README

Kvc

December 18, 2018

Contents

This directory contains the experiment setup and results from the winter simulation conference paper "Just in time Parallel Simulation.

The data structure evaluation is contained in $\text{data}_{\text{structureeval}}$ directory. The results reported in the paper were evaluated on a 2017 Macbook pro with a 2.3 GHz Intel Core i5-7360U processor (while plugged in). Additionally new results have been added from a Intel Core i7-4790 CPU @ 3.6GHz x8

1 Installation

You need python 2 and pypy and the dev packages for them. Installation from source is how I did it in osx and in Ubuntu I installed with apt-get install pypy python pypy-dev python-dev

The Beta version of Pypy is better and the Beta version of LuaJit is MUCH better but you can still see the trends in the Linux chart.

2 Tests and explainations

2.1 Test 1 static priorities

In this test we add 1,000,000 items to the priority queue with increasing priorities. i.e. Loop (i=0; i < 1000000; ++i) {enqueue(i,NULL)} then loop again to dequeue all events

2.2 Test 2 random priorities

In this test we add 1,000,000 items to the priority queue with random priorities. i.e. Loop (i=0; i < 1000000; ++i) {enqueue(rand(0,1000000),NULL)} then loop again to dequeue all events

2.3 Test 2 random priorities

In this test we add 1,000,000 items to the priority queue with random priorities. i.e. while (#enqueued < 1000000){ Loop (i=0; i < rand(1000); ++i) {enqueue(rand(0,1000000),NULL)} Loop (i=0; i < rand(1000); ++i) {dequeue()} } so the total number of elements in the queue are always dynamic but the total enqueue/dequeue number is equal to 1000000.

2.4 number of runs:

I believe that the average of the fastest runs should be used. You can run it for example 10 times with 100 executions and take the best of the 100 for each 10 and then take the average of the 10. To restate: take the fastest execution of 100 runs. Do this 10 times and then average the 10. My intuition is that this method should remove some of the OS uncertainties. It has occured to me to note that the dynamic compiler DOES need some time to 'warmup'

3 List of Directories:

To interpret the results: three lines are printed, each line corresponds to test 1,2,3

Change the numbers and repeats to be smaller if you want it to run faster

3.1 heap - cpythons heapq module

- CPython: run with `python ./test_heap.py`
- Pypy: run with `pypy ./test_heap.py`

3.2 pyheap - python heap

- CPython: run with `python ./test_heap.py`
- Pypy: run with `pypy ./test_heap.py`

The pypy pyheap and heapq should be identical because there is no c compiled heapq in the pypy Jit compiler

3.3 cheap cffi - cffi API mode

- CPython: compile module with `python heap_inter.py` run with `python test_heap.py`
- Pypy: compile module with `pypy heap_inter.py` run with `pypy test_heap.py`

3.4 cheap ctype

run make and then

- CPython: run with `python ./test_heap.py`
- Pypy: run with `pypy ./test_heap.py`

3.5 calendarQ

- CPython: run with `python ./test_heap.py`
- Pypy: run with `pypy ./test_heap.py`

the problem with the calendar queue is that it is not space optimized so the growth/shrink operations take a long time

3.6 fibheap - native python code - fibonacci heap

- CPython: run with `python ./test_heap.py`
- Pypy: run with `pypy ./test_heap.py`

3.7 Lua - holds all the simian for lua

- `Luajit test_heap.lua`

Each test is commented out and can just be ran manually with an external timer like ‘time‘ each test here can also run it like 10 times so i just script via command line to run 10 times, average, and divide by 10

3.8 cheap c

Again comment which function you want to run

- Compile with `gcc -Ofast cheap.c -o cheap`
- or try to compile with other options??
- run with `./cheap`

3.9 misc

There are 4 configurations of 2-tiered queues. binary and fibonacci. I ran these with a bunch of configurations of the LA-PDES benchmark but there was no speedup compared to a single tier queues

4 Results

4.1 Results Macbook Pro (OSX)

Implemenation	Static priorities	Random Priorities	Interleaved Operations
CPython C Heapq	3.434	9.167	3.720
CPython Python Heapq	8.417	13.261	7.789
CPython CFFI (API mode)	3.541	5.974	5.227
CPython Ctypes	6.349	12.4	10.2
Pypy heapq (default module)	0.855	2.246	0.371
Pypy Python priority Queue	0.873	2.243	0.371
Pypy CFFI	1.903	2.832	1.660
Pypy Ctypes	18.4	18.5	18.5
Pypy Calendar Queue (python)	1.049	3.158	15.371
Pypy Fibonocci Heap (python)	1.353	9.947	1.259
LuaJIT Pure Lua	0.933	2.271	0.574
Pure C code	0.850	1.588	0.663

4.2 Results in Linux machine

I only did 10/10 for number of runs so it didnt take all day, If you run it 100/100 it takes like a week.

Implemenation	Static Priorities	Random Priorities	Interleaved Operations
CPython C Heapq	2.215	6.398	2.302
CPython Python Heapq	4.696	8.743	4.213
CPython CFFI (API mode)	2.502	4.061	2.789
CPython CTypes	2.475	3.35	3.421
Pypy heapq	0.814	2.694	0.366
Pypy python priority Q	0.774	2.526	0.366
Pypy CFFI	1.673	2.684	1.418
Pypy Ctypes	6.96	7.183	7.112
Pypy Calendar Queue	1.203	3.796	15.136
Pypy Fibonocci heap	1.196	10.844	1.365
LuaJit Pure Lua	0.968	2.533	0.872
C code	0.986	1.759	0.8334