CS 6210: Advanced Operating Systems

Implementation of Completely Fair scheduler in GTThreads library

Final Report

Shubhojit Chattopadhyay
ssc3@gatech.edu

I. Introduction

The GTThreads library given to the class had O[1] thread scheduling system implemented in it. The exact details of how O[1] was implemented in GTThreads is given in the interim report. In this project, this O[1] scheduler was replaced by **SCHED_OTHER** type of Completely Fair Scheduler. Further, each uthread now multiplies a matrix of it's own and stores in a single matrix (to reduce working memory footprint).

II. Details of CFS implementation:

The CFS implementation involves changing the following modules:

- Using Rbtrees instead of Runqueues
- Insertion in trees based on Vruntime
- Calculation of time slice based on Nice values and Weight (using real linux kernel values)
- Checking for preemption

1. Rbtree

The Rbtree library used is this one:
http://en.literateprograms.org/Red-black_tree_(C)

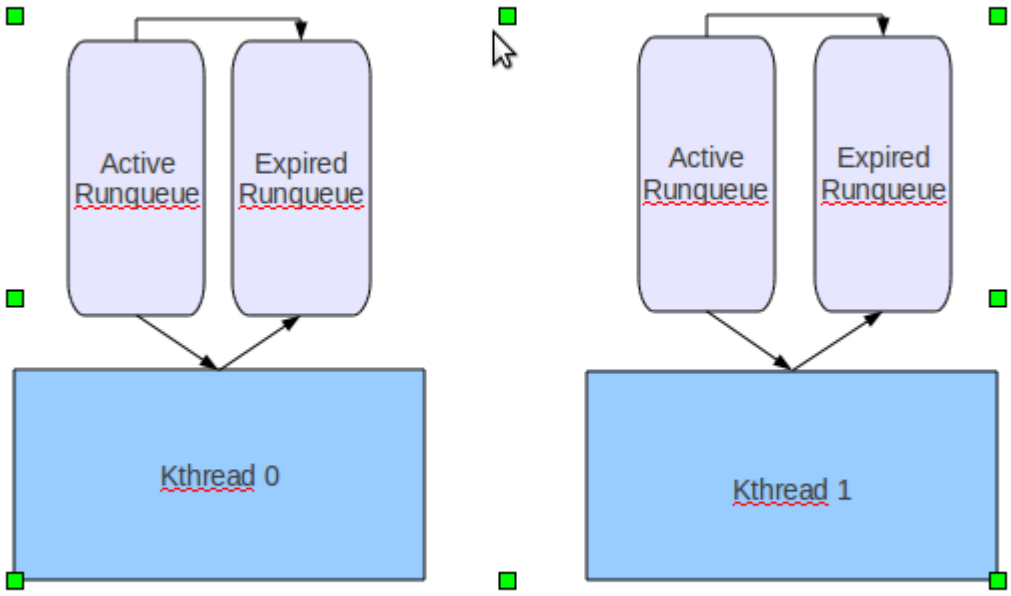
This library is extremely stable and performs very efficient memory allocation and deallocation, while adhering to the 5 basic rules of an RB tree which keep it balanced.

Note: There is a flaw with this library. While inserting a new node into a tree, if there is another node already present in the tree with the same node, the new node overwrites the already present node.

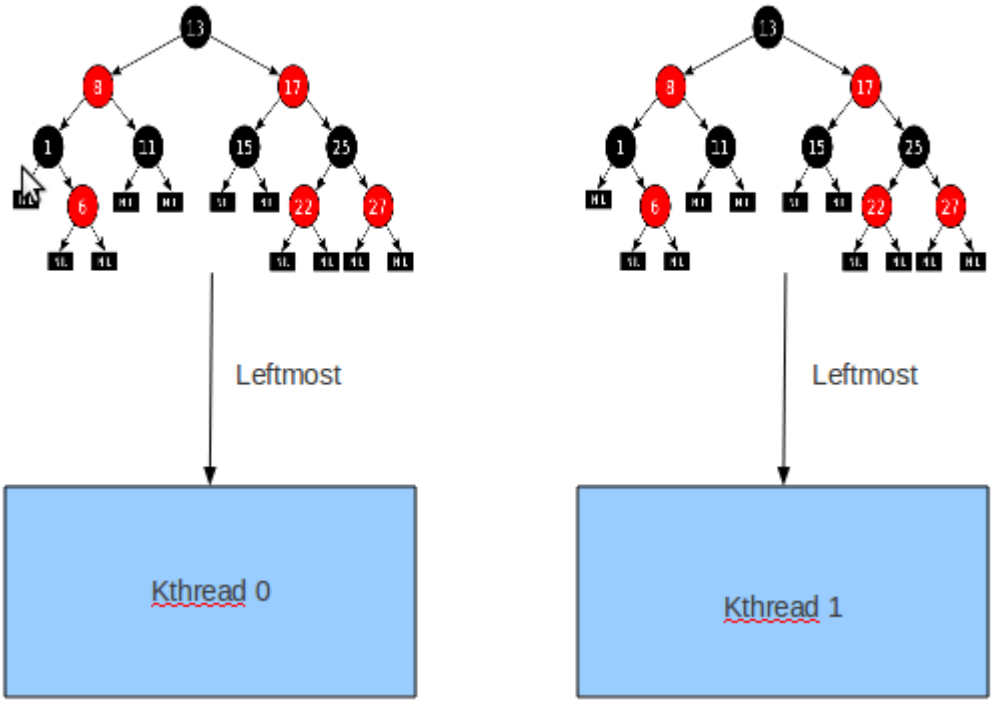
Each kthread has it's own Rbtree and schedules the leftmost thread whenever any new uthread is WAKING or the running uthread's timeslice has expired.

The following diagram illustrates how CFS Rbtree has been used and how it is different from O[1] implementation in GTThreads

O[1] scheduler Diagram



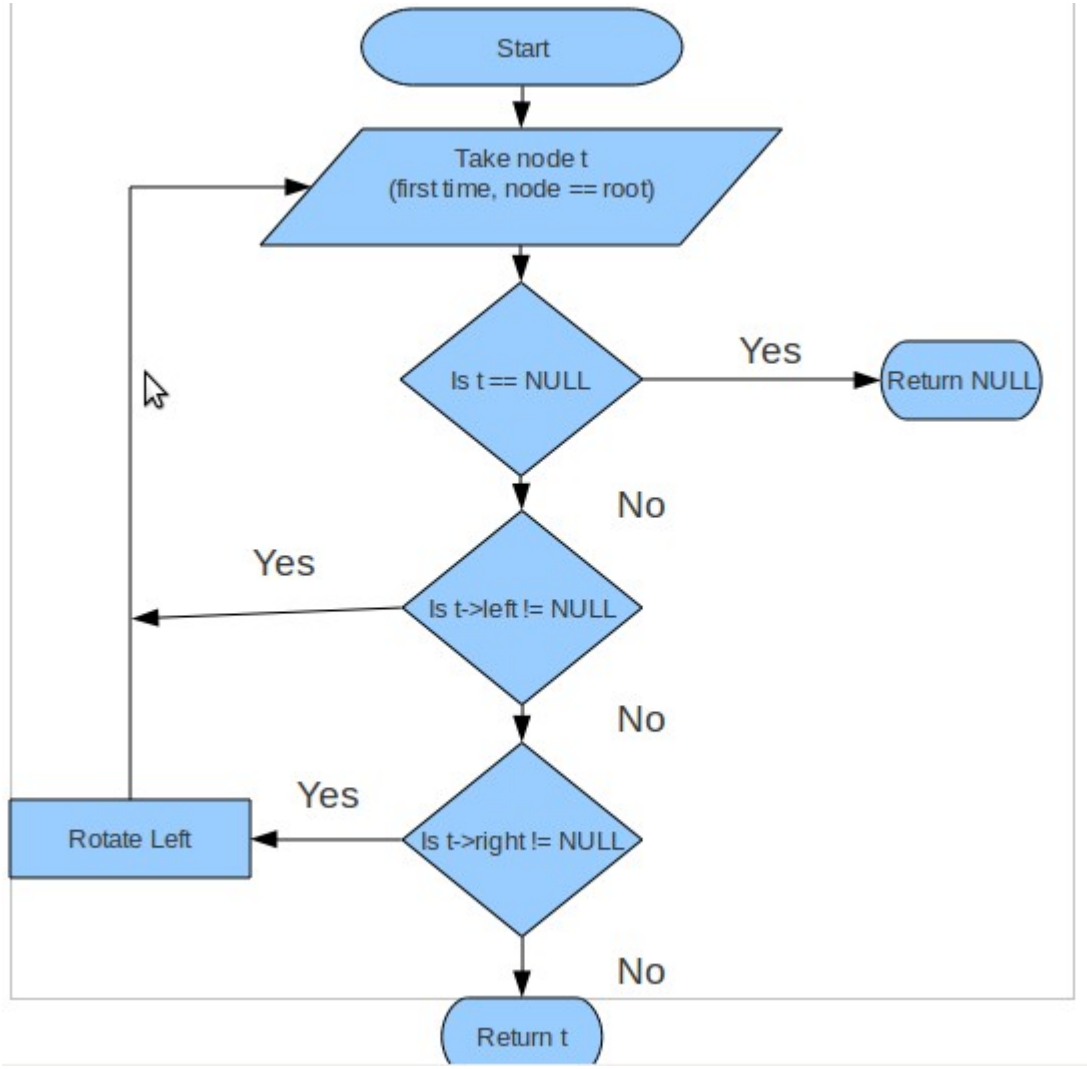
Rbtree implementation



As we can see, the whole notion of Active and Expired runqueue has been replaced by an Rbtree. When a new uthread is created, it is added into the Rbtree. The key for the tree is a void* to the uthread's vruntime and the value (or data) of each node in the tree is the uthread being inserted itself. The vruntime calculation is explained in detail later.

When a new thread is WAKING or the timeslice of current thread expires, uthread_schedule() function is called. Here the currently executing thread is put to RUNNABLE state and reinserted into the tree. After this, the sched_find_best_thread() function picks the leftmost node from the tree and passes it to uthread_schedule. This algorithm is a **custom algorithm** and makes use of the fact that Rbtrees are extremely balanced (this find_leftmost_node() algorithm cannot be used for an unbalanced tree) and is being used for performance reasons. The algorithm for finding leftmost node is shown below.

Finding leftmost node algorithm:



2. Vruntime calculations:

An attempt was made to make Vruntime calculations as **close to the real CFS calculations.** The calculations are shown below:

- The initial vruntime (when a thread is in INIT state and before adding into the tree) is made equal to tid. The original intention was to make vruntime of all threads in INIT state equal to zero. But this was not possible since, in the Rbtree library, a new node would overwrite an old node in the tree if they both have the same key

- While inserting into the tree, it's vruntime is calculated as:

```
delta = u_new->vruntime – runq->min_vruntime
if (delta > 0)
    return delta;
else
    return u_new->vruntime;
```

This is done in order to maintain cfs_rq->min_vruntime to be a **monotonically increasing** value tracking the leftmost vruntime in the tree

- While scheduling, the current time is calculated with **nanosecond resolution** and stored inside u_obj->start_time

```
clock_gettime(CLOCK_THREAD_CPUTIME_ID, &u_obj->start_time);
```

- When it's run it's timeslice/preempted by a WAKING uthread, it's runtime is calculated as

```
clock_gettime(CLOCK_THREAD_CPUTIME_ID, &u_obj->end_time);
u_obj->vruntime = u_obj->end_time.tv_ns – u_obj->start_time.tv_ns
```

- Then, the uthread is added back into the Rbtree, after making the calculations given in step 2

The following things were intentionally not implemented:

- **Dynamic assigning of Nice()** values was not done because it wasn't clear as to what algorithm is used to decide nice values of different threads.
- Nice values for smaller uthreads are higher and for bigger uthreads are smaller (i.e. For threads multiplying 32x32, nice values are 10 and for 128x128, it's -5). These values have been assigned after experimentation to provide the best results. No algorithm was used to calculate these values.

3. Timeslice calculations:

The timeslice calculation follows the same implementation as done in actual linux CFS. The code was written in an unrolled manner without too many function calls in order to make it look simple:

- *Period = sysctl_sched_latency;*
- *If (!(nr_running > sched_nr_latency)*
  *period *= nr_running;*
  period = period/sched_nr_latency
- *vslice = period*Weight[NICE VALUE]/(sum_of_weights_in_rbtree)*
- *if (vslice > 4000000ULL)*
  kthread_init_vtalrm_timeslice(vslice)
  else
  kthread_init_vtalrm_timeslice(4000000ULL)

Note: The sum of weights in the tree is using NICE Value weights given in the actual linux 2.6.24 kernel/sched.c

static const u32 prio_to_wmult[40] = {
/* -20 */ 48388, 59856, 76040, 92818, 118348,
/* -15 */ 147320, 184698, 229616, 287308, 360437,
/* -10 */ 449829, 563644, 704093, 875809, 1099582,
/* -5 */ 1376151, 1717300, 2157191, 2708050, 3363326,
/* 0 */ 4194304, 5237765, 6557202, 8165337, 10153587,
/* 5 */ 12820798, 15790321, 19976592, 24970740, 31350126,
/* 10 */ 39045157, 49367440, 61356676, 76695844, 95443717,
/* 15 */ 119304647, 148102320, 186737708, 238609294, 286331153,
};

4. Checking for Preemption:

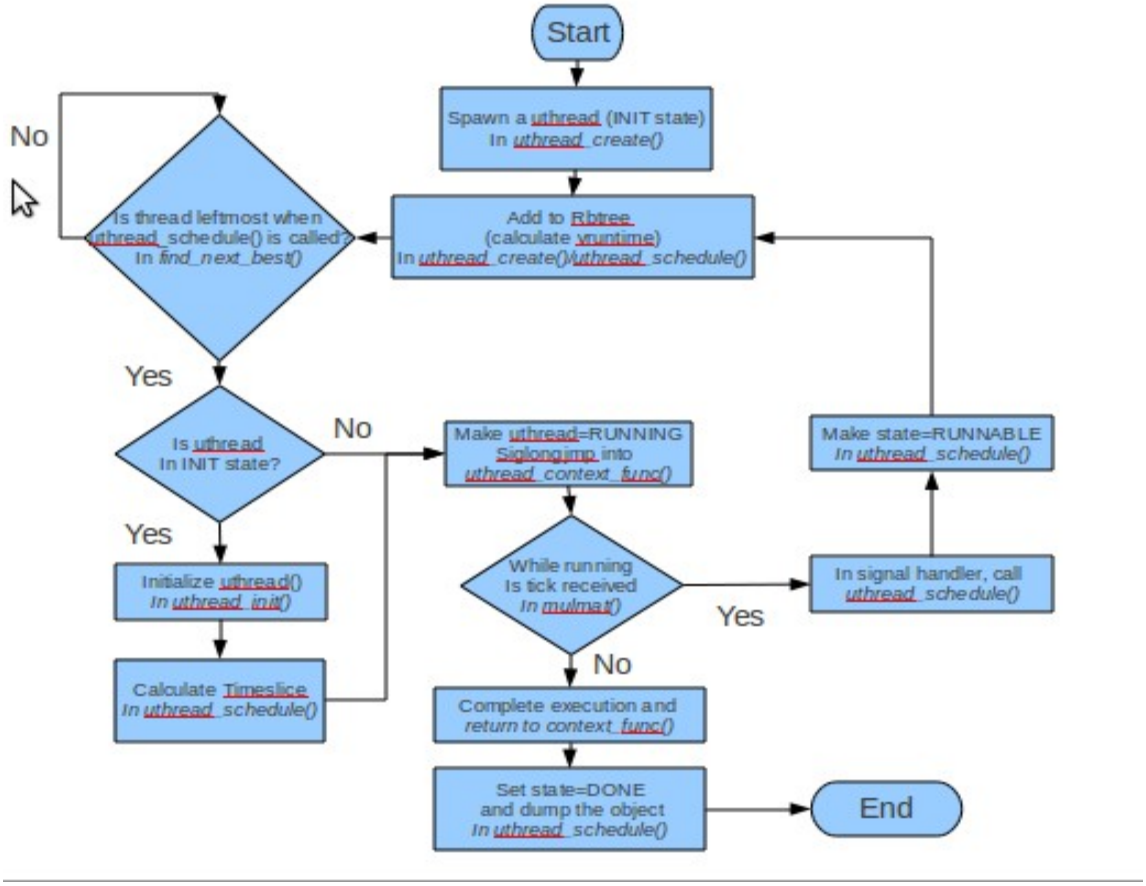
An important part of integrating CFS is checking for preemptions. In O[1], there was no need to check for preemptions. However, in CFS, a running thread can be preempted by another thread. This preemption can happen whenever an interrupt is raised. This interrupt in gtthreads is the SIGVTALRM signal.

The SIGVTALRM signal is a signal sent by the kernel to the process at some regular intervals. This regular interval is actually the timeslice of the running thread. The general operational algorithm is as follows:

- On uthread_schedule(), take the leftmost node from the tree
- Calculate it's timeslice
- Make SIGVTALRM interval = timeslice and run that thread
- After the timer expires, system sends a signal to the running uthread
- This signal is handled by ksched_priority()
- ksched_priority() calls uthread_schedule(), where this running uthread is added back to the tree

- Now, we check for the leftmost node again. If the previously running thread still has lesser vruntime, it is scheduled, else the leftmost is scheduled

5. **Lifecycle of a uthread:**



III. Sched_Yield()

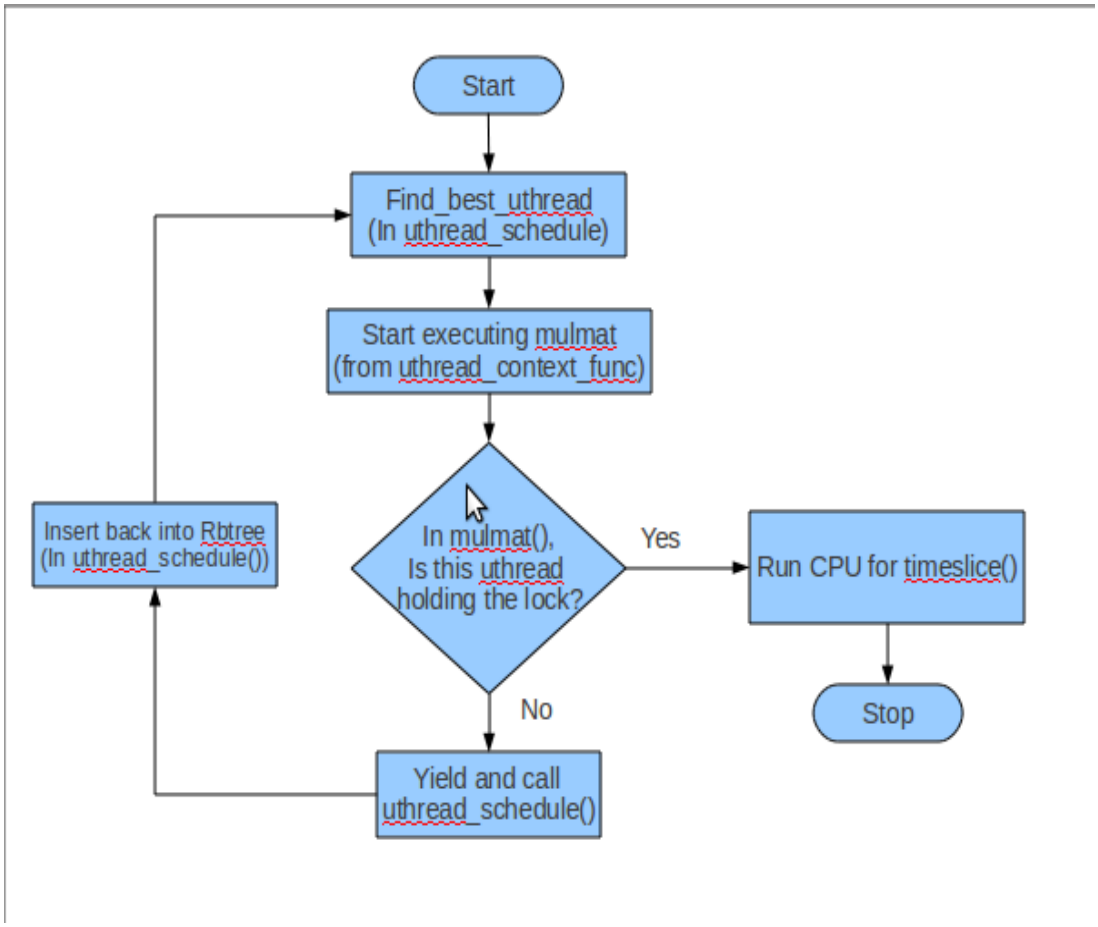
Sched_yield() is a voluntary preemption function. The running thread simply gives away control of the CPU to the scheduler, which decides which thread to be scheduled next. From the system perspective, this essentially means **“enqueue and dequeue”**.

The biggest use of sched_yield() is when we are using locks/mutexes. If one thread holds the lock for some critical section, the other threads (when scheduled) unnecessarily occupy the CPU until their timeslice expires. These other threads (which do not have the lock) should voluntarily preempt themselves when such a thing happens so as to maximize CPU usage.

In GTThreads, this has been realized using a spinlock. In mulmat function, gt_spinlock is put on ptr->C[][]. This lock can be held by only one thread at a time. The other threads on arriving at the lock, preempt themselves by calling yield_custom() function. This function in turn calls uthread_schedule() which is basically responsible for saving the context of the current thread and schedules the next best

thread.

The basic function flow is shown below:



This would increase performance as compared to the earlier case where there was no yielding, since the CPU is getting maximally utilized.

IV. Matrix Multiplication

Each uthread has to compute a matrix of it's own. The matrices are initialized in init_matrices() (inside main()) and given the right SIZE and values. After that, they are assigned NICE VALUE WEIGHTS based on their computation size. This then helps in deciding Vruntime.

Also, mulmat now calculates one full matrix multiplication. This means that each time uthread_create is called, a new thread is spawned which calculates a matrix multiplication of it's own. The matrix multiplication results are stored in a new C[tid][i][j] matrix.

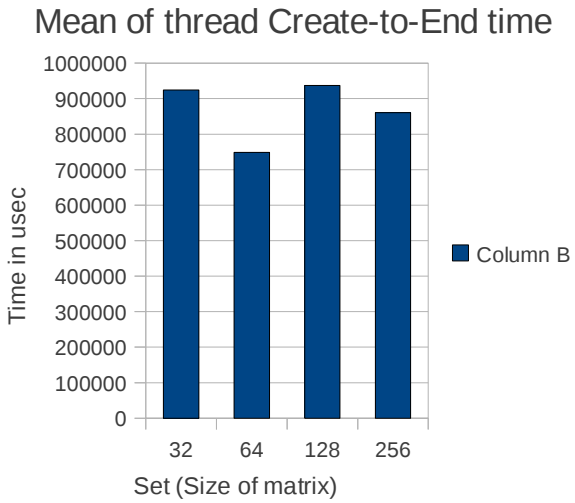
Matrix multiplication also has a lock provided for C[][][] matrix. This lock is useful for demonstrating sched_yield() as explained above.

V. Observations:

The experiment was run on Shuttle cluster provided by CoC at gatech. The results were collected by running the modified gtthreads program **10 times** and then, taking an **average of the results**. The whole set of results can be found in **output.log** file in the tarball.

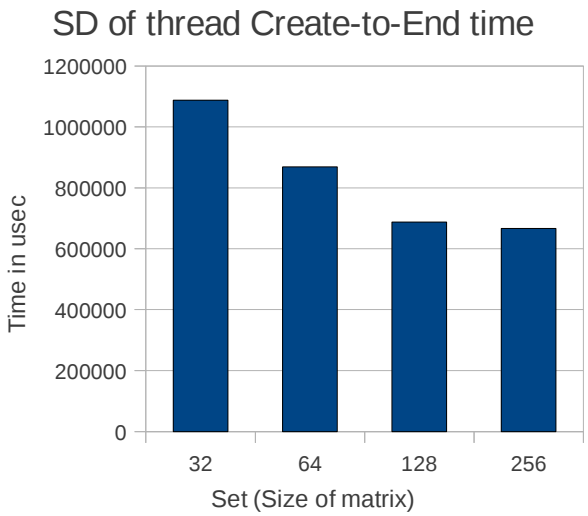
a) Time of uthread creation to the end time

Mean Value:



- As we can see, the create to end time for all sets of threads is pretty uniform
- This shows that the scheduling algorithm was pretty fair to matrices of all sizes
- Note that this is not a measure of actual CPU runtimes. It is just the average of how much time a set spends in the tree, from the point of creation

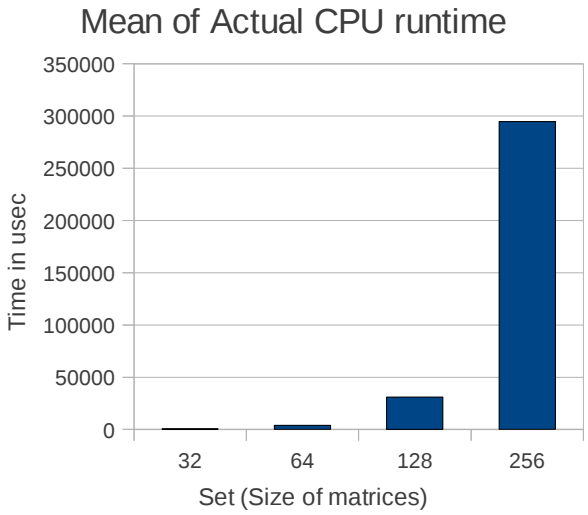
Standard Deviation:



- The Standard deviation seems to be pretty much stable as well
- A slightly decline in standard deviation is observed as the size of the matrices increases, for no particular reason

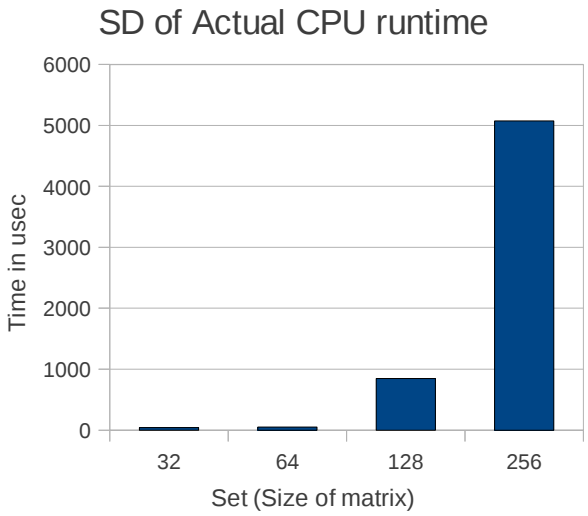
b) Mean - Time of actual runtime on CPU

Mean Value:



- Clearly, it was observed that the actual CPU runtime for uthreads of higher sizes was more than that for uthreads of lower sizes
- This is purely because of the fact that a thread with large matrix computation, would need longer computation time as compared to other threads

Standard Deviation:



- Again, clearly, as the size of the matrices increase, the SD increases too
- This is plainly due to the fact that for bigger matrices the computation takes a longer time and the time of clock_gettime() might not always be close to the mean

VI. Conclusion

Thus, SCHED_OTHER mode of CFS was implemeted in GTThreads. The implementation was as close to that of actual linux 2.6.24 kernel as possible. Nice values were statically assigned to each thread. Sched_yield() was also implemented for preemption of a thread when it does not have a lock to a critical section.