**Non­-Uniform Memory Access (NUMA) Architecture and Optimization**

Chenghu He

G45573591

George Washington University

**Abstract**

The non-uniform memory access (NUMA) is a technology for memory performance improvement in multiprocessor computer architecture. The multiprocessor architecture is widely used in high ­performance servers and the CPUs are very sensitive to memory bandwidth. The NUMA structure allows each CPU has an individual memory controller and shares the memory with other controllers. This structure can avoid the bottleneck of a single memory controller. The tradeoff of NUMA is that the data access from one CPU to other CPU’s memory unit will cast longer time than the access from the CPU to its own memory unit. The challenge is that the operating system should optimize the memory management for the NUMA structure. This paper will introduce the architecture of NUMA, analyze the memory technology related to NUMA, and discuss how to support and optimize the NUMA structure in systems.

*Keywords* *:* NUMA, memory, CPU, server system

1. **Introduction**

The computer architecture is based on a structure of storage and processor. Modern computers use memory to store both data and instructions for program running. The process of running a program can be simply divided into several steps. First, when the computer is running, the machine needs to read a block of memory for getting the detail of instructions. Each instruction may have several parts for the action of data manipulation. Second, the instruction may need the data from a selected address from the memory. Third, some instruction will write the result into an address of memory for storage. All the steps show that the access of memory is very frequent in the running of a normal computer. Even if the computer is only used for computing, it still needs memory to store all the steps and some results.

When the speed of CPU become faster, it is very expensive to keep the memory to the same speed of CPUs. Though the cache is used in the CPU, the size of cache is still much less than the necessary memory. For example, when a modern commodity computer equips with a 2.5GHz CPU core, the DDR3 memory may run in 1.6GHz, and the bandwidth is always insufficient. This problem is not critical in limited number of processors. But when a multiple-processor structure is designed for a powerful server, the processors have to share the memory for their running. In a traditional structure, the memory in a system is managed by a central memory controller. It is very difficult to provide a high bandwidth in that controller.

To address this problem, several proposals had been discussed. One of them is providing each processor an individual memory controller. There are many benefits of using this structure. First, the memory controller is put near the processor, so it is easier to access memory in a higher speed. Second, the total memory bandwidth is expanded in this way without design a new memory controller. The downside is that the programs may need to access some data from other processers’ memory when it runs in one of the processers. And the latency is not uniform when the program uses different controllers. So, this new memory architecture has been named non-uniform memory access (NUMA), and the normal memory with a central is called uniform memory access (UMA). The challenge of (NUMA) is trying to use one processor’s own memory as frequent as possible.

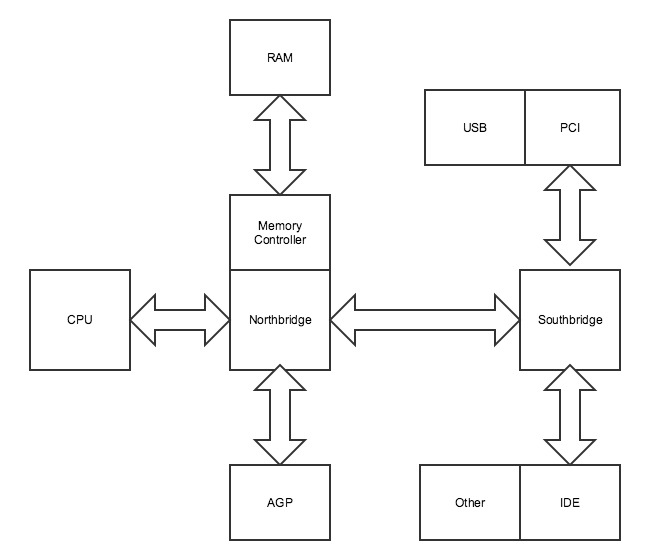
This paper is focused on the implementation of the NUMA structure and optimization. The structure will compare designs of several main companies and discuss the tradeoffs. Section 2 will start from the history of memory controller and then describe all the existed NUMA hardware structures. Section 3 is written about the system support of NUMA and some optimization ways. The optimization will focus on how to reduce the number of access to slow memory in the program design. There are many useful applications in parallel computing in section 4. The performance is evaluated in section 5. In section 6, all the researches are concluded together.

1. **Hardware**

The issue of NUMA comes with the modification of computer hardware. When a new technique appears, it is always designed to solve a certain problem. The new generate of memory controller is one of this kind of products, and it do solve the problem of limited bandwidth in memory access. But this change also causes a new issue that is the non-uniform distribution of memory access. In the worst case, if all the requests are from other CPU’s memory block, they will cause higher latency. In this part, we will discuss this issue and figure out how the hardware of NUMA works in modern computers.

* 1. **History of Memory Controller**

Diagram 1 North and South Bridge and Memory Controller



The main memory which is installed as several chips in the mother board is a kind of semiconductor circuits and the controller is the logic circuit which takes responsibility of managing them. The controller will provide functions such as basic reading and writing, and then synchronize the working of multiple chips or blocks. The controller is also very important to the speed of memory running. Though the clock frequency of chips is the main issue to improve the memory performance, the controller is also become more complicated when the speed is higher.

There are two main approach of designing memory controller. One is put the controller outside the main processor and the other is put it inside core processor. In history, the controller has been outside processor for a long time. And the computer was used to a structure called north and south bridges. Diagram 1 is a typical north and south bridge structure.

From diagram 1, the memory controller is inside the Northbridge chip and working with RAM chips for memory access. This architecture has a long time of history when the north and south bridges were the main methods of data switches in the computer system. This design is different of modern computer architectures. The main problem is that the memory bandwidth is limited by the performance of Northbridge chip which is normally much slower than CPU. Another problem is that the memory will share the bandwidth with other requests such as graphic interfaces. And it is worse when there are multiple processors in one system.

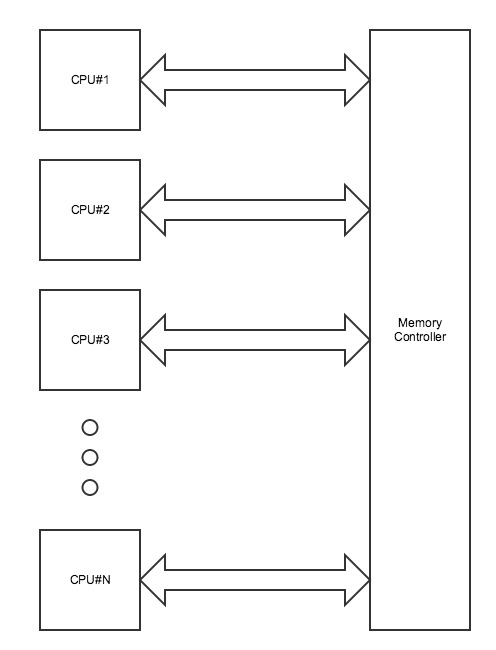
Modern computer processors often include a inside chip memory controller as parts of CPU functions. The AMD’s Athlon64 is a typical use case of this design approach. The Intel’s Core series CPU also use this technique called integrated memory controller (IMC). And now, the IMC is the standard of modern memory controller structure. The benefit of the IMC is reducing the latency between CPU and memory. It also give memory chips an individual bandwidth for improving performance.

Though there are many technical improvements for memory controllers in recent years. For example, the memory uses DDR technical to double the running speed and use Dual-Chanel memory to give higher bandwidth. But the improvement with IMC is more significant because it opens a way to high performance in parallel computing with multiple processors.

* 1. **Uniform Memory Access (UMA)**

The parallel computing is not restricted in an individual computer. This technique has a long history of using multiple computers. But, nowadays it is very common for a single computer running with multiple cores in one chip. That means the most common situation is that running programs with multiple processor in one system. Before the multiple core CPU come to smart devices such as cell phones, the multi-processor techniques are first used for server machines. Even before the memory controller is integrated into the CPU model, there are also many use cases of installing several processors in one mother board. That is why we have the UMA memory access standard before the appearance of NUMA memory.

Diagram 2 A UMA structure Example



The structure of a typical UMA structure is showed in diagram 2.

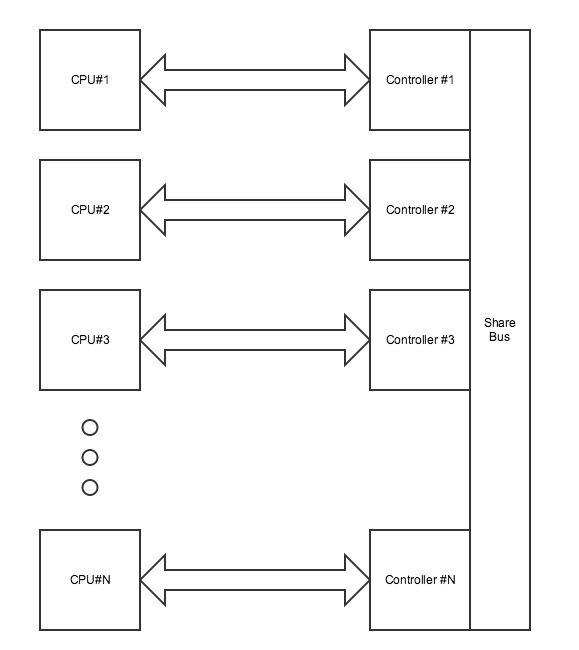
From the diagram 2, the system has more than one CPU in the same memory controller. When each CPU needs to access the memory, it will use the controller for this action. This structure shows that every CPU has the same speed to request the memory controller. And the memory controller will request the memory block at the same speed definitely because the real RAM block is only connect to the controller. That means all the CPUs have the same speed of reading and writing memory. That why we call this structure the UMA.

Actually, the UMA is the default mode of memory access, and it is not need to modify any previous hardware and software for the parallel function of multiple processors. The downside is UMA is already mentioned, that is the limit of bandwidth in the single memory controller is hard to improve. Furthermore, it is very complicated for one controller to deal with more than one processor. Indeed, the over head of switching among processors will cause higher latency in the programs. Then, the NUMA structure appeared in severs with multiple processors.

* 1. **Non-Uniform Memory Access (NUMA)**

This research will show the difference between traditional UMA structure and NUMA structure. The typical NUMA connection is showed in Diagram 3. From this diagram we can see that there are multiple controllers in NUMA compared to UMA. Actually, this is the consequence of IMC structure which is mentioned in the history of memory controller. Every CPU has an individual memory controller means it is very fast to access the memory part which is located in its own controller. And it is also simplify the structure of each controller so it is possible to run the controller in a higher speed. Diagram 3 also shows a memory share bus that is used to access the memory from other memory controller. It is obvious a slow way to access data, but sometimes this kind of access may be unavoidable. So there are works for software design.

Diagram 3 NUMA structure



This structure has a high efficiency when the data is only read from one CPU’s own controller. The design is focused on providing an easy way to combine the power of calculation by adding more processors. Actually there are many programs which only need limited memory locations but will read and write them in high frequency. This kind of programs is the best programs for the NUMA structure because the improvement of performance is obviously in these cases.

In other more common cases, such as running commercial software, the situation become very complicated. It is hard to say the NUMA make the program runs faster or slower. The access from one CPU’s own controller is faster than the UMA, but the access from other CPUs’ controllers is slower. But the NUMA is widely spread because when there are more and more processors in one system, the limit of memory bandwidth is more and more important. That means if we want to solve to problem of memory bandwidth in a decent way when there are a lot of CPUs, the NUMA structure becomes the only choice. So, the application of NUMA must be considered as an objective problem. And the following research should also work on this base situation.

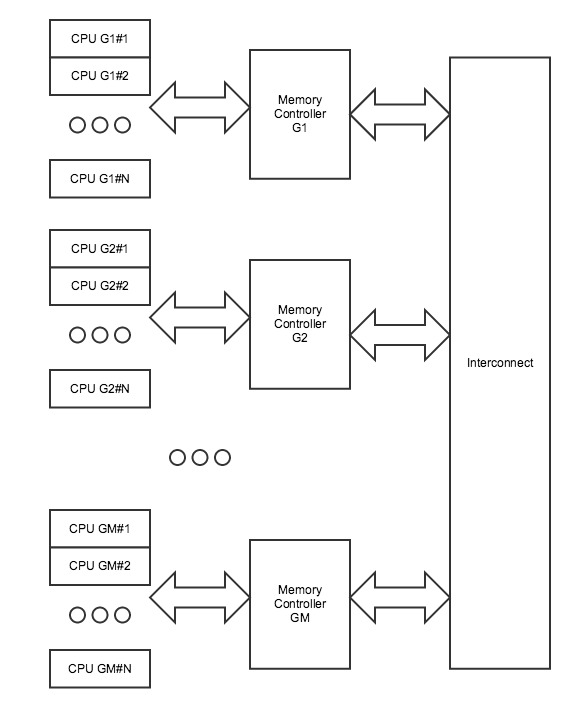
* 1. **CC-NUMA**

The development of NUMA structure also lead to some issues such as cache design improvement. When the cache of different CPUs is individual, there is a huge waste of running performance, then the cache coherent NUMA (CC-NUMA) is appeared as an alternative plan of normal NUMA structure.

To implement CC-NUMA structure, an inter-CPU bus is designed to connect each CPU for sharing the cache information. Though there are many good cases in using this cache sharing bus, it may also have many conflicts in the running and that may cause potential lower performance for special programs. Then, some protocols are also developed. MESIF protocol is one of them. It is used to reduce the number of cache coherence requirements in the process of running programs. This technique has already been implemented in the Intel Itanium and AMD Opteron processors as a good improvement of NUMA structure.

* 1. **Intel Architectures**

Diagram 4 QPI Connections



The NUMA compatibility is provided in Intel series of CPU since 2007. The CPUs include the Nehalem and Tukwila series. Both x86 and Itanium structure is supported. Intel also shares a common chipset for these CPUs. To provide the connections among lots of CPUs, a new technique named quick path interconnect (QPI) is introduced at that time. The design of this architecture is demonstrated in diagram 4.

From the diagram 4, the memory controller is also shared in several CPUs. These CPUs are defined in one group and share a same high performance memory controller. Then, the interconnection is used to share the memory among different controllers. This structure is more flexible than the standard NUMA proposal. As the number of controllers is limited in a complex system, the number of inter-controller memory access is also reduced. On the other hand, the processors in one controller are also limited, so the controller’s performance could be adjusted in an acceptable range. That’s the benefit of Intel’s proposal.

Though these features are very attractive, it is still a NUMA structure with the same problem of normal NUMA, and the performance is also based on good design of systems and applications above this structure.

* 1. **AMD Architectures**

The proposal of putting memory controllers in a commercial CPU is started by AMD from 2003. At that time, AMD announced the Opteron processor with a technique called HyperTransport (HT). And the AMD’s NUMA implementation is actually based on that HT. Different with Intel’s QPI, the HT is not only used to connect multiple CPUs, it is also used to connect the chipsets. This technique is focused on provide a high speed connections for wide bandwidth. This is a fundamental technique for the development of NUMA memory access.

1. **Systems**

To be straight forward, the best way is limited the CPU only reads data from its own controller, but it is hard for programmer’s to write programs in this way. Modern programming language is highly abstracted, so there is no concept of memory access management in program design. It is also hard to write programs only for a special use case and platform. So, this kind of work is the job of operating system and compilers. The best optimization way should avoid affecting the programmer’s work. Then, this part is focused on the existing techniques and optimization and provides the way to improve the performance of NUMA structures.

* 1. **Microsoft Support**

There are many operating systems provided by Microsoft include the support of NUMA structure. And these systems also provide NUMA APIs to programs for management. The Table 1 is a list of NUMA APIs provided by Microsoft in its operating systems.

Table 1 Microsoft NUMA APIs

(Referred from MSDN website)

|  |  |
| --- | --- |
| Function | Description |
| AllocateUserPhysicalPagesNuma | Allocates physical memory pages to be mapped and unmapped within any Address Windowing Extensions (AWE) region of a specified process and specifies the NUMA node for the physical memory. |
| CreateFileMappingNuma | Creates or opens a named or unnamed file mapping object for a specified file, and specifies the NUMA node for the physical memory. |
| GetLogicalProcessorInformation | Retrieves information about logical processors and related hardware. |
| GetLogicalProcessorInformationEx | Retrieves information about the relationships of logical processors and related hardware. |
| GetNumaAvailableMemoryNode | Retrieves the amount of memory available in the specified node. |
| GetNumaAvailableMemoryNodeEx | Retrieves the amount of memory available in a node specified as a USHORT value. |
| GetNumaHighestNodeNumber | Retrieves the node that currently has the highest number. |
| GetNumaNodeProcessorMask | Retrieves the processor mask for the specified node. |
| GetNumaNodeProcessorMaskEx | Retrieves the processor mask for a node specified as a USHORT value. |
| GetNumaProcessorNode | Retrieves the node number for the specified processor. |
| GetNumaProcessorNodeEx | Retrieves the node number as a USHORT value for the specified processor. |
| GetNumaProximityNode | Retrieves the node number for the specified proximity identifier. |
| GetNumaProximityNodeEx | Retrieves the node number as a USHORT value for the specified proximity identifier. |
| MapViewOfFileExNuma | Maps a view of a file mapping into the address space of a calling process, and specifies the NUMA node for the physical memory. |
| VirtualAllocExNuma | Reserves or commits a region of memory within the virtual address space of the specified process, and specifies the NUMA node for the physical memory. |

From table 1, we can see the Microsoft defined a concept named Nodes for the management of processors in NUMA servers. One node is a small system of CPUs in which CPUs can share their memory access in high speed, and then the connection among nodes is regarded as slowly interconnected. This definition is very similar to Intel’s QPI proposal. Then, Microsoft’s proposal also requires the Operating System to determine the group of threads which needs to use the same part of memory. Once the OS know the thread group, the system should let them run in the same Node of CPUs. The APIs allow developer to select a good way to use their programs a particular group of CPUs in their running times.

For example, the developer can use GetNumaHighestNodeNumber to determine whether a system is supporting NUMA usages. Then, they can get a list of NUMA Nodes by using GetProcessAffinityMask, GetNumaProcessorNode, and GetNumaNodeProcessorMask. After that, the programmer can choose one Node to run their programs by setting SetProcessAffinityMask and SetThreadAffinityMask. In this way, the program should have a significant improvement of performance on proper NUMA hardware.

* 1. **Linux Support**

The NUMA support is implemented in the Linux Kernel by building with an option named CONFIG\_NUMA. When a program would like to use system calls implemented to support NUMA, the library libnuma should also be included in the compiling process by adding –lnuma. The related system calls is listed in Table 2.

Table 2 The NUMA system calls in Linux

(Referred from Linux Man files)

|  |  |
| --- | --- |
| System Calls | Descriptions |
| get\_mempolicy() | Retrieve the NUMA policy of the calling process or of a memory address, depending on the setting of flags |
| mbind() | Set the NUMA memory policy, which consists of a policy mode and zero or more nodes, for the memory range starting with addr and continuing for len bytes.  The memory policy defines from which node memory is allocated. |
| migrate\_pages() | Move all pages of the process pid that are in memory nodes old\_nodes to the memory nodes in new\_nodes.  Pages not located in any node in old nodes will not be migrated.  As far as possible, the kernel maintains the relative topology relationship inside old\_nodes during the migration to new\_nodes. |
| move\_pages() | Move the specified pages of the process pid to the memory nodes specified by nodes.  The result of the move is reflected in status. The flags indicate constraints on the pages to be moved. |
| set\_mempolicy() | Set the NUMA memory policy of the calling process, which consists of a policy mode and zero or more nodes, to the values specified by the mode, nodemask and maxnode arguments |

The Linux system also uses a file named /proc/[pid]/numa\_maps to show the status of NUMA nodes and allocations. This file is read-only and reading this file could get all the memory ranges used by processes. From the table 2, the usage of NUMA in Linux system is very clear. The user can get the information of memory nodes in the current system. Then the program can use set\_mempolicy() and mbind() to specify the memory nodes using in one process. This support provide tools to restrict the memory allocations for a particular program, then give the developer powerful tools to manage them. But this implementation is highly depended on the developer’s strategy. It just a tool to allocate memory for NUMA structure, it doesn’t do it automatically for users. That means developers should figure out their own plans to ultimate their hardware with careful design.

* 1. **VMware Support**

There are two types of virtual machine hypervisor in VMware’s products. The type 1 hypervisor is directly working on hardware and type 2 is working on other operating systems. People may only be familiar with type 2 virtual machine because that is very common in our daily life. But the type 1 virtual machine hypervisor is also widely used especially in enterprise-level servers. For the type 2 virtual machine, the support of NUMA is depends on the actual system running this virtual machine. But for the type 1 products such as ESX and ESXi, the NUMA management should be considered because these products already work like operating systems.

The VMware has developed a sophisticated NUMA scheduling algorithm to keep the balance of memory locality. There are three steps of implementation in VMware’s ESX/ESXi. The steps are showed in Table 3.

Table 3 VMware vSphere 4 NUMA Support

(Referred from VMware vSphere 4 website)

|  |  |
| --- | --- |
| Steps | Descriptions |
| Step 1 | Each virtual machine managed by the NUMA scheduler is assigned a home node.  A home node is one of the system’s NUMA nodes containing processors and local memory, as indicated by the System Resource Allocation Table (SRAT). |
| Step 2 | When memory is allocated to a virtual machine, the ESX/ESXi host preferentially allocates it from the home node. |
| Step 3 | The NUMA scheduler can dynamically change a virtual machine’s home node to respond to changes in system load.  The scheduler might move a virtual machine to a new home node to reduce processor load imbalance.  Because this might cause more of its memory to be remote, the scheduler might move the virtual machine’s memory dynamically to its new home node to improve memory locality.  The NUMA scheduler might also swap virtual machines between nodes when this improves overall memory locality. |

From the table 3, it shows that VMware has developed a smart NUMA scheduler for the virtual machines to keep memory locality. This scheduler is smart because it not only assign the memory access to nodes for applications. It also has the ability to move or swap virtual machines between two nodes for overall performance. Though the move looks inefficient, it actually balanced the load of different nodes. And this approach focused on improving the overall system performance. There are also many other benefits of VMware’s approach. For example, even if the OS running on VMware products doesn’t support NUMA, this system can also share the benefit of NUMA structure.

1. **Applications**

The benefits of using NUMA are based on the design of hardware and software. Nowadays, modern computers all have good solutions for NUMA structure. The operating systems are also ready for the NUMA optimization. Though there are automatic NUMA allocate policies, the question of how to use NUMA memory in a proper way is still a good question of software developers. Now, there are several good usages of NUMA structure in real applications, and these applications also show the performance improvement of NUMA.

* 1. **Java Garbage Collector**

The programming language Java tries to run all the programs in a abstract computing machine called Java virtual machine (JVM). This is the base of Java and its multi-platform features. In the Java virtual machine system, one important issue is to collect any used resources when they are not needed to use again. This job is called garbage collection. This feature can give programmer convenience when they try to allocate memory without remembering freeing them. The only tradeoff is that this garbage collector will cost some resources of CPU and memory.

To save the cost of garbage collector, Oracle has announced a technique name HotSpot to address a series of issues. The NUMA optimization is one of them. The allocator in JVM would take the advantage of NUMA structure and optimize the running time of garbage collector. The technical details are described in Table 4.

Table 4 Java HotSpot NUMA Collector Enhancements

(Referred from Oracle Web site)

|  |  |
| --- | --- |
| Features | Description |
| Feature 1 | The allocator controls the Eden space of the young generation of the heap, where most of the new objects are created. |
| Feature 2 | The allocator divides the space into regions each of which is placed in the memory of a specific node. |
| Feature 3 | The allocator relies on a hypothesis that a thread that allocates the object will be the most likely to use the object. |
| Feature 4 | To ensure the fastest access to the new object, the allocator places it in the region local to the allocating thread. |
| Feature 5 | The region can be dynamically resized to reflect the allocation rate of the application threads running on different nodes.  That makes it possible to increase performance even of single-threaded applications. |
| Feature 6 | In addition, “from” and “to” survivor spaces of the young generation, the old generation, and the permanent generation have page interleaving turned on for them.  This ensures that all threads have equal access latencies to these spaces on average. |

From table 4, it is obvious that Java has developed a detailed algorithm to achieve its NUMA-aware collector in JVM. This technique shows how a NUMA-aware software design can help improve performance of program running. From Oracle’s website, the new version of NUMA allocator has a 30% increase in 32 bit NUMA hardware and a 40% increase in 64 bit hardware.

* 1. **Hash Joins**

In the implementation of relational database management system, the operation of join is one of basic database functions. There are many algorithms to implement the join functions. Hash join is one of these efficient functions. The large database system often need to run hash join in multiple processors, that means the running of hash join often happens on computers with NUMA structure.

Previous implementation of hash join considered some environmental parameters, but basically developed in the UMA structure. Recently, some researchers have announced a new NUMA-aware implementation of hash join. Their approach is based on a lock-free synchronization mechanism. The steps of implementation of the synchronization mechanism are demonstrated in table 5.

Table 5 NUMA-aware Hash join synchronization steps

|  |  |
| --- | --- |
| Steps | Descriptions |
| Step 1 | Implement buckets as triples (h, k, v) |
| Step 2 | Use h as a marker which signals whether a bucket is empty or already in use |
| Step 3 | During the build phase, the threads first check if the marker is set |
| Step 4 | If the corresponding bucket is empty, then exchange the value zero by then hash value within an atomic Compare-and-Swap operation (CAS) |
| Step 5 | If the marker has already been set, the action fails and try again on the next write position |
| Step 6 | Once the CAS succeeds, the corresponding thread implicitly has exclusive write access to the corresponding bucket and no further synchronization is needed for storing the node |

From Table 5, a synchronizing mechanism is formed. Based on it, a NUMA-aware hash join algorithm is implemented. This implementation succeeded to have a higher performance.

1. **Conclusions**

This paper introduced the concepts of NUMA memory access and gave the design proposals of hardware and software. The non-uniform memory access (NUMA) is a technology for memory performance improvement in multiprocessor computer architecture. The multiprocessor architecture is widely used in high ­performance servers and the CPUs are very sensitive to memory bandwidth. The NUMA structure allows each CPU has an individual memory controller and shares the memory with other controllers. This structure can avoid the bottleneck of a single memory controller.

The tradeoff of NUMA is that the data access from one CPU to other CPU’s memory unit will cast longer time than the access from the CPU to its own memory unit. The challenge is that the operating system should optimize the memory management for the NUMA structure. The issue of NUMA comes with the modification of computer hardware. When a new technique appears, it is always designed to solve a certain problem. The new generate of memory controller is one of this kind of products, and it do solve the problem of limited bandwidth in memory access. But this change also causes a new issue that is the non-uniform distribution of memory access. In the worst case, if all the requests are from other CPU’s memory block, they will cause higher latency. To be straight forward, the best way is limited the CPU only reads data from its own controller, but it is hard for programmer’s to write programs in this way.

Modern programming language is highly abstracted, so there is no concept of memory access management in program design. It is also hard to write programs only for a special use case and platform. So, this kind of work is the job of operating system and compilers. The best optimization way should avoid affecting the programmer’s work. The benefits of using NUMA are based on the design of hardware and software. Nowadays, modern computers all have good solutions for NUMA structure. The operating systems are also ready for the NUMA optimization. Though there are automatic NUMA allocate policies, the question of how to use NUMA memory in a proper way is still a good question for software developers.

**References**

Blagodurov, S., Zhuravlev, S., Fedorova, A., & Kamali, A. (2010, September). A case for NUMA-aware contention management on multicore systems. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques (pp. 557-558). ACM.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., ... & Sarkar, V. (2005). X10: an object-oriented approach to non-uniform cluster computing. Acm Sigplan Notices, 40(10), 519-538.

Manchanda, N., & Anand, K. (2010). Non-Uniform Memory Access (NUMA). New York University.

Bhuyan, L. N., Iyer, R., Wang, H. J., & Kumar, A. (2000). Impact of CC-NUMA memory management policies on the application performance of multistage switching networks. Parallel and Distributed Systems, IEEE Transactions on, 11(3), 230-246.

Acacio, M. E., González, J., García, J. M., & Duato, J. (2002, November). Owner prediction for accelerating cache-to-cache transfer misses in a cc-NUMA architecture. In Supercomputing, ACM/IEEE 2002 Conference (pp. 49-49). IEEE.

Bull, J. M., & Johnson, C. (2002, September). Data distribution, migration and replication on a cc-NUMA architecture. In Proceedings of the fourth European workshop on OpenMP.

Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., ... & Roth, M. (2013). Traffic management: a holistic approach to memory placement on NUMA systems. ACM SIGPLAN Notices, 48(4), 381-394.

Lachaize, R., Lepers, B., & Quéma, V. (2012, June). MemProf: A Memory Profiler for NUMA Multicore Systems. In USENIX Annual Technical Conference (pp. 53-64).

Li, Y., Pandis, I., Mueller, R., Raman, V., & Lohman, G. M. (2013). NUMA-aware algorithms: the case of data shuffling. In CIDR.

Lang, H., Leis, V., Albutiu, M. C., Neumann, T., & Kemper, A. (2015). Massively parallel NUMA-aware hash joins. In In Memory Data Management and Analysis (pp. 3-14). Springer International Publishing.