CPS311 - COMPUTER ORGANIZATION

The Same Program Written Sequentially, and Using Various Parallelization Strategies

```
/* This program counts the total number of factors of a given number, starting at 1
 * and up to but not including the number itself.   The method used is to simply try
 * each possible divisor and count the ones that actually are - not a particularly
 * elegant approach - but makes for a good demonstration of various parallelization
 * strategies.
 *
 * $Smake: g++ wtime.o -lrt -o %F %f
 * Copyright (c) 2013, 2019 - Russell C. Bjork
 */

/* SEQUENTIAL VERSION - ALL PROCESSING IS DONE BY THE CPU */

using namespace std;
#include <iostream>
#include <iomanip>
#include "wtime.h"

unsigned long number;
unsigned int factorCount;

/* Count the factors of a number within a certain range.
 *
 * Parameters:      number - the number to factor
 *                  lo - the first number in the range
 *                  hi - the first number _not_ in the range - i.e. the factors
 *                         are in the range [ lo .. hi - 1 ]
 * Returns:         the count of the factors of numbers in this range
 */
int countFactors(long number, long lo, long hi)
{
       int count = 0;
       for (long i = lo; i < hi; i ++)
              if (number % i == 0)
                     count ++;
       return count;
}

int main()
{
       cout << "Number for which to count factors? ";
       cin >> number;
       if (cin.good())
       {
              double start = wtime();    // Get the time when computation of factors started
              int count = countFactors(number, 1, number);
              double end = wtime();      // Get the time when computation of factors finished
              cout << number << " has " << count << " factor(s) less than itself." << endl;
              cout << "Computation took " << setprecision(4) << end-start << " seconds." << endl;
              return 0;
       }
       else
       {
              cerr << "Malformed number" << endl;
              return 1;
       }
}
```

(Only main program and changes compared to the sequential version are shown. countFactors() is the same as sequential)

```
/* This program counts the total number of factors of a given number, starting at 1
 * and up to but not including the number itself.   This program divides the task into
 * two parts, each of which is run as a separate thread on a 2 or more-core CPU,
 * Note that this demo uses only 2 cores even if the CPU has more!
 *
 * COMPUTATION PARALLELIZED USING PTHREADS */
....
#include <pthread.h>
....
// Data and code for the threads
struct thread_data
{
    pthread_t tid;
    long lo, hi;
    int count;
} threadData1, threadData2;
void * threadCode(void * arg)
{
    thread_data * data = (thread_data *) arg;
    data -> count = countFactors(number, data -> lo, data -> hi);
}
....
int main() {
       cout << "Number for which to count factors? ";
       cin >> number;
       if (cin.good())
       {
              double start = wtime();    // Get the time when computation of factors started
              // Create the data that will be used by the two threads
              threadData1.lo = 1; threadData1.hi = number/2;
              threadData2.lo = number/2; threadData2.hi = number;
              // Start two threads
              pthread_create(& threadData1.tid, NULL, threadCode, & threadData1);
              pthread_create(& threadData2.tid, NULL, threadCode, & threadData2);
              // Wait for both to complete
              pthread_join(threadData1.tid, NULL);
              pthread_join(threadData2.tid, NULL);
              // Combine counts calculated by the two threads
              int count = threadData1.count + threadData2.count;
              double end = wtime();      // Get the time when computation of factors finished
              cout << number << " has " << count << " factor(s) less than itself." << endl;
              cout << "Computation took " << setprecision(4) << end-start << " seconds." << endl;
              return 0;
       }
       else {
              cerr << "Malformed number" << endl;
              return 1;
       }
}
```

Compilation command on linux is:

```
g++ -pthread wtime.o -lrt -o countFactors_pthreads countFactors_pthreads.cc
```

```
/* COMPUTATION PARALLELIZED USING OMP */
```

(Only countFactors() and changes compared to the sequential version are shown. Main program is the same as sequential)

```
....

#include <omp.h>

....

/* Count the factors of a number within a certain range.
 *
 * Parameters:      number - the number to factor
 *                         lo - the first number in the range
 *                         hi - the first number _not_ in the range - i.e. the factors
 *                                 in the range [ lo .. hi - 1 ]
 * Returns:         the count of the factors of numbers in this range
 */
int countFactors(long number, long lo, long hi)
{
       int count = 0;
       #pragma omp parallel for default(shared) reduction(+:count)
       for (long i = lo; i < hi; i ++)
              if (number % i == 0)
                     count ++;
       return count;
}
```

Compilation command on linux is:

```
g++ -fopenmp wtime.o -lrt -o countFactors_omp countFactors_omp.cc
```

```
/* This program counts the total number of factors of a given number, starting at 1
 * and up to but not including the number itself.
 *
 * $Smake: nvcc wtime.o -lrt -o %F %f
 *
 * Copyright (c) 2013, 2015 - Russell C. Bjork
 */

/* COMPUTATION PARALLELIZED USING CUDA */

using namespace std;
#include <iostream>
#include <iomanip>
#include <stdio.h>
#include <cuda_runtime.h>
#include "wtime.h"

unsigned long number;
unsigned int factorCount;

// The number of threads that will be used
#define THREADS 1024

/* This function is executed by the GPU's.  It count the factors of a number within a
 * subrange (designated by local variables lo and hi, where lo is the first value in
 * the subrange to be considered and hi is the first value _not_ to be considered.)
 * Each thread calculates its lo and hi values from its index.   The partial counts are
 * stored in an array on the device and are then copied back to the CPU and summed to
 * the final answer.
 *
 * Parameter: number - the number whose factors are being counted
 * Parameter: partialCount - the array of partial counts - each thread fills in the
 *            value corresponding to its index
 *
 */
__global__
void countFactors(unsigned long number, unsigned int * partialCount)
{
    unsigned long divisorsPerThread = (long) ceil(((double) number) / THREADS);
    unsigned long lo = (unsigned long) (threadIdx.x * divisorsPerThread);
    unsigned long hi = lo + divisorsPerThread;
    if (lo == 0)
        lo = 1;
    if (hi > number)
        hi = number;

    unsigned int threadPartialCount = 0;
    for (long i = lo; i < hi; i ++)
        if (number % i == 0)
            threadPartialCount ++;

    // Save the partial count found by this thread in the array on the device
    partialCount[threadIdx.x] = threadPartialCount;
}
```

```
int main()
{
    unsigned long number;
    cout << "Number for which to count factors? ";
    cin >> number;
    if (cin.good())
    {
        // Note the time at which processing started
        double start = wtime();

        // Variable to hold status of GPU operations.
        cudaError_t err = cudaSuccess;

        // Allocate memory on the device for the partial counts
        unsigned int * d_partialCount = NULL;
        err = cudaMalloc((void **) & d_partialCount, THREADS * sizeof(unsigned int));
        if (err != cudaSuccess)
        {
            fprintf(stderr, "Failed to allocate array on device (error code %s): \n",
                   cudaGetErrorString(err));
            exit(1);
        }

        // Start the parallel kernels on the GPU
        countFactors <<<1, THREADS >>>(number, d_partialCount);

        // Copy result back from GPU when all threads have finished
        unsigned int partialCount[THREADS];
        err = cudaMemcpy(partialCount, d_partialCount, THREADS * sizeof(unsigned int),
                         cudaMemcpyDeviceToHost);
        if (err != cudaSuccess)
        {
            fprintf(stderr, "Failed to copy array from device (error code %s): \n",
                   cudaGetErrorString(err));
            exit(1);
        }

        // Reduce by summing the partial counts
        factorCount = 0;
        for (int i = 0; i < THREADS; i ++)
            factorCount += partialCount[i];

        // Note the time when processing completed
        double end = wtime();

        // Output the results
        cout << number << " has " << factorCount << " factor(s) less than itself." << endl;
        cout << "Computation took " << setprecision(4) << end-start << " seconds." << endl;
        return 0;
    }
    else
    {
        cerr << "Malformed number" << endl;
        return 1;
    }
}
```

Compilation command on linux is:

```
nvcc wtime.o -lrt -o countFactors_cuda countFactors_cuda.cu
```