SYSTEM

Intel Paragon MPP: Getting the Most from All of the Nodes

Intel Paragon MPP: Getting the Most from All of the Nodes

Shannon Miller

The intel Paragon may no longer be considered state-of-the-art technology in terms of performance capabilities, but for a hybrid GP/MP system such as the one at RUS, it is possible to "stretch" the performance of the system by utilizing the dual-CPUs of the MP nodes at the same time as the single-CPU GP nodes. The Paragon at RUS can then calculate loops using up to 154 CPUs vs. the 113 compute CPUs which would otherwise be used. The trick is balancing the load; a process for which, in this case, no standard library exists.

When the 41 multi-processor (MP) nodes were added to the system last year, it was explained[1] how applications could be compiled with the `-Mconcur` compiler switch and executed on either GP (general purpose) or MP nodes, or both at once. The start-up code linked in with the executable would automatically determine whether the node on which it was running had one or two compute CPUs, and, in the latter case, would spawn off an extra thread so that the second CPU on MP nodes would participate in the work performed in loops.

This works great if all nodes running the application are homogeneous. If they are all GP nodes, each node computes the loops with one thread each, as though the program was *not* compiled with the `-Mconcur` switch; and if they are all MP nodes, each node utilizes two threads in parallel, providing up to twice the computing performance for loop-based calculations.

If, however, such an application is run on a combination of GP and MP nodes at the same time, little or no performance improvement will be seen for most parallel applications. This is because, in general, mathematical loops are evenly distributed among all the nodes. A calculation involving, say, one million iterations, may be evenly distributed to 100 nodes performing 10,000 iterations each. If the application was compiled for concurrent threads and run on both GP and MP nodes at once, the MP nodes would indeed complete their work twice as fast as the GP nodes. However, a global call which usually exists after the loop or at the end of the application would force the MP nodes to sit idle, waiting for the GP nodes to finish their work.

One way to overcome this "hurry up and wait" syndrome is to give the MP nodes twice as much work as the GP nodes, so that they'll finish at about the same time. How easily this can be done depends a great deal on the design of the application. In any case it cannot be done in a very portable way, so compiler preprocessor directives are required for applications which are intended to run on other platforms besides the intel Paragon system.

Giving MP Nodes More Work

A simple yet illustrative example is taken from the parallel pi calculation program provided with the Paragon in /usr/share/examples/fortran/pi. This program is commonly used to demonstrate parallel programming. It estimates the value of pi by using the n-point quadrature rule to evaluate the definite integral

$$\pi = \int_0^1 \frac{+}{(1+x^2)} dx$$

Listing 1: The Modified Program

```
 1  c  Modified example from Chapter 7 of the Paragon User's Guide
 2  c  Compile with:  if77 -nx -o pinode pinode.f mytype.o -Mconcur

 3        program pinode
 4        include `fnx.h'
 5        double precision h,sum,x,pi,f,a,tmp
 6        integer n, itmp
 7        integer nodes, iam, intsiz
 8  c Start, end, and total (elapsed) time
 9        double precision stime, etime, ttime
10  c Start and end part of a given node's "block" to compute
11        integer sblock, eblock
12        integer mynumcpus, totnumcpus

13        data intsiz / 4 /

14  c Define the function
15        f(a) = 4.0d0/(1.d0 + a*a)

16  c Do some bookkeeping
17        iam   = mynode()
18        nodes = numnodes()
19        mynumcpus = mytype()

20  c     n = number of intervals, we'll use 100 million for our tests
21        n = 100000000

22  c Calculate the scaling factor
23        h = 1.d0/n
24  c Find out how many CPUs in the entire partition are available
25        totnumcpus = mynumcpus
26        call gisum(totnumcpus,1,itmp)
27        if(iam.eq.0) then
28           print *,'There are a total of',totnumcpus,
29       &    ` CPUs available to work.'
30        endif

31  c Divvy up the workload based on individual node type.
32  c Since any node could be either GP or MP, we should do this in an
33  c incomplete round-robin fashion and just take turns allocating some work.
34  c In this case we let node 0 take it's chunk, and then pass a token
35  c around to the other nodes to take theirs.  The token is the ending
36  c iteration number that the previous node will be using.
37        if(iam .eq. 0) then
38           sblock = 1
39           eblock = dble(n)/totnumcpus*mynumcpus + 0.5
40           call csend(1,eblock,intsiz,1,0)
41        else
42           call crecv(1,sblock,intsiz)
43  c Protect against rounding errors affecting the final iteration assigned
```

```
44           if(iam .lt. (nodes - 1)) then
45              eblock = sblock + dble(n)/totnumcpus*mynumcpus + 0.5
46           else
47              eblock = n
48           endif
49           sblock = sblock + 1
50           if(iam .lt. (nodes - 1)) then
51              call csend(1,eblock,intsiz,iam+1,0)
52           endif
53        endif

54        print *,'Node',iam,' has',mynumcpus,' CPUs'
55        print *,'Node',iam,' has iterations',sblock,' to',eblock

56  c Integrate.  The value of x used to calculate the slice is
57  c the value at the midpoint of the integration slice.
58        sum = 0.d0
59        call gsync()
60        stime = dclock()
61  c Original method was striped and dispersed
62  c      do 10 i = iam+1,n,nodes
63  c New method "blocks" the data for easier distribution among
64  c a varied number of GP and MP nodes
65        do 10 i = sblock,eblock
66           x = h * (dble(i) - 0.5d0)
67           sum = sum + f(x)
68  10    continue
69        pi = h * sum
70        call gdsum(pi,1,tmp)
71        etime = dclock()
72        ttime = etime - stime

73        if(iam .eq. 0 )then
74  c Output the answer
75           print *,' The value of pi for',n,' intervals is',pi
76                           print *,' Time required for calculation is',ttime,' seconc
77        endif
78        end
```

Listing 2: Modified pi Calculation Program

The following changes were made to the original pi program:

- The interleaved striped domain decomposition was changed to a block striped domain decomposition
- An additional function `mytype()` was added which returns the number of CPUs available for computing on the node
- Loop timing and output code was added in order to observe the effect of the above changes

On 113 compute nodes, this modified version of the program performs the definite integral calculation 29% faster than the original program (the theorectical limit would be a 36% improvement, given 72 GP nodes and 41 MP nodes in the system). The cost of this additional improvement is with the loop setup: the simple "incomplete round-robin" method used here to divide up the work takes about 4 seconds on 113 nodes. Obviously the process is worthwhile only if the loop takes a relatively long time to compute, and there are few loops in the program (or a program with many loops is written such that the uneven load distribution can be performed a minimum number of times).

Determining the Number of Compute CPUs per Node

Since there is no standard way for an application to know which nodes are GP nodes (1 CPU) and which are MP (2 CPUs), the key to the above modification is the addition of the new routine

`mytype()`, which itself makes use of undocumented (and unsupported by intel) system calls. See listing 2.

```
 1  /* S. Miller, Compile with:  cc -c mytype.c  */
 2  /* To use:  link mytype.o with existing C or Fortran code. */
 3  #include <string.h>
 4  #include <nx.h>
 5  #include <malloc.h>
 6  #include <sys/types.h>
 7  #include <stdlib.h>
 8  /* C Interface */
 9  int mytype()        /* returns 1 for GP node, 2 for MP node */
10  {               /* -1 on error */
11      char **attrs;
12      char *ptr;
13      int nodetype = -1;
14      int iamphys;
15
16      iamphys = _myphysnode();
17      if(nx_node_attr (".",&attrs) == -1)
18              return(-1);
19      if((ptr = strstr(attrs[iamphys],"GP")) != NULL)
20              nodetype = 1;
21      else if((ptr = strstr(attrs[iamphys],"MP")) != NULL)
22              nodetype = 2;


23      else    nodetype = -1;
24      free(attrs);
25      return(nodetype);
26  }
27  /* Fortran interface to above */
28  int mytype_()
29  {
30      return(mytype());
31  }
```

Listing 3: Function to Return Available # of CPUs on the Node

To keep the example simple, only the `mytype()` function was written, but any similar function could be written to, for example, return an array listing the number of CPUs on each node assigned to the application (hint: use the standard `nx_app_nodes()` call in combination with the above). A worker/manager-modelled program could use such a function instead of communicating the information via message-passing.

The modified pi program can not know on which node types it will run. If it's given a 4-node partition, all 4 nodes might be GP nodes, or all MP nodes, or some combination of both in any order. Thus, the nodes must first determine how many CPUs are available altogether. This is done by each node calling the new mytype() function and then performing a global sum operation (lines 19 and 26 in listing 1). On the RUS 113-node system, the sum will be 154 CPUs. This number is then used to determine how much work each CPU should perform (amount of work divided by 154).

Node 0 in the application then allocates an appropriate amount of work for itself, taking twice as much if it's an MP node than if it's only a GP node. It then transmits it's "last iteration number assignment" to node 1. Node 1 assigns itself the next iteration number up through an interation number appropriate for itself (again, based on whether it's a GP or MP node). This process is repeated until all nodes have determined the iteration counts on which they will work. See lines 31 - 53 in listing 1. Since rounding errors could cause minor descrepancies in the iteration numbers, special assurances are made: node 0 starts with iteration 1, the last node ends with the last iteration, and all nodes in-between ensure that no iterations are skipped or duplicated.

Using MPI Instead of NX Message-Passing Calls

The example program utilizes native Paragon "NX" message-passing calls. The same technique described in this article may also be applied to an MPI program. However some portability is lost because the call to `mytype()` is still necessary. The `MPI_Get_processor_name()` call returns useful information about the given node on the Paragon, but unfortunately it does not return the node *type* (MP vs. GP) or number of compute CPUs on the node.

Cautions

Be certain that the loops in the parallel application optimize correctly when the compiler switch `-Mconcur` is used. A single subroutine call inside the loop can cause the inliner to utilize only one instead of two CPUs on an MP node. See the appropriate Paragon Compiler User's Guide[3, 4, 5] for details.

Likewise, if the environment variable `$DFLT_NCPUS` is set to 1, only one CPU will be used on each node, including the MP nodes. In this case the optimizations described in this article will yield a *slower* run time than if they were not at all implemented. Usually it's easiest to simply leave this environment variable undefined.

Applicability to Real-World Applications

For many problems, it may not be worth the effort to use MP nodes (both CPUs) and GP nodes simultaneously. The loop(s) may be too complex or have loop dependancies which make the load-balancing process difficult or impossible.

For a program such as the example provided here, which uses domain decomposition, the load balances well when the amount of work for each iteration through the loop is approximately equal. Interleaving stripes among the nodes, as in the original pi example, is much better if the problem to be solved is not balanced to begin with (see [2], page 7-4 for an illustration).

If control decomposition is used, the concept is the same. The master node should determine how many CPUs each worker node has available, and assign those workers which are MP nodes twice as much work (or whatever is appropriate based on how much faster the loop can be calculated by an MP node vs. a GP node). If a handshaking model is used where worker nodes come to the master for more work when each task is completed, the load balancing is automatic (since the MP nodes will check back with the master more often than the GP nodes will).

References

[1] Miller, S., Sihling, D., Geiger, A.: intel Paragon MPP: System wurde verdoppelt, BI. 10-12 1996, p. 8.
(PostScript: `paragon:/usr/local/share/bi-mp.ps`)

[2] Paragon System User's Guide, Order Number 312489.
(PostScript: `paragon:/usr/share/ps.docs/psug.ps`)

[3] Paragon System Fortran Compiler User's Guide, Order Number 312491.
(PostScript: `paragon:/usr/share/ps.docs/pfug.ps`)

[4] Paragon System C Compiler User's Guide, April 1996, Order Number 312490.
(PostScript: `paragon:/usr/share/ps.docs/pcug.ps`)

[5] Paragon System C++ Compiler User's Guide, April 1996, Order Number 313151.
(PostScript: `paragon:/usr/share/ps.docs/pcppug.ps`)

Shannon Miller, NA-5744
E-Mail: smiller@rus.uni-stuttgart.de or smiller@co.intel.com