Message Passing for Parallel Processing of Pressure-Sensitive Paint Images

Wim Ruyten and William E. Sisson
Aerospace Testing Alliance, Arnold AFB, TN
{wim.ruyten, William.Sisson}@arnold.af.mil

Abstract

A message-passing scheme is described that allows parallel processing of pressure-sensitive paint images on a machine with multiple processors or a cluster with multiple nodes. The scheme implements the use of forks and pipes in the former case and socket-based TCP/IP communications in the latter. The approach demonstrates how multiple copies of a nonparallel legacy code (in this case, NASA's Green Boot software) can be made to run in parallel in either a parent-child or a client-server configuration. Results are presented for benchmark data from wind tunnel tests of an F-16C fighter jet model and NASA's X-38 Crew Return Vehicle.

1. Introduction

Pressure-sensitive paint (PSP) has established itself as an important test and evaluation tool for measuring full-field pressure distributions on test articles in aerodynamic test facilities, particularly transonic wind tunnels[1]. At the Arnold Engineering Development Center (AEDC), these pressure distributions are obtained by processing image data from up to eight digital cameras in the test section of the wind tunnel (see Figure 1). Final data are presented to the customer on a three-dimensional (3D) grid of the test article. Processing steps include automatic target detection, image registration, image alignment, reflected-light correction, conversion of signal ratios to pressure, and mapping of image data to the 3D grid[2,3].

Data processing is accomplished with Green Boot, a code that was developed originally by NASA Ames Research Center and McDonnell Douglas Aerospace (MDA). The code consists of some 90,000 lines of C (with some FORTRAN), offers extensive Graphical User Interface support, and has provisions for script-based processing. It has always been possible to run multiple copies of the code as separate processes. However, management of the required databases and script files has proven to be cumbersome and an impediment to achieving automated processing in near real time (i.e., within 15 seconds of taking data).

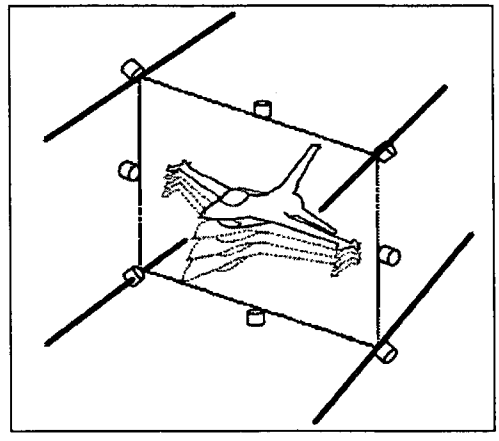

Figure 1. Wind tunnel test section with test article and eight cameras

2. Parallellization Approach

Figure 2 illustrates how data processing requirements naturally suggest a master-slave implementation of parallel processing PSP image data: A master process initiates a request to process a data point, whereupon up to eight slave processes (one for each camera) perform partial processing of image data from a particular camera, resulting in processed data that are mapped to the 3-D grid. The master process then collects this mapped data from the slaves, merges this data onto a single 3-D grid, and completes processing of the data. These steps are described in detail in Reference 3. The present paper focuses on the message-passing scheme that was developed to implement this concept.

Figure 3 lists a typical macro that is executed by the master process: It defines the names of the to-be-processed images and the final 3-D data file on the basis of run and sequence numbers of the data points. It then instructs each of the slave processes in Figure 2 to execute the script `slave_macro` for a particular camera, merges

0-7695-2259-9/04 $20.00 © 2004 IEEE

the returned data onto the 3-D grid, and saves the final result in Plot-3D (or other) format. The INSTRUCT and MERGE_3D commands are built into the Green Boot code, whereas _define and _save are script files that are defined by the user in terms of native Green Boot commands. The variable $P3D is set by the script _define.

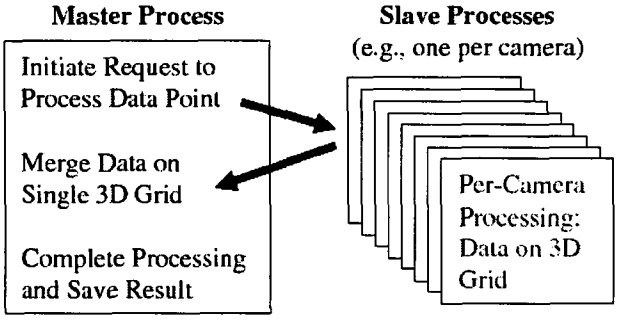

Figure 2. Master-slave approach to PSP data processing

```
_define $1 $2                          // $1, $2 define data point
foreach CAM (01 02 ... 08)             // Loop over all cameras
  INSTRUCT $CAM slave_macro $1 $2 $CAM // Process $CAM on Slave $CAM
end                                    // End of loop
MERGE_3D $P3D                          // Combine data on 3D grid
_save $P3D                             // Save final result
```

Figure 3. Typical master macro script

In Figure 3, the instruction "INSTRUCT $CAM..." causes the script "slave_macro $1 $2 $CAM" to be executed on the slave process assigned to camera $CAM. (Here, $1 and $2 represent the run and sequence numbers of the data point.) Figure 4 lists a typical realization of slave_macro. It loads the reference and run images from the specified camera; finds the registration targets; registers the images by determining the mapping transformation from 2-D image space to 3-D model space, aligns the two images (to account for small rotations and/or shifts between the two); ratios the images; converts the ratio to a pressure or pressure coefficient (based on a calibration equation); maps the resulting pressure to the 3-D grid; and sends this information back to the master process. The variables $REF, $RUN, and $P3DCAM in Figure 4 are each set by the script _define. All scripts prefaced with an underscore are defined by the user.

The processing sequences from Figures 3 and 4 can be used both with intensity-based PSP (in which a reference image is obtained with the tunnel at atmosphere) and with lifetime-based PSP, in which both the reference and the run images are obtained at the run condition. Variations on these scripts are possible. For example, in order to perform a correction for reflected fluorescent light, the reference and run images both have to be fully mapped to the 3-D grid before conversion to pressure can take place. In this case, the master macro performs MERGE_3D on two sets of images, performs the reflected-light correction, and only then performs the _convert macro before saving the final result.

```
_define $1 $2 $1          // $1, $2, $3 define data point, camera
_load $REF $3             // Load reference image (e.g., wind-off)
_load $RUN $2             // Load run image (e.g., wind-on)
_register $REF            // Find targets and register ref image
_register $RUN            // Find targets and register run image
_align $RUN $REF          // Align the run image to the ref image
_convert $RUN $REF        // Ratio images and convert to pressure
_map $RUN $P3DCAM         // Map pressure data to 3D grid for this camera
_send $P3DCAM             // Send result back to master process
```

Figure 4. Typical realization of a slave_master in Figure 3

3. Implementation

The master-slave communications scheme from Figure 2 has been implemented in two ways. In the first approach, the master process is forked repeatedly before X-window support is requested, and write and read pipes are established from the parent process (the master) to the resulting child processes (the slaves). In the second approach, slave processes are started up in server mode on one or more host machines at preselected ports, and the master process acts as a client that requests a socket-based TCP/IP connection to each slave[4]. In both cases, a set of file descriptors is established on each process (fdw[] for writing, fdr[] for reading), as illustrated in Figure 5. This allows the program to communicate between processes by performing system-defined WRITE and READ calls on these file descriptors, as illustrated in Figure 6 for a set of data contained in a buffer, buf. The while loop in Figure 6 ensures that data packets are received in full, even if more than one set of write/read operations is required. Typically, 0.9 to 1.4MB of data are returned to the master process per slave process per data point. These data are buffered in packets of up to 10,000 bytes, a size that appears to work well for both the pipe and the socket implementations.

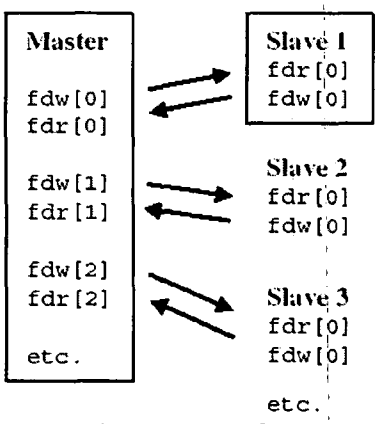

Figure 5. Implementation of master-slave communications through file descriptors

```
int len = sizeof(buf);                  // Length of data in buf
write(fdw[0],&len,sizeof(len));         // Write length of data
write(fdw[0],buf,len);                  // Write data

int len, nr=0;                          // Initialize num bytes read
read(fdr[s],&len,sizeof(len));          // Read length of data from "s"
while(nr!=len)                          // Loop until all data read
  nr+=read(fdr[s],&buf[nr],len-nr);     // Read data from slave "s"
```

Figure 6. Interprocess communication via system-level `WRITE` and `READ` calls

Figures 7 and 8 show partial source code for the establishment of the file descriptors in the two scenarios. The fork/pipe implementation in Figure 7 is straightforward and requires separate pipes for writing and reading. The socket implementation from Figure 8 is somewhat more involved, but allows both reading and writing on a single socket. Byte swapping, if required, is performed by the master process only. (For example, in Figure 6, the variable `len` would be byte-swapped prior to the first write operation.)

```
for (int s=0; s<N; s++) {               // Request N slave processes
  int MS[2],SM[2], pid;
  if (pipe(MS)<0 || pipe(SM)<0) error...  // Create the pipes
  if ((pid=fork())<0) error...          // Fork the main process
  if (pid==0) {                         // Parent process (master)
    fdw[s]=MS[1]; close(MS[0]);         // Set file descriptors
    fdr[s]=SM[0]; close(SM[1]);
    continue;
  }
  else if (pid>0) {                     // Child process (slave)
    fdw[0]=SM[1]; close(SM[0]);         // Set file descriptors
    fdr[0]=MS[0]; close(MS[1]);
    (enter slave mode...)               // Indefinite loop
  }
}
```

Figure 7. Fork-pipe implementation with master process as parent, slave processes as children

```
Start slave process in server mode:
struct sockaddr_in sock;  int fd;
(populate sock: AF_INET, port number, INADDR_ANY)       // Specify port
int fd_tmp = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);   // TCP-IP socket
if (bind(fd_tmp,&sock,sizeof(sock))<0) error...
if (listen(fd_tmp,MAXQUEUE)<0) error...                 // Set queue size
if ((fd=accept(fd_tmp,NULL,NULL))<0) error...           // Connect client
fdr[0]=fdw[0]=fd; close(fd_tmp);                        // File descriptors
(enter monitoring mode...)

Start master process as client to slave s:
struct sockaddr_in sock; struct hostent *host;
if ((host=gethostbyname(hostname))==NULL) error...      // Specify host
(populate sock: AF_INET, port number, host->h_addr)     // Specify port
int fd = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);       // TCP-IP socket
if (connect(fd,&sock,sizeof(sock))<0) error...          // Connect server
fdr[s]=fdw[s]=fd;                                       // File descriptors
(continue with main program...)
```

Figure 8. TCP/IP implementation with master process as client, slave processes as servers

The fork/pipe approach is particularly suited to a single machine with multiple processors, such as a SGI architecture. It has the advantage that the resulting slave processes are tied directly to the master process, so that separate management of the slave processes is not required. The socket approach is intended for use on a cluster of machines in which multiple processors communicate via an Ethernet connection. In this case, one must exercise care to ensure that the slave processes are running and communicating when the master requests that a data point be processed. In either case, the resulting scheme is much more robust than the one that was used in the prototype version described in Reference 3, which relied on the use of instruction files that were written to and read from a common file system. In particular, conflicts that arose when one process was trying to read from the file system while another was in the process of writing to it (or vice versa) are avoided with both the fork/pipe and socket approaches, which perform buffering and synchronization of read and write operations implicitly. Master and slave processes still access a common database for tracking information related to the raw and processed image data. This is a commercially available SQL database that is designed specifically to synchronize access by multiple processes.

To facilitate management of the slave processes on a cluster, a Green-Boot-specific daemon process is registered on each node of the cluster under the super-daemon `inetd`[4]. This allows a Green Boot slave process to be started, polled, and shut down remotely from the node on which the master process is executed (typically, the front end of the cluster).

Some further details on the operational modes of the improved Green Boot code are presented in the Appendix.

4. Results

Correct functioning of the code was verified on several platforms, including an SGI Octane 2 (with dual IP30, R14000 processors running at 600 MHz), an older SGI Origin 2000 (with eight IP27, R10000 processors running at 195 MHz), and a Linux cluster consisting of a front end and eight nodes. The Linux cluster (running Red Hat Linux 7.3) is powered by dual P4 Xeon processors on both the front end and the nodes, running at 2.4 GHz and 2.2 GHz, respectively. Communication between the front end and the nodes is through an HP switch with a 1-Gbit/sec Ethernet connection to the front end and 100-Mbit/sec connections to each of the nodes. Measured transfer rates on the cluster (with one master process and eight slave processes on the nodes) exceed 80 Mbit/sec.

When the code was run on a single multiprocessor machine, no significant difference in total processing time was found between the fork/pipe and socket implementations. This is consistent with the fact that interprocess communication requires far less CPU time than does the actual processing of the data. As expected, best performance was obtained on the Linux cluster, which has the highest processing speed among the platforms tested. Total processing times per data point for two benchmark data sets (involving an F-16C fighter jet and NASA's X-38 Crew Return Vehicle) are detailed in Reference 3 for the three machines quoted above, and are shown in Table 1, for both "2-D" processing (without a reflected-light correction) and "3-D" processing (with a reflected-light correction). In the F-16C case, image data

from eight cameras were processed on a master process supported by eight slave processes; for the X-38 case, image data from six cameras were processed on a master process supported by six slaves. In both cases, four 1024×1024×16-bit images are used for each camera (wind-off and wind-on images, each with an associated black image that is subtracted as part of the `_load` macro in Figure 4), and final data are mapped onto a 3-D grid with more than 300,000 grid points. Figure 9 shows sample results for the two benchmarks, with color representing the value of the measured pressure (red represents high pressure, blue represents low pressure).

Table 1. Processing times per data point in seconds

	SGI Origin 2000 (8 Proc, 195 MHz)		SGI Octane 2 (8 Proc, 600 MHz)		Linux Cluster (1+8 Proc, 2.2 GHz)	
	F-16C	X-38	F-16C	X-38	F-16C	X-38
2D (w/o refl)	27	22	30	22	7	6
3D (w/o refl)	41	31	38	28	10	8

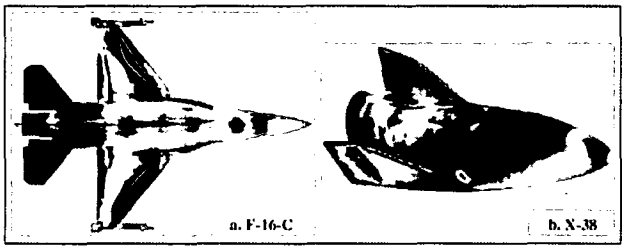

Figure 9. Examples of processed data for the two benchmark cases.

From a performance perspective, the results in Table 1 are particularly significant in that a single data point can be processed in well under 15 seconds on the Linux cluster. This was the AEDC goal for achieving near-real-time processing of PSP test data.

5. Conclusion

It has been successfully demonstrated that it is possible to parallellize an intrinsically nonparallel legacy code, particularly one that lends itself to a master-slave configuration in which the master process initiates a request to process a data point, slave processes perform partial processing of the data, and the master process completes processing of the data after combining the results from the slaves. Communication between the master and the slaves can be accomplished either across pipes by repeatedly forking the main process (on a multiprocessor machine) or (on a cluster) by establishing socket-based TCP/IP communications between master and slaves, with each slave running as a server process on a node, and the master process running as a single client on the front end of the cluster. This approach to parallellization may be applicable to data processing schemes other than those used here for pressure-sensitive paint.

6. Appendix: Green Boot Run Modes

The improved Green Boot code can be run in four modes, depending on the presence of an optional master-slave argument on the command line:

(1) "`$PATH/gb config`": Start a single process on the specified configuration files (i.e., a collection of data files with different extensions, each of which has "`config`" as the root name of the file. (This is the original Green Boot mode.)

(2) "`$PATH/gb config -forkN`": Start a master process that will fork `N` slave processes. The slaves are terminated upon exiting the master process.

(3) "`$PATH/gb config -portP &`": Start (in background mode) a slave process that will, upon being contacted by a master client process, establish a TCP/IP connection to the master process on (slave) port number `P`. This command can be performed either manually on the machine upon which the slave process is to be run, or automatically, as the slave process can be spawned by the `inetd` super server, when the super server receives a remote request from the master process. The slave process is terminated when the server receives the built-in Green Boot command "`QUIT`" from the master process.

(4) "`$PATH/gb config -serv`": Start a master process that will request that a series of slave processes be spawned remotely, depending on the contents of a setup file. The setup file contains, for each camera, the hostname and port number of the node that is to run a slave process for that camera in server mode, as well as the paths to the database files and the Green Boot executable on the node. When the master process is able to verify that all of the slaves are in listening mode (see Figure 8), the socket connections are established. When the user exits the master process, the master sends a `QUIT` command to each of the slave processes.

Acknowledgments

The research reported herein was performed by the Arnold Engineering Development Center (AEDC), Air Force Materiel Command. Work and analysis for this research were performed by personnel of Aerospace Testing Alliance, the operations, maintenance,

information management, and support contractor for AEDC. Further reproduction is authorized to satisfy needs of the US Government. This work was supported in part by funding from the Test and Evaluation Program of the Air Force Office of Scientific Research, managed by Dr. Neil Glassman. The Linux cluster was purchased with funds provided by the High Performance Computing Modernization Program Office.

References

1. Bell, J.H., E.T. Schairer, L.A Hand, and R.D Mehta, "Surface Pressure Measurements Using Luminescent Coatings." *Annual Review of Fluid Mechanics*, Vol. 33, 2001, pp. 155–206.

2. Ruyten, W., M. Sellers, R. Clippard, and M. Craig, "Pressure-Sensitive Paint in Wind-Tunnel Testing: A Computational Challenge." DoD HPC Users Group Conference, Albuquerque, NM, June 5–8, 2000.

3. Ruyten, W. and S. Sellers, "On-Line Processing of Pressure-Sensitive Paint Images." *AIAA Paper 2003-3947*, 21st AIAA Applied Aerodynamics Conference, Orlando, FL, June 23–26, 2003.

4. Quinton, R., "An Introduction to Socket Programming." http://www.uwo.ca/its/doc/courses/notes/socket/.