A Performance Analysis Tool for CCC

Sheng-Chau Juang Nai-Wei Lin
Department of Computer Science and Information Engineering
National Chung Cheng University
Chiayi, Taiwan 621, R.O.C.
{jsc88,naiwei}@cs.ccu.edu.tw

Abstract

CCC is a high-level parallel programming language that supports both data and task parallelism. In CCC, data parallelism is specified in single-instruction-multiple-data model, while task parallelism is specified in multiple-instruction-multiple-data model. In CCC, task parallelism supports both message passing communication abstraction and shared variables synchronization abstraction. This paper describes the design and implementation of a performance analysis tool for CCC. This tool provides both numerical performance statistics and graphical performance visualization. The information provided by this tool can significantly help programmers understand the performance behaviors of parallel CCC programs.

1. Introduction

CCC is a high-level parallel programming language that supports both *data* and *task* parallelism. A CCC program consists of a collection of coordinated concurrent tasks. Data parallelism in CCC is specified in *single-instruction-multiple-data* (SIMD) model; In other words, data-parallel tasks are executed synchronously and perform the same operation on different data. Shared-memory abstraction is provided to support concurrent read, concurrent write, and reduction operations among data-parallel tasks. In contrast, task parallelism in CCC is specified in *multiple-instruction-multiple-data* (MIMD) model; In other words, task-parallel tasks are executed asynchronously and usually perform different operations on different data. Both message-passing communication abstraction and shared variables synchronization abstraction are provided to facilitate various interaction patterns among task-parallel tasks. These salient features make CCC cover most parallel programming paradigms.

The performance of parallel programs is much more difficult to analyze and understand than that of sequential programs. Both parallel programming models and parallel computer architectures have deep influences on the performance of parallel programs. In general, to achieve the peak performance of a parallel program, many iterations of performance tuning are necessary. Hence, it becomes very crucial to provide suitable performance analysis tools for the development of parallel programs.

Performance analysis includes performance instrumentation and performance visualization. Performance instrumentation involves statically inserting performance collecting instructions into the programs and dynamically measures and records the collected performance profiles. Performance visualization analyzes the performance profiles and presents the results using user-friendly tables and graphics.

The remainder of this paper is organized as follows. Section 2 gives a simple overview of the CCC programming language. Section 3 describes the part of the tool that performs performance instrumentation. Section 4 describes the part of the tool that performs performance visualization. Section 5 presents the instrumentation overheads of the tool. Section 6 briefly reviews related work. Finally, conclusions are given in Section 7.

2. The Programming Language CCC

The language CCC is designed as a small extension of the language C. This small extension of C mainly provides abstractions for specifying the concurrency, communication, and synchronization of parallel tasks.

We describe the data parallelism first. The concurrency abstraction for the data parallelism is specified by the definition of the *domain* construct and the invocation of the *data-parallel functions* defined in *domain* construct.

```
domain name [n] {
  variable_declarations;
  function_definitions;
 }
```

Each data-parallel function defined here represents a collection of *n* functions that will execute in parallel. An invocation of a data-parallel function will concurrently create *n* tasks. The synchronization abstraction is implicitly specified by the synchronous semantics of the SIMD model. The communication abstraction is implicitly specified by the concurrent read, concurrent write, and reduction operations on the variables defined within the *domain* construct. The *data distribution* specification can be given in the parameter specification of function definitions.

We now describe the task parallelism. The concurrency abstraction for the task parallelism is specified via the definition of task-parallel functions and the *parallel section* constructs. A task-parallel function

```
task function_name(parameters);
```

is declared by putting the keyword task before its function definition. There are two forms of parallel sections. The first one is used to concurrently invoke a group of task-parallel functions.

```
par {
  func_1(…);
  func_2(…);
    …
  func_n(…);
}
```

This example will execute the *n* task-parallel functions *func_1*, *func_2*, …, and *func_n* as *n* tasks in parallel. Parallel sections are executed in fork-join form, that is, the parallel section exits only when all created tasks exit. The second one is used to concurrently invoke multiple instances of a group of task-parallel functions.

```
parfor (init_expr; exit_expr; step_expr) {
  func_1;
  func_2;
    …
  func_n;
}
```

This example will execute multiple instances of the *n* task-parallel functions *func_1*, *func_2*, …, and *func_n* as tasks in parallel. The semantics for the control expressions *init_expr*; *exit_expr*; and *step_expr* is the same as that in the *for* construct in C.

The communication abstraction for the task parallelism is specified via *typed channels* (or *typed asynchronous message queues*). There are four types of channels: *pipes*, *spliters*, *mergers*, and *multiplexers*. Pipes are for one-to-one communication. These are the most basic channels. The other three types of channels can be implemented using pipes. Spliters are for one-to-many communication. These channels are useful for producer-consumer applications with single producers. Mergers are for many-to-one communication. These channels are useful for client-server applications with a single server. Multiplexers are many-to-many communication. These channels are useful for producer-consumer applications with multiple producers or client-server applications with multiple servers.

The messages in the channels can be accessed via the following two functions

```
msg = receive(channel);
send(channel, msg);
```

Channels provide a simple abstraction for implementing communication in message-passing programming model.

The synchronization abstraction for the task parallelism is specified via *monitors*.

```
monitor name {
  variable_declarations;
  condition_variable_declarations;
  function_definitions;
}
```

Monitors are structured and efficient constructs for implementing both the *mutual exclusion* and *condition synchronization* in shared-variable programming model. The functions defined in monitors are mutually exclusive by default. A read keyword can be put before a function definition if that function only reads the variables in the monitors. Multiple read functions can be invoked concurrently.

Three functions *wait*(*cond*), *signal*(*cond*), and *signalall*(*cond*) are provided to manipulate *condition variables*. The function that calls the signal function will continue to execute after signaling. Nested monitor calls are allowed in the language. A task releases monitor exclusion when it makes a nested monitor calls, and it needs to reacquire monitor exclusion when it returns from the call.

3. Performance Instrumentation

This section describes the part of the tool that performs performance instrumentation and outputs a performance file. In the implementation of the CCC compiler (on 4-x86 SMP running Solaris) mentioned in this paper, the tasks in CCC are implemented as threads.

To support performance instrumentation for CCC, the tool traces the timing information for the events listed in Table 1.

The events 11~15 are events for thread (or task) management. The events 31~34 are events for communication via channels among task-parallel threads. The events 41~47 are events for synchronization via monitors among task-parallel threads. The events 51~52 are events for barrier synchronization in data-parallel threads.

Event Name	Event Identifier
ThreadCreate	11
ThreadEntry	12
ThreadExit	13
JoinBegin	14
JoinEnd	15
SendBegin	31
SendEnd	32
ReceiveBegin	33
ReceiveEnd	34
LockBegin	41
LockEnd	42
Unlock	43
WaitBegin	44
WaitEnd	45
Signal	46
SignalAll	47
BarrierBegin	51
BarrierEnd	52

Table 1：Event names and event identifiers.

The tool uses the Solaris function clock_gettime to directly access hardware clock to provide timimng information with precision upto nanoseconds (10^{-9} seconds).

The timing information for these events are traced and recorded using the following function

```
void event_trace(int eid, event_t info);
```

where eid is the event identifier and info contains the timing and event-specific information.

To facilitate performance analysis, in addition to timing information, we also need to record event-specific information for each event. For example, for ThreadCreate event, we also need to record the identifier of the created thread, the type (task-parallel or data-parallel) of the thread, and the identifier of the parallel section in which the thread is created. For SendBegin, SendEnd, ReceiceBegin, and ReceiveEnd events, we also need to record the name of the channel on which these events occur.

When the CCC compiler is invoked with trace flag on, the CCC compiler will automatically insert the calls to functions clock_gettime and event_trace and the declarations and initializations of associated data structures at the appropriate points of the generated target program. At the end of the target program, it will output the traced performance profiles to a file, called performance file.

4. Performance Visualization

This section describes the part of the tool that inputs a performance file and performs performance visualization. This part is implemented using Qt, a multi-platform C++ GUI library [1]. It analyzes instrumented performance profiles and displays both numerical performance statistics as tables and performance visualization as graphics.

It displays performance information in five fashions: the simulator, the thread explorer, the channel explorer, the monitor explorer, and the barrier explorer. The simulator presents the integrated performance behaviors of the program. Others present the performance behaviors of the program related to a specific mechanism of the parallelism.

The simulator displays the original performance file in a human-readable form and plays an animation to simulate the execution of the program. A screenshot of the simulator is shown in Figure 1. It includes two windows. The left window displays the original performance file. The right window simulates the execution of the program by playing an animation to chronologically show the occurrences of all the traced performance events. It uses different colors to mark different events. The animation can be played continuously by varying speeds and can be played stepwise event-by-event.

The thread explorer extracts and displays the performance information related to thread management. A screenshot of the thread explorer is shown in Figure 2. It includes two windows. The left window displays the parent-child relationships among threads. These relationships are represented as a tree. Parallel sections are differently marked in order to distinguish task-parallel and data-parallel threads.

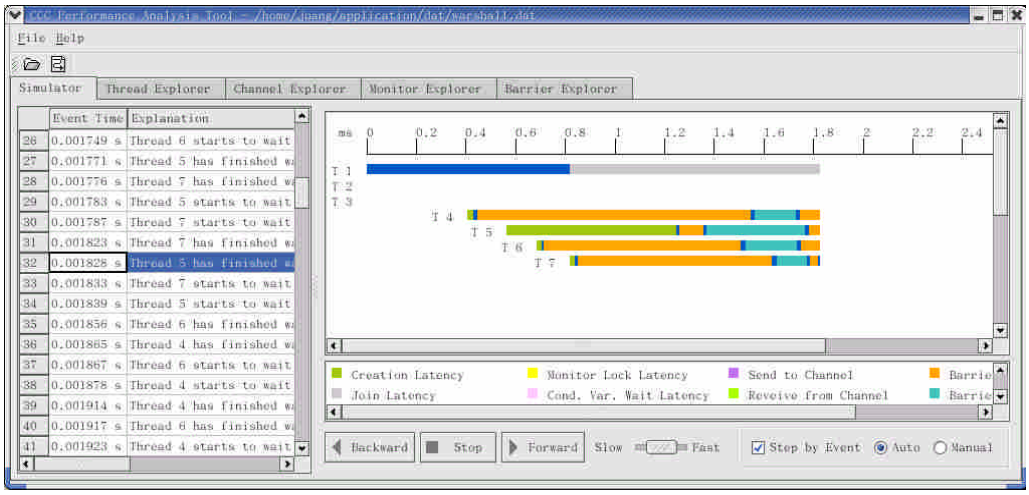

Figure 1. The simulator of the performance analysis tool.

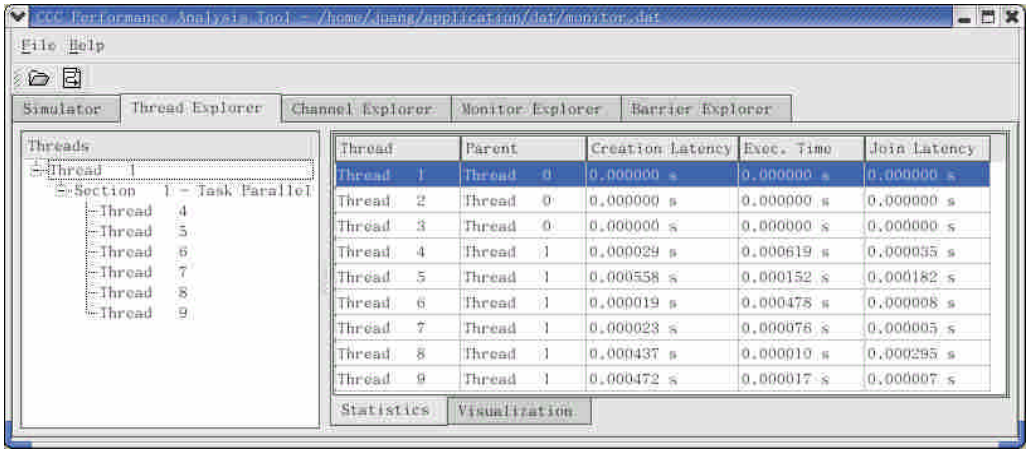

Figure 2. The thread explorer of the performance analysis tool – 1.

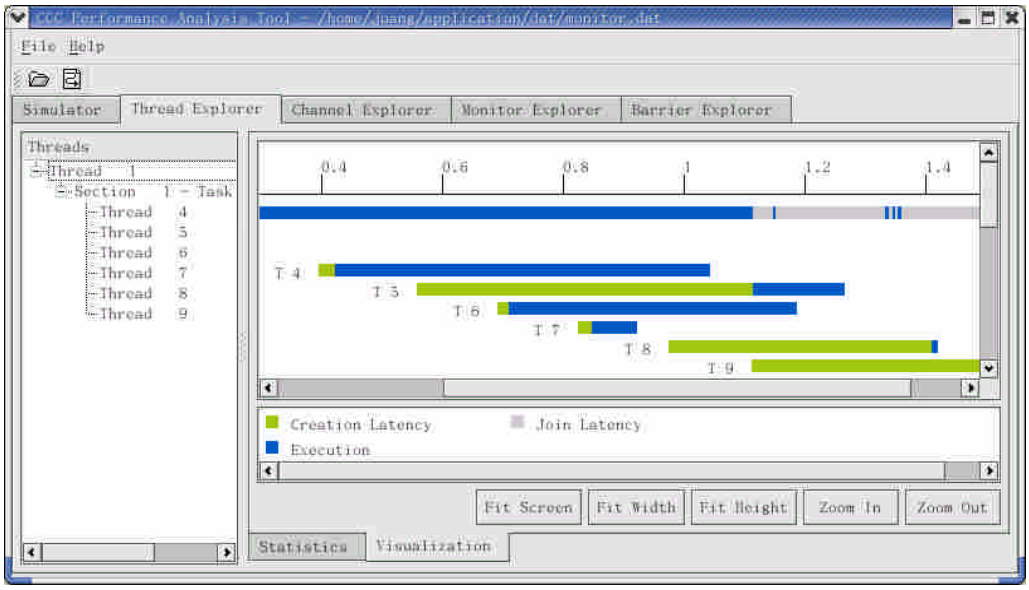

Figure 3. The thread explorer of the performance analysis tool – 2.

The right window contains two pages. The first page displays the numerical performance statistics as a table. A screenshot of the first page is shown in Figure 2. For each thread, the table includes the thread identifier of its parent thread, the *creation latency* (the time between the requesting for a thread creation by the parent thread and the starting of execution by the thread), the *execution time* (the time between the starting of execution by the thread and the exiting of execution and the requesting of a join by the thread), and the *join latency* (the time between the requesting of a join by the thread and the complete finishing of the join).

The second page displays graphical performance visualization. A screenshot of the second page is shown in Figure 3. It uses different colors to mark different events in the thread management. It also contains the function to zoom in or zoom out the graphics.

The channel explorer extracts and displays the performance information related to thread communication. A screenshot of the channel explorer is shown in Figure 4. It includes two windows. The left window displays the sender-receiver relationships among threads for each channel. These relationships are represented as a tree.

The right window contains two pages. The first page displays the numerical performance statistics as a table. A screenshot of the first page is shown in Figure 4. For each thread, the table includes the *name* of a channel, the *role* (sender or receiver) of this thread on this channel, the *times of communication* (the times this thread sends to (or receives from) this channel), the *total latency* (the total time spent for sending messages to (or receiving messages from) this channel), the *average latency* (the average time spent for sending a message to (or receiving a message from) this channel).

The second page displays graphical performance visualization. A screenshot of the second page is shown in Figure 5. It uses different colors to mark different events in the communication. It also contains the function to zoom in or zoom out the graphics.

The monitor explorer extracts and displays the performance information related to thread synchronization. A screenshot of the monitor explorer is shown in Figure 6. It includes two windows. The left window displays the threads that participate in the synchronization of each monitor and the wait-signal relationships among threads for each condition variable in that monitor. These relationships are represented as a tree.

The right window contains two pages. The first page displays the numerical performance statistics as a table. A snapshot of the first page is shown in Figure 6. For each thread, the table includes the *name* of a monitor, the *times of lock* (the times this thread requests to enter this monitor), the *total lock latency* (the total time spent for waiting to enter this monitor), the *average lock latency* (the average time spent for waiting to enter this monitor), .the *times of wait* (the times this thread waits for a condition variable in this monitor), the *total wait latency* (the total time spent for waiting for a condition variable), the *average lock latency* (the average time spent for waiting for a condition variable), and the *times of signal* (the times this thread signals a condition variable in this monitor).

The second page displays graphical performance visualization. A screenshot of the second page is shown in Figure 7. It uses different colors to mark different events in the synchronization. It also contains the function to zoom in or zoom out the graphics.

The barrier explorer extracts and displays the performance information related to synchronization among data-parallel threads. A screenshot of the barrier explorer is shown in Figure 8. It includes two windows. The left window displays the threads that participate in the synchronization.

The right window contains two pages. The first page displays the numerical performance statistics as a table. A snapshot of the first page is shown in Figure 8. For each thread, the table includes the *name* of a data-parallel function, the *times of barriers* (the times the thread performs barrier synchronization), the *total barrier latency* (the total time spent for waiting the completion of all barrier synchronizations), the *average lock latency* (the average time spent for waiting the completion of a barrier synchronization).

The second page displays graphical performance visualization. A screenshot of the second page is shown in Figure 9. It uses two different colors to distinguish two adjacent barrier synchronizations. It also contains the function to zoom in or zoom out the graphics.

The information provided by this performance analysis and visualization tool can significantly help programmers understand the performance behaviors of parallel CCC programs.

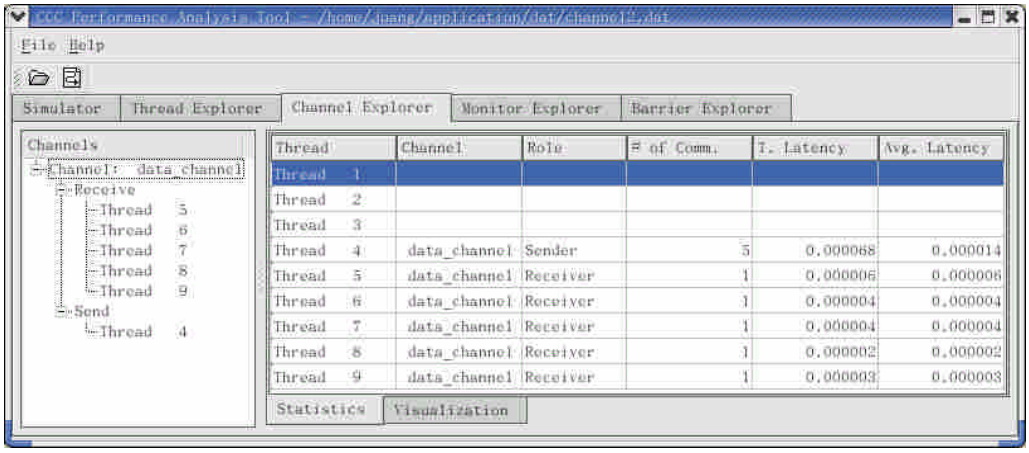

Figure 4. The channel explorer of the performance analysis tool – 1.

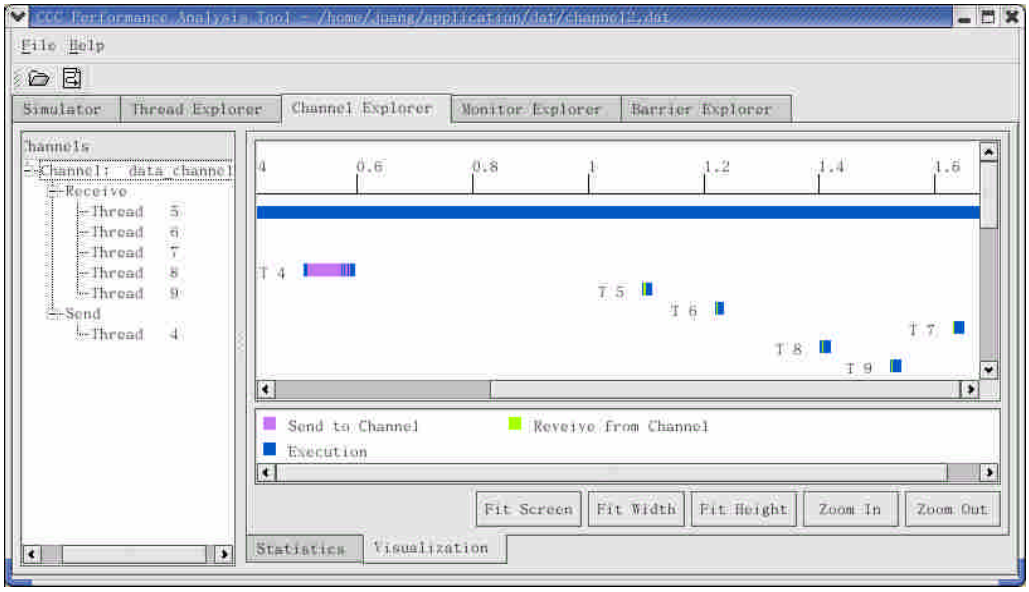

Figure 5. The channel explorer of the performance analysis tool – 2.

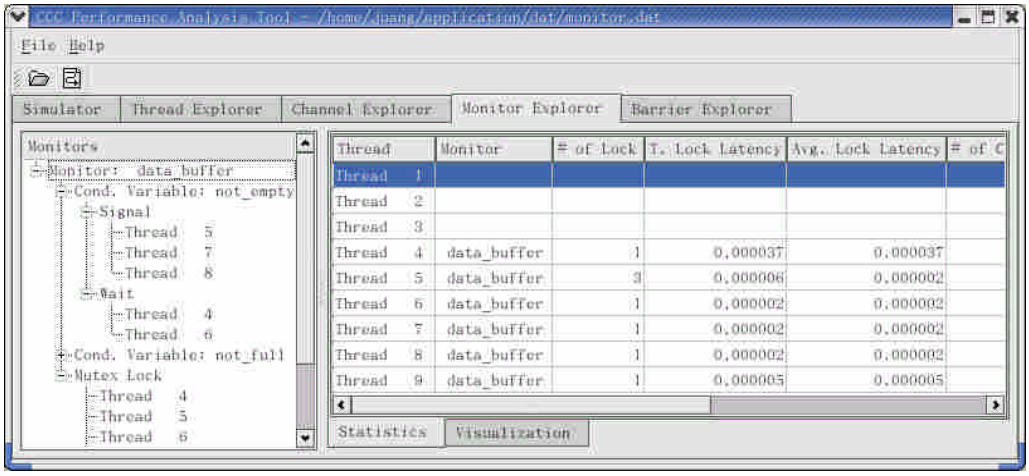

Figure 6. The monitor explorer of the performance analysis tool – 1.

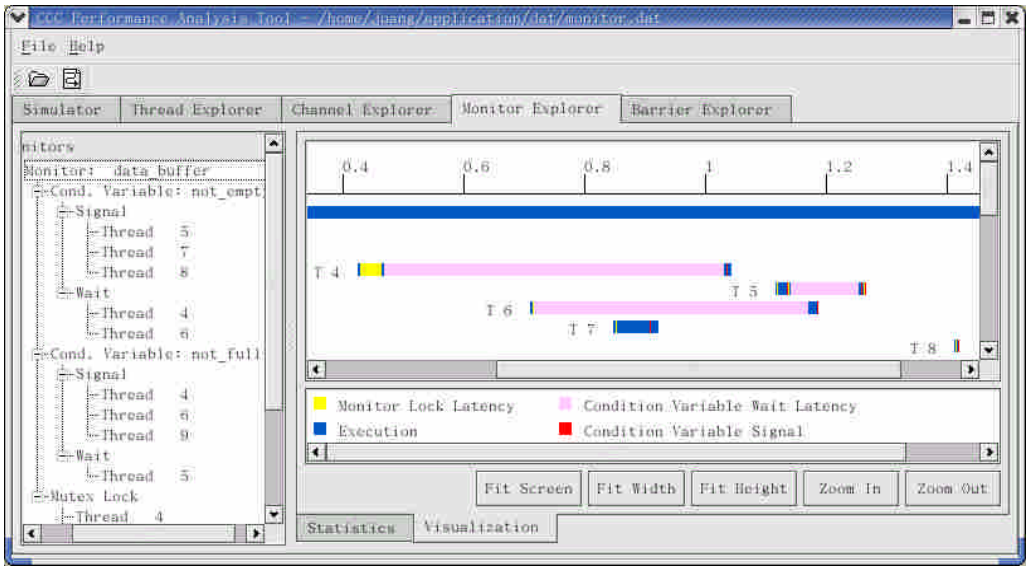

Figure 7. The monitor explorer of the performance analysis tool – 2.

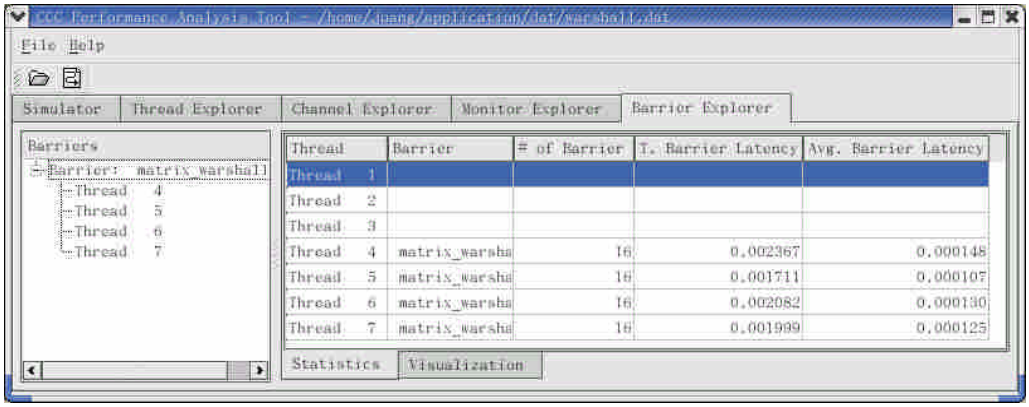

Figure 8. The barrier explorer of the performance analysis tool – 1.

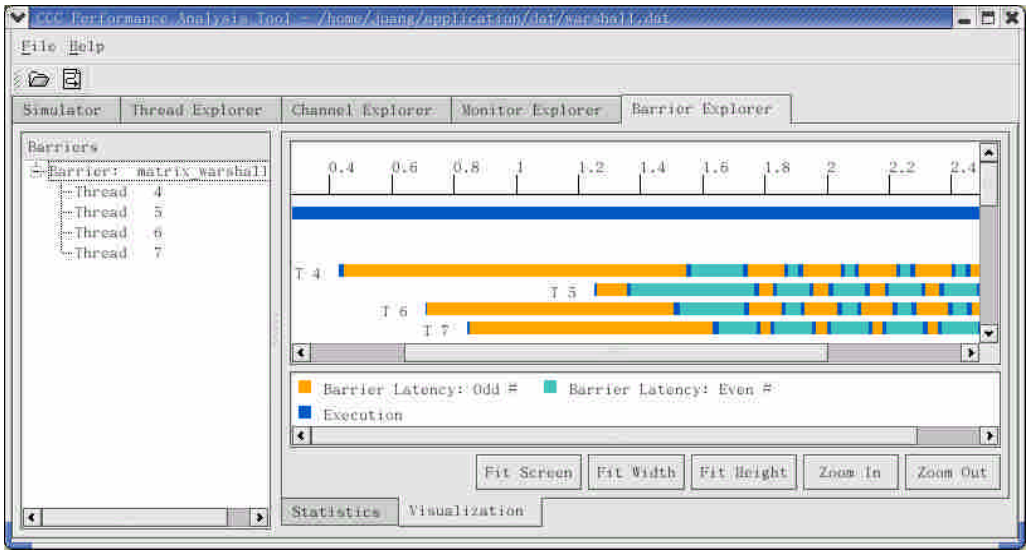

Figure 9. The barrier explorer of the performance analysis tool – 2.

5. Instrumentation Overheads

This section presents the preliminary measurements of the instrumentation overheads of the tool. We use a suit of benchmarks that contains three programs (Radar, Airshed, and Stereo) using task parallelism and three programs (MatrixMul, Warshall, and Gaussian) using data parallelism. The measurements are performed on a symmetric multiprocessor with 4 X86 processors and running Solaris operating system. The results are shown in Table 2, Figures 10 and 11.

Program	Original	Instrument (# of event)	Overhead	
Radar	0.499989	0.502118 (40)	0.002129	0.43%
Airshed	1.406167	1.413430 (800)	0.007263	0.52%
Stereo	0.453641	0.459904 (10363)	0.006263	1.38%
MatrixMul	0.001523	0.001575 (20)	0.000052	3.41%
Warshall	0.002933	0.003069 (148)	0.000136	4.62%
Gaussian	0.004540	0.004782 (268)	0.000242	5.34%

Time Unit: sec

Table 2. The instrumentation overheads.

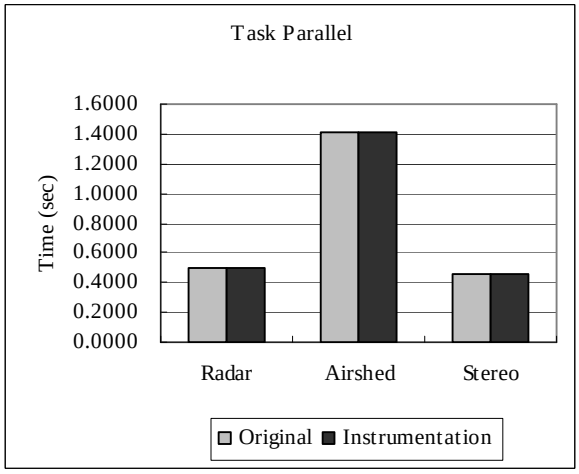

Figure 10. The instrumentation overheads of task-parallel benchmark programs.

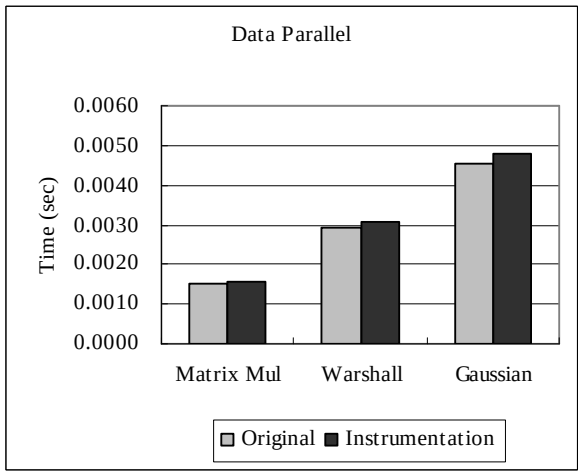

Figure 11. The instrumentation overheads of data-parallel benchmark programs.

The measured instrumentation overheads for this suit of benchmark are at worst 5.34% and average 2.61%. Since the three data-parallel programs run for a vary short time, the overheads are more observable. We believe that the overheads should be acceptable for programs that run for a reasonably long time.

6. Related Work

The design and implementation of performance analysis tools for parallel programs is an active research area. Most performance analysis tools developed so far support specific programming models or computer architectures. Among the tools developed, most notable results are AIMS, developed at NASA Ames Research Center [4], Paradyn, developed at the University of Wisconsin [3], and SvPable, developed at the University of Illinois [2].

The tool AIMS supports the performance instrumentation and visualization only for message passing programs. A special feature of the tool Paradyn is that its performance instrumentation and visualization are performed during runtime. Hence, it is possible to dynamically control the instrumentation according to the status of program's performance. The tool SvPable provides language and architecture independent performance instrumentation and visualization based on a general meta-meta-format (SDDF) of performance data.

7. Conclusions

CCC is a high-level parallel programming language that supports both data and task parallelism. In CCC,

data parallelism is specified in single-instruction-multiple-data model, while task parallelism is specified in multiple-instruction-multiple-data model. In CCC, task parallelism supports both message passing communication abstraction and shared variables synchronization abstraction. This paper has described the design and implementation of a performance analysis tool for CCC. This tool provides both numerical performance statistics and graphical performance visualization. The information provided by this tool can significantly help programmers understand the performance behaviors of parallel CCC programs.

Acknowledgements

This work was supported in part by the National Science Council of R.O.C. under grant number NSC-91-2213-E-194-007.

References

[1] Kalle Dalheimer, *Programming With Qt*, 2nd edition, O'Reilly & Associates, Inc., Jan 2002.

[2] L. De Rose and D. A. Reed. SvPable: A Multi-Language Performance Analysis System. *Proceedings of 1999 International Conference on Parallel Processing*, September, pp. 311-318.

[3] B. P. Miller, M. D. Callaghan, J. M., Cargille, J. K., Hollingsworth, B. R., Irvin, K. L., Karavanic, K., Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer* 28, 11 (November 1995), 37-46.

[4] J. C. Yan, S. R. Sarukkai, and P. Mehra. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs Using the AIMS Toolkit. *Software Practice & Experience* 25, 4 (April 1995), 429-461.