University of Washington - Bothell

CSS 600 – Independent Study.

Wave2D – Dynamic Multi-Threaded Load Balancing

Experiment Report

Submitted by:

Bhargav Mistry

Contents

Introduction:

The application Wave2D was modified and dynamic load balancing algorithm was implemented between multiple threads. The document is a report which depicts the experiment results and conclusions.

Platform:

The application is implemented using Java language and is compiled and executed on Linux platform.

Test Machine configuration:

The configuration of the test machine used is:

Operating system: GNU/Linux
Processor: i686 athlon.
Hardware platform: i386

Processor details:
3.6 GHz
Quad core
1024 * 2 KB cache.

RAM: 2GB

Test execution clock time:

Following are the test execution details and the machine activity. From the below command details we understand that there was some activity going on, on the test machine so the results might show the extra noise:

```
[bhargavm@uw1-320-16 final]$ date; who;
Wed Aug 17 01:43:06 PDT 2011
fumik    pts/0        2011-08-16 08:42 (c-67-170-75-39.hsd1.wa.comcast.net)
dslab    pts/1        2011-08-16 19:05 (c-67-168-134-77.hsd1.wa.comcast.net)
bhargavm pts/2        2011-08-16 23:59 (50-47-27-83.evrt.wa.frontiernet.net)
[bhargavm@uw1-320-16 final]$
```

Wave2D dynamic load balancing overview:

Wave2D program accepts a simulation space size and divides the space equally among given total number of threads. These threads perform the computation on respective data slices. The current Wave2D load balancing algorithm checks the total time used by a particular thread, in the processor and determines the highest time consuming thread. The algorithm then dynamically re-computes the boundary of that particular slice and at the same time corrects the boundaries of adjacent threads. This way when the slice size of the highest time consuming thread is reduced, the total time taken by that thread would be less in the next cycle and the extra load would be shared among adjacent threads.

Hence dynamic load balancing is achieved by changing the boundaries dynamically during runtime.

Boundary change decision making:

The algorithm currently computes an average of all the times taken by all threads. Then a difference factor is determined by subtracting the average from the actual time. The values are then sorted and thread with highest difference factor is determined as the candidate for the boundary change. The application is further modified to accommodate one more runtime argument which determines how many top time consuming slices should be load balanced. E.g. if the argument passed is “1” then the highest thread will be load balanced, if the argument passed is “2” then the top 2 highest time consuming threads will be balanced and so on.

The Wave2D wave computation logic is modified and a condition is placed around it such that, it will compute only if the 3 cells at the same position across the 3 vertical layers are not same. This means that if the 3 cells which show the depth of the wave across three layers, are not same then only that part of the wave will be computed otherwise the computation will not happen if the cells have same values. This way we save some amount of cpu cycles.

Wave2D load balancing source code:

Pls refer next page.

Test Results :

1) Execution using 0 priority load balancing:

java -Xmx512m Wave2D 500 2000 2000 4 0
arguments : simulation size : [500] time : [2000] time_interval : [2000] nThreads : [4]
Total cpu time take by thread : [1] = 11690 ms lowerLimit:[375]upperlimit:[499]
Total cpu time take by thread : [9] = 11370 ms lowerLimit:[125]upperlimit:[249]
Total cpu time take by thread : [8] = 11540 ms lowerLimit:[0]upperlimit:[124]
Total cpu time take by thread : [10] = 11410 ms lowerLimit:[250]upperlimit:[374]
Average : 11502.5

The above test is executed with priority value as 0 (see last argument). This means that the program will run without doing any dynamic load balancing.
The average cpu time take by all the threads is : 11502.5

2) Execution using 1 priority load balancing:

java -Xmx512m Wave2D 500 2000 2000 4 1
arguments : simulation size : [500] time : [2000] time_interval : [2000] nThreads : [4]
Total cpu time take by thread : [9] = 12750 ms lowerLimit:[244]upperlimit:[337]
Total cpu time take by thread : [1] = 5520 ms lowerLimit:[433]upperlimit:[499]
Total cpu time take by thread : [8] = 12870 ms lowerLimit:[0]upperlimit:[243]
Total cpu time take by thread : [10] = 8000 ms lowerLimit:[338]upperlimit:[432]
Average : 9785

The above test is executed with priority value as 1 (see last argument). This means that the program will run without doing any dynamic load balancing.
The average cpu time take by all the threads is : 9785

3) Execution using 2 priority load balancing:

java -Xmx512m Wave2D 500 2000 2000 4 2
arguments : simulation size : [500] time : [2000] time_interval : [2000] nThreads : [4]
Total cpu time take by thread : [10] = 9060 ms lowerLimit:[337]upperlimit:[444]
Total cpu time take by thread : [1] = 4430 ms lowerLimit:[445]upperlimit:[499]
Total cpu time take by thread : [9] = 12590 ms lowerLimit:[178]upperlimit:[336]
Total cpu time take by thread : [8] = 14000 ms lowerLimit:[0]upperlimit:[177]
Average : 10020

The above test is executed with priority value as 2 (see last argument). This means that the program will run without doing any dynamic load balancing.

The average cpu time take by all the threads is : 10020

4) Execution using 3 priority load balancing:

java -Xmx512m Wave2D 500 2000 2000 4 3
arguments : simulation size : [500] time : [2000] time_interval : [2000] nThreads : [4]
Total cpu time take by thread : [8] = 11410 ms lowerLimit:[0]upperlimit:[151]
Total cpu time take by thread : [1] = 1080 ms lowerLimit:[2189]upperlimit:[2188]
Total cpu time take by thread : [9] = 25000 ms lowerLimit:[152]upperlimit:[499]
Total cpu time take by thread : [10] = 1000 ms lowerLimit:[500]upperlimit:[499]
Average : 9622.5

The above test is executed with priority value as 3 (see last argument). This means that the program will run without doing any dynamic load balancing.

The average cpu time take by all the threads is : 9622.5

Conclusion:

With 0 threads: 11502.5
With 1 thread: 9785
With 2 thread: 10020
With 3 thread: 9622.5

From the above tests we can conclude that performance degrades when using top 2 threads as compared to 1 thread and 3 threads.

Improvement analysis:

Top 3 threads Vs. 0 Threads = (11502.5/9622.5) = 1.19 times.
Top 2 threads Vs. 0 Threads = (11502.5/10020) = 1.14 times.
Top 1 thread Vs. 0 Threads = (11502.5/9785) = 1.17 times.

Discussion - Limitations:

Following are the discussion/limitation points derived from the above analysis:

- The experiment was executed when other processes were running on the machine hence we cannot for sure conclude that the above results are perfect. The same test should be executed again when the machine is completely idle and no other user is logged in.
- During my rigorous testing with scenario where top 3 threads are used for load balancing, the application did crash couple of times and this issue is intermittent. The first two scenarios with 1 thread and 2 thread load balancing works fine. This area needs more testing and/or logic implementation change. This item is taken as future work.
- When total execution time of the application is taken into consideration, there is no performance improvement. This is because of the user code overhead. This overhead needs to be reduced by further optimizing the code. This item is taken as future work.
- A threshold was not determined for this application as to when the dynamic load balancing should kick in. The application by default, load balanced after every cycle. This must be changed in future.
- One idea to completely separate out the extra overhead from the main application is to execute the load balancing logic as a separate daemon thread altogether on a separate core. This thread then will communicate with the main thread via synchronous mechanism. This is taken up as future research item.