Under the Hood: NUMA on POWER7 in IBM i

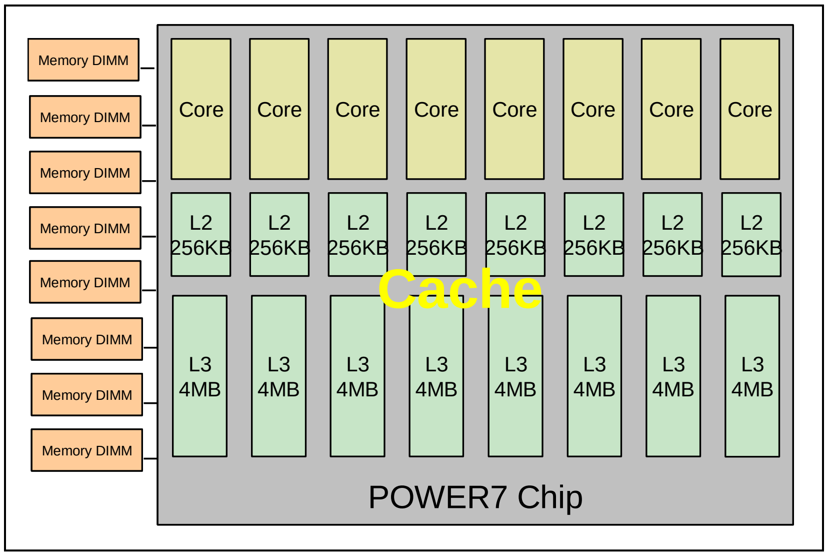
January 28, 2010

Disclaimer – NUMA on POWER7 in IBM i

NUMA on POWER7 in IBM i

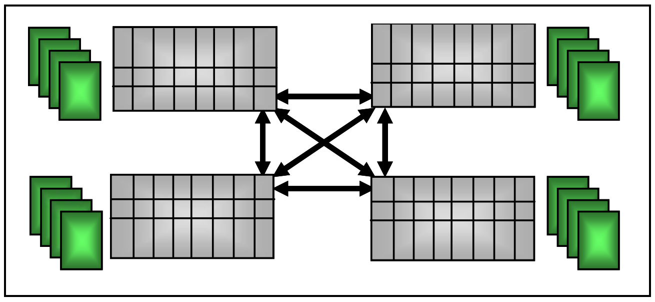
A lot has been written on multi-core computing, when all it really is is just a more dense packaging of more processor cores onto fewer chips. For example, IBM's PowerPC POWER7 chip supports eight cores per chip as compared to POWER6's two. That is a lot of compute power on a single chip. What this paper is all about is what happens when the compute capacity of such a single chip POWER7 is not enough, requiring a larger system to be built from multiple of such chips. Such a system takes on the characteristics of a computer architecture called NUMA (Non-Uniform Memory Access), a concept that you can use to further drive up the performance capacity of such a system.

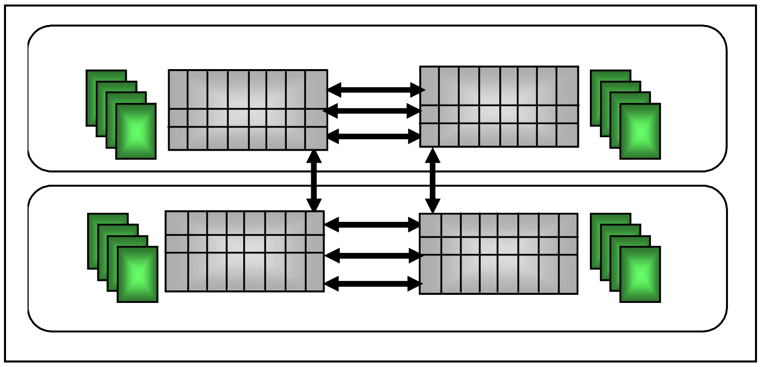
Let's start with some basics of what a multi-chip, multi-core POWER7 looks like. At its most simple, a POWER7 chip consists of eight processor cores, each with on-core L1 Instruction and Data caches, a rapid access L2 cache, and a larger longer-access L3 cache. The L3 caches are easiest to think of as core-private L3 caches, but the contents of all L2 and L3 caches are accessible from all processor cores. Further, data cast out of one core's L3 may be automatically cast into available space of another L3 on the chip. These characteristics make the L3 cache effectively a shared cache as well.



Each POWER7 chip also has memory controllers allowing direct access to a number of memory DIMMs, typically 8. Using, for example, 8 GByte DIMMs, each such POWER7 chip can directly access up to 64 GBytes of physical memory.

This chip and attached memory happens to be the basic building block of all of the POWER7-based systems from the POWER7 blades, through the POWER7 mid-range, and into the POWER7 high end. So let's look at two different ways used to create a 4-chip system, one seen in the multi-card low end and another in the multi-drawer mid-range systems.





Key starting points are that, even though such a system consists of multiple chips and multiple external cards or drawers,

1. All of the processor cores on any chip can access any of the memory in the entire system.
2. Every processor core on any chip can access the contents of any core's cache, no matter which chip.

Multiple chips or not, this is a cache-coherent SMP; **all memory is accessible, all cache is coherently maintained.** Functionally, it does not matter on what core a program's thread is executing or in what memory it finds its data and instructions; this is a cache-coherent SMP, a concept assumed by your applications and the operating systems.

But this paper is all about performance and maximizing the system's capacity. Looking at these two 4-chip systems, you will notice that from the point of view of some reference core - say the leftmost core of the upper left chip - it would take less time to access the memory locally attached to that reference core's chip than it would to access the memory attached to another chip. This difference is largely the basis for calling this a NUMA-like system; the memory access latencies are Non-Uniform.

Seeing this, an obvious question might be "Why would they design something like this?" The key to the answer is bandwidth. The amount of bandwidth available for accessing memory scales up linearly with the number of chips. For example, when a core makes an access from its local memory it uses the memory bus for a few cycles; the bus is effectively not available during this period for any other access. Increasing the rate of memory accesses, the bus can become very busy; the effect is further delayed accesses and increased average memory access latency. On the other hand, a memory access by another core on another chip also made to its own local memory is done absolutely concurrently with the first; the memory busses are completely separate. A more rapid access to local memory and scalable bus bandwidth is largely what a NUMA-based system produces. (For comparison, a system guaranteeing uniform memory access would be one wherein all processor chips access memory through a single common unit providing multiple ports for all of the systems memory. This common unit can also offer a lot of bandwidth, but it also adds latency to all memory accesses.)

So there is some advantage to NUMA, but it is Non-Uniform Memory Access. So what is the time difference on POWER7 between the local memory access time and the time to access the memory attached to another chip - something defined as a remote memory access? Best case - where a local memory access might take roughly 1/10 of a microsecond, a remote access from a directly attached chip's memory takes about 1/3 longer. Some of the chips in some systems are not directly connected and as a result memory accesses latencies between those indirectly connected take still longer. In a 4-chip system, if the OS/hypervisor were unaware of this effect, random distribution of data in memory would result in roughly ¾ of all accesses seeing this additional time; ¼ would see the faster local memory accesses.

The relative latencies may change and there are certainly a lot more cores and SMT hardware threads on a POWER7 chip, but this basic NUMA-like topology has been used by PowerPC processors since POWER4. So this is not really a new concept. But it happens that POWER7's support of so many hardware threads does rather matter. More on this as later.

The trick to improving system capacity in such a system is to do what increases the probability that the access is from local memory. IBM i and the hypervisor automatically provide some of this; we'll look at some of this later. Awareness of this effect and doing relatively small things can increase the probability of faster local access still further. Since such accesses are being done faster, work is completing sooner, increasing the capacity of the system to take on additional work and speeding the execution of individual tasks.

In general, NUMA management is largely the management of the relationship between the "node" where a task executes and the "nodal" location where that task will find its data. We call this "Memory Affinity".

Cache Affinity As Well

Contrary to what you may have learned in your college computer science courses, most modern processors do not directly access main storage. Most modern processor cores directly access only cache. The outrageously fast speeds at which instructions get executed on these processors occurs only when the data or instruction stream is held in each processor's cache. It is only when the cache access is unsuccessful that main storage is accessed, at which point the processor likely waits for what can be many hundreds if not thousands of processor cycles. A POWER7 core capable of executing up to five instructions per cycle could have executed a lot of instructions in the time it took to access main storage just once. Indeed, it is because of these long waits to access main storage that the cache exists in the first place. The multi-gigahertz frequencies of these processors would be pointless if each instruction and the data they accessed was accessed only from main storage.

Since cache is so important to the system's performance capacity and of individual tasks, the hardware design in this area is quite complex, but suffice it to say that the hardware does what it can to keep recently - and soon to be - accessed data in a cache somewhere. Often enough to support well, a task executing on a core and incurring a cache miss (i.e., the needed data is not then in that core's cache) will find the block of data it is looking for in some other core's cache. This "some other core" might be on the same chip or on another chip. (Of course, as we've implied earlier, such a cache miss might instead need to access local or remote memory.)

So, as with relative local vs. remote memory access latencies, how long are we talking about for various types of cache accesses? As mentioned earlier, each POWER7 core is directly associated with

1. L1 Data and Instruction Cache(s) .... 1 Processor cycle to access
2. L2 Cache ... Just a few cycles to access
3. L3 Cache ... A couple/few tens of cycles to access.

In the event of an access incurring a cache miss on these, the needed data might be found

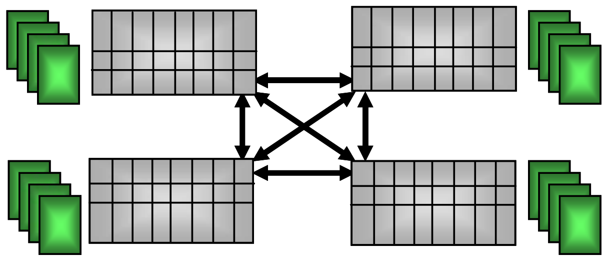
1. In the cache of a core on the same chip ... Over one hundred cycles and faster than memory.

2. In the cache of a core on a different chip ... A few hundred cycles and occasionally far more. Relatively speaking, this is a big difference. From a software point of view, what we are speaking of here is the relatively concurrent sharing of data between currently executing tasks. If the tasks doing such sharing happen to be executing on cores of the same chip, they can see the smaller latency of a local cache's access. If instead, the tasks happen to be executing on different chips, they instead see the considerably longer cross-chip latency.

As with all physical memory being accessible from all cores, the SMP rules require that the contents of all cache is accessible as well. So no matter the location, it all works. And even without knowledge of this NUMA effect, it works well. But as with the affinity to memory, there are simple techniques available to increase the probability that a task incurs the latency of a local cache access rather than having a cache miss be satisfied by the contents of another chip's cache. This, too, speeds the execution of individual tasks and so the total system capacity.

As a related concept, tt happens that individual tasks have a tendency to move, sometimes frequently, between processor cores. A task executing on one core, after temporarily losing the core (e.g., page fault or mutex lock contention), soon thereafter returns to execution on another. During the period executing on the first core, the task had incurred a number of cache misses, pulling much of the data state it needs into the core's cache. This data remains in the cache for a while even when the task is simply waiting. And, on POWER7, because of a notion called lateral cast-out, this same data might have been moved into the L3 cache of another core on the same chip to keep it in the data cached just slightly longer. When the task returns to a dispatchable state, being dispatched to any core meets its functional needs. But, ideally, when the task continues, the task would have been dispatched onto the core where its cache state might still reside. Failing there, because of the relatively rapid access of the cache state from a local cache, the next best core onto which to dispatch this task is a core on the same chip where it had last executed. The IBM i tends to automatically manage to this effect, but here too there are often simple techniques to more strongly influence this effect of "Cache Affinity".

A Busy Link Adds Time



Let's next go back and look at the links between the chips. As you've seen, any cross-chip cache fills - as well as cast-outs from cache to memory - use the link between the chips. As with accesses over the memory bus(es), if such are concurrently attempted across the same link, portions of the operation are serialized; when two concurrent operations are desired over the same link, one must wait for the use of the link. (Of course, concurrent operations over different links don't delay each other.) As more and more data is being passed between the chips, the probability for such delays increases. At heavy rates of data transfer, a given request might end up getting queued up behind quite a few preceding requests. The effect is that, as the link utilization increases, so does the average wait time for a given request. This shows up as still more latency - potentially many times more latency - for such inter-chip memory and cache accesses.

So what we are saying is that, over and above the nominal additional latency associated a remote storage or cache access, there is still more latency stemming from the high utilization of the inter-chip links, potentially becoming quite noticable with the high utilization of the cores. This, though, is also quite workload dependent.

Fortunately, as a higher percentage of the memory and cache accesses find themselves being satisfied locally, a lower percentage of the storage access requests find themselves needing to use a link at all. So increasing the probability of a local storage access, thereby decreasing the use of the links, also has the effect of speeding up those accesses which must be made over those links to remote storage. This is a nice side effect.

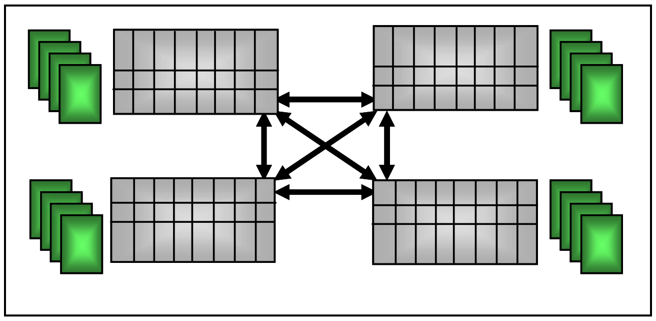
Actually Improving the System's Capacity

OK, you see the point; these multi-chip POWER7-based system function just fine as the cache-coherent SMPs they are designed to be, but by being aware of the NUMA-like characteristics of the actual hardware topology, a considerable performance benefit can be squeezed out of the system. So in the next few sections, we'll look at a few cases in point.

A common theme, though, are the notions of
- Placing the "work" onto cores closest to the location where the data can be found and
- Assigning storage from main storage closest to the cores where the work wants to be executing and
- Arranging for tasks sharing data to be on cores close to each other.

It's easy to say, but more tricky in practice. Let's next look at a few examples.

The Placement of Logical Partitions



The design of the POWER7 system's hypervisor is fully aware of the effects described so far. But, at this writing, there is intentionally nothing in the partition-specifying user interface where the NUMA characteristics show through. For example, when specifying the resources to be assigned to one or more partitions, you specify

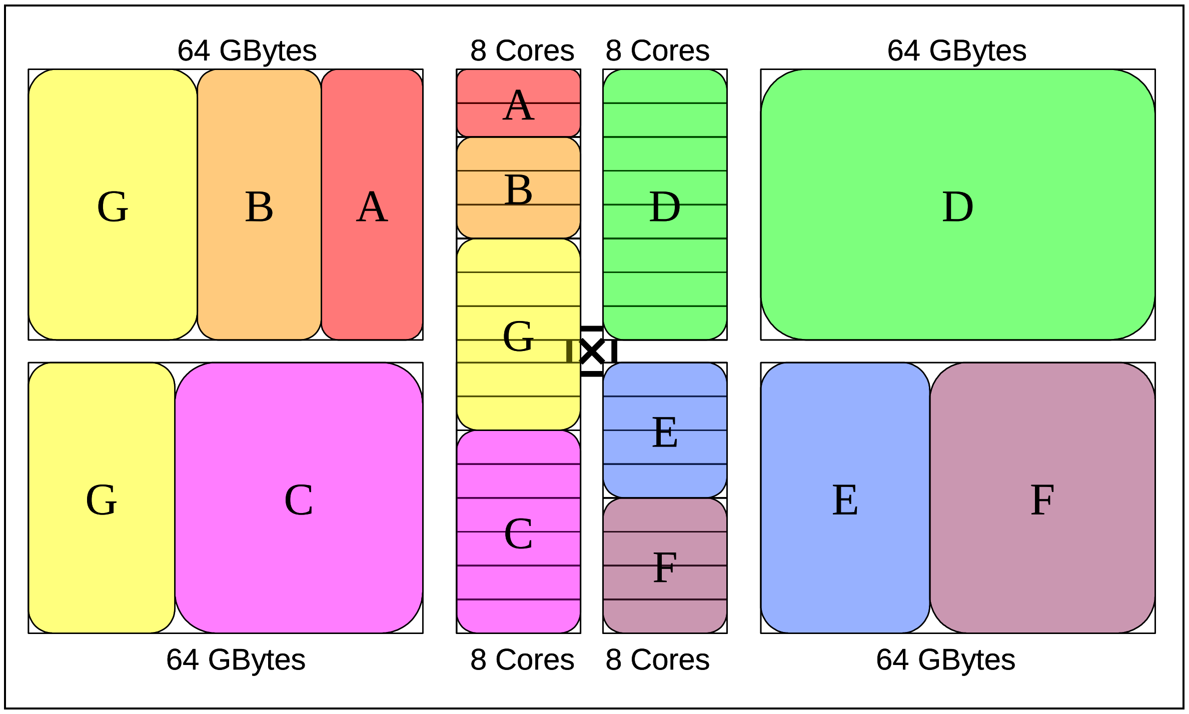
1. The capacity of each partition in terms of cores (this representing the fraction of the capacity nominally available in the system),
2. The amount of physical memory to be used by this partition,
3. Whether the partition is to be a shared-processor or dedicated-processor partition
4. For shared partitions, the number of virtual processors intended to use the partition's capacity.

There is intentionally no statement here about the location of these resources. And, again, the hypervisor is well aware of the advantages of managing partitions per the NUMA-based resources of the system.

So it becomes the hypervisor's responsibility to allocate these specified resources of all partitions in ways that do not require the partitions to be aware of the NUMA-based hardware characteristics. As a simple example using our 4-chip system
, let's define eight dedicated-processor partitions, each specifying the use of four cores worth of capacity and 1/8th of the total memory of the system. To acheive that, the hypervisor has a lot of options as to how these resources might get assigned. For example, each partition's memory could be striped across the memory of all chips. But as you've seen, each partition would see the maximum capacity if the partition's four cores were assigned from a single chip and if the partition's memory were completely allocated from behind that same chip.

This example seems obvious enough, but now consider the near infinite number of combinations that all partitions can be defined along with the ways that memory DIMMs can be purchased for a system. On top of that, after such a definition, additional partitions can be added and the resource sizes of partitions can be changed. And it remains the hypervisor's responsibility to find ways to approach an optimal placement solution for all of them. In the above example, the solution is straightforward. But given the breadth of the options for specifying partition resources, some non-optimal solutions will occur.

The solution comes with your awareness of the physical location of memory resources and cores associated with each chip. Since you can not specify exact location of these resources, the key is to arrange for the memory sizes requested for each partition to align as nicely as possible with the memory available behind the chips. You are describing the partition resources in such a way that the hypervisor can easily discover how to arrange the allocation of cores and memory per the NUMA-like topology of the system. Obviously, wherever possible, cores and memory should be allocated from single chips so that remote accesses are not done at all. But this cannot always be achieved. If partitions must span multiple chips, minimizing the number of chips is often the next best solution, but when doing so, balancing memory and core resource across chips is also desirable.



Consider the following figure with "nicely" packaged dedicated-processor partition resources. They <u>can</u> be nicely packaged <u>because</u> the hypervisor recognized how to efficiently package those resources. Even a subtle difference in the memory size of one partition might significantly alter this picture. It is because these partition resource sizes were defined with awareness of these boundaries - of the number of cores per chip and of the size of the memory DIMMs behind each chip - that the packaging fit as you see below. Even so, in this picture one partition did end up with resources crossing node boundaries, management of which now becomes the responsibility of the partition's OS.

A Shared-Processor Partitions Variation

The specification of shared-processor partition's processor resources is different than for dedicated-processor partitions. First, the total processor capacity is represented as being some unit subset of the number of cores in the system. In our 4-chip example, the shared-processor pool can be something less than or equal to the total number of cores (32 here) in the system. From within this total shared-pool capacity, each shared-processor partition is assigned some capacity in units of tenths of a core's capacity (so ranging from 0.1 through 32). This defines the capacity limits for the partition, but the OS of each partition is more aware of the specified number of virtual processors for that partition. As with dedicated-processor partitions dispatching tasks to cores, a shared-processor partition dispatches work to a virtual processor; the hypervisor then temporarily attaches a virtual processors to a core in the shared pool. Even though the partition's compute capacity might be some fraction of a core, the number of virtual processors that can be specified is some integer number. We'll get back to virtual processors shortly.

No different than dedicated-processor partitions, each shared-processor partition is also defined by the amount of memory that it requires. And just as with dedicated-processor partitions, the hypervisor has the responsibility of finding a physical location of that amount of memory[1]. Indeed, using the memory sizes specified in our multi-partition system shown earlier, and assuming that system was just one large shared pool, it is entirely possible for shared processor partitions to have their memory allocated in exactly the same way.

So given nodal memory assignment, when the hypervisor is asked to temporarily assign a shared-processor partition's virtual processor to a core, onto which core should it first attempt to assign it? For most of the partition's shown in the earlier figure, the memory for that partition resides on only one node. It follows that there exists a preferred node onto which that partition's virtual processor(s) ought to be assigned. Notice that it is a preferred node, not the required node. If all of the cores of that node were busily supporting other virtual processors, it is often quite acceptable - just not preferred - to attach such a virtual processor to another node's core if available. The point is, the hypervisor assigns each shared-partition's virtual processor a preferred node identifier, allowing dedicated-processor-like performance characteristics when the competition of virtual processors for available cores is not excessive. However, when not available, another chip's core gets chosen, storage accesses from there occur, and the virtual processor's task(s) execute there, albeit a more slowly.

The point of this section is that, even with all the flexibility of shared-processor partitions, with some knowledge of the chip and memory boundaries of the hardware, you can set things up to allow the system to execute faster.

IBM i Does Memory Affinity

With eight cores per chip, a lot of partitions have a pretty fair chance of seeing uniform memory accesses; the NUMA characteristics of the total system don't much matter to a partition whose cores reside on one chip and whose physical memory resides behind this same chip. But some partitions are less fortunate, usually because they are simply large partitions requiring the use of multiple chips[2]. It is here where the partition's operating system gets involved. We look at this next.

[1]This is true for shared-processor partitions which are also dedicated-memory partitions. For shared-processor partitions which are also shared-memory partitions, the location and even the amount of physical memory being used by the partitions in the shared-memory pool can be very fluid. For the most part, 10NUMA-based optimizations are not really applicable to shared-memory partitions.

[2]There are systems based on having fewer cores per chip (e.g., 4 or 6) and the "TurboCore" systems wherein an 8-core chip is being configured to used only four of the eight cores. The fewer cores per chip increase the opportunity for a partition to span multiple chips.

As mentioned earlier, from each partition's point of view the basics of NUMA management are to:
1. Allocate the memory used by a task from behind the node on whose cores the task prefers to execute. (i.e., allocate the data where the work is.)
2. Assign a task to the cores of a node where its whose memory contains the working set for that task. (i.e., assign the work where the data resides.)
3. Arrange for tasks most likely to be sharing the contents of cache to be executing on cores within the same node. (i.e., share a chip's cache efficiently.)

In doing so, the OS knows that it is important that all of the partition's core and memory resources are being used in a balanced fashion. It's not a good idea to have tasks waiting to use the core resources of one node while cores remain unused for a while on another. Similarly, if there is excessive pressure for memory on one node and far less pressure on another, storage ought to be allocated outside of the preferred node. The longer NUMA-like latencies we are talking about don't justify wasting such resources. But, assuming support for such balance, we proceed.

So, with balance in mind, IBM i assigns each newly created task a Home Node ID, essentially suggesting for it its preferred POWER7 chip. Each time that a task enters a dispatchable state, the IBM i's task dispatcher first attempts to dispatch the task on a core of this preferred node. With POWER7's SMT4 (4-way Simultaneous Multi-Threading) and so 32 dispatchable locations on each 8-core chip, there is a good probability that it will succeed.

The other half of memory affinity is the need for a task executing on a core to find that its cache misses are satisfied from memory directly attached to the chip. Here, too, this Home Node identifier is used. For example, when this task page faults - the page needed is not yet in physical memory - an available physical page must be found for the needed page. IBM i could find an available page from all of the partition's memory, but instead the first place it looks is in the physical memory of the task's Home Node. When next the task is dispatched to this Home Node and it accesses this page, it will successfully access local memory.

It's rather straightforward, but as we'll see it's not always quite that simple. Let's consider a few problem cases:

- For all the capacity of a single 8-core POWER7 chip, it is entirely possible that all of the SMT slots of all the Home Node's cores are busy when a task becomes dispatchable. Obviously, if all other nodes are similarly busy, this task can just wait its turn. But if a task could be immediately dispatched onto another node's core, should it take advantage of that core then - and execute slightly slower - or wait for a slot on its Home Node to free up? That's a function of how quickly a Home Node's core frees up and/or how soon the available remote core would have been used.
- A partition might need more memory than is available from behind one chip, but the compute capacity needed by that same partition might be satisfied from the cores of a single chip. The hypervisor might have had the option of also splitting those cores across multiple chips to be in balance with the memory, but let' say it didn't. In this case, all of the tasks of the system share

the same Home Node, and they would still prefer to have their memory allocated only for the chip’s local memory, but obviously all of the partition’s memory should be used when needed.

- Dynamic LPAR provides the opportunity for altering 1) the amount of memory available to a partition, 2) the number of cores used by a dedicated-processor partition, and 3) the compute capacity and number of virtual processors seen by shared-processor partitions. It follows that the nodal location of these resources also change. Indeed, what might have been a nicely packaged partition might become otherwise. It is the partition’s responsibility to recognize these physical changes and to rebalance the preferred locations of the current tasks.

Realizing the dynamic nature of the hardware resources that a partition might own and realizing that the system executes work acceptably - albeit nonoptimally - even if nothing is done about NUMA, IBM i provides no means of explicitly specifying a particular physical node within which a task must execute and allocate memory. So the types of controls that IBM i provides to better control affinity are advisory in nature. Basically, you can control

1. How strongly the partition ought to attempt to place a task onto its Home Node and
2. The grouping of tasks under a common Home Node, whether those tasks belong to the same Process or whether they are tasks associated with multiple Jobs.

The strength of the affinity of a task for its Home Node (either *NORMAL or *HIGH) can be specified over the whole of the partition via the system value QTHDRSCAFN. Affinity strength can also be specified on job-by-job basis via routing entries.

The concept of grouping threads to share the same node is the same whether those threads are from a multithreaded job or multiple jobs, but the means of achieving it is somewhat different. In the case of multithreaded jobs, it is possible to indicate either on a job basis (saying*GROUP or *NOGROUP on THDRSCAFN within a routing entry or prestart job entry) or system-wide (by using the QTHDRSCAFN system value) to control whether the multiple threads of a job should share the same Home Node, or whether each should be assigned an arbitrary Home Node. In the case of the common Home Node, the first thread of a job is assigned a Home Node and all subsequent threads share the same Home Node.

To enable groups of jobs to share a Home Node, create a resource affinity group via a routing entry or prestarted job entry by specifying RSCAFNGRP = *YES. Then all threads of all jobs in the group are assigned the same Home Node. It is when the first thread of all of these jobs is created that the system assigns the actual Home Node value.

The system value and job attributes described above allow a level of control without direct knowledge of the number of nodes in the system. Where you want even tighter control, IBM i does offer limited API programming support to control what physical node a thread can be dispatched to. This requires the knowledge of Machine Interface (MI) programming. Basically, the Materialize Process Attribute (MATPRATR) MI instruction can be used to retrieve the Home Node ID that a particular thread is executing on. Then the SPAWN programming API has been enhanced such that you can specify the Node ID that a thread should be dispatched to.

Realizing that jobs come and go over time, it is possible that the settings of each thread's Home Node might result in a misbalanced use of processor and main storage resources. For this reason, the system watches for balance and reassigns the Home Node of some threads to return resource utilization to balance. Jobs and threads which have been logically grouped together by specifying *GROUP on the QTHDRSCAFN system value, or via routing entries, will be kept together as a group if they are moved. The determination of a misbalanced state and the actual movement is not done rashly, instead some time passes before a system is deemed to be misbalanced. Even so, a system value is provided (QTHDRSCADJ) which specifies whether or not the system should make adjustments to the affinity of threads currently running in the system. By default the QTHDRSCADJ system value is set to automatic adjustment. Although this action is the default behavior, if you wanted to "freeze" the location of all threads (i.e. disable this tuner), you set the QTHDRSCADJ system value to not dynamically adjust the threads.

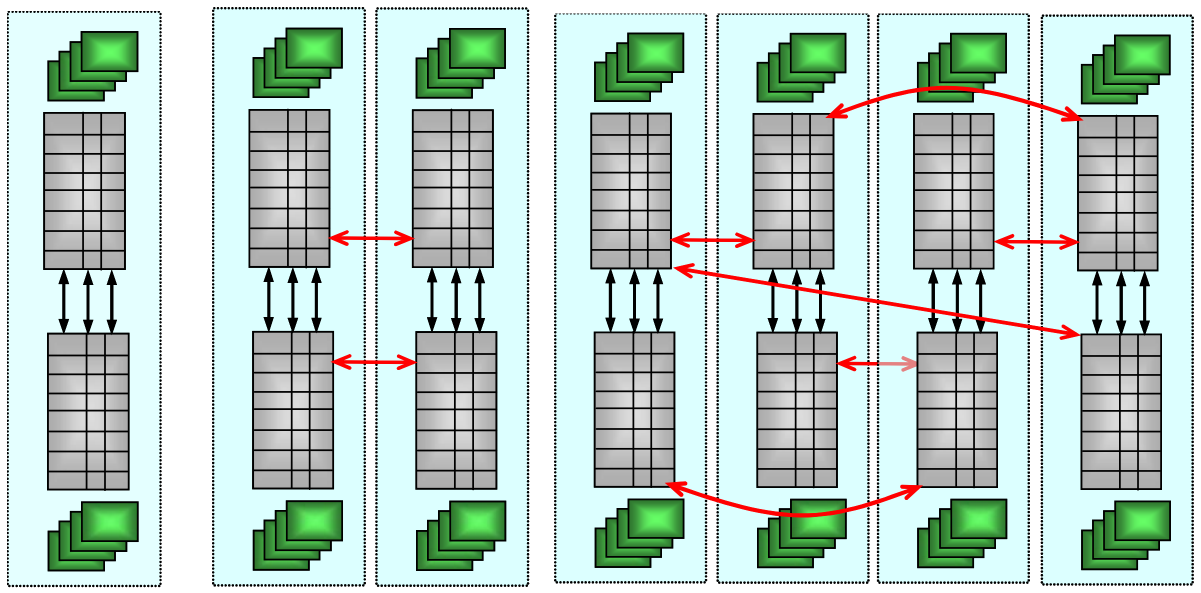
"TurboCore"

You have seen how each POWER7 chip has up to 8 physical cores. But suppose that you only need four of the cores on a chip. This is the starting point for the notion of "TurboCore", a technology to produce better performance when only these fewer cores are needed. We'll explain further by putting together the pieces.

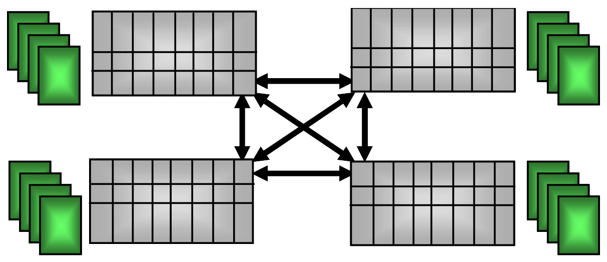
- Each of the cores on a chip has its own 4 Mbyte L3 cache. The hardware's cache management on each chip allows for some of the contents lost ("cast-out") from a core's L3 to be temporarily stored into the L3 cache of another core; this capability becomes especially useful when one or more of the cores happen to be relatively idle. So, in the case of TurboCore, when you have such a fully functional 8-core chip using only four cores, that also means that 4 of the 8 cores - and their L3 caches - are being used to execute instructions and the remaining four cores are effectively idle - "Napping" is the power-management term used. In such a state, the cache of these remaining cores remains functional. The cache topology of such a TurboCore chip then appears as a set of four cores each with their own private 4M L3 plus an extra 16 Mbytes of shared L3 cache per chip. More cache often translates into better performance.
- Each chip has a nominal frequency which is set partly because that frequency also determines the amount of heat that it generates; this is heat that must be removed from the system. Indeed, assuming a linear increase in a core's frequency, the rate of heat generation increases faster than this. And, again, we have eight cores per chip all potentially executing instructions at some maximum rate, each running at the chip's nominal frequency, producing heat which must be removed. But now suppose that your system had only four of the eight cores executing instructions. It follows that the chip can be more easily cooled. It also follows that if the chip's design were capable of a super-nominal frequency, that the chip could still be cooled if run at this super-nominal frequency. TurboCore's frequency improvement on POWER7 is roughly 7%.

So, eight physical cores and cache per chip are available, but if only four were needed, these four could be allowed to run at a slightly higher frequency and to perceive some additional cache. For these four cores, the higher frequency and cache often translate into better performance over what they are capable of without it. This is not to say that the performance capacity of these faster four cores is the same as what is available from an 8-core chip. But, instead, because you are only using four per chip, the super-nominal frequency and additional cache is available to provide better performance than what is available with a 4-core chip alone.

Good news, right? So why is this part of a paper on POWER7 and NUMA? At its simplest, fewer cores per chip also typically means more chips. Further, the TurboCore capability is available on particular models of the POWER7 systems whose chip topology is as is shown in the following figures. With physically eight cores per chip, these systems have - from left to right - 16, 32, and 64 cores per system.



Comparing this middle (2-drawer, 32-way) system to the 32-way system shown previously (and reproduced at right), the multiple links between pairs of chips - physically, the two chips of a drawer - have more bandwidth between pairs of chips than the system at right. Nominal latencies for cross chip accesses are roughly the same, but the extra bandwidth available between the chips of each drawer helps these latencies stay near nominal as the cross-chip traffic between these two chips increases. But for cross-drawer traffic, the advantage is considerably less; indeed, unlike the system at right, not all chips are directly connected, this adds extra latency as a result. As previously discussed, the operating systems know of this difference in topology and have ways of dealing with it.



This is nothing new from a NUMA point of view, so let's now factor in TurboCore. In TurboCore there are 4 cores per chip available (along with higher frequency and more cache per core), not the usual eight. This difference is key; at its simplest, for the same number of cores, the TurboCore option will require twice as many chips and drawers. More chips can mean more cross-chip and cross-drawer traffic and so longer storage access latencies, but not always. And as we've seen, increased cross-chip traffic can mean decreased system capacity, potentially decreasing the benefit of TurboCore that the extra chips were intending to provide. How much these NUMA characteristics offset TurboCore's otherwise obvious benefits depends on how the entire system is to be used, so let's look at a few cases in point.

We create here a few examples by mapping one or more partitions of various sizes onto the drawer-based hardware capable of supporting TurboCore. All cases will use the same number of cores (16), comparing the 8-core chips with the 4-core TurboCore option within one or more dedicated-processor partitions.

Four 4-Core Partitions ...

- When using the full 8-core chips, these four partitions are packaged two to a chip and so within the two chips of one drawer. Here the frequency is nominal and when all cores are executing instructions, there is no additional cache[3]. If each chip's memory is sufficient for two partitions, the two active partitions of that chip are competing for the use of the common memory busses of their chip.
- When using TurboCore, the needed sixteen cores require four chips and two drawers. Each partition's four cores are packaged from within one chip. Using only four cores per chip, each partition is also using TurboCore's extra cache and the super-nominal frequency. Let's also assume that each partition's memory can be assigned from behind that partition's processor chip.[4]

Notice in particular that for both cases practically all storage accesses are confined to one chip. From the point of view of each partition, TurboCore or not, this is not a NUMA system. As a result, the partition's storage access latencies are only those of local accesses. As a result, each partition also sees practically all of the benefit of TurboCore's higher frequency and larger cache. This would seem to be an ideal use of TurboCore since the extra chips (and drawer) did not also add additional storage access latency.

Two 8-core Dedicated-Processor Partitions ...

- When using the full 8-core chips, the system seen by the two partitions is just one drawer containing the two processor chips. Each partition has its own chip of eight cores. Assuming that the partition's memory can be allocated exclusively from behind this one chip,

[3]There is additional cache for those periods when one or more of these cores is relatively idle. Indeed, while only one core is executing instructions, that core would perceive its own 4M L3 plus 28M of shared L3 cache.

[4]The number of DIMM slots per processor chip is unchanged at eight. So with extra processor chips also comes more DIMM slots and so the opportunity for more local memory.

the chip's main storage busses are also used exclusively by the partition owning that chip's cores. As you've seen, one chip means no NUMA characteristics. Further, when there is frequent sharing of cached data, the data transfer time between each chip's cores is relatively rapid. From a NUMA point of view, this is a near ideal environment for a partition. Still, for those periods where all of the partition's cores are actively executing, there is not really any additional cache perceivable by a partition. And with eight active cores, the frequency is nominal.

- With TurboCore, the two 8-core partitions require four 4-core processor chips and so two drawers. Each 8-core partition can reside within the pair of chips within a single drawer, ideally also accessing the memory found only in its drawer. Being 4-core chips, each is running at the super-nominal frequency and providing its partition and extra 16 Mbytes of shared cache. Here, spanning two chips, both partitions are aware that it is part of a NUMA system. Where the cross-chip storage accesses can be minimized, we also see minimum performance impact from the hardware's NUMA-like characteristics. For this reason, many numeric intensive or essentially multiple single-threaded workloads within a partition would consider TurboCore beneficial. As the rate of cross-chip storage accesses increase, as the probability that a storage access of the other chip's memory and cache increases, the system capacity also decreases. Recall also, though, that the two chips of a drawer - the two chips used by each partition - are also capable of supporting some quite considerable bandwidth for inter-chip traffic. Recall here that we are comparing the capacity of an 8-core partition residing on one chip with the capacity of this same 8-core partition on two TurboCore chips, with both doing the same work. Where the applications managed by the TurboCore's two-chip partition can minimize the cross-chip traffic, all of TurboCore's performance advantages remain. Applications requiring even some moderate level of such inter-chip traffic may find some considerable performance benefit from TurboCore.

One 16-core Dedicated-Processor Partition ....

- When using 8-core chips, this one 16-core partition resides within the two chips of a single drawer. Frequency is typically at nominal and, when executing on all cores, there is no extra L3 cache available. It is clearly a NUMA-like system from the point of view of the partition. As such, the two chips introduce some level of added storage latencies impacting system capacity. Fortunately, for these two chips of a single drawer, there is quite considerable inter-chip bandwidth to limit inter-chip latency increase as system utilization increases. Even with no partition NUMA management, the random probability of a remote memory access is only 50%.
- With TurboCore, the one partition is within two drawers and so four chips. The partition's tasks are each assigned to one of now four nodes. The partition's memory resides behind four processor chips. With four nodes, the operating system's responsibility to manage the NUMA characteristics become more difficult. As compared to the 2-chip partition, the rate of inter-chip traffic will be higher, but perhaps not excessively so. But this multi-drawer configuration also introduces the skinnier pipe and multi-hop latencies found in multi-drawer traffic. But if the application set allowed for easy NUMA management, inter-chip traffic can nonetheless remain minimal. And minimal added storage latencies also means having the full performance advantages of TurboCore.

Once again we have a tradeoff between the capacity benefits of a higher frequency and more cache versus the capacity offsetting effects of, here, multiple forms of cross-chip traffic. Where the OS can minimize this cross-chip and/or cross-drawer traffic for some set of workloads, much of the benefit of TurboCore still remains. But as the rate of cross-chip traffic increases, as the probability of longer storage access latencies increases, the benefit of TurboCore becomes less clear.

We can continue on with many additional scenarios but the story ought to be fairly clear by now. If dealing with multiple partitions packaged in such a way that the partitions each wouldn't even be aware that the entire system is NUMA-like, TurboCore's higher frequency and larger cache per core would provide each partition additional capacity and better single-threaded performance. Even when dealing with larger partitions where the partition is aware that it is managing the NUMA-like behavior, where the cross-chip and cross-drawer traffic is not excessive, even then much of TurboCore's benefits can be seen. But there are also some cases where TurboCore's need to spread partitions out across multiple chips and drawers would result in offsetting effects from the added storage latencies; TurboCore simply implies more chips and possibly drawers.

The bottom line, TurboCore's potential need for more chips also carries the potential for a higher average storage access latency due to cross-chip storage accesses. TurboCore's higher frequency and larger cache provide for better performance. As you've seen, cross-chip storage accesses decrease system capacity. So we have a trade-off. And, ultimately, the use planned for a TurboCore-based system needs to have allowed for enough benefit to justify the extra cost in hardware and energy.

As a positive case in point, the workload SPECjbb was run within a single 32-core partition, both in a normal 2-drawer (4 chips, 8 cores per chip) configuration and a TurboCore 4-drawer (8 chips, 4 cores per chip, 16M extra L3). This benchmark is characterized by minimal sharing between threads and is capable of very nodal behavior. Rather an ideal situation. For this reason, SPECjbb benefitted considerably, well in excess of the simple boost in frequency, to the tune of 18%.