Chapter 11
Temanejo: Debugging of Thread-Based Task-Parallel Programs in StarSS

Rainer Keller, Steffen Brinkmann, José Gracia, and Christoph Niethammer

Abstract To make use of manycore processors and even accelerators, several parallel programming paradigms exist, such as OpenMP, CAPS HMPP and the StarSs programming model. All of these programming models provide the means for programmers to express parallelism in the source code, identifying tasks and for all but OpenMP the dependency between those, allowing the compiler and the runtime to schedule tasks onto multiple concurrent executing entities, like threads in a many-core systems. While the programmer may have a good overview of which parts of the code may be run independently as separate tasks on a fine granular level, the overall execution behavior may not be obvious at first. This paper describes the usability features of the newly developed Temanejo debugger.[1]

11.1 Introduction

Parallel programming adds another level of complexity in every respect, especially when debugging the application. Different parallel programming models have different perspectives in that regard – parallel programming models based on processes allow having debuggers attach to each individual process. For programming models that are based on threads, finding bugs is a more difficult feat due nature of threads. In Unix environments, threaded programs share resources such as signals, file

[1]This work was supported by the European Community's Seventh Framework Programme [FP7-INFRASTRUCTURES-2010-2] as project grant "Toward Exaflop Applications" (TEXT) under grant agreement number 261580.

R. Keller (✉) · S. Brinkmann · J. Gracia · C. Niethammer
HLRS, Nobelstrasse 19, Stuttgart, Germany
e-mail: keller@hlrs.de; brinkmann@hlrs.de; gracia@hlrs.de; niethammer@hlrs.de

H. Brunst et al. (eds.), *Tools for High Performance Computing 2011*,
DOI 10.1007/978-3-642-31476-6_11, © Springer-Verlag Berlin Heidelberg 2012

```
#include <stdio.h>

#pragma css task input(a) output(out)
void fib(int a, int *out) {
    int tmp1, tmp2;
    if (a <= 1)
        *out = 1;
    else {
        fib (a-1, &tmp1);
        fib (a-2, &tmp2);
        *out = tmp1 + tmp2;
    }
}

int main(void)
{
    int var = 50;
    int result ;
#pragma css start
    fib (var-1, &result);
#pragma css finish
    printf ("fib(%d) = %d\n", var, result);
    return 0;
}
```

Listing 11.1 StarSs example of parallel, recursive Fibonacci computation

descriptors, and most importantly the memory address space. In order not to create race conditions, multiple threads need to synchronize access to shared resources.

The StarSs programming model [5] allows to specify tasks and their inter-dependencies by annotating the sequential code using pragma statements (in C) and comments (Fortran). Code 25 shows an example written in C, with the StarSs statements being followed by the `css` pragmas.[2]

While compilers not supporting StarSs ignore the pragmas in Code 25, BSC's SmpSs compiler generates wrapper functions out of the specified tasks, with the StarSs runtime scheduling the tasks to threads calling these wrapper functions, which then invoke the underlying code generated by the host-compiler. The threads are beingcreated upon the entering of the `start` section. Currently, the C, C++ and Fortran77 and Fortran90 languages are supported – of course, due to it's thread-parallel nature, several unsafe practices may not be used in the taskified code, such as C's long-jumps, unprotected access to shared resources (mainly global memory), C++ exceptions, or in Fortran unprotected access to common blocks or passing temporary arrays. As with OpenMP the details of the underlying Thread-model (Posix Threads, Solaris Threads) [4] as well as the semantics of the Memory Model [1] are left to the compiler. Similar to OpenMP, the StarSs programming

[2] The acronym `css` is due to BSC first developing the SuperScalar compilers for the IBM Cell.

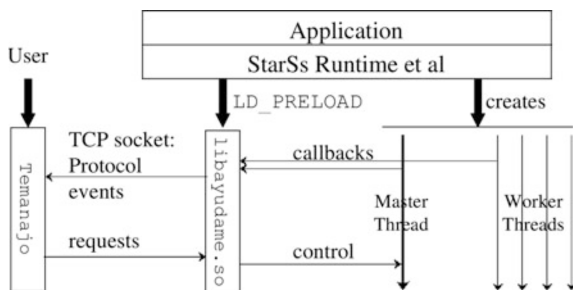

Fig. 11.1 Interaction between StarSs runtime, Ayudame and Temanejo

model allows incremental parallelization of the application, focusing first on large, independent tasks, leaving the source code's structure mostly intact – even more so, as soon as future compiler releases allow taskifying code sequences within functions.

The StarSs' runtime handles the dependency tracking using the parameter's address being passed to the task. As soon as all input dependencies are being met, the task may be scheduled to run. Compared to the classical OpenMP approach focusing on parallelising large loops, StarSs allows a much more dynamic execution, possibly making better use of resources. As will be shown below, the task-graph is a directed acyclic graph, which is dynamically generated at runtime. This graph may become very large and is executed non-deterministically.

However, this non-determinism may create problems when debugging the application. In this paper, we will revisit the Temanejo-debugger [2], developed at the High Performance Computing Center Stuttgart (HLRS) within the frame of the TEXT project.

11.2 Implementation

The graphical debugger Temanejo[3] is logically separated from the library that interacts with the runtime, the so-called Ayudame-library.[4] This separation allows to attach to SMPSs instances running on compute nodes, with limited capabilities, having the graphical display of tasks on the programmer's workstation. Ayudame provides multiple hooks for the runtime, e.g. to retrieve the number and names of tasks in the parallel section and API to steer the runtime. Figure 11.1 shows the interaction of the library Ayudame, here being LD_PRELOADed to the run-time, in order to satisfy the callbacks into the hooks. The Master Thread here stands for

[3] Spanish for 'I Handle You'.

[4] Spanish for 'Help me'.

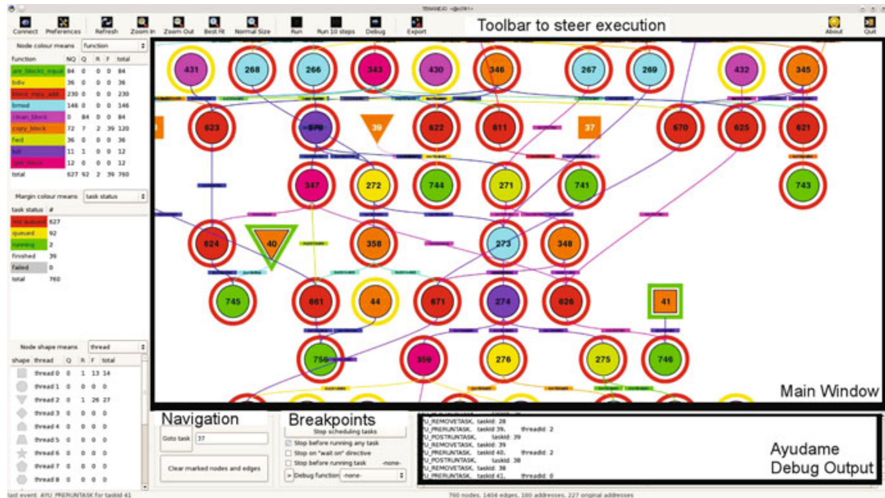

Fig. 11.2 Screenshot of the Temanejo GUI zoomed in, while debugging a LU-factorization example

SMPSs' runtime scheduling the tasks. The data exchanged between the Ayudame library and the Temanejo debugger is kept lean as a binary protocol, in principle allowing to attach and detach from the debugged process. The communication is currently done using TCP. If the user issues commands from the Temanejo GUI, it is being forward to the Ayudame library as a requests, which then relays it to the runtime. All the events on the run-time site are being collected and buffered to be sent in chunks via a separate thread decoupling the runtime and the GUI. On the GUI's side again data is received in a separate thread analyzing the event changes and updating the graphical representation.

11.3 Debugging Capabilities

Debugging implies seeing, understanding and intercepting the execution of an application. The Temanejo debugger allows programmers to visualize even large task graphs and efficiently interact with the run-time to analyze large applications.

Figure 11.2 presents a screenshot (edited for better printing) of Temanejo, here with full detail and heavily zoomed to enlarge the features of this graph.

Each task is visualized by a node in the main window of Temanejo. The user may zoom and drag through the main window and click on single nodes, aka the tasks, offering further functionality as described later. The node's color and shape by default encode the task's function and when the task is scheduled and running, the actual thread number it is running on. Moreover, the screenshot in Fig. 11.2 additionally shows the dependencies including the address of the dependency.

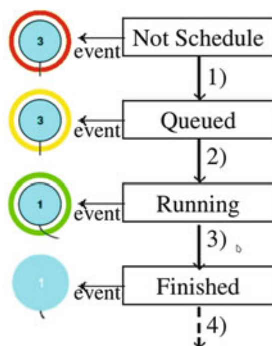

Fig. 11.3 Symbols representing the states within the life-time of a task, each change in state will generate an event in Ayudame

This nicely visualizes the applications capabilities in terms of parallel execution, e.g. narrow graphs represent situations in the execution where little opportunities for parallelism exist.

The node's border or margin encodes the tasks state Fig. 11.3 shows Temanejo's representation of the task state life-cycle. Tasks change their state upon the following conditions and emit the following event (numbering corresponding to Fig. 11.3):

1. If all input dependencies are fulfilled, the task may be enqueued.
2. If the task is being dequeued from the runable queue it is scheduled for execution to a thread.
3. If the task is finished running, another event is being generated to update the display. In reality it is being dropped from all queues.
4. At last, the tasks depending on the output of the finished tasks are being notified.

With the left-most pane of the display, the visualization attributes of the main window may be changed. Switching away from the default of the task's function, one may encode the executing thread (instead of it's shape), to distinctly visualize the likelihood of threads executing "close" tasks (the SMPSs runtime assumes dependent tasks benefit from data reusing the cache). Furthermore, one may set the node's color as it's execution time, showing specifically long-running tasks after the execution. Within large numbers of tasks the graph may get very bloated. Using the navigation controls, one may jump to a specific task and within the left-most pane change the node's color to encode the distance to the selected node allowing visual control of which subtree of the graph will remain dominant consumer of CPU time until the node in question will be executed.

In the toolbar, the application may be steered. By default, after attaching, the runtime is stopped from executing further tasks. The user may single-step through the application, receiving control over the SMPSs runtime after each schedule. Furthermore one may execute ten tasks at a time (or a setable number of steps) and regain control afterwards with a possibly drastically changed graph (due to new, added nodes, or lots of dependencies being cleared and many more tasks in runnable state). A nice feature to interact with the application developer is the export

functionality, allowing single screenshots of the current graph and even movie generation of evolving task graphs.

The Ayudame Debug output in the lower right corner shows all the events emitted by the Ayudame library in human readable text, allowing easy searching for task numbers if executing multiple tasks at a time.

Most importantly a debugger will allow steering the application. Since Temanejo is not a full-fledged low-level debugger, the underlying system-debugger, here gdb, is being integrated. One may at any time attach to a task with the gdb, opening another terminal and from there use the usual techniques to query memory's state, reset variables and single-step through the binary. Temanejo helps in that regard, that the actual function and not the wrapper function is being attached to.

11.4 Related Work

Debugging multiple threads with it's many types of possible faults allows for a multitude of solutions. The standard tools required for debugging are traditional debuggers such as the GNU debugger (`gdb`); it allows switching between multiple threads or issuing specific commands using the `thread` keyword. However, to `gdb` the tasklets being created are functions of different name (including line number information).

Temanejo aims at a higher level visualizing and steering the task-graph's execution. Still, it relys on the underlying debugger, like `gdb` to attach to the generated functions to debug the actual source code.

An important class of faults are unprotected access to memory shared between threads. As of now, this class of bugs may not be found with our tool. However, using techniques similar to the valgrind- and pin-based tools developed in [3], accesses to global memory areas registered prior to starting the tasks could be tracked and checked for unordered access. There has been recent work by BSC to detect memory overrun and access errors of parameters passed to tasklets using the `valgrind`-tool.

11.5 Outlook

The tool has proven useful to programmers of StarSs-parallelized applications. We aim to extend the tool in other scenarios, such as debugging OpenMP tasking constructs, even though much of the visual capabilities for dependency tracking is lost. There are endeavours to abstract the Ayudame library in order to fit into a Pthread-based custom runtime system in order to visualize a large-scale numerical simulation. Moreover we hope to make good use of extended three-dimensional visualization, where one attribute leads to different panes in the third dimension, allowing spatial separation for better control of large graphs.

Acknowledgements We would like to thank Barcelona Supercomputing Center (BSC) and the partners in the TEXT project for their support in the development of their compiler suite.

References

1. Adve, S.V., Boehm, H.J.: Memory models: a case for rethinking parallel languages and hardware. Commun. ACM **53**(8), 90–101 (2010). doi: 10.1145/1787234.1787255
2. Brinkmann, S., Gracia, J., Niethammer, C., Keller, R.: Temanejo – a debugger for task based parallel programming models. In: Proceedings of ParCo'11, Gent, vol. abs/1112.4604, Gent, Belgium (2011)
3. Fan, S., Keller, R., Resch, M.: Advanced memory checking frameworks for MPI parallel applications in Open MPI. In: Tools for High Performance Computing. Springer, Berlin (2011). Submitted for publication
4. Northrup, C.J.: Programming with UNIX Threads. Wiley, New York (1996)
5. Perez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. IEEE Int'l Conference on Cluster Computing (Cluster 2008), Tsukuba, pp. 142–151 (2008)