CS403 Parallel Programming
Lab01 - Introduction

Topics covered:

1. Peak Performance
2. Basic Linux commands
3. Introduction to POSIX threads

Peak Performance

Whenever you look at performance, it's useful to understand what you're expecting

- Machine peak is the maximum number of arithmetic operations a processor (or set of them) can perform
- Treat integer and floating point separately and be clear on how wide (# bits) the operands have
- If you see a number higher than peak, you have a mistake in your timers, formulas, or …

Example: Power5 processors in Bassiat NERSC

- 1.9 GHz with 4 double precision floating point ops/cycle 7.6 GFlop/s peak per processor
- Two fpunits, each can do a fused MADD (multiply-add)

Problems:

1. Consider a memory system with a level 1 cache of 32 KB and DRAM of 512 MB with the processor operating at 1 GHz. The latency to L1 cache is 1 cycle and the latency to DRAM is 100 cycles. In each memory cycle, the processor fetches four words (cache line size is 4 words). What is the peak achievable performance of a dot product of two vectors?

```
/* dot product loop */
for (i = 0; i < dim; i++)
     dot_prod += a[i] * b[i];
```

2. Now consider the problem of multiplying a dense matrix with a vector using a two-loop dot-product formulation. The matrix is of dimension 4K x 4K. (Each row of the matrix takes 16 KB of storage.) What is the peak achievable performance of this technique using a two-loop dot-product based matrix-vector product?

```
/* matrix-vector product loop */
for (i = 0; i < dim; i++)
     for (j = 0; j < dim; j++)
          c[i] += a[i][j] * b[i];
```

Linux commands

top: provides dynamic real-time view of individual jobs running on the system.

gnome-system-monitor: shows which programs are running and how much processor time, memory, and disk space are being used. This gives an overall system view whereas the "top" instruction represents a detailed perspective.

lscpu: display information about the cpu architecture

You can also get the same information from “`cat /proc/cpuinfo`”.

time: Get total program execution time in the shell.
In the output, *real* means "wall-clock time", while *user* and *sys* show CPU clock time, split between regular code and system calls.

Usage:

```
$top
$gnome-system-monitor
$lscpu
$cat /proc/cpuinfo
$time ./a.out
```

Introduction to POSIX threads

In shared memory multiprocessor architectures, threads can be used to implement parallelism. Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads.

To compile:

```
gcc -pthread -o hello myhello.c
```

Sample codes:

```
/****************************************************************************
**
* FILE: hello.c
* DESCRIPTION:
*   A "hello world" Pthreads program.  Demonstrates thread creation and
*   termination.
* AUTHOR: Blaise Barney
* LAST REVISED: 08/09/11
****************************************************************************
*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS    5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}

/****************************************************************************
*
* FILE: join.c
* DESCRIPTION:
*   This example demonstrates how to "wait" for thread completions by using
*   the Pthread join routine.  Threads are explicitly created in a joinable
*   state for portability reasons. Use of the pthread_exit status argument is
*   also shown. Compare to detached.c
* AUTHOR: 8/98 Blaise Barney
* LAST REVISED:  01/30/09
*****************************************************************************
*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS    4

void *BusyWork(void *t)
{
   int i;
   long tid;
   double result=0.0;
   tid = (long)t;
   printf("Thread %ld starting...\n",tid);
   for (i=0; i<1000000; i++)
   {
      result = result + sin(i) * tan(i);
   }
   printf("Thread %ld done. Result = %e\n",tid, result);
   pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc;
   long t;
   void *status;

   /* Initialize and set thread detached attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
      if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
         }
      }

   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join() is %d\n", rc);
         exit(-1);
         }
      printf("Main: completed join with thread %ld having a status of
%ld\n",t,(long)status);
      }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

Sample codes are available in:
https://computing.llnl.gov/tutorials/pthreads/samples/hello.c
https://computing.llnl.gov/tutorials/pthreads/samples/hello_arg1.c
https://computing.llnl.gov/tutorials/pthreads/samples/join.c
https://computing.llnl.gov/tutorials/pthreads/samples/detached.c

Lab problems:

1. Familiarize yourself with the Linux commands and POSIX thread code given in this handout.
2. Solutions for Problems 1-2 on peak performance.
3. Using the basic Linux commands find the cache size, bandwidth number of processors on your system.
4. Write a C-code using POSIX threads to create an unbalanced load using `sleep` command and `hello.c`. The sample sleep times are given below:
    a. thread-1: 1000 sec, thread-2: 5000 sec, thread-3: 20 sec, thread-4: 1200 sec.
    b. Measure the total time taken for the complete execution of code with and without the additional sleep command.

    Hint: You will require to include `unistd.h` for successful compilation.
5. Write a C-code using POSIX threads about matrix multiplication.
    a. Take overall execution time measurement using `time` command for different application size and thread count for the serial and parallel code.
    b. Observe `gnome-system-monitor` output as your fire up different thread counts.
    c. Use the overall execution time measurements to plot and comment upon the speed-up.

Lab-report:

1. Numerical problem and solution
2. Programming problem
    - Objective or summary of problem statement
    - Pseudo-code
    - Measurements / results
    - Conclusion

NOTE: (1) In your lab-report, attach screenshots whenever necessary.
(2) For the programming questions, include your pseudo-code (in lab-report) and C-code *separately*.