

---

## Application Execution Guidelines for vSMP Foundation Aggregated Virtual Machine

Document Date : June 11, 2009

---

### Overview

vSMP Foundation is a virtualization software that creates a single virtual machine over multiple x86-based systems. The resources of these systems are aggregated and available to the virtual machine in a transparent manner. Depending on the number of systems, and their resources, the aggregated virtual machine can provide large number of processors, I/O devices, and memory footprint, with similar characteristics to SMP systems. This document will refer to a virtual machine provided by vSMP Foundation as an aggregated virtual machine (AVM).

A unique characteristic of vSMP Foundation is its usage of local memory to enhance system performance using sophisticated caching techniques. Similarly to other large-scale multi-processor systems, processor and system caches can significantly improve application performance if the placement policy for the application's threads or processes (hereafter, "tasks") is carefully chosen. vSMP Foundation implements more aggressive caching techniques comparing to traditional multi-processor systems. Due to the caching effect of vSMP Foundation, keeping tasks executed on the same processor(s), system or specific set of systems throughout the run, may improve the task performance.

vSMP Foundation aggressive caching techniques provide a unique combination of Non-Uniformed Memory Architecture (NUMA) and Cache-Only Memory Architecture (COMA). While the memory architecture resulted from combination of NUMA and COMA requires a full textbook to be thoroughly explored and explained, in the scope of this document it suffices to note that:

1. Every system that is part of an aggregated virtual machine is considered a NUMA node, at the highest level of the NUMA hierarchy.
  - a. Multiple levels of NUMA nodes may exist if the underlying systems are NUMA as well (Intel's QPI platforms or AMD DCA platforms).
2. The total memory available in all the NUMA nodes is less than the total memory on all systems comprising the AVM. The remainder is used by vSMP Foundation and its caching system.
3. Unlike NUMA, the memory locality in COMA is transient over time:
  - a. A cache-line accessed from a specific NUMA node will become local to it. Subsequent access to that cache-line is at local NUMA node memory access speed as long as no write accesses are made to it from processors on other NUMA nodes.

The cache-line will stop being local to the NUMA node due to a write access from a different NUMA node, or due to cache evacuation resulting from caching of other cache-lines. Keeping the above in mind, note that an Operating System task relocation policy may have a detrimental effect on caching performance. Special care should be taken to minimize task relocation.

---

## Process Placement

### Tools

To assist the user with assigning tasks to specific processor cores, Linux offer a few easy-to-use tools. In Linux, these take the form of command-line utilities that are detailed below.

#### **numactl**

The **numactl** utility is used as a prefix on the command line of a command the user wishes to execute. Of the options to **numactl**, the most useful for process placement are the **cpunodebind=<nodes>** and the **--physcpubind=<CPUs>** arguments. The use of **numactl** with these arguments will restrict the task to run only on the NUMA nodes or the CPUs listed in the argument, respectively.

For example, consider an AVM comprised of 2 systems, each with 2 quad-core processors, totaling 16 cores. Such AVM will present the Operating System with a NUMA hierarchy of 2 NUMA nodes (numbered 0 and 1), each containing 2 processors. The following command runs the task **<myapp>** only on the second NUMA node (NUMA node 1 when counting from 0, which represent the AVM's second system):

```
numactl --cpunodebind=1 <myapp> [arguments]
```

The task and any other tasks it spawns will be restricted to run only on CPUs that reside on the requested NUMA node. In the case above, the target NUMA node is 1, and the CPUs are CPUs 8 through 15.

Using the **--physcpubind** argument, a task can be placed to specific CPUs. The placement described in the example above could be achieved using the command below, which makes use of CPU numbers instead of NUMA node numbers:

```
numactl --physcpubind=8-15 <myapp> [arguments]
```

As with the prior example, the task and any other tasks it launches to be restricted to run only on CPUs 8 through 15, which correspond to the second system in the AVM of this example.

In both of these cases, each task will have the same CPU affinity mask, 0xFF00, which represents CPUs 8-15. Whenever a task has multiple bits set in its affinity mask, the Linux kernel will dynamically assign it to one of the CPUs represented by the mask when it is in "runnable" state. The task will be moved among these CPUs based on Linux scheduling policy, and will never be scheduled to CPUs outside the ones represented by the mask.

Note that vSMP Foundation manages its cache on a per-system level, hence restricting tasks to a specific system may be sufficient (if the system has enough CPUs to service the task) and a finer-grained assignment to specific CPUs may not be required.

The **numactl** command can also provide information about the system's hardware. Please refer to numactl documentation for more information.

### **taskset**

Another Linux utility that provides a refined assignment of tasks to CPUs is **taskset**. Similar to **numactl**, **taskset** is used as a prefix to a command to execute. The **taskset** command argument **--cpu-list** is followed by a list of CPUs to be used by the task. The list of CPUs can contain one or more CPU numbers separated by commas or a range of CPUs expressed as **<lowest>-<highest>**. For example, if the user wants the task **<myapp>** to run only on CPU 5:

```
taskset --cpu-list 5 <myapp> [arguments]
```

A group of CPUs can also be selected. For the AVM described in the former example, the following commands all have the same effect:

```
taskset --cpu-list 8-15 <myapp> [arguments]  
numactl --physcpubind=8-15 <myapp> [arguments]  
numactl --cpunodebind=1 <myapp> [arguments]
```

**taskset** can also be used to set the affinity mask for tasks that are already running. For more information, please refer to the **taskset** man page.

---

## Typical Use Cases

### *Serial Jobs*

Serial jobs have only a single task. Optimal performance will be achieved by placing a serial job, using either **numactl** or **taskset**, to a specific NUMA node or even to a specific CPU. System-level placement will utilize the system-level cache to the maximum extent possible. Specific CPU placement will also have optimal utilization of the CPU cache.

### *OpenMP Jobs*

For OpenMP jobs, ScaleMP's recommendation is to use Intel's OpenMP runtime library, which would yield the best result.

Intel's OpenMP runtime library has the ability to control the affinity of OpenMP threads to physical processing units, controlled via the **KMP\_AFFINITY** environment variable. Documentation for the **KMP\_AFFINITY** environment variable can be found at:

[http://www.intel.com/software/products/compilers/docs/fmac/doc\\_files/source/extfile/optaps\\_for/common/optaps\\_openmp\\_thread\\_affinity.htm](http://www.intel.com/software/products/compilers/docs/fmac/doc_files/source/extfile/optaps_for/common/optaps_openmp_thread_affinity.htm)

The most important practice when running an OpenMP application is to keep the threads running in a "packed" manner on minimal number of NUMA nodes. Adding the **verbose** key to the **KMP\_AFFINITY** environment variable will print details of the actual affinity of the threads to CPUs to the stdout. An example of how to run a task with 4 threads in a "packed mode":

```
export OMP_NUM_THREADS=4
export KMP_AFFINITY="granularity=fine,compact,verbose"
```

Another important practice is to use the latest Intel Compilers which include scalability enhancements that lead to more scalable binary code, resulting in better utilization of the aggregated virtual machine. Several important aspects related to placement and scalability available in the following versions of Intel Compiler suite:

1. 9.1.x series: 9.1.052 and later
2. 10.1.x series: 10.1.011 and later
3. All 11.x versions or later

## **MPI Jobs**

Running MPI applications on the aggregated virtual machine can be done with various MPI implementations (e.g. HP-MPI, Intel-MPI, MPICH1, MPICH2, etc.). Commercial applications typically support one or more MPI implementation options.

While it is possible to use any MPI implementation, using MPICH2 tuned for vSMP Foundation is most likely to yield the best performance. MPICH2 as well as MPICH1 tuned for vSMP Foundation can be obtained from ScaleMP's support portal (<http://support.scalemp.com>).

## **MPICH**

For applications that support MPICH, it is recommended to use MPICH2 tuned for vSMP Foundation. The latest version can be obtained from ScaleMP's support portal, under the 'Libraries' section.

### **Obtaining MPICH2 tuned for vSMP Foundation:**

1. Log-in to ScaleMP's support portal (<http://support.scalemp.com>)
2. Under the 'Libraries / MPICH2 tuned for the vSMP Foundation architecture' section, download the rpm, as well as the 'README' file
3. Install the rpm: `rpm --install [latest rpm]`  
MPICH2 installs into: /opt/mpich2\_vSMP

### **Configuring your application:**

Make sure your application is configured to use MPICH2, and is pointing to the right MPICH2 directory (e.g. /opt/mpich2\_vSMP).

Please check the application specific guidelines for instructions of how to set up MPICH2 with your specific application.

### **Environment variables:**

The following environment variables should be set before running your application

```
export VSMP_PLACEMENT=packed
export VSMP_MEM_PIN=yes
```

## **HP-MPI**

When running an application with HPMPI, the following environment variables should be set prior to running the application:

```
export MPI_BIND_MAP=0,1,2,3,4,5,6,7 (For example)  
export MPIRUN_OPTIONS="-cpu_bind=map_cpu,v"  
export HPMP_FRAGSIZE=131072  
export MPI_SHMEMCNTL=16,24000000,4000000
```

**MPI\_BIND\_MAP** specifies a list of CPUs to which MPI ranks will be affined. The list should be replaced with a list of integers, zero to #cpus-1. For more information on HP-MPI CPU affinity settings, refer to the HP-MPI user's guide available from <http://docs.hp.com/en/B6060-96022/B6060-96022.pdf>.

## **Intel MPI (3.2 and above)**

For applications that use Intel MPI, it is recommended to set the following environment variables prior to running the application, to yield improved performance:

```
export I_MPI_SHM_BUFFER_SIZE=131072  
export I_MPI_PIN_MODE=mpd  
export I_MPI_PIN_PROCESSOR_LIST=0,1,2,3,4,5,6,7 (For example)
```

Please note that the **I\_MPI\_PIN\_PROCESSOR\_LIST** variable has additional value options not mentioned here. Please refer to the Intel MPI library reference manual for more information. Older versions of Intel MPI use the deprecated **I\_MPI\_PIN\_PROCS** variable instead.

In addition, since Intel MPI and MPICH2 are binary compatible, you can use MPICH2 tuned for vSMP Foundation instead of Intel MPI.

---

## Advanced Topics and Other Considerations

### *Over-subscribing System Resources*

While it is beneficial to restrict the execution of tasks to specific CPUs, it is important not to do so naively. All factors should be taken into consideration when choosing the CPUs to assign tasks to. Common examples of factors to be taken into consideration are detailed in the following sections.

#### **CPU Resources**

It is recommended to avoid assigning tasks to CPUs that already have tasks assigned to them; it is usually better to queue jobs than to have more tasks running on the system than the number of CPUs, and you should usually avoid placing processes on specific CPUs that are already subscribed if other CPUs on the AVM are vacant.

#### **Memory resources**

A group of tasks may require the use of large amounts of memory. Since local memory access - a main benefit of COMA - is limited to the amount of memory available on a per NUMA node basis, it is better not to affine several processes using high amount of memory on the same NUMA node, even if that means leaving some of the CPUs on that node vacant. As pointed in the Overview section, every system can have more memory than the comprising NUMA nodes.

For example, an AVM comprised of 2 sub-systems, each carrying 2 dual-core processors, and 16GB of RAM, totaling 8 CPUs (cores) and 32GB<sup>1</sup> of RAM, is used to run 4 processes, each using 7GB. Optimal performance will be achieved by placing 2 tasks per NUMA node rather than placing all tasks on the same NUMA node.

#### **I/O Resources**

Some tasks require significant use of I/O, and especially disk/storage I/O. If multiple tasks needing significant storage throughput (for example, for solver scratch space) are executed in the same AVM, it is recommended to do one of the following:

1. Placed the tasks on separate nodes, and direct them to the extent possible to use drives attached to the local NUMA node. This will ensure that the drives do not become a bottleneck for executing the tasks concurrently.
2. Aggregate the maximum available drives into a single logical volume (using the Linux command **mdadm**), and directing all tasks to use that logical volume. Different drive aggregation policies (such as RAID level 0,5,6,etc.) provide different levels of performance. Please consult ScaleMP for optimal storage configuration.

---

<sup>1</sup> The amount of memory available to the Operating System running within the AVM will be less than 32GB, as vSMP Foundation and its caches make use of a portion of the available memory.

### ***Placing tasks after invocation (Lazy Placement)***

In some cases, there is no good way to control the launch/fork flow for a given task, or the functionality of the **numactl** and **taskset** commands is not comprehensive enough to achieve the best placement. In that case, the user may need to combine the use of other system commands (interactively or using a shell script) to achieve the required effect.

Consider a task spawning 4 processes, each spawning 2 threads, where the required placement is to have each process and its child threads running on the same NUMA node. In that case, a shell script can be used to implement such placement strategy by using system commands such as **ps** to obtain the current state of the application, **awk** to parse ps's output, and **taskset** to assign selected tasks to CPUs or nodes.

### ***libc Memory Management***

High mmap and munmap system call rates will cause a contention on the zone\_lock within the Linux kernel. It is recommended to increase the threshold for malloc memory allocation in order to reduce libc calls to mmap and munmap. Increasing the threshold for malloc can be done as follows:

```
export MALLOC_TOP_PAD_=20000000
```

Notes:

1. The exact syntax depends on the shell used (csh, tsch, sh, bash). The above example is for sh and bash.
2. The extra '\_' (underbar) character at the end of the variable name is required.

Since mmap and munmap are often used, it is recommended that the relevant environment variable be set for each user using the AVM. Note that the downside of increasing the threshold for malloc is a "waste" of few megabytes per process. Since most AVMs offer very large memory (in order of hundreds of GB), it is a well-worth tradeoff for the performance gain.

### ***System Shared Memory Segment Limitations***

The shared-memory segment for inter-process communications is limited by default in some Linux distributions. If a process tries to create a large shared-memory segment, an error will occur. Here is an example of an error from the MPI daemon due to shared-memory segment limitation:

```
mpid: Cannot create shared memory segment of 225255424 bytes
```

To extend the size of the shared-memory segment in Linux, please execute the command below as root:

```
echo 9999999999 > /proc/sys/kernel/shmmax
```