The program:

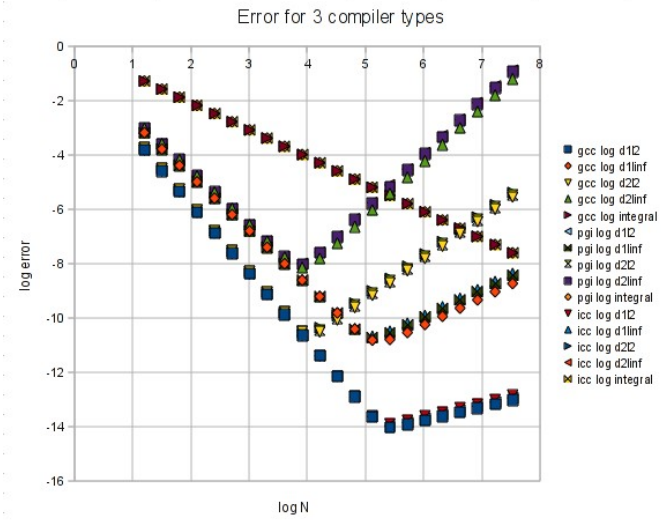
- Is serial (runs on a single core).
- Performs benchmarks a simple function (sin). This allows for error analysis because the integral and derivatives to be computed in closed form.
- Computes first-order derivative (three-point method).
- Computes second-order derivative (three-point method).
- Computes integral (trapezoid method).
- Computes first- and second-order derivatives in closed form.
- Calculates and prints errors (L-2 and L-Inf norm).
- Passes Valgrind tests for memory errors.
- Performs the above for N ranging from 2^4 to 2^{25}, in multiples of 2. This allows for performance analysis.

The machines:

- A local laptop
  - Intel Core2 Duo P8400
  - 3GB RAM.
  - gcc version 4.4.1
- Tufts Cluster:
  - Intel Xeon E5440
  - 16GB RAM
  - gcc version 4.1.2
  - pgCC version 10.0-0
  - icc version 10.1
- Ranger:
  - AMD Opteron 8354
  - 32GB RAM
  - gcc version 3.4.6
  - pgCC 7.2-5
  - icc version 10.1
- Kraken:
  - AMD Opteron 285
  - 8GB RAM
  - gcc version 4.1.2
  - pgCC 9.0-4

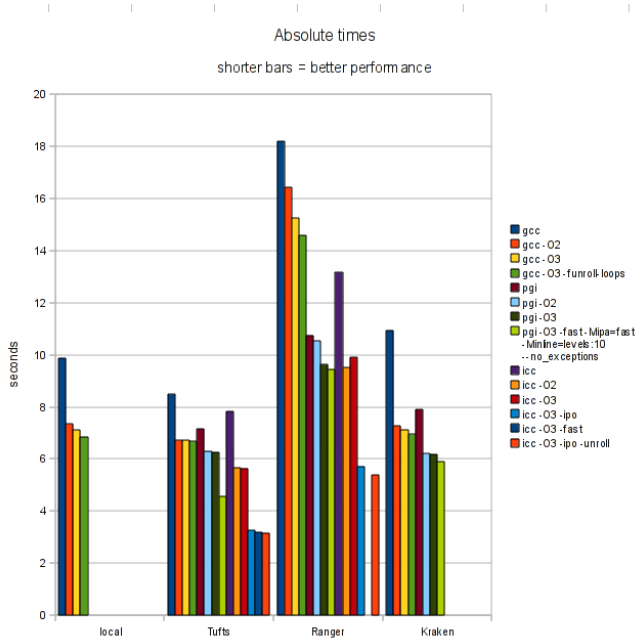
The results:

- Notes:
  - GCC (various versions) was used on all machines with all relevant optimization options (see the time graphs below).
  - PGI (various versions) was used on Tufts, Ranger, and Kraken, with all relevant

optimization options.
    - ICC (various versions) was used on Tufts and Ranger. It was not available on Kraken. Additionally, because Ranger uses an AMD CPU, the "-fast" option of ICC did not produce working code on this machine.
- Errors plotted on a log scale with respect to N (plotted on a log scale also):
    - 

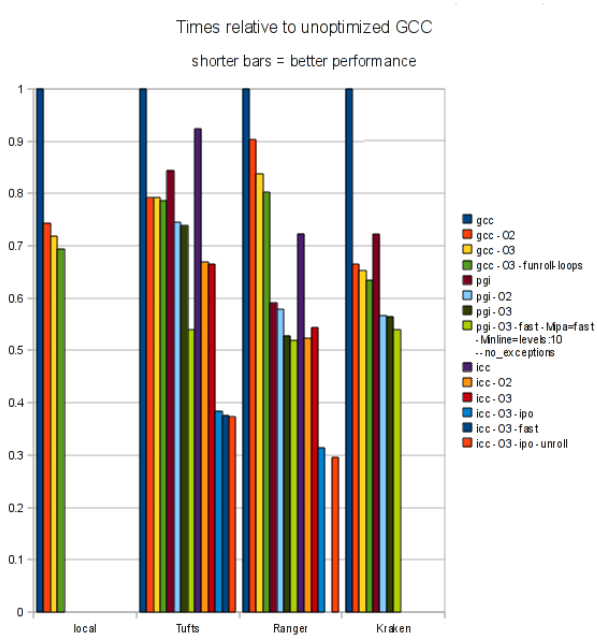
    - For the integral, the error keeps getting lower as we increase N (decrease Δx). We are not reaching the point where double precision starts to cause problems.
    - For the derivatives, the error keeps getting lower as we increase N, but beyond a certain point it begins to climb again. At this point, double precision becomes insufficient for the calculation, and it starts introducing larger errors.
    - The results are equal and consistent between the three compilers and various optimization options, BEFORE the low-point in the curve is reached. AFTER this point, the results differ slightly between GCC and PGI, and between unoptimized ICC (which is like GCC) and optimized ICC. The reasons for this could be anything between different runtime C libraries to different hardware instructions being used for floating point operations.

- Run times:
  - Absolute run times:



  - These are measured CPU times. CPU time was measured instead of wall clock time to avoid variations in results due to CPU time being shared between many users. (For example, the Ranger login node had over 220 users at the time the tests were run.) System time (for IO, memory allocation, process cleanup etc.) ranged between 900ms and 2s on the various machines.
  - It was found that the PGI and ICC compilers are AMAZINGLY better at optimizing this simple program than GCC. This is probably because GCC performs no (or limited) interprocedural optimization, and it does not use Intel-specific vector instructions, opting instead to make the compiled program work on any compatible platform. On another interesting note, PGI and the Intel compiler both printed out messages about which optimizations were performed (loop vectorization for instance), whereas GCC did not output any such messages.

- Relative run times:



- Times were also measured relative to the runtime of unoptimized GCC. The Intel compiler achieved the most impressive results (more than three times faster run times).

Resources used:

- PGI User's Guide (section on Optimization): http://www.pgroup.com/doc/pgiug.pdf
- Quick-Reference Guide to Optimization with Intel Compilers version 11: http://cache-www.intel.com/cd/00/00/22/23/222300_222300.pdf