© 2014 IJIRT | Volume 1 Issue 6 | ISSN : 2349-6002

SHARED MEMORY MULTIPROCESSOR-A BRIEF STUDY

Shivangi Kukreja,Nikita Pahuja,Rashmi Dewan
Student,Computer Science&Engineering,Maharshi Dayanand University Gurgaon,Haryana,India

***Abstract*- The intention of this paper is to provide an overview on the subject of Advanced computer Architecture. The overview includes its presence in hardware and software and different types of organization. This paper also covers different part of shared memory multiprocessor. Through this paper we are creating awareness among the people about this rising field of multiprocessor. This paper also offers a comprehensive number of references for each concept in SHARED MEMORY MULTIPROCESSOR.**

I. INTRODUCTION

In the mid-1980s, shared-memory multiprocessors were pretty expensive and pretty rare. Now, as hardware costs are dropping, they are becoming commonplace. Many home computer systems in the under-$3000 range have a socket for a second CPU. Home computer operating systems are providing the capability to use more than one processor to improve system performance. Rather than specialized resources locked away in a central computing facility, these shared-memory processors are often viewed as a logical extension of the desktop. These systems run the same operating system (UNIX or NT) as the desktop and many of the same applications from a workstation will execute on these multiprocessor servers.

Typically a workstation will have from 1 to 4 processors and a server system will have 4 to 64 processors. Shared-memory multiprocessors have a significant advantage over other multiprocessors because all the processors share the same view of the memory, as shown in (Reference).

These processors are also described as uniform memory access (also known as UMA) systems. This designation indicates that memory is equally accessible to all processors with the same performance.

The popularity of these systems is not due simply to the demand for high performance computing. These systems are excellent at providing high throughput for a multiprocessing load, and function effectively as high-performance database servers, network servers, and Internet servers. Within limits, their throughput is increased linearly as more processors are added.

In this paper we are not so interested in the performance of database or Internet servers. That is too passé; buy more processors, get better throughput. We are interested in pure, raw, unadulterated compute speed for our high performance application. Instead of running hundreds of small jobs, we want to utilize all $750,000 worth of hardware for our single job.

The challenge is to find techniques that make a program that takes an hour to complete using one processor, complete in less than a minute using 64 processors. This is not trivial. Throughout this paper so far, we have been on an endless quest for parallelism.

The cost of a shared-memory multiprocessor can range from $4000 to $30 million. Some example systems include multiple-processor Intel systems from a wide range of vendors, SGI Power Challenge Series, HP/Convex C-Series, DEC AlphaServers, Cray vector/parallel processors, and Sun Enterprise systems. The SGI Origin 2000, HP/Convex Exemplar, Data General AV-20000, and Sequent NUMAQ-2000 all are uniform-memory, symmetric multiprocessing systems that can be linked to form even larger shared nonuniform memory-access systems. Among these systems, as the price increases, the number of CPUs increases, the performance of individual CPUs increases, and the memory performance increases.

II. SHARED MEMORY MULTIPROCESSOR ORGANIZATION

2.1) UMA

Uniform memory access (UMA) is a shared memory architecture used in parallel computers. All the processors in the UMA model share the physical memory uniformly. In a UMA architecture, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data. Uniform memory access computer architectures are often contrasted with non-uniform memory access (NUMA) architectures. In the UMA architecture, each processor may use a private cache. Peripherals are also shared in some fashion. The UMA model is suitable for general purpose and time sharing applications by multiple users. It can be used to speed up the execution of a single large program in time critical applications.

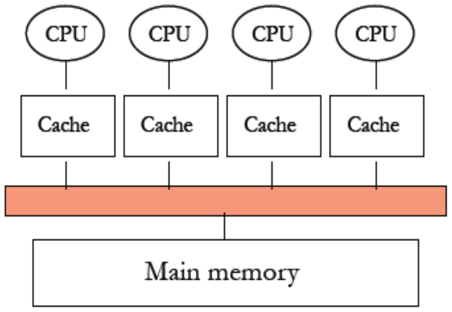

FIG 1

2.2) NUMA

Non-uniform memory access (**NUMA**) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). The benefits of NUMA are limited to particular workloads, notably on servers where the data are often associated strongly with certain tasks or users.

NUMA architectures logically follow in scaling from symmetric multiprocessing (SMP) architectures. They were developed commercially during the 1990s by Burroughs (later Unisys), Convex Computer (later Hewlett-Packard), Honeywell Information Systems Italy (HISI) (later Groupe Bull), Silicon Graphics (later Silicon Graphics International), Sequent Computer Systems (later IBM), Data General (later EMC), and Digital (later Compaq, now HP). Techniques developed by these companies later featured in a variety of Unix-likeoperating systems, and to an extent in Windows NT

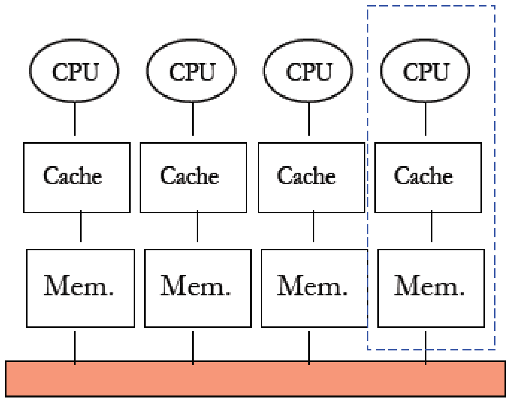

FIG 2

2.2.1) CCNUMA

Nearly all CPU architectures use a small amount of very fast non-shared memory known as cache to exploit locality of reference in memory accesses. With NUMA, maintaining cache coherence across shared memory has a significant overhead. Although simpler to design and build, non-cache-coherent NUMA systems become prohibitively complex to program in the standard von Neumann architecture programming model.

Typically, ccNUMA uses inter-processor communication between cache controllers to keep a consistent memory image when more than one cache stores the same memory location. For this reason, ccNUMA may perform poorly when multiple processors attempt to access the same memory area in rapid succession. Support for NUMA in operating systems attempts to reduce the frequency of this kind of access by allocating processors and memory in NUMA-friendly ways and by avoiding scheduling and locking algorithms that make NUMA-unfriendly accesses necessary.

Alternatively, cache coherency protocols such as the MESIF protocol attempt to reduce the communication required to maintain cache coherency. Scalable Coherent Interface (SCI) is an IEEE standard defining a directory-based cache coherency protocol to avoid scalability limitations found in earlier multiprocessor systems. For example, SCI is used as the basis for the NumaConnect technology.

As of 2011, ccNUMA systems are multiprocessor systems based on the AMD Opteron processor, which can be implemented without external logic, and the IntelItanium processor, which requires the chipset to support NUMA. Examples of ccNUMA-enabled chipsets are the SGI Shub (Super hub), the Intel E8870, the HP sx2000 (used in the Integrity and Superdome servers), and those found in NEC Itanium-based systems. Earlier ccNUMA systems such as those from Silicon Graphics were based on MIPS processors and the DECAlpha 21364 (EV7) processor.

Topology of a ccNUMA Bulldozer server.

FIG 3

III. HARDWARE SUPPORT

3.1) CACHE COHERENCE

in a shared memorymultiprocessor system with a separate cache memory for each processor, it is possible to have many copies of any one instruction operand: one copy in the main memory and one in each cache memory. When one copy of an operand is changed, the other copies of the operand must be changed also. Cache coherence is the discipline that ensures that changes in the values of shared operands are propagated throughout the system in a timely fashion.

There are three distinct levels of cache coherence:

- every write operation appears to occur instantaneously
- all processors see exactly the same sequence of changes of values for each separate operand
- Different processors may see an operation and assume different sequences of values; this is considered to be a non-coherent behavior.

In both level 2 behavior and level 3 behaviors, a program can observe stale data. Recently, computer designers have come to realize that the programming discipline required to deal with level 2 behaviors is sufficient to deal also with level 3 behavior. Therefore, at some point only level 1 and level 3 behavior will be seen in machines.

3.1.1) CACHE COHERENCE MODEL

- **Idea:**
  - ➢ Keep track of what processors have copies of what data.
  - ➢ Enforce that at any given time a single value of every data exists.
  - ➢ By getting rid of copies of the data with old values - invalidate protocols.
  - ➢ By updating everyonefs copy of the data - update protocols.

- **In practice:**
  - ➢ Guarantee that old values are eventually invalidated/updated (write propagation)(recall that without synchronization there is no guarantee that a load will return the new value anyway)
  - ➢ Guarantee that only a single processor is allowed to modify a certain datum at any given time (write serialization)
  - ➢ Must appear as if no caches were present

- **Implementation**
  - ➢ Can be in either hardware or software, but software schemes are not very

practical (and will not be discussed further in this course)

- ➢ Add state bits to cache lines to track state of the line
- ➢ Most common: Modified, Owned, Exclusive, Shared, Invalid
- ➢ Protocols usually named after the states supported
- ➢ Cache lines transition between states upon load/store operations from the local processor and by remote processors
- ➢ These state transitions must guarantee the invariant: no two cache copies can be simultaneously modified
- ➢ SWMR: Single writer multiple readers

Example - MSI protocol, MESI protocol.

3.2) MEMORY CONSISTENCY

It basically covers

- When should writes propagate?
- How are memory operations ordered?
- What value should a read return?

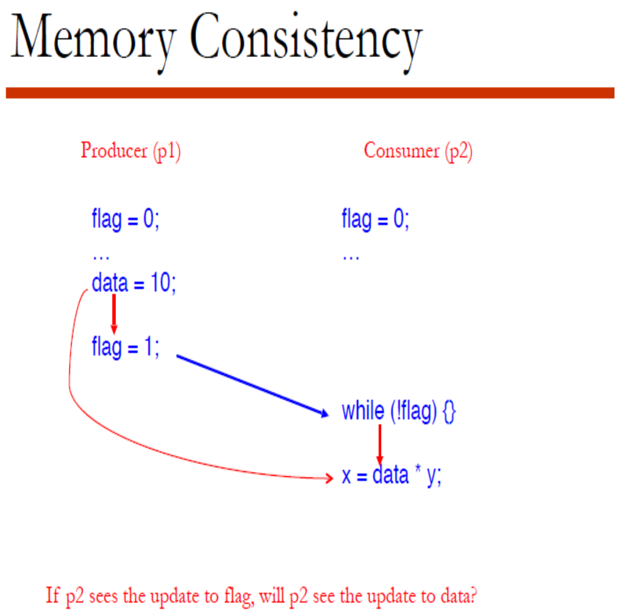

FIG 4

3.3) PRIMITIVE SYNCHRONIZATION

It covers:

- Memory fences: memory ordering on demand
- Read-Modify-writes: support for locks (critical sections)

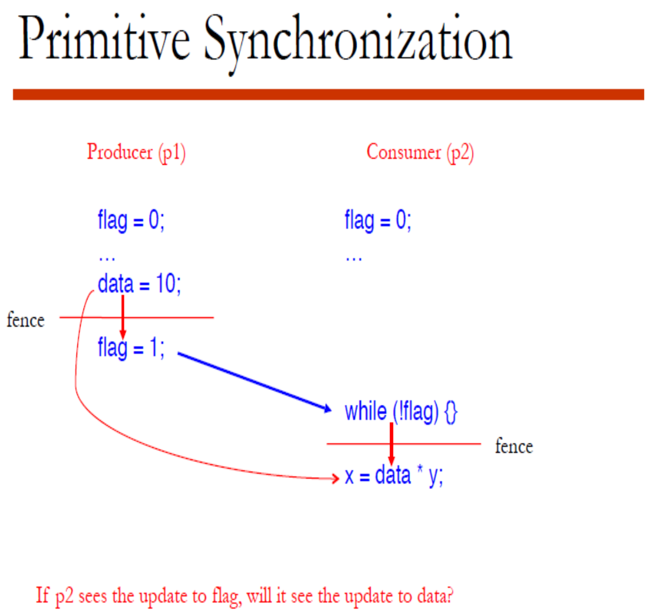

FIG 5

IV. IN SOFTWARE

In computer software, *shared memory* is either

- a method of inter-process communication (IPC), i.e. a way of exchanging data between programs running at the same time. One process will create an area in RAM which other processes can access, *or*
- a method of conserving memory space by directing accesses to what would ordinarily be copies of a piece of data to a single instance instead, by using virtual memory mappings or with explicit support of the program in question. This is most often used for shared libraries and for XIP.

Since both processes can access the shared memory area like regular working memory, this is a very fast way of communication (as opposed to other mechanisms of IPC such as named pipes, Unix domain sockets or CORBA). On the other hand, it is less powerful, as for example the communicating processes must be running on the same machine (of other IPC methods only Internet Domain sockets (not UNIX sockets) can use a computer network), and care must be taken to avoid issues if processes sharing memory are running on separate CPUs and the underlying architecture is not cache coherent.

IPC by shared memory is used for example to transfer images between the application and the X server on Unix systems, or inside the IStream object returned by CoMarshalInterThreadInterfaceInStream in the COM libraries under Windows.

Dynamic libraries are generally held in memory once and mapped to multiple processes, and only pages that had to be customized for the individual process (because a symbol resolved differently there) are duplicated, usually with a mechanism known as copy-on-write that transparently copies the page when a write is attempted, and then lets the write succeed on the private copy.

4.1) SUPPORT ON UNIX PLATFORMS

POSIX provides a standardized API for using shared memory, *POSIX Shared Memory*. This uses the function shm_open from sys/mman.h.[1] POSIX interprocess communication (part of the POSIX:XSI Extension) includes the shared-memory functions shmat, shmctl, shmdt and shmget.UNIX System V provides an API for shared memory as well. This uses shmget from sys/shm.h. BSD systems provide "anonymous mapped memory" which can be used by several processes.

The shared memory created by shm_open is persistent. It stays in the system until explicitly removed by a process. This has a drawback that if the process crashes and fails to clean up shared memory it will stay until system shutdown. To avoid this issue mmap can be used to create a shared memory. Two communicating processes should open a temporary file with the same name and do mmap on it to get a file mapping in the memory. As a result changes in mapped memory are visible by both processes at the same time. The advantage of these approaches is that when both processes exit, OS will automatically close the files and remove shared memory.

Recent Linux distributions based on the 2.6 kernel have started to offer /dev/shm as shared memory in the form of a RAM disk, more specifically as a world-writable directory (a directory in which every user of the system can create files) that is stored in memory. Both the RedHat and Debian based distributions include it by default. Support for this type of RAM disk is completely optional within the kernel configuration file.

SUPPORT ON OTHER PLATFORMS

On Windows the function CreateSharedMemory can be used to create a shared memory. Alternatively one can use CreateFileMapping and MapViewOfFile functions

Some C++ libraries provide a portable and object-oriented access to shared memory functionality. For example, Boost contains Boost.Interprocess C++ Library.Qt provides QSharedMemory class.[9]

There is native support for shared memory also in programming languages besides C/C++. For example, PHP provides API to create a shared memory, similar to POSIX functions.

SUMMARY

In computing, **shared memory** is memory that may be simultaneously accessed by multiple programs with intent to provide communication among them or avoid redundant copies. Shared memory is an efficient means of passing data between programs. Depending on context, programs may run on a single processor or on multiple separate processors.

Using memory for communication inside a single program, for example among its multiple threads, is also referred to as shared memory.

DISCLOSURE STATEMENT

There is no financial support for this research work from the funding agency.

ACKNOWLEDGMENTS

We thank our guide for his timely help, giving outstanding ideas and encouragement to finish this research work successfully.

SIDE BAR

Comparison: it is an act of assessment or evaluation of things side by side in order to see to what extent they are similar or different. It is used to bring out similarities or differences between two things of same type mostly to discover essential features or meaning either scientifically or otherwise.

Content: The amount of things contained in something. Things written or spoken in a book, an article, a programme, a speech, etc.

DEFINITION

- **SHARED:** use, occupy, or enjoy (something) jointly with another or others.
- **MEMORY:**the faculty by which the mind stores and remembers information

- **MULTIPROCESSOR:** a computer with more than one central processor.
- **DYNAMIC:** characterized by constant change, activity, or progress.

REFERENCES

1. Documentation of shm_open from the Single UNIX Specification
2. Robbins, Kay A.; Steven Robbins (2003). *UNIX systems programming: communication, concurrency, and threads* (2 ed.). Prentice Hall PTR.p. 512.ISBN 978-0-13-042411-2.Retrieved 2011-05-13. "The POSIX interprocess communication (IPC) is part of the POSIX:XSI Extension and has its origin in UNIX System V interprocess communication."
3. Shared memory facility from the Single UNIX Specification.
4. Stevens, Richard (1999). *UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications.* (2 ed.). Prentice Hall PTR.p. 311.ISBN 0-13-081081-9.
5. ChristophRohland, Hugh Dickins, KOSAKI Motohiro. "tmpfs.txt".kernel.org.Retrieved 2010-03-16.

RELATED REFERENCES

1. CreateSharedMemory function from Win32-SDK
2. Creating Named Shared Memory from MSDN.
3. Boost.Interprocess C++ Library
4. QSharedMemory Class Reference
5. Shared Memory Functions in PHP-API