A Debugger for Computational Grid Applications[*]

Robert Hood, Gabriele Jost[†]

Numerical Aerospace Simulation Systems Division
NASA Ames Research Center

Abstract

The *p2d2* project at NAS has built a debugger for applications running on heterogeneous computational grids. It employs a client-server architecture to simplify the implementation. Its user interface has been designed to provide process control and state examination functions on a computation containing a large number of processes. It can find processes participating in distributed computations even when those processes were not created under debugger control. These process identification techniques work both on conventional distributed executions as well as those on a computational grid.

1. Introduction

While tools for debugging computationally intensive programs have improved substantially in the last few years [2] [11], there are two areas where further improvement is needed. First, existing tools do not cope well with applications running on heterogeneous computing platforms. Second, they do not provide sufficiently abstract and scalable operations for examining and controlling execution. This combination of inadequacies is particularly felt by programmers building applications to run on large-scale computational grids, such as NASA's Information Power Grid [6], which is based on the Globus toolkit [3].

In order to meet these needs, the *p2d2*[‡] project in the Numerical Aerospace Simulation (NAS) Division of the NASA Ames Research Center has taken their existing debugger and extended its capabilities. The original goals of the project were to build a debugger that was both portable across a variety of target machines and whose user interface scaled to be able to debug at least 256 processes [4]. In this paper we report on the effort to enhance that debugger to work in a computational grid environment. We begin with a discussion of how *p2d2*'s architecture accommodates the debugging of heterogeneous computations.

2. Accommodating Heterogeneous Computations

Debuggers, even serial ones, are inherently nonportable. Their basic task is to take a user request at the source level, map it to the machine level where it can be performed, and then map the result back to the source level. To accomplish this they rely on information and services from a variety of sources, for example:

- The compiler provides source line and symbol mapping data.

[*]This work was supported through NASA contract NAS 2-14303.

[†]The authors are employees of MRJ Technology Solutions, Inc. Their address is: M/S T27A-1, NASA Ames Research Center, Moffett Field, CA 94035-1000; e-mail: `{rhood,gjost}@nas.nasa.gov`.

[‡]"*P2d2*" stands for *p*ortable *p*arallel/*d*istributed *d*ebugger.

- The operating system provides services for starting and stopping processes.
- The target machine architecture defines a trap instruction that can be used for implementing breakpoints.

Thus, making a debugger portable from one target platform to another is a difficult task. Further complicating matters, a debugger for heterogeneous computations must solve these portability issues in a way that enables different target platforms to be available at the same time.

We handled these portability issues in *p2d2* by using a client-server architecture to isolate the platform-dependent code in a *debugger server* (see Figure 1). The server defines a collection of C++ objects that would exist in a debugging session, such as `Process` and `Stack`. The client consists of those parts of the debugger that deal with the distributed nature of the target computation and with the user interface, and it can be implemented in a highly portable fashion. For example, if the client has a `Process *p`, it could resume execution in it by invoking the operation `p->Continue()`. The object collection is discussed in detail in a previous paper [5].

In the initial version of *p2d2* we decided to build a debugger server based on *gdb* from the Free Software Foundation. There were two reasons for this approach: *gdb*'s source is freely available, and *gdb* is itself highly portable. In the *gdb*-based implementation of *p2d2* the remote server of Figure 1 is replaced with an instance of *gdb*. The debugger server is then an implementation of the C++ objects that uses *gdb* commands to perform any requested debugger server requests.

3. User Interface Issues

From a user's perspective, a debugger has two primary functions:

- *process control,* where the user is permitted to start execution of the target computation and to describe circumstances under which it should stop; and
- *state examination,* where the user can scrutinize expression values, source code, run-time stack, and other components of the current computational state.

The challenge in a multiprocess debugger is to provide these functions in a way that scales well to a large number of processes. In particular, the challenge for process control is to provide a way to propagate a single process control request, such as ***Continue,*** to a collection of processes. The challenge for state examination is to provide both an abstract, top-level view of the computation as well as information about a single process that has the same level of detail as a serial debugger would have.

To address the process control challenge, *p2d2* uses the notion of a *control set,* which is the collection of target processes that are subject to process control requests. The user has a variety of ways of setting membership in the control set. When a process control op-

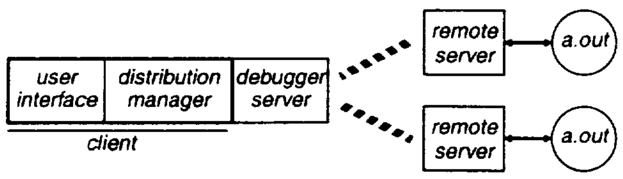


FIGURE 1. The client-server architecture of *p2d2*.

eration such as setting a breakpoint or continuing execution is requested, it is forwarded to all processes in the control set.

To address the state examination challenge, *p2d2* defines three zooming levels, providing a varying degree of abstraction versus detail:

- a top-level view, called the *process grid*, that provides a programmable display showing a few bits of information about all processes in the computation,
- an intermediate-level view provides a line of text summarizing the state of all processes in a user-selected set called the *focus group*, and
- a low-level view provides full information about a single user-selected process called the *focus process*.

The selection of the focus group and the focus process are done in the process grid display. When the user changes one of the selections, the display is updated to reflect the information about the new focus. For example, if the user changes the focus process, then all state examination displays that are focus-process-sensitive, such as the source and stack displays, will be updated.

Perhaps the most novel feature of the *p2d2* user interface is the programmability of the process grid described earlier. This feature permits a quick scan of a large number of processes to isolate a process behaving in an unexpected manner. Such a process is a good candidate for closer scrutiny as the focus process. This feature is illustrated in Figure 4.

If the debugger is to be useful, it must provide mechanisms for handling mundane tasks. In particular, a debugger for distributed programs should be able to find and control all of the processes participating in a computation. The user should not be required to filter through lists of processes running on a large number of machines in order to determine which of them belongs to a job. In the next section we describe how *p2d2* addresses this problem.

4. Finding the Computation's Processes

One of the implementation challenges facing a distributed debugger is how to acquire control over the processes in the computation. As with serial debuggers there are two cases to consider:

1. the computation was initiated from the debugger when the user invoked the ***Run*** command, and
2. the user initiated the computation outside of the debugger and then requested that the debugger "attach" to it.

In order to handle case 1, the debugger needs to resolve a conflict with the process starting mechanism (*e.g., mpirun, globusrun, pvmrun*) that initiates the distributed computation. The conflict comes about because both the debugger and the process starter want to control the actual `fork()` and `exec()` that start the individual processes. A customary way to resolve this conflict is for the process starting mechanism to allow a user-supplied proxy program (sometimes called a *tasker*) to perform the `fork` and `exec`. Both *pvmrun* and *globusrun* permit the debugger to gain control over process creation in this way.

If *p2d2* is going to be used to initiate a Globus job, the user must include the clause

```
(paradyn="P2D2_HOST P2D2_PORT p2d2 /u/p2d2/bin/gdbserver")
```

in the RSL script to be handed off to *globusrun*. This indicates that `/u/p2d2/bin/gdb-`

`server` should be used as a tasker. When the user requests a ***Run*** the following happens.

- *P2d2* invokes *globusrun*, changing the `P2D2_HOST` and `P2D2_PORT` strings in the RSL script to the machine name on which *p2d2* is running and the number of a tasker contact port that it created.
- When *globusrun* starts the tasker, it passes it the machine name and port number that *p2d2* wrote in the RSL script. The tasker and *p2d2* then establish a socket.
- The tasker starts the target executable and reports the target's pid on the socket. The target sleeps.
- *P2d2* asks tasker to start *gdb* and then tells the *gdb* to attach to the target.

This results with process diagram shown in Figure 2.

In order to handle case 2 above, where the user requests that the debugger attach to an existing computation, the debugger needs:

- a list of the processes that are participating in a computation, and
- a mechanism for gaining control over them.

If a tasking mechanism exists, it can be used to meet these needs. For example, if *p2d2* is to be used to attach to an existing Globus job, the job must have been started with the `/u/p2d2/bin/gdbserver` tasker described previously. The resulting tasker processes will each create a port and record the contact information in a single file in the file system. When the user starts up *p2d2* and asks for it to attach to the processes named in the file, the debugger will use the contact information to establish sockets with the taskers. Each tasker can then start up a *gdb* which will attach to the target process.

In our discussions so far, we have relied on a tasking mechanism at process startup. Unfortunately MPI-1 [8] does not have such a feature, because process creation is not part of the standard. To handle MPI jobs, *p2d2* uses *rsh* to run a copy of *gdb* on the machine where the target process exists. There are two remaining needs:

- a list of pairs [*machine, pid*] for each process in job, and
- a way to keep a newly started process from executing code.

The second condition allows us to handle debugger-initiated runs in an identical manner to run initiated outside of the debugger. We can address both of these needs by using the profiling mechanism of MPI and providing a specialized version of `MPI_Init()`.

The `MPI_Init` used by *p2d2* does the following.

- It calls `PMPI_Init()`.
- The process with rank 0 gets the machine name and process ID for all processes. It writes that data in file system.
- If the process was initiated from the debugger, it goes into an infinite sleep loop.

When the debugger attaches, it establishes any necessary breakpoints, terminates the sleep loop, and then continues execution.

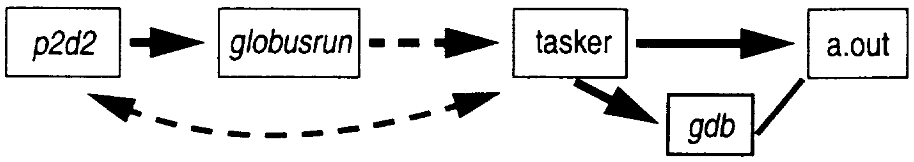


FIGURE 2. After *p2d2* initiates a *globusrun*.

There are two minor limitations in the version of `MPI_Init` used by *p2d2*: it is not possible to debug the code that executes before it `MPI_Init` called, and the user must link the application with *p2d2*'s version of `MPI_Init`. The latter condition could lead to a conflict if other libraries want to use the profiling mechanism of MPI.

While these limitations exist, in practice they restrict *p2d2*'s capabilities very little. Furthermore, we are hopeful that an *mpirun* based on the process control operations in MPI-2 [8] will provide a tasking mechanism that will eliminate the restrictions altogether.

5. Related Work

There are three commercially available distributed debuggers of note. *TotalView* [2], from Etnus, is a third party debugger that runs on a number of high performance computing platforms. It is currently not capable of debugging heterogeneous computations. Furthermore, while it is capable of debugging thousands of processes and threads, the user interactions are at a fairly low level. *Prism* [11], from Sun Microsystems is derived from the Thinking Machines product of the same name. It is not portable to systems other than Sun. While its user interface led the way in scalability, it too, could be more abstract. The final distributed debugger of note is *Jessie* [9] from SGI/Cray. It is a Java-based front end to *gdb* in much the same style as *p2d2*. [Jessie was just announced; we will have a more detailed comparison to it in the proceedings version of the paper.]

6. Project Status and Future Work

The current *p2d2* system has been demonstrated on several target architectures. After the recent work to accommodate Globus computations, it has been successfully used to control 128 processes running on 3 different SGI Origins on the IPG. It has also been used on heterogeneous computations running under Globus (see Figure 3), as well as MPI, and PVM [10].

In the near future, we want to use the Globus database, MDS, to record information about jobs started outside the debugger with *globusrun*. This will enable us to attach to Globus computations without a reliance on the target systems sharing a file system with the debugger host. We will also complete an array viewer that is capable of displaying a "global" view of data that has been distributed across multiple processes (see Figure 5).

Further in the future we will, if there is sufficient user demand, adapt *p2d2* to work with Legion[7] and Condor [1]. We also plan to enhance *p2d2* to find differences between serial and distributed versions of the same code. This could be particularly useful when computer-aided parallelization tools are used to perform domain decomposition.

7. Conclusions

In this paper we have described a debugger for heterogeneous, distributed programs. As we discussed, a client-server model greatly simplifies the implementation. Its user interface has been designed to provide process control and state examination functions on a computation containing a large number of processes. We also described several approaches for finding processes participating in a distributed computation and how those techniques could be used in a computational grid environment.

8. References

[1] The Condor Project. `http://www.cs.wisc.edu/condor/`.

[2] Etnus, Inc. The TotalView Multiprocess debugger. `http://www.etnus.com/products/totalview/`.

[3] The Globus Project. `http://www.globus.org/`.

[4] R. Hood. "The *p2d2* Project: Building a Portable Distributed Debugger," *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, May 1996.

[5] R. Hood and D. Cheng. Accommodating heterogeneity in a debugger—a client-server approach. *Proceedings of the Twenty-eighth Annual Hawaii International Conference on System Sciences*, Jan. 1995. [Also published in an extended form as a chapter in *Tools and Environments for Parallel and Distributed Systems*, Kluwer Academic Publishers, Norwell, MA.]

[6] The Information Power Grid Project Plan. `http://www.nas.nasa.gov/~wej/IPG/`.

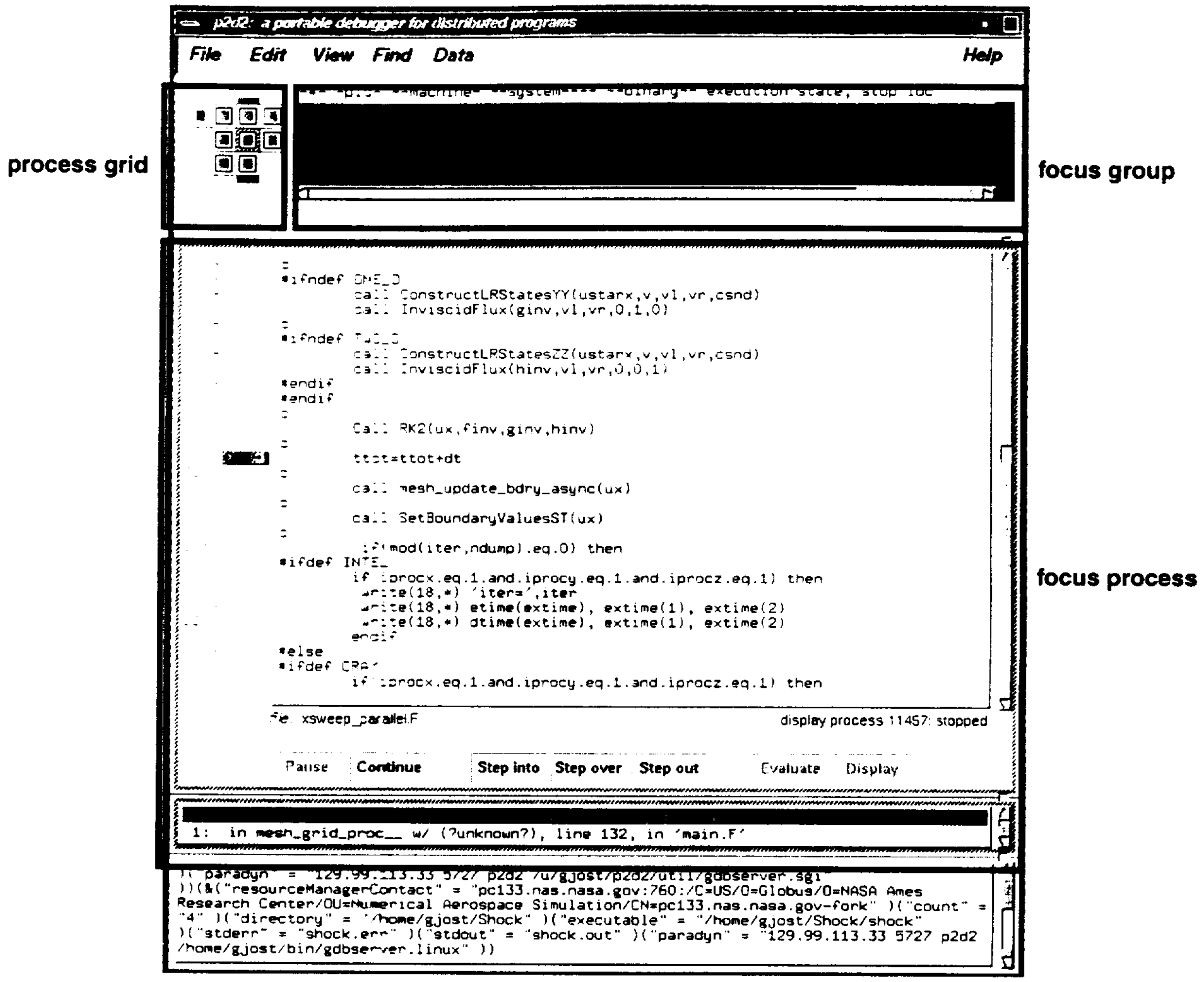
[7] The Legion Project: `http://legion.virginia.edu/`.

[8] MPI committee. Message Passing Interface. `http://www-unix.mcs.anl.gov/mpi/`.

[9] SGI, Inc. The Jessie Cross Platform Integrated Development Environment. `http://oss.sgi.com/projects/jessie/`.

[10] Vaidy Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience* 2(4):315-339, 1990.

[11] Thinking Machines Corporation. *Prism User's Guide*. Thinking Machines Corporation, Cambridge, MA, Dec. 1991; also: `http://www.sun.com/servers/hpc/software/configuration.html#prism`.



```
c
#ifndef ONE_D
        call ConstructLRStatesYY(ustarx,v,vl,vr,csnd)
        call InviscidFlux(ginv,vl,vr,0,1,0)
c
#ifndef TWO_D
        call ConstructLRStatesZZ(ustarx,v,vl,vr,csnd)
        call InviscidFlux(hinv,vl,vr,0,0,1)
#endif
#endif
c
        Call RK2(ux,finv,ginv,hinv)
c
        ttot=ttot+dt
c
        call mesh_update_bdry_async(ux)
c
        call SetBoundaryValuesST(ux)
c
         if(mod(iter,ndump).eq.0) then
#ifdef INTEL
        if (iprocx.eq.1.and.iprocy.eq.1.and.iprocz.eq.1) then
         write(18,*) 'iter=',iter
         write(18,*) etime(extime), extime(1), extime(2)
         write(18,*) dtime(extime), extime(1), extime(2)
        endif
#else
#ifdef CRAY
        if (iprocx.eq.1.and.iprocy.eq.1.and.iprocz.eq.1) then
```

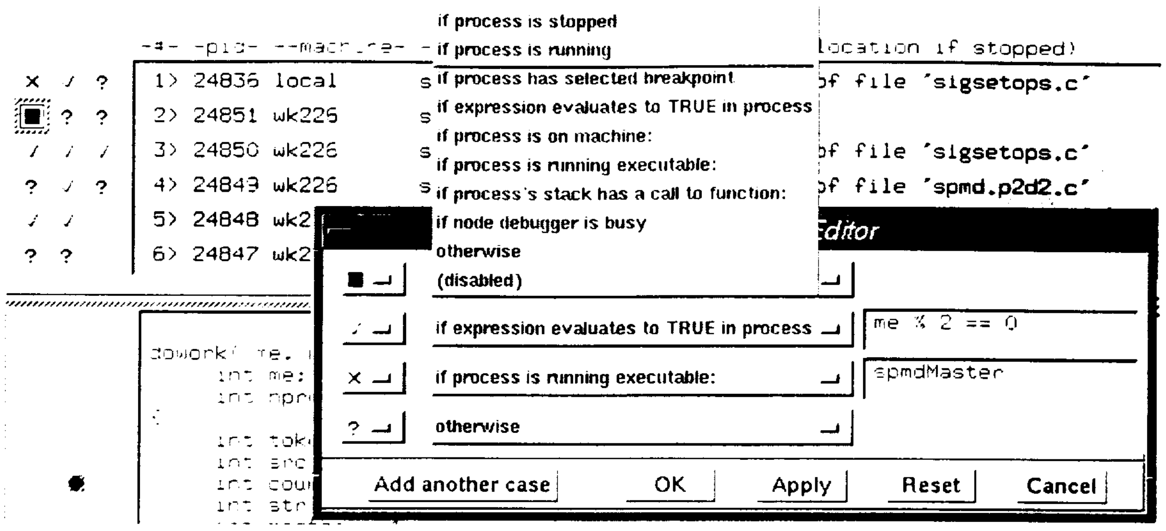
```
File: xsweep_parallel.F                         display process 11457: stopped
```

```
 1:  in mesh_grid_proc__ w/ (?unknown?), line 132, in 'main.F'
```

```
)( paradyn" = "129.99.113.33 5727 p2d2 /u/gjost/p2d2/util/gdbserver.sgi
))(&("resourceManagerContact" = "pc133.nas.nasa.gov:760:/C=US/O=Globus/O=NASA Ames
Research Center/OU=Numerical Aerospace Simulation/CN=pc133.nas.nasa.gov-fork" )("count" =
"4" )("directory" = "/home/gjost/Shock" )("executable" = "/home/gjost/Shock/shock"
)("stderr" = "shock.err" )("stdout" = "shock.out" )("paradyn" = "129.99.113.33 5727 p2d2
/home/gjost/bin/gdbserver.linux" ))
```

FIGURE 3. Debugging a heterogeneous computation running under Globus.

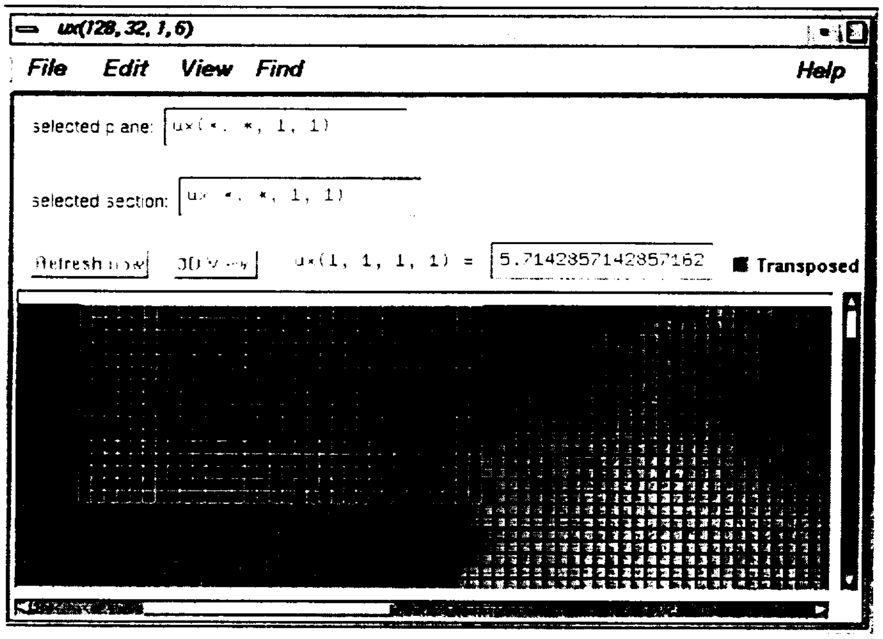
Four processes are running on an SGI Origin (local) and four on a PC running Linux (indus). The *process grid* shows all of the processes in the computation. The *focus group* displays one line of text for each process in the selected column of the process grid. The *focus process* part of the display resembles a serial debugger on the single process selected in the process grid.


if process is stopped
if process is running
if process has selected breakpoint
if expression evaluates to TRUE in process
if process is on machine:
if process is running executable:
if process's stack has a call to function:
if node debugger is busy
otherwise
(disabled)
if expression evaluates to TRUE in process
me % 2 == 0
if process is running executable:
spmdMaster
otherwise
Add another case
OK
Apply
Reset
Cancel

```
                                                    location if stopped)
x ✓ ?   1> 24836 local   s                          of file 'sigsetops.c'
■ ? ?   2> 24851 wk225   s
✓ ✓ ✓   3> 24850 wk226   s                          of file 'sigsetops.c'
? ✓ ?   4> 24843 wk226   s                          of file 'spmd.p2d2.c'
✓ ✓     5> 24848 wk2
? ?     6> 24847 wk2
```

FIGURE 4. Programming the process grid.

In this example, the process grid view has been programmed so that running processes are depicted as green rectangles, processes where the expression "me % 2 == 3" is true are depicted with a checkmark, processes running "spmdMaster" are marked with an X, and all other processes with a question mark. For each process, this list of predicates is evaluated in order until a true one is encountered. The process is then depicted in the grid using the picture corresponding to that predicate.



selected plane: ux(*, *, 1, 1)

selected section: ux(*, *, 1, 1)

ux(1, 1, 1, 1) = 5.7142857142857162

FIGURE 5. Distributed array visualization.

The picture shows a global display of a 2-dimensional slice of the 4-dimensional array **ux** at the breakpoint indicated in figure 3. The code is a parallel implementation of the Euler equations in 3D. The array **ux** is distributed across 8 processors of an SGI Origin. The local parts of array **ux** are gathered from each processor and assembled to a global picture. The array elements that reside on the focus process are highlighted.