Parallel Architectures

MICHAEL J. FLYNN AND KEVIN W. RUDD

Stanford University ⟨flynn@Umunhum.Stanford.edu⟩; ⟨kevin@Umunhum.Stanford.edu⟩

PARALLEL ARCHITECTURES

Parallel or concurrent operation has many different forms within a computer system. Using a model based on the different streams used in the computation process, we represent some of the different kinds of parallelism available. A stream is a sequence of objects such as data, or of actions such as instructions. Each stream is independent of all other streams, and each element of a stream can consist of one or more objects or actions. We thus have four combinations that describe most familiar parallel architectures:

(1) SISD: single instruction, single data stream. This is the traditional uniprocessor [Figure 1(a)].
(2) SIMD: single instruction, multiple data stream. This includes vector processors as well as massively parallel processors [Figure 1(b)].
(3) MISD: multiple instruction, single data stream. These are typically systolic arrays [Figure 1(c)].
(4) MIMD: multiple instruction, multiple data stream. This includes traditional multiprocessors as well as the newer networks of workstations [Figure 1(d)].

Each of these combinations characterizes a class of architectures and a corresponding type of parallelism.

SISD

The SISD class of processor architecture is the most familiar class and has the least obvious concurrency of any of the models, yet a good deal of concurrency can be present. Pipelining is a straightforward approach that is based on concurrently performing different phases of processing an instruction. This does not achieve concurrency of execution (with multiple actions being taken on objects) but does achieve a concurrency of processing—an improvement in efficiency upon which almost all processors depend today.

Techniques that exploit concurrency of execution, often called instruction-level parallelism (ILP), are also common. Two architectures that exploit ILP are *superscalar* and VLIW (very long instruction word). These techniques schedule different operations to execute concurrently based on analyzing the dependencies between the operations within the instruction stream—dynamically at run time in a superscalar processor and statically at compile time in a VLIW processor. Both ILP approaches trade off adaptability against complexity—the superscalar processor is adaptable but complex whereas the VLIW processor is not adaptable but simple. Both superscalar and VLIW use the same compiler techniques to achieve high performance.

The current trend for SISD processors is towards superscalar designs in order to exploit available ILP as well as existing object code. In the marketplace there are few VLIW designs, due to code compatibility issues, although advances in compiler technology may cause this to change. However, research in all aspects of ILP is fundamental to the development of improved architectures in all classes because of the frequent use of SISD architectures as the processor elements in most implementations.

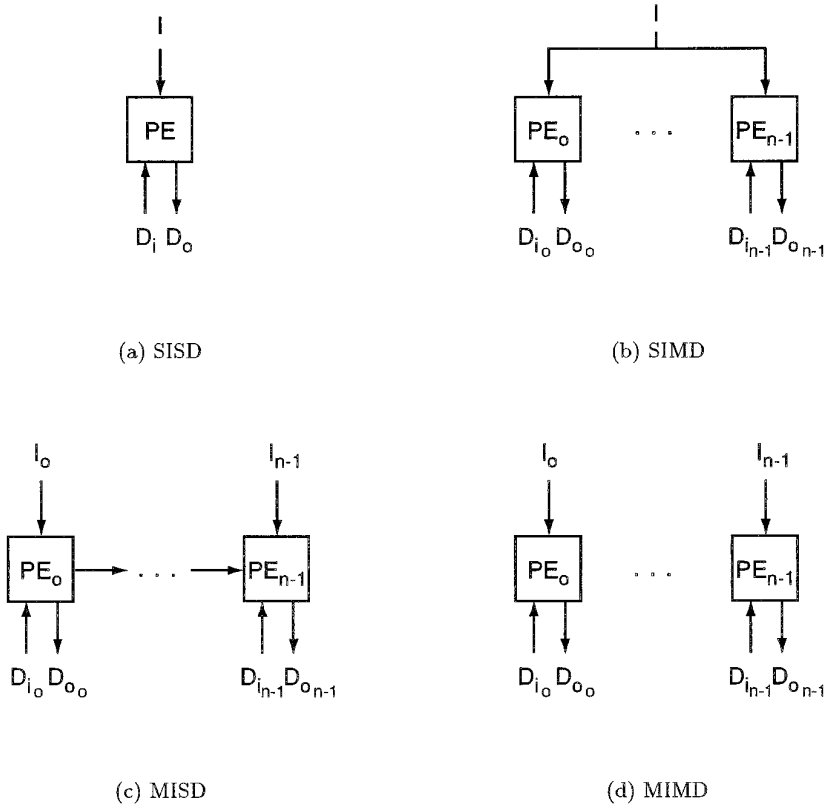

Figure 1. The stream model.

SIMD

The SIMD class of processor architecture includes both array and vector processors. This processor is a natural response to the use of certain regular data structures such as vectors and matrices. Two different architectures, array processors and vector processors, have been developed to address these structures.

An array processor has many processor elements operating in parallel on many data elements. A vector processor has a single processor element that operates in sequence on many data elements. Both types of processors use a single operation to perform many actions. An array processor depends on the massive size of the data sets to achieve its efficiency (and thus is often referred to as a massively parallel processor), with a typical array processor consisting of hundreds to tens of thousands of relatively simple processors operating together. A vector processor depends on the same regularity of action as an array processor but on smaller data sets and relies on extreme pipelining and high clock rates to reduce the overall latency of the operation.

There have not been a significant number of array architectures developed due to a limited application base and market requirement. There has been a trend towards more and more complex processor elements due to increases in chip density, and recent array architectures blur the distinction between SIMD and MIMD configurations. On the other hand, many different kinds of vector processors have developed dramatically through the years. Starting with simple memory-based vec-

tor processors, modern vector processors have developed into high-performance multiprocessors capable of addressing both SIMD and MIMD parallelism.

MISD

Although it is easy to both envision and design MISD processors, there has been little interest in this type of parallel architecture. The reason, so far anyway, is that no ready programming constructs easily map programs into the MISD organization.

Abstractly, the MISD is a pipeline of multiple independently executing functional units operating on a single stream of data, forwarding results from one functional unit to the next. On the microarchitecture level, this is exactly what the vector processor does. However, in the vector pipeline the operations are simply fragments of an assembly-level operation, as distinct from being a complete operation in themselves. Surprisingly, some of the earliest attempts at computers in the 1940s could be seen as the MISD concept. They used plug boards for programs, where data in the form of a punched card was introduced into the first stage of a multistage processor. A sequential series of actions was taken in which the intermediate results were forwarded from stage to stage until at the final stage a result was punched into a new card.

MIMD

The MIMD class of parallel architecture is the most familiar and possibly most basic form of parallel processor: it consists of multiple interconnected processor elements. Unlike the SIMD processor, each processor element executes completely independently (although typically the same program). Although there is no requirement that all processor elements be identical, most MIMD configurations are homogeneous with all processor elements identical.

When communications between processor elements are performed through a shared memory address space (either global or distributed between processor elements, called distributed shared memory to distinguish it from distributed memory), two significant problems arise. The first is mainlining memory consistency—the programmer-visible ordering effects of memory references both within a processor element and between different processor elements. The second is maintaining cache coherency—the programmer-invisible mechanism to ensure that all processor elements see the same value for a given memory location. The memory consistency problem is usually solved through a combination of hardware and software techniques. The cache coherency problem is usually solved exclusively through hardware techniques.

There have been many configurations of MIMD processors that have ranged from the traditional processor described in this section to loosely coupled processors based on networking commodity workstations through a local area network. These configurations differ primarily in the interconnection network between processor elements that range from on-chip arbitration between multiple processor elements on one chip to wide-area networks between continents, the tradeoffs being between the latency of communications and the size limitations on the system.

LOOKING FORWARD

We are celebrating the first 50 years of electronic digital computers—the past, as it were, is history, and it is instructive to change our perspective and to look forward and consider not what has been done but what must be done. Just as in the past there will be larger, faster, more complex computers with more memory, more storage, and more complications. We cannot expect that processors will be limited to the "simple" uniprocessors, multiprocessors, array processors, and vector processors we have today. We cannot expect that the

programming environments will be limited to the simple imperative programming languages and tools that we have today.

As before, we can expect that memory cost (on a per-bit basis) will continue its decline so that systems will contain larger and larger memory spaces. We are already seeing this effect in the latest processor designs that have doubled the "standard" address size, yielding an increase from 4,294,967,296 addresses (with 32 bits) to 18,446,744,073,709,551,616 addresses (with 64 bits). We can expect that interconnection networks will continue to grow in both scale and performance. The growth in the Internet in the last few years is phenomenal and the increase in the use of optics in the interconnection network has made this increase at least feasible.

However, we cannot expect that the ease of programming these improved configurations will advance—as the available parallelism of computer systems increases, exploiting this parallelism becomes the limiting factor. There are two aspects of this problem: finding large degrees of parallelism (typically an *algorithmic* or *partitioning* problem) and efficiently managing the available parallelism to achieve high performance (typically a *scheduling* or *placement* problem). Of course, all the solutions to these problems must ensure that correctness is satisfied. It does not matter how fast the program runs if it does not produce the correct result. Solving these problems will require many developments and changes, few of which are foreseeable.

Although not satisfying, we can certainly say that programming paradigms, compiler techniques, algorithm designs, and operating systems are all fair game, but these are likely only pieces of the puzzle. Indeed, broad new approaches to the representation of physical problems may be required. The good news from all this is that there is no dearth of work to be done in this area. Although improvements can certainly be made to a single processor element, the performance benefits of these improvements are complementary and at this point are nowhere near the scale of performance available through exploiting parallelism. Clearly, providing parallelism of order n is much easier than increasing the execution rate (for example, the clock speed) by a factor of n.

The continued drive for higher- and higher-performance systems thus leads us to one simple conclusion: the future is parallel. The first electronic computers provided a speedup of 10,000 compared to the mechanical computers of 50 years ago. The challenge for the future is to realize parallel processors that provide a similar speedup over a broad range of applications. There is much work to be done here. . .let us be on with it!

REFERENCES

There are thousands of references dealing with the many aspects of parallel architectures. These references comprise only a very small subset of accessible publications, but provide the interested reader with jumping-off points for further exploration.

FLYNN, M. J. 1995. *Computer Architecture: Pipelined and Parallel Processor Design.* Jones and Bartlett, Boston.

HOCKNEY, R. W. AND JESSHOPE, C. R. 1988. *Parallel Computers 2: Architecture, Programming and Algorithms*, 2nd ed. Adam Hilger, Bristol.

HOCKNEY, R. W. AND JESSHOPE, C. R. 1981. *Parallel Computers: Architecture, Programming and Algorithms*. Adam Hilger, Bristol.

HWANG, K. 1993. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, New York.

IBBETT, R. N. AND TOPHAM, N. P. 1989a. *Architecture of High Performance Computers*, Vol. I. *Uniprocessors and Vector Processors*. Springer-Verlag, New York.

IBBETT, R. N. AND TOPHAM, N. P. 1989b. *Architecture of High Performance Computers*, Vol. II. *Array Processors and Multiprocessor Systems*. Springer-Verlag, New York.

KUHN, R. H. AND PADUA, D. A. EDS. 1981. *Tutorial on Parallel Processing*. IEEE Computer Society Press, Los Alamitos, CA.

WOLFE, M. J. 1996. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Reading, MA.