Configuring Tru64 UNIX® for Large Memory Applications in a NUMA Environment

NUMA Overview 1

AlphaServer GS320 Family Architecture 1

AlphaServer GS1280 Family Architecture 2

Tru64 UNIX Architecture 2

RAD Configurations 3
- GS320 Family RAD Configuration 3
- GS1280 Family RAD Configuration 3

VM Hot Spots 3

Enlarging RADs 4

Conclusion 5

Performance Tests 5

NUMA Overview

In most application environments, where many processes use a relatively small address space, Tru64 UNIX's default Non-Uniform Memory Access (NUMA) implementation does a good job of managing system resources. However, when the application environment consists of a small number of processes actively using a large amount of memory, performance can suffer.

Tru64 UNIX supports two NUMA hardware architectures: the AlphaServer GS320 family (GS80, GS160 and GS320), which uses a hierarchical switch, and the AlphaServer GS1280 family (ES47, ES80 and GS1280), which uses interprocessor ports.

AlphaServer GS320 Family Architecture

The AlphaServer GS320 family is built using multiple Quad Building Blocks (QBBs). Each QBB can contain up to four processors and up to 32 gigabytes of memory. The system buses for each QBB are tied together via a hierarchical switch. A QBB is able to access resources on a remote QBB through the hierarchical switch. However, those remote accesses are slow compared to a local access, taking approximately three-times longer.

AlphaServer GS1280 Family Architecture

The AlphaServer GS1280 family is based around the processor itself. Each processor has four interprocessor ports (IP). Labeled for the cardinal compass directions, they tie each processor to the adjacent processors. Using the IPs, each processor is capable of communicating directly with adjacent processors. In addition, each processor may have memory and I/O controllers directly attached through additional ports. With this architecture, each processor has direct access to any memory and I/O controllers attached directly to it, without having to traverse a system bus.

On the ES47 and ES80 systems, CPUs are tied together in a ring-like manner using the North and South IPs only. On GS1280 systems, they form a toroid mesh using all four IP connections. Each processor is able to access the memory and devices connected to the other processors though this mesh with significantly increased bandwidth compared with previous systems using a more conventional bus.

With Tru64 UNIX version 5.1B, a GS1280 system supports up to 64 processors, with each processor supporting up to 8 GB of memory. Although the system supports processors with no memory, it is generally advisable not to configure the system in that manner. A fully configured system has 512 GB of physical memory

Tru64 UNIX Architecture

Tru64 UNIX configures the system into Resource Affinity Domains (RAD). A RAD is a software-based grouping of resources that are in a close physical proximity. Generally, each RAD contains one or more processors and a portion of the system's memory. It may also contain various I/O device controllers.

Each RAD maintains the data structures to control scheduling, memory management, and such. Memory is broken into pages, and the operating system manages each page as a whole unit. The memory manager for each RAD acts as if it were an independent system. Under normal circumstances, there is little sharing of memory resources between the RADs.

Each RAD keeps its own list of active pages, inactive pages, Unified Buffer Cache (UBC) pages, and free pages, along with dedicated kernel threads and associated locks to manage them. As with all virtual memory operating systems, it tries to maintain a minimum number of pages on the free list. Once the free page list drops below a certain size, the memory manager begins operations to recover memory.

Just as every thread has a pointer to the processor it last ran on, it also has a pointer to its "home" RAD. As a thread runs, by default any memory allocated will be from its home RAD. Because a thread tries to run on its home RAD, locality of reference to the memory is generally maintained.

As the system runs, the load can become unbalanced. Some RADs have too much work, while other RADs are relatively idle. The scheduler maintains a variable called `desired_load`. This is, in a sense, the average load on the system. If a processor's load is below the value specified with `desired_load`, it will try to pull work from processors whose load is above that level. The goal is to keep the load as even as possible across the entire system.

The first step is calling `thread_try_steal()`, which looks for runnable threads on nearby RADs that are within `sched_distance` hops away and run them remotely. The thread's home RAD remains unchanged; the thread is simply scheduled to run on the "borrowing" RAD.

After `thread_try_steal()` comes `rad_try_steal()`. In this case, the RAD is looking for entire tasks that can be migrated from an overloaded RAD. This will change the task's home RAD to the "stealing" RAD and move its memory resources as well. However, this does have timing safeguards built in to prevent the system from constantly moving processes around.

Overall, the system makes effective use of available resources. A thread maintains a close proximity to the resources it uses. By relocating those resources, you can improve load balance.

The basic design principles of the Tru64 UNIX NUMA implementation can be summed up as such:

- Resources that are close in proximity are grouped into a RAD.
- Each RAD is responsible for managing its resources.
- Each task has a home RAD.
- A schedulable thread will try to remain on its home RAD.
- By default, a running thread will allocate resources from its home RAD.
- An idle RAD will try to pull threads from an overloaded RAD.
- The system should appear as a traditional SMP environment to user applications.

RAD Configurations

The system configures each RAD at boot time. What constitutes a RAD is dependent upon the family of system.

GS320 Family RAD Configuration

In the GS320 family of systems, a direct, one-to-one relationship exists between the RAD (a software abstraction) and the QBB (a physical piece of hardware). Each QBB will have a RAD to control it and its resources. Likewise, each RAD will have only one QBB.

For example, a system with 16 processors and 32 GB of memory, configured homogeneously, would consist of four QBBs, with each QBB containing four processors and having 8 GB of physical memory. They would be tied together via a hierarchical switch.

When booted, the system would have four RADs, with each RAD controlling four processors and managing 8 GB of memory.

GS1280 Family RAD Configuration

On a GS1280, the RAD is a more flexible software abstraction to help control resources. By default, a RAD is created for each processor on the system, with each RAD managing the memory and controllers attached its processor.

For example, a system with 16 processors and 32 GB of memory, configured homogeneously, would consist of 16 processors, each with 2 GB of memory.

When booted, the system would have 16 RADs, with each RAD controlling a single processor and managing 2 GB of memory.

VM Hot Spots

In almost all instances, the resource management and load balancing algorithms do a good job of maintaining system performance. However, for some workloads, the single processor per RAD configuration can actually interfere with performance. The VM subsystem, when allocating memory for a thread, focuses primarily on the memory in the thread's home RAD and ignores the free memory in other RADs.

Heavy memory usage on a RAD can force the VM subsystem to page out (or even swap out) processes, with no regard for the memory available on other RADs. If more memory is actively used than is physically available on a RAD, it causes that RAD's VM subsystem to begin memory

recovery operations. It is possible for a RAD to be thrashing while a surplus of free memory is present on the overall system.

The scheduling load balancing is unable to help eliminate the overload. Trying to relocate the task would simply relocate the imbalance. Several methods are available for resolving this problem.

Enlarging RADs

One option is to increase the resources available to a RAD. In the past, this has involved adding more memory to the system. However, over sizing a system can be a very expensive solution.

Since its release, Tru64 UNIX Version 5.1B has provided a hidden sysconfig parameter called `cpus_in_rad`. For a GS1280 system, it tells the kernel how many processors to configure into each RAD. It has no effect on other system types, even when it is set.

Increasing `cpus_in_rad` enlarges the size of each RAD on the system. The threads and data structures responsible for managing system resources have more resources to deal with before reaching a starved state.

Prior to Version 5.1B-3, `cpus_in_rad` was limited to a maximum value of 2. When set to 2, each pair of processors would be configured as a RAD. On a homogenously configured system, this would double the amount of resources available to each RAD. The tradeoff is it cuts the number of RADs in half.

For our example system from above with 16 CPUs and 32 GB of memory, setting `cpus_in_rad` to 2 would give eight RADs, each managing 4 GB of memory.

With the release of Version 5.1B-3, the limit of 2 for `cpus_in_rad` was raised to 32. The allowable settings of `cpus_in_rad` are powers of 2. If it is set above 32, it will be lowered to 32; if set to value that is not a power of 2, it will fall back to the default. If the total number of processors is not evenly divisible by `cpus_in_rad`, the last RAD will only contain the remaining processors.

So, with our example system, setting `cpus_in_rad` to 4 would give us 4 RADS, each managing 8 GB of memory. From a software view point, it would be configured similar to the GS320 example. An application could be moved from the example GS320 system and placed onto the example GS1280 system with only minor adjustments, if any at all.

In theory, `cpus_in_rad` can be raised to minimize system overhead. But as with all tunable parameters, there are trade offs when adjusting it. One of those trade offs is that by increasing `cpus_in_rad`, fewer RADs are available to manage those resources. This causes a heavier load to be placed upon the fewer remaining threads and structures responsible for managing those resources. In some instances, a system could be rendered unusable by increasing `cpus_in_rad` too high.

The major concern with setting `cpus_in_rad` too high relates to locking contention. As the number of RADs decreases, the granularity of locking decreases proportionally. Many software locks used by the operating system to protect resources grow in scope as the size of a RAD increases. In our example of setting `cpus_in_rad` to 4, the lock protecting page queues would now have 4 times the contention.

For an environment where an application consistently exceeds the resources available to a single processor on a GS1280-family system, or an application drastically exceeds the memory resources of a processor, increasing `cpus_in_rad` could be of benefit.

By default, `cpus_in_rad` remains at the default value provided by the hardware. To change it, modify the generic sysconfig attribute `cpus_in_rad` and reboot the system.

Conclusion

Tru64 UNIX generally does a good job of managing the system resources. In addition, there are NUMA library calls to allow a user application to have greater control of how its resources are allocated and controlled by the operating system.

However, if an application's requirements exceed the capability of a single RAD and the application makes no attempt to control how those resources are allocated, system performance degrades rapidly. This is contrary to the design goal of keeping the system view and performance similar to traditional SMP systems.

To address this, the parameter `cpus_in_rad` was expanded. This allows system architects, system managers, and those responsible for system performance greater control over their system configuration from a software view point.

There is no recommended value to set `cpus_in_rad` to. If that value were to exist, it would be a constant and not a tunable. As in all tuning, what produces excellent performance in one environment could produce an unstable system in another.

Performance Tests

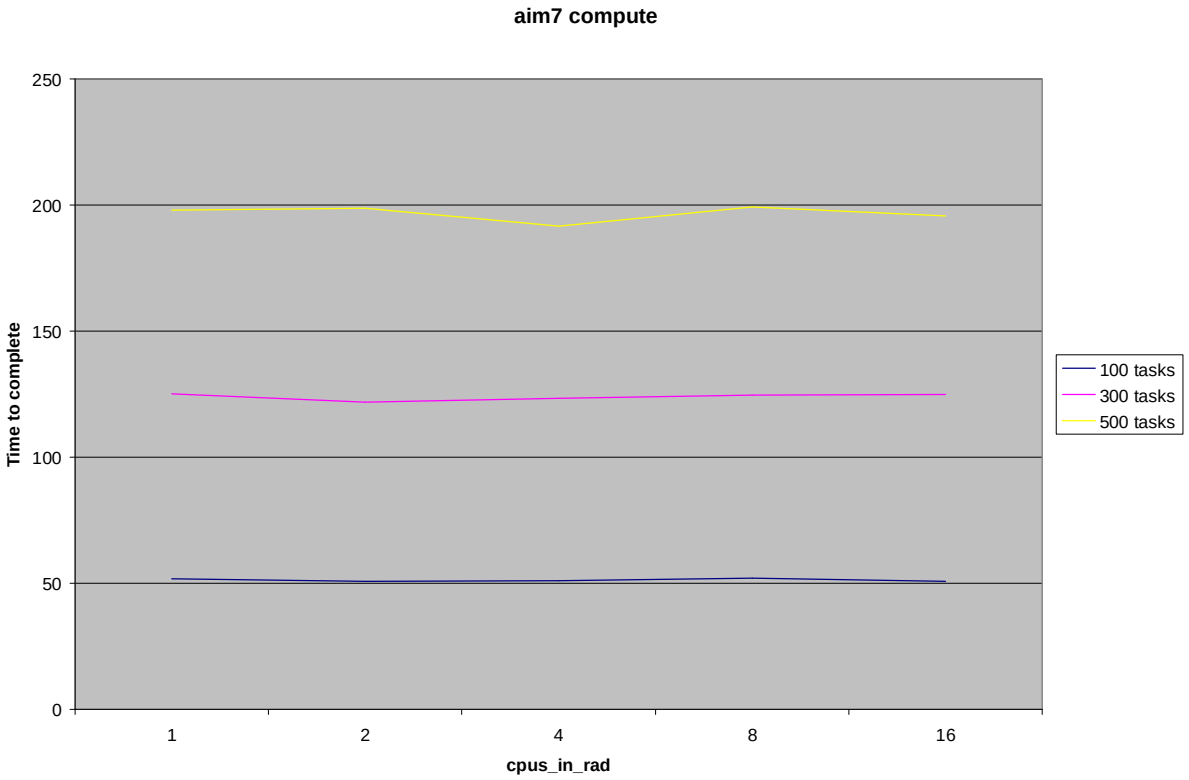
Several performance tests were run to check impact of changing `cpus_in_rad`.

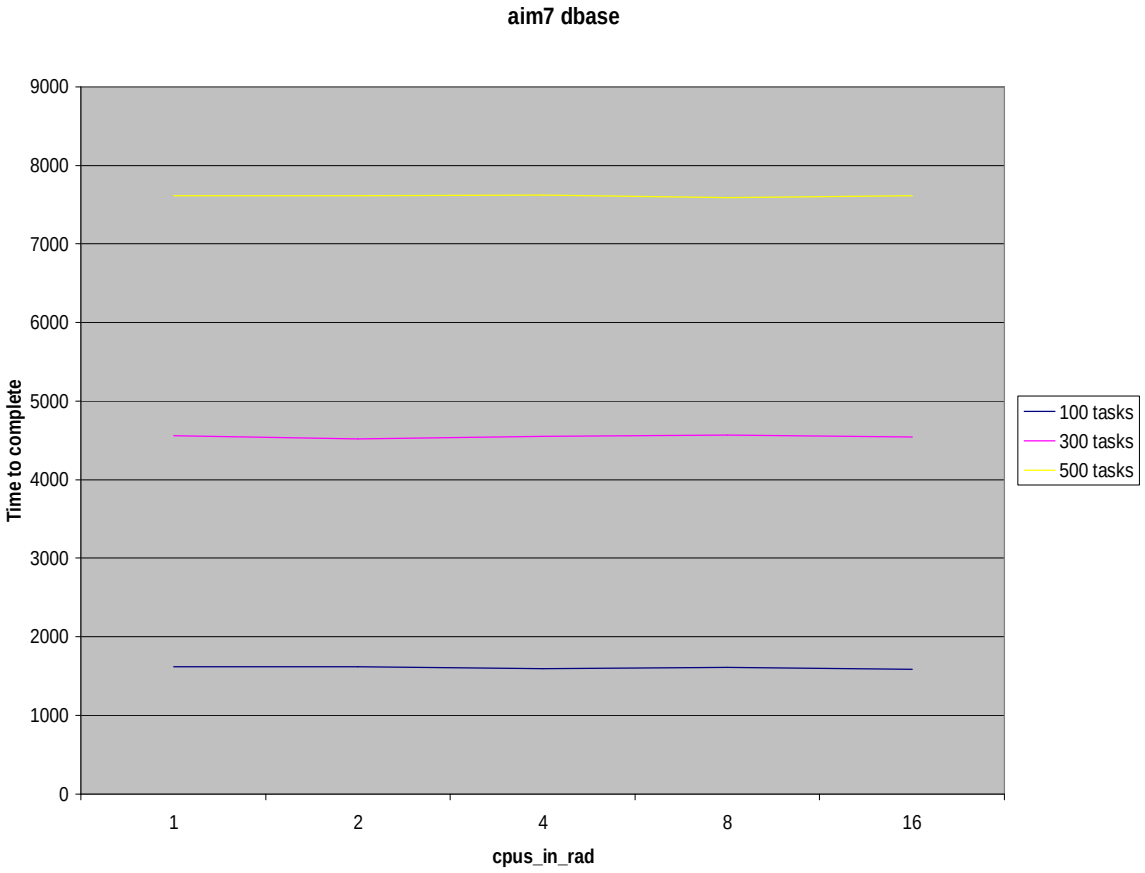
The aim7 compute and dbase performance benchmarks were ran on a 32p GS1280 with a load of 100, 300 and 500 tasks. This was to simulate a fairly normal work load.

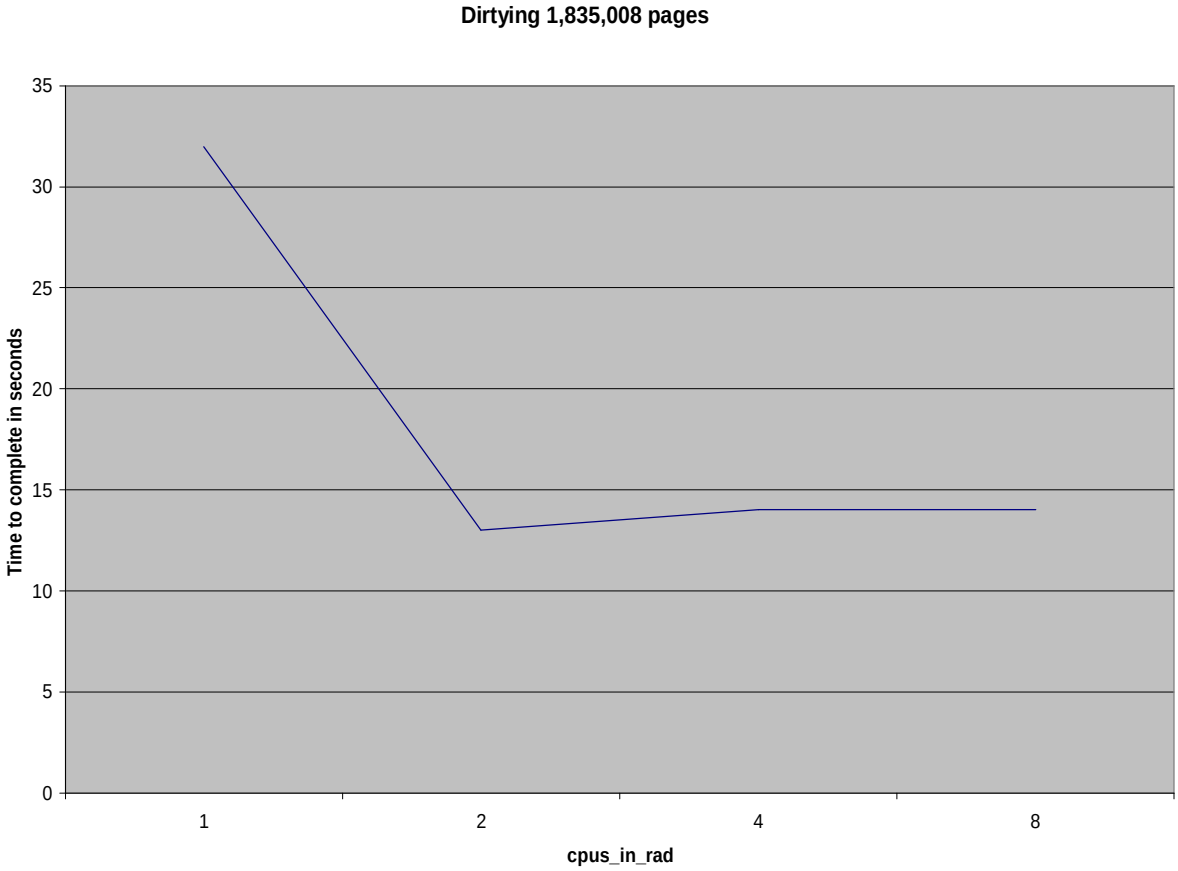
Several custom programs were run on a 16p GS1280 with different settings for `cpus_in_rad` to simulate a sever load.

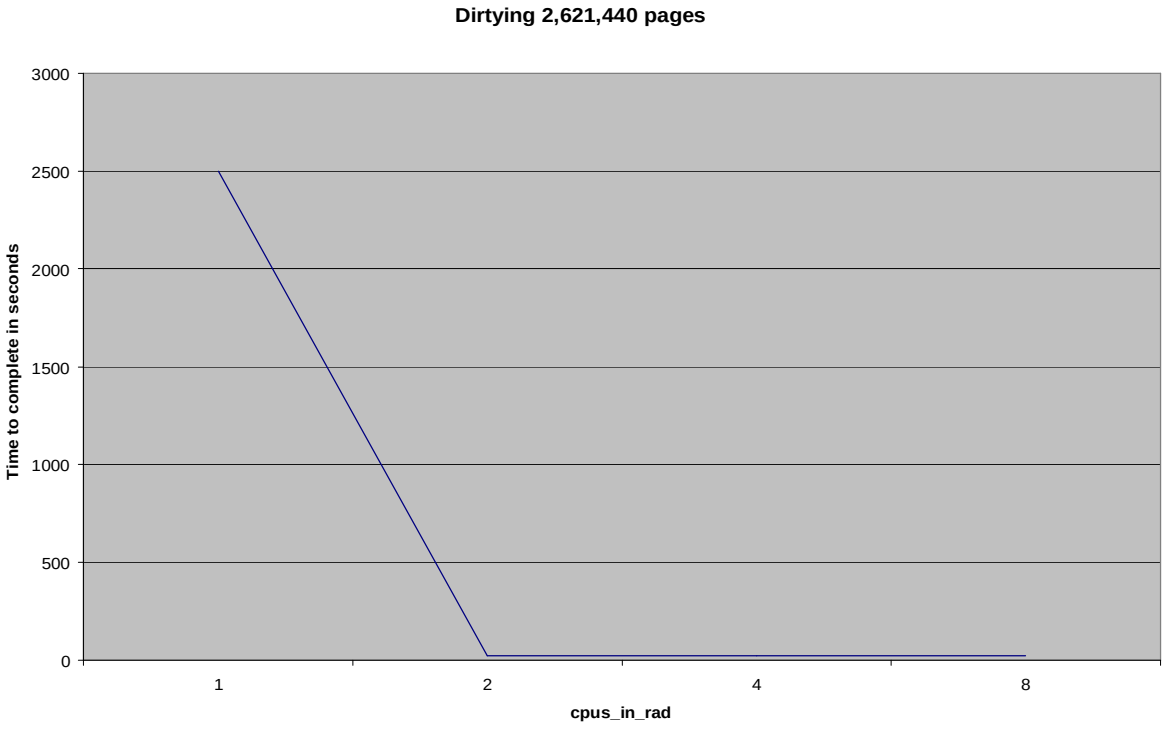
The first dirtied an array of memory 3 times. Different sizes of the array were run. The goal here was to stress the VM subsystem.

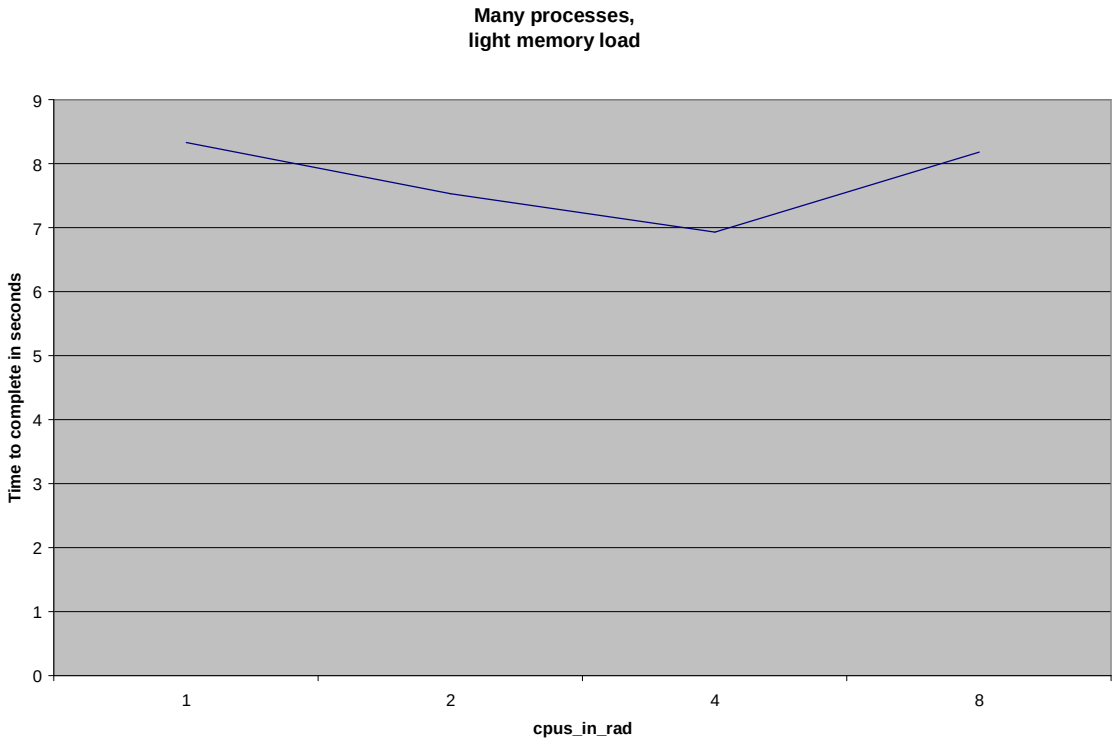
The second had 1000 processes dirty a variable amount of memory. The goal here was to stress scheduling.


aim7 compute
250
200
150
100
50
0
Time to complete
1
2
4
8
16
cpus_in_rad
100 tasks
300 tasks
500 tasks


aim7 dbase
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Time to complete
1
2
4
8
16
cpus_in_rad
100 tasks
300 tasks
500 tasks


Dirtying 1,835,008 pages
Time to complete in seconds
35
30
25
20
15
10
5
0
1
2
4
8
cpus_in_rad


Dirtying 2,621,440 pages
Time to complete in seconds
3000
2500
2000
1500
1000
500
0
1
2
4
8
cpus_in_rad


Many processes,
light memory load
Time to complete in seconds
0
1
2
3
4
5
6
7
8
9
1
2
4
8
cpus_in_rad