ccNUMA overview

Table of contents

Introduction 3

Executive summary 3

Problem statement 3

Introduction to ccNUMA 3
- Physical memory characteristics 3
- Cache coherency 4
- Interleaved memory 4
- Local memory 4

HP high-end SMP servers and the cellular architecture 4
- Past implementations 5
- Locality domains 5
- Comparison of existing HP 9000 high-end SMP servers to the new HP Integrity SMP servers 6

Competitive situation 7
- SGI IRIX 7
- SGI Altix 64-bit Linux systems 7
- Tru64 7
- Linux 8
- IBM NUMA-Q 8
- IBM pSeries 8
- Sun 8

Impact on applications 9

Memory topology 9

Process management 9
- Launch policies 9
- Default launch policies 11
- Discovering the execution domain topology 12
- Utilizing the execution domain 12
- Discovering a process's locality bindings 12
- Inheritance of launch policies 13

Memory allocation 15
malloc() 15
mmap() and shmget() 15
Fortran/common/ 15
Performance-optimized page sizing (POPS) 15
Memory exhaustion and paging 16
Page migration 16

Summary 16

References 16
HP-UX 11i release names and release identifiers 17

Introduction

ccNUMA (cache coherent Non-Uniform Memory Architecture) systems offer programmers and users the simplicity and flexibility of symmetric multiprocessing (SMP) with the memory scalability of clusters. In a ccNUMA system, processors, memory, and I/O are grouped together into cells. The latency and bandwidth characteristics of communication within a cell are "fast," while going outside a cell is "slow." Since the memory in ccNUMA systems is physically distributed but logically shared, these systems offer better performance to applications that are optimized to use their features. For non-optimized applications, they still offer better performance since the default behavior is designed to be benign—if not beneficial—and they still have access to much larger shared resources of memory, CPUs, and disk space.

HP-UX has made use of ccNUMA technology in its newest, most scalable machines. The HP Integrity systems will have ccNUMA support in HP-UX 11i v2, while the HP 9000 systems will have ccNUMA support in HP-UX 11i v3.

Executive summary

The newest, largest HP systems make use of a design called ccNUMA. In this design, processors, memory, and I/O are grouped together into cells. The latency and bandwidth characteristics of memory within a cell are "fast," while accesses outside a cell are "slow." HP-UX makes use of this design by attempting to keep the majority of accesses local to the same cell. This is achieved through good default behavior and by providing interfaces for ccNUMA-aware applications to make the best use of the architecture.

Problem statement

SMP systems cannot offer sufficient memory bandwidth for large numbers of processors without incurring excessive penalties due to memory latency. ccNUMA architectures offer scalable memory bandwidth while keeping memory latencies reasonable—often delivering latencies in the same class as much smaller systems. It is intended that many applications can run unmodified and perform well, but in order to reach maximum performance, important applications may need to be modified to exploit the new features of HP-UX 11i v2.

Introduction to ccNUMA

Physical memory characteristics

Physical memory refers to the actual physical arrangement and connection of the memory to the rest of the computer system. Since applications running on HP-UX deal with the system in terms of *virtual memory*, knowing about and optimizing applications with respect to physical memory is not a common concept in application design.

In cell-based systems like high-end HP servers, memory is arranged in cells, often symmetrically throughout the system. But it is not required that memory be symmetric, and due to budget constraints and failures, memory may not be evenly distributed over all the cells. These systems can support any amount of memory up to 32 GB on a cell.

Capacity is the quantity of memory that is available to an application.

Latency is the time it takes for a memory reference from a processor to be satisfied by the memory system. Often measured in nanoseconds, this is typically the "load to use" latency. This quantity includes the time the processor takes to put the request on the memory bus, and the time it takes to transfer the request to the memory controller and deal with all the cache hierarchy, satisfy the request, and return the data to the register on the processor requesting the data.

Bandwidth is the rate at which data is transferred to the processor or to the I/O subsystem.

Occupancy is the amount of time a memory image exists in memory. This is an issue because a lot of interesting applications have very high degrees of occupancy, meaning they exist for long periods of time. This gives the system opportunities to modify and optimize the performance of an application.

Cache coherency

All modern multiprocessor systems are cache coherent. This is necessary since processor caches have copies of data that is in memory. When one processor modifies data in memory location, it does so in its cache, on a cache line basis. The process of modifying the data must also include the step of notifying any other processors that may have that cache-line that the data in their cache is now invalid. The next time the cache line is referenced, this prompts the memory system to force the owning processor (the one that modified the data) to write the data back to memory and then—in a separate step or simultaneously—cause it to be reloaded into the consuming processor's cache. This is not a simple problem, and solving this problem efficiently is crucial to the successful implementation of the overall memory system. HP high-end SMP servers implement this mechanism with a directory-based coherency mechanism. This scheme has proven to be very efficient and scalable.

Directory-based coherency systems differ from other, simpler means, such as "snoopy" schemes, in that the memory system state is catalogued in the memory system itself. Snoopy methods "listen" to the memory bus and watch for memory addresses that the processor may have encached. The directory is in the memory system, and it tells any requesting processor whether the cache line is clean and unmodified, or modified and dirty, and which cache owns the cache line or is encached in some other processor's cache but not modified (yet). This method is much more scalable and simpler to implement on a large-scale system than traditional snoopy methods.

Interleaved memory

Interleaved memory is memory for shared objects or data structures. A portion of memory is taken from cells of the system—typically all of the cells—and is mixed together in a round robin fashion of cache-line-size chunks. It has the characteristic that memory accesses take a uniform amount of time. In other words, it has uniform latency no matter which processor accesses it.

Local memory

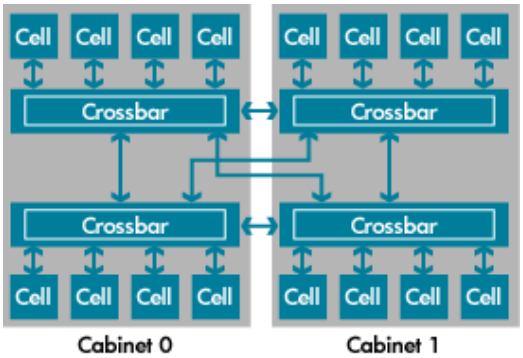
Local memory is memory for private objects or data structures. It is still accessible to any processor, but processors that are on the same cell will enjoy the lowest latency for memory accesses. Accesses from other cells will take longer, or have greater latency, than accesses from within the same cell.

It should be pointed out that local memory is interleaved only over the memory banks of the local cell.

HP high-end SMP servers and the cellular architecture

HP high-end SMP servers are organized in *cells* of processors, memory, and I/O connections. These cells are connected by crossbars. Four cells connect to a crossbar, and the crossbar connects to other crossbars to form the interconnect. Memory latency is smallest when referencing memory locally. In other words, it is smallest from memory on the cell that the processor is located on. The next nearest memory locality is the memory in the other three cells connected to the same crossbar. Next is the memory located in the cells connected to the other three crossbars. Memory transactions should never cross over more than two crossbars. But memory traffic between system cabinets does have a greater latency than traffic within the cabinet. This is illustrated in Figure 1.

Figure 1. HP high-end SMP servers—logical diagram



Past implementations

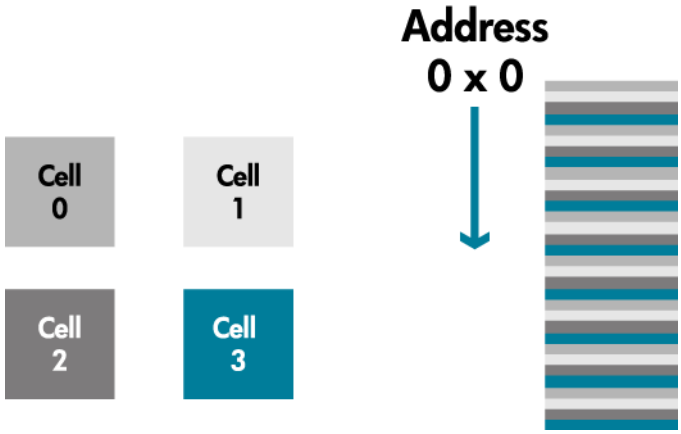
With HP-UX 11i v1, all memory is interleaved over all the cells. The interleaving is done on a cache-line basis. This makes the memory latency appear uniform. It should be noted that latency is greater for large systems than for small systems since there are more memory references across more crossbars, on average.

For example, for a 16 CPU, 4-cell system, a memory reference is satisfied ¼ of the time locally and ¾ of the time remotely. The remote accesses all are satisfied with one hop across the crossbar (see Figure 1). So the average latency is $(T_{local} + 3T_{remote})/4$.

For a 32 CPU, 8-cell system, 1/8 of the references will be local, and 7/8 will be remote. But the remote accesses will have different latencies. Now, 3/8 of the references will be across 1 crossbar, and 4/8 will be across 2 crossbars. This yields an average latency of $(T_{local} + 3T_{remote} + 4T_{veryremote})/8$.

A key feature of the original HP high-end SMP implementation is that memory bandwidth from the local cell is essentially the same as the bandwidth available through the interconnect. Combined with the fact that data is interleaved over all the cells, this makes the system perform essentially as a Uniform Memory Access (UMA) system. This makes performance of applications uniform and repeatable. From the application's perspective, there is no need to distinguish between local and remote memory for the purposes of memory accesses.

Figure 2. Interleave mapping of cache lines over cells



Locality domains

A useful concept is the locality domain (LDOM). A locality domain consists of a related collection of processors, memory, and peripheral resources that compose a fundamental building block of the system. All processors and peripheral devices in a given locality domain have equal latency to the memory contained within that locality domain.

A cell is a locality domain. The interleave memory region is a locality, but not a locality domain since it contains no processors or peripherals.

Comparison of existing HP 9000 high-end SMP servers to the new HP Integrity SMP servers

The primary physical difference between the old and current high-end SMP servers from HP is the improvement of the cell memory bandwidth. The primary logical difference between the two systems is the segregation of memory into local memory and interleave memory, as shown in Figure 3.

Figure 3. HP Integrity server memory logical diagram

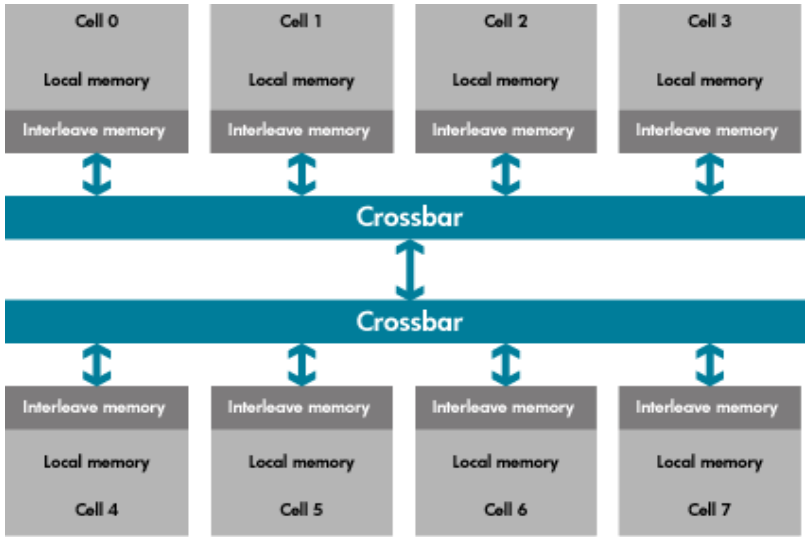


Table 1. Comparison of generations of HP Superdome servers

	HP 9000 Superdome	HP Integrity Server
Bandwidth (sustained)		
CPU buses/cell	5.2 GB/s	10.8 GB/s
I/O/cell (avg. mix)	1.0 GB/s	1.2 GB/s
Memory/cell	3.2 GB/s	7.2 GB/s
Crossbar/cell	5.2 GB/s	6.4 GB/s
Latency (load-to-use)		
Local memory	212 ns	243 ns
1-hop memory	302 ns	423 ns
2-hop memory	366 ns	479 ns
Cache-to-cache	4 hop	4 hop
Cache-line size	64 bytes	128 bytes

The HP 9000 Superdome memory system bandwidth is balanced between local and remote performance in a proportion that makes treating the memory system as physically uniform a useful simplification. The HP Integrity server increases memory bandwidth in all respects compared to the HP 9000 Superdome. Memory bandwidth to the processors is doubled, and bandwidth from the local memory system is more than doubled. Crossbar bandwidth increases by 25%, and I/O bandwidth increases by 20%. Since the local memory bandwidth increases much more in proportion to the crossbar bandwidth, the HP Integrity server exhibits stronger ccNUMA behavior.

The HP 9000 Superdome running HP-UX 11i v1.1 acts as UMA with an effective latency closer to 2-hop performance. For example, an 8-cell configuration has an effective latency of 323 ns. The HP Integrity server delivers 243 ns latency and 7.2 GB/s memory bandwidth if the dataset can be contained in local memory. This is a 33% improvement in memory latency, if the applications can take advantage of the performance features of the memory system. By default, HP-UX 11i v2 allocates memory for applications from the local memory, or LDOM. For most applications, specifically applications that do not use multiple processors, this ensures optimal performance of the applications and the memory system.

For parallel applications or multiprocess applications, the situation is somewhat different, but correct behavior is still assured. Existing HP-UX 11i v1.6 applications will work correctly. To run optimally, some applications may need to be slightly modified, but most will not. For many applications, the only changes are in the launcher scripts.

This paper documents the tools and resources available for developers to take maximum advantage of the features of HP-UX 11i v2.

Competitive situation

Most of the commercially available multiprocessors are implementing some variation of a ccNUMA architecture. However, they have all implemented different interfaces to exploit the ccNUMA features of the systems. Unfortunately, there is no standard means available yet to program these systems.

SGI IRIX

SGI IRIX was the first commercially successful ccNUMA operating system. It was introduced in 1997 with the Origin2000 system, marking the first instance of a widely used system of this class.

The Origin is organized in nodes, with two processors connected to a hub, which is also connected to the node's memory. The node then shares with one other node, an I/O hub, which is connected to peripherals. The node itself is connected to the interconnect network.

SGI has provided two main tools for developers and users: dlook and dplace. The dlook diagnostic tool helps developers and administrators understand what the application is physically doing on the system in terms of memory usage, contention, and bandwidth.

The dplace tool is encountered most often, since it is used in scripts and queuing systems to actually launch the application in an efficient manner, considering current activity on the system and what resources the user wants to make use of.

SGI Altix 64-bit Linux systems

The SGI Altix is organized the same way as the Origin system. The main difference is the incorporation of the Intel® Itanium® 2 processor and the use of Linux as the operating system.

SGI has extended Linux to work efficiently on a ccNUMA multiprocessor. They have also ported and provided dlook and dplace to aid developers and users in making the most efficient use of the system.

Tru64

On Tru64 UNIX® systems, the building blocks that make up a NUMA system are mapped to structures called Resource Affinity Domains (RADs). A RAD identifies the set of CPUs, memory arrays, and I/O buses that, when used together, allow the system to work most efficiently.

Starting with Tru64 UNIX version 5.1, the operating system makes a best effort to:

- Schedule all threads of a multithreaded application on CPUs in the same RAD
- Allocate memory for each process or application thread in the same RAD as the CPU where the process or thread is running

The default NUMA-aware algorithms for scheduling and allocating resources to a process or thread work well when the resources in one RAD can accommodate the number of threads and the memory demands in any one application.

The NUMA application programming interfaces (APIs) allow applications to make scheduling and resource allocation decisions based on advance knowledge of the application's resource needs and behavior. Proper manipulation of system resources and process scheduling through NUMA APIs has the following potential advantages:

- An application can notify the operating system of relationships between processes and threads that should be scheduled on the same RAD and, if migration to another RAD becomes advantageous, must be moved together.
- A very large and complex application whose resource demands and number of threads exceed the capacity of one RAD can stripe its CPU cycles, I/O load, and the memory that contains program data across RADs.

Linux

Linux does not have any inherent ccNUMA support.

Linux has limited support for SMP architectures. It is optimized for two to four CPUs, with support for up to eight CPUs. The implementation does not preclude Non-Uniform Memory Access, and cache coherency is assumed, so ccNUMA is available—it is just not very efficient. Furthermore, there is limited support for memory locks and semaphores, and little attention is paid by the kernel developers to efficiency of the multiprocessor versions of Linux. (For more information, see http://sources.redhat.com/ecos/docs-latest/ref/hal-smp-support.html.)

IBM NUMA-Q

IBM NUMA-Q implements a ccNUMA architecture using Intel Pentium® processors. The system is organized into four-CPU quads that include memory and disk controllers. The quads are linked together using a relatively low-bandwidth interconnect called IQ-Link. All memory is shared among all processors, and I/O devices are also shared. A primary goal of this architecture is to ensure continuous access to attached I/O devices, and it accomplishes this through redundant links to external devices.

The operating system, DYNIX/ptx, maximizes system performance by locating the memory and I/O connections close to the calling process, preferably on the same quad. This operating system is based on System VR4 and is extended to handle large numbers of processors and users.

Windows NT® and Linux also are available, but they are limited to being deployed in a quad and therefore do not need to support any ccNUMA features.

IBM pSeries

AIX 5L has some features to accommodate the different latencies of the Regatta architecture. The basic physical arrangement of the POWER4 series processors is four dual-core POWER4 processors, four 32 MB L3 caches shared by all eight CPUs, and a multi-chip module that ties the processors and memory together, along with I/O and a connection to the rest of the system. A p690 system consists of four of these modules. AIX 5L provides some specialized scheduling algorithms to keep applications near their data in order to reduce memory traffic and latency.

AIX 5L also implements a Large Page feature. By default, pages are 4 KB. The administrator can designate a certain amount of the total memory to be made of Large Pages, defined as 16 MB. Upon a reboot, these pages are available to applications, if the user and the application have been authorized by the administrator to take advantage of this feature.

Sun

Sun high-end servers, such as the Enterprise15000, are ccNUMA systems.

Sun provides a tool to optimize application performance, called Memory Placement Optimization (MPO). It attempts to place processes as close as possible to the memory they are using in order to reduce latency and memory contention.

Impact on applications

Many applications will not be adversely affected by ccNUMA, but some key types will. Applications that need to be aware of ccNUMA include:

- Pthreads applications
- SystemV shared memory applications
- Message passing applications. MPI is ccNUMA-aware with version 1.8.4; some applications will need to be relinked to take advantage of optimizations in MPI for cell local memory
- Applications that use anonymous mmap() to allocate memory

Applications generally not adversely affected include:

- Single-threaded applications
- Parallel applications run in single CPU mode—both shared memory and distributed or message passing versions
- OpenMP applications in general do not need to be changed, but they may need to be relinked if they are built with archive versions of libomp. If the application is built with the shared version of libomp, this problem is avoided.

Memory topology

HP-UX 11i v2 provides several interfaces to determine the memory configuration of a system or a process. Pstat_getlocality() returns system-wide information, while pstat_getproclocality() returns per-process information. These calls return a mapping of the memory allocated by an application (or all applications) to the LDOMs of the system. This will include the interleave region, which is a locality but not a *Locality Domain*.

For pstat_getlocality(), additional information is provided that describes the processor configuration for each LDOM.

An application could use pstat_getproclocality() to make sure that memory is actually allocated in the same LDOM that requested it.

These pstat() utilities should be used in conjunction with the mpctl() calls (described next) to get a complete topology for the application's execution domain.

For systems that do not incorporate the cellular architecture of the HP Integrity servers, these interfaces will behave as if the system is one LDOM, which, in fact, it is.

Process management

Process management and scheduling are enhanced in HP-UX 11i v2 release for ccNUMA systems. They provide default optimal behavior in application placement and scheduling on HP Integrity servers. They also provide explicit programming interfaces for applications that want to further exploit the ccNUMA characteristics of the underlying system. Specifically, the applications can manage their launch policies as well as processor and locality domain binding.

Launch policies

Launch policies are used by the process manager to best assign where a process or thread starts to execute and what the physical relationship of the processes or threads are to each other. Every process and thread has an assigned launch policy. The launch policy for a thread need not match the launch policy for its process. There are three ways to set launch policies:

- mpctl(2)
- pthread_launch_policy_np(3T)
- mpsched(1)

HP-UX 11i v2 provides several thread-launch policies and process-launch policies for distributing work in various ways among a system's locality domains. These launch policies determine how HP-UX selects the locality domains into which it launches threads or processes.

HP-UX thread-launch and process-launch policies are independent of one another.

When a process creates another process (via fork() or vfork()), the child process inherits the parent process's launch policy. The initial thread in the child process inherits the launch policy of the creating thread (and not that of its process). Other threads in a multi-threaded process inherit their launch policy from the creating thread.

The launch policies (except the None policy) guarantee that HP-UX binds threads and processes to the locality domains in which they start execution. This locality binding causes HP-UX to not migrate the threads and processes across localities.

HP-UX 11i v2 supports the following thread-launch and process-launch policies. Detailed descriptions are in Table 2.

- **None**—Is the default launch policy for threads or processes; see "Default launch policies" for details
- **Round Robin**—Launches by alternating the thread or process placement among all locality domains until all locality domains are selected once, then starts over as needed
- **Fill First**—Fills a locality by placing threads or processes in the locality until all processors are selected. It then spills over to another locality as needed. Once all localities are filled, it starts over as needed.
- **Packed**—Places all threads or processes in the same locality domain; it does not spill over
- **Least Loaded**—Places each thread or process in the locality domain that is least-loaded at the time of its creation

HP-UX supports these policies as thread-launch policies and as process-launch policies. The policies are inherited by child threads and processes as described in the "Inheritance of launch policies" section below.

All launch policies permit the processors in a server's localities to be *oversubscribed*. That is, the number of threads and processes in an application may exceed the number of processors in a single locality, or it may exceed the total number of processors in a server.

The Packed policy causes threads and processes to be launched only within a single locality. All other policies distribute threads and processes among all locality domains using various methods.

Table 2. Launch policy descriptions

Launch policy description	Interfaces

None
None is the default policy for launching threads and processes.
Under the None policy, HP-UX tries to launch threads and processes in a manner that is best for performance.
Do not rely on this policy for a specific launch behavior, as None may differ on future HP hardware platforms and HP-UX releases.
	pthread_launch_policy_np() support:
	`PTHREAD_POLICY_NONE_NP`
	mpctl() system call support:
	`MPC_SETPROCESS_NONE` and `MPC_SETLWP_NONE`

Round Robin
The Round Robin policy launches each new thread or process in a round-robin (alternating) fashion across all locality domains.
Under Round Robin, the newly created thread or process is launched on the least loaded locality, and each subsequent thread or process launches round-robin into remaining localities, each time selecting the least loaded of the remaining localities. Once all localities are used, round-robin starts over, again allowing placement in all localities and beginning with the least loaded locality.
The launch policy tree only extends from the parent to its immediate children. The children then become the root of a new launch policy tree.
	`mpsched` policy name: `RR`
	pthread_launch_policy_np() support:
	`PTHREAD_POLICY_RR_NP`
	mpctl() system call support:
	`MPC_SETPROCESS_RR` and
	`MPC_SETLWP_RR`

Round Robin Tree
This request establishes a tree-based round-robin launch policy for the specified thread. This request differs from PTHREAD_POLICY_RR_NP in which threads become part of the launch tree. This launch policy includes all descendents of the target thread in the launch tree.
The LDOMs used will be selected in the order in which they were created at boot time. So, the process or thread distribution of an application with several levels in the launch tree may vary between runs.
	`mpsched` policy name: `RR_TREE`
	pthread_launch_policy_np() support:
	`PTHREAD_POLICY_RR_TREE_NP`
	mpctl() system call support:
	`MPC_LAUNCH_POLICY_RR_TREE`

Fill First
The Fill First policy launches each new thread or process into the same locality until the number of launched threads or processes equals the number of processors in the locality. Threads or processes then spill over into the next locality and launch there, as needed, until it is filled.
Once all localities are filled, the Fill First policy starts over. When this occurs, all locality domains are again available for placement and Fill First selects the least loaded locality, fills it as needed, and then selects the least loaded of the remaining localities.
	`mpsched` policy name: `FILL`
	pthread_launch_policy_np() support:
	`PTHREAD_POLICY_FILL_NP`
	mpctl() system call support:
	`MPC_SETPROCESS_FILL` and `MPC_SETLWP_FILL`

Fill First is the default policy for programs that are gang-scheduled. The launch policy tree only extends from the parent to its immediate children. The children then become the root of a new launch policy tree.	

Fill Tree
This request establishes a tree-based Fill First launch policy for the specified thread. This request differs from PTHREAD_POLICY_FILL_NP in which threads become part of the launch tree. This launch policy includes all descendents of the target thread in the launch tree.
The LDOMs used will be selected in the order in which they were created at boot time. So, the process or thread distribution of an application with several levels in the launch tree may vary between runs.
	`mpsched` policy name: `FILL_TREE`
	pthread_launch_policy_np() support:
	`PTHREAD_POLICY_FILL_TREE_NP`
	mpctl() system call support:
	`MPC_LAUNCH_POLICY_FILL_TREE`

Packed
The Packed launch policy causes all of the program's threads or processes to be launched into the same locality.
Under the Packed policy, a locality domain can become oversubscribed. That is, the number of threads and processes may exceed the number of processors in the locality. Threads and processes launched under the Packed policy do not spill over into other localities.
	`mpsched` policy name: `PACKED`
	pthread_launch_policy_np() support:
	`PTHREAD_POLICY_PACKED_NP`
	mpctl() system call support:
	`MPC_SETPROCESS_PACKED` and `MPC_SETLWP_PACKED`

Least Loaded
The Least Loaded policy launches newly created threads or processes into the least loaded locality domain in the system at the time of creation.
The least loaded locality domain is the one with the lowest load average within the last second, based on the number of active runnable threads and processes (excluding those in a short-term I/O wait).
	`mpsched` policy name: `LL`
	pthread_launch_policy_np() support:
	`PTHREAD_POLICY_LEASTLOAD_NP`
	mpctl() system call support:
	`MPC_SETPROCESS_LEASTLOAD` and `MPC_SETLWP_LEASTLOAD`

Default launch policies

By default, HP-UX launches threads and processes using a launch policy called None. The None policy implies that HP-UX is free to launch threads and processes as it determines is best for system performance.

No locality binding is established by the None policy, so HP-UX may migrate threads and processes that use this policy to another LDOM.

You should not rely on the None launch policy to provide any specific behavior, because it may change from release to release or from hardware platform to hardware platform.

If you need to guarantee that a specific launch behavior is maintained when running a program on a variety of systems, specify a thread-launch policy or process-launch policy for the program.

HP-UX 11i v2 for HP Integrity server systems uses the following methods for launching threads and processes using the None launch policy:

- **Threads** are first placed by the default HP-UX 11i v2 thread-launch policy (None) into the current locality. Then, if there are more threads than processors in the current locality, threads spill over into another locality, which is selected for highest performance. This is equivalent to a Fill First thread-launch policy with no binding.
- **Processes** are placed by the default HP-UX 11i v2 process-launch policy (None) into the locality with the fewest threads and processes currently in the run queue. This is equivalent to a Least Loaded process-launch policy with no binding.

The None policy may perform differently on other hardware platforms and in future HP-UX releases. All the launch policies are described in Table 2, "Launch policy descriptions."

Discovering the execution domain topology

The mpctl() system call is used to discover the system or the processor set (pset) topology. It can also be used to work backward to discover which pset a processor or LDOM belongs on.

Table 3. MPCTL arguments to find system topology

System CPUs	System LDOMs	Processor Set CPUs	Processor Set LDOMs
MPC_GETNUMSPUS_SYS	MPC_GETNUMLDOMS_SYS	MPC_GETNUMSPUS	MPC_GETNUMLDOMS
MPC_GETFIRSTSPU_SYS	MPC_GETFIRSTLDOM_SYS	MPC_GETFIRSTSPU	MPC_GETFIRSTLDOM
MPC_GETNEXTSPU_SYS	MPC_GETNEXTLDOM_SYS	MPC_GETNEXTSPU	MPC_GETNEXTLDOM
MPC_GETCURRENTSPU	MPC_LDOMSPUS_SYS		MPC_LDOMSPUS
	MPC_SPUTOLDOM		

For application developers, it is probably best to focus on the processor set (pset), since psets will sometimes be used to manage applications and users. If a system is not using psets, the whole system becomes the default pset. For more information, refer to the Processor sets white paper, www.hp.com/products1/unix/operating/hpux11i_proc_sets.html.

Utilizing the execution domain

The mpctl system call is used to assign processes and threads to a specific LDOM and/or specific processors. A thread is also referred to as a Light Weight Process (LWP).

Table 4. MPCTL arguments to bind processes and threads

Process	Thread (Light Weight Process)
MPC_SETPROCESS	MPC_SETLWP
MPC_SETPROCESS_FORCE	MPC_SETLWP_FORCE
MPC_SETLDOM	MPC_SETLWPLDOM

The MPC_SETPROCESS and MPC_SETPROCESS_FORCE assign a process to a specific processor, whereas MPC_SETLWP and MPC_SETLWP_FORCE assign a thread to a specific processor. The MPC_SETPROCESS and MPC_SETLWP request an advisory assignment, a request per se. The MPC_SETPROCESS_FORCE and MPC_SETLWP_FORCE request mandatory assignment that the scheduler will obey until the system configuration changes. The MPC_SETLDOM and MPC_SETLWPLDOM assign a process or thread to a locality domain.

Discovering a process's locality bindings

The mpctl system call can be used to discover what kinds of bindings are assigned for a process or thread.

Table 5. MPCTL arguments to find binding values

Process	Threads
MPC_GETPROCESS_BINDVALUE	MPC_GETLWP_BINDINGTYPE

Inheritance of launch policies

Every thread and process has a Launch Policy associated with it. If no explicit policy is assigned, then, in general, the child process or thread inherits its policy from its parent. There is no explicit relationship between the launch policy of a process and its child threads.

HP-UX maintains which thread-launch and process-launch policies are in effect for threads and processes by using *policy trees*.

A policy tree is a collection of threads or processes that share the same launch policies and *launch count* details. The launch count and the thread-launch or process-launch policy are maintained by the policy tree's *root*.

HP-UX 11i v2 supports two types of policy trees:

- **Thread policy trees** are maintained as part of the thread structure and include a thread-launch policy and launch count.
- **Process policy trees** are maintained as part of the process structure and include a process-launch policy and launch count.

HP-UX refers to the root of the associated policy tree when a pthread_create() call creates a new thread, or when a fork() call creates a new process. HP-UX launches the new thread or process using the thread-launch or process-launch policy specified by the root. HP-UX also updates the policy tree's launch count.

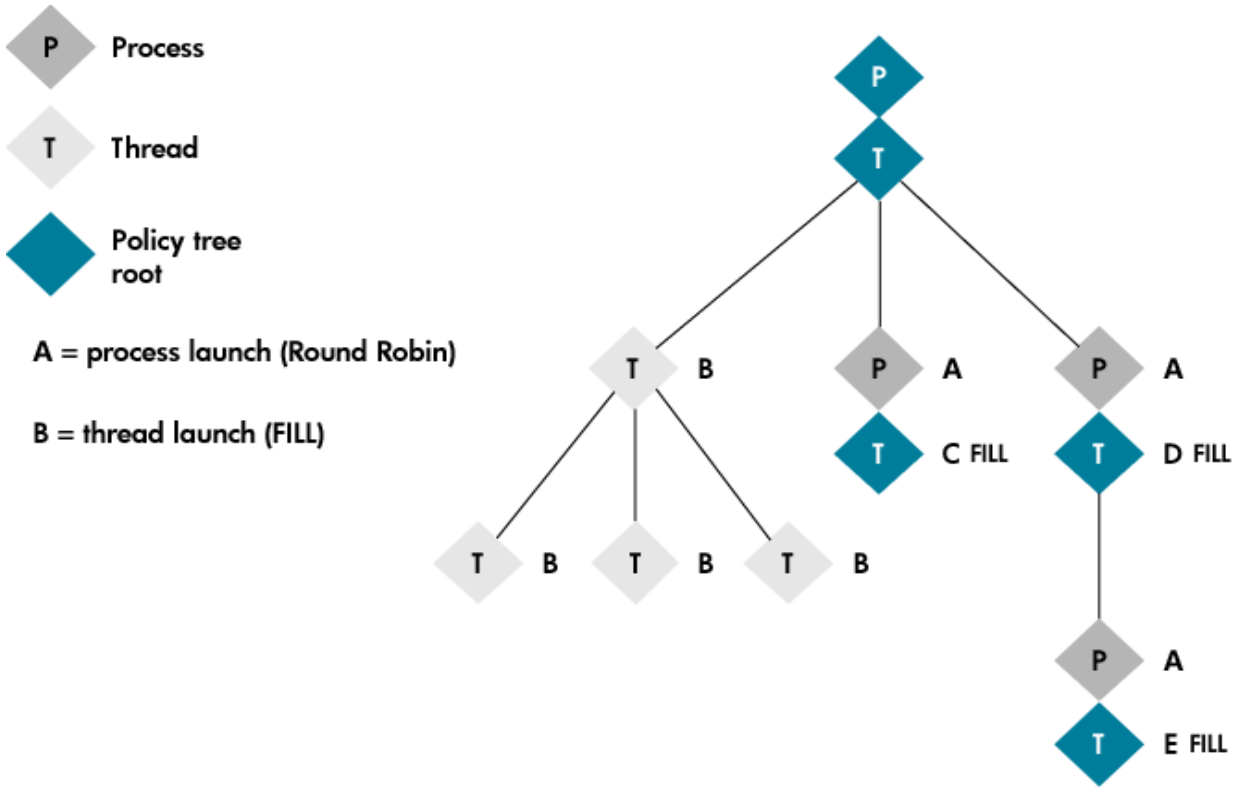
HP-UX maintains the launch details to track the number and location of threads or processes launched within a policy tree. Every policy tree has its own launch details.

HP-UX establishes a new policy tree, which has a new launch count and may have a different launch policy, under the following circumstances:

- When a thread changes the thread-launch policy, such as when calling the pthread_launch_policy_np() routine or mpctl() system call, the target thread becomes the new policy tree's root.
- When a process has its process-launch policy changed, such as when mpctl() is called, the process is the new policy tree's root; any thread in a process can change the process-launch policy.
- When a process calls fork() to create a process, the new process becomes a member of the same process policy tree that the parent process belongs to. The thread in the new process becomes the root of a new thread policy tree, which inherits the thread-launch policy of the creating thread.

Figure 4 shows a sample process, the threads and processes it creates, and the associated policy trees and launch policies.

Figure 4. Sample HP-UX launch policy trees



The first process in Figure 4 is launched with a Round Robin Tree process-launch policy and a Fill First Tree thread-launch policy. As a result, the root of the process policy tree (labeled A) establishes a Round Robin Tree process-launch policy, and the root of the thread policy tree (B) establishes a Fill First Tree thread-launch policy.

The thread created by the initial thread in Figure 4 is a member of the same thread policy tree, and HP-UX launches it using the Fill First Tree thread-launch policy. Likewise, threads created by the secondary thread also are members of the same thread policy tree (B).

For the two child processes created by the initial thread in Figure 4, HP-UX creates one process and one thread for each. The child processes are members of the same policy tree (A) as the thread's parent and are launched using the Round Robin Tree process-launch policy. The child processes' threads, however, are the roots of new thread policy trees.

These new thread policy trees (C and D) inherit the Fill First Tree launch policy from the creating thread's policy tree (B).

Finally, the thread in policy tree D calls fork() to create yet another child process. This new process is a member of the same policy tree as the thread's process (A). The new thread that is created by the fork() call is the root of a new thread policy tree (E) which inherits the thread-launch policy from the thread's policy tree (D).

If we consider this example with Fill First and Round Robin launch policies, only the parent and its direct children will form the launch policy trees. The children become the root of another launch policy tree. By limiting the launch policy tree this way, the application's behavior becomes more deterministic.

Memory allocation

The overall strategy of the memory system is to keep private data local and shared data global. Therefore, the default behavior will be adequate for most applications. Application developers who understand the data structures of the application and how the data is accessed can use the tools presented here to optimize performance.

malloc()

Malloc and the new operator (C++) are probably the most common memory allocators, but there is no special method to control what memory locality they get their memory from. Malloc and the new operator will attempt to satisfy their memory allocations from the local memory of the LDOM they are called in.

mmap() and shmget()

Mmap is the primary tool for allocating and managing ccNUMA memory.

Table 6. Memory allocation flags

MMAP FLAG	SHMGET FLAG	Physical memory will come from …
MAP_MEM_LOCAL	IPC_MEM_LOCAL	The locality of the CPU making the system call
MAP_MEM_INTERLEAVED	IPC_MEM_INTERLEAVED	Interleaved memory (round robin or closest first if no interleaved memory available)
MAP_MEM_FIRST_TOUCH	IPC_MEM_FIRST_TOUCH	The locality of the first CPU to touch the page

If MAP_SHARED is specified, then MAP_MEM_INTERLEAVED is the default. For MAP_PRIVATE, MAP_MEM_FIRST_TOUCH is the default. For MAP_MEM_LOCAL and MAP_MEM_FIRST_TOUCH, the allocation occurs in the LDOM of the calling thread or process. Use mpctl() (described in "Utilizing the execution domain") to ascertain and control the execution domain.

Fortran/common/

Fortran common and static data (such as local variables compiled with +save) is mapped to the processes data/bss section. The system command, chatr, has been extended to select where to map this memory. The default is to map this data in local memory. Some programs will want to map common to interleave memory, especially if all the threads cannot execute within one LDOM. Depending on the application, reference patterns to data may also lead to choosing interleave memory as the appropriate place to map data in /common/.

Performance-optimized page sizing (POPS)

HP-UX 11i v2 implements performance-optimized page sizing (POPS). Traditionally, UNIX implements a 4 KB page, but for modern applications this is much too small to efficiently map large amounts of memory. Using 4 KB pages, the processor can only map small regions of an application's memory, and it incurs a very large penalty doing TLB translations that degrade performance. POPS transparently increases the page size for applications up to the vps_ceiling kernel tunable. Applications are not aware of the page size. Developers and users can set page size hints through link-time flags and using the chatr tool.

The largest page size supported by HP-UX 11i v2 is 4 GB. Valid values are 4 KB, 16 KB, 64 KB, … , 1 GB, 4 GB. That is, $4K^n$, where n varies from 1 to 11.

When a process allocates memory in the interleaved region, it is important to match the requested amount of memory to the page sizes available for the process. For example, ensure that the amount of memory requested is a simple multiple of a page size that is available on the system.

Memory exhaustion and paging

The VM system has a strategy to allocate memory. The order in which localities are searched depends on the memory object. Objects that desire interleaved memory will first search the interleaved locality and then perform a round-robin search among all the local memory localities. Objects that desire cell local memory will first search that local memory and then will search other localities in a closest-to-furthest order.

When no memory is available from the desired locality, the operating system will attempt to provide the "next best" locality possible. If that locality also has no memory available, then the next best will be chosen, and so on, until the entire system has been searched. It is important to remember that requests for a particular locality of memory are always *hints*, never guarantees.

When memory is exhausted and a page must be paged out, the VM system has a similar strategy when it must page it back in. The VM system will use the same policy it used originally to choose the locality to start its allocation task.

Page migration

Page migration is not supported directly. In the case of memory exhaustion, memory pages may be paged to backing storage (swap space). When it is paged back in, it will use the same heuristics to determine where physical memory should come from. If the process has migrated from one LDOM to another LDOM, the parts of the process that desire cell local memory will first look in the cell local memory of the new LDOM.

Summary

HP-UX 11i v2 provides application developers with a comprehensive set of tools to maximize performance on the HP Integrity servers. Most applications will be able to exploit the system with no changes, and applications that do need to change have a complete set of tools to manage data locality and the execution domain.

References

HP-UX SCA Programming and Process Management, Version 1.0, October 18, 1999.

VM ccNUMA: Physical Memory Allocation and Page Placement, Design Overview & Specification, Version 1.3.

HP-UX Variable Page-size for IA Design Overview & Specification, May 16, 2002.

HP-UX PM ccNUMA Investigation Report, Version 0.2, Harshad Parekh and Anushree Ramakrishnan.

HP-UX 11i release names and release identifiers

With HP-UX 11i, HP delivers a highly available, secure, and manageable operating system that meets the demands of end-to-end Internet-critical computing. HP-UX 11i supports enterprise, mission-critical, and technical computing environments. HP-UX 11i is available on both PA-RISC systems and Itanium-based systems.

Each HP-UX 11i release has an associated release name and release identifier. The uname (1) command with the -r option returns the release identifier. The following table shows the releases available for HP-UX 11i.

Table 7. HP-UX 11i releases

Release name	Release identifier	Supported processor architecture
HP-UX 11i v1	B.11.11	PA-RISC
HP-UX 11i v1.5	B.11.20	Intel Itanium
HP-UX 11i v1.6	B.11.22	Intel Itanium
HP-UX 11i v2	B.11.23	Intel Itanium

09/2003

5981-7419EN