Performance Monitoring on a Shared-Memory Multiprocessor

Ziya Aral, Ilya Gertner, Jim Grier, Greg Schaffer
Encore Computer Corporation
257 Cedar Hill Street
Marlborough, Ma. 01752-3004

Introduction

For the majority of applications, the only reason for parallelization is improved performance on multiprocessors. Parallelizing an existing application is easy: start multiple instances of the same program, check all data references that have the potential of being cross referenced from different processors and introduce locks to protect those shared data structures. The result is a parallel program that implements the same functionality as its sequential ancestor. If that parallel program now runs faster and scales efficiently to the number of available processors, the job of parallelization is essentially complete.

If, however, as in the majority of cases, the program does not run efficiently in parallel and is not able to use the processors that are available, the really difficult task begins. The program must be modified or redesigned to take advantage of the available parallel hardware. Otherwise the significant performance advantages of multiprocessors will remain unrealized.

Performance debugging on a multiprocessor is as important as functional debugging on unprocessors [17]. Nevertheless, performance debugging still relies largely on conventional tools and utilities. This paper argues that the capabilities of multiprocessors may be used to help overcome the performance problems created by them. In the process, the assumptions underlying some existing performance analysis tools are re-examined in light of the new technology.

Existing UNIX performance measurement tools such as gprof are important for understanding the behavior of both sequential and parallel applications. They also demonstrate significant limitations. Some of these limitations such as the intrusiveness of performance monitors, the overwhelming volume of output they generate, and their inability to recognize system overhead charged on behalf of user applications, have been well noted in the past [12]. Other problems are less apparent and appear only as a result of attempting to use such tools to improve the performance of real parallel applications.
This paper describes our experience in attempting to improve the performance of a complex parallel program, a multitreaded symbolic debugger. One result of that experience was to gain insight into the limitations of existing Unix performance tools and utilities. Another result was to better determine the working specification for new types of parallel tools and the performance debugging methods that correspond to them.

The performance monitor described here is implemented in Parasight, a novel parallel programming environment designed to apply the capabilities of shared-memory multiprocessors to performance analysis and debugging. Parasight runs under MACH, a new UNIX 4.3BSD kernel implementation [14]. It has also been ported to UMAX 5.0, a more conventional parallel adaptation of the AT&T's System V [5]. The graphical representation used by the performance monitor described here is similar to the graphical representation employed by PIEscope, a component of PIE [16]. The multi-threaded debugger which here serves as target is itself another part of the Parasight environment [2].

Parasight has been implemented on the Encore Multimax (TM) and maps closely to the hardware of this multiprocessor [4, 13]. Multiple processors are used to concurrently run both the target and performance analysis tools; shared-memory is used to effect communication and control. Program instrumentation is performed at run-time and moved implemented user rather than system level. As a result, the latencies inherent in kernel intervention are bypassed and an improvement in efficiency of several orders of magnitude is gained [3].

Parasight is a child of the multiprocessor environment. It is capable of utilizing many processors to aid debugging and performance analysis. It is intended as an application of parallel processors to the development of parallel programs. Nevertheless, one byproduct of Parasight has been that these facilities have proven equally useful when applied to sequential programs. Moreover, some of the techniques employed seem to have direct application for uniprocessor architectures.

Overview

Parasight is a parallel programming environment for symmetric shared-memory multiprocessors. It can be applied to the performance analysis of both parallel and sequential programs. In contrast to conventional Unix tools, Parasight supports non-intrusive monitoring of shared memory. In addition, Parasight provides facilities for low-cost user-level instrumentation of application programs.

The motivation for Parasight comes from the difficulties inherent in developing parallel programs. The issues involved are so overwhelming that some researchers have come up with completely new programming environments including sophisticated graphics-oriented user interfaces, new parallel languages, compilers, performance analysis tools and debuggers [5, 11, 17, 15]. While these are often very powerful tools, they are generally confined to their own environments.

Parasight was influenced by these developments but has an essentially different motivation. Parasight is designed to support the analysis of parallel programs developed with conventional tools and languages as well as with new languages and techniques. The underlying assumption of Parasight is that low-cost program instrumentation, access to shared memory and symbols, and easy availability of interactive parallelism from within the performance analysis environment will be of general usefulness to most programming methodologies irrespective of the specific paradigms employed. Parasight is therefore intended as a general-purpose platform from which users may apply the high-level abstractions of their choice to independently developed applications.

The key feature of Parasight is its ability to dynamically create and link "parasite" programs into a parallel application. These are then able to interact with program memory and to create and control program instrumentation.

The idea of parasite programs grew out of the desire to support "programming for observability", an important concept for performance efficient parallel programming [16], and the need to support high-level debugging functions for parallel programs [1]. Parasites can be thought of as instrumentation programs that have been written for the sole purpose of "observing" a target application.

While parasites have access to the memory and symbols of a target, this is not sufficient. Some form of target instrumentation is necessary to detect conditions contingent on a specific event in the target's execution behavior. Parasight provides for instrumentation points which can be dynamically inserted and deleted from a running target. To reduce intrusiveness, "light-weight" instrumentation primitives called "scan-points" have been provided. These may be inserted and deleted at run-time.

Scan-points can be thought of as generalized breakpoints. They can call an arbitrarily complex, user-supplied, subroutine which is independently loaded and linked in the same manner as parasites. Different subroutines may be attached to different scan-points and the subroutine attached to any individual scan-point may be changed dynamically.

When invoked, Parasight takes control of the address space it shares with the target application. It spawns a thread of control executing the Parasight command line. It uses a resident dynamic loader to load and link the user's target program and builds a memory resident symbol table which is later used to link parasites. The target is effectively embedded in Parasight.

Once the Parasight environment has been created, the target program may be executed normally with no perfor-
mance penalty imposed on it. Computation proceeds without any regard to Parasight itself. Likewise, parasites may be created and started and instrumentation may be installed. The parasites created will execute as separate threads of control in the same space, typically on processors different from those used by the target.

The flexibility of parasites comes from the simplicity of the memory model. Since parasites run as threads in the same shared memory space, they have access to the target program's global symbols as well as symbols declared in other parasites. All routines declared in Parasight, in the target, or in parasites are cross-callable. The memory resident symbol table is in a standard UNIX format. The table is updated each time a new parasite is loaded. Global symbols are available to the environment without the need to stop target program execution.

Parasites may be written to monitor a specific user program or they may be much more general, implementing, for example, a symbolic debugger or interactive performance monitor.

User Interface

The new performance monitor runs on top of the X11 package. Figure 1 contains the major windows that control the Parasight environment.

The top (First) window displays the currently available parasites that include the user program (test1) and tools such as the C interpreter (cint), source-level debugger (cexe), incremental compiler (icom), the display manager that controls the image in Figure 1 (XPara), and the performance monitor (paragraph). This window has commands (or buttons) that control user programs (commands run, start and stop) and major tools (command Tools).

The Second window provides an interface to the management of this Unix objects such as directories and files. With a click of a button the user selects a file (or a program) that is brought up in the third window.

The third window is a minimal editor or a file browser that is tightly integrated into the environment. With a click of a button the user selects a line that can be instrumented to either insert a breakpoint or sensor. The same editor can be also used to single-step a program or insert a scan-point to call any user specified function. This window also includes commands that interact tightly with many other subsystems. For example, the button Value invokes the C interpreter to evaluate the selected variable. We are currently working on the commands insert/delete that manipulate the program source and incrementally compiles the changed code.

The performance monitor in Figure 2 (called paragraph) is invoked with the tools button. It supports three functions: Analysis, Instrumentation, and Display. Paragraph can analyze a program and display it at the desired level of abstraction. The most commonly used display is the call graph shown in Figure 2. Upon request, the button "show structure" will exhibit all control statements. Having displayed the program, the user can browse through it with the mouse by scrolling the graph along both the horizontal and vertical dimensions.

While browsing the program, the user may instrument the key points of the program with the command button. Having instrumented the program, the user runs it. The results are collected in a circular buffer. The circular buffer is continuously scanned and the results are displayed in the bargraph show in Figure 3.
Implementation

The new performance monitor is implemented using both the scan-points and parasites. Scan-points replace compiler inserted performance monitoring code; parasites collect data from the scan-points and display the results. Additional processors are utilized to produce graphical images of both the program structure and performance monitoring results.

The implementation of the monitor can be conveniently described in four steps:

- Analysis,
- Instrumentation,
- Collection,
- Display.

The analysis of the program sources was implemented as part of a C language syntactic analyzer. For every procedure call or, optionally, for every control statement (conditional or loop) an entry was generated and later used to compose a graphical representation of the program.

The graphical representation in the form of an extended call graph was instrumented at key points selected by the user via a "point and click" interface. The user manually selected the point to be instrumented or specified a set of entry calls all sharing similar naming conventions (e.g. all functions in a subsystem that begin with the prefix "foo").

The graphical interface was originally implemented as a simple text-based call tree but later rewritten as an X11 interface which mimics the presentation of PIEscope, a performance tool which is part of the PIE project at CMU [17]. This interface is described in the User’s Manual [10].

Having located the points in the program where monitoring points needed to be inserted, scan-points were used to implement the instrumentation. The actual routines for inserting monitor points were trivial and could be readily extended to support custom requirements:

```c
/* insert Mcount scan-point at function entry */
monitor_function(name)
char *name;
{
    addr = function_addr(name);
    do_insert(addr, &sp);
    sp->func = Mcount;
    sp->trigger = P_FUNCTION;
}

/* monitor entire application program */
monitor_program()
{
    for (proc = procs[0]; proc++)
        monitor_function(proc->name);
}
```

The function monitor_function above inserts a scan-point for a function-call. The function monitor_program inserts a scan-points for all function-calls. The function Mcount() below is a routine that performs the collection of statistics. The implementation details of the new performance monitor are straightforward and are described in simplified “C” code below. The performance monitoring scan-point function, Mcount(), simply stores call information in a circular buffer.

```c
/* save (from, to) link in circular buffer */
Mcount(sp)
SCANPOINT *sp;
{
    p = alloc_cirbuf();
    p->selfpc = sp->pc;
    p->frompc = calculate_frompc();
    p->time = get_time();  /* save time-stamp */
}
```

The display of statistics is performed with the functions monitor_collect() which extracts statistics from the circular buffer and monitor_display which displays the statistics in a graphical form. Both functions are running in another parasite thereby reducing to minimum the interference on the running target program.

```c
/* collect process monitor saved by Mcount */
monitor_collect()
{
    for (;;) {
        check_buffers();  /* check for overflow */
        p = get_from_buffer() /* get next monitored datum */
            build_dynamic_call_graph(p->selfpc, p->frompc);
    }
}
```

```c
/* output dynamic call graph in symbolic form */
monitor_display()
{
    from = function_name(frompc);
    to = function_name(selfpc);
    count = get_count();
    monitor_output(from, to, count, filter);
}
```

The monitor_display() routine is very flexible. The user may provide an arc filtering routine to limit what is displayed in the window. As an example, the routine filterarc(), below, checks for nodes in an output list. Only the nodes on the list appear as displayed triples.
filter_arc(from, to)
int from, to;
{
    int i;
    for (i=0; i < output_list[0]; i++)
    if ((output_list[1] == from) ||
        (output_list[1] == to))
        return(TRUE);
    return(FALSE);
}

To allow rapid development and experimentation with the performance monitor, the parasitic "C" interpreter 'cint' was used to directly exercise fine control. Something of the flavor of the environment can be seen in how easily the user can thus modify the output list of the example filter from the "C" interpreter:
Cint: output-list[0] = 2;
Cint: output-list[1] = get_address(func1);
Cint: output-list[2] = get_address(func2);

The function filter.arc() and the user script of interaction with the "C" interpreter represent only one example of the flexibility with which the user can start, stop, and control performance monitoring. Different filtering functions may be defined and applied to the raw performance data. For example, the user may incorporate filtering routines to output only the events of interest. Along these lines, we are currently working on a race-condition detector which uses this trace to describe the sequence of events that leads to a race condition.

Example

This section describes, as a simple example, the application of the monitor to the source-level debugger "cexe" (Controlled Execution Environment) which runs as just another parasite in the Parasight Environment [2]. Cexe makes use of a very fast instruction emulator to implement single stepping rather than relying on the Operating System's stepping services (in UNIX, ptrace(2)). As a result, fine grained tracing using instruction stepping should be much faster with cexe than with a conventional debugger. In practice however, the stepping performance of cexe was found to be not much faster than a normal debugger. The performance monitor was applied to answer why.

A cexe session in which a user single-stepped a parallel program "foo" was employed. A sample of the single-stepping session was:
cexe: a (3,4) /* apply to threads 3 and 4 */
cexe: step 10 /* step 10 stat. for both threads */
[4] breakpoint foo.c:18 - trigger 0x3
[3] foo.c:23 j = cthread_index();
[4] foo.c:18 j = cthread_null();
    j = cthread_null();

The purpose of this example was to demonstrate the behavior of cexe's single step mechanism. It should be noted that cexe is a quite complex parallel program implementing a parallel source-level debugger.

The monitor was applied in the four stages mentioned earlier:

- the first step Analyzes the program sources and produces a graphical representation of the program's control structure.
- the second step Instruments the key points in the user program.
- the third step Collects statistics of the selected events in a circular buffer.
- the fourth step Displays the statistics in a graphical form.

A user begins by starting the Parasight console window [Figure 1] that supports the management of parasites and scan-points (the top first region in the Figure 1), UNIX object browser (the second region), editor (the third region) and language interpreter (the fourth region). In the top of the region, the user invokes Paragraph, the parasite that implements the new monitor.

The user begins by invoking the Analyzer that produces a graphical representation of the control structures in the
target program [Figure 2]. For large programs, the user navigates through the graph by moving the window using the scroll bar along the horizontal and vertical axes. Having identified the broad areas of interest, the user selects the specific points of interest by clicking on the mouse button. In the figure, the user selects and instruments the points findScanPoint, branch and CEXEstep that account for the most of the time in single-stepping a program. Then, the user runs the program to collect statistics accumulated in the circular buffer. Finally, the user displays the results of the statistics in the bargraph [Figure 3]. Depending upon the density of the results, the user may zoom-in or zoom-out of the statistics [Figure 4].

The actual measurements in the Figure 3 and Figure 4 represent an instance when a parallel program is single-stepping a source. In one case, the single stepping occurs with much higher frequency because the steps are complex function calls.

Using this simple mechanism, an error was quickly located in the cexe code. An unanticipated interaction between two simultaneously stepping threads of control was causing the invocation of address to symbolic translation, a costly process, far more often than was really necessary. An important performance bug was easily removed.

Conclusions

Parasight has been implemented on the Multimax running both the MACH and UMAX V.0 operating systems. It has been applied to the performance monitoring of the source-level debugger, a complex parallel program. Preliminary results demonstrate the applicability of Parasight to both extending existing performance monitors (e.g. gprof) and writing custom performance monitors.

In extending standard performance-monitoring tools such as gprof, with little effort we were able to implement a function performance monitor that supports the same functionality as gprof but is interactive and less intrusive. The new performance monitor runs concurrently with the target program. Upon user request, it displays statistics as they are being accumulated. The user may also provide custom filtering procedures that display only portions of the collected data. In addition, the new performance monitor is less intrusive than the old gprof. Part of the instrumented code overhead has been moved to another processor. Gprof averaged 36 instructions for every instrumented subroutine call. The new performance monitor needs only 11 instructions to store the statistics of the call in a circular buffer. The circular buffer is processed by a parasite program that runs on an adjacent processor and has no effect on the target program.

Our new performance monitor is very useful for collecting preliminary measurements to identify bottlenecks in a program. It remains, however, a coarse tool that provides only a broad overview of the program's behavior. While it is useful for understanding the weight of a given function in an application or for identifying "hot" spots, it still remains for the user to translate this information into specific changes made to the target. In practice, we have identified a cycle in which this coarse analysis of program behavior alternates with fine instrumentation of key points in the target and with modifications to the target itself. The main function of the new performance monitor has been to allow the user to "zoom-in" on problem areas.

Once key points have been identified, our environment appears to be very powerful. Using Parasight, custom performance-monitoring routines may be inserted at arbitrary locations in the program. Different abstractions may be formulated and applied to the same program to extract different kinds of information.

Parasight has been developed for the Multimax, a moderately parallel computer. In the future, it will be applied to ULTRA, a massively parallel system totalling more than one thousand MIPS. The concept of parasite programs running on adjacent processors and monitoring the same shared memory space seems to scale naturally to such massively parallel architectures. In addition, low-cost program instrumentation primitives open new opportunities for high-level program instrumentation at run-time. Eventually, we plan to take advantage of system design tools and graphics interfaces to automatically instrument programs at key points and watch their behavior in terms of graphically described abstract models. We are only beginning to learn the potential of parallel architectures.

Acknowledgments

The authors wish to acknowledge Dr. Zary Segall and Dr. Dalibor Vraslovic whose comments on Parasight and work on PIE provided an inspiration for the graphics display package in Parasight.

References


Figure 1 Parasight Console.
Figure 2  Call Graph.