Mon Carlo Problem

- One of the most important tips was using **rand()** function as it uses a shared seed for tracking seed status so it is **not thread safe !**

  The Solution was using **rand_r(&seed)** instead and use private seeds for individual Threads. (Initial seeds first will be generated using time based rand()s on main thread)

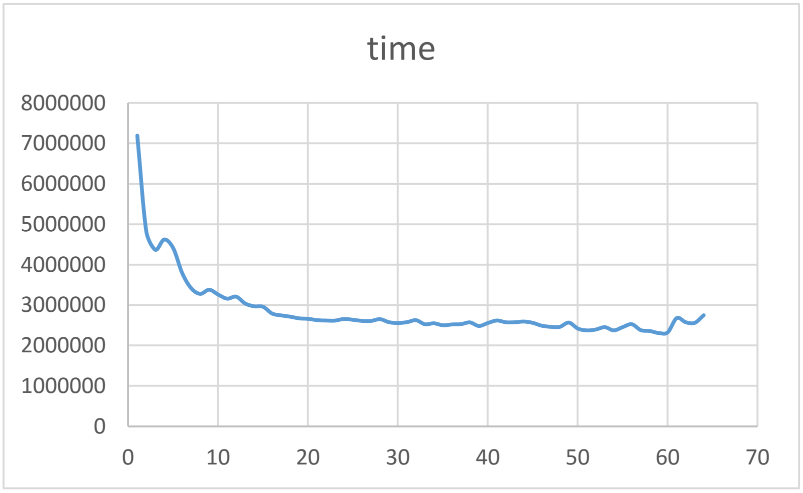
- Another Important Tips was using **reduction** of *circle_hit* wen counting Circle-Area hits.
- Also it was considered to Make Thread workers as light as possible to increase Speed.

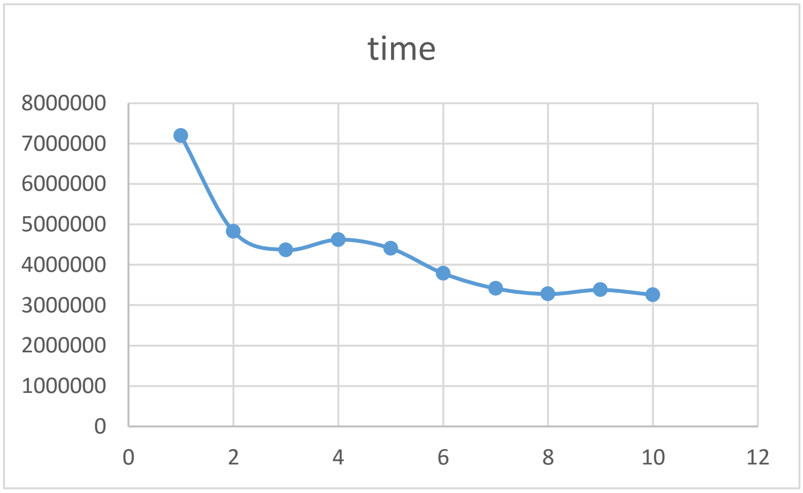
Results

- On Cygwin Environment Host with **8** Core Corei7 CPU. And **100M** Test Cases with Threads in Range **[1-64]**

Best Time: **2308.679 MS**

Best Estimate's error: **0.00000806**





As you can see, Increasing Number of threads more than **8** gives no better performance …

Detailed results are inside Moncarlo_ThreadBenchmark_100000000.ods.

Thread Loops

Open MP Loop Collapse Directive

https://software.intel.com/en-us/articles/openmp-loop-collapse-directive

Open MP* Loop Collapse Directive

Use the Open MP collapse-clause to increase the total number of iterations that will be partitioned across the available number of OMP threads by reducing the granularity of work to be done by each thread. If the amount of work to be done by each thread is non-trivial (after collapsing is applied), this may improve the parallel scalability of the OMP application.

You can improve performance by avoiding use of the collapsed-loop indices (if possible) inside the collapse loop-nest (since the compiler has to recreate them from the collapsed loop-indices using divide/mod operations AND the uses are complicated enough that they don't get dead-code-eliminated as part of compiler optimizations)

```
#pragma omp parallel for collapse(2)

  for (i = 0; i < imax; i++) {

    for (j = 0; j < jmax; j++) a[ j + jmax*i] = 1.;

  }
```

Modified example for better performance:

```
#pragma omp parallel for collapse(2)

  for (i = 0; i < imax; i++) {

     for (j = 0; j < jmax; j++) a[k++] = 1.;

  }
```

Thread Pools

“A thread pool is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application. The thread pool is primarily used to reduce the number of application threads and provide management of the worker threads. Applications can queue work items, associate work with waitable handles, automatically queue based on a timer, and bind with I/O.” (*MSDN*)

According to OpenMP Documentations It can have similar behavior using Loop Scheduling and **Chunk-Sizes** Of tasks