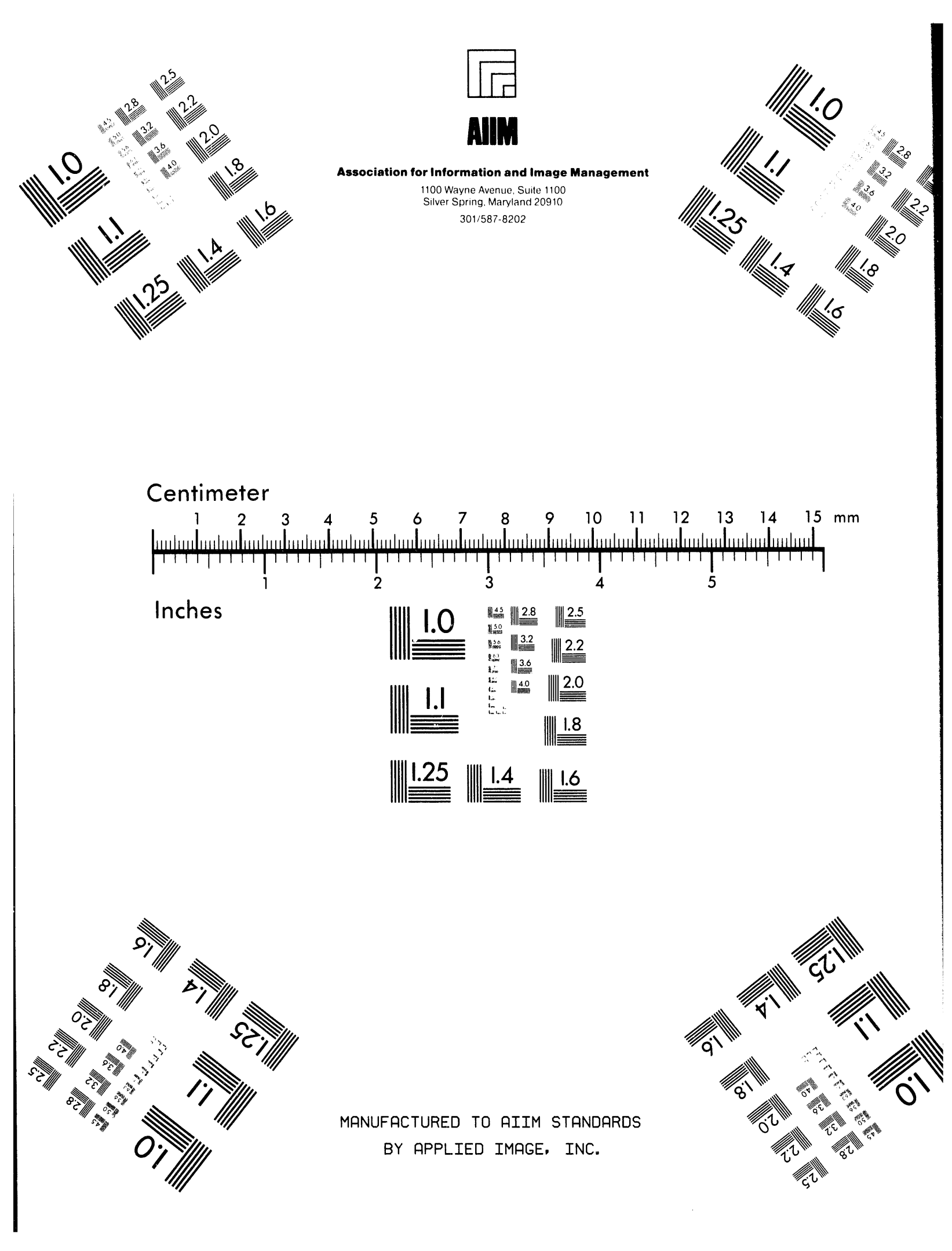
AIIM
Association for Information and Image Management
1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910
301/587-8202
Centimeter
Inches
MANUFACTURED TO AIIM STANDARDS
BY APPLIED IMAGE, INC.

1 of 1

SUNMOS for the Intel Paragon
A Brief User's Guide*†

Arthur B. Maccabe
University of New Mexico
Albuquerque, NM 87131
maccabe@cs.unm.edu

Kevin S. McCurley
Sandia National Laboratories
Albuquerque, NM 87185-1109
mccurley@cs.sandia.gov

Rolf Riesen
Sandia National Laboratories
Albuquerque, NM 87185-1109
rolf@cs.sandia.gov

Stephen R. Wheat
Sandia National Laboratories
Albuquerque, NM 87185-1109
srwheat@cs.sandia.gov

1 Background

SUNMOS is an acronym for **S**andia/**UNM** **O**perating **S**ystem. It was originally developed for the nCUBE-2 MIMD supercomputer between January and December of 1991. Between April and August of 1993, SUNMOS was ported to the Intel Paragon. This document provides a quick overview of how to compile and run jobs using the SUNMOS environment on the Paragon.

The primary goal of SUNMOS is to provide high performance message passing and process support while consuming a minimal amount of memory. As an example of its capabilities, SUNMOS Release 1.4 occupies approximately 240K of memory on a Paragon node, and is able to send messages at bandwidths of 165 megabytes per second with latencies as low as 42 microseconds using Intel NX calls. By contrast, Release 1.2 of OSF/1 for the Paragon occupies approximately 7 megabytes of memory on a node, has a peak bandwidth of 65 megabytes per second, and latencies as low as 42 microseconds (the communication numbers numbers are reported elsewhere in these proceedings [1]).

A Paragon running SUNMOS will use OSF in the .service and .io partitions, but will have SUNMOS loaded on all or some of the compute nodes in place of OSF. Compute nodes running SUNMOS do not appear in the .compute partition. The number and configuration of SUNMOS and OSF compute nodes is decided at boot time; see [3] for further details.

Through emulation libraries, SUNMOS currently supports many of the nCUBE message passing routines (e.g., `nread` and `nwrite`) and many of the NX message passing routines (e.g., `csend`, `isend`, `crecv`, and `irecv`). In addition, the standard SUNMOS library supports the C standard I/O library and the FORTRAN I/O library. As a consequence, many nCUBE and Intel NX codes will run under SUNMOS without modification.

A separate document [2] describes the differences between standard Intel NX routines and the SUNMOS emulation library. Another document [3] describes the installation procedure. In addition to these documents, there are man pages for the following SUNMOS commands and library routines:

```
yod fyod fservers create_yod_config
getcomm getpcb showmesh _nsend/_nrecv
```

All of these are available by anonymous ftp from `ftp.cs.sandia.gov` in `pub/sunmos/doc`. If you need additional support or have further questions, send email to `sunmos-support@cs.unm.edu`.

2 Compiling SUNMOS applications

At Sandia, SUNMOS executables are generated on Sun workstations using the Intel supplied cross compilers. While it is possible to compile application programs on the service nodes on a Paragon, we do not recommend this practice.

The SUNMOS distribution comes with three shell scripts that can be used to compile programs written in C, C++, and FORTRAN. These scripts are called `sicc`, `siCC`, and `sif77`, respectively. These scripts invoke the appropriate cross compilers and link the application with the SUNMOS libraries.

These shell scripts, other SUNMOS utilities, and other useful files are located in a single directory tree. Check with your local system administrator for the location of this directory at your site.

The `sunmos` directory contains several subdirectories:

*This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000.

†This document was written on May 31, 1994. You can obtain the latest version of SUNMOS documentation of via anonymous ftp from `ftp.cs.sandia.gov` in the directory `pub/sunmos/doc`.

`current/bin` contains executables for the SUNMOS utilities. You will want this directory in your path.

`current/man` contains documentation on SUNMOS utilities. You may want to add this directory to your manpath.

`current/include` contains header files for the SUNMOS libraries. This directory is automatically searched when you use the SUNMOS compiler scripts, so you shouldn't need to reference this directory explicitly.

`current/lib` contains the SUNMOS libraries. This directory is automatically searched when you use the SUNMOS compiler scripts, so you shouldn't need to reference this directory explicitly.

With the exception of the -nx flag, all compiler flags are treated the same way under SUNMOS as under OSF, so there should be minimal changes required to makefiles. The -nx flag is used under OSF to link OSF libraries for compute nodes, and *should not* be used under SUNMOS.

3 Running applications—yod

The previous section described a cross-compilation environment on a Sun workstation. There are several utilities on the Paragon itself located in the directory `/sunmos/bin`. The most important ones are:

`yod` – the generic host node program; handles mesh allocation, program load and execution, and file I/O. For users familiar with the nCUBE, this corresponds to the xnc command. For users familiar with the Intel Delta, this corresponds to the mexec command.

`showmesh` – shows the current mesh allocation. An alternate tool called `showparts` is available via anonymous ftp from `ftp.cs.sandia.gov` in the directory `pub/paragon-contrib`.

`getcomm` – displays the communication buffers for a stopped process (useful in debugging).

`getpcb` – displays the process control block for a stopped process (useful in debugging).

In the remainder of this section we describe how `yod` is used to allocate, load and support the execution of application programs. In the next section we describe how to use `getcomm` and `getpcb` during application debugging. Once you are familiar with the basic operation of these utilities, you should consult the man pages for further details.

The `yod` program runs in the service partition, and handles all requests from the SUNMOS compute nodes that it controls (much like the proxy process under OSF). A special message passing module written by Intel allows communication between `yod` and SUNMOS running on compute nodes.

One important point about `yod`: when you abort a job under control of `yod`, you should be very careful about how you kill `yod`. The correct way to do this is to issue a control-C to the yod, or to type `kill <pid>` from another shell. *Do not* use `kill -9` or `kill -KILL` to kill a `yod` process. Doing so will leave your nodes allocated, and prevent future runs on those nodes until the machine is rebooted (or until `create_yod_config` is run again). If you run `create_yod_config` while there are running jobs, these runs are likely to be corrupted, so avoid running it unnecessarily.

3.1 An example

Before we consider the command line options supported by `yod`, it is instructive to consider a simple command line:

```
% yod -sz 8 a.out 100 200
```

This command line instructs `yod` to allocate 8 nodes from the SUNMOS nodes and load the application `a.out` on all 8 of these nodes. Any arguments after the name of the file containing the executable are passed to the application as command line arguments. In this case, each of the 8 application processes is provided with the command line arguments '100' and '200'.

3.2 Node allocation

When you load and execute an application program, `yod` allocates the nodes that the application uses. When the application completes, `yod` reclaims the nodes for use by other applications.

There are three command line options that control the algorithm `yod` uses to allocate nodes: `-size`, `-allocation`, and `-base`.

The `-size` option can be abbreviated as `-sz` (as shown in the previous example). This option controls the number of nodes allocated for the application. If you don't specify a size, the default is to allocate a single SUNMOS node. The size can be specified as an integer (e.g., 8) or a rectangle (e.g., 2x4). When the size is specified as a rectangle, the height is given first.

When you just specify the number of nodes to be allocated, `yod` first tries to allocate a rectangular region of the mesh, trying to keep the region as square as possible. If `yod` is unable to allocate a rectangular region, it will try to scatter the application processes throughout the mesh (still trying to keep them reasonably close together). If you specify the size as a rectangle, `yod` will only consider rectangular regions of the specified shape.

Using the `-allocation` option, you can control the allocations that `yod` will consider. `Yod` recognizes three allocation modes: strict, lax, and any. When the mode is *strict*, `yod` will only attempt contiguous allocations, i.e., the processes in your application will not span nodes used by the processes in another application. This mode is useful if you are concerned that

another application might saturate a communication channel and interfere with the communication in your application.

When the allocation mode is *lax*, **yod** will still attempt to perform a strict allocation. If that fails, **yod** will then try to allocate nodes in a rectangular shape, spanning rows and columns with other application processes. (This is the default allocation mode when you specify the size as a rectangle.)

When the allocation mode is *any*, **yod** will first attempt a strict allocation. If that fails, **yod** will then attempt a lax allocation. If that fails, **yod** will then try to find the desired number of node anywhere in the mesh. (This is the default allocation mode when you specify the size as a single number.)

The **-base** option controls the starting position for an allocation in the mesh. This option is useful when you need to take advantage of differences in the nodes (e.g., different memory sizes).

3.3 Controlling memory allocation

When an application process is loaded, SUNMOS allocates memory for five distinct regions: code (text), static data, communication, stack, and heap. The code region is always just large enough to hold the code for the process. Similarly, the static data region is always just large enough to hold the static data for the process. By default, SUNMOS allocates 256K bytes for the communication and stack regions. After is has allocated memory for the code, communication, and stack regions, SUNMOS, by default allocates the remainder of the application memory (i.e., the memory that is not used by SUNMOS itself) for the application's heap region.

You can directly control the amount of memory allocated for the communication, stack, and heap regions using the **-comm**, **-stack**, and **-heap** command line options of **yod**. Each of these options takes a single number, the number of bytes to allocate for the specified region. As an example, if you use **-comm 1000000**, then SUNMOS will allocate 1,000,000 bytes of storage on each node for message buffers.

The communication region is used to buffer messages that have arrived at the receiving node, but not yet requested by the application process. If you allocate too little space for the communication region, SUNMOS will abort the application processes as soon as a message arrives when there is not sufficient space in communication region to hold the message.

The stack region is used to hold local variables. If you use large automatic arrays, you may exceed the default stack size. If your application exceeds the size allocated for the stack region, the results are unpredictable; however, you will most likely get a data access fault during execution.

The heap region is used to hold the dynamic space used by an application process. In C programs, this space is accessed using the standard library routines: malloc, calloc, and free.

4 Debugging (such as it is)

There is currently no support for a debugger under SUNMOS (aarrgh!). We recommend debugging under OSF, which has integrated debugger support. Support for a debugger may be added in the future.

As a poor man's substitute for a debugger, you can observe the front panel lights and use the getcomm and getpcb utilities to find the current state of a hung process.

4.1 The lights

The lights on the front of the machine can sometimes be used to diagnose what went wrong. Each node has 6 lights associated with it: one red light, and five bar lights. The bar lights on each node will be referred to as numbered 1-5, counting from the top.

- when a node is not running a user process, the lights repeatedly blink in the pattern 1—2—3—4—5—4—3—2—1.
- when the primary processor on a node has faulted, the lights repeatedly blink the pattern: 3—2,4—1,5—2,4—3.
- when the coprocessor on a node faults (and the primary processor does not) light # 3 comes on and stays on.
- when the processor is running in user mode, light # 1 comes on and stays on. This includes the case when a user is blocked waiting for a message.
- when a node is in system mode but hung receiving a message, light # 4 is on. This is usually accompanied by the red light on the node being on, and will usually require a reboot of the machine. This indicates an OS failure and should be investigated further if it happens.
- when a node is in system mode but hung sending a message, light # 5 is on. This is usually accompanied by the red light on the node being on, and will usually require a reboot of the machine. This indicates an OS failure and should be investigated further if it happens.

The case of when a user is blocked waiting for a message is indistinguishable from when a user is running happily on a node, but this can be discovered using **getcomm** (described below).

4.2 Inspecting the message queues of hung jobs

The **getcomm** utility can be used to display the list of unreceived messages and posted receives, and whether the user process is currently blocked waiting on a message.

```
% ge% getcomm -n 12

Comm Comm Buff Analysis for node 12 (logical d0)
--------------------------------------------------------

User User has not processed the following message(s) yet:
!  l!  len= 2, type= 55 (0x00000037), src= 13 (0x0000000d), src_pid 1
!   !        logical src= 1 (0x00000001)
!   !        dst= 12 (0x0000000c), dst_pid 1

========== time stamp = 0x0000017a

User User is waiting the following message(s):
?  l?  len= 0, type= 54 (0x00000036), src 65535 (0x0000ffff), pid 65535 (0x0000ffff)
?   ?        time stamp = 0x0000017e, msg body= 0xf0f7b9d8
```

Figure 1: Sample output from getcomm.

A A sample output for **getcomm** is given in Figure 1. This This shows the output from a one node of a program that that was run on physical nodes 12 and 13, running to a certa certain point and hanging.

Th The output from **getcomm** shows that node 12 (using ping physical node numbering) is currently blocked waitiwaiting for a message of type 54 and length 0 from any sany source (**0x0000ffff**), and has received a message from from logical node 1 (physical node 13) of type 55 and lengtllength 2 bytes in the system buffer.

Th The utility **getpcb** is a bit more obscure—see the man man page. One use is to decide what state a node is in. Nin. Note that running **getpcb** or **getcomm** while the node node is sending a message will cause a fault in itself, so yoso you should only use these on hung jobs.

4.3 4.3 Deciphering node fault dumps

W When a SUNMOS node process faults (yes, it can happhappen), it displays some very crude register and process icess information that can be used to diagnose the failure (ure (those of you old enough to remember the 1960's will fwill feel right at home). An example of such a dump is givis given in Figure 2.

Th This indicates that node 2 faulted at an instruction whoswhose hex value is **0x139d0001**, located at address **0x180x18c18**. The table of 32 values is the contents of the 32 in32 integer registers. The instruction itself is probably of litof little use without a disassembler, but the address can bcan be used to find the routine in which the program faultefaulted. You can run

```
r    nm860 -f -v a.out > a.out.map
```

on ycon your SUNMOS executable **a.out** to produce the map map **a.out.map**. By looking at the starting addresses of thof the functions in your program, you can locate the addreaddress where the fault occurred. The dump in Figure 2ure 2 was produced by a program whose dump contains tains the following lines:

```
_mai_main     |0x00018b80|extern|  |  |  |.text
_qrf_qrfact   |0x00019240|extern|  |  |  |.text
```

Since the logical address **18c18** from the dump is after **18b80** and before **19240**, you can tell that the fault occurred in **main()**. Furthermore, **18c18-18b80 = 98** in hex, or 152 in decimal. Since each instruction is 4 bytes, this is 38 instructions into **main()**. From inspection of the output from

```
sicc -S -Manno main.c
```

we find that the instruction is

```
ld.l 56(sp), r28
```

which is generated from the line

```
algvalue = *algptr
```

From this you might suspect the problem is dereferencing a null pointer **algptr**. This is not fun—or particularly efficient—but sometimes better than nothing. If you're faint of heart, then try the old 1970's way of inserting print statements.

5 I/O and fyod

The **fyod** program provides scalable file I/O service to SUNMOS nodes. If this program is not used, then all I/O from a SUNMOS job is funneled through the **yod** process on the service partition of the Paragon. While this is simple and adequate for small numbers of nodes or small amounts of I/O, it represents bottleneck in I/O bandwidth.

fyod is intended to help with this difficulty. Users can start an **fyod** process on the .io partition to handle requests to different directories in parallel. As an example, suppose the paragon has a 2 node .io partition with 2 RAID devices: **/raid/io_01** and **/raid/io_02**. The user could have these managed separately by typing

```
fyod -sz 2 -pn .io -dir /raid/io_##
```

```
NODE 0xd: proc 0x0, psr 0x208a0, epsr 0x21080402, fsr 0x350000a1
     pc(log) 0x18c18, pc(phys) 0xf0e64c18, instr 0x139d0001, sp 0x5edb990
     FAULT TYPE: Data Access Fault (bad address)

     r0:  0x00000000,   r1:  0x00046a28,   r2:  0x05edb990,   r3:  0x00000000
     r4:  0x05655bfc,   r5:  0x04029498,   r6:  0x0402bb84,   r7:  0x0402bbbc
     r8:  0x04028c10,   r9:  0x05655ba8,   r10: 0x04028c00,   r11: 0x0401c028
     r12: 0x0401c024,   r13: 0x04028c08,   r14: 0x04028c0c,   r15: 0x0402ab28
     r16: 0x00000001,   r17: 0x05655ba8,   r18: 0x05655bb0,   r19: 0x00000013
     r20: 0x00000000,   r21: 0x00000001,   r22: 0x062dc000,   r23: 0x00000000
     r24: 0x00000000,   r25: 0x062dc000,   r26: 0x00000000,   r27: 0x062dc000
     r28: 0x00000001,   r29: 0x1b564b18,   r30: 0xffff0000,   r31: 0x04020000
```

Figure 2: Output from a node that faulted under SUNMOS

The use of ## indicates that requests to open a file from a directory with ## replaced by one of the strings "01" or "02" are directed to the appropriate I/O node. A similar paradigm exists in the NX library when a file is opened: a string of three or more # characters in a file name is replaced by the node number of the job opening them). To describe how the **fyod** program works, we start with an **fopen** request made by a SUNMOS node. This is sent to the yod process responsible for the job. **yod** then starts by looking up in a directory of file services. If an **fyod** service is found that handles the directory in which the file resides, the open request is sent to the appropriate **fyod**. This **fyod** opens the file and sends a response back to the SUNMOS node. Future I/O traffic for this file travels between the SUNMOS node and the I/O node directly, without intervention from the **yod** process. A user can display the directory of existing **fyod** services by typing the command **fservers**. At some point in the future, **fyod** will likely become part of the boot procedure, eliminating the need for users to manage their own I/O servers. Further details on **fyod** appear in the man page. **fyod** requires a large segment of wired memory on an I/O node, so it is not advisable to run more than one of these on a single I/O node. This will also result in a decrease in performance.

6 Advanced features of yod

6.1 Heterogeneous loads

A recently added feature to **yod** was the ability to load different executables on different nodes, all within a single application. The exact syntax for doing this should be contained in the man page, but roughly speaking you use `yod -F loadmap` to specify that loads are controlled by the contents of the file **loadmap**. An example file would be of the form:

```
-sz 16x32
prog1 16x2:0 -comm 4000000
prog2 16x30:2 -comm 1000000
```

This specifies that **yod** should allocate 512 SUNMOS nodes arranged in a 16x32 rectangle. The SUNMOS executable **prog1** should be loaded on the 16x2 rectangle anchored at node 0, i.e., columns 0 and 1 of the 16x32 rectangle. Further, **prog2** should be loaded on the 16x30 rectangle anchored at node 2, i.e., columns 2–31 of the original rectangle. Note that the two programs are supplied with different comm sizes (other **yod** options except for size options can go here). The specification of the "anchor nodes" 0 and 2 for the two applications are required, because **yod** does not do tiling. Another example is

```
-sz 512
prog1 32 -comm 4000000
prog2 480 -comm 1000000
```

This more flexible specification allows **yod** to allocate *any* 512 nodes obeying the allocation strategy, and assign the executables to the nodes in node order (**prog1** goes on nodes 0,...,31, etc.).

6.2 Use of the second processor

The GP node of an Intel Paragon has two i860XP processors that share access to the memory (MP nodes will have even more). Originally Intel planned to use this processor as a communications coprocessor, with the goal of lower latency and ability to overlap computation and communication on a node. Under SUNMOS, there are currently three modes supported for use of the second processor:

mode 0 affectionately called "heater mode"; where the second processor is inactive.

mode 1 the second processor is used as a communication coprocessor. This results in significantly lower latencies for message passing, and allows better overlap between communication and computation.

mode 2 the second processor can be used as an additional compute processor, with shared memory. This has been referred to by some as "SUNMOS turbo mode".

Modes 1 and 2 will not work reliably with some early hardware, but seem to work reliably now on the Sandia hardware. These modes are selected through the `-proc` option to `yod`.

Mode 2 requires a user to specify some work to be done on the second processor via a function or subroutine call that takes a single integer argument. There are currently some restrictions on what can be called on the second processor. In particular, you cannot print from the coprocessor, send messages, or use certain system calls that use static variables (and hence are not reentrant). The two processors will share the heap, but have separate stacks (as a result, the use of the second processor will consume extra memory on a node). The performance improvement that is achieved using mode 2 depends primarily on the degree to which the processors use their memory caches. We have witnessed speedups as high as 95% for codes that reuse their caches very well. In particular, it was possible to achieve a speed higher than the rated peak of the Paragon using this method (!).

In the C language, the interface to the second processor in mode 2 is through a function `cop()`, whose prototype is:

```
cop(void (*f)(), volatile unsigned *flag,
    void *arg);
```

The function `f` to be run on the second processor should have a prototype

```
void f(int arg);
```

In order to run `f(arg)` on the second processor, you would use statements like

```
volatile int flag=0;
cop(f,&flag,&arg);
```

The main processor can then go off to do other work, and when the function `f` completes on the second processor, `flag` will be incremented. The main processor can later check for completion of `f` by inspecting the value of `flag` to see if it was incremented.

There is currently no interface to the second processor directly from Fortran, but this will be corrected in future versions of SUNMOS. For now, the following piece of C code can be used. In a file called `fcop.c`, put the following lines:

```
static volatile iflag;
void fcop_(void (*addthem)(), int *j)
{
  iflag=0;
  cop(addthem,&iflag,j);
}
void fcopdone_(int *flag)
{
  *flag = iflag;
}
```

Compile `fcop.c` using the command

```
%sicc -c -O3 fcop.c
```

Then link the file `fcop.o` with the rest of your application. In order to call a subroutine

```
subroutine f(i)
```

on the second processor, you use the line

```
call fcop(f,i)
```

Be sure to declare that f is an external subroutine. You can later check for completion using the lines

```
iflag = 0
call fcopdone(iflag)
   ... other work on main processor
if (iflag .eq. 1) then
   ... f completed
```

The file `fcop.c` is currently in the directory `/home/u/mccurley/fcop` inside the `cs.sandia.gov` domain, along with a test Fortran program called `test.f`. It should also be on `ftp.cs.sandia.gov` in the directory `pub/paragon-contrib`.

7 Other issues

There has been a data corruption problem during the load in some of the early 32-megabyte nodes on the Sandia machine. If you observe this, it will produce an error message of the form shown in Figure 3. This informs you that the text or data segment of your loaded program produces an incorrect checksum, so that the program may give incorrect results. One way to protect against it is to specify a heap that fits in 16 megabytes when loading on 32 megabyte nodes.

The processor modes 1 and 2 are not sufficiently tested, and some bugs are known to exist on some hardware. The Sandia machine has had all of the node boards upgraded to fix the known bugs, and any new bugs should be reported to `sunmos-dev@cs.sandia.gov`. Other sites may observe problems with these modes if their hardware is an early version and has not been upgraded.

At present there is no document describing the limitations of the nCUBE emulation library, but some notable ommissions are the lack of global operations such as `ndsumn()` and `nglobal()`. Broadcasting by giving a destination of -1 to `nwrite()` is also not currently supported.

Floating point exception handling for IEEE floating point arithmetic is currently not implemented in SUNMOS. This is a design decision dictated by the fact that such exception handling is done in software, is incredibly slow, occurs rarely, and would essentially double the size of the SUNMOS operating system.

Constructive feedback on this document is welcome. Send comments to `mccurley@cs.sandia.gov`. Send complaints to `/dev/null`.

```
BAD TEXT LOAD: tmin=0x10023222 tmax=0x10012122 s30sum=1001
BAD DATA LOAD: dmin=0x10921213 dmax=0x0x110271 s30sum=1172
```

Figure 3: Error messages from a bad load.

References

[1] Bernard Traversat, Bill Nitzberg, and Sam Fineberg, Experience with SUNMOS on the Paragon XP/S-15, in ISUG-94.

[2] Kevin S. McCurley, Intel NX compatibility under SUNMOS, Sandia National Laboratories Technnical Report # 93-2618.

[3] T. Mack Stallcup, Installation Instructions for SUNMOS.

[4] man pages for yod, fyod, fservers, getpcb, getcomm, showmesh, and nsend/nrecv.

DATE
FILMED
8/25/94
END