Multi-core Microprocessors

V Rajaraman

Multi-core microprocessor is an interconnected set of independent processors called cores integrated on a single silicon chip. These processing cores communicate and cooperate with one another to execute one or more programs faster than a single core processor. In this article we describe how and why these types of processors evolved. We also describe the basics of how multi-core microprocessors are programmed.

V Rajaraman is at the Indian Institute of Science, Bengaluru. Several generations of scientists and engineers in India have learnt computer science using his lucidly written textbooks on programming and computer fundamentals. His current research interests are parallel computing and history of computing.

Introduction

Today every computer uses a multi-core microprocessor. This is not only true for desktop computers but also for laptops, tablets, and even microprocessors used in smartphones. Each 'core' is an independent processor and in multi-core systems these cores work in parallel to speed up the processing.

Reviewing the evolution of microprocessors, the first processor that came integrated in a chip was the Intel 4004, a 4-bit microprocessor that was introduced in 1971 and was primarily used in pocket calculators. This was followed by the Intel 8008, an 8-bit microprocessor introduced in 1972 that was used to build a full-fledged computer. Based on empirical data, Gordon Moore of Fairchild Semiconductors predicted that the number of transistors which could be packed in an integrated circuit would double almost every two years. This is often called Moore's law and has surprisingly been accurate till date (See *Box* 1). The increase in the number of transistors in a microprocessor was facilitated by decreasing the size of each transistor. Decreasing the size of transistors enabled faster clock speeds to be used to drive the circuits using these transistors and resultant improvement in the processing speed. Packing more transistors in a chip also enabled designers to improve the architecture of microprocessors in many

Keywords
Moore's law, evolution of multi-core processors, programming multi-core processors.

The speed mismatch between the processor and the main memory in a microcomputer was reduced by increasing the number of registers in the processor and by introducing on-chip cache memory.

ways. The first improvement was increase in the chunks of data that could be processed in each clock cycle. It was increased from 8 bits to 64 bits by doubling the data path width every few years (See *Table* 1). Increasing the data path width also allowed a microprocessor to directly address a larger main memory. Throughout the history of computers, the processor was much faster than the main memory. Thus, reducing the speed mismatch between the processor and the main memory in a microcomputer was the next problem to be addressed. This was done by increasing the number of registers in the processor and by introducing on-chip cache memory using the larger number of transistors that could be packed in a chip. Availability of a large number of transistors in a chip also enabled architects to increase the number of arithmetic units in a processor. Multiple arithmetic units allowed a processor to execute several instructions in one clock cycle. This is called *instruction level parallel processing*. Another architectural method of increasing the speed of processing besides increasing the clock speed was by *pipelining*. In pipelining, the instruction cycle of a processor is broken up into p steps, each taking approximately equal time to execute. The p steps of a set of sequential instructions are overlapped when a set of instructions are executed sequentially (as in an assembly line) thereby increasing the speed of execution of a large sequence of independent instructions p fold. All these techniques, namely, increasing the clock frequency, increasing the data path width, executing several instructions in one clock cycle, increasing on-chip memory, and pipelining that were used to increase the speed of a single processor in a chip could not be sustained as will be explained in what follows.

Why Multi-core?

The number of transistors packed in a single silicon chip has been doubling every two years.

We saw in the last section that the number of transistors packed in a single silicon chip has been doubling every two years with the result that around 2006 designers were able to pack about 240 million transistors in a chip.

Box 1. Moore's Law

In 1965 Gordon Moore* who was working at Fairchild Semiconductors predicted, based on empirical data available since integrated circuits were invented in 1958, that the number of transistors in an integrated circuit chip would double every two years. This became a self-fulfilling prophecy in the sense that the semiconductor industry took it as a challenge and as a benchmark to be achieved. A semiconductor manufacturers' association was formed comprising chip fabricators and all the suppliers who supplied materials and components to the industry to cooperate and provide better components, purer materials, and better fabrication techniques that enabled doubling the number of transistors in chips every two years. *Table* 1 indicates how the number of transistors has increased in microprocessor chips made by Intel – a major manufacturer of integrated circuits. In 1971, there were 2300 transistors in the microprocessors made by Intel. By 2016, it had reached 7,200,000,000 – an increase by a factor about 3×10^6 in 23 years, very close to Moore's law's prediction. The width of the gate in a transistor in the present technology (called 14 nm technology) is about 50 times the diameter of a silicon atom. Any further increase in the number of transistors fabricated in a planar chip will lead to the transistor gate size approaching 4 to 5 atoms. At this size quantum effects will be evident and a transistor may not work reliably. Thus we may soon reach the end of Moore's law as we know it now [1].

*Gordon E Moore, Cramming More Components, *Electronics*, Vol.38, No.8, pp.114–117, April 1965.

The question arose on how to effectively use these transistors in designing microprocessors. One major point considered was that most programs are written using a sequential programming language such as C or C++. Many instructions in these languages could be transformed by compilers to a set of instructions that could be carried out in parallel. For example, in the instruction:

$a = (b + c) - (d * f) - (g/h),$

the add operation, the multiply operation, and the divide operation could be carried out simultaneously, *i.e.*, in parallel. This is called *instruction-level parallelism*. This parallelism could be exploited provided there are three arithmetic units in the processor. Thus designers began putting many arithmetic units in processors. However, the number of arithmetic units could not be increased beyond four as this simple type of parallelism does not occur often enough in programs and that resulted in idle arithmetic units. Another parallelism is *thread-level parallelism* [2]. A thread in a program may be defined as a small set of sequential

In some programs, a set of threads are independent of one another and could be carried out concurrently using multiple arithmetic units.

Year	**Intel Processor Model**	**Data Path Width**	**Number of Transistors**	**Clock Speed**
1971	4004	4	2300	740 KHz
1972	8008	8	3500	500 KHz
1977	8085	8	6500	3 MHz
1978	8086	16	29,000	5 MHz
1982	80186	16	55,000	6 MHz
1982	80286	16	134,000	6 MHz
1985	80386	32	275,000	16–40 MHz
1989	80486	32	1,180,000	25 MHz
1993	Pentium 1	32	3,100,000	60–66 MHz
1995	Pentium Pro	32	5,500,000	150–200 MHz
1999	Pentium 3	32	9,500,000	450–660 MHz
2001	Itanium 1	64	25,000,000	733–800 MHz
2003	Pentium M	64	77,000,000	0.9–1.7 GHz
2006	Core 2 duo	64	291,000,000	1.8–2 GHz
2008	Core i7 Quad	64	730,000,000	2.66–3.2 GHz
2010	8 core Xeon Nehalem – Ex	64	2,300,000,000	1.73-2.66 GHz
2016	22 core Xeon Boradwell	64	7,200,000,000	2.2–3.6 GHz

Table 1. Timeline of progress of Intel* processors.

(*Intel processors are chosen as they are widely used and represent how a major manufacturer of microprocessors steadily increased the number of transistors in their processor and the clock speed. The data given above is abstracted from the articles on Intel microprocessor chronology and on transistor counts in microprocessors published by *Wikipedia*. The numbers in the table are indicative as there are numerous models of the same Intel microprocessor such as Pentium 3).

instructions that can be scheduled to be processed as a unit sharing the same processor resources. A simple example of a thread is a set of sequential instructions in a *for* loop. In some programs, a set of threads are independent of one another and could be carried out concurrently using multiple arithmetic units. In practice, it is found that most programs have limited thread-level parallelism. Thus it is not cost effective to increase arithmetic units in a processor beyond three or four to assist thread-level parallel processing. Novel architectures such as very long instruction word processors in which an instruction length was 128 to 256 bits and packed several instructions that could be carried out in parallel were also designed. But these had limited success as compilers to exploit this architecture were difficult to write particularly because programs were most often written in languages such as C that are inherently sequential and use pointers liberally. It was found that designing a single processor with a complex structure to exploit parallelism in sequential programs was becoming difficult. Designing complex processors is expensive and prone to errors. Thus, the limit on instruction and thread-level processing coupled with design complexity prompted designers to explore additional methods of utilising the transistors available in a chip.

Another method of using transistors is to deploy them for fabricating on-chip memory. Throughout the evolution of computer architecture, memory speed was always slower than the speed of processors. Besides this, main memories were external to the chip and fetching data from an external memory to the processor using a bus was slow. Computers had to wait for instructions and data to arrive at the CPU. This was alleviated by the invention of caches, a small, fast memory that could be integrated as part of a processor chip using the increasing number of transistors available in the chip. Separate caches for instructions and data are now the norm. Designers thus started increasing the size of the caches and on-chip memory to utilise the large number of available transistors to speed up processing. This also had diminishing returns. It is effective to have a cache that can store a set of instructions and data that would be immediately required by a processor (called a working set). Much larger cache than a working set is not very effective. Therefore, there is a limit beyond which it is not cost-effective to increase the cache size. Designers were forced to examine more alternatives to use the available transistors. An obvious idea was to replicate processors (called 'cores'), place them on a single chip, and interconnect them to work co-

The limit on instruction and thread-level processing coupled with design complexity prompted designers to explore additional methods of utilising the transistors available in a chip.

Design cost of multi-core processors is lower than the cost of designing very complex single processors. This is because in a multi-core processor a simple processor is designed and replicated.

operatively to execute programs. This requires programming constructs that will assist in extracting parallelism in programs. We will explain these constructs later in this article. Design cost of multi-core processors is lower than the cost of designing very complex single processors. This is because in a multi-core processor a simple processor is designed and replicated.

The idea of using several independent processors to work simultaneously and cooperate to execute a single program is quite old. This was a very active area of research in the late 1970s and early 1980s [3, 5]. As the circuit interconnecting the processors was outside the chip and slow in late 1970s, research challenges were different. However, many of the ideas developed during this period could be reused in designing multi-core processors.

Limits on Increasing Clock Frequency

As we saw, the speed of microprocessors was increased by both architectural innovations and increasing the clock speed. Whereas the number of transistors increased exponentially leading to architectural innovations, the increase of clock speed was gradual (See *Table* 1) and has remained constant around 3 GHz since 2006. This has been mainly due to heat dissipation in processors when the clock speed increased. There are three components of heat dissipation in processors when transistor switches toggle. They are:

Whereas the number of transistors increased exponentially leading to architectural innovations, the increase of clock speed was gradual and has remained constant around 3 GHz since 2006. This has been mainly due to heat dissipation in processors when the clock speed increased.

1. The capacitance associated with a transistor switch charges and discharges when it switches states. This causes power dissipation and consequently heating of the transistor. If C is the equivalent capacitance of a transistor, V_d the voltage at which the transistor operates, and f the clock frequency, the power dissipation when a transistor switches is proportional to CV_d^2f. This is called *dynamic power dissipation*. As f increases, the power dissipation and consequently the heat dissipation increases.

2. A small current called *leakage current* I_l flows continuously between different doped parts of a transistor. The leakage current

increases when a transistor becomes smaller. The leakage current causes a power dissipation proportional to $V_d I_l$.

3. During the small but finite time when a transistor gate toggles, a direct path exists between the voltage source and the ground. This causes a *short circuit current* I_{st} and leads to a power dissipation proportional to $V_d I_{st}$. This also increases as a transistor becomes small.

All the three components cause an increase in power dissipation when the size of transistors decreases. However, the dynamic power dissipation increases rapidly as the clock frequency increases. This causes an inordinate increase in the temperature of transistors in a chip as it heats up. When the frequency becomes 4 GHz, the heat generated will cause the transistors in the chip to almost melt unless the chip is cooled by special cooling techniques such as chilled water piping on which the chip is mounted. This is impractical for processors used in laptop computers. Thus, the only practical means of using the increasing number of transistors in a chip to increase the speed of execution of programs by microprocessors is to pack many processing cores inside one chip and make them work cooperatively to execute programs. As the same processor core is replicated in a chip, the design cost of chips could be controlled. This led to the emergence of multi-core microprocessors.

The dynamic power dissipation increases rapidly as the clock frequency increases causing an inordinate increase in the temperature of transistors in a chip as it heats up.

Structure of Multi-core Microprocessors

A simple definition of a multi-core microprocessor is [3]:

"It is an interconnected set of independent processing cores integrated on a single silicon chip. These processing cores communicate and cooperate with one another to execute one or more programs".

The key points in the above definition are:

1. Set of independent processors called processing cores are integrated in a single silicon chip.

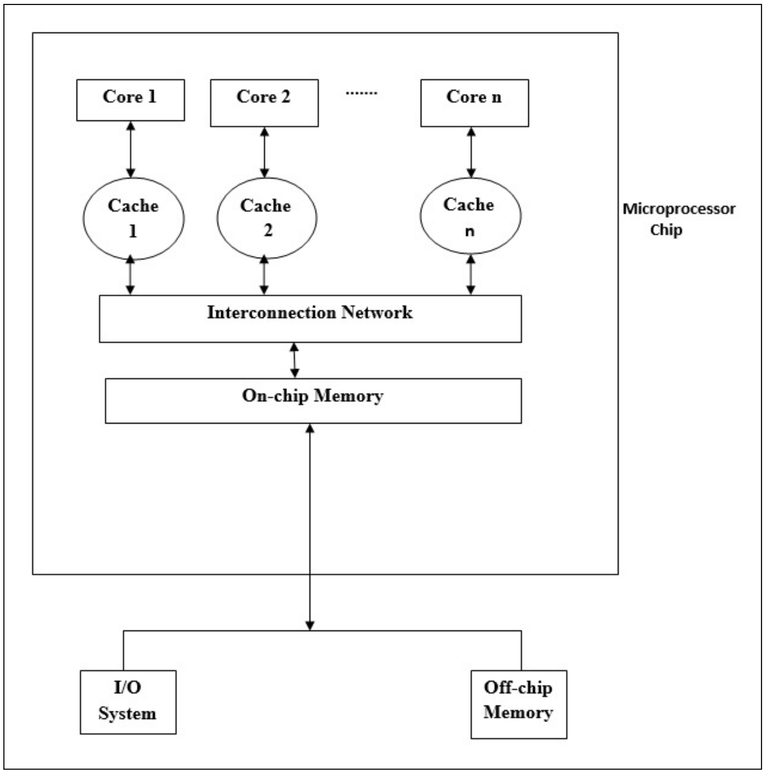

Figure 1. Structure of a multicore microprocessor.

2. The cores are interconnected and communicate with one another.
3. The processing cores cooperate to execute programs.

A block diagram of a general multi-core microprocessor is given in *Figure* 1.

A variety of multi-core microprocessors may be fabricated as the three crucial features of multi-core chips may be varied as listed below [3]:

Types of Processor Cores

1. A simple single threaded processor.
2. A multi-threaded complex processor.
3. A processor that is just an arithmetic logic unit with a multiplier and an accumulator (MAC unit).

4. A hybrid system combining simple and complex processors.

Number of Processing Cores

1. A small number of identical cores < 16.
2. A medium number of cores (of the order of hundreds).
3. A very large number (thousands) of multiply accumulate (MAC) units.

Interconnection of processor cores

1. The cores may be interconnected using a bus.
2. The cores may be interconnected by a ring, a crossbar switch, or they may form a grid.

Mode of Cooperation

1. Each processing core may be assigned separate programs. All the cores execute the programs assigned to them independently and concurrently. This mode of cooperation is called *Request-Level Parallel Processing*.
2. All processing cores execute the same program concurrently but on different data sets. This is called *Single Program Multiple Data* mode of parallel processing (called SPMD).
3. All processing cores execute a single instruction on multiple data streams or threads. This is called *Single Instruction Multiple Data* or *Thread Computing* (SIMD) or (SIMT).
4. A single program is split into a number of threads of instructions and stored in a shared memory. The threads are scheduled on different processors in such a way that all the processing cores cooperate to execute the program in minimum time by keeping all the cores concurrently processing threads. This is called *Multiple Instruction Multiple Data Stream* (MIMD) parallel processing. In this model, synchronising the execution of threads which

When each processing core is assigned separate programs, all the cores executing the programs assigned to them independently and concurrently, it is called *Request-Level Parallel Processing.*

execute concurrently in different cores is essential and requires clever programming.

Examining the description given above, it is clear that a large variety of multi-core microprocessors could be built. Normally when the number of cores is below 16, the cores are interconnected by a bus which is the simplest method of interconnecting processors. Beyond around 16 cores contention for using the common bus causes reduction of the speed of multi-core processors. Thus other types of interconnections are used. When there are large number of processors (exceeding 16) and an interconnection system other than a bus are used, they are classified as *many-core processors*. This definition is somewhat arbitrary. We, however, will use this definition in this article. In many core processors, the cores are interconnected by networks such as grid, ring, *etc*., and the programming method uses message passing between cores to cooperatively solve problems.

When the number of cores is of the order of thousands, then the cores are simple *Multiply Accumulate Units* (MAC units). These type of multi-core processors are useful for image processing in which a huge number of pixels can be independently processed by the cores and in calculations that require arithmetic operations using very large matrices. Such processors are called *General Purpose Graphics Processing Units* (GPGPUs). They are quite often used as co-processors in supercomputers.

Normally when the number of cores is below 16, the cores are interconnected by a bus which is the simplest method of interconnecting processors.

In this article, we will describe only multi-core microprocessors and not many-core microprocessors or GPGPUs

Design Consideration of Multi-core Microprocessor

While designing multi-core microprocessors it is necessary to consider the following:

1. Applications of the microprocessor.

2. The power dissipated by the system. It should not exceed 300 watts (a rule of thumb) if the microprocessor is to be air-cooled.

3. Spreading the heat dissipation evenly without creating hot spots in chips.

4. On-chip interconnection network should normally be planar as the interconnection wires are placed in a layer on top of the processor layer. In multi-core microprocessors, a bus is used to interconnect the processors.

5. The cost of designing multi-core processors is reduced if a single module, namely, the core, cache, and bus interconnection is replicated as the maximum design effort is required in designing a core and its cache and in verifying the design.

6. All multi-core microprocessors use an interconnection bus and share the main memory. The programming model used is what is known as a shared memory programming in which a program to be executed is stored in the memory shared by all the cores. When a shared memory program is used, the instruction set of a processing core should have instructions to *fork* from a process, *lock* a process, *unlock* a process and *join* a process after forking. These instructions are essential to write parallel programs.

The cost of designing multi-core processors is reduced if a single module, namely, the core, cache, and bus interconnection is replicated as the maximum design effort is required in designing a core and its cache and in verifying the design.

Bus Connected Multi-core Microprocessors

We depict in *Figure* 2 a typical bus connected multi-core processor with n cores. In 2001 IBM Power4 was the first dual-core processor to be released. The number of cores has doubled every few years. The speed of a multi-core microprocessor increases almost linearly as the number of cores increases as long as the cores do not have to communicate with one another. There are many problems where the cores can work independently without having to communicate with one another. We will describe such problems in a later section. However, many problems require the cores to communicate with one another and cooperate to solve a problem. In such cases the interconnection bus becomes a bottleneck as it is to be shared by all the cores. This limits the number of cores to about 16. Observe that each core in *Figure* 2 has its

The speed of a multi-core microprocessor increases almost linearly as the number of cores increases as long as the cores do not have to communicate with one another. There are many problems where the cores can work independently without having to communicate with one another.

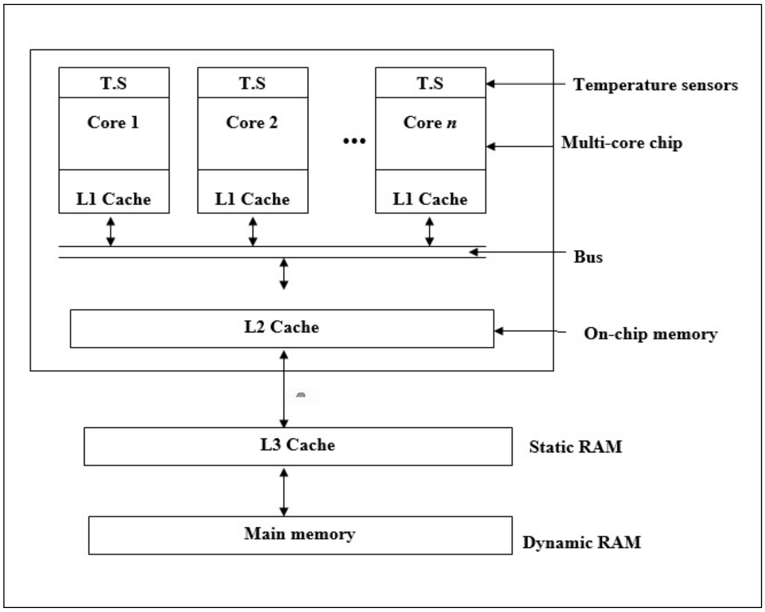

Figure 2. A bus connector multi-core microprocessor.

Temperature sensors are used to switch off a core when it gets heated and distribute its load to other cores.

own cache called an L1 cache. The L1 cache is divided into 2 parts – a data cache to store data that will be immediately required by the core and an instruction cache where a segment of a program needed immediately by the processing core is stored. The data and instruction cache sizes vary between 16 and 64 KB.

Another interesting part of a core is the temperature sensor. Temperature sensors are not normally used in single core systems. In multi-core systems, they are used particularly if the cores are complex processors such as Intel's i7 (a multi-threaded superscalar processor). Such processors tend to heat up at high frequencies (3 GHz). If a core heats up its neighbouring cores may also get affected and the chip may fail. Temperature sensors are used to switch off a core when it gets heated and distribute its load to other cores. In *Figure* 2 we have shown an L2 cache as a shared on-chip memory. This is feasible as the huge number of transistors available on a chip now may be used to fabricate L2 cache. Memory does not get heated unlike processors as they switch only when data is written or read from them. The memory outside the chip is divided into a static RAM used as an L3 cache

Microprocessors use an on-chip cache to place in it data and instructions immediately needed for computation.

and a larger dynamic RAM main memory. Some recent chips integrate L3 cache also inside the chip.

Cache Coherence Problem

Microprocessors use an on-chip cache to place in it data and instructions immediately needed for computation. This is because reading from and writing data in a cache is over hundred times faster than reading/writing data from/to memory outside a chip. It is not always easy to predict what data/instructions will be needed during the course of execution of a program. Normally programs are sequential and systems have evolved to make good predictions of what data would be needed and are to be kept in the cache. However, the prediction is not always correct and often the data needed by a program running in a processor is not in the cache. This is called a *cache-miss*. If there is a cache-miss, the required data has to be fetched from the off-chip memory which would slow down the execution of programs.

The situation in a shared memory multi-core processor is more complicated as each core has its own cache. When a core writes data in its cache other cores will not know this unless they are informed. Thus, another core that is working concurrently may read old data written earlier and not the latest data written by another core. In other words, all caches may not have the same value for a given variable, that is, all the caches may not be in a *coherent state*. The hardware has to ensure cache coherence to get correct results when a program is executed. Cache-coherence protocols have been designed and implemented in the cache controller hardware of multi-core microprocessors to ensure cache coherence and correct execution of parallel programs. There are many protocols each with its own advantages and disadvantages. Discussion of these protocols is outside the scope of this article and readers may refer to [3] in the suggested reading list to learn about them.

When a multi-core microprocessor is used as the main processor, normally they use a multi-tasking operating system that allows the execution of many tasks concurrently.

Programming Multi-core Microprocessor

There are three models of programming multi-core microprocessors. They are:

1. *Request Level Parallel Programming*

When a multi-core microprocessor is used as the main processor of a tablet, laptop, or desktop computer normally they use a multi-tasking operating system that allows executing many tasks concurrently. For example, a user may be using a word processor while listening to an Internet radio station and concurrently scanning a file for viruses. Each of these tasks may be assigned to a different core in a multi-core system. There will be hardly any interaction among these different tasks allowing all the cores to cater to *requests* for processing concurrently and independently tasks assigned to them. This style of parallel programming is called *Request Level Parallel Programming*. A four-core processor will be almost four times faster than a single-core processor assuming that the same core is replicated.

2. *Single Program Multiple Data Parallel Programming*

Many problems lead themselves to this mode of processing. For example, if a frequency chart of distribution by the age of a large population of 10 million persons is to be plotted, then data pertaining to 2.5 million persons may be assigned to each core of a four-core system. These four cores can concurrently execute identical programs to plot the frequency distribution by age for the population assigned to them. These results can then be merged. This style of parallel programming is called *Single Program Multiple Data programming* (SPMD). There are many problems in which a data set of n data items can be partitioned into k chunks each with (n/k) data items where k is the number of cores. Each data set may be assigned to a core that can work independently. Thus, we get almost k fold increase in speed of processing compared to a single core processor. Some examples where SPMD style of programming is effective are: Monte Carlo simulation, statistical calculations on large populations, protein folding, and image processing. In such problems, multi-core computers are very efficient as cores process tasks independently and rarely communicate. Such problems are often called embarrassingly parallel.

2. Multiple Instruction Multiple Data Programming

Two statements – *fork* and *join* – are added to a programming language to enable the creation of processes and for waiting for them to complete and re-join the main program.

In a shared memory multi-core processor in which a program stored in a shared memory is to be executed by the cores cooperatively, the program is viewed as a collection of processes. Each core is assigned a different process to be executed with the required data that is stored in the shared memory. The cores execute processes assigned to them independently. When all the cores complete the task assigned to them, they re-join to complete execution of the program. Two statements are added to a programming language to enable the creation of processes and for waiting for them to complete and re-join the main program. These statements are *fork* to create a process and *join* when the invoking process needs the results from the invoked process to continue processing. For example, consider the following statements of a parallel program in which P1 and P2 are two processes to be executed in parallel [3]:

```
Core x          Core y
begin P1        begin P1
   ⋮               ⋮
fork P2;
   ⋮               ⋮
join P2;
   ⋮               ⋮
end P2          end P2
```

In the above program a process P1 is being executed in *Core x*. When it encounters the statement *fork* P2 it invoked a process P2 in another *Core y*. The processes in *Core x* and *Core y* execute concurrently. When the process P1 encounters *join* P2 statement, it examines whether the process P2 running in *Core y* has ter-

When multiple processes work concurrently in different cores and update data stored in the shared memory, it is necessary to ensure that a shared variable value is not initialised or updated independently and simultaneously by these processes.

minated. If yes P1 takes the result from *Core y* and continues processing. *Core y* is free to be assigned another process. If no then *Core x* waits until *Core y* completes P2 to get the results in order to continue processing.

When multiple processes work concurrently in different cores and update data stored in the shared memory, it is necessary to ensure that a shared variable value is not initialised or updated independently and simultaneously by these processes. We illustrate this by a parallel program to compute sum $\leftarrow sum + f(A) + f(B)$. Suppose we write the following parallel program:

```
Core x                          Core y
begin P1                        begin P2
      ⋮                           ⋮
fork P2;                        sum ← $sum + f(B)$;
      ⋮                               ⋮
sum ← $sum + f(A)$;             end P2
      ⋮
join P2;
      ⋮
end P1
```

Suppose process P1 loads sum in its register to add $f(A)$ to it. If process P2 also loads sum in its local register to add $f(B)$. Process P1 will be having $sum + f(A)$ in its local register, and P2 will have $sum + f(B)$ in its local register. If both store the result back in the shared memory the memory will have either $sum + f(A)$ or $sum + f(B)$ depending on which process stored the result first. The result expected is however $sum+f(A)+f(B)$. We have to thus ensure that the two processes store sum in the main memory in a sequential order. To ensure this parallel programming languages add a statement lock $< variablename >$. If a process locks a variable name no other process can access this variable until it is unlocked by the process that locked it. This ensures that only one process can update a shared variable name at a time. The program to compute $sum + f(A) + f(B)$ is written below using the lock and unlock statements.

```
Core x                          Core y
begin P1                        begin P2
      ⋮                          ⋮
fork P2;                        lock sum;
      ⋮                         $sum \leftarrow sum + f(B)$;
lock sum;                       unlock sum;
$sum \leftarrow sum + f(A)$;     ⋮
unlock sum;                     end P2
      ⋮
join P2;
      ⋮
end P1
```

In the above program, the process that reaches *lock* sum first will update it. Until it is *unlocked* sum cannot be accessed by any other process using proper locking mechanism. After it is *unlocked* the other process can update it. This serialises the operation on a shared variable. As was pointed out earlier in this article the machine instructions of cores to be used in a multi-core microprocessor normally incorporate *fork*, *join*, *lock* and *unlock* instructions.

An important requirement to correctly execute multiple processes concurrently is known as *sequential consistency* [3]. It is defined as:

"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operation of all the processors were executed in some sequential order and the operations of each individual processor occur in this sequence in the order specified by its program".

The machine instructions of cores to be used in a multi-core microprocessor normally incorporate in addition to *fork*, *hand*, *join*, *lock* and *unlock* instructions.

In order to ensure this in hardware, each processor must appear to issue and complete memory operations one at a time in program

A multiprocessor is sequentially consistent if the result of any execution is the same as if the operation of all the processors were executed in some sequential order and the operations of each individual processor occur in this sequence in the order specified by its program.

order. Hardware protocol used to ensure cache coherence will ensure sequential consistency [3].

Conclusions

Multi-core microprocessors have now become the main stream. Their major advantages are that they work faster on many applications. As clock speed is below 4 GHz, they do not become too hot. It is possible to switch a job from a core that is hot to another relatively cool core to prevent failure of the microprocessor. Operating Systems now support parallel execution of programs in multi-core microprocessors. Design cost of multi-core processors can be kept under control as identical cores are replicated on a chip. The major disadvantage of multi-core computers is the difficulty in programming general problems. Scheduling tasks to optimize the utilisation of all cores is not easy. Request level programming tasks and tasks that can be solved using SPMD mode of programming are easy to schedule using all the cores and are most often used.

Acknowledgment

I thank Dr. S Balakrishnan, Prof. C Siva Ram Murthy, and Dr. S Ramani for reviewing this article and their constructive comments that helped me to improve this article.

Address for Correspondence
V Rajaraman
Honorary Professor
Super Computer Education & Research Centre
Indian Institute of Science
Bengaluru 560 012
Email:
rajaram@serc.iisc.in

Suggested Reading

[1] **M Mitchell Waldrop, The Chips are Down for Moore's Law, *Nature*, Vol.530, 11 Feb, pp.144–147, 2016.**

[2] **Vankat Arun, Multithreaded Processors, *Resonance*, Vol.20, No.9, pp.344–55, 2015.**

[3] **V Rajaraman, and C Siva Ram Murthy, *Parallel Computer Architecture and Programming*, 2nd edition, PHI Learning, Delhi, July 2016.**

[4] **G Blake, R G Dreslinski and T Mudge, A Survey of Multicore Processors, *IEEE Signal Processing*, Vol.26, No.6, pp.26–37, Nov.2009.**

[5] **P J Denning and J B Dennis, Resurgence of Parallelism, *Communications of ACM*, Vol.53, No.6, pp.30–32, August 2010.**