Computer Science 6365 **January 24, 2008**

Lecture #4: Performance Concepts

Professor: S. Lennart Johnsson *TA: Wei Ding*

1 Speedup

In this lecture, we examine one of the key performance measures for computing systems.

1.1 Definitions

Let the time required for a task prior to an enhancement be T_{old}; let the time after an improvement be T_{imp}. The speedup is defined as

$$\text{Speedup } S = \frac{T_{old}}{T_{imp}}.$$

An alternative definition can be made in terms of the performance before and after the enhancement. Let R_{old} be the computational rate before the enhancement; let R_{imp} be the rate after the enhancement. Then,

$$\text{Speedup } S = \frac{R_{imp}}{R_{old}}.$$

The enhancement can take many forms, such as improved hardware features, faster memory, faster functional units, shorter pipelines, improved data allocation in a banked memory system, improved vectorization by a compiler, or multiple processors.

Most enhancements do not apply uniformly to a task (system clock rate being an exception). Hence, it is important to factor out the different parts of a computation and consider the speedup for each part. As an example, assume that there are two sections of a program, one which achieves a speedup S, and a second which is not affected. Let the fraction of the time T_{old} which is affected by the enhancement be f. Then, the total time after the enhancement is

$$T_t = (1-f)\cdot T_{old} + \frac{f\cdot T_{old}}{S}$$

and the total speedup S_t is

$$S_t = \frac{1}{1-f+\frac{f}{S}}.$$

This expression is known as **Amdahl's law**. It expresses the law of diminishing returns: the incremental gain from improving the performance of a portion of the computation diminishes as further improvements are added. The ratio

$$\frac{1}{1-f}$$

is known as **Ware's law**. It is an upper bound on the speedup. We now consider two examples of speedup.

Example 1: A Trip from Nevada to California [2]

You must go from Nevada to California over the Sierra Nevada mountains and through the desert to Los Angeles. For the desert part of the trip, you have several vehicles available. But the route through the mountains goes through ecologically sensitive areas and you must walk. This walk takes 20 hours. For the 200 mile desert run you can use one of the following options:

1. Walk at an average rate of 4 miles per hour.
2. Ride a bike at 10 miles per hour.
3. Drive a Hyundai Excel at an average speed of 50 miles per hour.
4. Drive a Ferrari Testarossa at an average speed of 120 miles per hour.
5. Drive a rocket car at an average speed of 600 miles per hour.

The speedup for the desert part, as well as for the total trip, are shown in Table 1. The 150 fold speedup for the desert portion, using a rocket car instead of walking, only results in a total speedup by a factor of 3.4. Much of this speedup is already achieved by the Hyundai Excel alternative. Clearly, not much is gained by reducing the time for the desert portion further than this option, for a tremendous increase in cost. Ware's law states that $S_t \leq 3.5$ for this example.

Example 2: Improving the Speed of a CPU

Suppose one–half of the time for a computation is consumed by the CPU while the other half is consumed by I/O as well as other operations in which the CPU does not participate. Assume that the CPU speed can be improved by a factor of five for five times the cost. Let the CPU represent one–third of the total system cost. Then, with the CPU improvement, the speedup is

$$S_t = \frac{1}{1-0.5+\frac{0.5}{5}} = 1.67.$$

Vehicle for second part	Hours for second part	Speedup of second part	Hours for total trip	Total speedup
Feet	50.00	1.0	70.00	1.0
Bike	20.00	2.5	40.00	1.8
Excel	4.00	12.5	24.00	2.9
Testarossa	1.67	30.0	21.67	3.2
Rocket car	0.33	150.0	20.33	3.4

Table 1: Total and partial speedup for different means of transportation for one portion of a trip.

The total cost becomes $1 \times \frac{2}{3} + 5 \times \frac{1}{3} = 2.33$. Hence, the cost/performance ratio has increased.

In general, different parts of a computation may be subject to different speedup. Let fraction f_i have speedup S_i for $i = 1, 2, \ldots k$. Then,

$$S_t = \frac{1}{1 - \sum_{i=1}^{k} f_i + \sum_{i=1}^{k} \frac{f_i}{S_i}}.$$

1.2 Speedup in Parallel Computing

Perfect Speedup is achieved when the speedup from applying P processors to a problem is $S_t = P$. Let the fraction f of the computation that realizes perfect speedup be f, while the remaining fraction $1 - f$ realizes no speedup. Then, the total speedup for P processors, S_t^P, is

$$S_t^P = \frac{1}{1 - f + \frac{f}{P}} = \frac{P}{(1-f)P + f} \leq \frac{1}{1-f}.$$

The **Efficiency** is defined as

$$E_t^P = \frac{S_t^P}{P} = \frac{1}{(1-f)P + f}.$$

For perfect speedup, $E_t^P = 1$.

Uncritical application of this definition of speedup is not recommended (though often done). A change in architectural characteristics, in particular, application of many processors to the same task, may suggest a different algorithm. Also, in practice, the problem size tends to depend upon the number of processors, such that the fraction f becomes dependent upon P.

Two possible alternate definitions for parallel speedup are:

Alternative 1: Fair speedup

$$\text{Speedup} = \frac{\text{Time for best algorithm on a single processor}}{\text{Time for best algorithm on } P \text{ processors}}$$

Alternative 2: Scaled fair speedup

$$\text{Speedup} = \frac{\text{Time for best algorithm on a single processor for problem size } M}{\text{Time for best algorithm on } P \text{ processors for problem size } PM}$$

The first definition accounts for a different choice of algorithm; the second accounts for a scaling of the problem size with the number of processors. Both definitions are used and are relevant for different scenarios.

Remark. With the original definition of speedup, the asymptotic efficiency is zero, unless the speedup is perfect (i.e., $f = 1$).

Example 3

Consider a problem that scales as follows. One portion of the program is independent of the size of the problem, cannot be parallelized, and requires time T_0. The remainder of the program can be parallelized, scales perfectly, and requires time $P \cdot T_1$ on one processor. The total time for this problem on one processor would be

$$T_t^1 = T_0 + P \cdot T_1.$$

When run on P processors, the speedup is

$$S_t^P = \frac{T_t^1}{T_t^P} = \frac{T_0 + P \cdot T_1}{T_0 + T_1} = P - \frac{T_0}{T_0 + T_1}(P - 1).$$

Note that if $T_0 = 0$, then there is perfect speedup. For $T_0 = T_1$, $S_t^P \approx \frac{1}{2}P$, and for $T_0 \gg T_1$, $S_t^P \approx 1 + \frac{T_1}{T_0}P$.

2 Parallel architectures

Three generic parallel architectures are shown in Figure 1.

The top architecture has a bus which all processors use for the loading and storing of data. The capacity of the bus must be sufficiently high to support all processors. With an 8–byte wide bus and a clock frequency of 1 GHz for both processors and the bus, the bus may be a bottleneck even for a single processor. If the processor must fetch all operands from memory and store the result in memory, then it takes several cycles just to support a single operation on a single processor.

An architecture of this pure form is not very effective. In practice, programs exhibit locality of reference, both with respect to instructions and data. This characteristic allows a small local memory per processor to reduce the demands on the bus and the memory with proper scheduling of references.

In practice, parallel architectures based on a bus for interconnecting processors and memory modules do have a certain amount of memory local to a processor, but the number of processors is still very limited. Symmetric Multiprocessors (SMPs) typically use this form of architecture.

In the architecture illustrated in the center of Figure 1, the bus is replaced by a network. If the network offers sufficient routing flexibility and capacity, then each processor may experience a bandwidth to memory that is equal to that of its connecting port. For this property to hold, the capacity of the network must equal the sum of all the port rates for any access pattern. The data accesses must be distributed evenly among the memory units. Otherwise, either the memory or the network, or both, become bottlenecks.

Some systems are best characterized as hybrids between the architectures in which the memory is entirely distributed among the processors and architectures with a global memory.

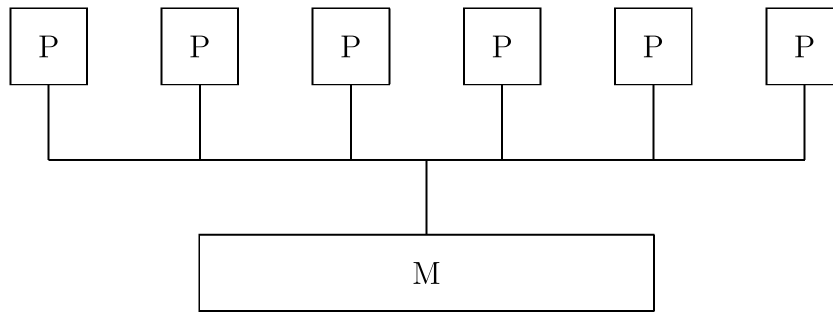
The current generation of high-end systems are typically constructed as networked SMP nodes. This is true for high-end products by Compaq, HP, IBM, SGI and Sun.

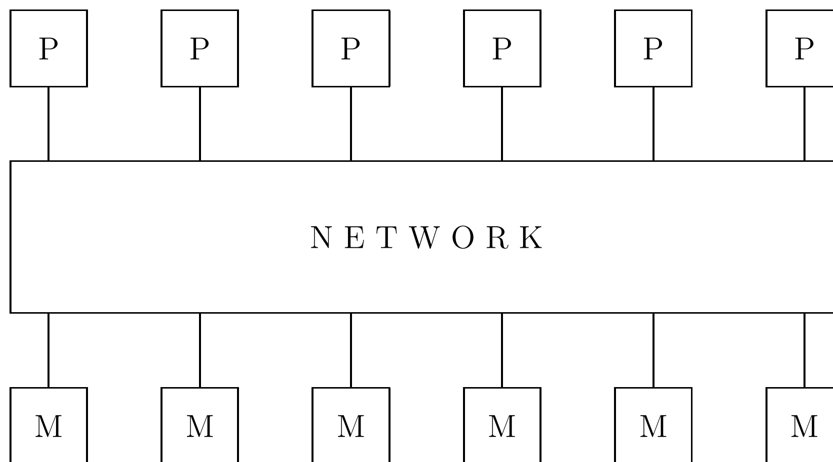
3 Parallel Terminology

Hardware

Distributed memory. In a distributed memory system, each processor has its own local memory. Typically, all of memory is divided up among the processors (i.e., there is no global memory). The term **multicomputer** is sometimes used to denote a distributed memory architecture.

Shared memory is used ambiguously to denote either a programming model or a hardware configuration. Used to describe a hardware configuration, shared memory denotes a separate, global memory accessible to the processors through a memory bus or through a network. The term **multiprocessor** is sometimes used to denote an architecture with a global shared memory.





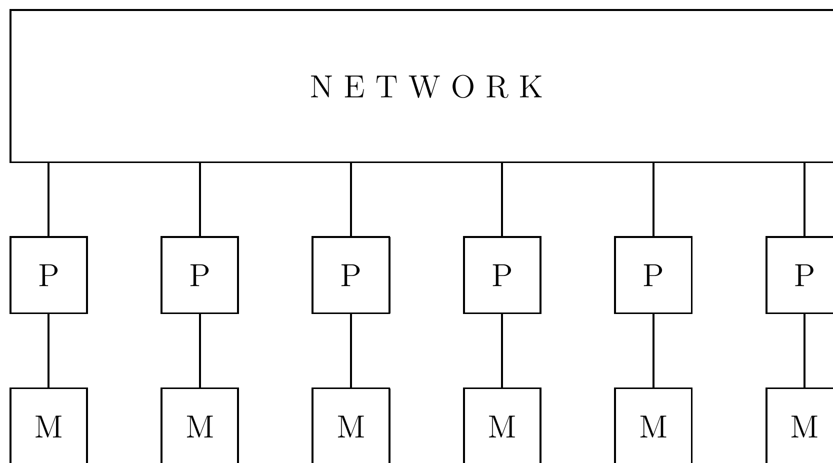


Figure 1: Three generic models of parallel architectures.

Programming models

Data parallelism. Shared address space, single program. Parallelism offered by operating on different data elements, typically different elements of an array. Data parallelism is **fine–grained** parallelism.

Message passing. In the message passing programming model processes exchange information and synchronize by passing messages. There are multiple threads of control. Address spaces are localized to each process. In the context of distributed memory architectures, the message passing programming model currently implies that each process is limited to reside in a single node, and the address space is limited to that of the node. The global data structures are maintained explicitly by the programmer.

Shared Memory. There is a shared address space and multiple threads of control. All memory accesses are treated as if they were equally fast. Synchronization of different processors takes place through shared variables (e.g., fetch–and–add, test–and–set).

Control parallelism. Parallelism offered by executing different threads of control. Control parallelism is often **coarse–grained**. Blocks of code that can be executed independently, are executed in parallel.

Some additional terminology

UMA. Uniform memory access. All memory locations have the same access time.

NUMA. Nonuniform memory access. Different memory locations have different access times, seen from a processor. Memory locations are "closer" to some processors than to others.

4 Parallel Instruction Modes

The first four of the following terms were coined by Flynn [1] as an attempt to establish a taxonomy for computers. The last is a fairly recent addition which has emerged with increased experience in the programming of parallel computers.

- SISD. Single Instruction stream, Single Data stream.
- SIMD. Single Instruction stream, Multiple Data streams.
- MIMD. Multiple Instruction streams, Multiple Data streams.
- MISD. Multiple Instruction streams, Single Data stream.
- SPMD. Single Program, Multiple Data streams.

4.1 SISD

A SISD computer is the familiar uniprocessor.

4.2 SIMD

Vector processors are sometimes considered as SIMD architectures, since a single instruction operates on multiple data elements. However, only a single functional unit (or functional units chained together) are used in a single instruction.

In a true SIMD computer, many functional units operate synchronously, controlled by a single program counter. SIMD machines have a dedicated processor for instruction decoding, address calculation, etc. This processor is often of the SISD variety and often referred to as the *front–end* or host. SIMD machines work well for data parallelism and, in particular, when operations are performed on large arrays. The Cell Broadband Engine (CBE) has eight Synergitic Processing Elements (SPEs) each of which is a SIMD (vector) computer

4.3 MIMD

In MIMD architectures, each processor may have a copy of the entire program which, experience has shown, may consume a measurable part of the memory. On the other hand, if the program is local to a processor, critical network bandwidth need not be used for instruction broadcast, or equivalently, chip or board connectors need not be reserved for instruction broadcast should a separate network be used for this purpose. Moreover, branching is handled easily in MIMD architectures, as are the execution of entirely different codes in different processors.

Many MIMD architectures are programmed using a **message passing** programming model. In such a programming model, the programmer controls the synchronization between execution streams through messages; all transfer of data between processors (processes) takes place via messages. In SIMD architectures, the synchronization is hidden from the programmer. Asynchronous systems can lead to bugs that are very difficult to find.

SIMD tends to be used for fine–grained architectures and MIMD for coarse–grained architectures. For a given amount of silicon, or gates, SIMD allows more of the logic to be devoted either to memory or to processing logic. The instruction decoder and finite state machine of a processor occupies a significant part of its chip area. Moreover, in a SIMD architecture there is a single copy of the program.

Clusters are examples of MIMD architectures since each processor executes its own instructions and has their own program.

4.4 MISD

Though it is one of the combinations of instruction and data streams in Flynn's taxonomy, no system seems to naturally fit this label. Systolic arrays are sometimes put in this category, but it is not an obvious fit.

4.5 SPMD

SPMD is a new classification that stands for **S**ingle **P**rogram **M**ultiple **D**ata streams. For the most part, this category has been introduced as a consequence of how most of the distributed memory architectures have been programmed. Typically, all nodes execute the same program, though on different data sets. The different copies of the programs take different paths through the code as a function of the data dependencies. Hence, the instruction mode could be SIMD, but is often MIMD. On the other hand, the MIMD mode allows each processor to run entirely different programs, hence offering greater flexibility than the SPMD mode. The Cell Broadband engine canbe viewed as a SPMD machine. The control processor holds the program, but the SPEs do not need to operate in lock step at the instruction level.

References

[1] Michael J. Flynn. Very high-speed computing systems. *Proc. of the IEEE*, 12:1901–1909, 1966.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantative Approach.* Morgan Kaufmann Publishers, Inc, 1990.