DEBUGGING PARALLEL PROGRAMS IN THE BSP SUPERSTEP CONTEXT

Vasil P. Vasilev
Vasil.P.Vasilev@comlab.ox.ac.uk
Oxford Parallel, Oxford University Computing Laboratory
Oxford, England

This report contains results from research carried out as part of the *BSP Environment* project of the EPSRC PSTPA Programme (Grant K40765) in August/September 1996.

Abstract

The systematic use of the BSP model greatly simplifies the debugging of parallel programs because the separation of communication from synchronization, produces a superstep structure in which there are regular points of reference at which the state of all processors is consistent. This provides a much easier basis on which to investigate the execution of the program. In effect the superstep structure reduces the problem to one of sequential debugging. It depends on the particular system what kind of debuggers can be used. We present some general guidelines covering most of the machines. We then deal with particular systems and the debuggers available on them. It will be necessary to consult the debugger documentation for detailed information.

1. How does the BSP model help in debugging parallel programs?

The BSP model allows easy debugging even with normal sequential debuggers. However, parallel ones are to be preferred as they can control several processes at once. The superstep context gives three major benefits in comparison with any other existing model when debugging parallel programs.

1. The beginning/end of a superstep is a convenient place to put breakpoints. This is a point that all processes reach at the same time. More precisely, the breakpoint can be placed so that it comes when all the communications between processes, initiated in the preceeding superstep, are guaranteed to be complete. At that point, the whole computation has reached a common, global state.

2. If a process is stopped within a superstep, its stage of execution is known, and it is certain that all the other processes are in the same superstep because they cannot get beyond the barrier synchronisation at end of the superstep.

3. Stopping one process does not change the outcomes of the calculations in other processes. Each process operates asynchronously until superstep end and the semantics of BSP communications makes the order of events in satisfying the requests for data transfers immaterial.

2. In General:

The following is valid for a general machine. Note that for some machines and/or debuggers some of the commands described might not exist, be different, or have problems. However, it is believed that a good debugger should have most of the required properties and behaviour.

First, consider placing a breakpoint after bsp_begin. If the program is run until it stops at this breakpoint, it will ensure that all the processes have been created. They can then be controlled.

Next, what about a breakpoint for every superstep? The best way to accomplish this is to put the breakpoint in every process at the very last line of the bsp_sync. (See below for instructions how to do this.) Each process will stop at the beginning of the next superstep, which is a position in which there is the maximum degree of consistency between the states of the processes. At this point all transfers of data between processes are complete. This is a sensible time to examine the data contents of each process.

The progress of one or more processes can be tracked using the single stepping features of the debugger or by setting additional breakpoints in the code. At the bsp_sync barrier the process will be waiting completion of the superstep by all processes.

At any time to force processes through the barrier, issue CONTINUE commands to the nonrunning processes. Execution will stop at the breakpoint established at the start of the next superstep.

This procedure offers maximum control.

For more selective control, use a conditional breakpoint at bsp_sync end. For example, break execution only when a certain number of supersteps have been executed, or when particular supersteps are to be executed.

To set a breakpoint at the beginning of each superstep, put it at line 25, file bsp_misc.c, function bsp_sync_end (this is the end of bsp_sync). If the debugger allows, a breakpoint, when bsp_sync_cpp (bsp_sync is a macro which calls bsp_sync_cpp) has finished, might be put instead. Then step through the code of one or several processes. When the breakpoint is reached, do "step" twice to get in the program, not the library routines.

BSP offers considerable flexibility. For example, the debugger may be attached only to some of the processes and attached to any others later if necessary. Execution of all processes is synchronised at each barrier.

This allows you, as an alternative, to single step only some of the processes. In general each stopped process might be at any point in a superstep. However, it is always certain that all the processes are in the same superstep. To move to the next superstep, ensure that all the stopped/breakpointed processes continue execution and go through the barrier.

For some debuggers it is possible to insert action points, i.e. when a point in the program (process) is reached, a sequence of commands to the debugger is executed. Using this feature when stepping a process, and reaching a bsp_sync call, all the processes might be continued automatically (this may only work for parallel debuggers). dbx provides such a feature for IRIX for example. A little script could be written to insert such points automatically. Such a debugger together with a simple script could be very near to an ideal BSP debugger.

PROBLEMS:

If it is necessary to attach a debugger to a running process, be sure that your program handles the signal SIGSTOP correctly. This is because when attaching to a process most debuggers stop it by sending this signal to it. Any existing system call may be affected by this, e.g. semaphore operations, I/O operations (including reading from the console) as well as explicit catching of signals. See below for a particular system/debugger to understand whether this is needed.

The debugger may not allow putting breakpoints in a running process. In this case, stop the process with interrupt (e.g. by typing Ctrl-C), change the current file if necessary, place the breakpoint, and continue the process. Note that the program might (and usually will) be changed by the effect of responding to the Ctrl-C. This might happen when the debugger and the process (program) have the same TTY as a console input. This could be avoided by catching SIGINT in the debugger for example. See the particular debugger's documentation for details on how to deal with each case.

3. Multiprocessor (or single processor with multitasking) systems

Sequential native debuggers:

Usually individual processes can be debugged with any sequential debugger. A debugger which allows attaching to a running process usually can be used to debug several (or all) processes at once by running several instances of the debugger (or even different kinds of debuggers), everyone of which is debugging a different process. In this case it is impossible to put any action points as described above but most of the commands could be issued manually. Also usually it is impossible to place breakpoints without stopping a running process. So the process should be stopped (e.g. with Ctrl-C), a breakpoint placed, and then the process continued again.

If the process is stopped in a system call or other library routines, then "up" and "down" the stack trace or "return from subroutine" can be issued to the debugger to get to the needed function, or the current file may be changed to put breakpoint there. If both of these are not available, then try continuing and stopping the process at different points.

It might be more convenient to keep most of the process running most of the time, and stopping them just when needed to check data.

If execution is stepped over a bsp_sync call in any process, then execution will wait until the other processes reach the barrier. In general, for any process to pass a barrier synchronisation point, all processes must pass. In this case issue CONTINUE commands manually, if necessary, to all processes that are not running.

Problems:

The debugger may not handle forked processes correctly (this is significant only for multiprocessor systems, not distributed ones). Some debuggers explicitly need the program to be linked with a library of theirs which supplies a special version of fork to enable the debugger to catch execution of fork in the program. They are usually OK. The others may cause the child processes to become "zombie" processes (marked Z as status when viewed with "ps"), i.e. almost terminated, just after being forked and actually not being executed.

Sequential debuggers which cause the forked processes to become "zombie" processes

(These are expected to be sequential native debuggers which do not catch forked processes (e.g. dbx, (xx)gdb for SunOS 4.1)): For dbx and (xx)gdb for SunOS the steps 1-3 are done by a script.

1. Create a line in the source after the bsp_begin() call (preferably immediately after it), which is executed only by process 0. Be sure it is not optimized by the compiler. (There is such a line in the library - line 32 in file bsp_misc.c, function bsp_begin_end.)

2. Start the debugger. Put a breakpoint at the line created above (or the one in the library). Start the program with the debugger. Wait till it reaches the breakpoint. Do not put any other breakpoint before that.

3. Obtain the children's PID (by ps, for example).

4. Attach other instances of the debugger to some of them in different shells.

5. Debug the processes as necessary. Other debuggers could attach to other processes afterwards and some could detach from running ones.

Note: If the program has finished execution and needs to be rerun, then all breakpoints except for the one created in 2. should be removed. For some debuggers (e.g. gdb) disabling removes the breakpoint but not the trap, and it is the trap that causes the problems. It might be better to restart the debugger or tell it to execute the commands in the file created by the pre-processor for BSP debugging. All user breakpoints will be lost.

This is necessary. Breakpoints placed before the children (processes with bsp_pid() > 0) are forked, are inherited by the children. A problem arises when a child reaches such a breakpoint. No debugger is attached to it. A trap arises, and this causes the child to terminate (If a debugger is attached to the process, it handles the trap. However, if it is not this particular instance that has placed the breakpoint, the process cannot be continued properly).

Parallel native debuggers (i.e. which can switch between processes):

They are usually easier to use than normal sequential ones. Most parallel debuggers provide commands that can be executed on a group of processes. This can make it easier to finish a superstep: when a process reaches a bsp_sync, then continue all nonrunning processes. A simple script could be used to define the groups automatically, but this is different for each particular debugger. However, some parallel debuggers are not controllable when any processes are run by some commands such as "continue", "next", "step", "return". In this case other commands such as "resume" should be used. The particular debugger's documentation should be consulted for details. Because of this problem sometimes it will be better to run several instances of the debugger attached to different processes (or groups of them).

4. Distributed systems:

Sequential native debuggers can be used on each node and attached to a running process of the program (only if the debugger can attach to running process, otherwise it is not possible to be used at all). However, the program should be run independently beforehand. Then act as for sequential native debuggers for multiprocessor systems except that process 0 should be also attached to instead of being run under the debugger initially. Also there are no problems with forked processes as fork() is not used in this case by BSPlib (every process is running on a different node).

Some debuggers offer remote debugging. They can be used by creating a session for each node where the process needs to be debugged. Then act as necessary (see above - general guidelines).

In both cases it is possible that just some of the processes are attached by a debugger, and the others are left running. The latter could be attached later if needed. Some debuggers offer attaching without stopping the process (actually they attach to it and then continue it). If the process is stopped it might be continued again usually without effects to the program.

5. Particular debuggers:

SunOS 4.1.x for SunSparc

Problem: If process 0 is killed by the debugger, the semaphores and the shared memory are not deallocated. This should be done manually as the program cannot catch this signal and do this itself. Normally SIGKILL cannot be caught by the program. In this case continue the program with a signal INT or QUIT. Then all the processes will terminate and cleanup.

dbx (dbxtool)*:* A sequential native debugger which causes the forked processes to become "zombie". Most of the general comments are valid except for:

It is almost impossible to make a useful pre-processor. When dbx runs and executes some initial commands from a file, it has not read the symbolic information yet. Instead, after running dbx and loading the executable file, execute the debugger command “source $BSPHOME/bin/bspdbxinit.SunOS”, where $BSPHOME is the directory where BSPlib is installed. This will execute the program, stop just after bsp_begin, put breakpoint at the beginning of each superstep. It will also allow the user to see the actual PIDs of the processes with bsp_pid greater than 0. Use “print $pid[p]”, where p is bsp_pid()-1 of the needed process.

When any of the processes with bsp_pid() > 0 has finished, it will appear in the debugger that have attached to it as " Unexpected SIGTRAP in..." or "stopped in...", instruction "ta 1". Eventually these processes will finish when process 0 has finished. This usually will not cause problems.

* This happens because there is a trap in the forked process in order to make it become "zombie" when finished before the parent and wait for it to finish too. The debugger catches any traps, but have no treatment available for this one.

dbx does support action points, but it cannot issue commands to its other instances.

When a process is interrupted, it could be in any file. So it is essential before putting a breakpoint to change to the desired file (with "file"), or issue "up" several times until the debugger gets in the desired function (and file). "up" does not change what will be run next.

Pressing Ctrl-C will not interrupt the process that is debugged but if this is process 0, this may, and usually will, interrupt some of the other processes. So it is not advisable unless all the other processes have a debugger attached to them. In the latter case all the processes will be stopped by pressing Ctrl-C where process 0 is debugged, but if pressed anywhere else just the current process will stop.

(xx)gdb: A sequential native debugger which causes the forked processes to become "zombie". Recommended. Most of the general comments are valid except for:

There is a pre-processor to be run for easier use of the gdb on a multiprocessor system (or single processor one with multitasking). Execute "bspgdb <bsp_pid> <options for gdb>" to run gdb in another window with a command file which puts action points for easy debugging of BSP programs. For details see its documentation.

When gdb attaches to a process, the latter can be in any function. Usually it is a system one. The command "up" may be used to get the debugger into the desired function. "up" does not change what will be run next.

When any of the processes with bsp_pid() > 0 has finished, the debugger will still hang. Then Ctrl-C might be pressed in the respective window to bring it to normal, and it will say "Program exited normally".

Normally pressing Ctrl-C in any window except that of process 0 will cause only the program to stop and will not affect it. Pressing Ctrl-C in the window where process 0 is debugged will cause the program to stop. If then it is continued, it will quit unless all the processes are attached by a debugger. See the documentation of gdb for details on how it handles SIGINT. The best way to avoid this problem is to run gdb and the program with different TTY's (option -tty to gdb).

totalview(TV): A parallel native debugger. Good debugger. All of the general comments are valid except for:

Before debugging with totalview, the program should be linked with its library, which has fork(). The options to be added when compiling are "-L$TVHOME/lib -ldbfork", where $TVHOME is the base directory where totalview is installed. Then the program behaves normally when run under the debugger. However, if the program is not linked with this library, the guidelines for debugger causing a process to become "zombie" should be followed.

totalview does not have action points as described above (it has action points but in a different sense). However, it does allow issuing a command to a group of processes, but one process may belong to only one group.

It also has the disadvantage that when issued with a command "step" or "next", the debugger is not in control until the execution of the command has finished. This means that if stepped over (with "next") bsp_sync(), if any other process is stopped, the process will keep on running, and the debugger will not allow other commands to be issued. If this happens, stop the debugger with Ctrl-C. Most probably the program will be in system call. Then press "g" or Go to continue the process. In this case the debugger is fully controllable. Now if the process should be stopped after bsp_sync(), a breakpoint should be placed just after the call, if not already there. If a pre-processor for BSP debugging is used, there will be a breakpoint already there, if not removed. In all cases the function where the bsp_sync() call is made, appears in the Stack Trace window just under _bsp_sync_cpp.

If a process is stopped with Ctrl-C in the totalview window, where the process is debugged, no signal is send to it. Ctrl-C stops the process only if the debugger is not in control. Otherwise use the command Halt. If Ctrl-C is pressed in the window where only the program is executing(not TV), this will cause the program to terminate, unless the debugger is made to catch SIGINT. For pressing Ctrl-C in the window where TV has been run, consult TV documentation.

A breakpoint can be put everywhere even in a running process. It is advisable to put breakpoints against a running process only in the next supersteps or places where the user is sure will be executed later.

AIX for IBM SP2 (Distributed system):

Problem due to the OS: due to non-standard prologues in some AIX system libraries, the debuggers do not sometimes display correct stack trace information. This could also cause difficulties for debugging source code at all. The user could try stopping the program at different places.

dbx (xde): Sequential native debugger. All the respectively written above is valid except for:

It is necessary to put some delay in the beginning of the program after bsp_begin in each process to be debugged to ensure that the debuggers will attach to the process before it reaches a certain point. Usually the best way to do this is to read from STDIN. Then after attaching to the needed processes, <Enter> should be pressed in the window where the main program is executed to continue it. Note that the debugger will send SIGSTOP to the process, which will make the reading fail.

When the processes finish, the debuggers attached should be quitted to free the nodes engaged by the program.

Problem: Interrupting with Ctrl-C causes the program to stop and when continued to exit.

pdbx: This is a parallel version of dbx. It launches instances of dbx on each node and controls them. All the general guidelines apply plus:

The advantages regarding dbx are that commands might be issued on a group of processes at once, and acting only in one window. However, there is a slight problem. When one (or several) of the process are running and the others are stopped, pdbx is almost not controllable. If a command is issued, it goes directly to dbx. First it is executed by the instances where the processes are stopped, but when the running processes stop, it is also executed by the other debuggers, which might be not desired. This usually happens if stepped over a bsp_sync() call or a subroutine that calls it only on several (but not all) processes. Then the processes will wait for the others to reach the barrier synchronisation point.

In this case interrupting them by pressing Ctrl-C and the issuing "cont" to pdbx will cause the process to exit.

Therefore when about to step over a bsp_sync call or subroutine calling it, issue "cont" to all debugged processes. If already stepped over this call, issue commands that will execute bsp_sync in the other processes. Beware that in this case the commands go directly to dbx, i.e. only its subcommands should be used. Also these commands will be executed by the debugger that is running a process later. Obviously this will cause problems, so using pdbx is not recommended unless the user is extremely careful.

totalview (native version): At present because of problems due to non-standard prologues in some AIX system libraries, the debugger does not have correct stack trace and line information. Difficult to debug the source code.

totalview (distributed version): (the current version has problems) It seems that it will have the same problems as above.

gdb: See gdb for SunOS 4.1.x for SunSparc except that the guidelines for Distributed Systems should be taken into account.

IRIX 6.x for SGI Power Challenge

dbx: A parallel debugger for multiprocessor system. Most of the general comments are valid except for:

To stop the program in the debugger without disturbing it, it can be started with dbxhang, and in another window dbx attached to it with -p or -P. Also in this way the input and output for the debugger and the program are in different windows. See the manual page of dbxhang and dbx for details.

Some of the processes may be left running free, others being attached, and some may be attached or detached later. There is only one group, and it contains all the processes to which dbx has attached. Also most of the commands can be executed on a particular process.

When 'next', 'step', 'cont', 'return' are executed, the debugger is not controllable until the process has stopped. If needed it can be stopped with Ctrl-C in this case but see above what to do to avoid disturbing the program. Otherwise, every process should be continued with SIGCONT when stopped at SIGINT.

There is a pre-processor to be run for easier use of the debugger. Execute "bspdbx <options>" to start the program in a different window, and the debugger in the current one, automatically set some variables of the debugger, put breakpoints at the beginning of each superstep (this is optional), and stop the wanted processes just after bsp_begin. For details see its documentation.

The debugger allows putting action points as described above but not working quite as described and expected in the case of handling conditional commands for several processes. If this is fixed in future releases, dbx can be made very near to an ideal BSP debugger.

At the moment the debugger stops the processes at some points where it is not supposed when using action points as above.

cvd: A parallel debugger for multiprocessor systems. Most of the general comments are valid except for:

The debugger is always controllable, and breakpoints can be put even if the process is running. The debugger cannot execute commands for several processes, but a new window is opened if several processes need to be controlled or the debugger switches between the processes. The actual PID should be known to do this. There is a separate tool, called Multiprocess View, which can show the state of all processes and can run a debugger for any of them.

It is possible to put breakpoints at the exit of a function. This could be used as described above.

gdb: *not available for IRIX 6.x yet*