2 Origin Programming

For the most part, programming Origin is no different than programming traditional shared memory multiprocessors such as Power Challenge 10000. This is, of course, largely because of the hardware, which makes Origin’s physically distributed memory function as shared memory with fairly uniform access times. It is also, however, a result of the operating system, which contains some significant new capabilities designed to keep the system running as efficiently as possible. Although most of these added capabilities are transparent to the user, some new tools are available for fine-tuning performance, and there is some new terminology to go along with them.

This section describes some of the new capabilities of IRIX 6.4 (also known as Cellular IRIX), gives you an understanding of what the operating system is trying to accomplish with them, and familiarizes you with the terminology used since it will show up when you get around to investigating ways of fine-tuning application performance.

2.1 It’s Just Shared Memory

Origin is a shared memory multiprocessor. From a user’s perspective, it looks and feels just like Silicon Graphics’ other shared memory multiprocessor, Power Challenge 10000: It runs the IRIX operating system, which provides the same multiuser time-sharing system users are familiar with. The same compilers, tools, and programming models are used. I/O devices may be accessed from any processor in exactly the same way, no matter which slot or module they are installed in. Your old codes run on Origin!

The hardware, however, is different from that of Power Challenge. There is no longer a central bus into which processors, memory, and I/O devices are installed. Instead, memory and peripherals are distributed among the processors, and the processors are connected to one another via an interconnection fabric of hub and router chips. This breaks the system bottleneck imposed by a common bus and provides unprecedented scalability for a shared memory system. A side effect, though, is that access times to different memories vary according to how many hubs and routers the information must pass through. Although the hardware has been designed so that overall memory access times are greatly reduced compared to Power Challenge and the variation in them is small, they are, nevertheless, nonuniform, and this is a change from the bus-based system. As a result, IRIX has been enhanced to take the nonuniform memory access (NUMA) into account. IRIX’s new capabilities are designed to minimize the effects of running on a machine with physically distributed memory and capitalize on the scalability of the memory hierarchy.

The following sections look at the new capabilities in IRIX. For most users, this is nonessential information. The NUMA effects that IRIX works to minimize are, first of all, only important to jobs that incur a lot of cache misses, and, second, they apply mostly to highly parallel programs running on larger systems. Nevertheless, introducing the concepts and terminology has some benefits. By understanding what the operating system is trying to accomplish, you can fine-tune some of the choices the operating system makes. In addition, it will make you aware of the shared memory practices that will lead to the
2.2 Cellular IRIX and Memory Locality Management

Cellular IRIX is Silicon Graphics’ distributed software architecture for scalability and resilience. It provides a single-system image with many advanced features required by a modern distributed shared memory operating system. It combines the virtues of the data center environment and the distributed workstation environment including the following:

- Scalability to 128 CPUs and 1TB of virtual memory
- Compatibility with previous versions of IRIX
- Availability: IRIS Failsafe, RAID, integrated checkpoint-restart
- Throughput: parallel languages and tools; interactive and batch scheduling

Cellular IRIX is documented in the *Cellular IRIX(TM) 6.4 Technical Report*.

### 2.2.1 Memory Locality

On a distributed shared memory architecture such as Origin, the latency for a CPU to access main memory is function of the "distance" to the physical memory accessed and contention on the internal network. This dependence of latency on distance leads to a new resource for the operating system to manage: *memory locality*. Ideally, each memory access should be satisfied from memory on the same node board as the CPU making the access. Of course, this is not always possible; some processes may require more memory than fits on one node board, and different threads in a parallel program, running on separate nodes, may need to access the same memory location. Nevertheless, a high degree of memory locality can in general be achieved, and the operating system works to maximize memory locality.

Cellular IRIX has been designed to operate efficiently in a multiuser environment on an architecture where memory locality is an important resource. For the vast majority of cases, executing an application in the default environment will yield a large fraction of the achievable performance. Cellular IRIX obtains optimal memory locality management through the use of the following mechanisms:

- Topology-aware initial placement. The operating system attempts to allocate the memory a process uses from the same node on which the process runs. If there is not sufficient memory on the node to allow this, the remainder of the memory will be allocated from nodes that the process is closest to.
- Dynamic memory migration. Data may be periodically moved from their current location to the node making the heaviest use of them. The operating system manages memory in *pages*: contiguous blocks that have a default size of 16KB. Thus, the smallest amount of data which can...
be migrated is one page, and this capability is sometimes called page migration.

- Replication of read-only memory. The operating system makes copies of data that are often read by multiple nodes — such as application and library code — and spreads the copies throughout the system. This reduces the latency to access such data, and their distribution eliminates contention for them within the interconnection fabric.

- Scheduling processes so that there is a lot of nearby memory. When selecting a CPU on which to initiate a job, the operating system will try to choose a node that, along with its neighbors, provides an ample supply of memory. Thus, should the process’s memory requirements grow over time, the additional memory will be allocated from nearby nodes, keeping the access latency low.

- Process scheduler knows memory affinity of processes. In juggling the demands of a multiuser environment, the operating system makes every attempt to keep processes running on CPUs close to the memories in which their data reside.

- Application writers may give placement hints. Users can ensure that data are placed in memory so as to minimize access times by using compiler directives and/or the `dplace` utility.

- Selectable and customizable policy choice. The operating system supports different policies that specify how data should be placed into node memories. The user may select a policy different from the default through the use of compiler directives and/or the `dplace` utility.

The operating system supports memory locality management through a set of low-level system calls. These are not of interest to the general user as the capabilities required to fine-tune performance have been made available in a high-level tool, `dplace`, and through compiler directives. But a couple of concepts that the system calls rely on are described since terminology derived from them is used by the high-level tools. These are memory locality domains (MLDs) and policy modules (PMs).

Every user of an Origin system implicitly makes use of MLDs and policy modules since the operating system uses them to maximize memory locality. These are completely transparent to the user do not need to be understood to use an Origin system. But for those of you interested in fine-tuning application performance — particularly of large parallel jobs — it can be useful to know not only that MLDs and policy modules exist, but which types of policies are supported and what those policies do.

back to top

### 2.2.2 Memory Locality Domains (MLDs)

To understand the issues involved in memory locality management, consider the scenario presented in the following diagram:
Here is a shared memory application that we want to run on four processors of an Origin system. This particular application exhibits a relatively simple memory access pattern, namely, 90% of each process’s cache misses are from memory accesses to an almost unshared section of memory, 5% to a section of memory shared with another process, and the remaining 5% to a section of memory shared with a third process (see the diagram below):
Lack of attention paid to memory locality could end up with the situation shown below: a couple of processes running in one corner of the machine and the other processes running in an opposite corner. This results in the second and third processes incurring longer-than-optimal memory latencies to their shared section of memory.

As it turns out, this is not that bad. Since the Origin hardware has been designed to keep the variation in memory latencies relatively small, and since accesses to the shared section of memory only account for 5% of two of the processes’ cache misses, the nonoptimal placement has only a tiny effect on the performance of the program.

But there are situations in which performance could be significantly affected. If absolutely no attention is paid to memory locality, the processes and memory could end up as shown below:
Here, each process runs on a different and distant node, and the sections of memory they use have been allocated from yet a different set of distant nodes. In this case, even the accesses to unshared sections of memory --- which account for 90% of each process’s cache misses --- are nonlocal, thus, increasing the costs of accessing memory. In addition, program performance can vary from run to run depending on how close each process ends up to its most-accessed memory.

The memory locality management mechanisms in Cellular IRIX are used to avoid such situations. Ideally, the processes and memory used by this application are placed in the machine as follows:
The first two processes run on the two CPUs in one node
The other two processes run on the adjacent node’s CPUs
The memory for the first pair of processes is allocated from the first node
The memory for the second pair of processes is allocated from the second node

To allow IRIX to achieve this ideal placement, two abstractions of physical memory nodes are used:

- Memory locality domains (MLDs)
- Memory locality domain sets (MLDSETs)

A memory locality domain is a source of physical memory. It can be one node, if there is sufficient memory available for the process(es) that run there, or several nodes within a given radius of a center node. For the example application, the operating system creates two MLDs, one for each pair of processes:
It is up to the operating system to decide where in the machine these two MLDs should be placed. Since optimal performance requires that they be placed on adjacent nodes, the operating system needs some additional information.

Memory locality domain sets describe how a program’s MLDs should be placed within the machine and whether they need to be located near any particular hardware devices (e.g., graphics). The first property is known as the *topology*, and the second as *resource affinity*. Several topology choices are available. The default is to let the operating system place the MLDs on a cluster of physical nodes that is as compact as possible. Other topologies allow MLDs to be placed in hypercube configurations (which are proper subsets of Origin’s interconnection topology) and on specific physical nodes. The diagram below shows the MLDs for the example application placed in a one-dimensional hypercube topology with resource affinity for a graphics device:
2.2.3 Policy Modules

With the MLDs and MLD set defined, the operating system is almost ready to attach the program’s processes to the MLDs, but first policy modules need to be created for the MLDs. Policy modules tell the operating system the following:

- How to place pages of memory in the MLDs
- Which page size(s) to use
- What fallback policies to use if the resource limitations prevent the preferred placement and page size choices from being carried out
- Whether page migration is enabled
- Whether replication of read-only text is enabled

The operating system uses a set of default policies unless instructed to do otherwise. The user may change the defaults through the utility `dplace(1)` or via compiler directives. Once the desired policies have been chosen, the operating system then maps processes to MLDs, as shown below:
This ensures that the application threads execute on the nodes from which the memory is allocated.

---

**Data Placement Policies**

Although it may be obvious to the application writer what the memory access patterns for the application are --- and hence which MLD each section of memory should be allocated from --- there is no way for the operating system to know this. The placement policies determine how data are allocated from the different MLDs in the MLD set.

Three memory placement policies are available:

- First-touch, in which the memory holding a data structure is allocated from the MLD whose associated process has first accessed the data structure
- Fixed, in which the data structure is allocated from a specific MLD
- Round-robin, in which the pages of memory making up a data structure are allocated in a round-robin fashion from the MLDs in the MLD Set.

The default policy is first-touch. Unless you use the compiler placement directives or MP library environment variables to override it, this policy dictates how data structures are allocated from the available MLDs. Under this policy, the first processor to touch a data element causes the page of memory holding the data element to be allocated from the memory of the node on which the process is running. This memory placement policy works well for programs that have been parallelized completely, from beginning to end.

Returning to the example program above, each processor accesses five pieces of data: a 90% piece
which is accessed by no other processor, and four 5\% pieces, each of which is also accessed by a
neighboring processor. In the ideal data layout, the 90\% piece each processor accesses is stored in its
local memory. In addition, the two adjacent 5\% pieces are stored locally, while the two remaining 5\% pieces are stored in the neighboring processors. This distributes responsibility for storing the data
equally among all processors and ensures they all incur the same cost for accessing nonlocal memory.

If the initialization of the data structures is done in parallel, each processor is the first to touch its 90\%
piece and the two adjacent 5\% pieces, causing those data to be allocated locally; this is exactly where we
want these data to reside. By initializing in this way, you the programmer use the first-touch policy to
guarantee good data placement. Although this extra consideration is not required in a bus-based system
such as Power Challenge, it is generally simple to do and does not require you to learn anything new,
such as the compiler directives, which also allow data to be placed optimally.

If a program has not been completely parallelized, the first-touch policy may not be the best one to use.
For the example application, if all the data are initialized by just one of the four processes, all the data
will be allocated from a single MLD, rather than being evenly spread out among the four MLDs. This
introduces two problems:

- Accesses that should be local are now remote.
- All the processes’ cache misses are satisfied from the same memory. This may cause a
  performance bottleneck.

What can you do to remedy this?

If migration is enabled, the data eventually move to where they are most often accessed, so ultimately
they should be placed correctly. This works well for applications in which there is one optimal data
placement and the application runs long enough for the data to migrate to their optimal locations.

But in more complicated applications, in which different placements of data are needed in different
phases of the program, there may not be enough time for the operating system to move the data to where
they need to be for a particular program phase before that phase has finished and a new one has begun.
For such applications, a round-robin placement policy may be best. Under this policy, the data are
evenly distributed among the MLDs. They are not likely to be in an optimal location at any particular
point in the program, but by spreading the data out, you avoid creating performance bottlenecks.

This is actually a more important consideration than getting the data to the place where the latency is the
lowest. The variation in memory latency is not that great in Origin systems, but if all the data are stored
in the memory of one node, dozens of processors are unable to access different pieces of the data with
anywhere near the bandwidth they could achieve if the data were evenly spread out among the
processing nodes.

The final placement policy is fixed placement, which places pages of memory in a specific MLD by
using the compiler placement directives. These are a convenient way to specify the optimal placement of
data. And since you can use different placements in different parts of a program, they are ideal for more
complicated applications in which multiple data placements are required.
The initial placement of data is important for consistently achieving high performance on multiprocessor applications. It is not an issue for single-processor applications since there is only one MLD from which to allocate memory. There is, however, one difference you may see running on Origin compared with a bus-based system. Applications with modest memory requirements are likely to succeed in allocating all their memory from the node on which they run. In such cases, all cache misses see local memory latencies. But as an application’s data requirements grow, it may need to draw memory resources from nearby nodes. As a result, some cache misses have longer latencies. Thus, the effective time for a cache miss can increase as the size of the data an application uses grows or if other jobs consume more of the memory on the node.

back to top

### 2.3 Recommendations for Achieving Good Performance

The memory locality management automatically performed by IRIX means that most Origin users can achieve good performance and enhanced scalability without having to program any differently than they do on a Power Challenge. But Origin does employ a NUMA architecture, which affects the performance of some programs which were developed on a uniform memory access architecture. This section summarizes when you can expect to see artifacts of the NUMA architecture and what tools you can use to minimize them.

Even though Origin is a highly parallel system, the vast majority of programs run on it use only a single processor. Nothing new or different needs to be done to achieve good performance on such applications; tune these just as you would a single-processor program that is to be run on a Power Challenge. Section 3, "Single-Processor Tuning," provides a detailed discussion of the tools available and steps you can take to improve the performance of uniprocessor programs.

There is one new capability in the Origin system, though, that single-processor programs can sometimes take advantage of to improve performance, namely, support for multiple page sizes. Recall that page size is one of the policies that IRIX 6.4 uses in allocating memory for processes. Normally, the default page size of 16 KB is used, but for programs that incur a performance penalty owing to a large number of TLB misses, using a larger page size can be beneficial. For single-processor programs, page size is controlled via `dplace(1)`, which is explained in the single-processor tuning section.

Single-processor tuning accounts for most of the performance work you do on Origin. Don’t get lazy here: using the proper compiler flags and coding practices can yield big improvements in program performance. Once you have made your best effort at single-processor tuning, you can turn your attention to issues related to parallel programming. Parallel programs are separate into two classes:

- **MP library programs.** These use the shared memory parallel programming directives that have been available on Silicon Graphics systems for many years.
- **Non-MP library programs.** These include MPI, PVM, and other threaded programs.

For parallel programs that do not use the Silicon Graphics MP directives, it is a good idea to use the
The dplace(1) utility. This is especially true for programs linked with the 2.0 release of MPI. This version of MPI was developed on Power Challenge. As a result, no attention was paid to making sure data structures used in the library are distributed among the MPI processes that make up the program. This can result in less than ideal scaling since some of these data structures may create bottlenecks. Using dplace can help in distributing these data, which will improve scalability. Using dplace on MPI jobs is described in Section 4.5.1. Note, though, that the 3.0 release of the MPI library will be updated for Origin; once it has been released, dplace should no longer be used with MPI jobs.

The second class of parallel programs are those that use the MP library. Programs in this class that make good use of the R10000’s caches see little effect from Origin’s NUMA architecture. The reason for this is that memory performance is only affected by memory latencies if the program spends a noticeable amount of time actually accessing memory, as opposed to cache. But cache-friendly programs have very few cache misses, so the vast majority of their data accesses are satisfied by the caches, not memory. Thus, the only NUMA effect cache-friendly parallel programs will see on Origin is scalability to a larger number of processors. The perfex(1) and speedshop(1) tools can be used to determine if a program is cache-friendly; their use should be an integral part of your tuning efforts. You can find a detailed discussion of their use in the single-processor tuning section.

Not all programs can be made cache friendly, however. That’s alright, it just means they will spend more time accessing memory than their cache-friendly brethren, and in general will run at lower performance. This is just one of the facts of life for cache-based systems, which is to say, all computer systems except very expensive vector processors.

In comparing the performance of noncache-friendly parallel programs on Origin with Power Challenge, you are likely to see a marked improvement in performance on Origin since its access times to local memory are significantly faster than Power Challenge. If you observe better performance and scalability, then the operating has system succeeded in placing the data so that the majority of the memory accesses are local, and there is nothing more you need do.

On the other hand, if there is a performance or scalability problem, the data have not been optimally placed and modifying the placement policies can fix this. First, try a round-robin placement policy. For programs that have been parallelized using the Silicon Graphics MP directives, enable round-robin placement using the _DSM_ROUND_ROBIN environment variable (see Section 4.5.8.1 for details). If this solves the performance problem, you’re done. If not, try enabling migration with the _DSM_MIGRATION environment variable (see Section 4.5.8.2 for details). In addition, round-robin placement and migration can be combined. In some cases, this achieves an optimal data layout more quickly than either technique used alone.

Often, these policy changes are all that is needed to fix a performance problem. They are convenient since they require no modifications to your program. In some instances, though, they will not fix a performance or scalability problem. In those cases, you need to modify the program to ensure that the data are optimally placed. This can be done in a couple of ways. You can use the default first-touch policy and program so that the data are first accessed by the processor in whose memory they should reside. This programming style is easy to use, and it does not require learning new compiler directives. This approach is discussed in Section 4.5.9.1. The second way to ensure proper data layout is to use the data placement directives, which permit you to specify the precise data layout that is optimal for your program. The use of these directives is described in Section 4.5.9.3.