Project -- Tour d'Algorithms: OpenMP Qualifier

Author: Mark McKenney
Organization: CS 516, Department of Computer Science, SIUE

Overview:

In this assignment you will investigate parallel programming through sorting algorithms using OpneMP. OpenMP is probably the easiest method to achieve a degugged parallel program, and has the potential for very good performance. If you have never used OpenMP, check the following:

1. The OpenMP wiki. It has lots of practical examples.
2. The OpenMP website. It has the full documentation.
3. The CyberGIS module on my website (under teaching). It has lots of explanations and working code examples.

Because of the trend to multi/many core architectures, performance scalability on new hardware requires the exploitation of parallel hardware. This assignment will let you practice the basics.

Description:

Code:

You will investigate three sorting algorithms.

1. Bubble/Bucket sort (any $O(n^2)$ sort algorithm used in conjunction with bucket sort)
2. Quick sort (expected $O(n \lg n)$)
3. Merge sort (worst case $O(n \lg n)$)

For **EACH** algorithm you will create 2 solutions, totaling in **6 solutions** (plus 1 additional solution described below). A **seperate** program the compiles to an individual executable is required for **EACH SOLOUTION**.

Each solution must take command line arguments in the following form:

```
[executable name] [number of random integers to generate]
[seed value for random number generation]
```

I will provide a skeleton file with a function to generate an array of random numbers that you will sort.

The exectuable names should be the following (where *s* indicates serial and *p* indicates parallel):

- `bbs`
- `bbp`
- `qss`
- `qsp`
- `mss`
- `msp`

You will also provide 1 solution that uses the STL sort algorithm for reference. This solution uses the same command line options of the others. It must be named:

- `reference`

Report:

You must turn in a report with graphs that depict the running times of your serial, parallel, and reference algorithms plotted together for varying sizes of arrays. You should use enough numbers in the arrays to generate times long enough to make sense out of. You will have 3 graphs, one for `bbs`, `bbp`, and `reference`, another for `qss`, `qsp`, and `refernce`, etc. Therefore, your report will have at least 3 sections, one for each graph and its associated explanation.

Your report MUST explain why the graphs are behaving as they are. It is OK if the parallel algorithm is actually slower than the serial algorithm, but you must explain why. Your explanation must be based in computer architecture (i.e., memory access/behavior, CPU architectural issues, OS overhead, etc).

Your report MUST explain the effects of hyperthreading. You should run your experiments with Hyperthreading TURNED OFF. Turn it on for one run of one algorithm and explain its effects on execution time.

Deliverables:

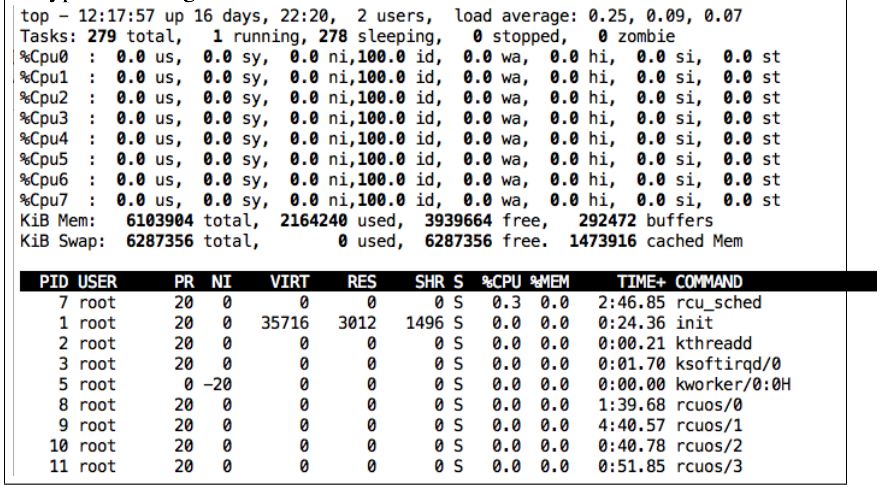
A single ZIP file containing:

1. A README file in **plain text** containing your group information and anything I need to know about running your program.
2. A Makefile with a default target that will build ALL of your executables on the *home.cs.siue.edu* server.
3. The source code for your 7 solutions. Your executables should print the running time at the end!
4. A **PDF** containing your report, with at least 4 sections (1 for each graph and a hyperthreading discussion) each containing a graph of a class of algorithms and the explanation for the behavior shown in the graph and reasoning about why that explanation is correct. The final section discusses hyperthreading. For the graphs, you may run on *home.cs.siue.edu*. For the hyperthreading discussion. you must run on a processor with hyperthreading. If you own such a processor, that is sufficient. If not, you may

use *jalapeno.cs.siue.edu*. Jalapeno has 8 hyperthreaded cores (resulting in 16 logical cores). Use the `top` command to verify that hyperthreading is turned on before you run your program. To verify, start the `top` program by typing `top` on the command line; then hit the `1` key, and the display will show all cores and their current load. Make sure you see 16 for hyperthreading. If you see 8, email me.

Ouput of `top` on jalapeno

After hitting the `1` key, 8 cores are shown. `top` will show 16 cores on jalapeno if hyperthreading is turned on.



```
top - 12:17:57 up 16 days, 22:20,  2 users,  load average: 0.25, 0.09, 0.07
Tasks: 279 total,   1 running, 278 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   6103904 total,  2164240 used,  3939664 free,   292472 buffers
KiB Swap:  6287356 total,        0 used,  6287356 free.  1473916 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    7 root      20   0       0      0      0 S   0.3  0.0   2:46.85 rcu_sched
    1 root      20   0   35716   3012   1496 S   0.0  0.0   0:24.36 init
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.21 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:01.70 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    8 root      20   0       0      0      0 S   0.0  0.0   1:39.68 rcuos/0
    9 root      20   0       0      0      0 S   0.0  0.0   4:40.57 rcuos/1
   10 root      20   0       0      0      0 S   0.0  0.0   0:40.78 rcuos/2
   11 root      20   0       0      0      0 S   0.0  0.0   0:51.85 rcuos/3
```

Grading

You will receive a 0 if you use a RAR format, do not have a makefile, do nnot submit a PDF, or do not have a README in plain text. Also, do not put any file extension on the README (e.g., `.txt`).

10% All deliverables provided in the correct format.
50% Source code compiles and executes without errors.
10% Graph 1 and explanation 1.
10% Graph 2 and explanation 2.
10% Graph 3 and explanation 3.
10% Hyperthreading execution times and explanation.

Skeleton File

You must use the following skeleton file. If you do not use it, be sure your code executes (with command line args) **EXACTLY** as this does.

```
#include <iostream>
#include <sstream>
#include <cstdlib>

// create an array of length size of random numbers
// returns a pointer to the array
// seed: seeds the random number generator
int * randNumArray( const int size, const int seed ) {

    srand( seed );
    int * array = new int[ size ];
    for( int i = 0; i < size; i ++ ) {
        array[i] = std::rand() % 1000000;
    }
    return array;
}

int main( int argc, char** argv ) {

    int * array;  // the poitner to the array of rands
    int size, seed; // values for the size of the array
                    // and the seed for generating
                    // random numbers

    // check the command line args
    if( argc < 3 ){
        std::cerr << "usage: "
            << argv[0]
            << " [amount of random nums to generate] [seed value for rand]"
            << std::endl;
        exit( -1 );
    }

    // convert cstrings to ints
    {
        std::stringstream ss1( argv[1] );
        ss1 >> size;
    }
    {
        std::stringstream ss1( argv[2] );
        ss1 >> seed;
    }

    // get the random numbers
    array = randNumArray( size, seed );

    // **************************
    // **************************
    // **************************
    //
    //  YOUR CODE HERE !!!
    //
    // **************************
    // **************************
    // **************************
    // delete the heap memory
    delete [] array;

}
```