

# 4 Multiprocessor Programming

## 4.1 Introduction

The selling point of a multiprocessor is that you can apply the power of multiple CPUs to your program, so that it completes in less time. The preceding text in this document has been about making a program run faster in a single CPU. Now we start running the optimized program concurrently on multiple CPUs, and discover:

- There is a mathematical limit on how fast a real program can go
- A variety of programming models can be used to express parallelism
- Parallel processing introduces new bottlenecks, but there are additional tools to deal with them

You can write a program that expresses parallel execution in the source code, or you can let the Power Fortran or Power C compiler find the parallelism that is implicit in a conventional, serial program.

[back to top](#)

---

## 4.2 Parallel Speedup and Amdahl's Law

When your program runs on more than one CPU, its total run time should be less. But how much less? And what are the limits on the speedup?

[back to top](#)

---

### 4.2.1 Adding CPUs to Shorten Execution Time

You can make your program run faster by distributing the work it does over multiple CPUs. There is always some part of the program's logic that has to be executed serially, by a single CPU. However, suppose there is one loop where the program spends 50% of the execution time. If you can divide the iterations of this loop so that half of them are done in one CPU while the other half are done at the same time in a different CPU, the whole loop can be finished in half the time. The result: a 25% reduction in program execution time.

The following topics cover the formal mathematical treatment of these ideas (Amdahl's law). There are two basic limits to the speedup you can achieve by parallel execution:

- The fraction of the program that can be run in parallel,  $p$ , is never 100%.
- Because of hardware constraints, after a certain point, there is less and less benefit from each added CPU.

Tuning for parallel execution comes down to battling these two limits. You strive to increase the parallel fraction,  $p$ , because in some cases even a small change in  $p$  (from 0.8 to 0.85, for example) makes a

dramatic change in the effectiveness of added CPUs.

And you work to ensure that each added CPU does a full CPU's work and does not interfere with the work of other CPUs. In the Origin architectures this means

- Evenly spreading the workload among the CPUs
- Eliminating false sharing and other types of memory contention between CPUs
- Making sure that the data used by each CPU are located in a memory near that CPU's node

There are two ways to obtain the use of multiple CPUs. You can write your source code to use explicit parallelism --- showing in the source code which parts of the program are to execute asynchronously and how the parts are to coordinate with each other. Section 4.3 is a survey of the programming models you can use for this, with pointers to the online manuals.

Alternatively you can take a conventional program in C, C++, or Fortran, and apply a precompiler to it to find the parallelism that is implicit in the code. These precompilers, IRIS Power C and Power Fortran, insert the source directives for parallel execution for you. Section 4.4 surveys these tools.

back to top

---

## 4.2.2 Parallel Speedup

If half the iterations of a DO-loop are performed on one CPU, and the other half run at the same time on a second CPU, the whole DO-loop should complete in half the time. For example,

```
for (j=0; j<MAX; ++j) { z[j] = a[j]*b[j]; }
```

The IRIS Power C package can automatically distribute such a loop over  $n$  CPUs (with  $n$  decided at run time based on the available hardware), so that each CPU performs  $MAX/n$  iterations.

The speedup gained from applying  $n$  CPUs,  $Speedup(n)$ , is the ratio of the one-CPU execution time to the  $n$ -CPU parallel execution time:  $Speedup(n) = T(1)/T(n)$ . If you measure the one-CPU execution time of a program at 100 seconds, and the program runs in 60 seconds with 2 CPUs,  $Speedup(2) = 100/60 = 1.67$ .

This number captures the improvement from adding hardware to the system. We expect  $T(n)$  to be less than  $T(1)$ ; if it is not, adding CPUs has made the program slower, and something is wrong! So  $Speedup(n)$  should be a number greater than 1.0, and the greater it is, the more pleased we are. Intuitively you might hope that the speedup would be equal to the number of CPUs --- twice as many CPUs, half the time --- but this ideal can never be achieved (well, almost never).

Normally, the number  $Speedup(n)$  must be less than  $n$ , reflecting the fact that not all parts of a program benefit from parallel execution. However, it is possible --- in rare situations --- for  $Speedup(n)$  to be larger than  $n$ . This is called a *superlinear* speedup --- the program has been sped up by more than the increase of CPUs.

A superlinear speedup does not really result from parallel execution. It comes about because each CPU

is now working on a smaller set of memory. The problem data handled by any one CPU fits better in cache, so each CPU executes faster than the single CPU could do. A superlinear speedup is welcome, but it indicates that the sequential program was being held back by cache effects.

back to top

### 4.2.3 Amdahl's Law

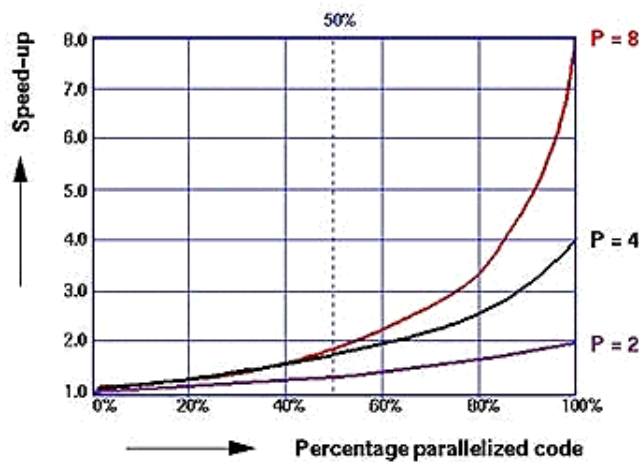
There are always parts of a program that you cannot make parallel --- code that must run serially. For example, consider the DO-loop. Some amount of code is devoted to setting up the loop, allocating the work between CPUs. Then comes the parallel part, with all CPUs running concurrently. At the end of the loop is more housekeeping that must be done serially; for example, if  $n$  does not divide MAX evenly, one CPU must execute the few iterations that are left over. The serial parts of the program cannot be speeded up by concurrency.

The mathematical statement of this idea is called Amdahl's law. Let  $p$  be the fraction of the program's code that *can* be made parallel ( $p$  is always a fraction less than 1.0.) The remaining  $(1-p)$  of the code must run serially. In practical cases,  $p$  ranges from 0.2 to 0.99.

The potential speedup for a program is proportional to  $p$  divided by the CPUs you can apply, plus the remaining serial part,  $(1-p)$ :

$$\text{Speedup}(n) = \frac{1}{(p/n) + (1-p)} \quad (\text{Amdahl's law: Speedup}(n) \text{ given } p)$$

Suppose  $p = 0.8$ ; then  $\text{Speedup}(2) = 1/(0.4+0.2) = 1.67$ , and  $\text{Speedup}(4) = 1/(0.2+0.2) = 2.5$ . The maximum possible speedup --- if you could apply an infinite number of CPUs --- would be  $1/(1-p)$ . The fraction  $p$  has a strong effect on the possible speedup, as shown in this graph:



In particular, the more CPUs you have, the more benefit you get from increasing  $p$ . Using only 4 CPUs, you need only  $p = 0.6$  to get half the ideal speedup. With 8 CPUs, you need  $p = 0.85$  to get half the ideal

speedup.

back to top

---

#### 4.2.4 Calculating the Parallel Fraction of a Program

You do not have to guess at the value of  $p$  for a given program. Measure the execution times  $T(1)$  and  $T(2)$  to calculate a measured  $\text{Speedup}(2) = T(1)/T(2)$ . The Amdahl's law equation can be rearranged to yield  $p$  when  $\text{Speedup}(n)$  is known:

$$p = \frac{2}{1} * \frac{\text{SpeedUp}(2) - 1}{\text{SpeedUp}(2)} \quad (\text{Amdahl's law: } p \text{ given Speedup}(2))$$

Suppose you measure  $T(1) = 188$  seconds and  $T(2) = 104$  seconds.

$$\begin{aligned} \text{SpeedUp}(2) &= 188/104 = 1.81 \\ p &= 2 * ((1.81-1)/1.81) = 2*(0.81/1.81) = 0.895 \end{aligned}$$

back to top

---

#### 4.2.5 Predicting Execution Time with $n$ CPUs

In some cases, the  $\text{Speedup}(2) = T(1)/T(2)$  is a value greater than 2, in other words, a superlinear speedup (described earlier). When this occurs, the formula in the preceding section returns a value of  $p$  greater than 1.0, clearly not useful. In this case you need to calculate  $p$  from two other more realistic timings, for example  $T(2)$  and  $T(3)$ . The general formula for  $p$  is:

$$p = \frac{\text{Speedup}(n) - \text{Speedup}(m)}{(1 - 1/n)*\text{Speedup}(n) - (1 - 1/m)*\text{Speedup}(m)}$$

where  $n$  and  $m$  are the two processor counts whose speedups are known. You can use this calculated value of  $p$  to extrapolate the potential speedup with higher numbers of processors. For example, what is the expected time with 4 CPUs?

$$\begin{aligned} \text{Speedup}(4) &= 1/((0.895/4)+(1-0.895)) = 3.04 \\ T(4) &= T(1)/\text{Speedup}(4) = 188/3.04 = 61.8 \end{aligned}$$

The calculation can be made routine using the computer. For example, following *awk* script, which you can copy and save as *amdahl.awk*, reads an input line containing at least two numbers that represent  $T(1), T(2), \dots$ :

```
{
  for (j=1;j<=NF;++j)
  { # save times T(n)
    t[j] = $j
  }
}
```

```

if (2==NF)
{ # use simple formula for p given T1, T2
  s2 = t[1]/t[2]
  p = 2*(s2-1)/s2
}
else
{ # use general formula on the last 2 inputs
  sm = t[1]/t[NF-1]
  sn = t[1]/t[NF]
  invm = 1/(NF-1)
  invn = 1/(NF)
  p = (sm - sn)/( sm*(1-invm) - sn*(1-invn) )
}
if (p<1)
{
  printf("\n#CPUs   SpeedUp(n)   T(n)   p=%6.3g\n",p)
  npat = "%5d   %6.3g   %8.3g\n"
  # print the actual times as given and their speedups
  printf(npat,1,1.0,t[1])
  for (j=2;j<=NF;++j)
  {
    printf(npat,j,t[1]/t[j],t[j])
  }
  # extrapolate using amdahl's law based on calculated p
  # first, for CPUs one at a time to 7
  for (j=NF+1;j<8;++j)
  {
    sj = 1/((p/j)+(1-p))
    printf(npat,j,sj,t[1]/sj)
  }
  # then 8, 16, 32, 64 and 128
  for (j=8;j<=128;j=j+j)
  {
    sj = 1/((p/j)+(1-p))
    printf(npat,j,sj,t[1]/sj)
  }
}
else
{
  printf("\np=%6.3g, hyperlinear speedup\n",p)
  printf("Enter a list of times for more more than %d CPUs\n",NF)
}
}
}

```

The output for one line of input looks like

```

% nawk -f amdahl.awk
106.0015 47.6993

p= 1.1, hyperlinear speedup
Enter a list of times for more more than 2 CPUs
106.0015 47.6993 33.5048 25.8661

#CPUs   SpeedUp(n)   T(n)   p= 0.969
  1         1         106
  2        2.22        47.7
  3        3.16        33.5
  4         4.1        25.9
  5        4.45        23.8
  6        5.19        20.4
  7         5.9         18

```

8	6.57	16.1
16	10.9	9.72
32	16.3	6.51
64	21.6	4.91
128	25.8	4.11

^D

These calculations are independent of most programming issues. They are not independent of hardware issues because Amdahl's law assumes that all CPUs are equal. At some point of increasing the number of processors, adding a CPU no longer affects run time in a linear way. For example, in the SGI Power Challenge architecture, although cache friendly codes scale closely with Amdahl's law up to the maximum number of processors, scaling of memory intensive applications slows as the system bus approaches saturation. If the bus bandwidth limit is reached, the actual speedup is less than predicted.

In Origin the situation is different and better. Some benchmarks on Origin scale very closely to Amdahl's law up to the maximum number of processors,  $n = 128$ . However, remember that there are two CPU's per node, so some applications (in particular, applications with large local memory bandwidth requirements) follow Amdahl's law on a per node basis rather than a per CPU basis. Furthermore, not all added CPUs are equal because some are farther removed from shared data and thus may have a greater latency to access that data. In general, if you can locate the data used by a CPU in the same node or a neighboring node, the difference in latencies is slight and the program speeds up in line with the prediction of Amdahl's law.

back to top

## 4.3 Explicit Models of Parallel Computation

You can use a variety of programming models to express parallel execution in your source program. For a longer discussion of these models, see the online book *Topics in IRIX Programming*.

### 4.3.1 Fortran Source with Directives

Your Fortran program can contain directives (specially formatted comments) that request parallel execution. When these directives are present in your program, the compiler works in conjunction with the MP runtime library, *libmp*, to run the program in parallel. There are three families of directives:

- The Parallel Computing Forum (PCF) directives permit you to specify general parallel execution.

Using PCF directives you can convert any block of code into a parallel region to be executed by multiple CPUs concurrently. You can have different parts of the region execute in different threads, or all threads can execute the entire region. You can specify parallel execution of the body of a DO-loop. There are directives that coordinate parallel threads, for example, to define critical sections.

- The `C$DOACROSS` directive and related directives permit you to specify parallel execution of the bodies of specified DO-loops.

Using `C$DOACROSS` you can distribute the iterations of a single DO-loop across multiple CPUs. You can control how the work is divided. For example, the CPUs can do alternate iterations, or each CPU can do a block of iterations.

- The data distribution directives such as `C$DISTRIBUTE` and the `affinity` clauses added to `C$DOACROSS` permit you to explicitly control data placement and affinity. These only have an effect when executing on Origin systems.

These directives complement `C$DOACROSS`, making it easy for you to distribute the contents of an array in different nodes so that each CPU is close to the data it uses in its iterations of a loop body.

All these directives are discussed in the *MIPSpro Fortran 77 Programmer's Guide* and the *MIPSpro Fortran 90 Programmer's Guide*, along with detailed instructions on their use. Many variations are possible in the use of these directives. The data distribution directives and the `affinity` clauses are discussed later in this document.

Your Fortran 77 or Fortran 90 program can exert direct control at run time over the MP library by calling the subroutines documented in the reference page *mp(3F)*.

[back to top](#)

---

### 4.3.2 C and C++ Source with Pragmas

Your C or C++ program can contain directives (specially formatted preprocessor lines that are also called pragmas) that request parallel execution. These pragmas are documented in detail in the *C Language Reference Manual*.

The C pragmas are similar in design to the PCF directives for Fortran. You use the pragmas to mark off a block of code as a parallel region. You can specify parallel execution of the body of a for-loop. Within a parallel region, you can mark off statements that must be executed by only one CPU at a time; this provides the equivalent of a critical section.

The data distribution directives and `affinity` clauses, which are available for Fortran, are not currently implemented for C (through version 7.1 the MIPSpro compilers). The one exception to this is the `page_place` pragma, which allows you to place ranges of data in the memory of specific processes of a parallel job. Like the Fortran data distribution directives, this only has an effect when running on an Origin system.

Your C/C++ program can exert direct control at run time over the MP library by calling the functions documented in the reference page *mp(3C)*. Environment variables described in the reference page *mp(3F)* also apply to C programs.

[back to top](#)

---

### **4.3.3 Message-Passing Models MPI and PVM**

There are two standard libraries, each designed to solve the problem of distributing a computation across not simply many CPUs but across many systems, possibly of different kinds. Both are supported on Origin servers.

The MPI (Message-Passing Interface) library is designed and standardized at Argonne National Laboratory, and is documented on the MPI home page.

The PVM (Portable Virtual Machine) library is designed and standardized at Oak Ridge National Laboratory, and is documented on the PVM home page.

The Silicon Graphics implementation of the MPI library generally offers better performance than the Silicon Graphics implementation of PVM, and MPI is the recommended library. The use of MPI and PVM programs on SGI systems is documented in the online book *MPI and PVM User's Guide*.

[back to top](#)

---

### **4.3.4 C Source Using POSIX Threads**

You can write a multithreaded program using the POSIX threads model and POSIX synchronization primitives (POSIX 1003.1b, threads, and 1003.1c, realtime facilities). The use of these libraries is documented in the online book *Topics in IRIX Programming*.

Through IRIX 6.4, the implementation of POSIX threads creates a certain number of IRIX processes and uses them to execute the pthreads. Typically the library creates fewer processes than the program creates pthreads (called an "m-on-n" implementation). You cannot control or predict which process will execute the code of any pthread at any time. When a pthread blocks, the process running it looks for another pthread to run.

[back to top](#)

---

### **4.3.5 C and C++ Source Using UNIX Processes**

You can write a multiprocess program using the IRIX *sproc(2)* function to create a share group of processes using a single address space. The use of the process model and shared memory arenas is covered in the book *Topics in IRIX Programming*.

[back to top](#)

---

## **4.4 Compiling Serial Code for Parallel Execution**

Fortran and C programs that are written in a straightforward, portable way, without explicit source directives for parallel execution, can be parallelized automatically.

To do so, you invoke a source-level analyzer that inspects your program for loops that can be parallelized. The analyzer inserts the necessary compiler directives to request parallel execution. You can insert high-level compiler directives that assist the analyzer in marking up the source code. You typically repeat the compile several times before the analyzer recognizes all the parallelism that you believe is there.

These source analyzers are named, documented, and priced as if they were separate products: Power Fortran 77, Power Fortran 90, and IRIS Power C. However, when one of them has been installed in your system, it is integrated with the compiler. You can invoke the analyzer as a separate command, or you can invoke it as a phase of the compiler: for a Fortran 77 or Fortran 90 program, the *-pfa* (power Fortran analyzer) compiler option; for a C program, the *-pca* (power C analyzer) option to the *cc* command.

The analyzer processes the source program after the macroprocessor (*cpp*), and before the compiler front-end performs syntax analysis. The analyzer examines all loops, and, when it can, it inserts compiler directives for parallel execution, just as if you had done so (see Section 4.3.1 and Section 4.3.2). The manuals that document this process are

- *IRIS Power C User's Guide* (007-0702-050)
- *MIPSpro Power Fortran 77 Programmer's Guide* (007-2363-002)
- *MIPSpro Power Fortran 90 Programmer's Guide* (007-2760-001)

You should also examine the reference pages *pfa*(1) and *pca*(1).

back to top

---

## 4.5 Tuning Parallel Code for Origin

Parallelizing a program is a big topic, worthy of a book all on its own. In fact, there are several good books on the subject. A particularly useful one is *Practical Parallel Programming* by Barr E. Bauer, Academic Press, 1992, which describes shared memory parallel programming using the directives mentioned above, as well as the automated facilities of the Power Fortran and Power C Accelerators. If you want to learn how to write parallel programs for Silicon Graphics machines, this is the book to read.

For those who prefer to do their reading on-line, in addition to the Silicon Graphics manuals referenced above, several tutorials and college courses on parallel programming are available on the web. Some of these can be found through the following links:

- *Introduction to Parallel Processing on SGI Shared Memory Computers, Course 3080, Boston University*
- *Multiprocessing by Message Passing, Course 3085, Boston University*
- *Multiprocessing with Fortran 90 (and HPF), Course 3070, Boston University*
- *Introduction to Parallel Programming, CSE 302/ ECE 392/ CS 329, University of Illinois at Urbana-Champaign*

- *Parallel Numerical Algorithms, CSE 412/CS 454/MATH 486, University of Illinois at Urbana-Champaign*
- *Applications of Parallel Computers, CS267, University of California, Berkeley*
- *Parallel Scientific Computing, Math 18.337, Massachusetts Institute of Technology.* (The course notes are available here, rather than on the newer web page.)
- *Designing and Building Parallel Programs*

Since parallel programming is such a large topic, we will not attempt to teach it in this tuning guide. Instead, we assume that you are already familiar with the basics and just concentrate on what is new for Origin.

back to top

---

### 4.5.1 Prescription for Performance

Of course, what's new about Origin is its novel Scalable Shared Memory Multi-Processor Architecture. This architecture preserves the familiar shared memory programming model of traditional bus-based SMPs but provides greater scalability. Gone, however, are the bus-based SMP's uniform memory access times. This change in architecture has a couple of implications:

1. First, you don't have to program any differently on an Origin than any other shared memory computer. Existing Power Challenge binaries run on Origin, and in most cases, they run with very good performance.
2. For the programs which don't scale as well as they should, taking account of the physically distributed memory will ensure that you regain optimum performance. This is data placement tuning, and we'll have a lot to say about it shortly.

Section 2.3 presented some recommendations for achieving good performance on Origin. These form the basis for the following prescription for tuning your programs:

1. First, tune single-processor performance.
  1. This is discussed in detail in Section 3.
  2. Multiple page sizes are available.
    1. Use *perfex(1)* to determine if TLB misses impose a significant performance penalty.
    2. If so, use *dplace(1)* or environment variables to increase page size.
2. Tuning parallel programs
  1. Has the program been properly parallelized?
    1. Is enough of the program parallelized (Amdahl's law)?
    2. Is the load well balanced?
  2. Tuning MP library programs
    1. Cache friendly programs: No data placement tuning needed.
    2. Noncache friendly programs.
      1. Is false sharing a problem?
        1. Check for false sharing using *perfex(1)* and *speedshop(1)*.
        2. If false sharing is a problem, modify data structures to fix it.
      2. Tuning Data Placement

1. First, try techniques involving no code modification.
  1. Try round-robin memory allocation.
  2. Try page migration.
  3. Try page migration and round-robin memory allocation.
2. If necessary, use techniques involving code modification.
  1. Make sure data are initialized in parallel and rely on first-touch allocation.
  2. Add regular data distribution directives (`c$distribute` and `c$page_place`).
  3. Add reshaped directives (`c$distribute_reshape`).
3. Tuning non-MP library programs (MPI, PVM, etc.): use `dplace(1)` to ensure optimal data placement.
4. Advanced tuning options (not useful in multiuser environments).
  1. For extremely memory intensive jobs, can run with one process per node.
  2. Can lock threads to processors for realtime processing.

This prescription outlines the steps you should take to improve the performance of programs running on Origin systems. Most of the performance issues that arise for Origin are the same as for Power Challenge (or other cache-based architectures); some, however, are specific to the NUMA nature of the Origin memory system.

[back to top](#)

---

#### 4.5.2 First Tune Single-Processor Performance

The majority of tuning you will perform on an Origin system is no different than on Power Challenge 10000, or any other cache-based shared memory computer. Whether your program is sequential or parallel, tuning starts with the single-processor tuning steps described in Section 3. The only thing new is that Origin's operating system, IRIX 6.4, allows multiple page sizes.

A page is the smallest contiguous block of memory that the operating system allocates to your program. Memory is divided into pages to allow the operating system to freely partition the physical memory of the system among the running processes while presenting a uniform virtual address space to all jobs. Your programs see a contiguous range of memory addresses but, in reality, the physical addresses can be all over the place.

The operating system translates the virtual addresses your programs use into the physical addresses the hardware requires. These translations are done for every memory access, so, to keep their overhead low, the 64 most recently used translations are cached in the translation lookaside buffer (TLB), which allows them to be carried out in hardware. It is only when a TLB miss occurs that the operating system needs to get directly involved in the address translation.

The page size used by the operating system is a compromise between imposing the smallest amount of system overhead and using the memory resources most efficiently. The default page size of 16 KB provides a fine enough granularity that not too much memory is wasted, yet it is large enough that each processor can usually operate efficiently within the 58 or so available TLB entries (the operating system

reserves a few entries for its own use). In some instances, though, not enough data are covered by the cached entries to allow good reuse of the TLB. In such cases, increasing the page size can often dramatically improve performance. The details of how to do this are covered in the single-processor tuning section, and the `adi2` example used there demonstrates when this type of performance issue can show up.

[back to top](#)

---

### 4.5.3 Has the Program Been Properly Parallelized?

Single-processor tuning is all that is required for the majority of programs since most jobs sequential. But parallel jobs tend to be the ones that use a lot of time and other system resources, so ensuring that they are optimized is important.

The first step in tuning a parallel program is making sure that it has been properly parallelized. This means that

1. Enough of the code needs to have been parallelized to allow the program to attain the desired speedup. The Amdahl's law extrapolations shown earlier will help in determining if this is a source of performance problems. If so, standard techniques for improving the parallelization need to be applied. See the references above if you need help with this.
2. Is the workload distributed evenly among the processors? If *speedshop(1)* is used to profile a parallel program, it will provide information on each of the parallel threads. You can use this to verify whether each thread takes about the same amount of time to carry out its pieces of the work. If not, changes, ranging from using more sophisticated scheduling types (such as cyclic, dynamic, or gss) to algorithmic redesign, may be required. Once again, you are referred to the references above for more help.

If the program has already been run with good success on another parallel system, then both issues are likely to have already been addressed. But if you are now running the program on more processors than you have previously, it is still possible to encounter problems in the parallelization which simply never showed up before. Be sure to revalidate that any limits in scalability are not due to Amdahl's law and be on the lookout for bugs in code that has not previously been stressed.

[back to top](#)

---

### 4.5.4 MP Library Programs

Although several programming models are available for creating parallel programs, the most popular approach on Silicon Graphics' shared memory multiprocessors is to make use of the MP library and associated compiler directives. The majority of this section on tuning parallel programs for Origin is devoted to the techniques you can use to tune MP library programs. Later, the Origin-specific aspects of tuning for non-MP library programs are also described.

As just seen, there are two key criteria for ensuring that a program is properly parallelized. These criteria are independent of the platform on which the program runs. In addition to these platform-independent criteria, there are other performance issues that arise for specific types of computers. In the case of Origin, issues that are common to all shared memory computers, such as cache coherency contention, of course need to be considered (and these will be discussed). But there is one new issue which also arises: data placement. Since this is something new to shared memory programmers, it is important to understand what it is and when it can and cannot cause performance problems. So what is data placement and how can it affect performance?

Although Origin is a shared memory computer, it utilizes physically distributed memory. The cost of accessing memory depends on which node the memory is located in and which node the CPU accessing it belongs to. If both CPU and memory are located in the same node, the minimum access time is achieved. Additional cost is incurred for each hub or router that a memory request must pass through. Optimal data placement is selecting which nodes the data and processes of a program must reside in to achieve the best performance.

Later, the techniques one can use to achieve optimal data placement will be discussed. But first, it is important to know when data placement is not an issue so you don't waste time trying to fix something that isn't broken. Single-process programs do not have problems with data placement. This is because the operating system runs a single-process program on the same node that its memory is allocated from, thus minimizing memory access times. Only programs that have very large memory requirements need memory to be allocated from additional nodes. But in these cases, the extra memory is obtained from the closest neighboring nodes, so once again, memory access times are minimized. Thus, data placement issues can only arise for parallel programs.

back to top

---

#### 4.5.5 Cache Friendly Programs

Data placement issues won't arise for all parallel programs: those which are cache friendly do not incur performance penalties if the data placement is not optimal. This is because such programs satisfy their memory requests primarily from cache, rather than main memory, and the time is the same for each processor to access its caches. Thus, if some memory requests take a bit longer than ideal, the net effect on the program will be negligible. So data placement can only be an issue for parallel programs which are very memory intensive and are not cache friendly. (This gives you one more incentive to apply the cache tuning techniques described earlier in the single-processor tuning section.)

You can determine how cache friendly a program is using *perfex(1)*. The command

```
% perfex -a -x -y a.out
```

will tell you how many primary and secondary cache misses the program *a.out* generates, what the cache hit ratios are, and it will estimate how much the cache misses cost. If they only account for a small percentage of the run time, then the program makes good use of the cache and its performance will not be affected by data placement.

On the other hand, if the time spent in cache misses is high, then the program is not cache friendly and

data placement could affect performance. But there are other issues common to all cache-based shared memory platforms which are more likely to be the source of performance problems, and we consider these in the next section.

[back to top](#)

---

#### 4.5.6 Non Cache Friendly Programs

If a properly parallelized MP library program does not scale as well as expected, there are several potential causes. The first thing you need to check for is whether false sharing or other forms of cache coherency contention are a problem.

Cache coherency contention can arise in any cache-based multiprocessor system. It can only be an issue, however, for data that are frequently updated or written. Data that are mostly read and rarely written do not cause cache coherency contention.

The mechanism used in Origin to maintain cache coherence was described earlier. When one CPU modifies a cache line, any other CPU that has a copy of the same cache line is notified, discards its copy, and fetches a new copy when it needs those data again. This can cause performance problems in two cases:

1. If one CPU repeatedly updates a cache line that other CPUs use for input, all the reading CPUs are forced to frequently retrieve a new copy of that cache line from memory. This slows all the reading CPUs.
2. If two or more CPUs repeatedly update the same cache line, they contend for the exclusive ownership of the cache line. Each CPU has to get ownership and fetch a new copy of the cache line before it can perform its update. This forces the updating CPUs to execute serially, as well as making all CPUs fetch a new copy of the cache line on every use.

The first of these is generic cache coherency contention. The latter is referred to as *false sharing*. Fortunately, the tools available for Origin can help you identify these problems. In both cases, *perfex(1)* will reveal a high number of cache invalidation events (cf. hardware counters 31, 12, 13, 28, and 29 --- refer to the description of *perfex* for details of using the hardware counters). In addition, the CPU(s) repeatedly updating the same cache line shows a high count of stores to shared cache lines (counter 31). Some examples of how these problems occur and what you can do to fix them are now presented.

[back to top](#)

---

##### 4.5.6.1 False Sharing

False sharing is best demonstrated with an example. Consider the following code:

```
integer m, n, i, j
real    a(m,n), s(m)
```

```

c$doacross local(i,j), shared(s,a)
  do i = 1, m
    s(i) = 0.0
    do j = 1, n
      s(i) = s(i) + a(i,j)
    enddo
  enddo

```

This code calculates the sums of the rows of a matrix. For simplicity, assume  $m = 4$  and that the code is run on up to four processors. What you observe is that the time for the parallel runs is *longer* than when just one processor is used. To understand what causes this, let's look at what happens when this loop is run in parallel.

The following is a timeline of the operations which are carried out (more or less) simultaneously:

t = 0	t = 1	t = 2	...
s(1) = 0.0	s(1) = s(1) + a(1,1)	s(1) = s(1) + a(1,2)	...
s(2) = 0.0	s(2) = s(2) + a(2,1)	s(2) = s(2) + a(2,2)	...
s(3) = 0.0	s(3) = s(3) + a(3,1)	s(3) = s(3) + a(3,2)	...
s(4) = 0.0	s(4) = s(4) + a(4,1)	s(4) = s(4) + a(4,2)	...

At each stage of the calculation, all four processors attempt to concurrently update one element of the sum array,  $s(m)$ . For a processor to update one element of  $s$ , it needs to gain exclusive access to the cache line holding the element it wishes to update. But since  $s$  is only four words in size, it is likely that  $s$  is contained entirely in a single cache line, so each processor needs exclusive access to *all* of  $s$ . Thus, only one processor at a time can update an element of  $s$ . Instead of operating in parallel, the calculation is serialized.

Actually, it's a bit worse than just serialized. For a processor to gain exclusive access to a cache line, it first needs to invalidate any cached copies of  $s$  which may reside in the other processors. Then it needs to read a fresh copy of the cache line from memory since the invalidations will have caused data in some other processor's cache to be written back to main memory. So, whereas in a sequential version, the element of  $s$  being updated can be kept in a register, in the parallel version, false sharing forces the value to continually be reread from memory, in addition to serializing the updates.

This serialization is purely a result of the unfortunate accident that the different elements of  $s$  ended up in the same cache line. If each element were in a separate cache line, each processor could keep a copy of the appropriate line in its cache, and the calculations could be done perfectly in parallel. This, then, shows how to fix the problem. All you need do is spread out the elements of  $s$  so that each  $i$ s in its own cache line. Here is a simple way to do this:

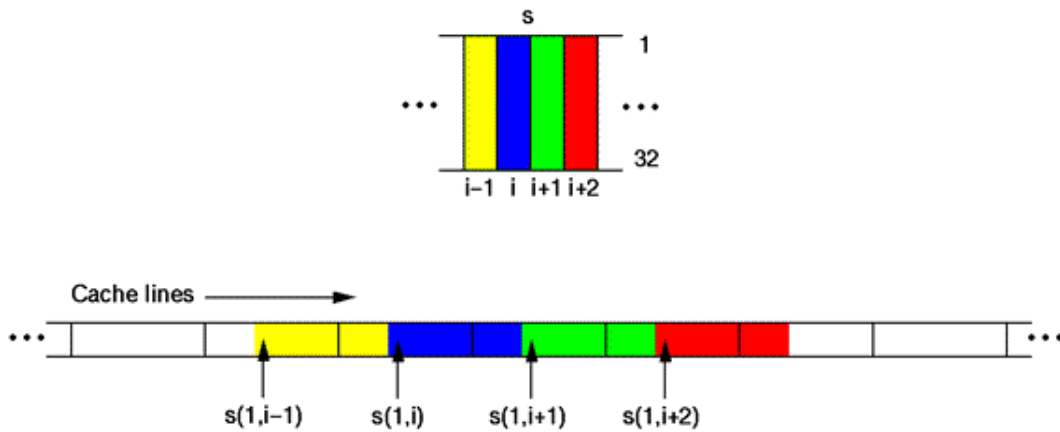
```

integer m, n, i, j
real    a(m,n), s(32,m)

c$doacross local(i,j), shared(s,a)
  do i = 1, m
    s(1,i) = 0.0
    do j = 1, n
      s(1,i) = s(1,i) + a(i,j)
    enddo
  enddo

```

Convert  $s$  to a two-dimensional array with the first dimension one (secondary) cache line in size.



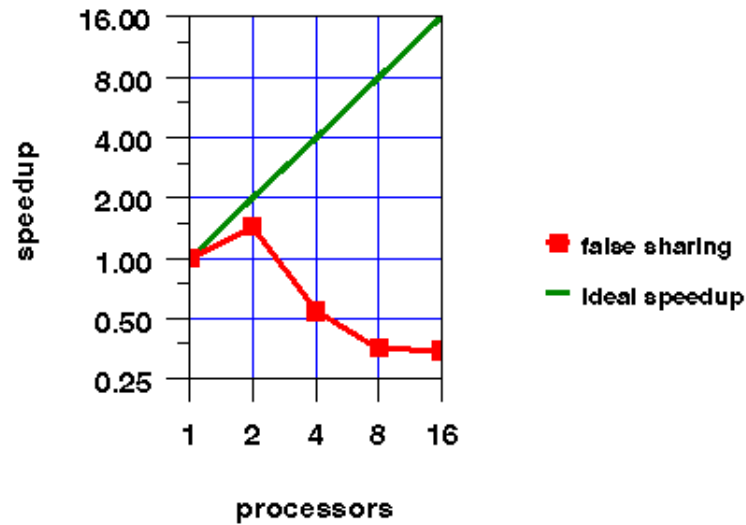
Thus, the elements  $s(1, 1)$ ,  $s(1, 2)$ ,  $s(1, 3)$  and  $s(1, 4)$  are guaranteed to be in their own, separate cache lines. Implemented this way, the code achieves perfect parallel speedup.

Note that false sharing is not an Origin-specific problem. This occurs in all cached-based, shared memory systems. Whenever an apparently properly parallelized MP library program fails to achieve the expected scaling, you need to identify whether false sharing is the cause.

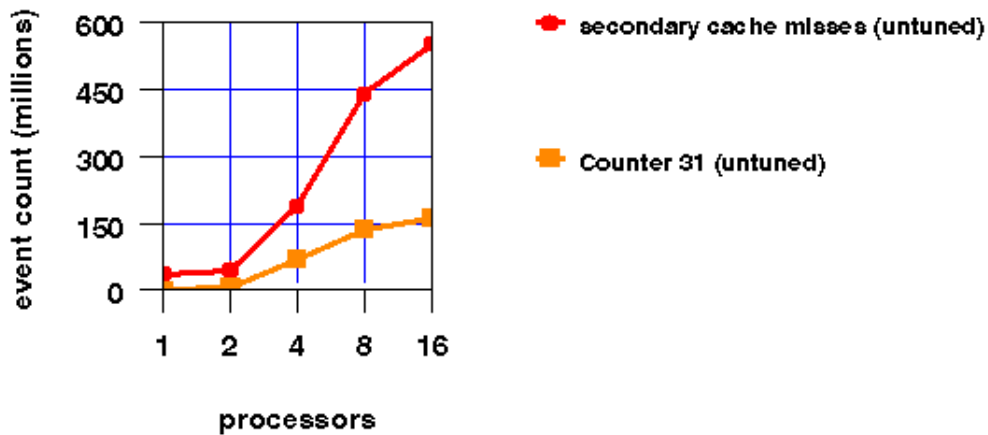
There are several possible sources of poor scaling and they can generally be distinguished by examining the event counts returned by *perfex(1)*:

- Load imbalance: is each thread doing the same amount of work? Typically, you can check this by seeing if all threads issue a comparable number of floating point operations (event 21).
- Excessive synchronization cost: are counts of store conditionals high (event 4)?
- False sharing: are counts of store exclusive to shared block high (event 31)?

To see how this works for a real code, we take an example from the Supercomputing '96 paper *Performance Analysis Using the MIPS R10000 Performance Counters*. In this example (which was run on a Power Challenge 10000 rather than an Origin), a weather modeling program shows poor parallel scaling (the red curve below) after an initial parallelization:



The counts returned from `perfex -a -x -y` reveal the following: The number of secondary data cache misses (event 26) increases as the number of processors is increased, as does the number of stores exclusive to a shared block (event 31). These counts are shown below as the red and orange curves, respectively. The large number of stores exclusive to a shared block, increasing with the secondary cache misses, indicates a likely problem with false sharing (the large number of secondary cache misses are a problem as well).



Next, to locate where the source of the problem, use `speedshop(1)`. There are several events which can be profiled to determine where false sharing occurs. The natural one to use is event 31, "store/prefetch exclusive to shared block in scache." Since there is no explicitly named experiment type for this event, profiling it requires setting the following environment variables:

```
% setenv _SPEEDSHOP_HWC_COUNTER_NUMBER 31
```

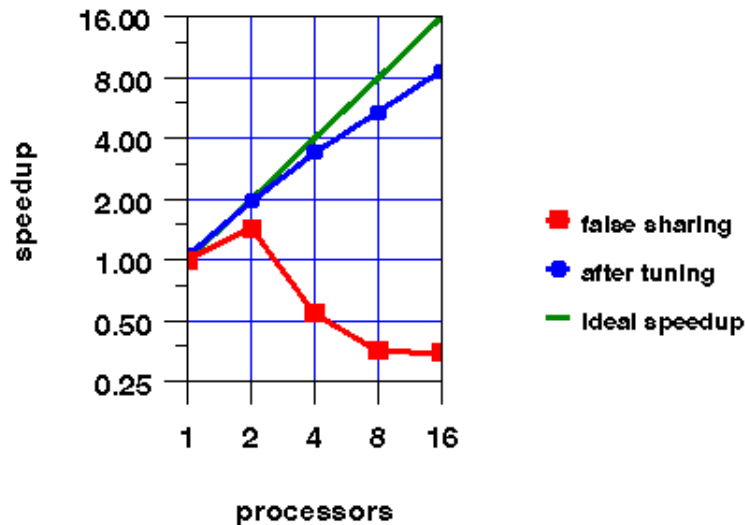
```
% setenv _SPEEDSHOP_HWC_COUNTER_OVERFLOW 99 # vary according to how
                                           # often you are willing to
                                           # generate counter
                                           # interrupts

% ssrun -exp prof_hwc a.out
```

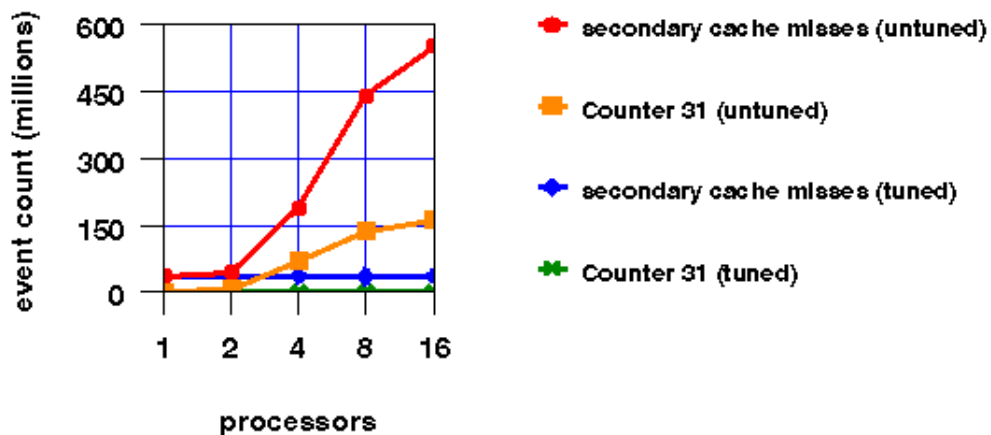
You can also profile interventions (event 12) or invalidations (event 13), but using event 31 is better since it shows where the source of the problem is in the program (some thread asking for exclusive access to a cache line) rather than the place a thread happens to be when an invalidation or intervention arrives.

Another event that can be profiled is secondary cache misses (event 26). For this event, you use the `dsc_hwc` experiment type and don't need to set any environment variables. Due to the invalidations generated by false sharing, each processor needs to read values from memory rather than its cache. So the parallel program incurs additional secondary cache misses compared with a sequential run of the program (for which there is no false sharing). As long as the program does not have a lot of other secondary cache misses, tracking them with speedshop will show where the false sharing occurs. For this program, profiling secondary cache misses was sufficient to locate the source of the false sharing.

*Speedshop* showed that the majority of secondary cache misses occurred in an accumulation step, similar to the row summation example above. Padding was used to move each processor's accumulation variable into its own cache line. After doing this, the performance improved dramatically as can be seen by the blue speedup curve in the following plot



and the blue and green curves below, which show, respectively, the counts of secondary cache misses and stores exclusive to a shared block for the tuned version of the code:



back to top

---

#### 4.5.6.2 General Recommendations for Correcting Cache Coherency Contention

As was seen in the above discussion of false sharing, you often deal with cache coherency contention by changing the layout of data in the source program. Sometimes you may have to make algorithmic changes as well. If profiling indicates that there is cache coherency contention, examine the parallel regions identified; any assignment to memory in the parallel regions is a possible source of the contention. You need to determine if that variable, or the data adjacent to it, is used by other CPUs at the same time. If so, the assignment forces the other CPUs to read a fresh copy of the cache line, and this is a source of coherency contention.

To deal with cache coherency contention, you have following general strategies:

1. Minimize the number of variables that are accessed by more than one CPU.
2. Segregate non-volatile (rarely updated) data items into cache lines different from volatile (frequently updated) items.
3. Isolate unrelated volatile items into separate cache lines to eliminate false sharing.
4. When volatile items are updated together, group them into single cache lines.

These can be applied to fix the coherency contention in the example situations below:

An update of one word (i.e., a 4-byte quantity) invalidates all the 31 other words in the same (secondary) cache line. When the other words are not related to the new data, false sharing results, as was described in the previous section. Use strategy 3 to eliminate the false sharing.

If your program has a block of global status variables that is visible to all parallel threads, you need to be careful. In the normal course of running the program, every CPU will cache a copy of most or all of this common area. Shared, read-only access does no harm. But if items in the block are volatile (frequently updated), those cache lines are invalidated often. For example, a global status area might contain the

anchor for a LIFO queue. Each time a thread puts or takes an item from the queue, it updates the queue head, invalidating a cache line.

It is inevitable that a queue anchor field will be frequently invalidated. The time cost, however, can be isolated to queue access by applying strategy 2. Allocate the queue anchor in separate memory from the global status area. Put a pointer to it (a non-volatile item) in the global status block. Now the cost of fetching the queue anchor is born only by CPUs that access the queue. If there are other items that are updated with the queue anchor --- such as a lock that controls exclusive access to the queue --- place them adjacent to the queue, aligned so that all are in the same cache line (strategy 4). However, if there are two queues that are updated at unrelated times, place each in its own cache line (strategy 3).

Synchronization objects such as locks, semaphores, and message queues are global variables and must be updated by each CPU that uses them. You may as well assume that these objects are always accessed at memory speeds, as opposed to cache speeds. You can do two things to reduce contention:

- Minimize contention for locks and semaphores through algorithmic design. In particular, use more, rather than fewer, semaphores, and make each one stand for the smallest resource possible to minimize the contention for any one resource. (Of course, this makes it more difficult to avoid deadlocks.)
- Never place unrelated synchronization objects in the same cache line. A lock or semaphore can be in the same cache line as the data that it controls since an update of one usually follows an update of the other. But unrelated locks or semaphores should be in different cache lines.

When you make a loop run in parallel using `C$DOACROSS`, try to ensure that each CPU operates on its own distinct sections of the input and output arrays. Sometimes this falls out naturally, but there are also compiler directives for just this purpose. These will be described later.

Carefully review the design of data collections that are used by parallel code. For example, the root and the first few branches of a binary tree are likely to be visited by every CPU that searches that tree, and they will be cached by every CPU. However, elements at higher levels in the tree may be visited by only a few CPUs. One option is to pre-build the top levels of the tree so that these levels never have to be updated once the program starts. Also, before you implement a balanced-tree algorithm, consider that tree-balancing can propagate modifications all the way to the root. It might be better to cut off balancing at a certain level and never disturb the lower levels of the tree. (Similar arguments apply to B-trees and other branching structures: the "thick" parts of the tree are widely cached, while the twigs are less so.)

Other classic data structures can cause memory contention, and algorithmic changes are needed to cure it:

- The two basic operations on a heap (also called a priority queue) are "get the top item" and "insert a new item." Each operation ripples a change from end to end of the heap-array. The same operations on a linked list are read-only at all nodes except for the one node that is directly affected.
- A hash table can be implemented compactly, with only a word or two in each entry. But that creates false sharing by putting several table entries (which are logically unrelated by definition) into the same cache line. Avoid false sharing: make each hash table entry a full 128 bytes, cache-aligned. Take advantage of the extra space to store a list of overflow hits in each entry. Such a list can be quickly scanned because the entire cache line is fetched as one operation.

back to top

---

### 4.5.7 Scalability and Data Placement

Once you are sure that false sharing and other forms of cache coherency contention are not a problem, if a memory intensive MP library program still exhibits scaling which is less than expected on Origin, data placement needs to be considered.

Optimizing data placement is a new performance issue for a shared memory computer. Since the cost of accessing all memory in a bus-based machine is uniform, programs written for traditional shared memory computers have no notion of data placement. To produce the best performance for these existing programs, you would like Origin's operating system to automatically ensure that all programs achieve, if not an optimal, then at least a good, data placement.

So how does IRIX try to optimize data placement? There are two things it wants to accomplish. It wants to optimize:

1. The program's topology: That is, processes making up the parallel program should be run on nearby nodes to minimize access costs for data they share.
2. The page placement: The memory required for the data a processor accesses (most) should be allocated from its own node.

Accomplishing these two tasks automatically for all programs is virtually impossible. The operating system simply doesn't have enough information to do a perfect job. So it does the best it can to approximate an optimal solution. In the earlier discussion of Cellular IRIX memory locality management, the policies and topology choices the operating system uses to try to optimize data placement were described: MLDsets arranged in cluster or cube configurations are used to achieve a well-performing topology; first-touch, round-robin and fixed page placement policies are available to try to get the initial page placements correct, while dynamic page migration can be used to correct placement issues which arise as a program runs. This technology produces good results for a great number of programs, but not for all programs. Now it's time to discuss what you can do to tune data placement should the operating system's efforts prove insufficient.

Data placement is tuned by specifying the appropriate policies and topologies, and if need be, programming with them in mind. They are specified using the utility *dplace(1)*, environment variables understood by the MP library, and compiler directives. Use of the environment variables, the programming techniques that ensure the data placement you want, and the compiler directives will be covered next. Data placement tuning using *dplace* will follow.

back to top

---

### 4.5.8 Tuning Data Placement for MP Library Programs

Unlike false sharing, there is no profiling technique that will tell you definitively that poor data

placement is hurting the performance of your program. This is a conclusion you have to arrive at after eliminating the other possibilities. Fortunately, though, once you suspect a data placement problem, many of the tuning options are very easy to perform, and you can often allay your suspicions or solve the problem with a few simple experiments.

The techniques for tuning data placement can be separated into two classes:

1. Use MP library environment variables to adjust the operating system's default data placement policies. This is simple to do, involves no modifications to your program, and is all you will have to do to solve many data placement problems.
2. Modify the program to ensure an optimal data placement. This approach requires more effort, so only try this if the first approach is not sufficient, or if you are developing a new application. The amount of effort in this approach ranges from simple things such as making sure that the program's data initializations --- as well as its calculations --- are parallelized, to adding some data distribution compiler directives, to modifying algorithms as would be done for purely distributed memory architectures.

The first approach is easiest and is covered first.

[back to top](#)

---

#### **4.5.8.1 First Try Round-Robin Placement**

The earlier discussion of Cellular IRIX memory locality management described the data placement policies used by the operating system. The default policy is called *first-touch*. Under this policy, the process which first touches (i.e., writes to or reads from) a page of memory causes that page to be allocated from the node on which the process is running. This policy works splendidly for sequential programs and many parallel programs as well. For example, this is just what you want for message-passing programs run on Origin. In such programs each process has its own separate data space. Except for messages sent between the processes, all accesses a process makes are to memory which should be local. Since each process initializes its own data, the memory required by those data will be allocated from the node the process is running on, thus making the accesses local.

But for some parallel programs, there can be unintended side effects to the first-touch policy. As an example, consider a program parallelized using the MP library. In parallelizing such a program, the programmer generally only worries about parallelizing those parts of the program that take the most amount of time. Often, the data initializations don't cost much, so they don't get parallelized. Under the first-touch policy, this means that all the data end up being allocated from the node running the master thread since it is the one which does the sequential initialization of those data. With all the data concentrated in one node (or within a small radius thereof), a bottleneck is created: all data accesses are satisfied by one hub, and this limits the memory bandwidth available to the program. If the program is only run on a few processors, you may not notice this bottleneck. But it will certainly prevent memory intensive programs from scaling to dozens of processors.

So how does one know if such a bottleneck is limiting the scalability of a program? One easy way to test this is to try other memory allocation policies. The first one you should try is *round-robin* allocation.

Under this policy, data are allocated in a round-robin fashion from all the nodes the program runs on. Thus, even if the data are initialized sequentially, the memory holding them will not be allocated from a single node; it will be evenly spread out among all the nodes running the program. This may not place the data in their optimal locations --- that is, the access times are not likely to be minimized --- but there will be no bottlenecks and scalability will not be limited.

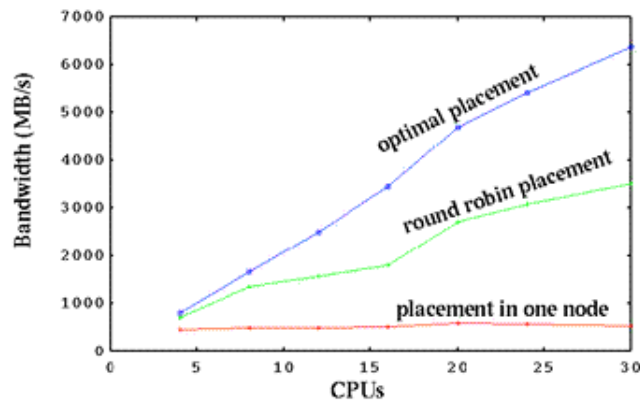
One enables round-robin data placement for an MP library program with the following environment variable setting:

```
% setenv _DSM_ROUND_ROBIN
```

The performance improvement this can make is demonstrated with an example. The plot below shows the performance achieved for three parallelized runs of the double-precision vector operation

$$a(i) = b(i) + q*c(i)$$

The bandwidth calculated assumes 24 bytes are moved for each vector element.



The red curve ("placement in one node") shows the performance you attain using a first-touch placement policy and a sequential initialization so that all the data end up in the memory of one node. The memory bottleneck this creates severely limits the scaling. The green curve ("round-robin placement") shows what happens when the data placement policy is changed to round-robin. The bottleneck disappears and the performance now scales with the number of processors. The performance isn't ideal --- the blue line ("optimal placement") shows what you measure if the data are placed optimally (which can be accomplished simply by initializing the data in parallel) --- but by spreading the data around, the round-robin policy makes the performance respectable.

back to top

---

#### 4.5.8.2 Try Enabling Migration

If a round-robin data placement solves the scaling problem that led you to tune the data placement, then you're done! But if the performance is still not up to your expectations, there are a couple of other

experiments you can try. The next one to perform is enabling page migration.

The IRIX automatic page migration facility is disabled by default because page migration is an expensive operation that impacts all CPUs, not just the ones used by the program whose data are being moved. You can enable dynamic page migration for a specific MP library program by setting the environment variable `_DSM_MIGRATION`. You can set it to either `ON` or `ALL_ON`:

```
% setenv _DSM_MIGRATION ON
% setenv _DSM_MIGRATION ALL_ON
```

When set to `ALL_ON`, *all* program data will be subject to migration. When set to `ON`, only those data structures which have *not* been explicitly placed via the compiler's data distribution directives (to be described shortly) will be migrated.

Enabling migration is beneficial when a poor initial placement of the data is responsible for limited performance. Note, though, that the program should run for at least a few seconds since the operating system needs some time to move the data to the best layout and they need to spend some time there to amortize the cost of migrating.

In addition to the MP library environment variables, migration can also be enabled as follows:

- For non-MP library programs, run the program using *dplace* with the *-migration* option. This will be described later.
- The system administrator can temporarily enable page migration for all programs using the *sn(1)* command or can enable it permanently by using *systune(1M)* to set the `numa_migr_base_enabled` system parameter. (For more information, see the comments in the system file `/var/sysgen/mtune/numa.`)

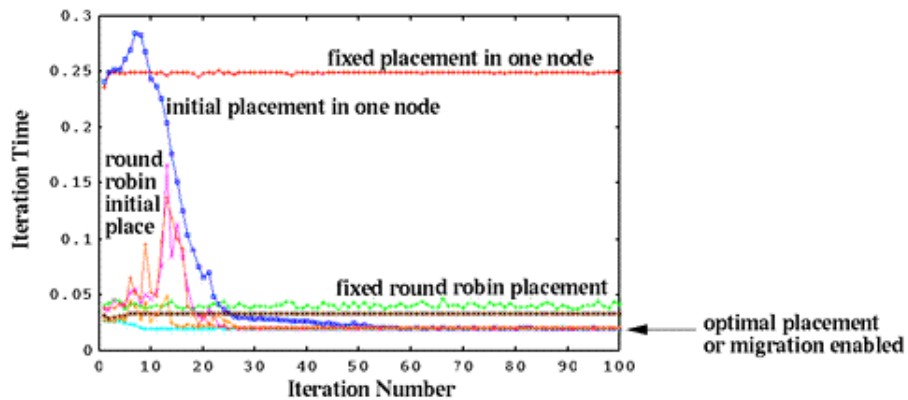
back to top

---

### 4.5.8.3 Combining Migration and Round-Robin Placement

You can also combine a round-robin initial placement with migration to try to reduce the cost of migrating from a poor initial location to the optimal one. In general, you won't know whether round-robin, migration, or both together will produce the best results without trying the different combinations, so tuning the data placement requires a bit of experimentation. Fortunately, the environment variables make these experiments easy to perform.

The diagram below shows the results of several experiments using the vector example from above. In addition to combinations of the round-robin policy and migration, the effect of serial and parallel initializations are shown. (If you do these experiments on your own existing code, you'll only need to do a few of these combinations since you won't be changing the initialization: after all, the reason you do these experiments is to find an easy way to fix a poor initial data placement!) For the results presented in the diagram, the vector operation was iterated 100 times and the time per iteration was plotted to see the performance improvement as the data move to a near optimal layout:

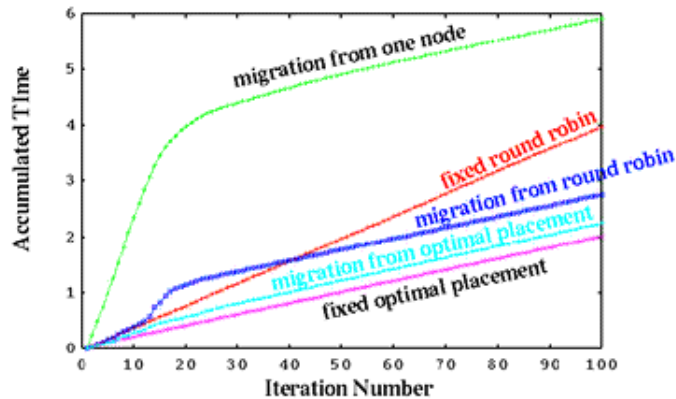


The red curve at the top ("fixed placement in one node") shows the performance for the poor initial placement caused by a sequential data initialization with a first-touch policy. No migration is used, so this performance is constant over all iterations. The green line near the bottom ("fixed round-robin placement") shows that a round-robin initial placement fixes most of the problems with the serial initialization: the time per iteration is five times faster than with the first-touch policy. Ultimately, though, the performance is still a factor of two slower than if an optimal data placement had been used. Once again, since migration is not enabled, the performance is constant over all iterations.

The flat curve just below the green curve shows the performance achieved when a parallel initialization is used in conjunction with a round-robin policy. Its per-iteration time is constant and nearly the same as when round-robin is used with a sequential initialization. This is to be expected: the round-robin policy spreads the data evenly among the processors, so it doesn't really matter how the data are initialized.

The remaining five curves all asymptote to the optimal time per iteration. Four of these eventually achieve the optimal time due to the use of migration. The turquoise curve uses a parallel initialization with a first-touch policy to achieve the optimal data placement from the outset. The orange curve just above it starts out the same, but for it, migration is enabled. The result of using migration on perfectly laid out data is to scramble them around a bit before finally settling down on the ideal time. Above this curve are a magenta and another orange curve ("round-robin initial placement"). They show the effect of combining migration and a round-robin policy. The only difference is that the magenta curve used a serial initialization while the orange curve used a parallel initialization. For these runs, the serial initialization took a little longer to reach the steady-state performance, but you should not conclude that this is a general property; when a round-robin policy is used, how the data are initialized doesn't really matter.

Finally, the blue curve ("initial placement in one node") shows how migration improves a poor initial placement in which all the data began in one node. It takes longer to migrate to the optimal placement than the other cases, indicating that by combining a round-robin policy with migration, one can do better than by using just one of the remedies alone. Dramatic evidence of this is shown by plotting the cumulative program run time:



These results show clearly that, for this program, migration eventually gets the data to their optimal location and that migrating from an initial round-robin placement is faster than migrating from an initial placement in which all the data are on one node. Note, though, that migration does take time: if only a small number of iterations are going to be performed, it is better to use a round-robin policy *without* migration than with it.

back to top

#### 4.5.8.4 Experimenting with Migration Levels

The above results were generated using the environment variable setting:

```
% setenv _DSM_MIGRATION ON
```

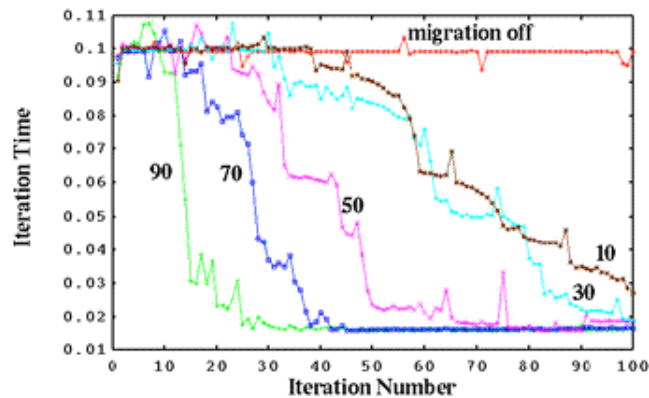
(This program has no data distribution directives in it, so in this case, there is no difference between a setting of ON and one of ALL\_ON.) But you can also adjust the migration level, that is, control how aggressively the operating system migrates data from one memory to another.

The migration level is controlled with the following environment variable setting:

```
% setenv _DSM_MIGRATION_LEVEL L
```

A high level means aggressive migration, and a low level means nonaggressive migration. A level of 0 means no migration.

The migration level can be used to experiment with how aggressively migration is performed. The plot below shows the improvement in time per iteration for various migration levels as memory is migrated from an initial placement in one node. As you would expect, with migration enabled, the data will eventually end up in an optimal location and the more aggressive migration is, the faster the pages get to where they should be.



The best level for a particular program will depend on how much time is available to move the data to their optimal location. A low migration level will do just fine for a long-running program with a poor initial data placement. But if a program can only afford a small amount of time for the data to redistribute, a more aggressive migration level will be needed. Keep in mind, though, that migration has an impact on system-wide performance and the more aggressive the migration, the greater the impact. For most programs the default migration level should be sufficient.

back to top

#### 4.5.8.5 Tuning Data Placement without Code Modification

While your results may vary, in general, it is recommend that you try the following experiments when you need to tune data placement:

- First see if using a round-robin policy fixes the scaling problem. If so, you need not try other experiments.
- Next try migration. If migration achieves about the same performance as round-robin, round-robin is to be preferred since it doesn't require operating system intervention and so will have no impact on other users of the system.
- If migration alone achieves a better performance than round-robin alone, it is worthwhile to see if the combination can do better still. As seen above, this combination might not improve the steady-state performance of your program, but it might get it to the steady-state performance faster.

Note that these data placement tuning experiments only apply to MP library programs:

- Sequential programs aren't affected by data placement.
- First-touch is the right choice for message passing programs.
- These environment variables are only understood by the MP library (for the other environment variables, directives and functions understood by *libmp*, see *mp(3)*).

Note also that you should only expect to see improvements for memory intensive programs which are

not cache friendly. If a program makes good use of the cache, data placement will not be an issue.

[back to top](#)

---

## 4.5.9 Modifying the Code to Tune Data Placement

The easiest approach to tuning data placement is to use MP library environment variables to adjust the policies the operating system uses to place data. Many times this is sufficient to solve any scaling problems you may encounter. In some cases, though, performance problems persist; if this occurs, you will need to tune the data placement via code modifications.

There are three levels of code modification that can be used:

1. Rely on the default first-touch policy and program with it in mind. This programming style is easy to use: often, fixing the data placement problems in a program is as simple as making sure all the data are initialized in parallel so that they start off in the memory of the proper processor.
2. Add regular data distribution directives (`c$distribute` and `c$page_place`). These directives allow you to specify how individual data structures should be distributed among the memory of the nodes running the program, subject to the constraint that the granularity of the data distribution is one page. These directives provide functionality similar to the data distribution directives in High Performance Fortran.
3. Add reshaped directives (e.g., `c$distribute_reshape`). Like the regular data distribution directives, these allow you to specify how individual data structures should be distributed among the memory of the nodes running the program, but they are *not* subject to single-page granularity constraints. What you lose by relaxing the single-page granularity restriction, however, is that data structures are not guaranteed to be contiguous. That is, they can have holes in them, and if you do not program properly, this can break your program. The standard data distribution directives will *not* alter the correctness of a program.

Each of these three approaches makes data placement something the programmer considers explicitly. This is a new concept for shared memory programming; it is something that simply doesn't apply to traditional bus-based multiprocessors. However, the concept has been used for years in programming distributed memory computers, so if you have written a message-passing program, you understand it well. But the fact that Origin really does employ shared memory makes writing these data-placement-aware programs easier than for nonshared memory computers.

Each of these three approaches will now be discussed.

[back to top](#)

---

### 4.5.9.1 Programming for the Default First-Touch Policy

If there is a single placement of data that is optimal for your program, you can use the default first-touch policy to cause each processor's share of the data to be allocated from memory local to its node. As a

simple example, consider parallelizing the following vector operation:

```
integer i, j, n, niters
parameter (n = 8*1024*1024, niters = 100)
real a(n), b(n), q
```

c initialization

```
do i = 1, n
  a(i) = 1.0 - 0.5*i
  b(i) = -10.0 + 0.01*(i*i)
enddo
```

c real work

```
do it = 1, niters
  q = 0.01*it
  do i = 1, n
    a(i) = a(i) + q*b(i)
  enddo
enddo
```

This vector operation is "embarrassingly parallel," so the work can be divided among the processors of a shared memory computer any way you would like. For example, if  $p$  is the number of processors, the first processor can carry out the first  $n/p$  iterations, the second processor the next  $n/p$  iterations, etc. (This is called a *simple* schedule type.) Alternatively, each thread can perform one of the first  $p$  iterations, then one of the next  $p$  iterations and so on. (This is called an *interleaved* schedule type.) But for cache-based machines, not all divisions of work produce the same performance. The reason for this is that if a processor accesses the element  $a(i)$ , the entire cache line containing  $a(i)$  is moved into its cache. If the same processor works on the rest of the elements in the cache line, they will not have to be moved into cache since they are already there. But if different processors work on the other elements, the cache line will have to be loaded into some processor's cache for each element. Even worse, false sharing is likely to occur. Thus, the performance is best for work allocations in which one processor is responsible for each element of the same cache line.

This brings up a subtle point involving the parallelization. In programming a typical distributed memory computer, you use *data distribution* to parallelize a program. That is, since a processor can only operate on data which are stored in its local memory, you break the arrays into pieces and store each piece in the memory of a single processor. This partitioning of the data then dictates what work a processor does: namely, it does the work that involves the data stored locally. Contrast this now with a shared memory computer. In a shared memory parallelization, you distribute *work* (e.g., iterations of a loop) to the processors, not data. Since the memory is shared, there really is no notion of data distribution: all data are equally accessible by all processors.

In practice, however, this distinction is generally lost since it is usually easy to find a correspondence between the work and the data. For example, in the vector operation above, the processor that carries out iteration  $i$  is the only one which touches the data elements  $a(i)$  and  $b(i)$ , so these data have effectively been distributed to that processor. The distinction is further blurred by the presence of caches, since, to achieve the best performance, the programmer needs to account for which cache lines are touched in carrying out a particular piece of work. Thus, even though the shared memory parallelization directives distribute work, programmers often think in terms of data distribution.

This now comes in handy when programming Origin. You can use the association of work with data to cause the data a processor works on to be allocated from its local memory. Here is the above vector operation parallelized for Origin:

```

integer i, j, n, niters
parameter (n = 8*1024*1024, niters = 1000)
real a(n), b(n), q

c initialization

c$doacross local(i), shared(a,b)
  do i = 1, n
    a(i) = 1.0 - 0.5*i
    b(i) = -10.0 + 0.01*(i*i)
  enddo

c real work

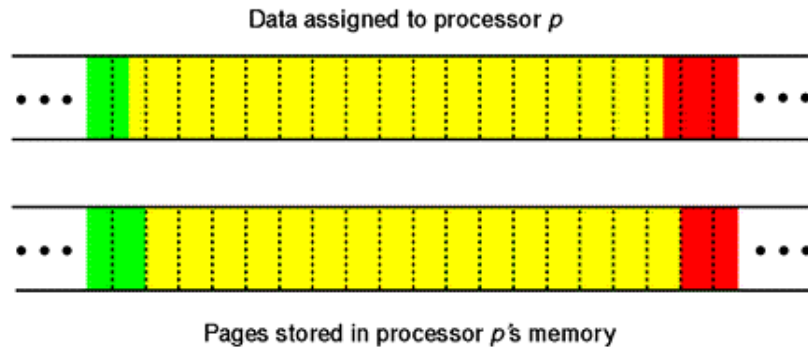
  do it = 1, niters
    q = 0.01*it
c$doacross local(i), shared(a,b,q)
  do i = 1, n
    a(i) = a(i) + q*b(i)
  enddo
enddo

```

It is exactly the same shared memory parallelization that is used on a bus-based machine! Note that since the schedule type is not specified, it defaults to simple: i.e., process 0 performs iterations 1 to  $n/p$ , process 1 performs iterations  $n/p+1$  to  $2*n/p$ , and so on. Use of the simple schedule type is important; more about that shortly. Since the initialization takes a small amount of time compared with the "real work," parallelizing it doesn't reduce the sequential portion of this code by much, so some programmers wouldn't bother to parallelize the first loop for a traditional shared memory computer. However, if you are relying on the first-touch policy to ensure a good data placement, parallelizing the initialization code in the same way as the "real work" is critical.

Due to the correspondence of iteration number with data element, the parallelization of the "real work" loop means that elements 1 to  $n/p$  of the vectors *a* and *b* are accessed by process 0. To minimize memory access times, you would like these data elements to be allocated from the memory of the node running process 0. Similarly, elements  $n/p+1$  to  $2*n/p$  are accessed by process 1, and you would like them allocated from the memory of the node running it. And so on. This is accomplished using the first-touch policy. The processor which first touches a data element causes the page holding that data element to be allocated from its local memory. Thus, if the data are to be allocated so that each processor can make local accesses during the "real work" section of the code, each processor must be the one to initialize its share of the data. This means that the initialization loop is parallelized the same way as the "real work" loop.

Now consider why the simple schedule type is important. Data are placed using a granularity of one page, so they will only end up in their optimal location if the same processor initializes all the data elements in a page. This is just like optimizing the parallelization to account for cache lines as was discussed above, only now you need to block the data into pages rather than lines. The default page size is 16 KB, or 4096 data elements; this is a fairly large number. Since the simple schedule type blocks together as many elements as possible for a single processor to work on ( $n/p$ ), it will create more optimally-allocated pages than any other work distribution.



For the example above,  $n = 8388608$ . If the program is run on 128 processors,  $n/p = 65536$ , which means that each processor's share of each array fills 16 pages ( $65536 \text{ elements} * 4\text{B/element} / 16 \text{ KB/page}$ ). It is unlikely that an array begins exactly on a page boundary, so you would expect 15 of a processor's 16 pages to contain only "its elements" and one page to contain some of "its elements" and some of another processor's elements. Although for the optimal data layout no pages would share elements from multiple processors, this small imperfection will have a negligible effect on performance.

On the other hand, if you use an interleaved schedule type, all processors repeatedly attempt to concurrently touch 128 consecutive data elements. Since 128 consecutive data elements are almost always on the same page, the resulting data placement could be anything from a random distribution of the pages to one in which all pages end up in the memory of a single processor. This initial data placement will certainly affect the performance.

In general, you should try to arrange it so that each processor's share of a data structure exceeds the size of a page. If you need finer granularity than this, you may need to consider using the reshaped directives which will be discussed later.

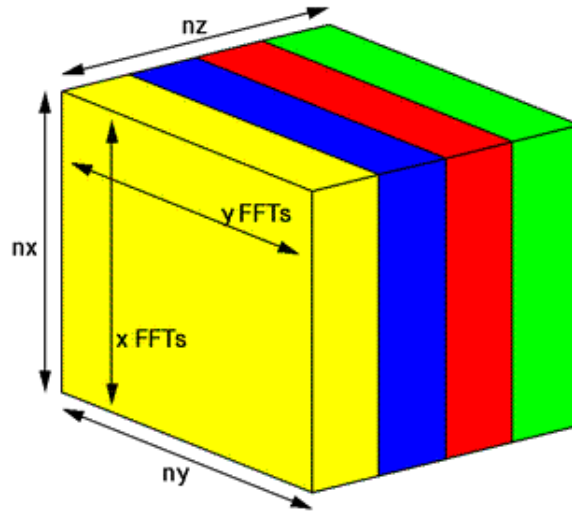
back to top

#### 4.5.9.2 First-Touch and Multiple Data Distributions

Programming for the default first-touch policy is effective if the application only requires one distribution of data. If different data placements are required at different points in the program, more sophisticated techniques are required. The data placement directives (see below) allow data structures to be redistributed at will, so they provide the most general approach to handling multiple data placements. But in many cases, you don't need to go to this extra effort.

For example, the initial data placement can be optimized with first-touch and migration can then be used to redistribute data later during the run of the program. In other cases, copying can be used to handle multiple data placements. As an example, consider the calculation used in the NAS FT benchmark. Basically this is a three-dimensional Fast Fourier Transform (FFT) carried out multiple times. It is implemented by performing one-dimensional FFTs in each of the three dimensions. As long as you

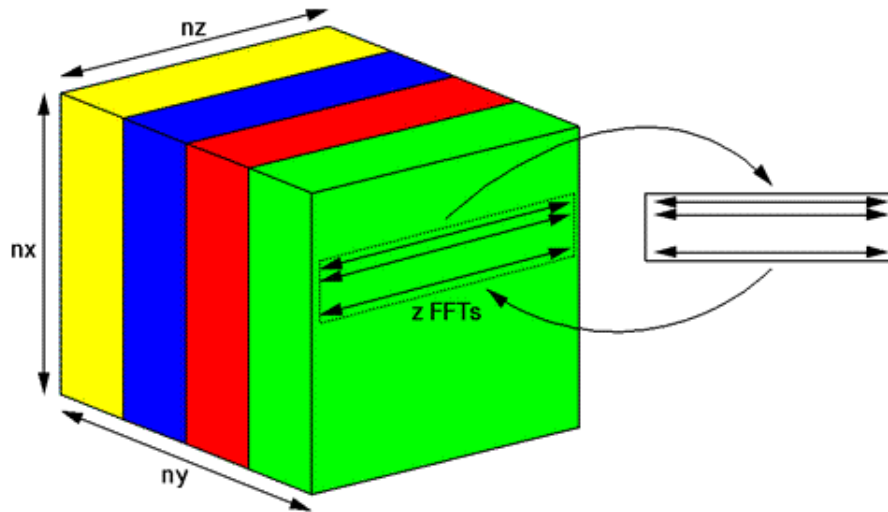
don't need more than 128-way parallelism, the x- and y-calculations may be parallelized by partitioning data along the z-axis:



For Origin first-touch is used possible to place  $n_z/p$  xy-planes in the local memory of each processor. The processors are then responsible for performing the FFTs in their share of planes. Once these are completed, z-transforms need to be done.

On a typical distributed memory computer, this presents a problem. Since the xy-planes have been distributed, no processor has access to all the data along the lines in the z-dimension, so the z-FFTs cannot be performed. The most common way to deal with this is to redistribute the data by transposing the array so that each processor holds zy-planes. One-dimensional FFTs may then be carried out along the first dimension in each plane. A transpose is convenient since it is a well-defined operation which can be optimized and packaged into a library routine. In addition, since it moves a lot of data together, it is more efficient than moving individual elements to perform z-FFTs one at a time.

Origin, however, is a shared memory computer, so this explicit data redistribution is not needed. Instead, split up the z-FFTs among the processors by partitioning work along the y-axis:



Each processor then copies several z-lines at a time into a scratch array, performs the z-FFTs on the copied data, and completes the operation by copying back the results.

Copying has several advantages over explicitly redistributing the data:

- It brings several z-lines directly into each processor's cache, and the z-FFTs are performed on these in-cache data. A transpose moves nonlocal memory to local memory, and this must then be moved into the cache in a separate step.
- It reduces TLB misses. Recall that this is the same technique used earlier in the single-processor tuning section to reduce TLB misses when operating on multi-dimensional arrays.
- It scales very well since it is perfectly parallel. Parallel transposes require a lot of synchronization points.
- It is easy to implement. Optimizing a transpose is a fair amount of work.

So, by combining it with copying, first-touch is applicable to this problem which ostensibly would require two data distributions.

back to top

### 4.5.9.3 Fortran Data Distribution Directives

You can create any data placement you want using the default first-touch policy. This programming style, however, may not make it easy for others to see what you are trying to accomplish. Furthermore, changing the data placement could require modifying a lot of code. An alternate way to distribute data is through compiler directives. Directives make the data placement clear since they state explicitly how the data are distributed. In addition, modifying the data distribution is easy since it only requires updating these comments rather than changing the program logic or the way the data structures are implemented.

The MIPSpro 7.1 compilers support two types of data distribution directives: regular and reshaped. Both

allow you to place blocks or stripes of Fortran arrays in the memory of the CPUs that operate on those data. The regular data distribution directives, however, are limited to single-page granularity whereas the reshaped directives permit arbitrary granularity. The regular data distribution directives are described now; the reshaped directives will be covered later.

An example is the easiest way to tell you how the regular data distribution directives are used. Below is the same simple vector operation used above in the section on programming for the default first-touch policy. In this implementation, though, distribution directives are used to ensure proper placement of the arrays.

```

integer i, j, n, niters
parameter (n = 8*1024*1024, niters = 1000)
c
c-----Note that the distribute directive is used after the arrays
c-----are declared.
c
      real a(n), b(n), q
c$distribute a(block), b(block)

c initialization

      do i = 1, n
        a(i) = 1.0 - 0.5*i
        b(i) = -10.0 + 0.01*(i*i)
      enddo

c real work

      do it = 1, niters
        q = 0.01*it
c$doacross local(i), shared(a,b,q), affinity (i) = data(a(i))
        do i = 1, n
          a(i) = a(i) + q*b(i)
        enddo
      enddo

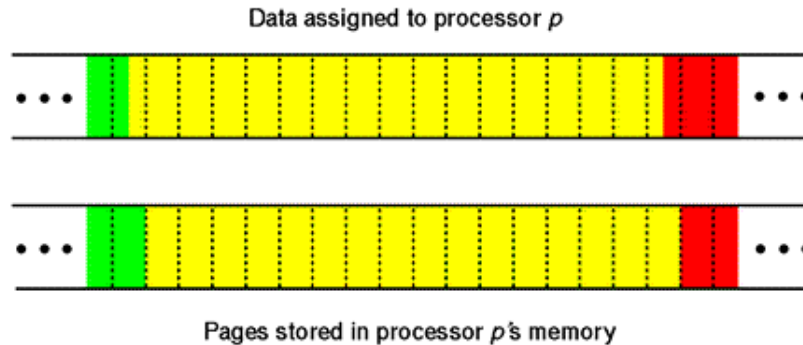
```

Two directives are used: The first, `c$distribute`, instructs the compiler to allocate the memory for the arrays `a` and `b` from all nodes on which the program runs. The second directive, `c$doacross ... affinity (i) = data (a(i))`, tells the compiler distribute work to the processors based on how the data are distributed. Note that as a result of the `c$distribute` directive, a parallel initialization loop is *not* required to ensure that the data are placed optimally (although it is still good programming practice to parallelize the data initializations). Let's look in detail at what these directives do.

The `c$distribute` directive distributes both arrays using a *block* mapping. That is, each array is partitioned into  $p$  blocks of size  $\text{ceil}(n/p)$  with the first block assigned to the first processor, the second block assigned to the second processor, and so on; here,  $p$  is the number of processors used to run the program. The intent of assigning blocks to processors is to allow them to be stored in the processors' local memory. There is a restriction, though, that a granularity of one page be used. This means that a page straddling blocks belonging to different processors must be stored entirely in the memory of a single node. Thus, nonlocal accesses may be required to some of the data on such a page.

You can see the effect of this by considering the following example: For  $n = 8388608$  and  $p = 128$ , the block size is  $\text{ceil}(n/p) = 65536$ , so each block is the size of 16 pages ( $65536 \text{ elements} * 4\text{B/element} / 16 \text{ KB/page}$ ). Since it is unlikely that an array begins exactly on a page boundary, each block typically

spans 15 complete pages and two partial pages:



The 15 complete pages are allocated from the local memory of the CPU responsible for the block, but the data in the two partial pages could end up stored in the memory of a different CPU. Which memory a page is stored in is determined by a heuristic in the compiler. For this block mapping, the first partial page is typically stored in the memory local to another processor, but the second partial page is stored in the same memory as the 15 complete pages. If the first partial page ends up in the same node as the rest of the block, all accesses are local. But if it ends up in a different node, nonlocal memory references will be required. In an optimal data layout, no pages share elements assigned to different nodes. An imperfect data distribution, however, has a negligible effect on performance as long as the block size exceeds a few pages since the ratio of nonlocal to local accesses is small. For cases in which the block size is less than a page, you either have to live with a larger fraction of nonlocal accesses or consider using the reshaped directives, which do not have the page granularity restriction.

Now let's look at the `c$doacross ... affinity (i) = data (a(i))` directive. The `c$doacross`, of course, instructs the compiler to run this loop in parallel. But instead of using a schedule type to distribute iterations to processors, an *affinity* clause is used. This new construct tells the compiler to execute iteration  $i$  on the CPU responsible for data element  $a(i)$ . Thus, work is divided among the CPUs according to the data distribution. The page granularity, though, is not considered in partitioning the work: The first CPU carries out iterations 1 to  $\text{ceil}(n/p)$ , the second CPU performs iterations  $\text{ceil}(n/p) + 1$  to  $2 * \text{ceil}(n/p)$ , and so on; this ensures a proper load balance at the expense of a few nonlocal memory accesses.

For the block distribution in this example, the *affinity* clause and a simple schedule type assign work to processors identically. The *affinity* clause, however, has advantages. First, for more complicated data distributions, there are no schedule types that can achieve the proper work distribution. Second, using the *affinity* clause makes the code easier to maintain since you can change the data distribution without having to modify the `c$doacross` directive to realign the work with the data.

back to top

---

#### 4.5.9.4 Regular Data Distribution Directives

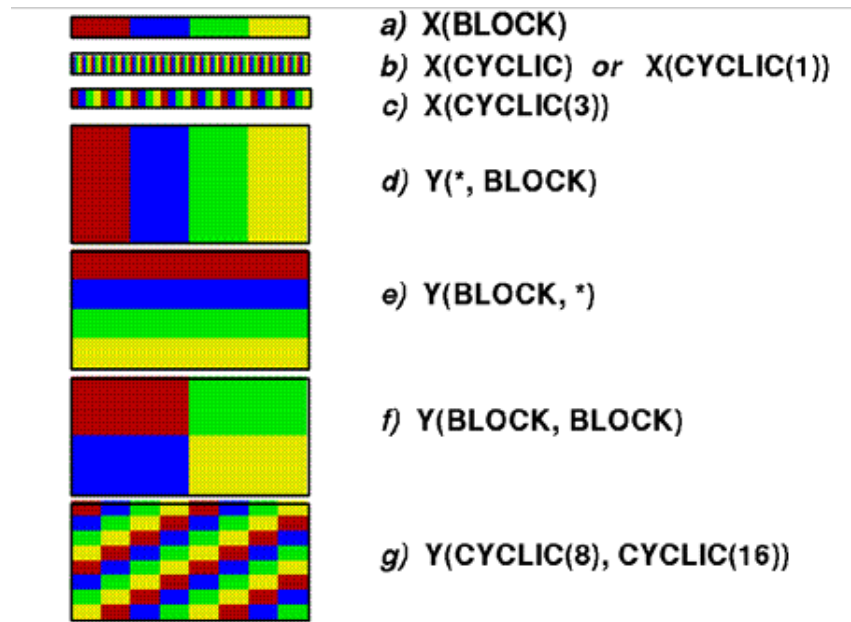
These two directives, of course, permit more general data distributions than seen in the above example. The following table lists the full syntax for them, as well as for the rest of the regular data distribution directives:

Directive	Purpose
c\$distribute A(<dist>,<dist>,...) [onto (<np>,<np>,...)]	Declares static distribution of array by pages.
c\$doacross affinity(i) = data(A(a*i+b))	Clause that puts loop iterations on the node that owns the data used in that iteration.
c\$doacross affinity(i) = thread(<expr>)	Clause that assigns loop iterations to threads based on the induction variable.
c\$doacross nest(i,j,...) [onto (<np>,<np>,...)]	Clause that allows perfectly nested parallel loops.
c\$page_place A(<addr>,<sz>,<thread>)	Declares specific placement of range of addresses to thread.
c\$redistribute A(<dist>,<dist>,...) [onto (<np>,<np>,...)]	Executable statement that redistributes array at runtime.
c\$dynamic A	Declares an array can be dynamically redistributed at runtime.

The notation <dist> specifies how the data for each dimension of an array are mapped to the CPUs. As seen above, setting <dist> to BLOCK specifies a block mapping. The other mappings are CYCLIC and CYCLIC(<expr>); \* is used to tell the compiler to not distribute a dimension.

A *cyclic* mapping means that elements in the specified dimension of the array are distributed to the CPUs running the program as you deal out a deck of cards. Thus, c\$distribute a(\*,CYCLIC) assigns columns 1,  $p+1$ ,  $2p+1$ , ... of the matrix a to the first processor, columns 2,  $p+2$ ,  $2p+2$ , ... to the second processor, and so on. CYCLIC(<expr>) is similar but represents a *block-cyclic* mapping. In this mapping, blocks, rather than individual elements, are assigned to processors cyclically with the block size given by the compile-time expression <expr>. Thus, c\$distribute a(\*,CYCLIC(2)) assigns columns 1, 2,  $2p+1$ ,  $2p+2$ , ... to the first processor, 3, 4,  $2p+3$ ,  $2p+4$ , ... to the second processor, and so on.

Combinations of the distribution options produce a large variety of mappings, as shown below:



In these pictures, the colors represent the assignment of data elements to processors. These assignments, however, do not take the single-page granularity into consideration. Only mappings that assign many contiguous elements to the same processor are likely to result in the intended distribution of data to the processors' local memory. Mappings *a* and *d* produce the desired data distributions for arrays of moderate size, while mappings *e* and *f* require large arrays for the data placements to be near-optimal. For the cyclic mappings --- *b*, *c*, and *g* --- you need to use the reshaped data distribution directives to achieve the intended results; these will be discussed shortly.

[back to top](#)

#### 4.5.9.4.1 The Onto Clause

If an array is distributed in multiple dimensions, the processors are apportioned as equally as possible across each dimension. For example, if an array has two distributed dimensions, an execution on 6 processors assigns three processors to the first dimension and two to the second ( $2 \times 2 = 6$ ). The optional *onto* clause allows you to override this default and explicitly control the number of processors in each dimension. so using `onto (2,3)` assigns two processors to the first dimension and three to the second.

## 6 Processors

`c$distribute A(block,block)`



`c$distribute A(block,block) onto (2,3)`



`c$distribute A(block,block) onto (1,2)`



`c$distribute A(block,block) onto (1,*)`



The values you provide specify the aspect ratio desired. If the available processors cannot exactly match the specified aspect ratio, the best approximation is used, so `onto (1,2)` assigns two processors to the first dimension and three to the second in a 6-processor execution. A special value, `*`, allows you to fix one or more dimensions with the remaining processors filling in the `*`-dimension; `onto (2,*)` assigns two processors to the first dimension and  $p/2$  to the second.

back to top

---

### 4.5.9.4.2 The Affinity Clause

The `affinity` clause has two forms. The `data` form, in which iterations are assigned to CPUs to match a data distribution, was used above. There, the correspondence between iterations and data distribution was quite simple. The `directive`, however, allows more complicated associations such as

```
c$distribute a(block,cyclic(1))

c$doacross local(i,j), shared(a)
c$&,    affinity(i) = data(a(2*i+3,j))
  do i = 1, n
    do j = 1, n
      a(2*i+3,j) = a(2*i+3,j-1)
    enddo
  enddo
```

The loop-index variable (`i` in this case) cannot appear in more than one dimension, and the expressions involving it are limited to the form  $a*i+b$ , where `a` and `b` are literal constants with `a` greater than zero.

In addition to the `data` form, a `thread` form, which executes iteration `i` on the thread given by an expression (modulo the number of threads), may also be used:

```

integer n, p, i
parameter (n = 8*1024*1024)
real a(n)

p = 1
c$ p = mp_numthreads()

c$doacross local (i), shared (a,p)
c&,    affinity(i) = thread(i/((n+p-1)/p))
do i = 1, n
    a(i) = 0.0
enddo

```

The expression may need to be evaluated in each iteration of the loop, so variables (other than the loop index) must be declared shared and not changed during the execution of the loop.

back to top

---

#### 4.5.9.4.3 The Nest Clause

Multi-dimensional mappings, such as  $f$  and  $g$  above, often benefit from parallelization over more than just the one loop to which a standard `c$doacross` directive applies. To permit such parallelizations, a `nest` clause is now available. It specifies that the full set of iterations in the loop nest may be executed concurrently. It works as follows:

```

real a(n,n)

c$doacross nest(i,j), local(i,j), shared(a)
do j = 1, n
do i = 1, n
    a(i,j) = 0.0
enddo
enddo

```

Here, all  $n \times n$   $i$ - and  $j$ -iterations are executed in parallel. To use this directive, the loops to be parallelized must be *perfectly nested*, that is, no code is allowed between any pair of `do` statements or any pair of `enddo` statements.

By adding an `affinity` clause, each processor will operate on its share of distributed data:

```

real a(n,n)
c$distribute a(block,block)

c$doacross nest(i,j), local(i,j), shared (a,n)
c&,&    affinity(i,j) = data(a(i,j))
do j = 1, n
do i = 1, n
    a(i,j) = 0.0
enddo
enddo

```

An `onto` clause may also be used to specify the aspect ratio of the parallelization:

```

        real a(m,n)
c$ distribute a(block,block)

c$doacross nest(i,j), local(i,j), shared (a,m,n)
c$&,      affinity(i,j) = data(a(i,j)) onto (2,1)
        do j = 1, n
            do i = 1, m
                a(i,j) = 0.0
            enddo
        enddo

```

In this example, twice as many processors will be used to parallelize the *i*-dimension as the *j*-dimension.

back to top

---

#### 4.5.9.4.4 The Page\_Place Directive

The block and cyclic mappings are well-suited for regular data structures such as the arrays pictured above. For irregular data structures, the `c$page_place` directive may be used. This directive allows you to assign ranges of data to precisely the processors you want, subject, of course, to the single-page granularity. `<addr>` is the starting address, `<sz>` is the size in bytes, and `<thread>` is the number of the destination processor. It is an executable statement, so you can use it to place statically or dynamically allocated data. (`c$distribute` is not an executable statement, so it may only be used on statically allocated data.)

An example of the use of `c$page_place` is the following:

```

        integer n, p, npp, i
        parameter (n = 8*1024*1024)
        real a(n)

        p = 1
c$      p = mp_numthreads()

c-----distribute a using a block mapping

        npp = (n + p-1)/p          ! number of elements per processor

c$doacross local(i), shared(a,npp)
        do i = 0, p-1
c$page_place (a(1 + i*npp), npp*4, i)
        enddo

```

In this example, `c$page_place` is used to create a block mapping of the array *a* onto *p* processors. If the array has not previously been distributed via first-touch or a `c$distribute` directive, `c$page_place` defines the initial data placement and incurs no cost. If the data have already been placed, `c$page_place` redistributes the array by migrating the pages from their initial locations to those specified by the directive. This data movement is not free, however, and it requires some amount of time to complete. If you use `c$page_place` and your intention is simply to place the data rather than redistribute them, make the `c$page_place` directive the first executable statement acting on the specified data structures and don't use other distribution directives on the same arrays.

If your intention is to redistribute the data, however, `c$page_place` will accomplish this. But for regularly distributed data, the `c$redistribute` directive will also work and is more convenient. This directive allows you to change a block and/or cyclic mapping defined by a `c$distribute` directive to another block and/or cyclic mapping. It generally needs to be used in conjunction with the `c$dynamic` directive, which declares that array distributions be calculated at run time rather than fixed at compile time. If you are interested in these directives, please refer to Chapter 6 of the *MIPSpro Fortran 77 Programmer's Guide* for more information on their use. No detailed discussion is presented here since you can generally achieve better performance through other techniques --- such as copying --- than through redistribution.

The `page_place` directive is currently the only data distribution directive available for C programs. It is used like the Fortran variant, but instead of supplying it the starting address, you use the first value in the array to be placed:

```
float a[N];
#pragma page_place (a[0], N*sizeof(float), iproc)
```

[back to top](#)

#### 4.5.9.5 Reshaped Data Distribution Directives

The regular data distribution directives provide an easy way to specify the desired data distribution, but they have a limitation: the distributions impose a granularity of one page. This limits how well they work for small arrays and cyclic distributions. For situations in which the single-page granularity is a problem, another directive is provided, `c$distribute_reshape`:

Directive	Purpose
<code>c\$distribute_reshape A(&lt;dist&gt;,&lt;dist&gt;,...) [onto (&lt;np&gt;,&lt;np&gt;,...)]</code>	Declares static reshaping of array regardless of page boundaries.

This directive distributes data precisely according to the block and/or cyclic mapping you specify. But it does so with an additional condition: it declares that your program makes no assumptions about the storage layout of reshaped arrays. The meaning and implications of this are best seen through an example.

[back to top](#)

##### 4.5.9.5.1 Reshaped Distribution Example

Below, a three-dimensional array `a` is declared and its `xy`-planes are distributed to processors' local memory using a reshaped cyclic mapping. Three calculations are then performed that normalize the entries along pencils in each dimension using the BLAS-1 routines `sasum` and `sscal` from CHALLENGEcomplib.

The reshaped data distribution means that each processor is assigned  $\text{ceil}(n_3/p)$  xy-planes in a cyclic order. Since the x- and y-dimensions are not distributed, each plane consumes  $n_1 * n_2$  consecutive data locations in normal Fortran storage order (i.e., the first  $n_1$  elements are down the first column, the next  $n_1$  elements are down the second column, and so on). But since the z-dimension is distributed, you can make no assumptions about the location of one plane compared with another. That is, even though standard Fortran storage order specifies that  $a(1,1,2)$  is stored  $n_1 * n_2$  elements after  $a(1,1,1)$ , when the reshape directive is used, *this is no longer guaranteed*.

```

integer i, j, k
integer n1, n2, n3
parameter (n1=200, n2=150, n3=150)
real a(n1, n2, n3)
c$ distribute_reshape(*,*,cyclic)
real sum

c-----This loop works correctly since each processor is assigned
c-----entire xy-planes
c
c$doacross local(k,j,sum), shared(a)
  do k = 1, n3
    do j = 1, n2
      sum = sasum(n1, a(1,j,k), 1)
      call sscal(n1, sum, a(1,j,k), 1)
    enddo
  enddo

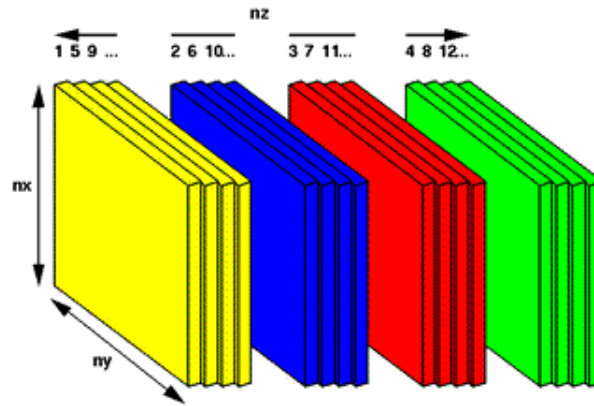
c-----This loop works correctly since each processor is assigned
c-----entire xy-planes
c
c$doacross local(k,i,sum), shared(a)
  do k = 1, n3
    do i = 1, n1
      sum = sasum(n2, a(i,1,k), n1)
      call sscal(n2, sum, a(i,1,k), n1)
    enddo
  enddo

c-----This loop fails since there can be holes in the storage layout
c-----between processors' share of the data
c
c$doacross local(j,i,sum), shared(a)
  do j = 1, n2
    do i = 1, n1
      sum = sasum(n3, a(i,j,1), n1*n2)
      call sscal(n3, sum, a(i,j,1), n1*n2)
    enddo
  enddo

```

The library routine `sasum` knows nothing about reshaped arrays. Its first argument simply tells it how many elements to sum, the second argument is the address at which to start summing, and the third argument is the stride, namely, the number of storage elements by which to increment the address to get to the next value included in the sum. In the first parallelized loop, values are summed down the x-dimension. These are adjacent to one another in memory, so the stride is 1. In the second loop, values are summed in the y-dimension. Each y-value is one length- $n_1$  column away from its predecessor, so these have a stride of  $n_1$ . Since the x- and y-dimensions are not distributed, these values are stored at the indicated strides, and the subroutine calls generate the expected results.

But for the z-dimension, this is not the case. You can make no assumptions about the storage layout in a distributed dimension, so the stride of  $n1*n2$  used in the third loop is not guaranteed to be correct. This third loop generates incorrect results when run in parallel. (If it is compiled without the `-mp` flag so that the directives are ignored, it, of course, does generate correct results.)



The problem in the z-dimension is the result of using a library routine that makes assumptions about the storage layout. You can correct this by removing those assumptions. One solution is to use standard array indexing instead of the library routine:

```
c-----This loop works since it makes no assumptions about the
c-----storage layout
c
c$doacross local(j,i,k,sum), shared(a)
  do j = 1, n2
    do i = 1, n1
      sum = 0.0
      do k = 1, n3
        sum = sum + abs(a(i,j,k))
      enddo
      do k = 1, n3
        a(i,j,k) = a(i,j,k)/sum
      enddo
    enddo
  enddo
```

No assumptions about the storage layout are made if you access elements of the array directly.

Another solution, which still allows you to employ the library routines, is to use copying:

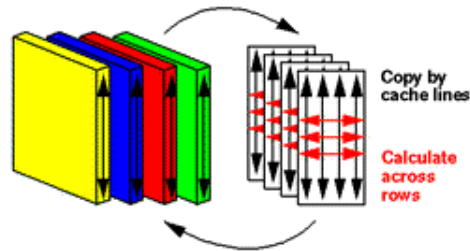
```
c-----This loop works since it makes no assumptions about the
c-----storage layout
c
c$doacross local(j,i,k,sum,tmp), shared(a)
  do j = 1, n2
    do i = 1, n1
      do k = 1, n3
        tmp(k) = a(i,j,k)
      enddo
      sum = sasum(n3, tmp, 1)
      call sscal(n3, sum, tmp, 1)
    enddo
  enddo
```

```

        enddo
    enddo

```

Since the copy loop directly accesses array elements, it does not make assumptions about the storage layout. And when a local scratch array is used, the stride is known, so `sasum` may be used safely. Note, though, that the following version, although it involves a bit more code and scratch space, is better since it makes much more effective use of cache lines:



c-----This loop works since it makes no assumptions about the  
c-----storage layout. It has better cache behavior than the  
c-----previous version.

```

c
c$doacross local(j,i,k,sum,tmp), shared(a)
    do j = 1, n2
        do k = 1, n3
            do i = 1, n1
                tmp(i,k) = a(i,j,k)
            enddo
        enddo
        do i = 1, n1
            sum = sasum(n3, tmp(i,1), n1)
            call sscal(n3, sum, tmp(i,1), n1)
        enddo
    enddo
enddo

```

`c$distribute_reshape`, although implemented for shared memory programs, is essentially a distributed memory construct. You use it to partition data structures the way you would on a distributed memory computer. Once the data structures are partitioned, accessing them in the distributed dimension is restricted. In considering whether you can safely perform an operation on a reshaped array, if it is something that wouldn't be allowed on a distributed memory computer, then its validity is suspect. But if the operation makes no assumptions about data layout, Origin's shared memory allows you to conveniently perform many operations that would be much more difficult to implement on a distributed memory computer.

back to top

#### 4.5.9.5.2 Rolling Your Own Reshaped Distribution

The purpose of the `c$distribute_reshape` directive is to achieve the desired data distribution without page granularity limitations. Note that you can accomplish the same thing through the use of dynamic memory allocation and pointers. All you have to do is allocate enough space for each processor so that it doesn't need to share a page with another processor, pack the pieces each processor is responsible for

into its pages of memory, use `first-touch` or `page_place` to make sure each processor's pages are stored locally, and set up an array of pointers so that the individual pieces may be accessed conveniently.

For the above example, this looks like the following:

```

int  init  = _init;
int  place = _place;
int  p     = mp_numthreads();
int  pgsz  = getpagesize();
int  n1    = N1;
int  n2    = N2;
int  n3    = N3;
int  npp, planes, ip, k, j, i;
float **a, *data, sum, tmp[N3][N1];

/* number of planes per processor
 */
npp    = (n3 + p-1)/p;
planes = npp*n1*n2*sizeof(float);

/* Allocate npp planes of size n1*n2*sizeof(float) for each
 * processor, and throw in an extra page of space so no
 * processors have to share space on the same page.
 */
data = (float *) malloc(pgsz + p*(planes + pgsz));

/* a holds pointers to the beginning of each plane. There are npp
 * per processor and p processors for a total of p*npp >= n3 pointers.
 */
a     = (float **) malloc(p*npp*sizeof(float *));

/* Fill in the pointers to each plane. The xy-planes are distributed
 * cyclically, so processor ip holds the data for the npp planes
 * ip, ip+p, ip+2*p, .... Due to the interleaving of ownership, this
 * loop should not be run in parallel since it will result in
 * false sharing.
 */
for (ip=0; ip<p; ip++) {
    a[ip] = (float *) ((char *) data + pgsz + ip*(planes + pgsz));
    for (k=ip+p; k<(p*npp); k+=p) {
        a[k] = a[k-p] + n1*n2;
    }
}

/* Use the page_place directive to force all the pages each processor
 * owns to be allocated from its local memory. (Alternatively,
 * first-touch could have been used.)
 */
if (place) {
    #pragma parallel
    #pragma local(ip)
    #pragma byvalue(p,a,npp,n1,n2)
    {
        #pragma pfor iterate(ip=0;p;1)
        for (ip=0; ip<p; ip++) {

            /* Processor ip's data starts at address a[ip] and is of size
             * npp*n1*n2*sizeof(float).
             */
            #pragma page_place (a[ip][0], planes, ip)

```

```

    }
}

/* First-touch initialization.
*/
if (init) {
    #pragma parallel
    #pragma local(ip,i)
    #pragma byvalue(p,a,npp,n1,n2,pgsz)
    {
        #pragma pfor iterate(ip=0;p;1)
        for (ip=0; ip<p; ip++) {

            for (i=0; i<(npp*n1*n2); i++) {
                a[ip][i] = 1.0;
            }
        }
    }
}

/* Calculate sums along x-pencils.
*/
#pragma parallel
#pragma local (k,j,sum)
#pragma byvalue(n3,n2,n1,a)

{
    #pragma pfor iterate(k=0;n3;1)
    for (k=0; k<n3; k++) {
        for (j=0; j<n2; j++) {
            sum = sasum(n1, &a[k][n1*j], 1);
            sscal(n1, sum, &a[k][n1*j], 1);
        }
    }
}

/* Calculate sums along y-pencils.
*/
#pragma parallel
#pragma local (k,i,sum)
#pragma byvalue(n3,n2,n1,a)
{
    #pragma pfor iterate(k=0;n2;1)
    for (k=0; k<n3; k++) {
        for (i=0; i<n1; i++) {
            sum = sasum(n2, &a[k][i], n1);
            sscal(n2, sum, &a[k][i], n1);
        }
    }
}

/* Calculate sums along z-pencils. Since there are holes in the
 * z-dimension, xz-planes are first copied into a scratch array
 * so sasum will have contiguous data to operate on.
*/
#pragma parallel
#pragma local (k,j,i,tmp,sum)
#pragma byvalue(n3,n2,n1,a)
{
    #pragma pfor iterate(j=0;n2;1)

```

```

for (j=0; j<n2; j++) {
  for (k=0; k<n3; k++) {
    for (i=0; i<n1; i++) {
      tmp[k][i] = a[k][i+n1*j];
    }
  }
  for (i=0; i<n1; i++) {
    sum = sasum(n3, &tmp[0][i], n1);
    sscal(n3, sum, &tmp[0][i], n1);
  }
}
}

```

Each processor is assigned some of the planes and stores them contiguously in one piece of a big data space. The first page of the data space is not used since *malloc(3C)* reserves a small part of it to track how much space was allocated. Writing into this page of memory causes it to be allocated from the node that performed the allocation. You have no control over which node this is, so if you want a perfect data placement, this page should be skipped. In addition, holes are left in the data space so that no pair of processors' pieces share the same page of memory. This allows the `page_place` pragma to be used to place the pages in the local memory of the processor to which they belong. While x- and y-calculations may be performed on the data as is, the holes force the z-data to be copied to a scratch array before the z-pencil calculations are performed.

back to top

---

#### 4.5.9.5.3 Reshaped Distribution Restrictions

If proper data placement is important to achieving the best performance from your program and if the page granularity limitations of the regular data distribution directives are an issue, then you need to use reshaped arrays. You can use either the `c$distribute_reshape` directive, or you can roll your own data structures using dynamic memory allocation and pointers as in the example above. If you use the directive, there are some restrictions you need to be aware of:

- The distribution of a reshaped array cannot be changed dynamically (that is, there is no `redistribute_reshape` directive)
- Initialized data cannot be reshaped
- Arrays that are explicitly allocated through *alloca(3C)/malloc(3C)* and accessed through pointers cannot be reshaped
- An array that is equivalenced to another array cannot be reshaped
- I/O for a reshaped array cannot be mixed with namelist I/O or a function call in the same I/O statement
- A `COMMON` block containing a reshaped array cannot be linked `-Xlocal`

In addition, some care must be used in passing reshaped arrays to subroutines. If you pass the entire array, then you are not likely to run into any problems as long as the declared size of the array in the subroutine matches the size of array that was passed to it. For example,

```

      real a(n1,n2)
c$distribute_reshape a(block,block)

```

```

call sub(a,n1,n2)
.
.
.
subroutine sub(a,n1,n2)
real a(n1,n2)
.
.
.

```

works fine, but

```

real a(n1,n2)
c$distributed_reshape a(block,block)

call sub(a,n1*n2)
.
.
.
subroutine sub(a,n)
real a(n)
.
.
.

```

doesn't since the subroutine expects an array of a different shape than has been passed to it.

Inside a subroutine, you don't need to declare how an array has been distributed. In fact, the subroutine is more general if you *don't* declare the distribution. The compiler will generate versions of the subroutine necessary to handle all distributions which are passed to it. For example, in the following case

```

real a(n1,n2)
c$distributed_reshape a(block,block)
real b(n1,n2)
c$distributed_reshape b(cyclic,cyclic)

call sub(a,n1,n2)
call sub(b,n1,n2)
.
.
.
subroutine sub(a,n1,n2)
real a(n1,n2)
.
.
.

```

the compiler will generate code to handle both distributions that are passed to the subroutine. As long as the data calculations are efficient for both distributions, this will achieve good performance. On the other hand, if a particular algorithm only works for a specific data distribution, you can declare the required distribution inside the subroutine by using a `c$distributed_reshape` directive there. Then, all calls to the subroutine that pass a mismatched distribution will cause compile- or link-time errors.

If you only want to pass part of a reshaped array to a subroutine, all is fine as long as you only pass a single processor's piece. The example that began this section shows the passing of a single processor's share of a reshaped array and why you cannot pass sections which span a distributed dimension.

Most errors in accessing reshaped arrays are caught at compile- or link-time. However, some errors, such as passing a portion of a reshaped array which spans a distributed dimension, can only be caught at run time. You can instruct the compiler to do runtime checks for these with the `-MP:check_reshape=on` flag. You should use this flag during the development and debugging of programs which use reshaped arrays. In addition, Chapter 6 of the *MIPSpro Fortran 77 Programmer's Guide* provides more information, such as implementation details, about reshaped arrays.

back to top

---

### 4.5.9.6 Investigating Data Distributions

Sometimes it is useful to check how data have been distributed. If you are using the data distribution directives, the first thing you should do is set the environment variable `_DSM_VERBOSE`. When it is set, the program will print out messages at run time which tell you whether the distribution directives are being used, which physical processors the threads are running on, and what the page sizes are. They look like the following:

```
% setenv _DSM_VERBOSE
% a.out
[ 0]: 16 CPUs, using 4 threads.
      1 processors per memory
      Migration: OFF
      MACHINE: IP27 --- is NUMA ---
      MACHINE: IP27 --- is NUMA ---
Created 4 MLDs
Created and placed MLD-set. Topology-free, Advisory.
MLD attachments are:
      MLD 0. Node /hw/module/3/slot/n4/node
      MLD 1. Node /hw/module/3/slot/n3/node
      MLD 2. Node /hw/module/3/slot/n1/node
      MLD 3. Node /hw/module/3/slot/n2/node
[ 0]: process_mldlink: thread 0 (pid 3832) to memory 0. -advisory-.
Pagesize: stack 16384 data 16384 text 16384 (bytes)
--- finished MLD initialization ---
      .
      .
      .
[ 0]: process_mldlink: thread 1 (pid 3797) to memory 1. -advisory-.
[ 0]: process_mldlink: thread 2 (pid 3828) to memory 2. -advisory-.
[ 0]: process_mldlink: thread 3 (pid 3843) to memory 3. -advisory-.
[ 0]: process_mldlink: thread 0 (pid 3832) to memory 0. -advisory-.
```

These messages are useful in verifying that the requested data placements are actually being performed.

The utility `dplace(1)` disables the data distribution directives and should not be used with MP library programs. If you do use it, however, these messages will at least make you aware of your mistake:

```
% dplace reshape
```

```
[ 0]: 16 CPUs, using 4 threads.  
      DSM disabled.  
      Migration: OFF  
      MACHINE: IP27 --- is NUMA ---
```

---

Once you have verified that the directives are enabled, you can look at the data distributions. For example, you may want to investigate what effect page granularity has had on an array distributed using one of the regular data distribution directives. You can obtain the necessary information for such an investigation with the `dsm_home_threadnum()` intrinsic. It simply takes an address as an argument and returns the number of the thread in whose local memory the page containing that address is stored. It is used as follows:

```
integer dsm_home_threadnum  
  
numthread = dsm_home_threadnum(array(i))
```

Since two CPUs are connected to each node and they share the same memory, this function returns the lowest numbered thread running on the node.

In addition to this information, you may also determine which physical node a page of memory is stored in. (Note, though, that this will generally change from run to run.) This is done via the `syssgi(2)` system call using the `SGI_PHYSP` command. The routine `va2pa` below translates a virtual address to a physical address using this system call, and this is translated to a node number with the following macro:

```
#define ADDR2NODE(A) ((int) (va2pa(A) >> 32))
```

Note that the node number is generally half the lowest numbered CPU on the node, so if you would prefer to use CPU numbers, multiply the return value by two.

Here is an example of the type of information you can obtain. These routines were used in a program which allocates two vectors, `a` and `b`, of size 256 KB, initializes them sequentially, and then prints out the locations of their pages. The default first-touch policy resulted in the following page placements for a run using five processors:

```
Distribution for array "a"  
address      byte index  thread  proc  
-----  
0xffffffffb4760 0x      0        0      12  
0xffffffffb8000 0x    38a0    0      12  
0xffffffffbc000 0x    78a0    0      12  
0xffffffffc0000 0x    b8a0    0      12  
0xffffffffc4000 0x    f8a0    0      12  
0xffffffffc8000 0x   138a0    0      12  
0xffffffffcc000 0x   178a0    0      12  
0xffffffffd0000 0x   1b8a0    0      12  
0xffffffffd4000 0x   1f8a0    0      12  
0xffffffffd8000 0x   238a0    0      12  
0xffffffffdc000 0x   278a0    0      12  
0xffffffe0000 0x   2b8a0    0      12  
0xffffffe4000 0x   2f8a0    0      12  
0xffffffe8000 0x   338a0    0      12  
0xfffffec0000 0x   378a0    0      12  
0xfffffff0000 0x   3b8a0    0      12
```

```
0xffffffff4000 0x 3f8a0 0 12
```

Distribution for array "b"

address	byte	index	thread	proc
0xffffffff6e0f8	0x	0	0	12
0xffffffff70000	0x	1f08	0	12
0xffffffff74000	0x	5f08	0	12
0xffffffff78000	0x	9f08	0	12
0xffffffff7c000	0x	df08	0	12
0xffffffff80000	0x	11f08	0	12
0xffffffff84000	0x	15f08	0	12
0xffffffff88000	0x	19f08	0	12
0xffffffff8c000	0x	1df08	0	12
0xffffffff90000	0x	21f08	0	12
0xffffffff94000	0x	25f08	0	12
0xffffffff98000	0x	29f08	0	12
0xffffffff9c000	0x	2df08	0	12
0xfffffa0000	0x	31f08	0	12
0xfffffa4000	0x	35f08	0	12
0xfffffa8000	0x	39f08	0	12
0xfffffac000	0x	3df08	0	12

The sequential initialization combined with the first-touch policy causes all the pages to be placed in the memory of the first thread (which for this run was processor 12, or node 6).

Here are the results from the same code after setting the environment variable `_DSM_ROUND_ROBIN` to use a round-robin page placement:

Distribution for array "a"

address	byte	index	thread	proc
0xfffffb4750	0x	0	2	2
0xfffffb8000	0x	38b0	0	0
0xfffffbc000	0x	78b0	4	4
0xfffffc0000	0x	b8b0	2	2
0xfffffc4000	0x	f8b0	0	0
0xfffffc8000	0x	138b0	4	4
0xfffffcc000	0x	178b0	2	2
0xfffffd0000	0x	1b8b0	0	0
0xfffffd4000	0x	1f8b0	4	4
0xfffffd8000	0x	238b0	2	2
0xfffffdc000	0x	278b0	0	0
0xfffffe0000	0x	2b8b0	4	4
0xfffffe4000	0x	2f8b0	2	2
0xfffffe8000	0x	338b0	0	0
0xfffffec000	0x	378b0	4	4
0xffffff0000	0x	3b8b0	2	2
0xffffff4000	0x	3f8b0	0	0

Distribution for array "b"

address	byte	index	thread	proc
0xfffff6e0e8	0x	0	0	0
0xfffff70000	0x	1f18	4	4
0xfffff74000	0x	5f18	2	2
0xfffff78000	0x	9f18	0	0
0xfffff7c000	0x	df18	4	4
0xfffff80000	0x	11f18	2	2
0xfffff84000	0x	15f18	0	0

0xffffffff88000	0x	19f18	4	4
0xffffffff8c000	0x	1df18	2	2
0xffffffff90000	0x	21f18	0	0
0xffffffff94000	0x	25f18	4	4
0xffffffff98000	0x	29f18	2	2
0xffffffff9c000	0x	2df18	0	0
0xfffffa0000	0x	31f18	4	4
0xfffffa4000	0x	35f18	2	2
0xfffffa8000	0x	39f18	0	0
0xfffffac000	0x	3df18	4	4

Thus, these routines allow you to easily verify that the round-robin placement really does spread the data over all three nodes used by this 5-processor run.

---

Here is the routine `va2pa`:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/syssgi.h>

__uint64_t
va2pa(
void      *va)
{
    __uint64_t    pa;
    __uint32_t    pfn;
    int           status;
    static int     lpgsz, pgsz = -1;

    if (pgsz < 0) {
        int itmp;

        pgsz = itmp = getpagesize();
        for (lpgsz=0; itmp>1; itmp>>=1, lpgsz++);
    }

    if ((status = syssgi(SGI_PHYSP,va,&pfn)) != 0) {
        perror("Virtual to physical mapping failed");
        exit(1);
    }
    pa = (((__uint64_t) pfn << lpgsz) | ((__uint64_t) va & (pgsz-1)));

    return (pa);
}
```

[back to top](#)

---

#### 4.5.10 Non-MP Library Programs and Dplace

Now we consider programs which do not use the MP library. These include message passing programs employing MPI, PVM and other libraries, as well as "roll your own parallelism" implemented via *fork(2)* or *sproc(2)*. If such programs have been properly parallelized but do not scale as expected on Origin, they may require data placement tuning.

If you are developing a new parallel program or are willing to modify the source of your existing programs, you can ensure a good data placement by adhering to the following programming practices which take advantage of the default first-touch data placement policy:

- In a program that starts multiple processes using *fork(2)*, each process should allocate and initialize its own memory areas. The memory then resides in the node where the process runs.
- In a program that starts multiple processes using *sproc(2)*, do not allocate all memory prior to creating child processes. In the parent process, allocate only memory used by the parent process and memory that is used by all processes. This memory will be located in the node where the parent process runs.

Each child process should allocate any memory areas that it uses exclusively. Those areas will be located in the node where that process runs.

Programs that violate these practices can end up with poor data placement, which will impact performance. In addition, programs that use existing libraries such as MPI 2.0, which do not adhere to these practices, can also have performance problems. For these programs, data placement is tuned with the utility *dplace(1)*, which is described now.

*Dplace* allows you to

- Change the page size used
- Enable page migration
- Specify the topology used by the threads of a parallel program
- Indicate resource affinities
- Assign memory ranges to particular nodes

It is a command line utility with the following syntax:

```
dplace [-place placement_file]  
       [-data_pagesize n-bytes]  
       [-stack_pagesize n-bytes]  
       [-text_pagesize n-bytes]  
       [-migration threshold]  
       [-propagate]  
       [-mustrun]  
       [-v[erbose]]  
       program [program-arguments]
```

In addition, there is also a subroutine library interface, *dplace(3)*, which not only allows one to accomplish from within a program the same things that the command line utility does, but also allows dynamic control over some of the data placement choices.

Although *dplace* can be used to adjust policies and topology for any program, in practice there are only a few types of programs that it will benefit, namely, those mentioned above: MPI 2.0 programs, PVM

programs, and parallel programs that explicitly use *sproc(2)* or *fork(2)*. *dplace* should in general *not* be used with MP library programs since it disables the data placement specifications made via the MP library compiler directives. For these programs, use the MP library environment variables instead of *dplace*.

We'll now explain *dplace*'s capabilities.

back to top

---

#### 4.5.10.1 Changing the Page Size

The section on single-processor tuning showed how to use *dplace* to change the page size:

```
% dplace -data_pagesize 64k -stack_pagesize 64k program
```

This command increases the size of two of the three types of pages --- data and stack --- from the 16 KB default to 64 KB; the text page size is unchanged. This is useful in situations in which TLB thrashing causes poor performance. In general, this problem only occurs for pages which store program data structures. Global variables, such as those in Fortran common blocks, are stored in data segments; their page size is controlled by *data\_pagesize*. Variables local to subroutines are allocated off the stack, so it is also useful to change *stack\_pagesize* when trying to fix TLB thrashing problems. Increasing the *text\_pagesize* is not generally of benefit for scientific programs.

There are some restrictions, however, on what page sizes can be used. First, the only valid page sizes are 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, and 16 MB. Second, the system administrator must specify the percentage of memory to be allocated to the various possible page sizes. Thus, if the system has not been set up to allow 1 MB pages, requesting this page size will be of no use. The page size percentages can be changed at run time, though, using the *systune(1M)* command:

```
# systune -i
Updates will be made to running system and /unix.install

systune-> lpage_watermarks

group: lpage_watermarks (dynamically changeable)
percent_totalmem_16m_pages = 0 (0x0)
percent_totalmem_4m_pages = 5 (0x5)
percent_totalmem_1m_pages = 50 (0x32)
percent_totalmem_256k_pages = 0 (0x0)
percent_totalmem_64k_pages = 10 (0xa)
percent_totalmem_16k_pages = 0 (0x0)

systune-> percent_totalmem_1m_pages 0
percent_totalmem_1m_pages = 50 (0x32)
Do you really want to change percent_totalmem_1m_pages to 0 (0x0)? (y/n) y

systune-> percent_totalmem_64k_pages 30
percent_totalmem_64k_pages = 10 (0xa)
Do you really want to change percent_totalmem_64k_pages to 30 (0x1e)? (y/n) y

systune-> lpage_watermarks
```

```
group: lpage_watermarks (dynamically changeable)
  percent_totalmem_16m_pages = 0 (0x0)
  percent_totalmem_4m_pages = 5 (0x5)
  percent_totalmem_1m_pages = 0 (0x0)
  percent_totalmem_256k_pages = 0 (0x0)
  percent_totalmem_64k_pages = 30 (0x1e)
  percent_totalmem_16k_pages = 0 (0x0)
```

systune-> q

back to top

---

### 4.5.10.2 Enabling Page Migration

The operating system tries to allocate the memory a process uses from the node on which it runs. But once memory has been allocated, its location is fixed. If the initial placement proves nonoptimal, performance may suffer. In such situations, page migration may be of help. Enabling page migration tells the operating system to move pages of memory to the nodes that access them most frequently, so a poor initial placement can be corrected. Recall that page placement is not an issue for single-processor jobs, so migration is only a consideration for multiprocessor programs. Furthermore, migration is only a potential benefit to programs employing a shared memory model; MPI, PVM, and other message-passing programs control data layout explicitly, so automated control of data placement would only get in their way.

By default, the IRIX automatic page migration facility is disabled because page migration is an expensive operation that impacts all CPUs, not just the ones used by the program whose data are being moved. The system administrator can temporarily enable page migration for all programs using the *sn(1)* command or can enable it permanently by using *systune(1M)* to set the `numa_migr_base_enabled` system parameter. (For more information, see the comments in `/var/sysgen/mtune/numa`.) Individual users who decide to use dynamic page migration for a specific program have two options. Most multiprocessor jobs use the MP library, so if you are going to use migration for an MP library program, you should use its environment variables, `_DSM_MIGRATION` and `_DSM_MIGRATION_LEVEL`; these are discussed elsewhere. For non-MP library programs, migration may be enabled using *dplace*:

```
% dplace -migration threshold program
```

The *threshold* is an integer between 0 and 100. It specifies the percentage difference between the number of remote memory accesses and local memory accesses, relative to a maximum counter value, that must be attained in order to trigger migration. Thus, a high threshold value causes migration to occur less often, whereas a low threshold causes it to happen more often. The threshold value 0 is special: it is used to turn migration off.

(Note that the operation of the migration threshold is inverted from the way the MP library's migration level parameter works: `level = 100 - threshold`. Thus, a high level is a low threshold, and vice versa. *dplace* currently uses the threshold value since it accurately represents how the operating system makes migration decisions. But since the threshold is a bit counter-intuitive as a measure of migration aggressiveness, *dplace* will add support for the migration level in its next release.)

If you wish to enable migration, use a conservative threshold, say 90. This won't impact other users of

the system too much, and it should permit enough migration to correct a poor initial data placement but not so much that the expense of moving data is more than the cost of accessing them remotely.

back to top

---

### 4.5.10.3 Specifying the Topology

When executing a parallel program, the operating system tries to arrange it so that the processes making up the parallel program run on nearby nodes to minimize access costs for data that are shared between them. Here, "nearby" is defined as minimizing a simple distance metric that measures how many router hops are required to get from one node to another. For example, for an 8-node (i.e., 16-processor) system, the distance between any two nodes using this metric is

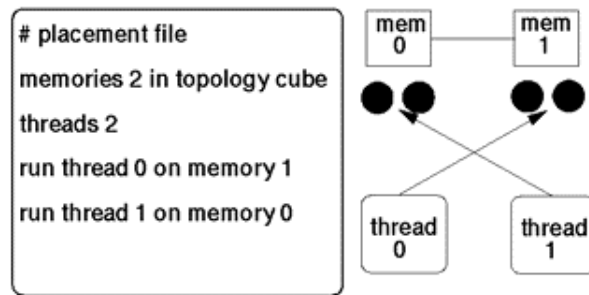
node	0	1	2	3	4	5	6	7
0	0	1	2	2	2	2	3	3
1	1	0	2	2	2	2	3	3
2	2	2	0	1	3	3	2	2
3	2	2	1	0	3	3	2	2
4	2	2	3	3	0	1	2	2
5	2	2	3	3	1	0	2	2
6	3	3	2	2	2	2	0	1
7	3	3	2	2	2	2	1	0

Thus, if the first two threads of a 6-way parallel job are placed on node 0, the second and third threads will be placed on node 1 since, at a distance of only 1, it is the closest node. The final two threads may be placed on any of nodes 2 through 5 since they are all a distance of 2 away.

For most applications, this *cluster* topology is perfect and one need not worry further about topology issues. For those cases in which it may be of benefit, however, *dplace* does provide you with a way of overriding this default placement and specifying other topologies. This is done through a *placement file*.

A placement file is a text file indicating how many nodes --- which are called *memories* to stress that they are a source of memory --- will be used to run a program, what topology the memories should be laid out in, and where the threads will run on those memories. The diagram below shows a simple placement file and its effect on the program:

## placement file specification



With this placement file, two memories are used to run this two-thread program; without it, just one memory would have been used since two CPUs are associated with each memory and only one CPU is needed per thread. In addition, the memories are laid out in a cube topology rather than the default cluster topology. For this simple case of just two memories, there is no difference; but if more memories had been requested, the cube topology would ensure that memories which should be hypercube neighbors end up adjacent to each other in the Origin system. Finally, threads are placed onto the memories in the reverse of the natural order, i.e., thread 0 is run on memory 1 and thread 1 is run on memory 0.

The above placement file has one limitation: it is not *scalable*. That is, if you want to change the number of threads, you need to change the placement file. This can be avoided by using environment variables to make the placement file scalable, as shown below:

```
# scalable placement_file
memories $NP in topology cluster # set up memories which are close
threads $NP # number of threads
# run the last thread on the first memory etc.
distribute threads $NP-1:0:-1 across memories
```

Here, instead of hard-coding the number of threads into the placement file, the environment variable `$NP` is used. When the placement file is read, the environment variable is replaced by its value taken from the shell. Thus, if `NP` is set to 2, the first two statements have the same effect as the hard-coded version above. To make the last two statements scalable, they are replaced with a "distribute" statement. This one statement specifies how all the threads are mapped to the memories. In this case, the last thread (`$NP-1`) is mapped to memory 0, the next-to-last thread is mapped to memory 1, and so on. The expression `Beg:End:Str` is the syntax used to represent the sequence `{Beg, Beg+Str, Beg+2*Str, ..., End}`. It is needed to specify that threads are mapped to memories in reverse order. If the normal order were desired, the last statement could have been written in any of the following ways:

```
# Explicitly indicate order of threads
distribute threads 0:$NP-1:1 across memories

# No Beg:End:Str means 0:last:1
distribute threads across memories

# The stride can be omitted if it is 1
distribute threads 0:$NP-1 across memories

# Same notation applies to "across memories" clause
```

```
distribute threads $NP-1:0:-1 across memories $NP-1:0:-1

# Same notation applies to "across memories" clause
distribute threads across memories 0:$NP-1:1
```

The two most common libraries used for parallel processing, the MP library and the MPI 2.0 library, already have environment variables defined to mean the "number of threads." These are `MP_SET_NUMTHREADS` and `MPI_NP`, respectively. If you use *dplace* to alter the default placement of a program employing either of these libraries, use the appropriate environment variable to make the placement file scalable.

Note that you may use simple arithmetic on the environment variables in placement files. A scalable placement file which uses the more conventional placement of two threads per memory is

```
# Standard 2 thread per memory scalable placement_file
memories ($MP_SET_NUMTHREADS + 1)/2 in topology cluster
threads $MP_SET_NUMTHREADS
distribute threads across memories
```

back to top

---

#### 4.5.10.4 Placement File Syntax

In general, you won't need to use placement files. (The exception is when you are using MPI 2.0, which we'll cover later.) However, if you do decide to use one, you need to include in it the following specifications (cf. *dplace(5)*):

1. The number of memories being used and, optionally, their topology. The general form for this is

```
memories m [[in] topology none|cluster|cube]
```

The topologies *none* and *cluster* are the same: they run the program on a set of nodes that are closest to each other (in the distance metric above). If no topology is specified, *cube* is used, which selects a sub-hypercube of nodes. Note that since the number of nodes in a hypercube is a power of two, this topology can have some strange side effects when used on machines which do not have a power of two number of nodes. In general, use the *cluster* topology.

There is one additional topology called *physical*, but we'll wait until the next section on resource affinities to describe it.

2. The number of threads

```
threads n
```

3. The assignment of threads to memories. There are two statements which are used to specify this:

```
run thread n on memory m [using cpu k]
```

and

```
distribute threads [t0:t1[:dt]] across memories [m0:m1[:dm]] \  
[block [m]] | [cyclic [n]]
```

The first statement is not recommended since it is not scalable. The second statement was seen above, but this is its general form. The additional clauses are [block [m]] and [cyclic[n]]. These indicate how threads are handed out to memories. Usually, the first two threads are run on the first memory, the next two threads are run on the second memory, and so on; this is a block mapping with a block size of two. In a cyclic mapping, threads are distributed among memories as you deal out a deck of cards, only you are allowed to deal  $n$  threads at a time to each memory. When you run out of memories, the "deal" resumes with the first memory. The default is to use a block mapping, and the default block size is  $\text{ceil}(\text{number of threads}/\text{number of memories})$ ; i.e., two if there are twice as many threads as memories, and 1 if the number of threads and memories are equal.

In addition to these specifications, you may also use a placement file rather than command line switches to change the page sizes, enable migration, and toggle verbose mode on and off. The syntax for these options is

```
policy stack|data|text pagesize  $n$  [k|K]
```

```
policy migration  $n$  [%]
```

```
mode verbose [on|off|toggle]
```

respectively.

back to top

---

#### 4.5.10.5 Indicating Resource Affinity

The hardware characteristics of a particular Origin system are maintained in what is known as the *hardware graph* (see *hwgraph(4)*). This is made visible to users through a pseudo-file system mounted at /hw. All interesting hardware and pseudo-devices have an entry in this pseudo-file system. For example, all the nodes in a system can be listed with the following command:

```
% find /hw -name node -print  
/hw/module/1/slot/n1/node  
/hw/module/1/slot/n2/node  
/hw/module/1/slot/n3/node  
/hw/module/1/slot/n4/node  
/hw/module/2/slot/n1/node  
/hw/module/2/slot/n2/node  
/hw/module/2/slot/n3/node  
/hw/module/2/slot/n4/node  
/hw/module/3/slot/n1/node  
/hw/module/3/slot/n2/node  
/hw/module/3/slot/n3/node  
/hw/module/3/slot/n4/node  
/hw/module/4/slot/n1/node  
/hw/module/4/slot/n2/node  
/hw/module/4/slot/n3/node  
/hw/module/4/slot/n4/node
```

This particular system has 16 nodes (i.e., 32 processors), which are contained in 4 modules.

Entries from the hardware graph are used to indicate which resources, if any, you want a program run near. This is done by adding a `near` clause to the `memories` line in the placement file:

```
memories m [[in] topology none|cluster|cube|physical] [near [/hw/*] +]
```

For example, if Infinite Reality hardware is installed, it shows up in the hardware graph as:

```
% find /hw -name kona -print
/hw/module/1/slot/io5/xwidget/kona
```

In this case, it is in the fifth I/O slot of module 1. You may then indicate that you want your program run near this hardware device by using a line such as

```
memories 3 in topology cluster near /hw/module/1/slot/io5/xwidget/kona
```

in the placement file.

You can also use the hardware graph entries and the `physical` topology to run the program on specific nodes:

```
memories 3 in topology physical near /hw/module/2/slot/n1/node \  
                                     /hw/module/1/slot/n2/node \  
                                     /hw/module/3/slot/n3/node
```

This line causes the program to be run on node 1 of module 2, node 2 of module 1, and node 3 of module 3. In general, though, it is best to let the operating system pick which nodes to run on since users' jobs are likely to collide with each other if the `physical` topology is used.

back to top

---

#### 4.5.10.6 Assigning Memory Ranges

Placement files can also be used to place specific ranges of virtual memory on a particular node. This is done with a line of the form:

```
place range k to l on memory m [[with] pagesize n [k|K]]
```

For example, a placement file can contain the lines

```
memories 2
threads 2
distribute threads across memories
place range 0xffffbffc000 to 0xffffc000000 on memory 0
place range 0xffffd2e4000 to 0xffffda68000 on memory 1
```

Generally, placing address ranges is only practical when used in conjunction with `dprof(1)`. This utility profiles the memory accesses in a program and, when the `-pout dplace_file` option is used, writes address placement lines like those shown above to the output file `dplace_file` to indicate the optimal

placements for the address ranges in the program. These address placement lines may then be inserted directly into a placement file for the program. Since address ranges are likely to change after recompilation, this use of a placement file is provided for cases in which existing binaries needs their data placements fine-tuned.

back to top

---

#### 4.5.10.7 Dynamic Specifications with Dplace Library

In addition to the static capabilities described above, *dplace* can also dynamically move data and threads. These tasks are accomplished with the commands:

```
migrate range k to l to memory m
move thread|pid n to memory m
```

Although these commands may be included in a placement file, they are of most use when issued at strategic points during the execution of a program. This is done via the subroutine library interface *dplace(3)*. The interface is simple: only two functions are provided, `dplace_file()` and `dplace_line()`. `dplace_file()` takes as its argument the name of a placement file; this routine is used to issue multiple *dplace* commands. The argument to `dplace_line()` is a string containing a single *dplace* command.

Below is some sample code showing how the dynamic specifications are issued via the library calls:

```
CHARACTER*128 s

np = mp_numthreads()
WRITE(s,*) 'memories ',np,' in topology cluster'
CALL dplace_line(s)

WRITE(s,*) 'threads ',np
CALL dplace_line(s)

DO i=0, np-1
  WRITE(s,*) 'run thread',i,' on memory',i
  CALL dplace_line(s)
  head = %loc( a( 1+i*(n/np) ) )
  tail = %loc( a( (i+1)*(n/np) ) )
  WRITE(s,*) 'place range',head,' to',tail,' on memory',i
  CALL dplace_line(s)
END DO

DO i=0, np-1
  WRITE(s,*) 'move thread',i,' to memory',np-1-i
  CALL dplace_line(s)
END DO

DO i=0, np-1
  head = %loc( a( 1+i*(n/np) ) )
  tail = %loc( a( (i+1)*(n/np) ) )
  WRITE(s,*) 'migrate range',head,' to',tail,' to memory',np-1-i
  CALL dplace_line(s)
END DO
```

END DO

The library is linked in with the flag `-ldplace`.

back to top

---

#### 4.5.11 Data Placement Tuning for Non-MP library Programs

Message-passing programs are the key examples of properly parallelized programs requiring the use of *dplace*(1) to tune data placement. The Silicon Graphics implementations of MPI and PVM (from Array Services 2.0) do not take the physically distributed memory of Origin into account. As a result, the memory for important data structures can be allocated from a single node, which creates contention for that node. You can use *top*(1) or *gr\_osview*(1) to look for a symptom of this problem, namely, the processes jumping between processors rather than remaining on the same one. To solve this problem, use *dplace* (described above) to explicitly place the processes and memory of your program on the desired nodes.

Most MPI implementations (including Silicon Graphics') use `$MPI_NP+1` threads for a `$MPI_NP`-processor job. The first thread is mostly inactive and should not be placed, so a typical placement file for an MPI application is the following:

```
# scalable placement_file for MPI
# two threads per memory
memories ($MPI_NP + 1)/2 in topology cluster # set up memories
threads $MPI_NP + 1 # number of threads
distribute threads 1:$MPI_NP across memories # ignore the lazy thread
```

This example uses two threads per memory, with the memories arranged in a cluster topology to reduce the number of router hops in remote references. `$MPI_NP` threads are distributed across the memories, and the first thread is left unplaced since it is mostly inactive.

MPI jobs are launched with the run command *mpirun*(1). To use *dplace* with an MPI job, the *dplace* command needs to follow the *mpirun* command:

```
% mpirun -np <nprocs> /usr/sbin/dplace <dplace_args> a.out <a.out args>
% mpirun -p <machine> <nprocs> /usr/sbin/dplace <dplace_args> \
./a.out <a.out args>
```

And, as is shown, you need to use the full path to the *dplace* binary. If you want to profile an MPI run while *dplace* is being used, put the *ssrun* command before the *dplace* command:

```
% mpirun -np <nprocs> /bin/ssrun <ssrun_args> /usr/sbin/dplace \
<dplace_args> a.out <a.out args>
% mpirun -p <machine> <nprocs> /bin/ssrun <ssrun_args> \
/usr/sbin/dplace <dplace_args> ./a.out <a.out args>
```

In addition to MPI programs, *dplace* should also be used with PVM jobs, and, if there is a scaling problem, with programs in which you have created your own parallel threads using *sproc*(2) or *fork*(2).

*dplace* should *not* be used with MP library programs, however, since the MP library uses compiler

directives to control data placement and it supplies its own environment variables for controlling the various policy choices. Using *dplace* on MP library programs disables the data placement done by the MP library, so their combination is likely to produce unintended results.

Also note that you cannot use *dplace* to associate memory segments with particular POSIX threads. A program that uses the current version of *pthread*s has no control over which process executes each thread. While you can allocate the data needed by a pthread within that thread, this does not ensure that the data will be near the pthread at any particular moment of execution. This will be addressed in future implementations of pthreads. You can, however, use *dplace* to launch a pthread program for other reasons, such as specifying the page size.

back to top

---

#### 4.5.12 Advanced Options

The MP library and *dplace*(1) provide a couple of advanced features which may be of use for realtime programs. These are

- Running just one process per node
- Locking processes to processors

Neither of these features, however, is particularly useful in the typical multi-user environment in which you must share the computer with other users.

The first feature can be of use to programs which have high memory bandwidth requirements. Since one CPU can use more than half of a node's bandwidth, spreading threads of a job across more nodes can result in higher sustained bandwidth. Of course, you are then limited to half the processors. But if you have two jobs that you want to run concurrently, each on half the processors, running only one thread of a very memory intensive job per node allows it to achieve top performance at the same time another cache friendly job runs.

For the MP library, you can run just one thread per node by setting the environment variable `_DSM_PPM` to 1:

```
% setenv _DSM_PPM 1
```

For *dplace*, use a placement file specifying the same number of memories as threads. In the `distribute` statement, you can explicitly use a block of 1, but this will be the default when the number of threads is the same as the number of memories:

```
memories $NP in topology cluster
threads $NP
distribute theads across memories block 1
```

The second feature has been available in IRIX for years via the `sysmp(MP_MUSTRUN, ...)` system call. The MP library and *dplace* now offer very convenient access to this capability. For the MP library, you simply need to set the environment variable `_DSM_MUSTRUN` to lock the threads onto the processors

where they start execution:

```
% setenv _DSM_MUSTRUN
```

For *dplace*, use the `-mustrun` flag:

```
% dplace -mustrun -place placement_file program
```

Both the MP library and *dplace* lock the processes onto the CPUs they start on; this will change from run to run. If you want to lock them onto specific physical CPUs, you will need to use a placement file specifying a physical topology or call `sysmp(MP_MUSTRUN, . . .)` explicitly.

back to top

---