Barrier Synchronization

In this project, you will evaluate several barrier synchronization algorithms presented in "Algorithms for Scalable Synchronization on Shared-Memory Multiproce by Mellor-Crummey and Scott. To do this, you will implement some of them using OpenMP and MPI, standard frameworks for parallel programming in shared me and distributed architectures, and then evaluate some experimental results.

Contents

Setup

1. Gain access to a well-maintained linux machine. If you do not have this already, then it is recommended that you follow the instructions for downloading a installing VirtualBox and the AOS Virtual Image.
2. Download the starter source code barrier.tar.gz
3. Check that OpenMP is installed with

   ldconfig -p | grep libgomp This should show the location fo the libgomp library.
4. Check that MPI is installed by confirming that mpicc and mpirun are in your path with

   which mpicc mpirun

Collaboration

For this and all projects, students must work on their own.

Programming for a Shared Memory Multiprocessor with OpenMP

OpenMP uses #pragma directives to direct the compiler to create a multithreaded program. A simple OpenMP program using barrier would look like this

```
int main(int argc, char** argv){
  int num_threads = 5;

  /* Prevents runtime from adjusting the number of threads. */
  omp_set_dynamic(0);

  /* Making sure that it worked. */
  if (omp_get_dynamic())
    printf("Warning: dynamic adjustment of threads has been set\n");

  /* Setting number of threads */
  omp_set_num_threads(num_threads);

  /* Code in the block following the #pragma directive is run in parallel */
#pragma omp parallel
   {

     /*Some code before barrier ..... */

     /* The barrier*/
     #pragma omp barrier

     /*Some code after the barrier..... */

   }

   return(0);
}
```

To replace OpenMP's barrier algorithm with our own, we define the following API:

```
/*This function should be called before the parallel section of the code.  It performs all necessary memory allocation and initialization necessary for
rrier with num_threads threads.*/
gtmp_init(int num_threads) -

/* This function causes the calling thread to wait until all other threads have reached the same barrier. */
gtmp_barrier()

/* This function performs any cleanup needed for the barrier. */
gtmp_finalize()
```

The above OpenMP program becomes the following when modified to use the gtmp barrier system

```
int main(int argc, char** argv){
  int num_threads = 5;

  /* Prevents runtime from adjusting the number of threads. */
  omp_set_dynamic(0);

  /* Making sure that it worked. */
  if (omp_get_dynamic())
    printf("Warning: dynamic adjustment of threads has been set\n");

  /* Setting number of threads */
  omp_set_num_threads(num_threads);

gtmp_init(num_threads);
  /* Code in the block following the #pragma directive is run in parallel */
#pragma omp parallel
   {

    /*Some code before barrier ..... */

    /* The barrier, instead of #pragma barrier*/
    gtmp_barrier();

    /*Some code after the barrier..... */

   }

  gtmp_finalize();

   return(0);
}
```

The code for the project uses OpenMP for the shared memory experiments, but the code on which you will be evaluated can be written entirely in C. You may fin
useful, however, to write some test code using OpenMP to help you debug your program. This is also an opportunity to learn about OpenMP more broadly. Here
some good resources.

- Wikipedia on OpenMP
- Official OpenMP site
- Using OpenMP and Examples (zip file)
- Atomic Built-ins

Your tasks for this part of the project are to:

1. Complete the implementation of the counter barrier in gtmp_counter.c
2. Complete the implementation of the MCS tree in gtmp_mcs.c
3. Improve the performance of the tree barrier implementation in gtmp_tree.c and explain why your implementation will run faster in the comments.
4. Reply to the email in the file Re:_OpenMP_Barrier.txt.

Distributed Memory with MPI

Although Mellor-Crummey and Scott describe algorithms for a shared memory environment, it is straightforward to adapt them to a distributed one. Instead of spinning on a variable waiting for its value to be changed, a thread waits for a message from another thread.

The standard framework for message passing in high performance parallel computing is MPI. More specifically, we will use the openmpi implementation of MPI-simple MPI program using barrier would look like this

```
int main(int argc, char** argv)
{
  int req_processes = 5;
  int num_processes;

  MPI_Init(&argc, &argv);

  /* Start of Parallel...*/

  /* Making sure that we got the correct number of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
  if(num_processes != req_processes){
    fprintf(stderr, "Wrong number of processes.  Required: %d, Actual: %d\n",
        req_processes, num_processes);
    exit(1);
  }

/* Code before barrier ...*/

  /* The barrier */
  MPI_Barrier(MPI_COMM_WORLD);

/* Code after barrier ...*/

  /* Cleanup */
  MPI_Finalize();

  return(0);
}
```

The API for your barrier system will be the same as for the first part of the project with the functions gtmpi_init(int num_threads), gtmpi_barrier(), and gtmpi_fin all having the same semantics.

Replacing the built-in MPI barrier with the gtmpi version, we have

```
int main(int argc, char** argv)
{
  int req_processes = 5;
  int num_processes;

  gtmpi_init(req_processes);
  MPI_Init(&argc, &argv);

  /* Start of Parallel...*/

  /* Making sure that we got the correct number of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
  if(num_processes != req_processes){
    fprintf(stderr, "Wrong number of processes.  Required: %d, Actual: %d\n",
        req_processes, num_processes);
    exit(1);
  }
/* Code before barrier ...*/

  /* The barrier */
  gtmpi_barrier();

/* Code after barrier ...*/

  /* Cleanup */
  MPI_Finalize();
  gtmpi_finalize();

  return(0);
}
```

As you program, think about which messages passing functions should be synchronous and which asynchronous. Some useful documentation on MPI can be fou
these resources:

- Wikipedia on MPI
- Tutorial

Tasks

Your tasks are the following:

1. Complete the implementation of the dissemination algorithm in gtmpi_dissemination.c.
2. Complete the implementation of the tournament algorithm in gtmpi_tournament.c.
3. Improve the implementation of the counter in gtmpi_counter.c.
4. Reply to the email Re:_MPI_Barrier.txt.

Note that your should only use MPI's point-to-point communication procedures (i.e. MPI_Isend, MPI_Irecv, MPI_Send, MPI_Recv, NOT MPI_BCast, MPI_Gather, et

Grading

Deliverable	Percentage
gtmp_counter.c	15%
gtmp_mcs.c	15%
gtmp_tree.c	10%
Re:_OpenMP_Barrier.txt	10%
gtmpi_dissemination.c	15%
gtmpi_tournament.c	15%
gtmpi_counter.c	10%
Re:_MPI_Barrier.txt	10%

This page was last edited on 2014/04/25 08:51:13.