Tuning a Timer

by

Rudolph A. Krutar

March 31, 1998

The Processing Graph Method Tool (PGMT) product is being released under the GNU General Public License Version 2, June 1991 and related documentation under the GNU Free Documentation License Version 1.1, March 2000.
http://www.gnu.org/licenses/gpl.html

Purpose:

The purpose of this note is to explain some fine points regarding performance measurement, especially the timing of computer software. As with any tool, understanding what the tool does and what it does not do is important to practical use of the tool. Therefore it is necessary to understand in detail what a timing function really does.

Problems:

This study arose because we noticed some anomalies in trying to time some MPI[1] programs using the freeware MPICH library[2]:

- Different processes running on the same processor would frequently report different timer resolutions.
- Timings of MPICH processes were inconsistent and scattered.

When running MPI programs under the shared memory option of MPICH, we have each process initialize various constants. In particular, the processor's timer resolution is initialized. This resolution is the so-called tick value. As all processes under shared memory run on the same processor, all processes should determine the same tick value. As they do not, the program chtick.c that calculates the tick value is not a function. An alternative program is presented below.

Unfortunately, the mpich library code for determining the timer resolution (the so-called tick) fails because it takes the minimum of several readings. Reading the high resolution timer once takes several ticks. Independent determinations of the resolution are therefore not consistent. We correct this problem by defining a tick function that takes the greatest common denominator of several short, independent unsigned integer time intervals. Each interval calculated is the unsigned difference of two integer times, accurately determining any interval shorter than the recycling time of the timer. For example, 3-1022 = 5 given a timer that returns a ten-bit unsigned integer time. The greatest common denominator of a few such differences will converge to the timer resolution.

```
/* This returns a value that is probably
 * the best value for minimum integer val that
 * could be returned.  It makes several
 * separate stabs at computing the tick value,
 * and calculates the greatest common divisor
 * of all the differences.
 */

fulltime RK_tick()          {
      fulltime To, dT, gcdT = (fulltime)0;
      int    cnt, icnt;

      To = RK_time((fulltime)0);           /* 0 == product of all */
      for (icnt=0; icnt<10; icnt++)    {
          cnt = 10000;                     /* To == TimeJ */
          while (cnt--)          {
             dT = RK_time(To);             /* dT == TimeK - TimeJ */
             if (dT) break;      };
             /* assume dT is elapsed time  * dT%tick == 0 */
             /* To += dT;                  /* To == TimeK */
             do { gcdT %= dT;              /* 0 <= gcdT=D0 < dT=D1 */
                  dT -= gcdT;              /* 0=gcdT-D0 < dT=D1-D0 */
                  gcdT += dT;              /* dT=D1-D0 < gcdT=D1 */
```

```
                 dT = gcdT - dT;          /* dT=D0 < gcdT=D1 */
               } while (dT);              /* dT == 0 */
            /* gcdT%tick == 0 */        };
  /* P(gcdT == tick) is high */
  return gcdT;              }
```

A program that was supposed to measure round trip interprocess communication speed showed that communicating with a different process is sometimes two orders of magnitude slower than a process communicating with itself. However, the measurements show much variance. What is the source of that variance? Much of that variance arises largely from system interrupts, such as those that maintain the console screen saver.

What do our measurements mean, if anything? The analysis below shows how to extract meaningful results from noisy timing data.

Approach:

I defined a benchmark function that should run in linear time as a function of its parameter, which is of the same integer type as the timer uses to represent times:

```
/* This procedure counts to N and stops.
 * It provides a standard for testing timers.
 */
typedef hrtime_t fulltime;
    void RK_count( fulltime N )      {
         while(--N) /*no-op*/;       }
```

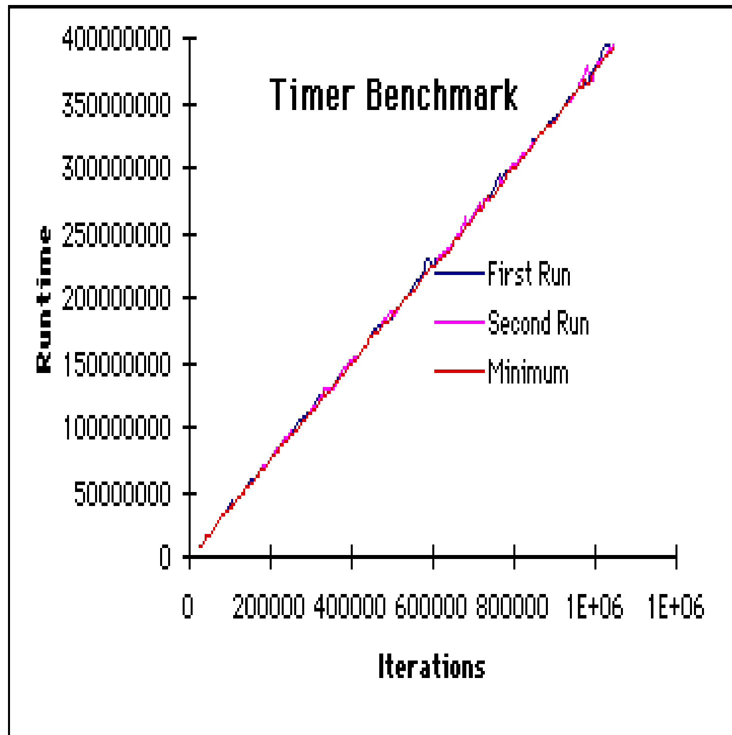
My approach was:

1. to time a program that should run in time that is a linear function of a parameter N,
2. to correlate actual run times with various values of N (mathematically and graphically in Excel),
3. to refine the benchmark software and analytic techniques as necessary.

Ideally, the graph should be linear, with the Y-intercept showing the overhead independent of the calling program's parameter. Other properties of the timer should be apparent in the shape of the graph.

Experiments:

The first experiment obtains timings T(N) for various values of N. Various bumps in the graphs for two runs can be attributed to sources outside the program (such as interrupts), because the minimum of the two curves appears linear. Any run will take a certain required amount of time to accomplish its own task. Any time that one run takes which another does not, was not required by the computation. We should therefore combine data for the same experiment from several runs by taking the minimum value rather than the average value, which has no significance in the timings we are trying to determine. We copy the timing data to an Excel spreadsheet for analysis and plotting:

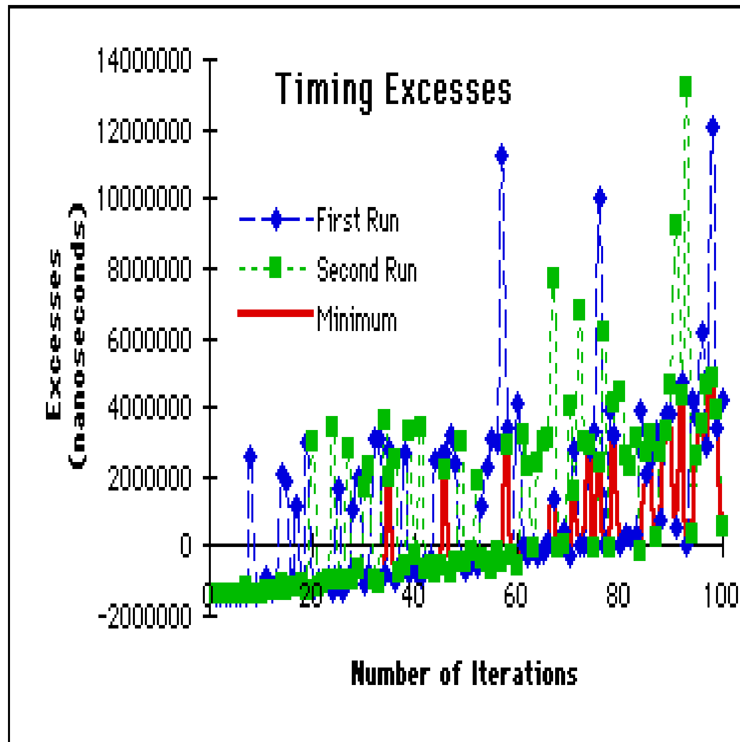


Plotting $T1_N$, $T2_N$, ..., Y_N=minimum(Tk_N for k=1,2,...) versus X_N =N for various large values on N shows a diagonal line with small bumps. The bumps are small relative to the range of T(N). A least squares fit of the data to a straight line is obtained statistically with:

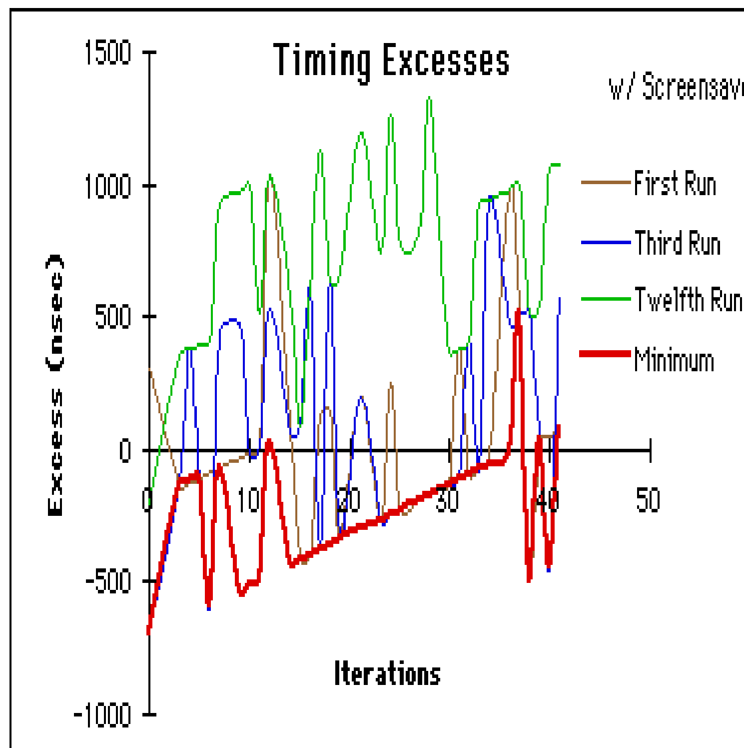
- $Y_N = f(X_N) = Yo + X_N * dY$ approximately
- dY = covariance(X,Y)/variance(X) = extra time required when N is incremented
- Yo = mean(Y) - dY*mean(X) = overhead time required independent of N

as shown in an old timing study[3].

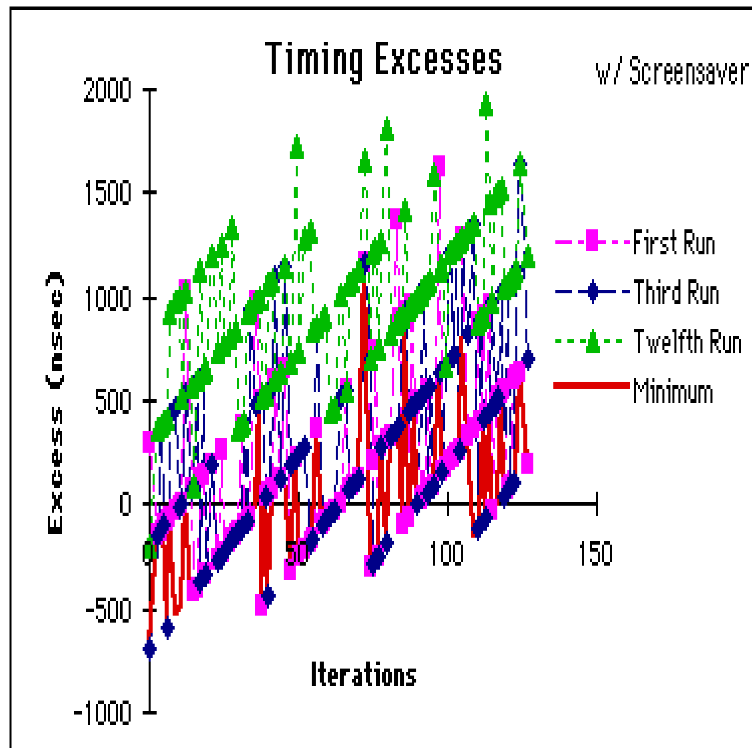
Small excess times over the straight line projection of required times, $T1_N - Y_N$, $T2_N - Y_N$, ..., should plot as a horizontal line, greatly magnifying the bumps that represent the excess times:



We expected the excess times graph to have a horizontal minimum, but it started negative and rose slowly. Expecting that interrupts are more likely during long runs, we increased the number of runs (showing only a few below) and reduced the range of N:

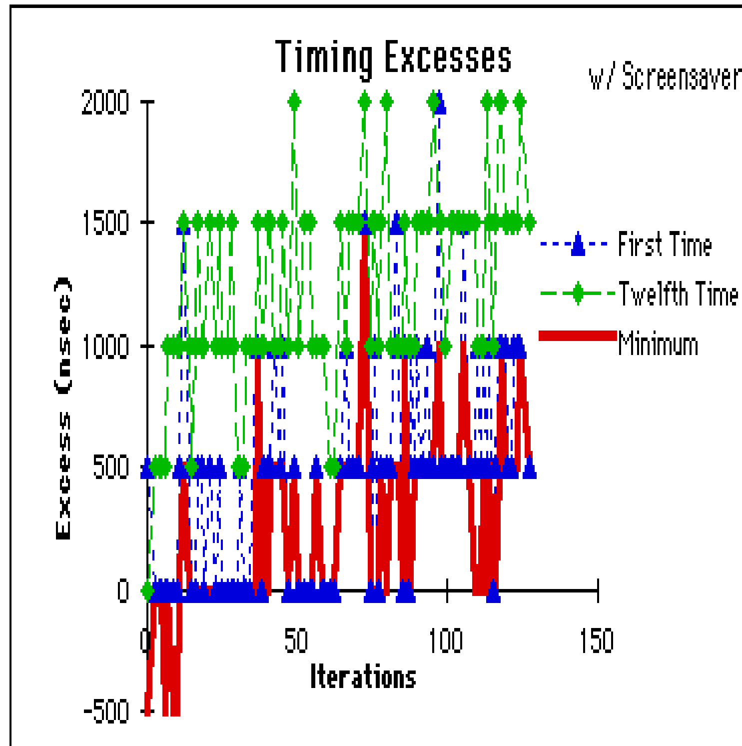


Seeing the discretization, but not desiring the rounded curves, we marked the data points and suppressed data smoothing in Excel's graph plotting package:



Why are the steps not horizontal? We eventually determined that roundoff error relating to the timer resolution causes this strange behavior as a kind of Moiré pattern. The timer resolution (500 ns) is a little larger than the unit cost (359 ns per step), resulting in a kind of beat. This problem depends on the resolution of the timer. Correcting this problem depends on determining the resolution of the timer.

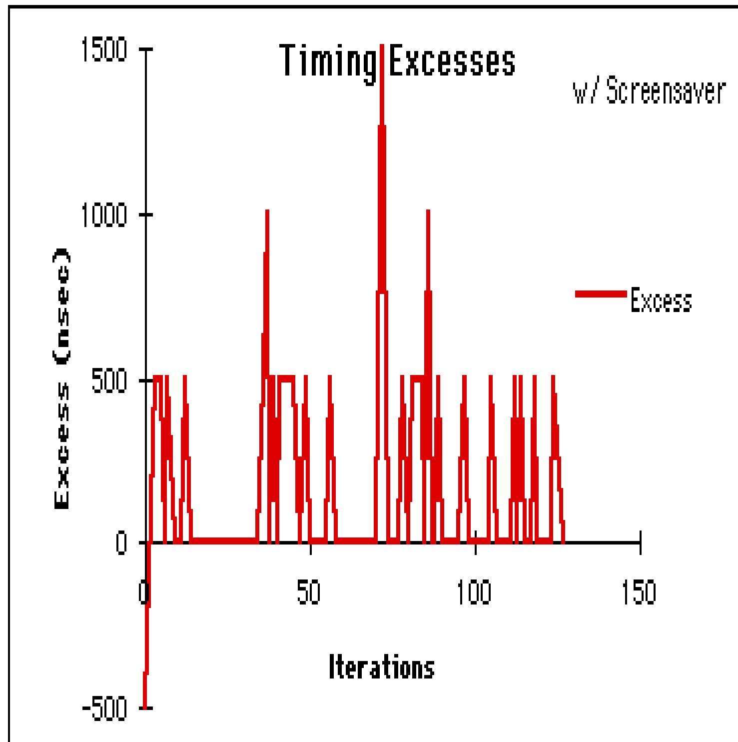
Changing the linear subtrahend to a linear step function cures the above Moiré problem.



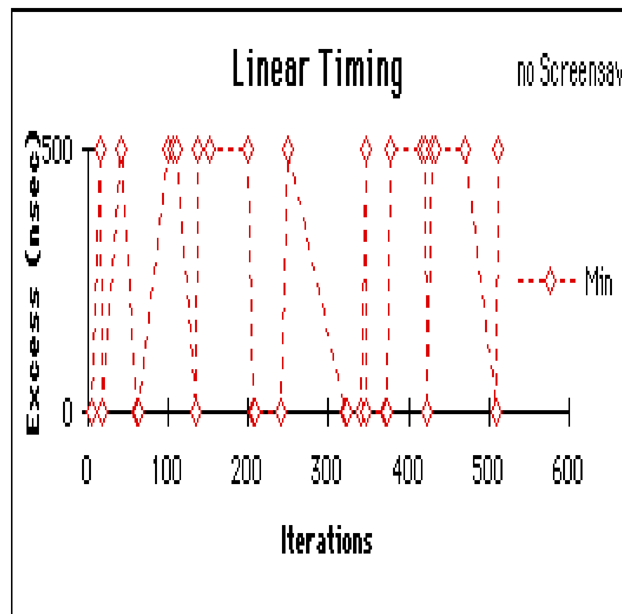
Some of the excesses are negative because our estimate of the required time is inflated by interrupts that support the console screen saver of the host processor. Interrupts are more likely to occur when longer benchmarks are running; these interrupts tend to increase timings for longer runs. Projections for very short runs will then be overestimated, leading to negative excess times. Finding a better fit to the data alleviates this problem:

- Given data $X_N > 0$, $Y_N > 0$ for N = 1,...,S1
- Find Yo, dY such that $f(X_N) = [Yo + X_N * dY] <= Y_N$,
- where $\sum (Y_N - f(X_N)) = \sum Y - Yo * \sum 1 - dY * \sum X$ is minimal,
- which occurs when Yo + dY*mean(X) is maximal.

This is a linear programming problem resulting in a “least values fit” with no negative excesses. A general solution to such a problem combines data from many trials without requiring specific trials to be rerun. The data used above (excepting case $X_N = 0$, which is ignored) results in a good least values fit (the X axis) lying just under all the data points:



Running on a host without a console removes many interfering interrupts. Increasing the number of runs reduces the overestimates, resulting in a horizontal line for the minimum timing (within the timer resolution), even for a larger range for N:



The timing data can be analyzed as it is collected by determining the lower half of the convex hull of timings as plotted against the iterations parameter. A forthcoming paper[4] will discuss a technique for accurately analyzing such ill-conditioned data.

We have shown how to get accurate timings for a simple benchmark program. We can use the same techniques to get good timing data for other programs. Just define a function that

should require time that is linear with its parameter. Correlating minimal timings with N reveals the unit cost of incrementing N to a greater precision than the timer resolution suggests. The Y intercept indicates all the fixed overhead in doing the timing.

For example, defining a program that performs a block copy of N bytes results in timings that reveal the block copy cost per byte. Declaring all buffers to be of some large fixed size makes any cost related to the buffer size constant.

For another example, consider a program that sets up some situation a hundred times, performing some action to be measured in N of those situations. Correlation will reveal the actual time required for the action as the slope of the last-values fit to the data. This method lets us time parts of a program that are not easy to run separately.

Conclusions:

The above study resulted in the following conclusions:

1. Independent timings should not be averaged.
2. The time required by a program is the smallest timing possible.
3. Excess over required time represents system overhead.
4. The minimum possible nonzero timing may exceed the timer resolution.
5. The tick value divides every small time interval, so the greatest common denominator converges to the tick value.

The average of independent timings is just the required time plus the average noise. In analyzing a program, we need to know the direct effects of that program. In predicting typical performance of a program, we can include average noise separately.

No timing of a program can show less time than the time required by the program. The smallest timing therefore is closest to the required time.

Subtracting the estimated required time (in ticks) from the measured time of a program shows the excess, which is noise due to system overhead. The excess is ideally zero, and plotting that excess against the parameter N magnifies the discrepancies.

An ideal timer would have very high resolution, so that accessing the timer may take many ticks. We assume that timer updates are always accomplished by adding some multiple of the tick value to some unsigned K-bit counter, so small differences will be unsigned multiples of the tick value.

Expectations:

The MPICH program chtick.c should be repaired, perhaps as shown in my tick function. We will post this note as a web page and notify various interested groups of its existence.

Our timing studies will surely benefit from this analysis. The PGMT scheduler will need accurate timing data to do its job well.

This note tells how to improve timing studies. Applying its lessons will ultimately result in better understanding and improved software performance.

Acknowledgments:

Roger Hillson has reviewed this note and made good suggestions.

[1] William Gropp, Ewing Lusk, Nathan Doss, Anthony Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard", white paper, Argonne National Laboratory and Mississippi State University.

[2] Peter S. Pavheco, *Parallel Programming with MPI,* Morgan Kaufmann Publishers (San Francisco) 1997, “Taking Timings”, pp.254-256.

[3] Rudolph A. Krutar, "Preprocessor Timing Study", NRL Technical Memorandum 5150-141:RAK:uv, Appendix D: Mathematical Analysis, 1982 Sep 10.

[4] Rudolph A. Krutar, “Flat Convex Hull”, forthcoming.