It's About Time: Multi-Resolution Timers for Scalable Performance Debugging

James B. White III
Oak Ridge National Laboratory

May 9, 2007

Abstract

Traditional performance profiling of highly parallel applications does not always give enough information to diagnose performance bugs, particularly those caused by load imbalances and performance variability, yet the data files for such profiling can grow linearly with parallel task count. In response to these limitations, I have developed application timers designed to limit data and reporting volumes at high task counts without dispersing the signals of load imbalance and performance variability. I will describe use of these timers to diagnose actual performance bugs running the Parallel Ocean Program on a Cray XT4.

1 Introduction

A goal of high-performance computing (HPC) is to accelerate computation, so developers of HPC software often expend significant effort to optimize the performance of that software on the target supercomputers. These optimization efforts are typically guided by timing the execution of the software components.

The main strategies for timing HPC software are automated tools and explicit calls to timing libraries. Each strategy has advantages and disadvantages. The advantages of automated tools are that they require no changes to the software source code and they may provide a wide range of performance-analysis capabilities. Such tools are often dependent on a particular hardware vendor, however, or they may be unavailable on the newest supercomputers. Also, timing data for highly parallel runs can overwhelm filesystems and post-processing tools.

A prominent example of an automated performance tool is CrayPat [1], available on current Cray supercomputers. CrayPat can automatically instrument parallel executables, so that these executables generate a performance-data file when run. CrayPat tools use this file to generate performance reports or provide interactive analysis through an X-Window graphical user interface.

Unfortunately, the performance-data file grows linearly with the number of parallel tasks in the generating run. The reporting and analysis tools, which use a single process, can run out of memory with files generated using a few thousand tasks. For example, "`pat_report`" runs out of memory, generating no report, for runs of the Parallel Ocean Program (POP) [2] using 4500 tasks on Jaguar, the Cray XT3/4 at Oak Ridge National Laboratory (ORNL). Thus it is currently unusable for runs using a moderate fraction of the 23,000+ processor cores available on Jaguar.

Timing libraries, conversely, can support runs of many thousands of tasks. They can also be more portable, often distributed with the source code of the parallel application. Because timing libraries can target specific regions of software, they can also have lower overhead than instrumentation by automated tools.

Timing libraries typically provide simple text out-

put at the end of a parallel run; this can be an advantage or disadvantage. The output may be short and easy to understand, but it may be inadequate to diagnose performance problems. And perhaps the main disadvantage of timing libraries is the need to modify the source code of the target application.

A prominent HPC application that uses timing libraries is the Community Climate System Model (CCSM) [3]. CCSM has separate model components for the Earth's atmosphere, ocean, land, and sea ice, where each component runs on a separate group of processors. Timers in the CCSM coupler guide load balancing of the components among available processors, but they do not provide adequate information to optimize each parallel component individually. Thus components may have their own timing library.

For example, this is the case for the Community Atmosphere Model (CAM) [4], the CCSM atmosphere component. Each parallel task creates a separate file with its timer output, so load imbalances are identifiable. Combing through the files to find such imbalances can be difficult, however, and the large number of files do not help diagnose performance variability within a task.

POP, the ocean component in CCSM, also has its own timers. POP appends a timer report to its standard output, a report that is short and easy to analyze. But it includes only maximum, minimum, and average totals for each timer, and this level of detail is inadequate to diagnose load imbalance and performance variability within POP.

In addition to acting as a component of CCSM, POP runs stand-alone ocean simulations. It was while optimizing POP for stand-alone execution that I experienced difficulty analyzing performance with CrayPat and the built-in POP timers. In response to this difficulty, I developed an experimental timer library for POP, one designed to identify load imbalance and performance variability. The following sections describe this library and my experience using it with POP.

2 Timer Interface

Initial design decisions for the timer library regarded the user interface. One possibility was to always refer to each timer by a unique character string. For example, the following Fortran subroutine calls would start and stop the timer called "`physics`".

```
call start_timer("physics")
call stop_timer("physics")
```

This interface has the advantage of ease of use; timers can be added without variable declarations and without passing timer arguments. It has the disadvantage of overhead; each call to the timer library requires a string search or hash. String search can have startling overhead for vector systems, such as the Cray X1E.

An interface using a timer type or integer handle avoids this overhead. Each timer is first declared and initialized, perhaps together with other timers in a single module or procedure.

```
type(timer) :: t
t = new_timer("physics")
```

The start and stop calls then use the variable "`t`", which provides direct access to the timer state.

```
call start_timer(t)
call stop_timer(t)
```

This interface has the disadvantage of inconvenience. Timer calls may no longer be local to the regions they measure, because the timers may need to be declared and initialized in a parent procedure. The interface can lead to a proliferation of timer arguments or module "`use`" statements.

I decided to use a compromise interface. Timers are identified by a unique integer, and that integer resides in a "`save`" variable in each procedure using the timer.

```
integer, save :: t = 0
call get_timer(t, "physics")
call start_timer(t)
call stop_timer(t)
```

On the first call to "`get_timer`" within the procedure, the library does a string search and, if necessary, creates a new timer, assigning "`t`" to the resulting timer. The value of "`t`" is then set, so subsequent

calls to "get_timer" just confirm that "t" matches the timer with name "physics". This confirmation requires a single string comparison. These calls can all be local to the procedure, with no argument passing or separate initialization.

The overhead associated with "get_timer" is then as follows. The very first call allocates and initializes timer space. If the timer argument is not set, the call performs a linear search. If the string argument is not matched, the call creates a new timer. If space exists for the new timer, the operation is very lightweight; otherwise the call re-allocates larger timer space and copies over the existing timers. If the timer argument is already set, the call just confirms that the timer name matches the string argument.

The resulting overhead for "start_timer" is minimal. It first checks the argument and the state of the target timer, using two "if" statements, and then calls the Fortran intrinsic subroutine "system_clock". The matching "stop_timer" first calls "system_clock" and then checks its argument and the state of the target timer. It then updates ten state variables for the timer, as follows.

1 Turn off the timer.

2 Increment the timer event count.

3 Increase the total tick count for the timer.

4,5 Update the maximum tick count and the corresponding event number.

6,7 Update the minimum tick count and the corresponding event number.

8,9 Update the second-largest tick count (2nd max) and the corresponding event number.

10 Find the appropriate bin in the timer's event histogram and increment its counter.

Pat Worley of ORNL suggested tracking the 2nd max for each timer. The runtime measured by a timer may include initialization work during the first event, so the 2nd max helps measure the variability of the remaining, non-initialization events. It might also prove useful for filtering other singleton outliers among timer events.

The event histograms are the feature that originally motivated development of these experimental timers. The bins of each histogram count events in particular tick ranges, and each histogram has bins at multiple resolutions. Each histogram has bins for each of 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 ticks, and then bins for each of the tick ranges 10-19, 20-29, ..., 90-99, and then 100-199, 200-299, ..., 900-999, and so on, with increasing order of magnitude, up to the range nearest the value of "count_max" returned by "system_clock". To find the right bin, "stop_timer" performs a linear search starting from the bins with the smallest tick counts. The search finds the right power of ten and then performs a single integer division to find the right bin within that power of ten. Thus the search performs one "if" test for events in the tick range 0-9, two for events in the range 10-99, and so on.

3 Reporting

A call to "report_timers" generates a timing report.

```
call report_timers(unit,comm)
```

The call creates a report for all the tasks in the MPI [5] communicator "comm", and the root task of "comm" prints the report to Fortran I/O unit "unit".

All the other calls to the timer library are local to each task; no parallel communication occurs in the library until the report. A given timer may be unique to a single task or an arbitrary subset of tasks within the target communicator. The report generator assumes that timers on separate tasks that have the same string name represent the same timer. The first step of the report generator is to identify the global list of timer names.

First each task sorts its local timer names in alphabetical order. A binary tree of MPI communication then creates the global list. A given task first receives the list from its child and merges the list with its own, eliminating repeats. It sends the resulting list to its parent. The root task gets the final sorted list and broadcasts it to all tasks (within the target communicator). Each task then creates a map from global-name indices to local-timer indices. This map

provides the many translations needed to generate the report.

The report starts with a summary profile, which serves as a table of contents for the rest of the report. The summary includes the maximum, minimum, and average, across tasks, of the total time accumulated in each timer. Thus the maximum is the largest total of any single task, not the largest aggregate time associated with a given timer.

Figure 1 shows a report summary from a POP run on Jaguar. Note that the disparity between the maximum, minimum, and average values for "`barotropic_driver`" and "`baroclinic_driver`" indicate a significant load imbalance.

After the summary report come detailed reports for each timer, listed in the same order of timers as in the summary. The first item presented after the timer name is a list of MPI ranks that used that timer. This list could be generated by a tree algorithm that reduces contiguous task lists to ranges as it goes. Each task provides just one true/false datum, however, so the current implementation gathers all the data to the root. The root then generates the rank list, reducing contiguous lists to ranges; for example, "`1,2,3,4,5,6,7,8,9,10,11`" reduces to "`1-11`".

Following the rank list are the following statistics.

events per task The maximum, minimum, and average number of events, across tasks, for that timer, followed by the number of tasks used in computing the average.

longest event The time of the longest single event for any task, in seconds and clock ticks, the task that measured that event, and the number of the event on that task.

max 2nd longest The time of the maximum, across tasks, of the second-longest event for each task, in seconds and clock ticks, the task that measured that event, and the number of the event on that task. Note that this is probably *not* the second longest event across all tasks.

shortest event The time of the shortest single event for any task, in seconds and clock ticks, the task that measured that event, and the number of the event on that task.

longest avg event The maximum across tasks of the average event time for each task, in seconds and clock ticks, the task that measured that average, and the number of events used in computing the average.

shortest avg event The minimum across tasks of the average event time for each task, in seconds and clock ticks, the task that measured that average, and the number of events used in computing the average.

avg event The average time of all events measured by that timer, in seconds and clock ticks, and the number of events used in computing the average.

max total The maximum across tasks of the total time measured on each task, in seconds and clock ticks, the task that measured that total, and the number of events summed in the total.

min total The minimum across tasks of the total time measured on each task, in seconds and clock ticks, the task that measured that total, and the number of events summed in the total.

avg total The average total time measured on each task, in seconds and clock ticks, and the number of tasks used in computing that average.

aggregate total The total time measured by that timer, summed across tasks, in seconds and clock ticks.

Note that only one task and event appear for extreme events, even if more than one event had that value.

Figure 2 shows a detailed report for the timer "`barotropic_driver`" from the same POP run used for Figure 1. It further illustrates the load imbalance found in Figure 1; the difference between maximum and minimum measurements is an order of magnitude.

The statistics described above are not significantly different from the content of other timer libraries. But the report continues with histogram listings, content which I believe is unique to this timer library.

```
*** profile for 360 tasks ***
 max total | min total | avg total | timer
-----------|-----------|-----------|--------------------
    841.11 |    841.08 |    841.09 | step
    760.65 |     70.28 |    181.93 | barotropic_driver
    760.49 |     68.89 |    179.52 | pcg_chrongear iteration
    697.03 |      6.82 |    570.62 | baroclinic_driver
      4.69 |      0.00 |      3.56 | pcg_chrongear preconditioning
```

Figure 1: Summary report from a POP run on Jaguar.

```
*** barotropic_driver
tasks: 0-359
events per task: 128 max, 128 min, 128 avg (360 tasks)
     longest event:     6.262 sec (6262 ticks), task 359, event 1
   max 2nd longest:     6.210 sec (6210 ticks), task 359, event 2
    shortest event:     0.465 sec (465 ticks), task 286, event 103
 longest avg event:     5.942 sec (5942 ticks), task 359, 128 event(s)
shortest avg event:     0.549 sec (549 ticks), task 286, 128 event(s)
         avg event:     1.421 sec (1421 ticks), 46080 event(s)
         max total:   760.647 sec (760647 ticks), task 270, 128 event(s)
         min total:    70.280 sec (70280 ticks), task 285, 128 event(s)
         avg total:   181.927 sec (181927 ticks), 360 tasks
   aggregate total: 65493.801 sec (65493795 ticks)
```

Figure 2: Detailed report for timer "`barotropic_driver`" from the same run used for Figure 1.

4 Reduced histograms

Each task has a separate multi-resolution histogram for each timer. A goal of the experimental timer library is to combine histograms across tasks in a way that makes the report manageable without losing important performance signals in the measurements. The first attempt to combine histograms used the idea of histogram "shape". For a given timer histogram, each task would sort the bins in order of event counts. Histograms with the same sorted order of bins were considered to have the same "shape", and these histograms were combined using maximum, minimum, and average statistics.

In the worst case, the number of resulting histograms could be as large as the number of different orders of the bins. If n is the number of bins, the number of possible orders of bins is $n!$, a number much larger than the largest possible number of parallel tasks. Thus, in the worst case, this combination scheme could lead to no reduction at all in the number of histograms.

The current combination scheme is more aggressive. For each timer, each task finds the bin with the maximum number of events. Histograms with the same maximum-count bin are combined across tasks, again using maximum, minimum, and average statistics. The number of possible histograms is just the number of bins, which is much smaller than the number of parallel tasks for a large job.

Figure 3 shows the histogram report for the timer "`barotropic_driver`" from the same POP run used for Figure 1. The report reduces 360 histograms, one from each of the 360 tasks, down to four. The combined histograms are listed in order of the tick range of the bin with the largest count, so the histogram dominated by the longest events comes first. Each histogram report first lists the tasks associated with that histogram, a list created using the same algorithm as for the timer task list, described earlier. Each nonzero bin then generates a line with the following content.

ticks The range of tick counts for events accumulated in that bin.

max The maximum number of events in that tick range measured by any single task, and the task that measured that number.

min The minimum number of events in that tick range measured by any single task, and the task that measured that number.

avg The average number of events in that tick range, averaged over the tasks listed for that histogram.

The first two histograms in Figure 3 show the tasks with most events of 4000–6000 ticks, and the second two show the tasks with most events of 6–7 ticks. Not only does the report show the load imbalance, it shows which tasks had which loads and how variable those loads were over the history of the run.

Using histograms like these, I was able to determine that the load imbalance comes from the data distribution used in this POP run. Blocks of grid points are distributed across tasks in a two-dimensional Cartesian grid, and some of the blocks have grid points that are all on land. Tasks with these land blocks have no computation to perform for large segments of the run, leading to the load imbalance.

A different distribution of blocks clarifies this load imbalance. I implemented a distribution in POP that packs all the blocks with ocean points to the low-number tasks [6]. Figure 4 shows the resulting histogram report analogous to Figure 3. The task lists are much simpler, showing that tasks 0–302 had events of 4000–7000 ticks, while tasks 303–359 had events of mostly 5–7 ticks. The histogram report thus shows that only 303 tasks have ocean blocks, so only 303 tasks are really needed. Figure 5 shows the analogous histogram report using the packed distribution and just 304 tasks. (The run uses 304 tasks instead of 303 because it is easier to request even task counts on Jaguar, which currently has dual-core processors.) The histogram shows only one task "wasted".

5 Scalability

The previous examples show results of runs using 360 tasks or fewer, well within the range of automated tools like CrayPat. The question remains of how this timing library performs for larger runs. Figures 6 and

```
*** baroclinic_driver
...
event histogram for tasks: 0,2-4,13-22,25,27,31-44,47-53,55-62,65-71,74-80,84-89,91-96,
102-107,109-114,121-124,127-132,138-141,145-151,155-159,163-169,172-178,180,182-184,
193-202,205,207,211-224,227-233,235-242,245-251,254-260,264-269,271-276,282-287,289-294,
301-304,307-312,318-321,325-331,335-339,343-349,352-358
6000-6999 ticks: 001 max (task 000), 001 min (task 000), 001 avg
5000-5999 ticks: 127 max (task 000), 127 min (task 000), 127 avg

event histogram for tasks: 23-24,45-46,54,63-64,72-73,81-83,90,97-101,108,115-120,
125-126,133-137,142-144,152-154,160-162,170-171,179,181,185-192,206,208-210
6000-6999 ticks: 001 max (task 023), 001 min (task 023), 001 avg
5000-5999 ticks: 001 max (task 045), 000 min (task 023), 000 avg
4000-4999 ticks: 127 max (task 023), 126 min (task 045), 126 avg

event histogram for tasks: 1,6,11
6000-6999 ticks: 01 max (task 001), 01 min (task 001), 01 avg
7 tick(s): 75 max (task 001), 69 min (task 006), 73 avg
6 tick(s): 58 max (task 006), 52 min (task 001), 54 avg

event histogram for tasks: 5,7-10,12,26,28-30,203-204,225-226,234,243-244,252-253,
261-263,270,277-281,288,295-300,305-306,313-317,322-324,332-334,340-342,350-351,359
6000-6999 ticks: 001 max (task 005), 001 min (task 005), 001 avg
7 tick(s): 057 max (task 005), 000 min (task 204), 006 avg
6 tick(s): 127 max (task 262), 070 min (task 005), 120 avg
5 tick(s): 005 max (task 204), 000 min (task 005), 000 avg
```

Figure 3: Histogram report for timer "`barotropic_driver`" from the same run used for Figure 1.

```
*** baroclinic_driver
...
event histogram for tasks: 0-122,180-302
6000-6999 ticks: 001 max (task 000), 001 min (task 000), 001 avg
5000-5999 ticks: 127 max (task 000), 127 min (task 000), 127 avg

event histogram for tasks: 123-179
6000-6999 ticks: 001 max (task 123), 001 min (task 123), 001 avg
4000-4999 ticks: 127 max (task 123), 127 min (task 123), 127 avg

event histogram for tasks: 303-359
6000-6999 ticks: 001 max (task 303), 001 min (task 303), 001 avg
7 tick(s): 009 max (task 340), 000 min (task 303), 003 avg
6 tick(s): 127 max (task 309), 118 min (task 340), 123 avg
5 tick(s): 001 max (task 303), 000 min (task 304), 000 avg
```

Figure 4: Histogram report analogous to Figure 3, but from a run using a packed instead of a Cartesian data distribution.

```
*** baroclinic_driver
...
event histogram for tasks: 0-150,152-302
6000-6999 ticks: 001 max (task 000), 001 min (task 000), 001 avg
5000-5999 ticks: 127 max (task 000), 127 min (task 000), 127 avg

event histogram for task: 151
6000-6999 ticks: 001 max (task 151), 001 min (task 151), 001 avg
4000-4999 ticks: 127 max (task 151), 127 min (task 151), 127 avg

event histogram for task: 303
6000-6999 ticks: 001 max (task 303), 001 min (task 303), 001 avg
7 tick(s): 006 max (task 303), 006 min (task 303), 006 avg
6 tick(s): 121 max (task 303), 121 min (task 303), 121 avg
```

Figure 5: Histogram report analogous to Figure 4, but from a run using 304 instead of 360 tasks.

```
*** baroclinic_driver
...
event histogram for tasks: 12-16,56-66,85-92,131-143,150-151,160-168,206-232,235
-245,283-307,310-323,327-328,356-383,385-406,412-414,429-458,460-490,495,497-499
,502-1282,1284-1357,1359-1432,1435-1508,1510-1551,1554-1583,1586-1601,1603-1622,
1624-1625,1629-1658,1662-1676,1679-1696,1704-1733,1737-1751,1754-1771,1779-1808,
1814-1825,1830-1848,1853-1883,1889-1900,1905-1924,1926-1957,1964-1975,1981-2031,
2040-2050,2055-2106,2113-2125,2130-2181,2187-2199,2206-2256,2261-2275,2282-2332,
2335-2345,2348,2358-2420,2431-2494,2505-2518,2520-2569,2580-2581,2584-2588,2590-
2592,2594-2644,2655-2656,2658-2663,2665-2667,2670-2700,2702-2720,2729-2730,2732-
2736,2747-2774,2777-2796,2801-2805,2807-2808,2823-2849,2855-2872,2874-2880,2897-
2924,2931-2955,2958-2959,2972-2998,3007-3029,3032-3034,3047-3072,3082-3101,3103-
3109,3125-3147,3158-3172,3175,3178-3183,3201-3221,3233-3248,3257-3259,3277-3296,
3309,3311-3325,3332-3334,3351-3372,3381-3382,3386-3401,3426-3446,3456-3457,3460-
3476,3502-3520,3530-3548,3550-3552,3580-3594,3605-3613,3615-3623,3625-3627,3659-
3662,3664-3669,3679-3688,3691-3698,3700-3701,3732,3734-3737,3741-3742,3754-3764,
3767-3778,3805-3812,3829-3839,3843-3849,3851-3853,3877-3886,3905-3914,3918-3928,
3951-3963,3983-3989,3993-4004,4025-4040,4058-4064,4068-4078,4099-4115,4132-4139,
4142-4154,4176-4191,4207-4229,4249-4266,4282-4304,4319-4320,4322-4342,4357-4379,
4393-4418,4431-4432,4434-4455,4468-4493
500-599 ticks: 001 max (task 0012), 001 min (task 0012), 001 avg
300-399 ticks: 127 max (task 0012), 127 min (task 0012), 127 avg

event histogram for tasks: 2182,2184-2186,2200-2205
500-599 ticks: 001 max (task 2182), 001 min (task 2182), 001 avg
10-19 ticks: 126 max (task 2182), 126 min (task 2182), 126 avg
9 tick(s): 001 max (task 2182), 001 min (task 2182), 001 avg

event histogram for tasks: 2,4
500-599 ticks: 001 max (task 0002), 001 min (task 0002), 001 avg
8 tick(s): 110 max (task 0002), 076 min (task 0004), 093 avg
7 tick(s): 051 max (task 0004), 017 min (task 0002), 034 avg

event histogram for tasks: 0-1,3,5-6,8-11,17-20,23,33-55,67-81,93-94,108-130,144
-149,152-157,169,182-193,195-205,233-234,256-263,265-267,271-282,308-309,330-331
,334-337,341,345-355,384,409-411,421-428,459,500-501
500-599 ticks: 001 max (task 0000), 001 min (task 0000), 001 avg
8 tick(s): 061 max (task 0003), 000 min (task 0008), 001 avg
7 tick(s): 106 max (task 0033), 065 min (task 0009), 081 avg
6 tick(s): 062 max (task 0009), 000 min (task 0000), 044 avg

...
```

Figure 6: Histogram report analogous to Figure 3, but from a run using smaller grid blocks and 4500 tasks. Continues in Figure 7.

```
...
event histogram for tasks: 7,21-22,24-32,82-84,95-107,158-159,170-181,194,246-25
5,264,268-270,324-326,329,332-333,338-340,342-344,407-408,415-420,491-494,496,12
83,1358,1433-1434,1509,1552-1553,1584-1585,1602,1623,1626-1628,1659-1661,1677-16
78,1697-1703,1734-1736,1752-1753,1772-1778,1809-1813,1826-1829,1849-1852,1884-18
88,1901-1904,1925,1958-1963,1976-1980,2032-2039,2051-2054,2107-2112,2126-2129,21
83,2257-2260,2276-2281,2333-2334,2346-2347,2349-2357,2421-2430,2495-2504,2519,25
70-2579,2582-2583,2589,2593,2645-2654,2657,2664,2668-2669,2701,2721-2728,2731,27
37-2746,2775-2776,2797-2800,2806,2809-2822,2850-2854,2873,2881-2896,2925-2930,29
56-2957,2960-2971,2999-3006,3030-3031,3035-3046,3073-3081,3102,3110-3124,3148-31
57,3173-3174,3176-3177,3184-3200,3222-3232,3249-3256,3260-3276,3297-3308,3310,33
26-3331,3335-3350,3373-3380,3383-3385,3402-3425,3447-3455,3458-3459,3477-3501,35
21-3529,3549,3553-3579,3595-3604,3614,3624,3628-3658,3663,3670-3678,3689-3690,36
99,3702-3731,3733,3738-3740,3743-3753,3765-3766,3779-3804,3813-3828,3840-3842,38
50,3854-3876,3887-3904,3915-3917,3929-3950,3964-3982,3990-3992,4005-4024,4041-40
57,4065-4067,4079-4098,4116-4131,4140-4141,4155-4175,4192-4206,4230-4248,4267-42
81,4305-4318,4321,4343-4356,4380-4392,4419-4430,4433,4456-4467,4494-4499
500-599 ticks: 001 max (task 0007), 001 min (task 0007), 001 avg
7 tick(s): 063 max (task 0338), 000 min (task 0026), 003 avg
6 tick(s): 126 max (task 0251), 064 min (task 0338), 117 avg
5 tick(s): 014 max (task 3173), 000 min (task 0007), 006 avg
```

Figure 7: Continuation of Figure 6.

7 show the histogram report analogous to Figure 3, but for a run using smaller grid blocks and 4500 tasks, too many tasks for CrayPat at the time of this writing. The positive result is that moving from 360 to 4500 tasks only increases the number of histograms from four to five. Less positive is the length of the task lists for some of the histograms; the longest list covers 18 lines.

These long task lists are a result of the Cartesian distribution and the irregular land patterns on the Earth. The packed distribution eliminates the problem, as shown in Figure 8 for 4500 tasks and in Figure 9 for the minimal 3268 tasks. POP with the packed distribution now reports the number of ocean blocks directly, so inferring this from the timer output is not necessary. Nevertheless, this number is clear from the clean report in Figure 8, and Figure 9 further shows the performance improvement (from 300–399 ticks down to 100–299 ticks) from the added efficiency of using the optimal task count.

6 Conclusions

Motivated by the need to debug performance in parallel applications resulting from load imbalance and, potentially, performance variability, I have developed a timer library with multi-resolution histograms and reduced reporting. The timers themselves are local to each parallel task, with each timer identified globally by a unique character string. The library reports timer values in reduced form, using maximum, second-maximum, minimum, and average values, along with task and event numbers of the extreme measurements. The report combines histograms with the same maximum bin, the bin with the largest event count.

In tests with POP runs, the timing library showed a scalable reduction in output without hiding sources of load imbalance and performance variability. The maximum-bin histogram reduction maintains a small number of reported histograms, but the task list for a given histogram can grow with task count.

Potential improvements to the library include measurement and reporting of timer overhead, OpenMP support, and "nicer" report output.

Acknowledgments

Pat Worley of ORNL provided useful suggestions and background on other timing libraries.

This research was sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, US Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

This research used resources of the National Center for Computational Sciences at ORNL, which is supported by the Office of Science of the US Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] *Using Cray Performance Analysis Tools*. Cray, Inc., 2006. Document S-2376-31 from "`http://docs.cray.com/`".

[2] *Parallel Ocean Program (POP) User Guide*. Los Alamos National Laboratory, March 23, 2003. See "`http://climate.lanl.gov/`".

[3] *CCSM3.0 User's Guide*. M Vertenstein, T Craig, T Henderson, S Murphy, GR Carr Jr, and N Norton. National Center for Atmospheric Research, June 25, 2004. See "`http://www.ccsm.ucar.edu/`".

[4] *User's Guide to the NCAR Community Atmosphere Model (CAM 3.0)*. JR McCaa, M Rothstein, BE Eaton, JM Rosinski, E Kluzek, and M Vertenstein. National Center for Atmospheric Research, June 2004. See "`http://www.ccsm.ucar.edu/models/atm-cam/`".

[5] *MPI: A Message-Passing Interface Standard*. Message Passing Interface Forum. University of Tennessee, 1995. See "`http://www.mpi-forum.org/`".

[6] "Performance Analysis of Production POP Runs on the Cray XT3." J White III. First Annual Cray Technical Workshop, USA. Nashville, February 26-27, 2007. See "`http://nccs.gov/news/workshops/cray/`".

```
*** baroclinic_driver
...
event histogram for tasks: 0-3267
500-599 ticks: 001 max (task 0000), 001 min (task 0000), 001 avg
300-399 ticks: 127 max (task 0001), 126 min (task 0000), 126 avg
200-299 ticks: 001 max (task 0000), 000 min (task 0001), 000 avg

event histogram for tasks: 3268-4499
500-599 ticks: 001 max (task 3268), 001 min (task 3268), 001 avg
6 tick(s): 126 max (task 3503), 110 min (task 3436), 118 avg
5 tick(s): 017 max (task 3436), 001 min (task 3503), 008 avg
```

Figure 8: Histogram report analogous to Figure 4, but from a run using smaller grid blocks and 4500 tasks.

```
*** baroclinic_driver
...
event histogram for tasks: 122-128,154,169-197,210,238-254
200-299 ticks: 104 max (task 0127), 065 min (task 0197), 087 avg
100-199 ticks: 063 max (task 0197), 024 min (task 0127), 040 avg

event histogram for tasks: 0-121,129-153,155-168,198-209,211-237,255-3267
200-299 ticks: 064 max (task 0166), 007 min (task 3054), 020 avg
100-199 ticks: 121 max (task 3054), 064 min (task 0166), 107 avg
```

Figure 9: Histogram report analogous to Figure 5, but from a run using smaller grid blocks and 3268 tasks.