Synchronization: Coordination at What Cost?

Sharon Brunett
Center for Advanced Computing Research
California Institute of Technology, Pasadena California
Pasadena, California
626 395 6734 , 626 584 5917 (fax)
sharoncacr.caltech.edu

March 2001

1 Introduction

A concurrent program relies heavily on shared components to enable its processes to work in unison. Processes working together to solve a problem must share information, or communicate. However, communication requires synchronization in order to assure shared data required by dependent parallel processes is accessed independently. One method for synchronizing processes and data access is through barriers.

OpenMP is an interface which defines compiler directives and library routines which allows a programmer to tell the compiler what parts of the program to execute concurrently, and where to place barrier synchronization points. Although barriers are necessary for desired behavior of a parallel code, they induce overhead as processes wait for one another. Further, the barrier operation itself has a cost, which is dependent upon the OpenMP run-time library implementation and execution platform.

This paper discusses: a simple technique for measuring overhead of the barrier synchronization directive, timings for the the barrier directive on the Hewlett- Packard Superdome; and the impact of the observed scaling behavior.

2 Overhead

2.1 What is Overhead?

The overhead of a parallel program is often written as $T_p - T_s/p$, where T_p is the time taken for a parallel program to run on p processors and T_s is the time taken for the same code to run sequentially. If a code scaled perfectly, we would see linear speed up for a fixed problem size as p increases. At some point, the problem size will become too small for large values of p. However, before that point we often see the overhead of process syncrhonization effect scalability.

2.2 What's an Easy Way to Measure Overhead?

We need two things to measure overhead. First, the time taken for code to execute in parallel. Second, the time taken for the same code to execute sequentially. In many scientific applications, there's code segments which must be executed sequentially, and sections one

hopes to parallelize. It makes sense to time code sections of interest, rather than just entire application execution time. After all, the overheads may be drastically different for various concurrent sections of an application.

To obtain sequential timings, identify a section of code which is to ultimately run in parallel, and measure its sequential execution time. Next, measure the time taken for the same section of code to execute within the specified OpenMP parallel directive. Comparing the two times yields the overhead of the given directive. The following example shows a method of measuring the sequential and parallel execution time for a given section of code, specifically including the OpenMP barrier directive:

```
  #define MICRO_SEC 1.0e6

  /* Calculate time taken to execute loop on 1 thread,
     weighted by the number of directives executed (FINE_GRAIN_LOOP_MAX) */

  for (i=0; i<=COARSE_GRAIN_LOOP_MAX; i++){
    start  = high_res_clock();
    for (j=0; j<FINE_GRAIN_LOOP_MAX; j++){
      do_nothing_routine(delay_val);
    }
    seq_time[i] = (high_res_clock() - start) * MICRO_SEC / (double) FINE_GRAIN_LOOP_MAX;
  }



  /* Calculate time taken to execute same loop as above, but multiple threads
     will be enabled due to OpenMP pragmas */

  for (i=0; i<=COARSE_GRAIN_LOOP_MAX; i++){
    start  = high_res_clock();

  /* OpenMP directive to declare each thread gets a private copy of j */
 #pragma omp parallel private(j)
    {
      for (j=0; j<FINE_GRAIN_LOOP_MAX; j++){
do_nothing_routine(delay_val);

  /* OpenMP directive to synchronize all threads. All threads must reach this point
     before execution continues  */

 #pragma omp barrier
      } /* for j*/
    }
    par_time[i] = (high_res_clock() - start) * MICRO_SEC / (double) FINE_GRAIN_LOOP_MAX;
  } /* for i */
```

The Overhead for the barrier call is obtained by calculating the difference between par_time[ ] and seq_time[ ].

2.3 Results

Barrier synchronization timings were run on the Hewlett Packard Superdome, with 32 550 MHz PA8600 CPUs. Test codes were compiled with the KAI Guide 3.9 compiler. In Table 1 below, overheads for the barrier directive, as a function of CPU (one thread/process per CPU) count are listed. Notice the dramatic increase in overhead, as we increase beyond ten CPUs! In fact, scaling was so poor, for 32 CPUs a slightly smaller number of CPUs/threads were measured, to help take into account any system daemons running. These runs were conducted in dedicated mode, implying typical production runs will probably incur higher overhead when competing with other jobs for resources.

Often times, barriers are implemented in two stages. The first phase consists of all slave threads checking in with a designated master thread. When all the slaves have checked in, a signal is sent from the master to the slaves, indicating the barrier directive has completed and each thread of execution participating in the barrier may proceed. It's not clear what aspect of the barrier implementation is responsible for the serious scaling problems. Further measurements, using different synchronization calls, may help isolate causes for poor performance as well as point to alternative directives for minimal overhead for given CPU counts. Made perfectly clear in the table below, barriers involving a substantial portion of the 32 CPU Superdome incur quite an overhead cost. Applications which need to scale well on the Superdome, will most certainly find barriers to be a severe bottleneck.

Threads	Barrier Overhead (micro sec)
32	23564
30	19430
16	4340
10	1145
8	629
4	85
3	55
2	27
1	2

Table 1. Barrier Overhead on HP Superdome

3 Conclusions

This paper presented a simple techniques for measuring overhead of barrier synchronization directives. Timings for the Hewlett- Packard Superdome are shown, along with discussion regarding the impact of barrier overhead when running on more than a few cpus.

Future work includes expanding the collection of platforms for evaluation, as well as the breadth of synchronization directives. For directives which perform poorly, further exploration into the implementation details will be undertaken, in hopes that the vendors will

consider modifying their approach or at least applications can be written or modified to avoid particularly poor performing synchronization methods.