**Operating Systems – CS2006**

**Parallel Programming Comparison of Sorting Algorithms**

**Group Members**

Ali Nadir 20K0325

Imtiaz Ali 20K0313

Ahmed Abdullah 20K0470



**Teacher Name: Miss Safia Baloch**

**Date: Thursday, May 26, 2022**

**Background and Motivation**

The project was chosen on the sole basis to recognize and appreciate the parallelization of sorting algorithms, namely Merge sort, Quick sort, and Radix sort, through multithreading, on Symmetric Multi-Processor systems (SMPs) with the help of popular multithreading APIs available for the C and C++ languages. The project also aimed to demonstrate the various advantages of multithreading, and compare the performance gains in terms of resource usage, execution time, performance costs, and scalability. For ease of implementation and uniformity of results, Ubuntu Linux was chosen since it supports up-to-date APIs.

**Methodology**

Exploit data parallelism by dividing an array of randomized integers, of a preferably large size, by forking multiple threads, all of which will apply parallelized sorting approaches of the algorithms listed. Each thread will be responsible for individually sorting a section of the array, which can be done either simultaneously or concurrently depending on kernel threads available in system resources. These threads will work synchronously, with synchronization primitives if needed. The output of the algorithms should be a correctly sorted array with execution times. Each of the parallel implementations along with serial implementation will be reiterated with increasing sizes of the array that will be graphed against the respective time taken for the algorithms to correctly sort the integer data. All three sorting algorithms will be implemented with both OpenMP and P-threads and analyzed. **4 threads used on 2 cores, 4 threads machine**

**Configuration of APIs**

* **OpenMP 3.1 used for OpenMP algorithms in C language**
* **Standard P-threads API used in C code**
* **Some code made portable with functions of Win32 API**
* **Both libraries are used on POSIX API for Ubuntu version 16.04**

**The results drawn for each algorithm will be compared based on:**

* **Execution Speed**
  + Measured with help of time difference before and after executing the initial parallel region of code. The total time taken was calculated by first finding the difference in number of clock cycles used with the clock() system call, which was then divided by the CLOCKS\_PER\_SEC constant to obtain the time difference in seconds.
* **Performance drawbacks**
  + Identify any additional overhead caused by thread creation and thread scheduling on performance, then correlate the effect of overhead with the execution time

**Algorithms**

**Merge Sort (Conquer then divide): O(nLogn)**

**P-threads:**

* Fork a team of specified number of threads from main thread using **pthread\_create** with default arguments.
* Each worker thread will spawn in the function:
  + **void\* thread\_merge\_sort (void\* arg)**:
    - Wrapper function that locally sorts a section of the global array.
    - Partitions the globally defined array into equal segments using constant NUM\_PER\_THREAD.
    - Defines local arguments for upper and lower bound using passed thread ID and makes nested call to **merge\_sort (int arr [], int left, int right).**
    - **Array is now locally sorted by each thread**
  + **void merge\_sections\_of\_array (int arr [], int number, int aggregation);**
    - Serves purpose of merging the partitions locally sorted before by threads
    - Single call made by main thread
    - Loops for every two consecutive threads
    - Makes nested call to merge (int arr [], int left, int middle, int right).
    - Make recursive call to merge sections that are not consecutive

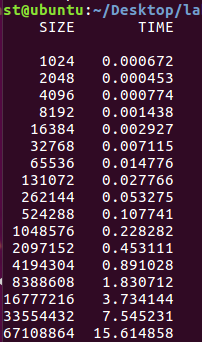
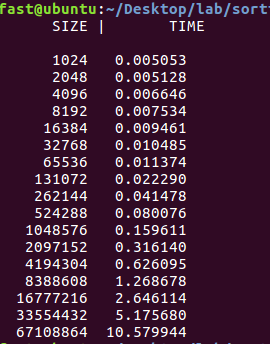
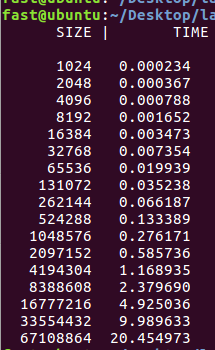
**OpenMP:**

* Single thread executes the parallel merge sort function void **mergeSort (int \*X, int n, int \*tmp)**
* Use of **OMP** construct **Task** which is similar to OpenMP **section** but works asynchronously. Two threads spawn in each recursive call to **mergeSort**.
* Make two recursive calls enclosed in **#pragma omp task shared(X) if (n > TASK\_SIZE)**
* **Shared(X) used since X is global array to be sorted, and if clause generates quick task**
* Wait for the tasks using **#pragma omp taskwait,** then call merge function

**Graphical Analysis**

* **P-thread and OMP showed faster execution time than sequential**
* **OpenMP was significantly faster than P-thread as can be seen in graph**
* **Similar times below 500000 size due to thread creation and scheduling overhead**

**PTHREADS OPENMP SERIAL**

****

**Quick Sort (Divide then Conquer): O(nLogn)**

**P-threads:**

* **void parallelQuicksort (double lyst [], int size, int tlevel):**
  + Fork a single thread that calls **parallelQuicksortHelper ()**
  + This thread has arguments in form of struct containing copy of array, bounds, and number of threads
* **void \*parallelQuicksortHelper (void \*threadarg):**
  + Recursive function to partition list into left and right sides
  + Fork two threads that will sort the left and right sides of their section of the array
  + Partition the array and store return value in variable **mid**
  + Arguments for these two threads is struct containing **low** and **high** **(mid-1, mid+1)**
  + Make recursive call for the two threads, so each call will spawn two more threads i.e., if using 4 threads, 16 threads spawned
  + BASE CASE: If array size sufficiently small, call sequential to complete the sorting

**OpenMP:**

* + Uses cutoff value of 1000 to switch to sequential quicksort if number of elements is less than 1000 in recursive function **quickSort\_parallel\_internal ()**
  + Use of task construct to asynchronously execute the two recursive calls in function **quickSort\_parallel\_internal ()**
  + Apart from these changes, sequential implementation is followed

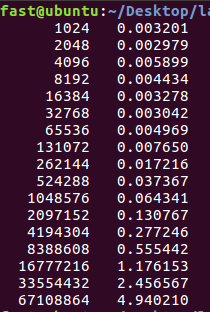
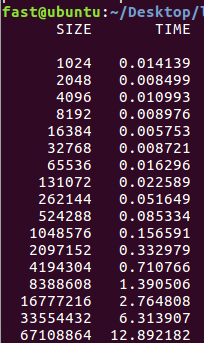
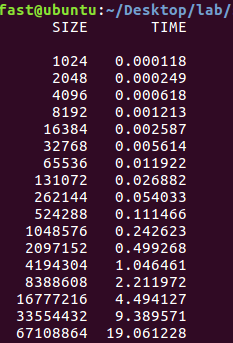
**Graphical Analysis**

* **P-thread and OMP showed significantly faster execution time than sequential**
* **OpenMP was faster than P-thread as can be seen in graph**
* **Similar times below 500000 size due to thread creation and scheduling overhead**

**Reasons for degraded performance of quicksort**

* **Minimal differences between parallel algorithms may be due to in place merge sort (2 copies created in merge function)**
* **Quicksort worsened at larger sizes due to duplicate values not being partitioned**
* **Random swapping requires values to be stored in L3 cache, so cache misses cause much higher sorting times**
* **Two threads created per recursive call which may exceed stack and cache size thus slower memory access as well (Higher memory consumption)**
* **Concurrent rather than parallel so lesser efficiency**

**OPENMP PTHREADS SERIAL**

**  **

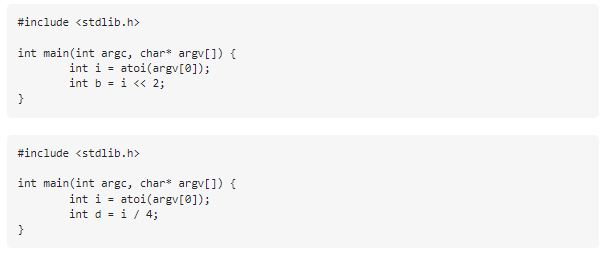
* **Results**
  + **OpenMP proved to be the fastest among the three**
  + **A significant speedup from 19 seconds to 4.9 seconds on 4 cores**
  + **This translates to roughly a 3.5x increase in performance**

**Radix Sort (Least-significant-digit bucketing): O(n\*d)**

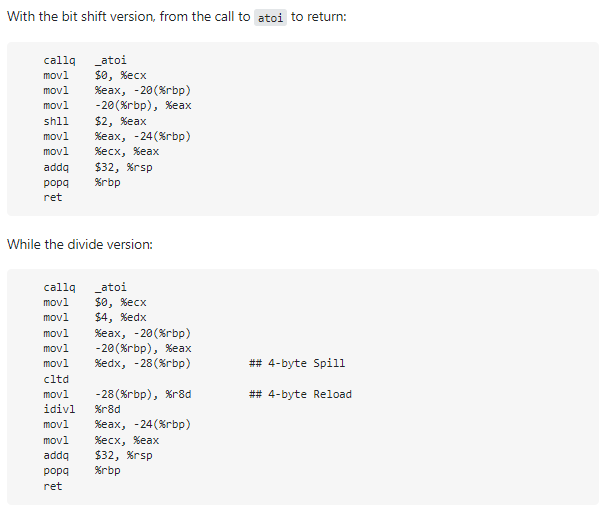
* **Uses bitwise operators (<<>>) instead of division/multiplication arithmetic operators (/\*)**
* **Bitwise operation is proven to be faster than arithmetic**

**HOW?**

**Consider this example assembly equivalent code for,**

****

**Assembly equivalent:**

****

**In the bit shift version,** the key instruction is **shll $2, %eax** which is a shift left logical - there's the divide, and everything else is just moving values around.

**In the divide version**, you can see the **idivl %r8d -** but just above that is a **cltd** (convert long to double) and some additional logic around the spill and reload. This additional work, knowing that we're dealing with a math rather than bits is often necessary to avoid various errors that can occur by doing just bit math.

By observing the above differences, it is therefore justified to use bitwise sorting with radix sort for performance gain

**Chosen algorithm: Least Significant Digit Radix Sort**

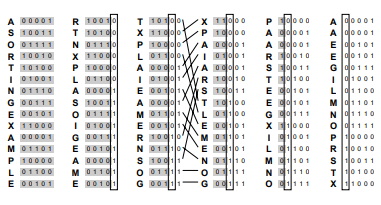


Fig. 1: SORTING DIGITS PER BIT IN BUCKETS

**Requirements**

* Base 256 used, which is size of 1 byte
* Use of >> for division, & for remainder

**Flow of program**

* Since we are sorting for 1 byte, and as integer has 4 bytes, loop program 4 times (for loop incrementing by 8 bits)
* 256 buckets used for each byte of each array element
* Each byte is considered to be a position/digit in base 256
* Find maximum element
* Divide max by increasing powers of 256
* **For each byte of the data elements (1 byte is the exponent) (for every position)**
  + Determine bucket number by shifting each array element by ith byte (determines key or digit position)
  + Add element to the bucket
  + Repeat for all array values
  + Now insert array values of each bucket back into original array in order of
  + Clear buckets
  + Repeat for each of the 4 bytes(columns) of data elements (serves as exponent)

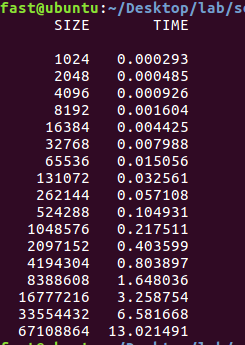
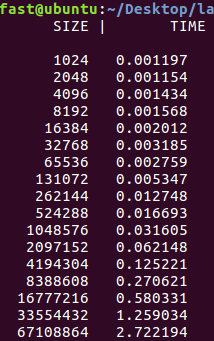
**P-threads:**

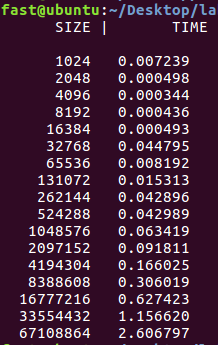
* Divide array into ranges equal to the number of threads and sort each range locally using **sort\_range ()** function that is executed by each thread using pthread\_create ()
* The **sort\_range ()** function counts the numbers in buckets and totals their indices after which the values in a range are processed by **put\_into\_buckets ()**
* **put\_into\_buckets ()**, places the values in the range into their correct positions according to their index that is determined by a bucket
* **Now back in parent function that called sort\_range ():**
  + Calculate indices for each bucket after local merge
  + Now merge the buckets for each thread by calling **put\_into\_buckets ()** for each thread

**OpenMP:**

* Using bitwise implementation, use a shared bucket of size 256 and a local bucket (private) for every thread
* Increment the count per bucket **asynchronously (no wait)**, and then add the local counts to shared bucket
* **Create barrier to wait for all threads to reach end of previous loop**
* Accumulate the bucket counts **using single thread**
* Since all threads add local bucket counts to shared bucket, subtract and update local buckets
* Calculate local bucket index using each element of array and place it in correct position through temp array.
* Parallel within parallel regions used, shared bucket updated using **critical clause (race condition avoided, mutual exclusion)**
* **Loop for every digit by incrementing shift (exponent) value**

**Graphical Analysis**

** SERIAL PTHREADS OPENMP**



* **Radix sort showed greatest efficiency in larger arrays due to no comparisons made**
* **Parallelization of bucketing much faster than divide and conquer**
* **Lesser thread creation and scheduling overhead due to iterative implementation**
* **Absence of recursion prevents useless threads from being created**
* **OpenMP again proved to be faster than P-threads**

**Conclusion:**

**OpenMP consistently outperformed P-threads in all sorting algorithms.**

**This is because OpenMP:**

* **Is faster**
* **Is scalable**
  + **OpenMP uses thread pools**
    - Threads are not spawned when pragma is encountered but are rather extracted from a pool of existing threads, which overcomes thread creation overhead to an extent. However, many of the non-active threads in the pool consume considerable memory as they are never deallocated. **This implementation may provide faster execution times since requests are immediately serviced by idle threads, but incurs significant amount of memory consumption**
  + **OpenMP utilizes implicit threading**
    - Thread management is **abstracted** in this API thus providing clarity in parallelism and guaranteeing optimization of pragma regions. In p-threads, explicit threading is emphasized which may worsen runtimes due to manual user thread invocation and synchronous operations such as the **fork-join model**
  + **OpenMP provides super-linear speedup**
    - By utilizing constructs such as tasks, for loop parallelization, reduction and no-wait, we are able to achieve significantly faster execution times
* **P-threads implements concurrent execution instead of parallel**
  + - This limitation prevents truly simultaneous execution of the algorithm on logical CPUs and only limits the parallelization to the user level. OpenMP is able to outperform P-threads thanks to it’s parallel implementation model.
* **It is far more scalable**
  + P-threads implementation almost always involved complete restructuring and modification of the sorting algorithm to allow each thread to independently execute in its spawned routine, thus the need for wrapper functions and private copies of arguments did arise. **OpenMP completely avoids the inclusion of such additional functions since one pragma statement is equivalent of, at times, entire p-threads routines.**

**References**

* **For API configuration**
  + [OpenMP Compilers & Tools - OpenMP](https://www.openmp.org/resources/openmp-compilers-tools/)
  + [c - Pthreads vs. OpenMP - Stack Overflow](https://stackoverflow.com/questions/3949901/pthreads-vs-openmp)
* **For comparison between P-threads and OpenMP**
  + [6.6. Implicit Threading and Language-based Threads — Computer Systems Fundamentals (jmu.edu)](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/ImplicitThreads.html)
  + [MSDN Magazine: Thread Pools - Scalable Multithreaded Programming with Thread Pools | Microsoft Docs](https://docs.microsoft.com/en-us/archive/msdn-magazine/2010/october/msdn-magazine-thread-pools-scalable-multithreaded-programming-with-thread-pools)
* **For radix sort implementation**
  + [radix.4up.pdf (princeton.edu)](https://www.cs.princeton.edu/courses/archive/spring02/cs226/lectures/radix.4up.pdf)
  + [Radix Sort — Data Structures and Algorithms in C++17 2.0 documentation (kurttest.com)](https://data-structs.kurttest.com/notes/radix-sort.html)
  + [Sorting algorithms/Radix sort - Rosetta Code](https://rosettacode.org/wiki/Sorting_algorithms/Radix_sort)
* **For implanting OpenMP directives**
  + [OpenMP support - IBM Documentation](https://www.ibm.com/docs/en/xl-c-and-cpp-linux/13.1.3?topic=v1313-openmp-support)
  + [#pragma omp task - IBM Documentation](https://www.ibm.com/docs/en/zos/2.4.0?topic=processing-pragma-omp-task)
  + [Chapter 3: nowait (aalto.fi)](https://ppc.cs.aalto.fi/ch3/nowait/)
* **For OpenMP snippets**
  + [lab6\_slides\_advanced\_openmp.pdf (cvut.cz)](https://cw.fel.cvut.cz/old/_media/courses/b4m35pag/lab6_slides_advanced_openmp.pdf)