High Performance Computing

Homework #2: Part B

Due: Saturday February 17 2015 by 11:59 PM (Midnight)

Email-based help Cutoff: 5:00 PM on Mon, Feb 16 2015

Maximum Points For This Part: 10

Objective
The objective of this part of the homework is used to use a given benchmark program to assess the performance impact of using the following two API methods to access individual elements in a `std::vector`: ▪ Using the `std::vector::at()` method (tip does bounds checking like Java/Python) ▪ Using the `std::vector::operator[]` method (tip does not do bounds checking)

Instructions:

1. Download the supplied benchmark program and study the code carefully. See if you are able to answer the following questions:
   a. How and why was the test vector size chosen?
   b. Why does the benchmark repeat the test many times?

2. Ensure the benchmark is calling the appropriate method, i.e., sum or sum_at, depending on the API method you would like to test.

3. Compile the program with optimizations enabled (eg: `-O3` for `gcc` or `–fast` for `icpc`)

4. Ensure you use an interactive job on Red Hawk to record timings and fill in this report.

5. Once you have filled-in the report, save it as a PDF file and submit.

11:59 PM (by Midnight) on Tue Feb 17 2015

Name: Nick Contini

Experimental Platform

The experiments documented in this report were conducted on the following platform:

Component	***Details***
CPU Model	Intel® Xeon(R) CPU E5520 @ 2.27GHz
CPU/Core Speed	2260.983 MHz
Main Memory (RAM) size	24725392 kB
Operating system used	Red Hat 4.4.6-4 w/ Linux 2.6.32-279.14.1.el6.x86_64
Interconnect type & speed (if applicable)	
Was machine dedicated to task (`yes`/`no`)	yes
Name and version of C++ compiler (if used)	icpc 15.0.0 20140723
Name and version of Java compiler (if used)	
Name and version of other non-standard software tools & components (if used)	

Performance Analysis

The vector size is chosen to most efficiently use the cache (L3 cache to be specific). By using the cache as much as possible, the program can take the most advantage of SIMD and other parallelizations. The test is run many times to ensure accurate time analysis. If the test were run once, the results may be inaccurate or too small to reach an intelligible difference in speed.

11:59 PM (by Midnight) on Tue Feb 17 2015

Document the statistics collated from your experiments conducted in the table below. Delete the first row with fictitious data included just to illustrate an example.

`std::vector` Element Access Mode	User Time (sec)	Elapsed Time (sec)	%CPU	Max resident size (KB)
Using `at` method (#1)	1.61	1.63	98	17376
Using `at` method (#2)	1.76	1.77	99	17360
Using `at` method (#3)	1.61	1.62	99	17376
Averages (of 3 runs)	**1.66**	**1.67**	**98**	**17370**

`std::vector` Element Access Mode	User Time (sec)	Elapsed Time (sec)	%CPU	Max resident size (KB)
Using `operator[]` (#1)	0.32	0.33	99	17408
Using `operator[]` (#2)	0.34	0.34	99	17392
Using `operator[]` (#3)	0.32	0.32	99	17376
Averages (of 3 runs)	**0.33**	**0.33**	**99**	**17392**

Using the above chart develop a report (10 sentences) discussing the following performance aspects (use as much space as needed):

- What is the functional difference between the use of at() method versus operator[]?
- What is the performance difference between the two approaches?
- When should a programmer use one versus the other?
- What are the implications on other languages (such as Java/Python) with references to accessing values in a `vector`-like data structure (such as: `ArrayList` in Java)

The functional difference between at() and the [] operator is that at() has boundary checking, whereas [] just blindly accesses a address. The boundary check adds a significant amount of time to the method, since there must be added code to not only check the boundary, but also to respond appropriately if the index is outside of the boundaries or not. This makes the [] operator around 5 times faster that at(). This means that if the programmer is sure that the index he is accessing will be valid, he should use the [] operator. Otherwise, the at() method will be much safer. This also implies that other languages that have special OutOfBoundary exceptions (such as Java) will most likely have similar implementations to at() for their ArrayList and other objects, meaning they also will perform slower than C++.

11:59 PM (by Midnight) on Tue Feb 17 2015