of Null figure 9.5
2. Projectable of Null figures 9.6, 9.7

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. Applications that are not feasible with present computing speeds, but would become feasible if speeds improved - e.g. improved weather forecasting, simulation of various natural and man-made systems, etc.

2. Volume of use - e.g. servers that must carry an ever-increasing load as more and more people make use of the services they provide.

3. People who use computers want instant results.

B. We have already seen that the time taken to perform a given task on a computer can be calculated using the following equation:

instructions X clock-cycles/instruction X seconds/clock-cycle

C. Likewise, we have already looked at three ways to make a computer system faster.

1. We can use better, faster algorithms - thus reducing the number of instructions that need to be executed.

a. Historically, a great deal of progress has been made in this area, and research certainly continues.

b. However, certain tasks have an inherent complexity that constitutes a lower bound on the complexity of any algorithmic solution to the problem.

Example: sorting by comparison is provably omega(n log n). While a better algorithm might improve the constant of proportionality some, no algorithm can do better than O(n log n) for this task.

2. We can use faster components (CPU's, memorys, etc) - thus reducing the time per clock cycle.

a. Again, significant progress has been made here. For example, the CPU's in the first PC's had clock rates on the order of 4 MHz - today systems are being marketed that exceed 3 GHz - almost a 1000-fold improvement.

b. While progress is continuing to be made in this area, we are nearing some fundamental physical limits

i. CPU clock speeds can be increased by making the individual features on the chip smaller, but there are fundamental limits on how small an individual feature can be.

ii. Moreover, faster CPU's consume more power, and heat dissipation considerations also seem to be limiting improvements in clock speeds.

iii. In fact, CPU clock speeds seem to have become "stuck" in the vicinity of 3 GHz or so, and for many years there has also been little improvement in such values as DRAM or disk access time.

3. We can reduce the average number of clock cycles needed per instruction by using strategies such as pipelining and superscalar architecture.

a. Pipelining seeks to reduce the average number of clock cycles needed per instruction to 1 by starting a new instruction on each clock, even as earlier instructions continue through subsequent steps in their execution.

b. Superscalar techniques can reduce the effective number of clock cycles per instruction to less than one by issuing more than one instruction at a time.

c. These strategies are sometimes called INSTRUCTION-LEVEL or FINE-GRAINED PARALLELISM. However, the degree of parallelism that can be achieved this way is inherently limited by sequential dependencies among instructions (data dependencies and branch dependencies).

D. We now look at another approach to improving performance, by the replication of functional units or whole CPU's, so that different portions of the same kind of work - or even different kinds of work that are part of the same task - can be done in parallel. This is known as COARSE-GRAINED PARALLELISM

As we approach fundamental limits in terms of efficiency of algorithms and raw CPU/memory speed, parallelism becomes the key to achieving further major improvements in computer speed. It is thus, in some sense, the "final frontier" of computer architecture.

E. Historically, some forms of parallelism have been present in computer systems from the earliest days through the use of specialized hardware subsystems for various Input-Output tasks.

1. One early example, which is still utilized in systems today, is the overlap of computation and disk input-output, as we discussed earlier.

2. Modern computer systems often use specialized processors to facilitate overlap between computation and various kinds of IO activities - e.g.

a. Disk controllers that allow disk seek/search to overlap computation.

b. Graphics processors that allow screen updates to overlap computation.

c. Network interfaces that allow processing of network packets to overlap computation.

etc.

E. For the remainder of this lecture, though, we will look at parallelism as it applies to the computation portion of a task - i.e. techniques that are especially useful for compute-bound tasks. In particular, we will look at three approaches:

1. Vector processoring.

2. Systems with multiple ALU's

3. Multiple-CPU systems - multiprocessors.

F. First, through, we want to look at a system for classifying parallel systems, known as FLYNN'S TAXONOMY.

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

2. Computers of the sort we have discussed up until now - i.e. all conventional 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.

3. 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. Vector processors and array processors - which we shall discuss shortly - fall into this category.

4. 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 insructions.

5. A MIMD machine has Multiple Instruction Streams, each operating on its own Data Stream. This kind of system, then, has multiple full CPU's, and is commonly known as a multiprocessor system.

II. Vector Processing

A. One interesting class of parallel machines is the class of Vector processors. Historically, some supercomputers of the 1980's and 1990's were built this way. Though there are no present examples of "pure" vector procesors, many CPU's include vector processing instructions that incorporate the basic idea into the ISA of a conventional CPU.

These differ from conventional CPU's in the following way:

1. A vector processor has a pipelined ALU.
2. Its instruction set has two kinds of instructions:
   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. This is done by piping entire vectors through the pipeline, 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.
   c. Example: If A, B, and C are two-dimensional matrices, then the matrix addition

      MAT A = B + C

      would require a pair of nested for loops on a conventional CPU, but a single machine instruction on a vector processor. (Though this instruction could take a long time to execute - typically one clock per element of the matrix, plus some additional clocks for setup and final flushing of the pipeline.)

B. Vector processors have been of two general kinds.

1. One has a set of vector registers, each of which holds some number of elements of a vector (e.g. 64). Instructions are provided to load a vector register from a vector in memory (given a starting address and stride) or to store a vector register into memory. Instructions are also provided to perform computations such as add on whole vectors.
   a. These instructions are pipelined, so that one element of the vector is handled on each clock once the operation is set up.
   b. A disadvantage is the fixed size of the vector registers. While these machines provide a way to specify that LESS than the maximum capacity of the registers is to be used, to work with vectors longer than the registers a loop is still needed, but this time to do a block at a time rather than a single element.
   c. Vector processing extensions present in current ISA's are of this type.
2. Vector processors have also been built that access operands directly in memory - so that a single instruction can specify that some operation be applied to corresponding elements of two full vectors, storing the result into a third, with execution being pipelined.

III. Arrays of ALU's

A. Another interesting class of parallel processor is a CPU that consists of multiple ALU's. Although there are no present examples of such systems, they are of historical interest. These differ from conventional CPU's in the following way:

1. A conventional CPU has a single (possibly pipelined) ALU. Thus, each instruction that is fetched and executed initiates a single computation in the ALU - though several such computations may overlap in time.

2. An array processor has an array of processing elements - perhaps scores or even hundreds.

   a. Each processing element contains its own ALU (which may be pipelined to achieve maximum speed.)

   b. Each processing element also has its own private memory - a set of registers and possibly a local RAM.

   c. A single user instruction may initiate a computation in each ALU (with each, of course, working on a different data item), or it may initiate computation only in a subset of the ALU's or even just one.

   d. To make full use of the power of the array processor, data must be distributed appropriately across the processing elements.

      Example: If we were working with matrices of 1000 elements each, and had 1000 PE's, we might store one element of each matrix in each PE - i.e. the [1] element of each matrix would be stored with PE 1. This would allow all elements of the matrix to be operated on at the same time.

3. The obvious use of such processors in in working with the vectors and matrices that arise frequently in scientific computation. For example, the matrix addition we considered above:

   MAT A = B + C

   could also be done by a single instruction on an array processor, with each ALU performing a single addition in parallel with all the others - assuming the elements of A, B, and C are distributed across the various processing elements as described above.

   a. Contrast this with the vector processor we just considered. On the vector processor, the additions are still done sequentially, but pipelining is used to finish one addition per clock period.

   b. On an array processor, all the additions are done at the same time, but by different ALU's - i.e. in at most a handful of clocks!

   c. Clearly, the array processor is even faster than the vector processor - but also more costly!

4. However, other applications arise as well - e.g. parallel searches - each processor compares the key of a desired piece of information against its locally-held information in parallel with other processors doing the same thing.

B. Historically, this idea found expression in various supercomputer architectures, as well as in a system known as "The Thinking Machine", which could have tens of thousands of ALU's.

IV. System Speed-Up by Using Multiple Full CPU's: Multiprocessors

A. A SISD machine has one CPU with one ALU, one memory, and one control. A SIMD machine has one CPU with either a pipelined ALU (vector processor) or else multiple ALU's and memories, but still one unit for fetching and interpreting instructions. Further parallelism can be achieved by using multiple complete CPU's.

1. Such a machine is called MIMD. It has Multiple Instruction Streams, each working on its own Data Stream - hence Multiple Data Streams as well. That is, each processor carries out its own set of instructions.

2. 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 ocassional, while in an MIMD machine all the CPU's work together on a common task.

B. One special case of the MIMD model is called SPMD - single program, multiple data. This is a case where one has a MIMD system, but each processor runs a copy of the same program, and performs its computation on a subset of the overall daa.

C. 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.

c. 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 explictly 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.

D. Further comments about shared memory systems.

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

a. In a Uniform Memory Access system (UMA), the time needed to access a given location in memory is the same, regardless of which processor accesses it. (Such systems are also called SYMMETRIC MULTIPROCESSING (SMP) SYSTEMS.)

b. 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. Either way, some mechanism is needed for SYNCHRONIZING accesses to shared memory.

a. Example: Suppose two different processors both need to periodically increment some shared variable (perhaps a counter of some sort.)

If the processors use load-store format instructions, the code would look something like:

```
lw $1, variable
nop
addi $1, $1, 1
sw $1, variable
```

Now suppose to processors happen to need to increment the variable at about the same time. Suppose, further, that its initial value is 40, so that it should be 42 after both increments take place. Finally, suppose the following sequence of operations occurs:

Processor 1	Processor 2
lw $1, variable	
nop	lw $1,variable
addi $1, $1, 1	nop
sw $1, variable	addi $1, $1, 1
	sw $1, variable

What is the final value of the variable? ASK

What went wrong?

b. Synchronization mechanisms are studied in detail in Computer Systems II. Suffice it to say 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 like the one just described.

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

   a. In a UMA system, it may 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. (Otherwise, contention for access the shared memory severly impacts performance. Thus, we must consider how the various CPU modules and memory modules are connected.

   b. 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.

   c. Two alternative connection structures that might be used.

      i. 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

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

         PROJECT - Null Figures 9.6, 9.7

4. Finally, because contention for access to shared memory would severely limit performance of a shared memory system, having each processor have its own cache is essential. (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. If a common bus is used to connect all the processors to the shared memory, then one possible solution is to use SNOOPY CACHEs, in conjunction with write-through caching.

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

      ii. 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 use shared memory? 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.

V. Conclusion

A. One ongoing research question is how to get a performance increase out of a parallel system that is commensurate with the investment in extra hardware.

1. The SPEEDUP of a parallel system is defined as

$$\frac{\text{Time to perform a given task on a single, non-parallel system}}{\text{Time to perform the same task on a parallel system}}$$

2. Ideally, we would achieve LINEAR SPEEDUP, where a parallel system with n processors has a speedup of n when compared with a single processor.

3. However, such speedup is never totally attainable, and often difficult to even approach, due to issues like:

a. Inherent sequentialness in the problem.

Example: Earlier we considered adding up the elements of a 1024 element vector using 512 processors. We showed that this can be done by the log sum algorithm in 10 steps. Since doing the addition on a single processor would take 1023 steps, the speedup is about 100 - far less than the ideal given the use of 512 processors.

An important principle, known as AMDAHL'S LAW (formulated by Gene Amdahl, a supercomputer architect) is that if a speedup S improves the performance of a portion P of a task, then the overall speedup is

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

Example: 50% of a task can be speedup by parallelism, and 50% of the task is inherently sequential. If the parallel part is speeded up by a factor of 10, the overall speedup is

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

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 case cited above, the maximum possible speedup is 2, which would be achieved if the time for the non-parallelizable portion remained the same, while the remaining time dropped to 0.

b. Various overheads associated with parallelism - e.g.

i. Contention for shared resources (bus, memory).

ii. Synchronization.

iii. Network overhead for communication.

4. Whole textbooks exist on this topic!

B. In addition to enhanced speed, though, there are other motivations for building parallel systems that may be even more important.

ASK

1. Reliability/availability

2. Incremental upgrade