VPPB - A Visualization and Performance Prediction Tool for Multithreaded Solaris Programs

Magnus Broberg, Lars Lundberg, and Håkan Grahn
Department of Computer Science
University of Karlskrona/Ronneby
Soft Center, S-372 25 Ronneby, Sweden
{Magnus.Broberg, Lars.Lundberg, Hakan.Grahn}@ide.hk-r.se

Abstract

Efficient performance tuning of parallel programs is often hard. In this paper we describe an approach that uses a uni-processor execution of a multithreaded program as reference to simulate a multiprocessor execution. The speed-up is predicted, and the program behaviour is visualized as a graph, which can be used in the performance tuning process.

The simulator considers scheduling as well as hardware parameters, e.g., the thread priority, no. of LWPs, and no. of CPUs. The visualization part shows the simulated execution in two graphs: one showing the threads' behaviour over time and the other the amount of parallelism over time. In the first graph is it possible to relate an event in the graph to the code line causing the event. Validation using a Sun multiprocessor with eight processors and five scientific parallel applications shows that the speed-up predictions are within +/-6% of a real execution.

1. Introduction

Parallel processing is an important way to increase the performance. It is often easier to develop parallel applications for shared memory multiprocessors than for message passing systems. Shared memory multiprocessors are therefore becoming increasingly important.

The thread concept in the Solaris operating system [13] makes it possible to write multithreaded programs which can be executed in parallel. Having multiple threads does, however, not guarantee that a program will run faster on a shared memory multiprocessor. One major performance problem is that thread synchronizations may create serialization bottlenecks which are often hard to detect.

Removing serialization bottlenecks is referred to as *performance tuning*. Different tools for visualizing the behaviour of, and thus the bottlenecks in, parallel programs have been developed [1, 2, 3, 5, 6, 9, 10, 11, 12, 14, 16, 18]. The tuning process may benefit significantly from using such tools.

Some performance visualization tools show the behaviour of one particular monitored multiprocessor execution of the parallel program [1, 2, 3, 5, 6, 9, 10]. If we monitor the execution on a multiprocessor with four processors such tools make it possible to detect bottlenecks which are present when using four processors. The problem with this approach is the lack of support for detecting bottlenecks which appear when using another number of processors.

There are a number of tools which make it possible to visualize the (predicted) behaviour of a parallel program using any number of processors. However, these tools are either developed for message passing systems [12] or for non-standard programming environments [11, 14, 16, 18].

In this paper we present a performance prediction and visualization tool called VPPB (**V**isualization of **P**arallel **P**rogram **B**ehaviour). Based on a monitored uni-processor execution, the VPPB system shows the (predicted) behaviour of a multithreaded Solaris program using any number of processors. To the best of our knowledge, VPPB is the only available tool which supports this kind of flexible performance tuning of parallel programs developed for shared memory multiprocessors using a widely spread standardized parallel programming environment (Solaris).

Validation using five scientific multithreaded programs from the SPLASH-2 suite [19] and a multiprocessor with eight processors showed that VPPB was able to predict the behaviour very accurately. The maximum difference between the real speed-up and the speed-up predicted by VPPB was 6%, and for most cases the difference was less than or equal to 1%. As discussed above, the predictions are based on recordings from a monitored uni-processor execution. The time overhead for doing these recordings was less than 3% for all five programs.

The paper is structured in the following way. Section 2 gives a short overview. In section 3 the implementation is described. Section 4 describes the validation. A small case study is shown in section 5. Section 6 discusses the limitations and applicability. Section 7 concludes the paper.

2. Overview of VPPB

The VPPB consists of three major parts, the *Recorder*, the *Simulator*, and the *Visualizer*. The workflow when using the VPPB system is shown in figure 1. The developer writes the multithreaded program, (a) in figure 1, compiles it, and an executable binary file is obtained. After that, the program is executed on a uni-processor. When

starting the monitored execution (b), the *Recorder* is automatically placed *between* the program and the standard thread library. Every time the program uses the routines in the thread library, the call passes through the *Recorder* (c) which records information about the call, i.e., the identity of the calling thread, the name of the called routine, the time the call was made, and other parameters. The *Recorder* then calls the original routine in the thread library. When the execution of the program finishes all the collected information is stored in a file, the *recorded information* (d). The recording is done without recompilation or relinking of the application.

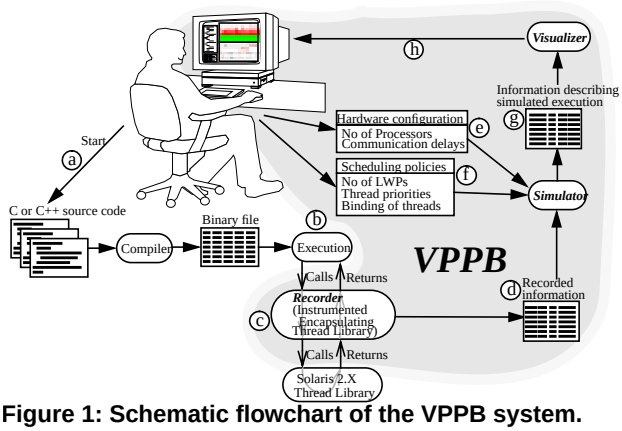


Figure 1: Schematic flowchart of the VPPB system.

The *Simulator* simulates a multiprocessor execution. The input for the simulator is the *recorded information*, (d) in figure 1, the hardware configuration (e), and scheduling policies (f). The output from the simulator is information describing the predicted execution (g).

Using the *Visualizer* the predicted parallel execution of the program can be inspected (h). The Visualizer uses the simulated execution (g) as input. When visualizing a simulation, it is possible for the developer to use the mouse to click on a certain interesting event, get the source code displayed, and the line making the call that generated the event highlighted. With these facilities the developer may detect problems in the program and can modify the source code (a). Then the developer can re-run the execution to inspect the performance change. The VPPB system is designed for C or C++ programs that uses the built-in thread package in the Solaris 2.X operating system.

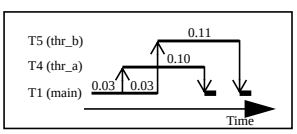
3. Tool Description and Implementation

3.1. The Recorder

In order to trace the behaviour of the program when executed on a uni-processor, the Recorder inserts probes when the program starts. The probes are inserted at specific events, i.e., before and after calls to the thread library, and they do not affect the behaviour or function of the program. For each event, the probes record the following information: when an event has occurred; the type of event, e.g., locking of a mutex; which object the event concerns, e.g., the identity of the mutex being used; the identity of the thread generating the event; and the location of the event in the source code. The data collected by the Recorder is kept in memory until the program terminates, then the recorded data is written to a log file. By using this technique, the intrusion is kept to a minimum.

We will use a small multithreaded program, found in the upper left corner of figure 2, as an example when demonstrating the functionality of the VPPB. The optimal parallel execution of this program can be found in the lower left corner of figure 2; a solid line denotes execution, no line that the thread is blocked, and an arrow represents an event. The Recorder executes the threads sequentially on a uni-processor. The output of the Recorder is the list of events found on the right side of figure 2. The sequentially ordered list is used as the behaviour profile when simulating a multiprocessor execution of the program.

```
void* thread(void *) {
 work();
}
int main() {
 thread_t thr_a, thr_b;
 thr_create(0, 0, thread, 0, 0, &thr_a);
 thr_create(0, 0, thread, 0, 0, &thr_b);
 thr_join(thr_a, 0, 0);
 thr_join(thr_b, 0, 0);
}
```



time	thread	action	duration
0.00	T1	start_collect	
0.0	T1	Running	0.03
0.03	T1	thr_create thr_a	0.04
0.0	T1	Running	0.03
0.10	T1	thr_create thr_b	0.02
0.0	T1	Running	0.05
0.17	T1	thr_join thr_a	0.05
0.0	T4	Running	0.10
0.32	T4	thr_exit	
0.0	T5	Running	0.11
0.43	T5	thr_exit	
0.0	T1	Running	0.10
0.53	T1	ok thr_join thr_a	0.05
0.0	T1	Running	0.01
0.59	T1	thr_join thr_b	0.05
0.0	T1	Running	0.10
0.74	T1	ok thr_join thr_b	0.05
0.0	T1	Running	0.01
0.80	T1	thr_exit	

Figure 2: An example program and the output from the Recorder. The operating system assigns the following thread identity numbers to the threads: main = 1, thr_a = 4, and thr_b = 5. We will use T1, T4, and T5, respectively, when we refer to the different threads.

Our current implementation of the Recorder is based on the ideas described in [8]. We insert a new library between the program and the dynamically linked library `libthread.so.1`, which implements the threads in Solaris 2.X. This is achieved by using the built-in facilities of Solaris with run-time linking and shared objects. The insertion is handled at program start-up by the run-time linker via an environment variable called `LD_PRELOAD`. The probes in the inserted library are exemplified in figure 3, where we show how `thr_exit` is implemented.

The probe does four things. First it looks up the address of the real implementation of `thr_exit` and stores it in a variable. This is done only the first time the probe is called. The next thing is to make a time stamp and store the data about the event. The next part of the code stores which source line the thread primitive was called from. Finally, the probe calls the original function in the Solaris thread library.

The time recorded for each event is wall clock time with a resolution of 1 microsecond. We are can not moni-

```
void thr_exit(void *status) {
    static void (* fptr)() = 0;
    if ( fptr == 0 ) {
        fptr = (void (*)())dlsym(RTLD_NEXT, "thr_exit");
        if ( fptr == NULL ) {
            (void) printf("dlopen: %s\n", dlerror());
            return;
        }
    }
    mthr_collect(THR_EXIT, thr_self(), BEFORE, -1);
    asm("set returnpointer, %l0");
    asm("st %i7, [%l0]");
    mthr_recallAddress();
    (*fptr)(status);
    return;
}
```

Figure 3: The implementation of the `thr_exit` probe.

tor the kernel switches between LWPs (lightweight processes) and are forced to do the monitoring on one single LWP. A more thorough discussion is found in Section 6.

We have divided the tracing of the source code location of the call to a probe into two steps. The first step is to record where the calling code is placed in memory. This is done by saving the return address which is the place where execution will continue after the function has returned. On the SPARC processor this return address is kept in a CPU register called i7. The second step translates the recorded memory addresses into specific source code lines. This is done by using a source code debugger and a small parser, which converts the output from the debugger into a format that is readable for the Simulator and Visualizer.

In every `thr_create` call, a function pointer is supplied. This pointer contains the start address of a new thread. The function pointer is recorded and the debugger is used to translate the address to the function name in the source code.

3.2. The Simulator

The Simulator emulates the scheduling in Solaris 2.5 [13]. In Solaris, threads are used at two levels [17]. The application programmers use user-level threads for expressing parallel execution within a process. Kernel threads are used within the operating system kernel. The kernel knows nothing about user-level threads.

Between the user-level and kernel threads are LWPs. Each Solaris process contains at least one LWP. In most cases, user-level threads are multiplexed on the LWPs of the process, such threads are referred to as unbound threads. It is, however, possible to bind a thread to an LWP. Compared to unbound threads, it is much more expensive to create and synchronize threads which are bound to an LWP [17].

There is a kernel thread for each LWP. However, some kernel threads have no associated LWP, e.g., a thread to service disk requests. Kernel threads are the only objects scheduled by the operating system, and they can either be multiplexed on the processors in the system, or bound to a specific processor. To some extent, the user can control thread scheduling, e.g. by binding threads to LWPs and LWPs to processors. It is also possible to indicate how many LWPs a certain process should have. The (user-level) threads can be created dynamically at run-time.

In the Simulator, threads may be manipulated in the following ways: Each thread can individually be unbound; bound to a LWP; or bound to a certain CPU. A thread that is bound to a CPU is automatically bound to an LWP. Each thread can individually be assigned a certain priority level. This will then override all manipulation of that thread's priority within the log file, e.g., the `thr_setprio` event for that thread will be ignored.

Binding a thread to a CPU can increase the speed of the program [7]. When a thread is moved to a different CPU, parts of the old cache contents has to be moved to the cache on the new processor. This may result in a performance-loss. The Simulator does not simulate the caches, but it is possible to use this facility to determine which thread to bind to which CPU in order to get the best result from a load balancing point of view.

Not only user-level threads has a priority level, but also the LWPs. The priority of an LWP is set by the operating system and is adjusted during run-time depending on, e.g., whether the LWP is interactive or only runs in batch mode. The simulator emulates the priority adjustment as it is handled in Solaris. The length of a time slice for an LWP is related to the priority level, thus we also adjust the time slice length during our simulation.

The following parameters can also be adjusted: the communication delay between the CPUs; the number of processors; and the number of LWPs. In this case the `thr_setconcurrency` in the program has no effect. The communication delay affects how fast an event on one CPU is propagated to another CPU.

The concept of `mutex_trylock` and similar try-operations are handled in the following way: If the thread gained access to the lock in the log file, the simulation will do a `mutex_lock`, otherwise no action is taken by the simulator. The `cond_timedwait` is handled as a delay if the operation timed out in the log file and as an ordinary `cond_wait` operation otherwise. Consequently, the information in the log file corresponds to a deterministic execution of the program with some minor exceptions, which are explained in Section 6.

Creating a bound thread is simulated to take 6.7 times longer than an unbound thread [17]. A synchronization on a semaphore takes 5.9 times longer [17] with bound threads than unbound. This value is used in the simulator for mutexes, conditions, and read/write locks, as well.

When running the Simulator, all events in the log file from the Recorder are sorted into a set of lists, one list for each thread as shown in figure 4.

Our simulation technique is an ordinary eventdriven approach. When the simulation starts, all threads are

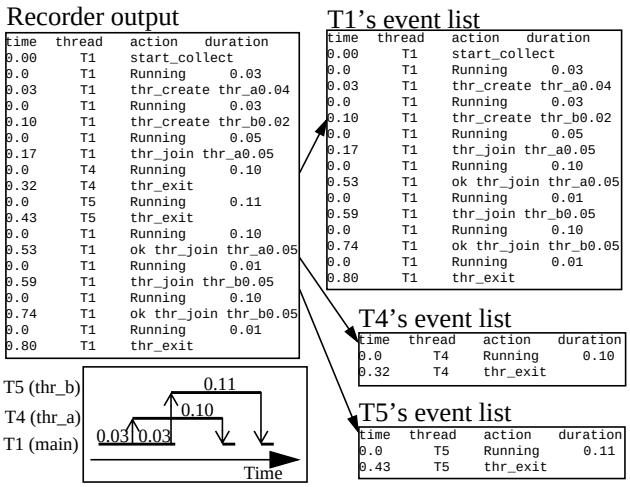


Figure 4: The Simulator's sorting of the log file from the Recorder. We use the same program as in figure 2.

marked as blocked except for the starting (main) thread, i.e., T1. In order to run the threads, the number of LWPs and CPUs specified by the user are simulated. Each (simulated) CPU picks a (simulated) LWP, which in turn picks a (simulated) thread. Each CPU executes the minimum time required for one of the threads to reach an event from the thread's list. The event is simulated and, if appropriate, some blocking or scheduling of threads or LWPs are done.

3.3. The Visualizer

The Visualizer offers two graphs: the parallelism vs. time graph, or *parallelism graph* for short; and the execution flow vs. time graph, or *execution flow graph* for short. The parallelism graph is the upper graph in figure 5. The higher the graph reaches the more parallelism exists in the application. The number of *running* threads are indicated with green. On top of the graph, all the threads that are *runnable* but not running are presented in red. It is easy see where the performance bottlenecks are in time as well as the potential parallelism. This kind of graph har previously been presented as two separete graphs in [15].

The execution flow graph (the lower graph in figure 5) contains more detailed information than the parallelism graph. In the execution flow graph the time is represented on the X-axis and the threads are represented on the Y-axis. A horizontal line indicates that the thread of that Y-position is executing, the lack of a line indicates that the thread can not execute, a grey line that the thread is ready to run but does not have any LWP or CPU to run on. Different events are displayed with different symbols and colours, e.g., all semaphores are shown in red, and the primitives `sema_post` and `sema_wait` are represented as an upward and a downward facing arrow, respectively.

The zoom utility can increase (or decrease) the magnification to an arbitrary magnification degree in steps of a factor of 1.5 or 3. The zoom keeps the left-most time fixed

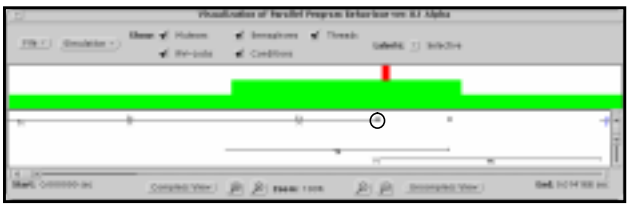


Figure 5: The execution parallelism and flow graphs after running a simulation.

in the execution flow graph. The user can mark a time interval in the parallelism graph , and the execution graph will automatically show only the marked interval.

When there are too many threads to fit in one display, irrelevant threads can be removed automatically. The compression only shows the threads active during the time interval shown in the execution flow graph. It is also possible to control which threads to be shown by hand, allowing the user to select which threads to show from a list.

By selecting a particular (interesting) event, e.g., when thread T1 joins with thread T4 (marked in Figure 5 with a circle), a popup window is shown that gives more information. The selected event starts to flash in the execution flow graph. The popup window gives information about the thread causing the event: the thread identity; the name of the function passed to the `thr_create` function; the time the thread started and ended; how long time the thread actually was working; and finally, the total execution time of the thread (including the time the thread was blocked or runnable). There are also information about the event: that the event was a join operation with thread T4; that the thread was running on CPU 0 in the simulated execution; when the event started, ended, and how long it took to perform; and the source code file and source code line.

The user can step to the previous or next event made by this thread. The execution flow graph is automatically scrolled in order to place the event in the centre of the window. The the popup window is updated with the corresponding data about the new event. Further, the user can find the next or previous similar event. This means that the next event caused by the same event type or variable, e.g., the next operation on the same mutex variable, will be found. Finally, the tool can start an editor with the source code file and highlight the line where the event took place.

4. Validation

The validation of the predictions was made using a subset of the SPLASH-2 benchmark suite [19]. The programs that we use from the SPLASH-2 suite are: Ocean (with data set 514-by-514 grid), Water-Spatial (512 molecules, 30 time step), FFT (4M points), Radix (16M keys, radix 1024), and LU (contiguous, 768x768 matrix, 16x16 blocks). All programs that we use are from the scientific and engineering domain.

The other benchmarks in the SPLASH-2 suite could not be used as validations. Barnes, Radiosity, Cholesky, and FMM could not run in one single LWP as required by the Recorder. The reason is that these programs all spin on a variable, and since the thread never yields the CPU, no other thread could possibly change the value of that variable. The program Raytrace and Volrend could not be used since all tasks that are executed by a thread are put in a queue. Whenever a thread is idle it steals a task from another thread's queue. The impact of using one LWP gives the result that only one thread steals all tasks, since it never yields the CPU.

All executions were made on a Sun Ultra Enterprise 4000 with 8 processors and 512MByte memory. Since the SPLASH-2 programs are designed to create one thread per physical processor, one log file were made for each processor setup when using the Recorder.

Table 1 shows the measured and predicted speed-up for the five programs. The real speed-up is the middle value of five executions of the program. The values between paranthesis show the maximum and minimum speedup for the executions. The error is defined as |((Real speed-up) - (Predicted speed-up))/(Real speed-up)|.

With one exception the predicted speed-up is very close to the real speed-up, i.e., the error is less than 1.5%. The exception is Ocean where the error is 6.2% on eight processors. However, the values between brackets show that the variations in the real speedup is rather large , and the predicted value is within the interval defined by the executions.

Table 1: Measured and predicted speed-ups.

Application/ Speed-up		2 processors	4 processors	8 processors
Ocean	Real	1.97 (1.97-1.98)	3.87 (3.85-3.89)	6.65 (6.42-7.11)
	Pred.	1.98	3.89	7.06
	Error	0.5%	0.5%	6.2%
Water-spatial	Real	1.99 (1.99-2.00)	3.95 (3.94-3.97)	7.67 (7.37-7.76)
	Pred.	1.98	3.91	7.56
	Error	0.5%	1.0%	1.4%
FFT	Real	1.55 (1.54-1.55)	2.14 (2.14-2.15)	2.62 (2.61-2.63)
	Pred.	1.55	2.14	2.63
	Error	0.0%	0.0%	0.4%
Radix	Real	2.00 (1.99-2.00)	3.99 (3.98-3.99)	7.79 (7.77-7.81)
	Pred.	1.98	3.95	7.87
	Error	1.0%	1.0%	1.0%
LU	Real	1.79 (1.78-1.80)	3.15 (3.12-3.15)	4.82 (4.74-4.90)
	Pred.	1.79	3.14	4.81
	Error	0.0%	0.3%	0.2%

Due to the recordings, the monitored uni-processor execution takes somewhat longer than an ordinary uni-processor execution of the program. However, our measurements showed that the execution time overhead for doing the recordings was very small. The maximum overhead, which was obtained for Ocean, was 2.6% of the total execution time. Another concern was the size of the log files. The largest log file, obtained for Ocean, was 1.4 MByte. This file could be handled without any problems. Consequently, neither the execution time overhead, nor the size of the log files caused any problems for these programs.

Programs with fine granularity generate more synchronization events, and thus larger log files, per time unit than coarse grained programs. The maximum number of events per second for our programs was 653 (Ocean). The uni-processor execution time for the five programs ranged from 60 seconds to 210 seconds. The size of the log files could become a problem for very long executions of fine grained programs.

We have done experiments with log files up to 15 MByte. Unfortunately the time required for obtaining the predicted speed-up values, and also the graph visualizing the behaviour of the program, increases for large log files.

5. A Simple Example

We use a producer-consumer problem to demonstrate how the tool can be used for improving the performance of an application. There are 150 Producers, each implemented by a thread, which inserts ten items in the buffer and then exits. There are 75 Consumers, picking one item each from the buffer. A semaphore is used to represent the number of items in the buffer, insertion and fetching of items is controlled by one mutex. The buffer size is large enough to avoid producer stalling as a result of a full buffer.

We began with making a log file on a uni-processor computer. After simulating the log file, we found that the program ran only 2.2% faster on 8 CPUs. To find out the reason of the poor performance, we use the Visualizer. A small part of the simulated execution is found in figure 6. In the execution flow graph we see that no threads are actually running in parallel. We also see that all threads are being blocked by a wait on a mutex, the arrow facing downwards. By clicking with the mouse on the arrows, we reach the conclusion that it is the same mutex causing the blocking for all threads. The mutex is the one that we use to lock the insertion and fetching.

When we have pinpointed the performance bottleneck we have to find a solution to our problem. One solution is to have 100 buffers with their own mutex locks. We keep a mutex for the whole buffer system to lock the small amount of time to check which buffer to insert the item in. We also have different mutexes for inserting and fetching.

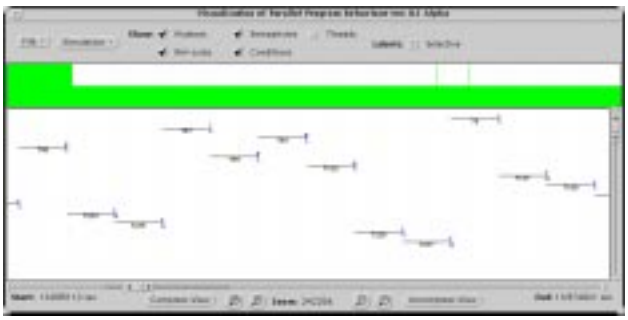


Figure 6: Parts of the execution of the initial program.

After making a new log of the improved program, we find that the program runs 7.75 times faster when using the simulated eight processor machine. A validation gives the speed-up of 7.90 on a real multiprocessor, thus the error in the prediction is only 1.9%. A picture of the simulated execution is found in figure 7. In the parallelism graph we can see that a larger number of thread are runnable but has no processor to run on. This is indicated by the high red part of the graph, and the constant low green part.

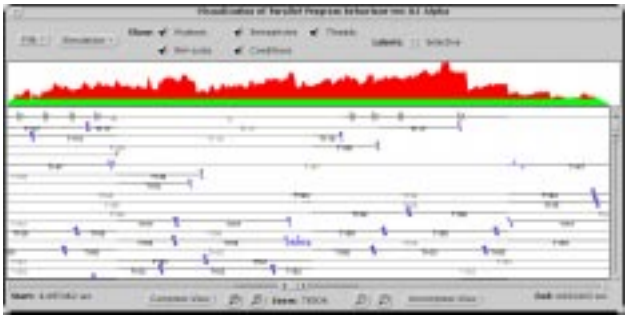


Figure 7: Simulated execution the improved program.

6. Discussion

Our approach is based on the assumption that the behaviour of the multithreaded program is (more or less) independent of the scheduling policy and the number of processors used for executing the program. Using the trace log file in a deterministic way when simulating and visualizing the execution may cause some problem. Some of them will be discussed below.

Conditions variables [17] are hard to simulate, since their behaviour depends on the value of an ordinary variable, which can not be traced by the Recorder. However, since it is common to use condition variables when implementing barriers, the simulator is designed to model the behaviour of a barrier as accurate as possible. The problem concerns the last thread that arrives at the barrier in the monitored execution. In the simulation that thread may be scheduled in a way that makes the thread arrive at the barrier before some other threads do. If the number of threads released during the recorded execution are less than those in the log file, the `cond_broadcast` will block the calling thread waiting for the correct number of threads to arrive at the barrier. Thus, the last thread arriving at the barrier releases all the waiting threads.

The primitive `thr_join` [17], waits until another (specific) thread has exited. It is possible to pass a wildcard to `thr_join`, meaning that the thread will wait for any thread the exit, which may not be the one that exited in the log file. Finally, the simulator does not consider the overhead for LWP context switches on a multiprocessor.

The Recorder can only be used when running one single LWP since the Recorder can not detect when an LWP's time slice is over and another LWP starts to execute. This makes it impossible to run a program with several threads where one thread executes in a tight loop during the whole execution since the loop will be the only one executing. Also having a thread spinning on a (ordinary, volatile) variable will cause a livelock for the same reason. Further reading on thread synchronization and scheduling can be found in [17]. Finally, our technique does not model I/O, and is therefore applicable only to CPU-intensive applications. We are currently working on solving this problem.

In the current implementation VPPB supports Solaris 2.X threads. However, the tool can easily be adjusted to support, e.g., POSIX threads [17] with only small modifications of the probes in the Recorder.

We have chosen not to use the recording facilities found in TNF [4], although the technique is similar to our Recorder. The main reason is that TNF uses a circular buffer to store the recorded information and thus information may be overwritten if the buffer is too small.

In [16] and [20] the authors stress the following issues: selective representation; integration between development time and run-time information; high-level and automated performance debugger; automated instrumentation of parallel programs; low overhead in monitoring program execution; and graphs and indices to expose performance bottlenecks. As mentioned throughout this paper all these requirements are met.

In parallel program development today there exists a number of tools, most ones with graphical (and even aural) displays. VPPB offers two different graphs, the execution flow graph and the parallelism graph. The execution flow graph is a commonly used graph, e.g., [5, 16]. We expect the parallelism graph to be very useful for detecting performance bottlenecks in large applications. A huge amount of graphs may cause more confusion than clarity of the performance problem as stated in [2]. Some other tools use statistical graphs [5]. The main problem with statistical graphs and data is that they often give only average values which are often useless since it is hard to identify when and where the program generated the statistics.

7. Conclusion

In this paper we describe a tool called VPPB, Visualization of Parallel Program Behaviour. The main goals are to predict the speed-up of a multithreaded application and to visualize the application's multiprocessor behaviour for the developer. The target programs are written in C or C++ and run on the Solaris 2.X operating system, an environment commonly used in both industry and academia.

Our approach relies on a monitored execution of the multithreaded application on a uni-processor. During that execution a log file is created containing all calls that the application made to the thread library. Then, the multiprocessor execution is simulated according to user supplied scheduling and hardware parameters. The result of the simulation is visualized graphically. The developer can then inspect the behaviour of the application as if it had been run on a multiprocessor without even having one.

The visualization of the execution is based on an execution flow graph along with some numeric data, a concept that previously has been shown to be successful [9]. The execution flow graph can be scrolled and zoomed, both in fixed steps and according to a specific time interval. A second graph shows the number of threads running at the moment, as well as the number of threads that are runnable but not running, i.e., the amount of available parallelism. VPPB gives the developer information enough to pin point the bottlenecks and correct them.

The tool also has a unique stepping facility, which gives the user of the tool a possibility to follow all operations on, e.g., a specific semaphore. Further, the tool supports an automatic mapping between an event in the execution flow graph and the source code line causing the event. It also starts an editor with the correct code line high-lighted.

We have validated the predicted speed-up using five benchmarks from the SPLASH-2 suite [19] and a multiprocessor with 8 processors. The predictions were found to be very accurate; for four of the applications the error was less than 2% as compared to a real multiprocessor execution. For the fifth application the error in the predicted speed-up was 6%. The intrusion made by the probes that collect the event log is very low; the execution time of the monitored application is prolonged by at most 3%.

References

[1] H. Chen, B. Shirazi, J. Yeh, H. Youn, and S. Thrane, "A Visualization Tool for Display and Interpretation of SISAL Programs," *Proc. ISCA Int'l Conf. on Parallel and Distributed Computing Systems*, pp. 734-739, Oct. 1994.

[2] J. K. Hollingsworth and B. P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems," *Proc. Int'l Conf. on Supercomputing*, pp. 185-194, Jul. 1993.

[3] A. Hondroudakis, "Performance Analysis Tools for Parallel Programs," Edinburgh Parallel Computer Centre, The University of Edinburgh, Jul. 1995.

[4] S. Kleiman, D. Shah, and B. Smaalders, "Programming with threads," Prentice Hall, 1996, ISBN 0-13-172389-8.

[5] E. Kraemer and J. Stasko, "The Visualization of Parallel Systems: An Overview," *J. of Parallel and Distributed Computing*, Vol. 18, pp. 105-117, 1993

[6] S. Lei and K. Zhang, "Performance Visualization of Message Passing Programs Using Relational Approach," *Proc. ISCA Int'l Conf. on Parallel and Distributed Computing Systems*, pp. 740-745, Oct. 1994.

[7] L. Lundberg, "Multiprocessor Performance Evaluation of Billing Gateway Systems for Telecommunication Applications," *Proc. ISCA Int'l Conf. on Parallel and Distributed Computer Systems*, pp. 225-231, Sep. 1996.

[8] L. Lundberg and M. Roos, "Predicting the speed-up of multithreaded programs," *Proc. IEEE Conf. on High Performance Computing*, pp. 386-392, Dec. 1997.

[9] W. E. Nagel and A. Arnold, "Performance Visualization of Parallel Programs - The PARvis Environment," *Proc. 1994 Intel Supercomputer Users Group (ISUG) Conference*, pp. 24 - 31, May 1994.

[10] G. J. Nutt, A. J. Griff, J. E. Mankovich, and J. D. McWhirter, "Extensible Parallel Program Performance Visualization," *Proc. Mascots '95*, 1995.

[11] E. Papaefstathiou, D. J. Kerbyson, G. R. Nudd, and T. J. Atherton, "An Overview of the CHIP^{3}S Performance Prediction Toolset for Parallel Systems," *Proc. 8th ISCA Int'l Conf. on Parallel and Distributed Computing Systems*, pp. 527-533, 1995.

[12] V. Pillet, J. Laboarta, T. Cortes, and S. Girona, "PARAVER: A Tool to visualize and Analyse Parallel Code," University of Politencia, Catalonia, CEPBA/UPC Report No. RR-95/03, Feb. 1995.

[13] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks, "SunOS 5.0 Multithreaded Architecture," Sun Soft, Sun Microsystems Inc., Sep. 1991.

[14] S. R. Sarukkai and D. Gannon, "SIEVE: A Performance Debugging Environment for Parallel Programs," *J. of Parallel and Distributed Computing*, Vol. 18, pp. 147-168, 1993.

[15] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Multiprocessors," MIT Press, 1989.

[16] Z. Segall and L. Rudolph, "PIE: A Programming and Instrumentation Environment for Parallel Processing," *IEEE Software*, 2(6):22-37, Nov. 1985.

[17] SunSoft, "Solaris Multithreaded Programming Guide," Prentice Hall, 1995.

[18] S. Toledo, "PERFSIM: A Tool for Automatic Performance Analysis of Data-Parallel Fortran Programs," *Proc. 5th Symp. on the Frontiers of Massively Parallel Computation*, IEEE Computer Society Press, Feb. 1995.

[19] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Annual Int'l Symp. on Computer Architecture*, pp. 24-36, Jun. 22-24 1995.

[20] J. Yan, S. Sarukkai, and P. Mehra, "Performance Measurements, Visualization and Modelling of Parallel and Distributed Programs using the AIMS Toolkit," *Software-Practice and Experience*, 25(4):429-461, Apr. 1995.