Thread Pool

OPENAIRINTERFACE

Mikel Irazabal

Mosaic5G

24/05/22

0 Outline

1. EoP and RT
2. Thread Pool Design
3. Testing Framework
4. Results
5. Can we do it better?
6. Next steps

1. EoP and RT
2. Thread Pool Design
3. Testing Framework
4. Results
5. Can we do it better?
6. Next steps

1 Why we need EoP?

- Ask a mechanical, structural or electrical engineer how far they would get without a heavy reliance on a firm mathematical foundation, and they will tell you, not far. Yet so-called software engineers often practice their art with little or no idea of the mathematical underpinnings of what they are doing. And then we wonder why software is notorious for being delivered late and full of bugs, while other engineers routinely deliver finished bridges, automobiles, electrical appliances, etc., on time and only with minor defects... *Martin Newell, preface of EoP*


1 What is RT?

- Real-time or real time describes various operations in computing or other processes that must guarantee **response times** within a specified time (deadline), usually a relatively short time.
- $RT \neq LowLatency$
- In fact, a RT system needs more time to finish a bunch of tasks compared with a generic system.
- CPU loses a large portion of its time scheduling what process will be executed next, and thus, doing the context switch.
- Therefore, the **total execution time is longer**, and that's why no one runs a preemptible kernel on webserver or database machines.
- $RT \neq LowLatency$

2 Outline

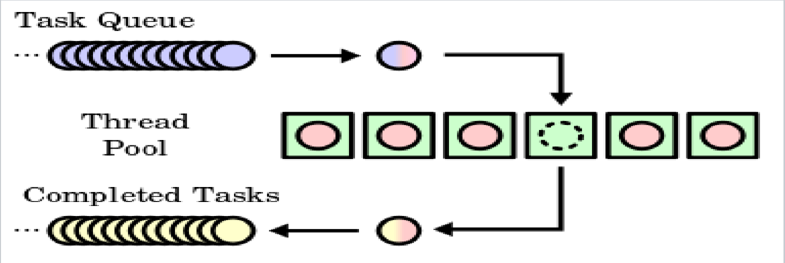
1. EoP and RT
2. Thread Pool Design
3. Testing Framework
4. Results
5. Can we do it better?
6. Next steps

2 Method: Tricks for formulating and solving problems (Creative thinking 1952 by C. Shannon)

- Simplification (get rid of enough detail, including practical aspects for intuitive understanding)
- Similarity (find a related known problem)
- Reformulate (avoid getting stuck in a rut)
- Generalize (usually guided by simplifications)
- Structural analysis (break problem into pieces)
- Inversion (work back from desired results)
- Use the above together and/or one after the other

2 Thread Pool: Definition

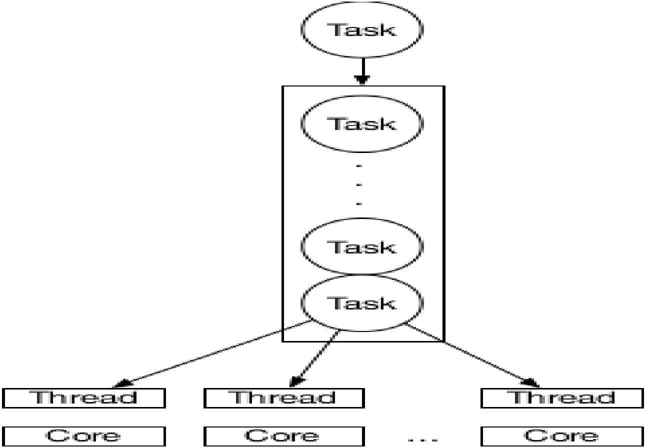
- In computer programming, a thread pool is a software design pattern for achieving concurrency of execution in a computer program.
- A thread pool maintains multiple threads waiting for tasks to be allocated for concurrent execution by the supervising program.
- By maintaining a pool of threads, the model increases performance and avoids latency



A sample thread pool (green boxes) with waiting tasks (blue) and completed tasks (yellow)

2 Classic Thread Pool

- One queue for all the tasks
- To avoid data races a mutex is needed
- Threads are preempted until signaled that a task arrived



2 Contemporary HW

- Many CPUS
- Modern CPUs are normally faster than memory access
- Modern CPUs are deeply pipelined
- CPU stalls happened when CPUs need to sync

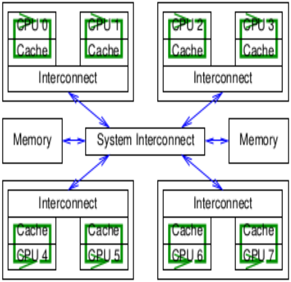
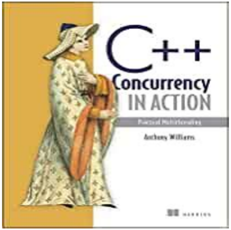


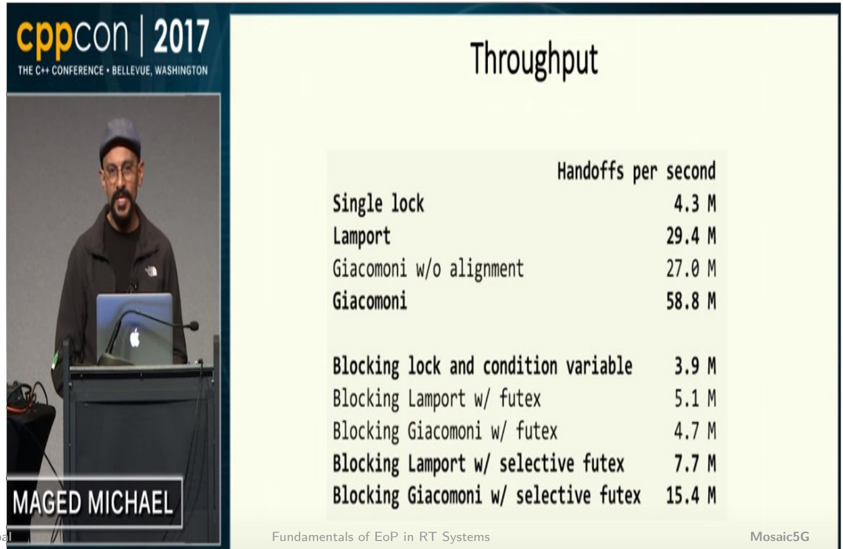
Figure: Fig from Is Parallel Programming Hard, And, If So, What Can You Do About It? by Paul McKenney

- One shared queue among threads
  - No paralellism
  - Cache line bouncing
  - Kernel context switch (i.e., due to pthread_mutex_lock)
  - "If we named it Bottleneck instead of Mutex, people may actually think twice before using them" — Anthony Williams



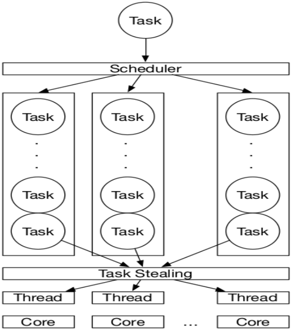
2 Result for SPSC using mutex w/ blocking

- Up to x12 times slower



2 New Thread Pool (inspired by Sean Parent's talk)

- One queue per thread, and thus, low contention
- Tasks are tried to pushed into a queue. If it doesn't succeed to acquire the mutex it tries the next one.
  - We don't switch to kernel space
- No affinity or priority set
- A work stealing algorithm tries to steal tasks from other queues if no more tasks are available in their corresponding queue



2 New Thread Pool API and DS

- We use a growing and shrinking circular buffer to increase data locality
- 384 vs 907 LOC
- We don't allocate unnecessarily in the heap

```
void init_task_manager(task_manager_t* man, uint32_t num_threads);
void free_task_manager(task_manager_t* man, void (*clean)(void* args) );

typedef struct{
  void* args;
  void (*func)(void* args);
} task_t;

void async_task_manager(task_manager_t* man, task_t t);
void wait_all_task_manager(task_manager_t* man);
```

1. EoP and RT
2. Thread Pool Design
3. **Testing Framework**
4. Results
5. Can we do it better?
6. Next steps

3 Testing framework

- Need a CPU intensive task -> Recursive Fibonacci series
- 1024x1024 tasks
- 4 Threads @11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- Fibonacci(19) 6 microseconds in my CPU

```
int fibonacci(int v)
{
  if(v < 2)
     return 1;
  return fibonacci(v-1) + fibonacci(v-2);
}
```

3 Fibonacci 19 1024x1024 tasks 4 Threads OAI

```
Performance counter stats for './oai_tp':
       9 413,28   task-clock               4,047 CPUs utilized
        371 374   context-switches        39,452 K/sec
              6   cpu-migrations           0,637 /sec
         25 332   page-faults              2,691 K/sec
 26 015 324 755   cycles                   2,764 GHz
 79 509 214 568   instructions             3,06  insn per cycle
 15 946 587 478   branches                 1,694 G/sec
     25 763 824   branch-misses            0,16% of all branches
110 809 313 960   slots                   11,772 G/sec
 12 737 686 632   topdown-retiring        10,6% retiring
 94 409 034 848   topdown-bad-spec        78,5% bad speculation
 11 992 590 438   topdown-fe-bound        10,0% frontend bound
  1 129 558 774   topdown-be-bound         0,9% backend bound
    2,326205397 seconds time elapsed
    7,781928000 seconds user
    2,203263000 seconds sys
```

3 Fibonacci 19 1024x1024 tasks 4 Threads WS


```
Performance counter stats for './ws_tp':
      6 389,02 msec task-clock          3,933 CPUs utilized
            36    context-switches   5,635 /sec
            10    cpu-migrations     1,565 /sec
        12 222   page-faults         1,913 K/sec
19 122 478 573   cycles              2,993 GHz
75 736 356 349   instructions        3,96  insn per cycle
15 130 162 030   branches            2,368 G/sec
    18 787 864   branch-misses       0,12% of all branches
93 419 040 280   slots               14,622 G/sec
63 656 603 019   topdown-retiring    67,6% retiring
 4 506 057 716   topdown-bad-spec     4,8% bad speculation
21 780 574 306   topdown-fe-bound    23,1% frontend bound
 4 249 158 071   topdown-be-bound     4,5% backend bound
   1,624340250 seconds time elapsed
   6,385665000 seconds user
   0,003998000 seconds sys
```

3 Conclusion

- 3.96 vs 3.04 insn per cycle
- 14.62 vs 11.77 G/sec slots
- 1.6 vs 2.32 elapsed time
- 6.39 vs 7.78 user time
- 0.004 vs 2.3 sys time
- 36 vs 371374 context-switches

3 Compare: C thread pool

- The most popular C Thread pool in github

About

A minimal but powerful thread pool in ANSI C

Readme

MIT license

1.6k stars

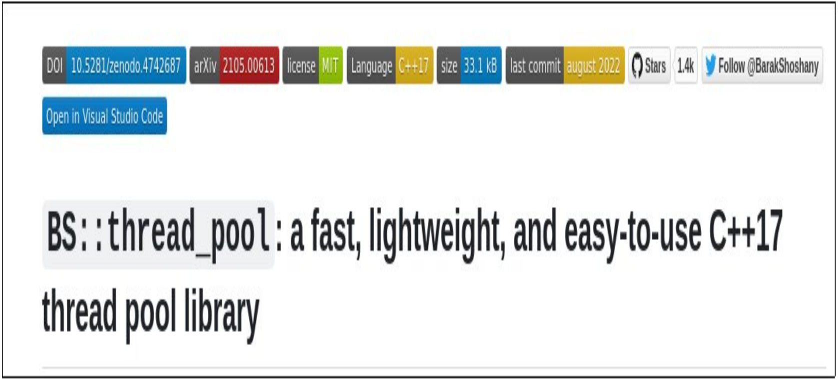
69 watching

522 forks

Report repository

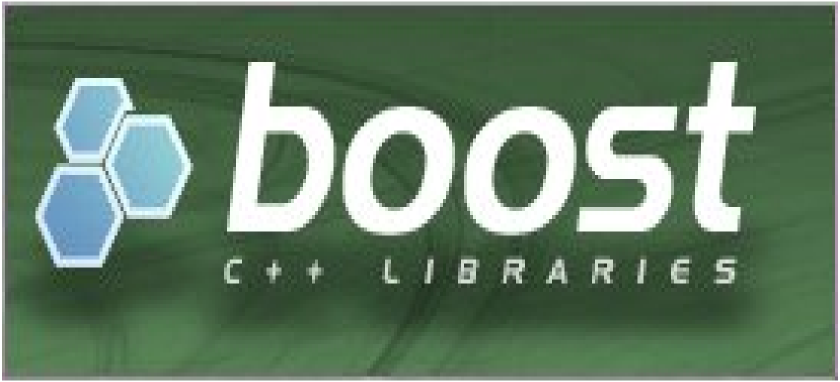
3 Compare: BS thread pool

- Popular C++ Thread pool in github
- Very good documentation and Popular C++ Thread pool in github
- Built from scratch with maximum performance in mind.



3 Compare: Boost

- Very popular high quality c++ libraries



3 Compare: GLib

- Foundational library for gnome projects

GLib

GLib is the low-level core library that forms the basis for projects such as GTK and GNOME. It provides data structure handling for C, portability wrappers, and interfaces for such runtime functionality as an event loop, threads, dynamic loading, and an object system.

The official download locations are: https://download.gnome.org/sources/glib

The official web site is: https://www.gtk.org/

Installation

3 Compare: libuv

- Foundational library for Node JS, Julia...

libuv

Overview

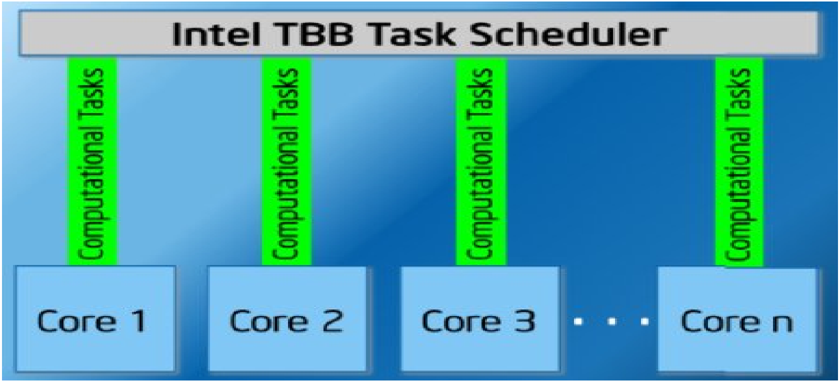
libuv is a multi-platform support library with a focus on asynchronous I/O. It was primarily developed for use by Node.js, but it's also used by Luvit, Julia, uvloop, and others.

3 Compare: OpenMP

- Wikipedia: OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran,[3] on many platforms, instruction-set architectures and operating systems, including Solaris, AIX, FreeBSD, HP-UX, Linux, macOS, and Windows. One of the most optimized

3 Compare: Intel TBB

- Intel® Threading Building Blocks



1. EoP and RT
2. Thread Pool Design
3. Testing Framework
4. Results
5. Can we do it better?
6. Next steps

4 Thread pools coloseum. 8x1024x1024 tasks 8 Threads @11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz

Fibonacci(19)

- **ws 10.1 sec**
- glib 10.76 sec
- boost 10.86 sec
- openMP 10.93 sec
- tbb 12.57 sec
- bs_tp 13.83 sec
- libuv 14.32 sec
- **oai 15.60 sec**
- c_tp 29.34 sec

Loop [1,3] times Fibonacci(19)

- **ws 19.53 sec**
- glib 20 sec
- openMP 20.90 sec
- boost 20.9 sec
- tbb 22.34 sec
- c_tp 22.42 sec
- bs_tp 23.47 sec
- **oai 26.17 sec**
- libuv 27.47 sec

Fibonacci [17,19]

- **ws 6.79 sec**
- boost 7.7 sec
- openMP 7.76 sec
- glib 7.79 sec
- libuv 9.19 sec
- tbb 9.63 sec
- bs_tp 9.67 sec
- **oai 13.24 sec**
- c_tp 34.76 sec

4 Conclusion

- The new TPool is fast
- We are probably close to a hardware limit
- Probably its simplicity is the key for these results

4 Run queue latency

- In RT systems, the run queue latency can have a considerable impact
- It is the sojourn time since a thread is signaled to run, until it gets scheduled
- This effect can add up to 100 microseconds of delay, ruining all the previous TPool effort
- Partial solution: wake up the thread before needed and make it spin in an atomic variable

4 Test nr_ulsim at Meduse with spin

Num. Threads	Symbol Proc. Time (us) OLD|NEW	ULSCH Decoding (us) OLD | NEW
1	716 | 690	2230 | 2135
2	394 | 376	1227 | 1129
4	235 | 205	679 | 605
8	**187**| 159	**419** | 344
16	193 | **110**	479 | 254
24	256 | 123	804 | **215**

4 Conclusion

- This approach scales
- However, it depends in the HW where OAI is running to "tune" when we should wake up the threads
- It consumes considerably more CPU
- The solution needs to be HW independent or it will not be

5 Outline

1. EoP and RT
2. Thread Pool Design
3. Testing Framework
4. Results
5. Can we do it better?
6. Next steps

5 Dyntick CPU

- Isolate some CPUs to test the TPool
- Let's only run our threads
- Download and compile the latest kernel with CONFIG_NO_HZ_FULL=y and CONFIG_RCU_NOCB_CPU=y
- Boot parameters isolcpus = 4-7 rcu_nocbs = 4-7 nohz_full = 4-7 rcu_nocbn_poll

5 Fibonacci 19 1024x1024 tasks 4 Threads WS


```
#!/bin/bash
# Full dyntick CPU on which we'll run the user loop,
# it must be part of nohz_full kernel parameter
TARGET=4
# Migrate all possible tasks to CPU 0
for P in $(ls /proc)
do
  if [ -x "/proc/$P/task/" ]
  then
    echo $P
    taskset -acp 0 $P
  fi
done
```

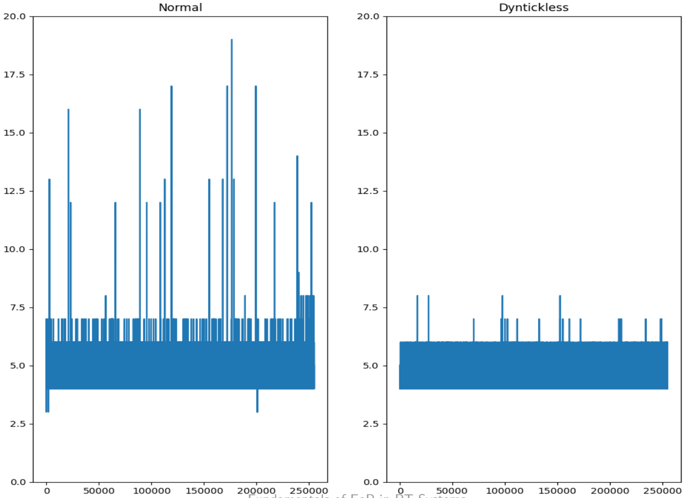
5 Fibonacci 19 1024x1024 tasks 4 Threads WS

```
# Migrate irqs to CPU 0
for D in $(ls /proc/irq)
do
  if [[ -x "/proc/irq/$D" && $D != "0" ]]
  then
    echo $D
    echo 1 > /proc/irq/$D/smp_affinity
  fi
done
# Delay the annoying vmstat timer far away
sysctl vm.stat_interval=120
# Shutdown nmi watchdog as it uses perf events
sysctl -w kernel.watchdog=0
# Remove -rt task runtime limit
echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

5 Fibonacci 19 1024x1024 tasks 4 Threads WS

```
# Pin the writeback workqueue to CPU0
echo 1 > /sys/bus/workqueue/devices/writeback/cpumask
DIR=/sys/kernel/debug/tracing
echo > $DIR/trace
echo 0 > $DIR/tracing_on
# Uncomment the below for more details on what disturbs the CPU
echo 0 > $DIR/events/irq/enable
echo 1 > $DIR/events/sched/sched_switch/enable
echo 1 > $DIR/events/workqueue/workqueue_queue_work/enable
echo 1 > $DIR/events/workqueue/workqueue_execute_start/enable
echo 1 > $DIR/events/timer/hrtimer_expire_entry/enable
echo 1 > $DIR/events/timer/tick_stop/enable
echo nop > $DIR/current_tracer
echo 1 > $DIR/tracing_on
# Run a 10 secs user loop on target
taskset -c $TARGET ./user_loop &
sleep 10
killall user_loop
# Checkout the trace in trace.* file
```

5 Elapsed time for calculating Fibonacci 19 in us for dyntickless normal mode



5 Run queue latency

- Unfortunately it does not resolve the problem
- "Real-time" policies suffer from the same problem for being waked up
- SCHED_FIFO or SCHED_RR do not react faster, even in isolated CPUs
- pthread_barrier does not provide a faster reaction time
- Avoiding glibc and calling directly the OS does not result in better results

```
syscall(SYS_futex, &man->futex, FUTEX_WAKE_PRIVATE,
          INT_MAX, NULL, NULL, 0);
```

6 Outline

1. EoP and RT
2. Thread Pool Design
3. Testing Framework
4. Results
5. Can we do it better?
6. Next steps

6 Next steps

- Current solution scales but is dependent on the HW
- Work has been started to try to adapt OAI's TPool API to lessen the amount of work to be done if we decide to switch
- The solution is to minimize linux scheduler's latency to assign the thread into the CPU
- There is no theoretical reasons of why the scheduler does not assign the threads faster to the CPU
- Maybe a non-preemtible task arrives when assigning the threads to the CPU. Then, the scheduler needs to be isolated from other interrupts

Thank you!