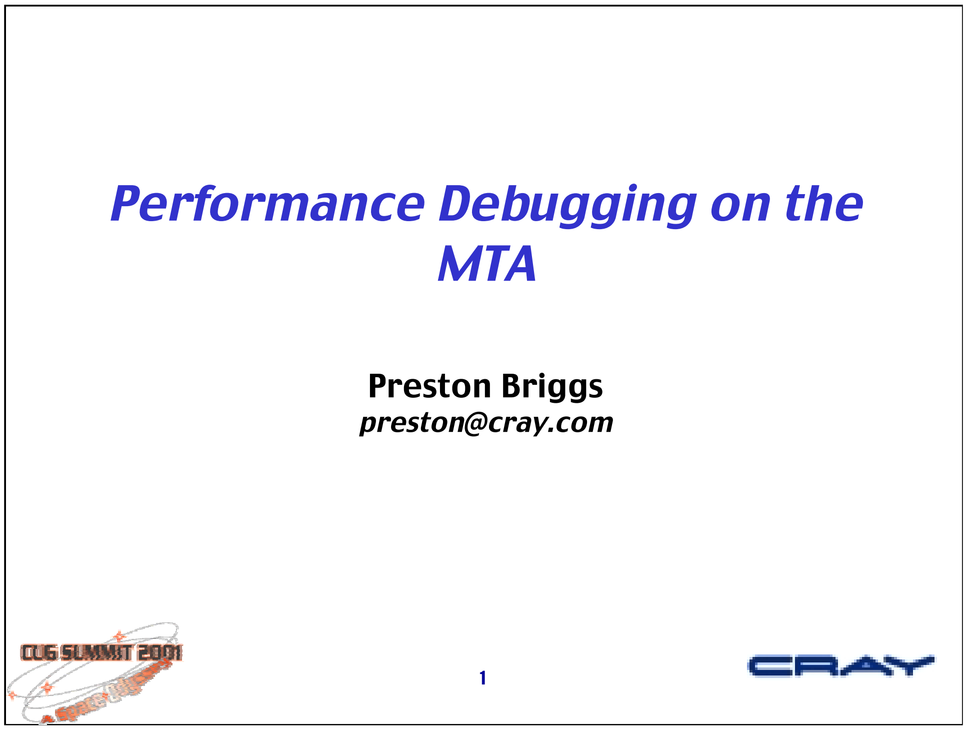
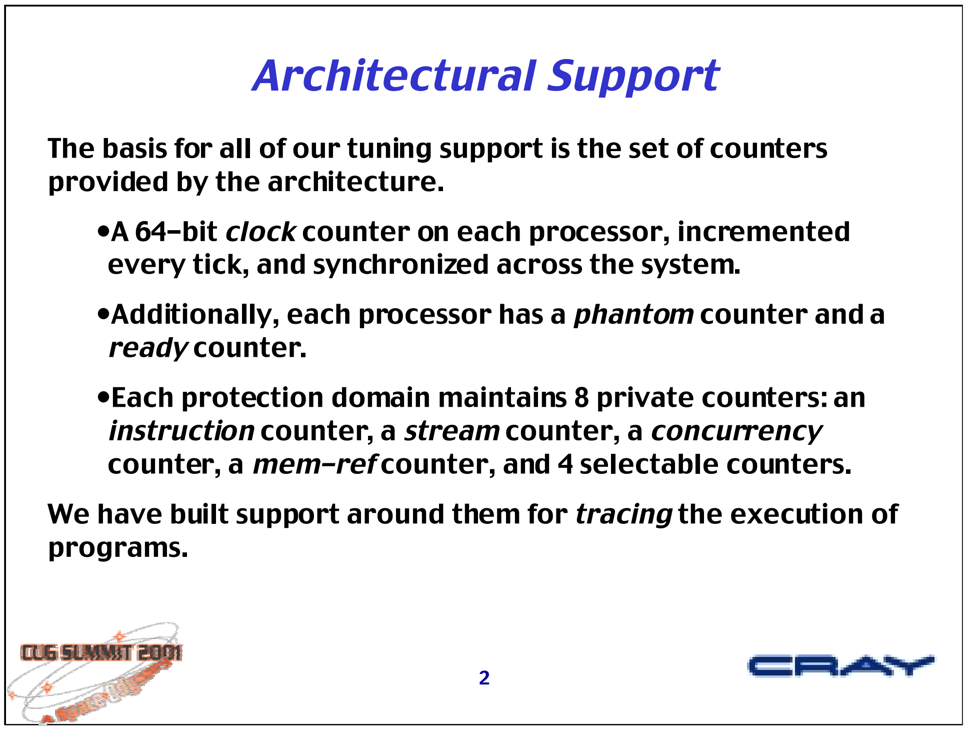


Performance Debugging on the MTA

Preston Briggs
preston@cray.com

This talk introduces a couple of tools for performance debugging. They work well together in practice and are the tools of choice for our benchmarking and applications efforts. I'll introduce each tool and show examples illustrating typical use.



Architectural Support

The basis for all of our tuning support is the set of counters provided by the architecture.

- A 64-bit *clock* counter on each processor, incremented every tick, and synchronized across the system.
- Additionally, each processor has a *phantom* counter and a *ready* counter.
- Each protection domain maintains 8 private counters: an *instruction* counter, a *stream* counter, a *concurrency* counter, a *mem-ref* counter, and 4 selectable counters.

We have built support around them for *tracing* the execution of programs.

The 4 selectable counters may be used to count events like nops, loads, stores, traps, flops, *etc*.

Each counter may be accessed from a user-level program in a single operation.

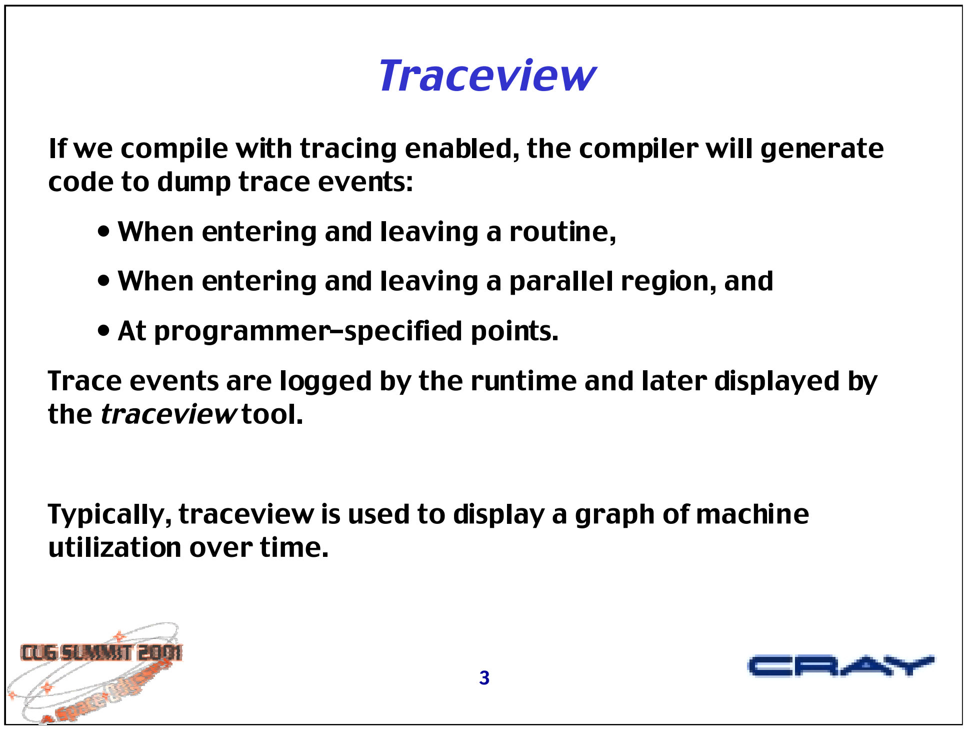
The *phantom* counter accumulates pipeline bubbles, when no instruction is ready to issue.

The *ready* counter sums the total number of streams ready at each tick.

The *instruction* counter is incremented each time an instruction issues in the domain (a protection domain is basically a single job).

The *stream* counter provides an indication of the average stream usage of a domain.

The *concurrency* counter provides an indication of the average number of memory operations in progress.



Traceview

If we compile with tracing enabled, the compiler will generate code to dump trace events:

- When entering and leaving a routine,
- When entering and leaving a parallel region, and
- At programmer-specified points.

Trace events are logged by the runtime and later displayed by the *traceview* tool.

Typically, traceview is used to display a graph of machine utilization over time.

To compile with tracing, use the `-trace` command-line flag.

Tracing can be enabled and disabled at any point in the program, using command-line flags or directives.

When the program is run, the trace events are dumped into a file called `trace.out`.

To examine the results, invoke traceview like this

```
traceview trace.out a.out
```

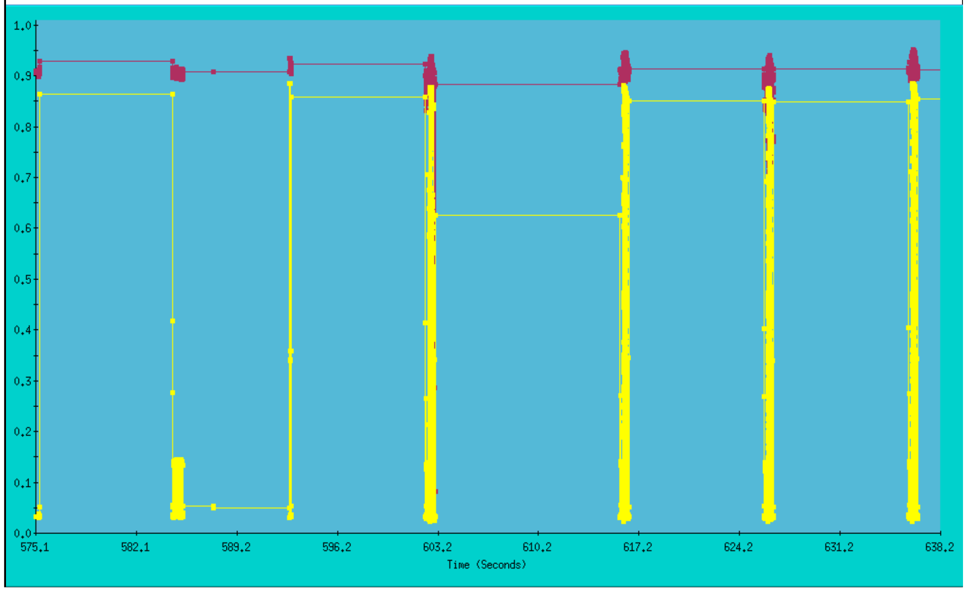
(substituting the name of your executable for `a.out`)

Usually, I'll limit tracing to a subset of the routines to help keep overheads down. Start tracing at the top-most routines in the call tree, exploring more deeply as you go. Typically, trace overheads are on the order of 1%.

Sometimes it's useful to add programmer-specified trace points, *e.g.,*

```
#pragma mta trace "end of loop"
```

An Example Trace



In this graph, the horizontal axis represent time, in seconds, and the vertical axis represents a rate (in this case, instructions per clock tick).

Traceview plots two lines:

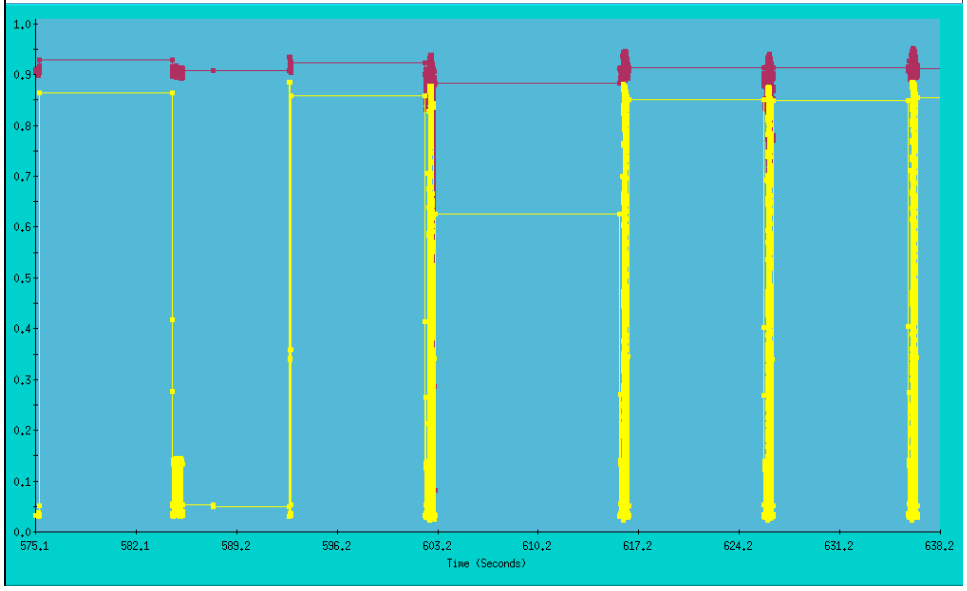
- The top one shows the number of available issue slots per tick, and
- The bottom one indicates the number of instructions issued per tick.

The difference between the two indicates *phantoms,* or wasted opportunities (our goal is to keep the machine busy by reducing phantoms).

Clicking on a particular location gives more detail, including the location and access to the source code via a canal window.

It’s also possible to plot the number of memory references and floating-point operations per tick.

Interpretation



This is a small snapshot from the middle of a longer run. By zooming in, we can examine particular areas more closely, if desired.

We're running on a single processor (since the availability peaks at almost 1 instruction/tick).

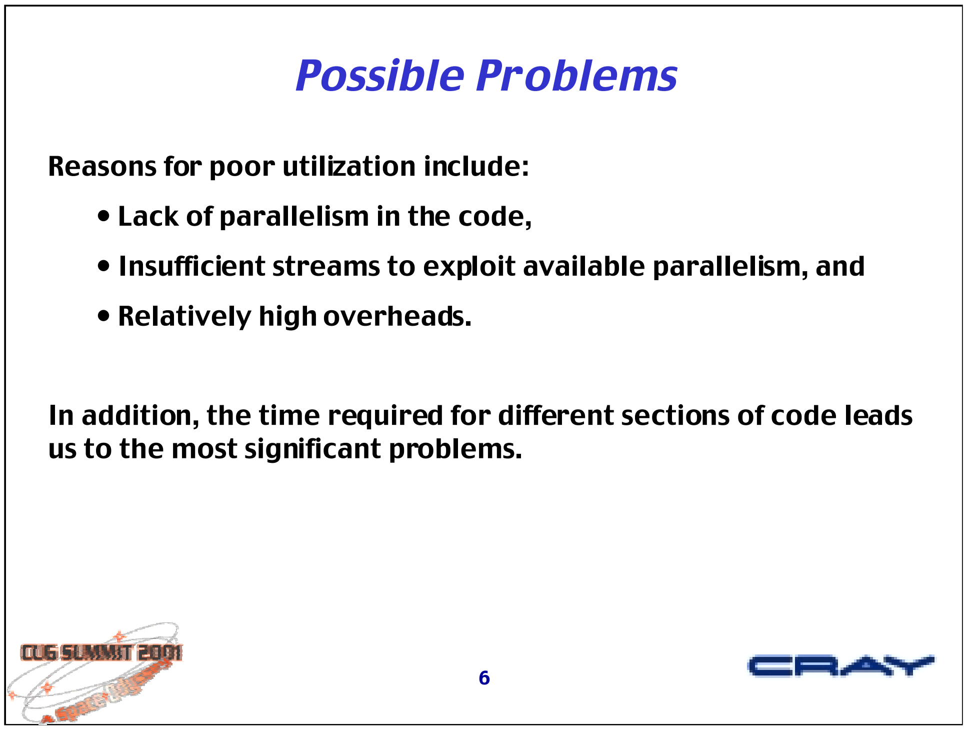
No other program is consuming much of the processor (since the availability stays high throughout).

We see a sequence of 5 big parallel loops (1 with relatively poor utilization), a low spot indicating a serial region, and a number of relatively unimportant noisy areas, indicating a series of short parallel loops (probably! We'd need to zoom in for a better look).

The first thing to examine here would be the serial region.

Next *might* be the parallel region of low utilization (or maybe we're done).

The rest looks good.



Possible Problems

Reasons for poor utilization include:

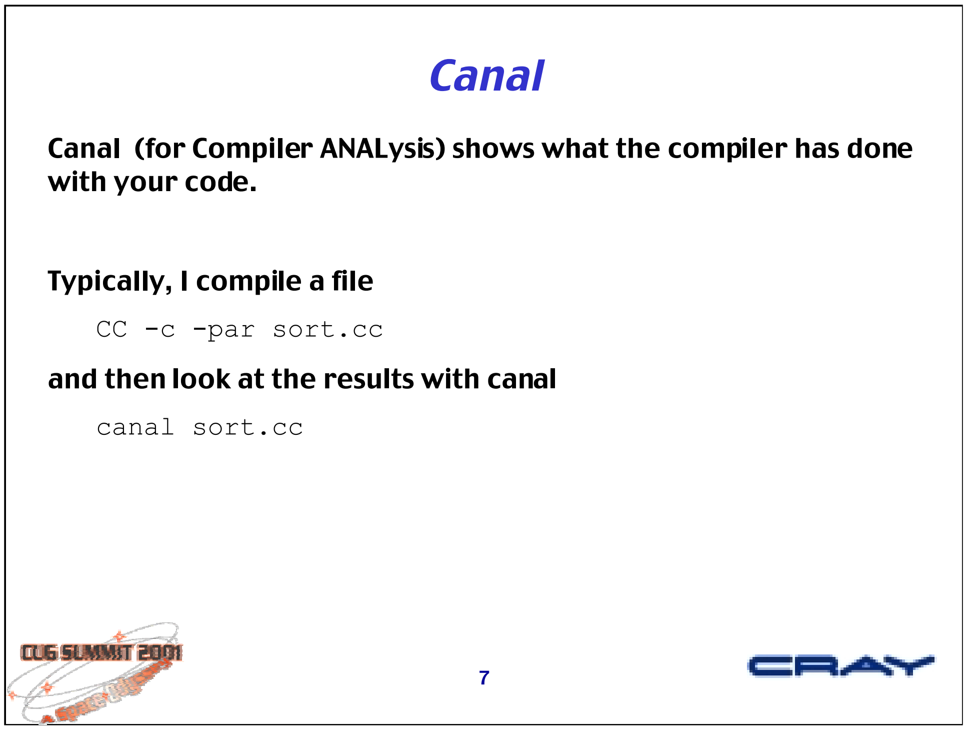
- Lack of parallelism in the code,
- Insufficient streams to exploit available parallelism, and
- Relatively high overheads.

In addition, the time required for different sections of code leads us to the most significant problems.

When you look at a traceview, focus on the big problems first.

Low utilization (compared to availability) means we need to get more streams running, somehow.

Lots of noise *may* indicate high overheads (as we quickly enter and exit a number of parallel regions). Zoom in for a closer look. Try rewriting the code to move the parallelism out (to a higher level).



Canal

Canal (for Compiler ANALysis) shows what the compiler has done with your code.

Typically, I compile a file

```
CC -c -par sort.cc
```

and then look at the results with canal

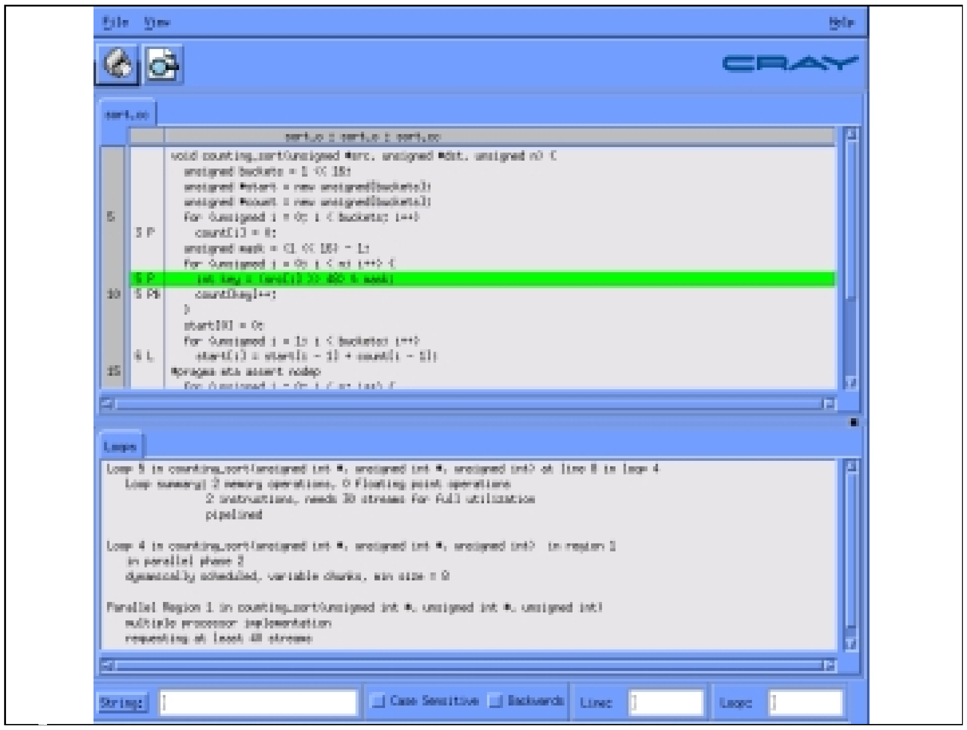
```
canal sort.cc
```

Canal shows us several things

- instruction counts for inner loops
- loops that have been parallelized
- functions that have been inlined
- places where the compiler has added synchronization
- statements that prevent automatic parallelization
- the extent of parallel regions

In addition, canal will report on many of the loop transformations used by the compiler.

Notice that I’ve used the `-par` flag on the command line. I use it here because parallelization is disabled by default for the C and C++ compilers.



```
              sort.o : sort.o : sort.cc
        void counting_sort(unsigned *src, unsigned *dst, unsigned n) {
          unsigned buckets = 1 << 16;
          unsigned *start = new unsigned[buckets];
          unsigned *count = new unsigned[buckets];
 5        for (unsigned i = 0; i < buckets; i++)
   3 P      count[i] = 0;
          unsigned mask = (1 << 16) - 1;
          for (unsigned i = 0; i < n; i++) {
   5 P      int key = (src[i] >> 16) & mask;
10 5 PN     count[key]++;
          }
          start[0] = 0;
          for (unsigned i = 1; i < buckets; i++)
   6 L      start[i] = start[i - 1] + count[i - 1];
15      #pragma mta assert nodep
          for (unsigned i = 0; i < n; i++) {
```

```
Loop 5 in counting_sort(unsigned int *, unsigned int *, unsigned int) at line 8 in loop 4
   Loop summary: 2 memory operations, 0 floating point operations
                 2 instructions, needs 30 streams for full utilization
                 pipelined

Loop 4 in counting_sort(unsigned int *, unsigned int *, unsigned int)  in region 1
   in parallel phase 2
   dynamically scheduled, variable chunks, min size = 8

Parallel Region 1 in counting_sort(unsigned int *, unsigned int *, unsigned int)
   multiple processor implementation
   requesting at least 40 streams
```

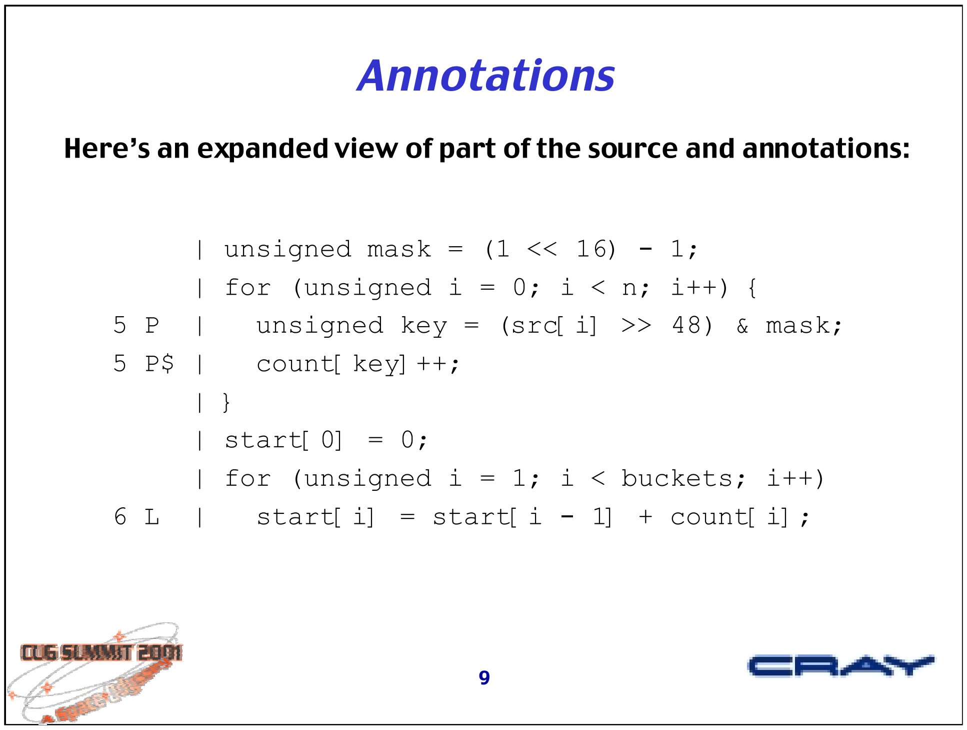
Here's what the GUI interface to canal looks like (it's also possible to produce a simple ASCII representation that's somewhat less useful).

The top half shows source code with annotations. From here, you can scoll around the file, plus examine the source for inlined functions.

The bottom half shows notes about the selected loop (where the selection is indicated by the highlight on line 9 in the code).

At the very bottom is a panel for searching.

At the very top is a limited selection of menus to let you examine the overall structure of the code, print, *etc.*



Annotations

Here's an expanded view of part of the source and annotations:

```
        | unsigned mask = (1 << 16) - 1;
        | for (unsigned i = 0; i < n; i++) {
5 P     |   unsigned key = (src[ i] >> 48) & mask;
5 P$    |   count[ key] ++;
        | }
        | start[ 0] = 0;
        | for (unsigned i = 1; i < buckets; i++)
6 L     |   start[ i] = start[ i - 1] + count[ i] ;
```

The number on the left refers to the number of the loop containing the statement. Loop numbers are assigned in a fairly arbitrary fashion by the compiler.

Note that the loop headers themselves aren't numbered, only the statements inside the loops.

The next few letters encode the annotation information for the loops containing the statement. For example, the sequence

```
5 P
```

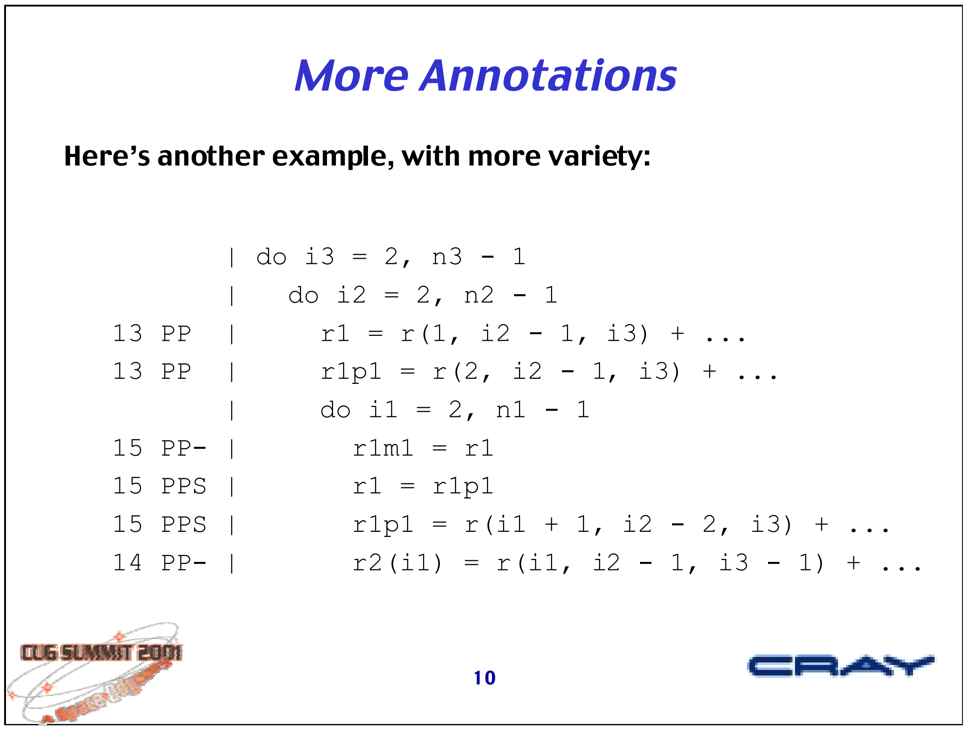
means that the containing loop (loop 5) has been parallelized. On the other hand, the sequence

```
6 L
```

means that the containing loop is a linear recurrence that has been rewritten to be solved in parallel.

Clicking on an annotated statement will cause canal to display notes about the loops enclosing the statement (plus remarks, if any) in the lower half of the window.

There are many possible annotations, all described in programming guide.



More Annotations

Here’s another example, with more variety:

```
        | do i3 = 2, n3 - 1
        |   do i2 = 2, n2 - 1
13 PP   |     r1 = r(1, i2 - 1, i3) + ...
13 PP   |     r1p1 = r(2, i2 - 1, i3) + ...
        |     do i1 = 2, n1 - 1
15 PP-  |       r1m1 = r1
15 PPS  |       r1 = r1p1
15 PPS  |       r1p1 = r(i1 + 1, i2 - 2, i3) + ...
14 PP-  |       r2(i1) = r(i1, i2 - 1, i3 - 1) + ...
```

Here, the annotation

```
13 PP
```

means that the two loops containing this statement have been parallelized.

Next, the annotation

```
15 PP-
```

means that while the outermost pair of containing loop have been parallelized, the innermost loop has been left serial (for reasons not having to do with this statement). On the other hand, the annotation

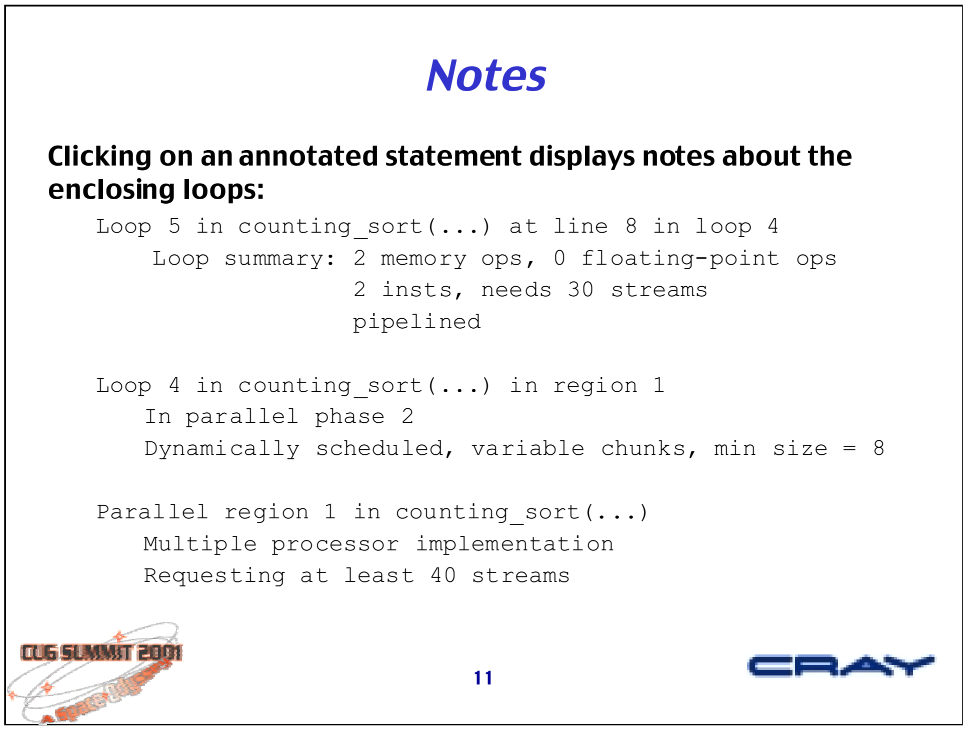
```
15 PPS
```

points out that this statement is at least partially responsible for serializing the inner loop.

Finally notice the statement marked

```
14 PP-
```

The fact that this statement has a different loop number than the previous statement (even though they are nested identically) points out that the compiler has performed some *loop distribution*, probably to reduce register demand.



Notes

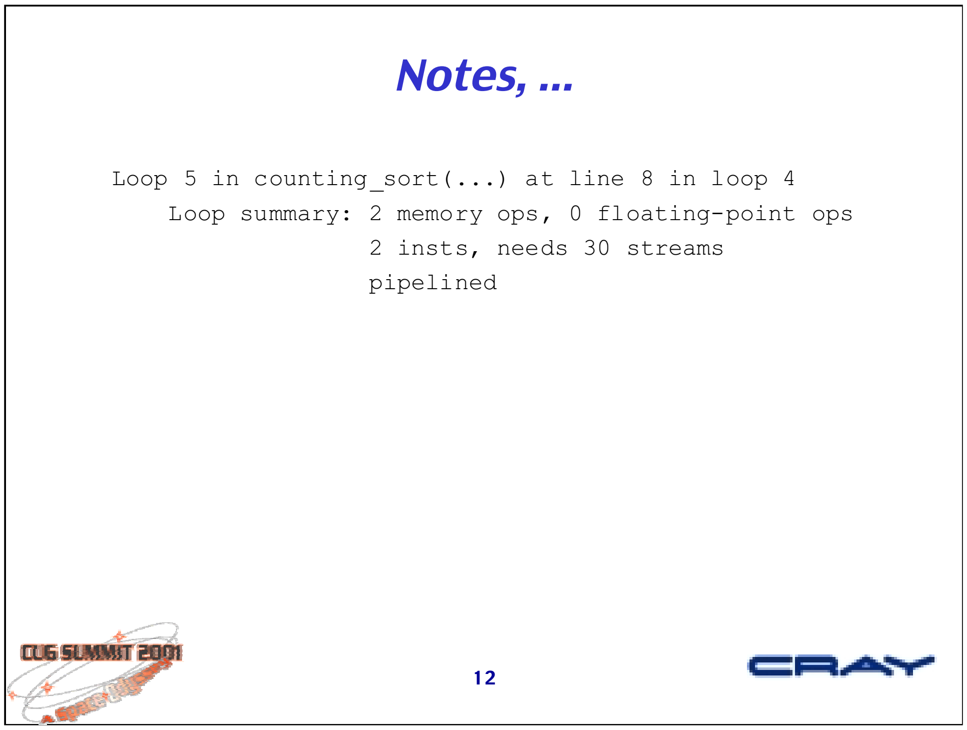
Clicking on an annotated statement displays notes about the enclosing loops:

```
Loop 5 in counting_sort(...) at line 8 in loop 4
    Loop summary: 2 memory ops, 0 floating-point ops
                  2 insts, needs 30 streams
                  pipelined

Loop 4 in counting_sort(...) in region 1
   In parallel phase 2
   Dynamically scheduled, variable chunks, min size = 8

Parallel region 1 in counting_sort(...)
   Multiple processor implementation
   Requesting at least 40 streams
```

There’s lot of interesting information here and you needn’t understand it all in the beginning. I’ll walk you through it all, in stages.

In this case, there is only one enclosing loop in the source code (labeled Loop 5). The next loop out (Loop 4) was created by the compiler as part of its implementation for this particular parallel loop.

The last note refers to the parallel region enclosing the whole set of loops.



Notes, ...

```
Loop 5 in counting_sort(...) at line 8 in loop 4
    Loop summary: 2 memory ops, 0 floating-point ops
                  2 insts, needs 30 streams
                  pipelined
```

Probably the first thing to look at is the number of instructions in the inner loop. In this case, we see 2 instructions/iteration.

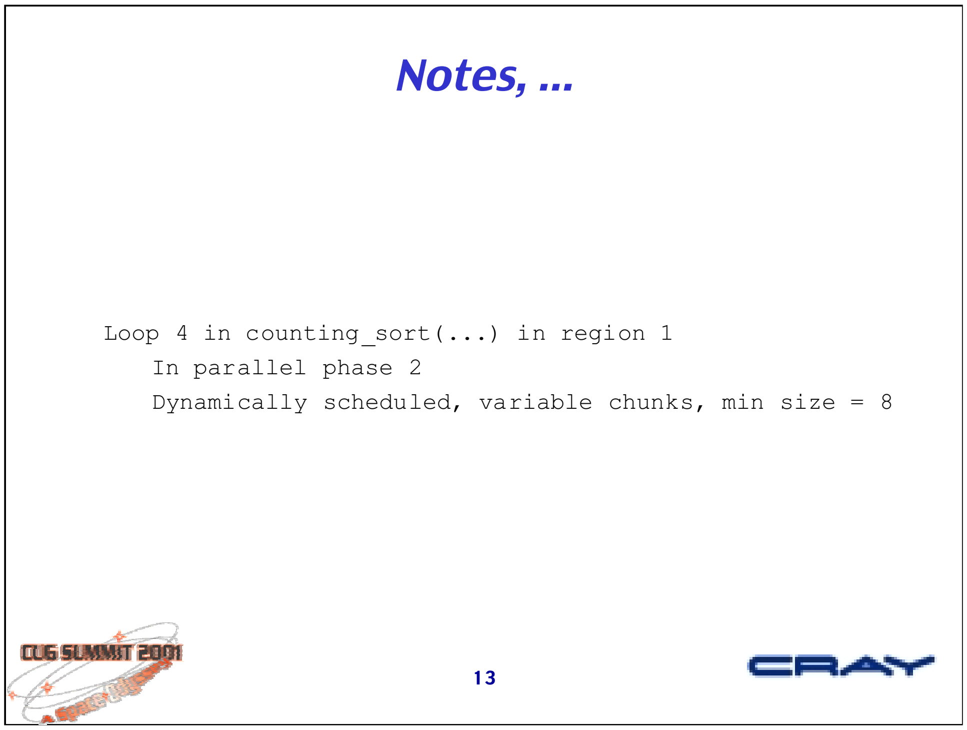
Next, we like to see that inner loops have been *software pipelined*. This refers to a compiler technique that significantly improves the instruction-level parallelism of a inner loop, plus helps reduce its size.

The notes about number of memory operations (loads, stores, fetch&adds) and floating-point operations (adds, subtract, multiplies) help explain how the work is balanced within the loop.

In this case, we see that the loop has 2 memory ops packed into 2 instructions/iteration. Since we can only have 1 memory reference/instruction, this indicates that the compiler has generated optimal code for this loop (that is, you could not do better in assembly language).

Finally, the note saying that 30 streams are required for full utilization is an indication of the compiler's estimate of the number of streams per processor it would like to have executing this loop.

Typically, when the program is executed, the code will request at least this many streams/processor for this loop. It may actually get less (if other streams are already busy).



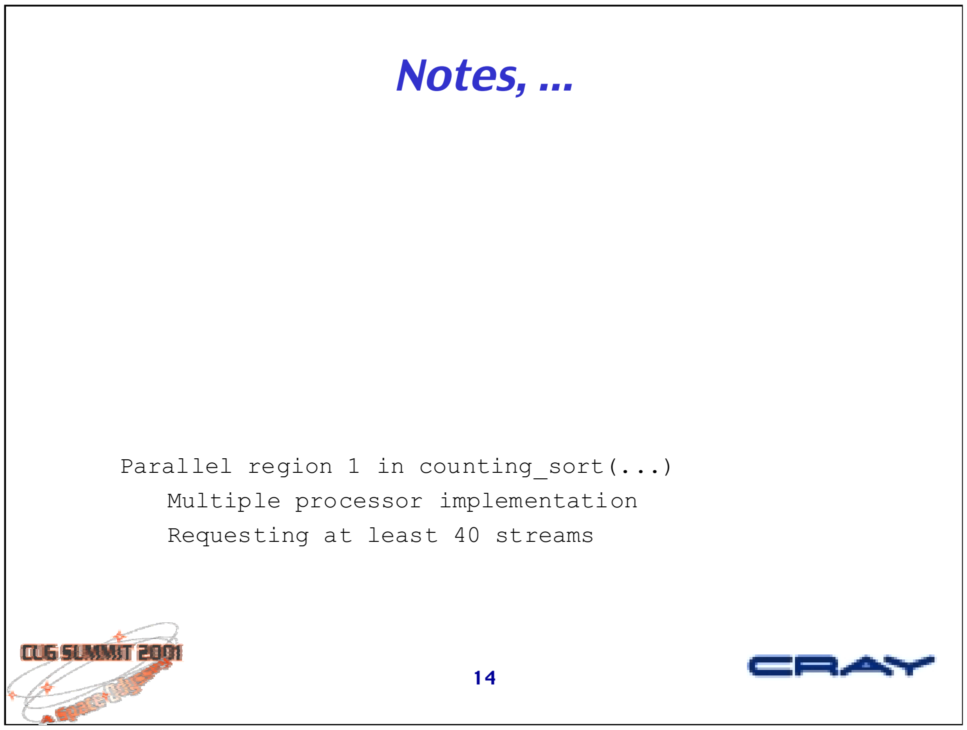
Notes, ...

```
Loop 4 in counting_sort(...) in region 1
   In parallel phase 2
   Dynamically scheduled, variable chunks, min size = 8
```

In this section, we get more information about the approach used by the compiler to parallelize the loop.

In this case, the compiler has chosen to use a technique called *dynamic self scheduling* (referred to here as dynamic scheduling with variable chunks). Other possibilities include interleaved, blocked, dynamic, and future scheduling. The programming guide discusses each of these approaches and the tradeoffs between them.

The other information here is that this loop is the second phase in region 1. Basically, a parallel region consists of a fork-join pair surrounding one or more phases separated by barriers. By examining canal's notes about phases and regions, the interested programmer can understand how the compiler has organized the code and work to help minimize overheads.

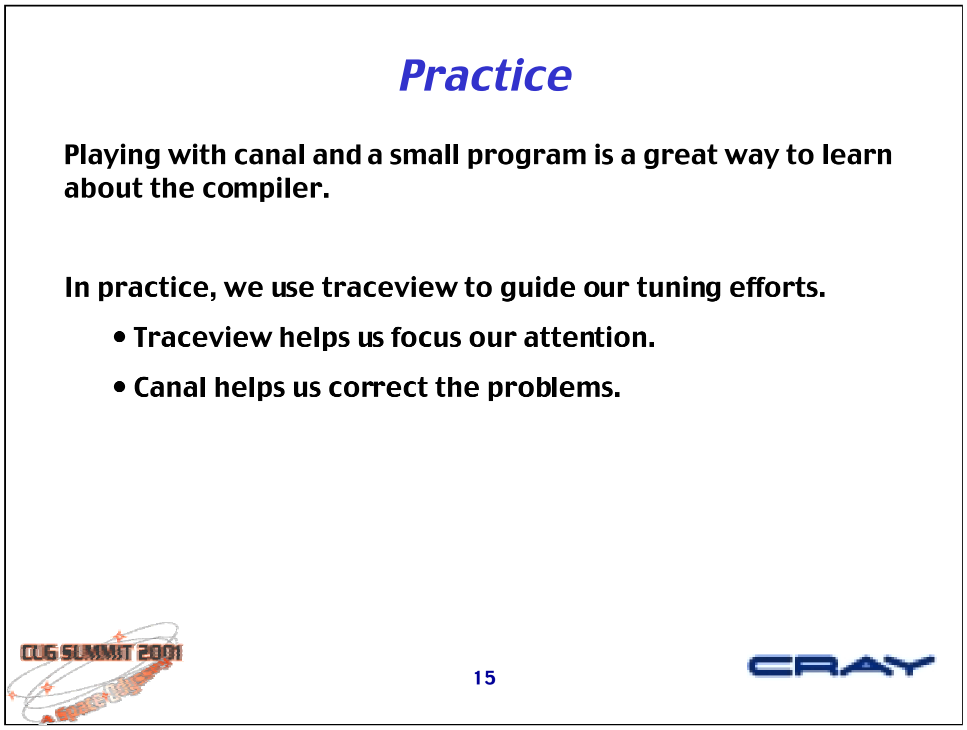


Notes, ...

```
Parallel region 1 in counting_sort(...)
    Multiple processor implementation
    Requesting at least 40 streams
```

There are two things to notice here:

- The compiler has generated code to use multiple processors for this region (the alternative is many threads on a single processor, which would have less overhead but less performance potential).
- The compiler wants 40 streams/processor for this region. Usually this number will be the maximum for all the loops in the region.



Practice

Playing with canal and a small program is a great way to learn about the compiler.

In practice, we use traceview to guide our tuning efforts.

- Traceview helps us focus our attention.
- Canal helps us correct the problems.

CUG SUMMIT 2001

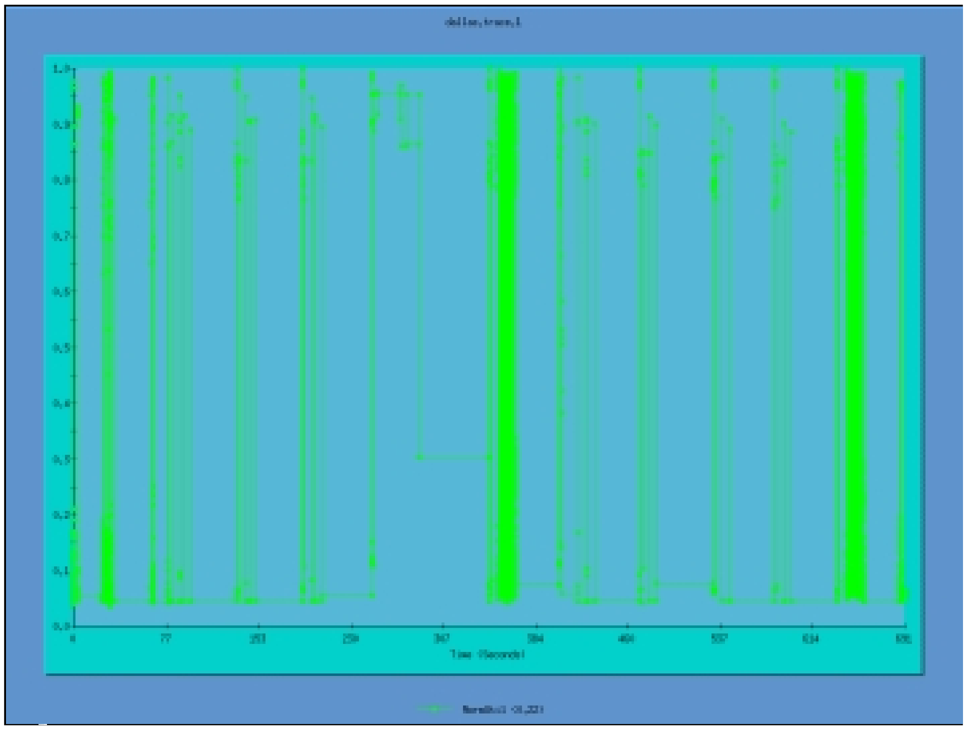
CRAY

When I write new code from scratch, I have strong opinions about how the compiler should do things, and I'll use canal to verify that the compiler is doing the right thing (or that it has come up with a better approach).

Then I'll run the code, again using traceview to verify my prejudices, fixing any surprises that appear.

When I port code, traceview guides me as I learn my way around, showing me what's significant and where I should be spending my efforts.

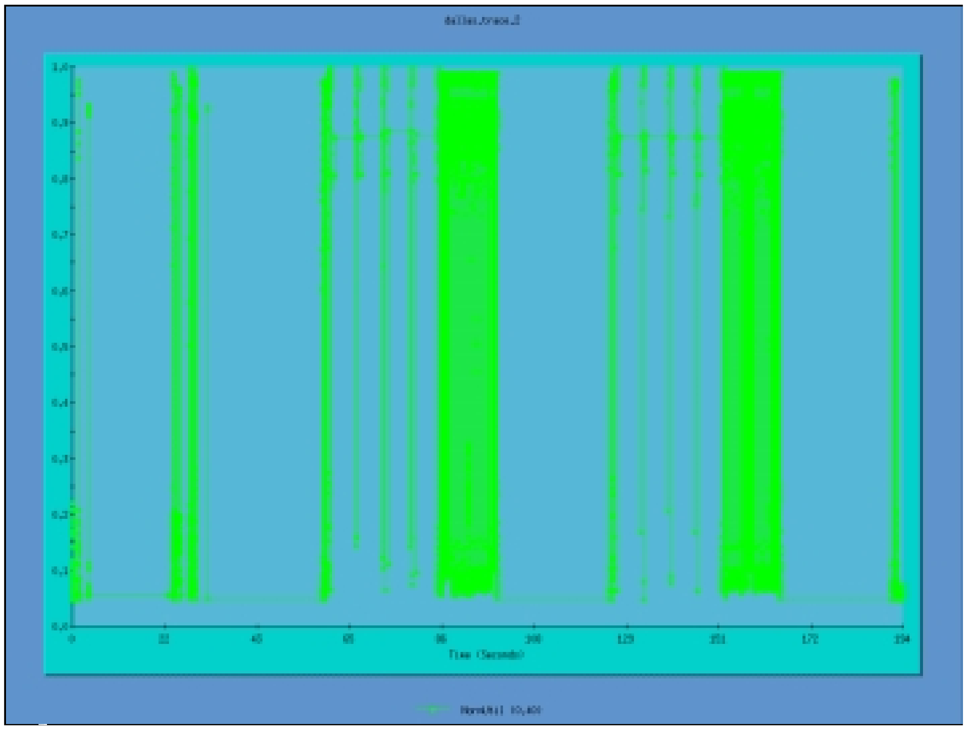
In all cases, traceview helps me understand when I am finished!



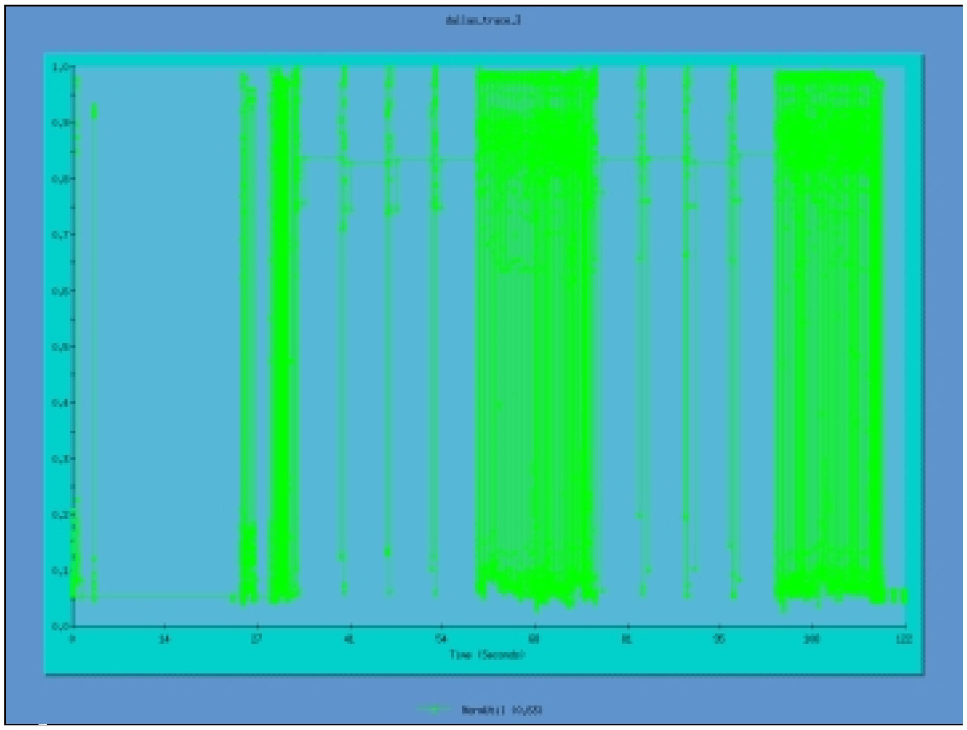
These last few slides show progress during a week-long period of tuning on a CFD code.

In this case, I've shown the *normalized utilization*; in other words, the plot indicates what percentage of the available cycles are being used by my program. Since this is early in the tuning process, the percentage is fairly low – about 22% across the complete run.

The advantage of displaying normalized utilization is that it "filters out" interference from other users and makes the number of processors irrelevant. We're only concerned with using all of the available instruction issue slots, regardless of how many there really are.

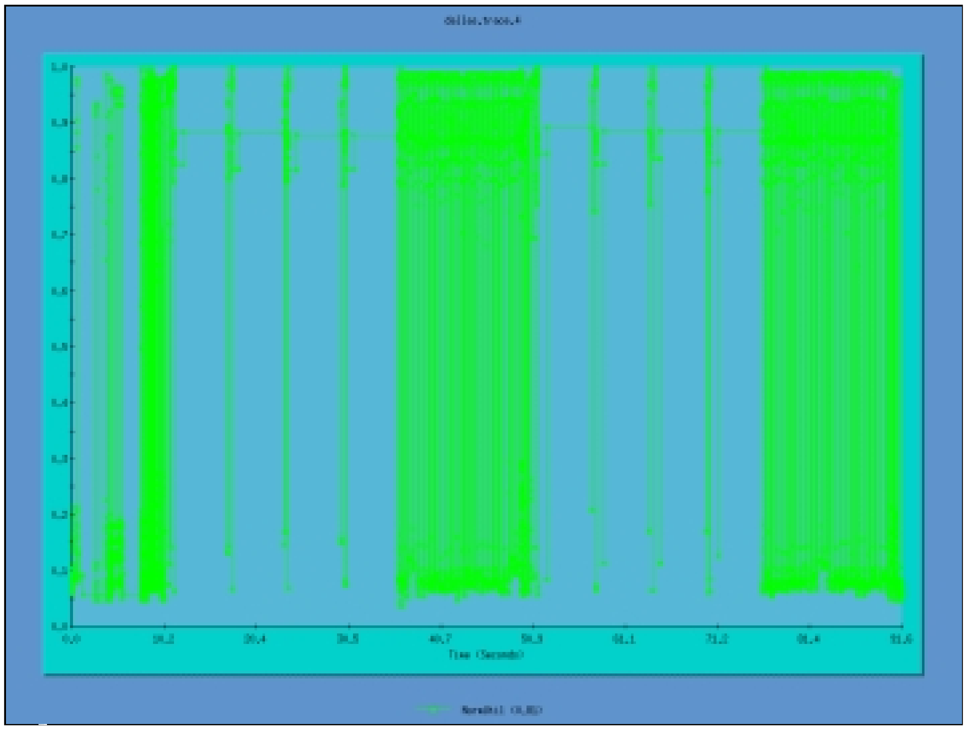


Here is there trace produced after rewriting a single routine. While there are still some significant serial sections, the utilization and runtime have both improved significantly.



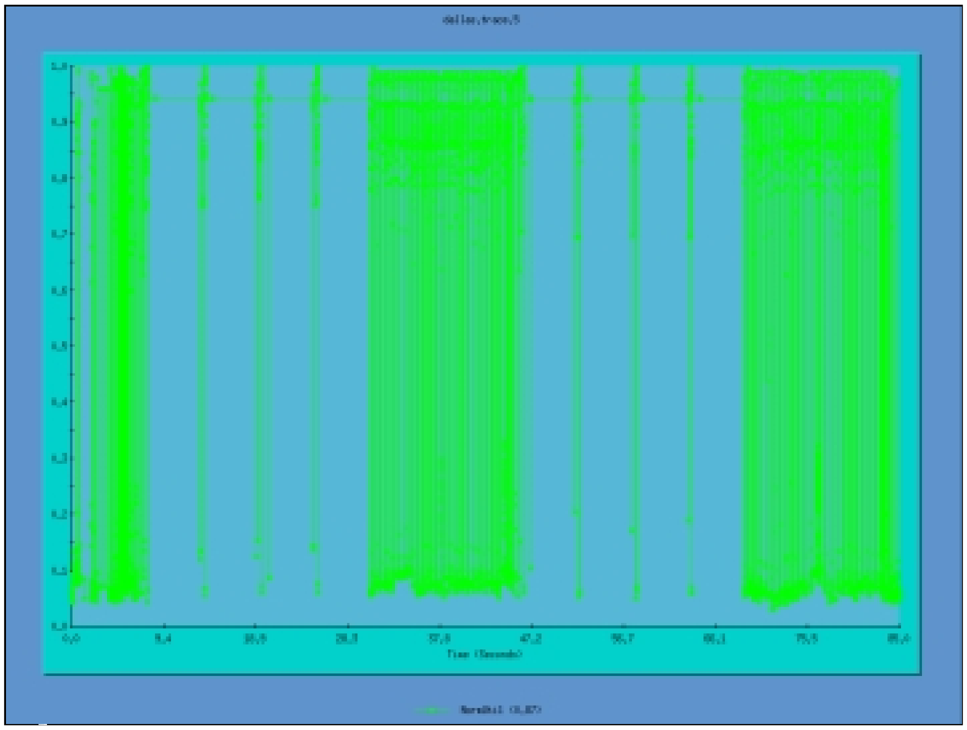
Here’s the result after fixing another routine.

Notice how the remaining problem areas seem to expand (since the rest of the program is shrinking). Amdahl would not be surprised.



Now we're getting to the point where there's very little left to do.

I attack the last remaining serial sections and try to improve the utilization in some of the big parallel regions (by increasing the number of streams requested by the compiler for those loops).



Looking pretty good. We're getting about 87% utilization across the entire run. The next step. would be to explore different input data, trying to exercise different parts of the code.

Notice that we've been working away without any indication of the number of processors available on the machine. In this case, I actually restricted myself to a single processor, in an effort to "play nice" on a busy machine, but the same approach could be used on 10 or 100 processors. In other words, this approach is scalable.