TR-I-0228

Tools for Monitoring Parallel Lisp Programs

Todd Kaufmann

1991 . 9

概要

Parallel architectures are becoming increasing popular today as a means of overcoming the speed limitations of convential serial computers. Since parallel machines are still fairly new, there does not yet exist a wide range of techniques, expertise, or tools for implementing parallel algorithms or portion serial algorithms to a parallel machine. The burden is on the programmer.

共有メモリー型密合並列計算機上でのプログラミングは、共有メモリーのロック等の独特の特徴があり、プログラムの最適化にあっては、通常の関数評価のプロファイリングでは、十分ではない。従って、マシン環境に合わせたプロファイラーの設計が必要となる。カーネギーメロン大学のシリアルマシン用プロファイラーを参考にして、Sequent/Symmetry上の並列Common Lisp (CLIP) 用に並列LISPプロファイラーを設計および作成した。本マニュアルでは、その特徴並びに使用法を述べる。特に並列プロセスの各種のオーバーヘッド並びに、ロックに係わる各種のオーバーヘッドの評価をサポートしたことにより、並列アルゴリズムの設計並びに、シリアルプログラムの並列化に有用である。

*ATR 自動翻訳電話研究所
ATR Interpreting Telephony Research Laboratories

Tools for Monitoring Parallel Lisp Programs

Todd Kaufmann

ATR Interpreting Telephony Research Laboratories
Sanpeidani, Inuidani, Seika-cho, Soraku-gun
Kyoto 619-02 Japan

E-mail: Todd_Kaufmann@cs.cmu.edu

Contents

Chapter 1

Introduction

Parallel architectures are becoming increasing popular today as a means of overcoming the speed limitations of convential serial computers. Since parallel machines are still fairly new, there does not yet exist a wide range of techniques, expertise, or tools for implementing parallel algorithms or porting serial algorithms to a parallel machine. The burden is on the programmer.

1.1 Need for monitoring tools

Once a set of processes are created and running in parallel, they may behave in a non-deterministic fashion or interact in unpredictable ways. Most of the processes may spend most of their time waiting to access a single object, resulting in very inefficient use of the processors, possibly worse than a serial implemention of the algorithm.

Described in this document are some tools that can help the programmer better understand how his program is running and utilizing resources in a parallel environment. These tools can help you understand

- how processes interact
- where execution time is spent

Although these tools can help you make your program more efficient, it won't help you choose the right algorithm.

1.2 How to load these files

On the Sequent Symmetry, they can be loaded in the following way:

```
<Initial lwp> (load "/usr1/todd/lisp/monitor")
; Fast loading /usr1/todd/lisp/monitor.fasl.
Warning: 'STUB-FUNCTION' is defined twice in /usr1/todd/lisp/monitor.lisp
Warning: 'STUB-FUNCTION' is defined twice in /usr1/todd/lisp/monitor.lisp

T
<Initial lwp> (load "/usr1/todd/lisp/lwp-monitor")
; Fast loading /usr1/todd/lisp/lwp-monitor.fasl.

T
```

The functions associated with `/usr1/todd/lisp/monitor` are detailed in Chapter 2, and the `lwp-monitor` functions are in Chapter 3. The two are completely independent, so you need only load the one that you use.

Chapter 2

Monitoring Functions

The functions described here are based on a package from CMU called `metering.lisp` which gathers timing statistics for specified functions while a program is running. They have been modified in order to work with CLiP and also provide information on a per-process basis, so you can see which processes are calling which functions how many times.

Sample usage of these functions and the specific features implemented for CLiP are shown here. For full documentation, you should see the file `monitor.lisp` which contains full comments.

2.1 Usage notes and suggestions

Start[1] by monitoring big pieces of the program, then carefully choose which functions close to, but not in, the inner loop are to be monitored next. Don't monitor functions that are called by other monitored functions: you will only confuse yourself.

If the per-call time reported is less than 0.1 seconds, then consider the clock resolution and profiling overhead before you believe the time. It may be that you will need to run your program many times in order to average out to a higher resolution.

CLiP note: If the numbers are outrageouly big (larger than possible) or negative, this means that the function began on one physical processor and then finished on another; it is even possible that it switched multiple times during execution, so even if it ends and begins on the same processor some portion of the recorded run time will not be shown, or run time of another function may be included. This will almost always happen if your function blocks (via sleepy-lock or `suspend-lwp`) waiting for another process and/or the number of lwps is greater that the number of physical processors allocated via pp:`set-numprocs`. Your only recourse is to try again until you are lucky, or (if possible) run it serially on a single processor.

The easiest way to use this package is to execute either

[1]Many of these comments are from the file `metering.lisp`, lightly edited.

```
    (mon:with-monitoring (names*) ()
       your-forms*)
```

or

```
    (mon:monitor-form your-form)
```

The former allows you to specify which functions will be monitored; the latter monitors all functions in the current package. Both automatically produce a table of statistics. Other variants can be constructed from the monitoring primitives, which are described below, along with a fuller description of these two macros.

2.1.1 Monitoring overhead

The added monitoring code takes time to run every time that the monitored function is called, which can disrupt the attempt to collect timing information. In order to avoid serious inflation of the times for functions that take little time to run, an estimate of the overhead due to monitoring is subtracted from the times reported for each function.

Although this correction works fairly well, it is not totally accurate, resulting in times that become increasingly meaningless for functions with short runtimes. For example, subtracting the estimated overhead may result in negative times for some functions. This is only a concern when the estimated profiling overhead is many times larger than reported total CPU time.

If you monitor functions that are called by monitored functions, in `:inclusive` mode the monitoring overhead for the inner function is subtracted from the CPU time for the outer function. [We do this by counting for each function not only the number of calls to *this* function, but also the number of monitored calls while it was running.] In `:exclusive` mode this is not necessary, since we subtract the monitoring time of inner functions, overhead and all.

Otherwise, the estimated monitoring overhead is not represented in the reported total CPU time. The sum of total CPU time and the estimated monitoring overhead should be close to the total CPU time for the entire monitoring run (as determined by `time`).

A timing overhead factor is computed at load time. This will be incorrect if the monitoring code is run in a different environment than this file was loaded in. For example, saving a core image on a high performance machine and running it on a low performance one will result in the use of an erroneously small overhead factor.

If your times vary widely, possible causes are:

- Garbage collection. Try turning it off, then running your code. Be warned that monitoring code will probably cons when it does `(get-internal-run-time)`, as this value is a bignum.
- Swapping. If you have enough memory, execute your form once before monitoring so that it will be swapped into memory. Otherwise, get a bigger machine!

- Resolution of internal-time-units-per-second. If this value is too low, then the timings become wild. You can try executing more of whatever your test is, but that will only work if some of your paths do not match the timer resolution.
- The above mentioned CLiP problem.

2.2 Function description

Note that it is not possible to measure consing with Allegro. If you use the generic (non-parallel) form of this package with Lucid or other Common Lisps, then it is possible to.

```
with-monitoring (&rest functions)
                (&optional (nested :exclusive)
                           (threshold 0.01)
                           (key :percent-time))
                &body body
```

The named functions will be set up for monitoring, the body forms executed, a table of results printed, and the functions unmonitored. The nested, threshold, and key arguments are passed to report-monitoring below.

```
monitor-form form
             &optional (nested :exclusive)
                       (threshold 0.01)
                       (key :percent-time)
```

All functions in the current package are set up for monitoring while the form is executed, and automatically unmonitored after a table of results has been printed. The nested, threshold, and key arguments are passed to report-monitoring below.

`*monitored-functions*` [*Variable*]
This holds a list of all functions that are currently being monitored.

`monitor &rest` *names* [*Macro*]
The named functions will be set up for monitoring by augmenting their function definitions with code that gathers statistical information about code performance. As with the `trace` macro, the function names are not evaluated. Calls the function `mon::monitoring-encapsulate` on each function name. If no names are specified, returns a list of all monitored functions.

If name is not a symbol, it is evaluated to return the appropriate closure. This allows you to monitor closures stored anywhere like in a variable, array or structure. Most other monitoring packages can't handle this.

`monitor-all &optional (`*package*` *package*)` [*Function*]

Monitors all functions in the specified package, which defaults to the current package.

`unmonitor &rest` *names* [*Macro*]

Removes monitoring code from the named functions. If no names are specified, all currently monitored functions are unmonitored.

`reset-monitoring-info` *name* [*Function*]

Resets the monitoring statistics for the specified function.

`reset-all-monitoring` [*Function*]

Resets the monitoring statistics for all monitored functions.

`monitored` *name* [*Function*]

Predicate to test whether a function is monitored.

```
report-monitoring &optional names
                            (nested :exclusive)
                            (threshold 0.01)
                            (key :percent-time)
```

Creates a table of monitoring information for the specified list of names, and displays the table using `display-monitoring-results`. If names is `:all` or `nil`, uses all currently monitored functions. Takes the following arguments:

nested specifies whether nested calls of monitored functions are included in the times for monitored functions.

- If `:inclusive`, the per-function information is for the entire duration of the monitored function, including any calls to other monitored functions. If functions A and B are monitored, and A calls B, then the accumulated time and consing for A will include the time and consing of B. Note: if a function calls itself recursively, the time spent in the inner call(s) may be counted several times.
- If `:exclusive`, the information excludes time attributed to calls to other monitored functions. This is the default.

threshold specifies that only functions which have been executed more than threshold percent of the time will be reported. Defaults to 1%. If a threshold of 0 is specified, all functions are listed, even those with 0 or negative running times (see note on overhead).

key specifies that the table be sorted by one of the following sort keys:

`:function`	alphabetically by function name
`:percent-time`	by percent of total execution time
`:percent-cons`	by percent of total consing
`:calls`	by number of times the function was called
`:time-per-call`	by average execution time per function
`:cons-per-call`	by average consing per function
`:time`	same as `:percent-time`
`:cons`	same as `:percent-cons`

```
display-monitoring-results &optional (threshold 0.01)
                                     (key :percent-time)
```

Prints a table showing for each named function:

- the total CPU time used in that function for all calls
- the total number of bytes consed in that function for all calls
- the total number of calls
- the average amount of CPU time per call
- the average amount of consing per call
- the percent of total execution time spent executing that function
- the percent of total consing spent consing in that function
- Summary totals of the CPU time, consing, and calls columns are printed. An estimate of the monitoring overhead is also printed. May be run even after unmonitoring all the functions, to play with the data.

  Sample table:

```
                                                   Cons
                    %     %                        Per        Total    Total
Function            Time  Cons  Calls  Sec/Call    Call       Time     Cons
-------------------------------------------------------------------------------
FIND-ROLE:          0.58  0.00    136  0.003521       0  0.478863        0
GROUP-ROLE:         0.35  0.00    365  0.000802       0  0.292760        0
GROUP-PROJECTOR:    0.05  0.00    102  0.000408       0  0.041648        0
FEATURE-P:          0.02  0.00    570  0.000028       0  0.015680        0
-------------------------------------------------------------------------------
TOTAL:                           1173                    0.828950        0
Estimated total monitoring overhead: 0.88 seconds
```

per-process-report *name* [*Function*]

Shows a list of number of calls and amount of time on a per-process basis for function name (Allegro CLiP only).

```
For function MY-FUNCTION:
 # calls      time(secs)      name of process
      2          0.85         process-5
      9          1.13         process-0
      5          0.98         process-2
      2          0.78         process-3
      4          0.72         process-1
```

Again, remember to not believe very strange times. It is possible that this will only happen for some of the processes, in which case you can probably believe the other ones.

Chapter 3

Monitoring Processes and Locks

The functions associated with lwp's and locks have been modified to record their usage and times. The following functions are modified:

lock functions: `acquire-sleepy-lock`, `release-sleepy-lock`, `with-spin-lock`[1].

lwp functions: `resume-lwp`, `suspend-lwp`, `kill-lwp`, `start-lwps`, and `make-lwp`[2].

The functions are modified by loading the file `monitor.lisp`. These modifications will remain in effect until you call the function `lwp-mon-off` which will revert to the original definitions of these functions. It can also be turned back on with `lwp-mon-on`.

3.1 Lock usage recording

The following information is recorded for each usage of each lock:

- The time the process first tried to acquire the lock
- The time the process actually acquired the lock
- The real and run time spent blocked (the period while waiting for a lock)
- The time the process released the lock
- The real and run time spent while holding the lock
- The name of the process holding the lock
- The maximum number of processes ever waiting for the lock
- How many times the lock has been used.

[1]`with-spin-lock` is a macro, so it is not possible to modify it without requiring all user code to be recompiled. Instead, the undocumented primitives `excl::release-spin-lock` and `excl::acquire-spin-lock` have been modified to achieve this behavior.

[2]`make-lwp` also is a macro; the actual modified function is `pp::make-lwp*` which is called by this.

3.2 Output functions

3.2.1 Lock usage summary

The data recorded is summarized by the function print-lock-usage. Here is a test function which will be used in the examples, and the resulting sample output.

```
<Initial lwp> (defvar *my-lock* (make-spin-lock))
*MY-LOCK*
<Initial lwp> (set-lock-name-for-printing *my-lock* "My lock")
:ADDED
<Initial lwp> (defun lock-test (n)
                (let ((count 0))
                  (make-lwp (dotimes (i n)
                              (with-spin-lock *my-lock* (incf count))) :name "lwp-1")
                  (make-lwp (dotimes (i n)
                              (with-spin-lock *my-lock* (incf count))) :name "lwp-2")
                  (start-lwps)
                  count))
LOCK-TEST
<Initial lwp> (lock-test 60)
Using 6 processors

120
<Initial lwp> (print-lock-usage)
usage  max #    time blocked     time held      max blocked      max held
count  waiting   run   real      run  real      run    real      run real       lock name
-----------------------------------------------------------------------------------------------
  4      4       0.02   0.01     0.00   0.00     0.02   0.01     0.00   0.00     Spin-0
                                                                              *** 4 waiting
 60      2       0.10   0.13     1.15   0.15     0.02   0.01     0.05   0.01     My lock
-----------------------------------------------------------------------------------------------
[ *** indicates that lock is still being waited for.]
```

(The lock named Spin-0 is a lock used by the CLiP system for scheduling.)

The descriptions of the fields are:

usage count the number of times attempted to acquire the lock. If some attempts are unsuccessful, then there will still be processes waiting for the lock when the all processes terminate. This is possible if the processes terminate abnormally (due to error or interrupt) or are killed by another process.

max # waiting the maximum number of processes waiting for a lock at any given time.

time blocked The total amount of time spent by all processes as they waited for the lock to become available.

time held The total amount of time for all processes between the time they acquired the lock and the time releasing it.

max blocked The longest time that any single process spent waiting for this lock.

max held The longest time any process held the lock.

lock name name of the lock.

Real time is the amount of clock time that has elapsed. Run time is the amount of CPU time used. Note that run time is calculated by getting the value of `(get-internal-run-time)` at an initial time, and then subtracting this from the value at a later time. One of the "features" of the Sequent Symmetry parallel processing operating system is that program may switch to other physical processors. A side effect of this is that the value of different calls to `(get-internal-run-time)` may be values from different processors, which will have no relation to each other. This can cause very large or negative run times to appear. When you see such numbers it is best to compare them with the real times to see if they make sense.

To clear and reset the lock usage statistics, use the following function:

`reset-lock-table (&optional` *clear-zeros* *[Function]*

Zero all entries for the locks in `*lock-held-times-hashtable*`. If optional argument *clear-zeros* is supplied, remove any entries for locks which have not been used (usage count = 0) as well.
`(progn (reset-lock-table) (reset-lock-table t))`

will have the result of emptying the table entirely.

3.2.2 Assigning names to locks

Locks have no name slot, appearing simply as `#<unlocked spin-lock @e1d1e9>` which makes them hard to distinguish apart when there is large number of them on your screen. The following are supplied to allow you to associate a name with the locks you have created, which the functions here will use for more readable output.

`set-lock-name-for-printing (`*lock name*`)` *[Function]*

Assign *name* to be used when *lock* is print by the functions `print-l-history` and `print-lock-usage`. Both functions call `pretty-print-lock` to get this name.

`*lock-name-mapping*` *[Global Variable]*

A list of (`<lockobj> . <print-name-for-lock>`) to make output from print-lock-usage more readable.

`maybe-generate-lock-name` (*lock*) [*Function*]

Get the pretty name of *lock*, generating one if necessary and adding to `*lock-name-mapping*` list. This is normally called by `print-l-history`.

3.2.3 Lock and process history timeline

The output functions can also display a timeline showing

- when lwp's are started, suspended, and resumed, and by which process
- when sleepy and spin locks are acquired and released by which process, and also when the process is blocked waiting for the lock.

Here is a simple test function and sample output:

```
<Initial lwp> (lock-test 3)
Using 3 processors

6
<Initial lwp> (print-l-history)
17:58:50.010:
17:58:50.015:
...
17:58:50.040:  My lock is BLOCK by lwp-1;  My lock is ACQUIRE by lwp-1;
               My lock is BLOCK by lwp-2;  My lock is RELEASE by lwp-1;
17:58:50.045:
17:58:50.050:  My lock is BLOCK by lwp-1;  My lock is ACQUIRE by lwp-1;
               My lock is RELEASE by lwp-2;  My lock is RELEASE by lwp-1;
               My lock is BLOCK by lwp-2;  My lock is ACQUIRE by lwp-2;
               My lock is BLOCK by lwp-1;  My lock is RELEASE by lwp-2;
17:58:50.055:
17:58:50.060:  Spin-0 is BLOCK by Scheduler0;  Spin-0 is ACQUIRE by Scheduler0;
               My lock is RELEASE by lwp-1;  My lock is ACQUIRE by lwp-2;
               My lock is RELEASE by lwp-2;  lwp-2 is KILL'd by Scheduler1;
               lwp-1 is KILL'd by Scheduler0;
17:58:50.065:
17:58:50.070:  Spin-0 is BLOCK by Scheduler1;  Spin-0 is ACQUIRE by Scheduler1;
Legend of Lock names:
Spin-0                          is #<unlocked spin-lock @5c9a11>
My lock                         is #<unlocked spin-lock @d05201>
```

The format of each message is *<object>* `is` *<action>* `by` *<lwp>*, where object can be a lock or lwp. The following actions may appear:

for locks: block, acquire, and release;

for lwps: start-lwp, resume, suspend, create, and kill.

You may not see all these types of items in your output if your program doesn't use these features. Occasionally there may be write-write conflicts during usage recording if several processing try to acquire a lock at the same time (trying to use a lock to overcome this problem would not help things). This is a minor issue since eventually the process will acquire the lock, and acquiring should not have this problem since only one process will acquire a lock at any given time, whereas several may block on the same lock at the same time. If the acquires and release happen fast enough to incur write-write conflicts, this indicates that it is not a bottleneck in your program anyway.

`print-l-history &key` *lwp lock*
 (*dont-show-lwps* `*dont-show-lwps*`)
 (*dont-show-locks* `*dont-show-locks*`)
 (*time-scale* 2)
 (*elision-threshold* `*normal-elision-threshold*`)
 (*initial-elision-threshold* `*initial-elision-threshold*`)
 time-zero

This is the main output function for process and lock history timelines. Without any arguments, an attempt is made to display them all in a readable fashion. The arguments allow focussing or filtering certain locks and processes, and changing the scale of the timeline.

lwp and *lock* allow focusing on a specific lwp or lock. *lock* must be a lock; *lwp* may be a lwp or the name of one. To specify no lwps or locks in the output, use the value `:none`, or use the functions `print-lwp-history` or `print-lock-history` which have the same effect as specifying `:none` for the type your don't wish to see.

dont-show-lwps — lwps which match an item of this list (either being a member or matching the name as a string) will not be shown in the output. For example, specifying the string `"Sched"` will filter out all scheduler processes from the timeline.

dont-show-locks Locks which match this (either eq to it or a member of the list) will not be shown in the output.

time-scale specifies by what factor the increment from one line listing the time to the next should be divided by. The following heuristic is used in calculating the increment: take the number of non-zero time differences between subsequent events (events occurring at the same time are counted as a single event) and divide by the number. This is then divided by *time-scale* and used as the increment. This seems to produce fairly good results most of the time. By increasing *time-scale* you spread out the events farther, but the spaces between them will be bigger. By decreasing *time-scale* you cause more events to be bunched together at the same time.

time-zero can be used for setting the time from which `print-l-history` starts; by default it is the first time a lock usage is recorded or the time when `start-lwps` is called, whichever is earlier.

elision-threshold To decrease the amount of empty lines with no events listed, after *elision-threshold* number of empty lines a "`...`" is printed to show that there are lines not shown. This produces a shorter listing, but it can be misleading because it looks like there is very little time between subsequent events. This can be made a large number to show the space between times more accurately, but the tradeoff is that it is harder to find the events among the blank lines.

initial-elision-threshold can be used for setting the initial number before eliding; after the first event, *elision-threshold* will be used.

`print-lwp-history` [*Function*]

Print a timeline showing times and events that occurred among lwp's. This is simply a call to `print-l-history`, with lock information suppressed via the `:lock :none` argument.

`print-lock-history` [*Function*]

Print a timeline showing times and events that occurred concerning sleepy and spin locks. This is simply a call to `print-l-history`, with lwp information suppressed via the `:lwp :none` argument.

`lwp-mon-on` [*Function*]

Enable lwp and lock monitoring.

`lwp-mon-off` [*Function*]

Revert to old lwp and lock function behavior (no monitoring).

3.2.4 Miscellaneous functions

These following functions are used internally by the lwp and lock recording mechanisms. You may want to record other events to trace your program.

`clear-lwp-history (&optional` *lwp*`)` [*Function*]

Removes all events associated with *lwp*. If *lwp* is `nil` or not specified, then all *lwp*'s events are removed. This normally happens when `start-lwps` is first called. This can be suppressed by setting the following global.

`*initial-start-lwp*` [*Global Variable*]

Bound in the redefinition of `pp:start-lwps` to `nil`, so that history isn't cleared by further calls within to `pp:start-lwps`. You could also set this to `nil` if you want to make several runs without clearing out the accumulated info each time.

`record-lwp-history (`*lwp type control*`)` [*Function*]

Record for *lwp* the *type* of event that occurred, and the associated lwp *control* which was responsible. This creates messages such as `proc-1 is KILL'd by proc-2`, where *lwp* would be the lwp `proc-1`, *type* would be `:kill`, and *control* would be the lwp `proc-2`. This event is recorded for the time at which this function is called.

`reset-lock-times (`*lock*`)` [*Function*]

Zeroes all the usage information associated with *lock* as printed by `print-lock-usage`. `reset-lock-times` is an interface to this function.

`clear-lock-history (&optional` *lock*`)` [*Function*]

Clears the events associated with *lock*. If *lock* is not specified, removes all lock events. This normally happens when `start-lwps` is first called, and can also be suppressed by `*initial-start-lwp*`.

`record-lock-history (`*lock type lwp*`)` [*Function*]

Record for *lock* the *type* of event that occurred, and the associated *lwp*. This is the analog of `record-lwp-history` for locks.

`print-internal-time-readably (`*itime* `&optional (`*stream* `t))` [*Function*]

Prints a humanly readable form of internal time, which has resolution of the order of 10 milliseconds.