Freeware for Cluster Computing

Ron Minnich, Maya Gokhale, Aaron Marks, Jim Kaba, John Degood
Sarnoff Corporation
Princeton, NJ 08540
rminnich@sarnoff.com

CyClone

- 144 nodes, 272 Pentiums
  - 1 PII/200
  - 256 P200MMX
  - 3 Cyrix 586/200
  - 12 P90
- “Cluster Fat Tree”:
  - 25 3COM 3C3000 100Bt switches
  - 2 3Com 3C9000 1000Bt switches
- Tiled display
- VIA network for 10 nodes (8 switched, 2 point-point)

Applications

- 3D rendering to replace Sarnoff’s 1024-processor SIMD engine (“Princeton Engine”)
- MPEG prototype encoding development
- Ray Tracing for Optics
- Creating encoded files
- Other device simulation, data transformation tasks
- Research in metacomputing

3D rendering to replace a 1024-processor SIMD engine

- PE is a 1024-processor SIMD supercomputer
- Equivalent to 16K CM-2 except PE has very fast video I/O
- PE: 4 fps, 256³ cube
- 128-processors on linux: 2 fps
- 32-node paragon at NASA: .25 fps, 128³ cube
MPEG prototype encoding development

- MPEG motion vector estimation is embarrassingly parallel with right software
- Limit before we got to sarnoff was 4 machines scaling
- We have “task bag” software with task dispatch/gather overhead of 2 ms/task
- Task bag supports hierarchy
- Can now easily scale to full CyClone cluster

Creating encoded files

- Can in the limit perform faster-than-real-time encoding
- Paid for the first cluster in a few months.
- Note: if you have seen parallel MPEG papers, and they’re not from Sarnoff, they’re probably wrong (all the ones we’ve seen to date are wrong ...)

Ray Tracing for Optics

- We’re using TNT (The Next Taskbag) to support an optical simulation
- Rays are traced via lens configurations
- This step follows analytical analysis and precedes actually building it
- Being used to support a seamless tiled display
- Before: 10K rays overnight on a powermac
- After: $10^9$ rays in 1-2 hours on full cluster

Other device simulation, data transformation tasks

- They come out of the woodwork as soon as the cluster appears ...
- Result: our two clusters (‘94 and ‘97 models) paid for themselves in six months
- BUT: none of the apps were traditional scientific apps such as: SOR, LU, Matrix Multiply, SPLASH 2, etc.
- These apps don’t scale on most clusters anyway (unless it’s 4 nodes or so)
Research in metacomputing

- DARPA paid for half the nodes
- Research in private name spaces, better control of TCP connections, and “Network Threads”

Background

- This is fifth in a series, going back to 1991
- Initially we built workstation clusters, but cut over to Pentiums in 1994
- First 16, then 32, then 48, ...
- Comparison:
  - 1994, 16 Indys, $327K
  - 1994, 16 P90s, $36K
  - P90s ran at 90% of Indys for apps of interest
  - Performance/Price was 8:1
  - And we could get source ...
- That ended the need for workstations ...

The case for clusters

- Answer: Clustering became practical in
  - 1997
  - 1994
  - 1991
  - 1984
- And the question is:
  - “What is 1984”

1984-1993

- Mary Mock shows how to beat a 7600
- Tom Nash and others at Fermi Labs start construction of “Crates”
- By 1991, clusters were making money for IBM, Sun (in internal use)
- Sun MICA 128-node cluster
- IBM basement full of RS/6000s
- ca. 1993, 1-2% of the world’s supercomputers were retired by clusters
- HP, IBM, Sun, DEC: 100 TFLOPS box will be a cluster
Progress since 1984

- What has happened: hand-built clusters for turnkey applications
- What has happened in some cases:
  - `cc myprogram.c`
  - `a.out`
- We had this environment at SRC in 1994
- But people still can not:
  - Watch
  - Debug
  - Easily control
- their program as an entity

Are Clusters Multiprocessors?

- Can we pretend:
  - Nodes are processors
  - Network is backplane bus
  - OSes should share name spaces, memory, paging, PID space, etc.?
- In our experience, no
- The cluster::=multiprocessor analogy does not work for us
- The two models have different reliability models, failure modes, latencies, etc.
- Our approach: Process-centric

Process-centric

- We focus on the top-level process, not a single computer or cluster
- Process locates resources and attaches as needed (Via Private Name Space)
- Process creates shared memory segments for export or imports other process’s shared memory segments (via Zounds)
- Process efficiently creates groups of processes for remote execution (via vex library)
- Obviously, this is freeware, or I wouldn’t be here ...

ZOUNDS

- Quick overview of ZOUNDS
- Rationale
- Programming Interface
- Performance
- Applications
- Conclusions
Zounds is part of a series of DSMs

- Mether (started 1988)
- MNFS (started 1992)
- CMA (started 1994, dropped 1995)
  - User-mode DSM built in C++
    - I really don’t like C++ that much any more ...
- To understand ZOUNDS, need to understand Mether/MNFS

Mether (1988-91)

- Software distributed shared memory
- Use custom protocols on Ethernet
- Supported two page sizes: 32 bytes, 8192 bytes
- Integrated into SunOS VM as a device driver
- Simple consistency model (WORM)
- User-controlled coherency
  - Standard cache ops such as purge, update, invalidate from user-mode

Mether Clients/Servers

- Memory Servers available throughout the network
- Client applications can run on Servers
- Uses UDP, SunOS device driver layers

MNFS (1992-)

- Modified NFS which supports Mether model
- Requires modifications to server and client
- Runs on SunOS, Solaris, Irix, AIX 3.2, and Solaris, FreeBSD 2.0.5R, NetBSD 0.9
- Use to support applications on:
  - 48-node SRC Cluster
  - Aurora Gigabit Network Testbed
  - 100+ nodes at a govt. site
  - U. Koeln in Germany
ZOUNDS Goals

- Simple, low-overhead coherency model
  - A la DEC Memory Channel, SCRAMNet
- Process-centric, not OS-centric, DSM:
  - Clients self-page via SEGV handler (< 10 μS)
  - Servers serve from a process image
  - Support multiple memory object types
- Designed to be a good match to the MINI Virtual Interface ATM card
  - I.E. Process-driven, not OS-driven, IO
- Not tied to any particular OS or transport
- Provide kernel/user client and server implementations

ZOUNDS

- Simple library of client and server calls
- Currently supports TCP/IP connections as well as IP multicast
- Experience has shown that non-DSM experts can easily parallelize code
- Not OS-Specific
- Use of IP multicast for updates is (to our knowledge) unique

Zounds library

- All server functions start with the letters zs
- All Client functions start with the letters zc
- Any user program can be a server-- even an ordinary application
- The program can start the server code and have it run asynchronously
- Client programs are self-paged: allows control of policy such as page size
- No special OS support is required for operation, save on older version of ***BSD
- OS extensions can improve performance

Zounds Server

- Servers are multithreaded or single-threaded (determined by programmer)
- Any application can be a server
- Servers can issue I/O requests for pages to clients -- I/O need not be only client-driven
- Servers track which clients have which pages
- Any data region can be the backing store: mapped files, arrays, SYSV shared memory
Sample Code

```c
ZSINFO *zs;         zs = zsalloc();        zs->maxclients = 1;        zs->size = 16384;        zs->key = 0;        zssetup(zs);        zservershow(stdout, zs);        if (nodetach)                zserve(zs);
else
    zsdetach(zs);
```

Pointer declaration
Allocate the server
Set parameters: maxclients
... size of the region
... “key”
Set up the server.
Show the server status
Check “nodetach”, run:
Synchronously until done
Asynchronously

Comments on server code

- There can be more than one server set up for:
  - Same region
  - Different region
  - Overlapping region
- For the asynchronous case, to make the server exit:
  - Set zs->zsexit to 1 (causes that server to exit)
  - zsexit to 1 (causes all servers to exit)
- In the asynchronous case, the server can send updates/invalidates to clients

ZOUNDS Client

- Clients attach one or more segments from one or more servers
- Clients can attach some or all of the remote segment
- Clients can page from themselves
- Clients can cause pages to be returned to servers, and cause the server to send:
  - invalidates to other clients
  - updates to other clients

Client Code Fragment

```c
ZCINFO zc;
char *name = “c097/2000”;        zc.off = 0;
zc.v = 0;
zc.size = 16384;
zcssetup(&zc, name);
```

Declare the client structure
This is the name (normally from argv)
Set offset into remote segment
Allow zounds to pick location in client memory
Set the size
Call setup function.
Using the client segment

- You can reference it as memory or do “I/O”
  
  ```c
  if (doread) { 
    doread indicates “I/O” path
    zcread(zc, (off_t) 0, size);  
    read from server
  } else {
    int n;
    n = * (int *) zc->v;  
    Reference the data
    zcinvalidate(zc, (off_t) 0, size);  
    Throw the reference away
  }
  ```

- I/O and Memory references can be interspersed and remain consistent

Multicast Setup

- Multicast support is useful for many types of applications
- For servers: set up a server with the normal path, then add a multicast port to it:
  ```c
  zsmulti(&zs, mcastip);
  ```
- For clients: set up a client and add a multicast port:
  ```c
  zcmulti(&zc, mcastip);
  ```

Additional Server Multicast Ops

- Servers can do multicast sync operations
  ```c
  for(;! zsexit; ) { 
    t.tv_sec = 1;
    t.tv_usec = 0;
    zserve_timeout(&zs, &t);
    /* sync all the pages for zs, starting at 0, dirty or not */
    zsmultisync(&zs, (off_t) 0, (size_t) 128, 0);
  }
  ```

- Once a second, update all multicast clients

Multicast Client Ops

- Clients can accept a multicast update:
  ```c
  Numbytes = zcmultiupdate(zc);
  ```
- `zcmultiupdate` consumes all pending multicast packets for the client
Applications

- Heat transfer solver
- Distributed tiling using simulated annealing (used multicast heavily)
- My favorite: world’s most expensive screen saver (video)

Performance

- SEGV handler performance:
  - 7 microseconds, Linux
  - 11 microseconds, FreeBSD
  - Times can be reduced with some careful redo in the kernel fault path: FreeBSD, 5.5 uS
- Page fault on FreeBSD: 1.22 mS/4096 byte page, 2.2 mS/16384 byte page, 512 microsecond/512 byte page
- Page fault on Linux

Multicast Update: A server can send at least 200 updates/second without loss at the client

Conclusions

- Zounds is simple:
  - To program
  - To use
  - To understand performance of
  - To understand communications of
- Page fault performance is as good as or better than NFS/MNFS
- Overall performance is quite good
- Policies are easily tuned by the user, should they be so bold

Global Name Spaces

- Global Name Spaces are a security and administration nightmare

Machine A

- /adisk1
- /adisk2

Machine B

- /bdisk1
- /bdisk2

Process is started on Machine ‘A’ from Machine ‘B’ (assume that all machines are running an automounter, as is the typical case)

Machine A

- /adisk1
- /net/b/bdisk1
- /net/b/bdisk2

Machine B

- /bdisk1
- /net/b/bdisk1
- /net/b/bdisk2

Result: B has access to A’s disks; all processes on A have access to B’s disks (i.e., standard file access control). Machine B must export disks possibly across administrative boundaries. The name of B’s disks changes as the process moves from machine to machine.
Private Name Spaces

Machine A

- /adisk1
- /adisk2

Process 1 on Machine A

- /bdisk1

Process 1 on Machine B

Process is restarted on Machine ‘A’ from Machine ‘B’. Process has only /bdisk1 in its private name space.

Process 1 on Machine A

- /bdisk1

Processes on A

- /adisk1
- /adisk2

Process 1 sees /bdisk1 at same place in private name space. No process on machine A can access /bdisk1. Process 1 can not access any file system resources on Machine A.

Private Name Space Advantages

- User-level mount protocol
- Improved security as a result of reduced unintended sharing and fewer privileged processes
- Checkpoint/Restart is much simpler
- Processes need less access to system resources (e.g. file system) to access files
- No need to convince sysadmins to export resources across organizations

Current Implementation

- User-mode on some OSes, kernel mode client on Linux
- One server process per client connection
- Name space inheritance works both locally and on different machines
- Complete transparency (tested on Linux)

How to get it

- www.sarnoff.com:8000
- Go to the metacomputing page and look for the source
- Source is covered under the GNU Programming License
Problem: for reasonable numbers of nodes (say 64 or more) it takes too long to start up remote processes.

Ideally, ‘Start’ and ‘End’ are zero.

![Table](image)

Does this really matter

- Not usually for small numbers (8 or so)
- But: consider a 128-minute job

<table>
<thead>
<tr>
<th>Time</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procs</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>Over-head</td>
<td>7</td>
<td>3</td>
<td>1.5</td>
<td>.6</td>
</tr>
</tbody>
</table>

Need fast, efficient encapsulation of remote processors for scalable computing.

Need easy ways to specify aggregates.

---

Hostlists

- Efficient way to specify, select, and communicate with aggregates of remote hosts.
- Model is to create a hostlist, then apply filters to it.
- Filters can be specified by regular expressions on host names or by selecting idle hosts via vector RPC.
- Can generate code for static, initialized hostlists.
- At end of filter step(s), operate on the hostlist.

---

Basic Types

- A host:
  ```c
  struct host
  {
  char *hostname;
  struct sockaddr_in hostaddr;
  /* other good stuff */
  char *hosttype; /* quote string of the form "sun4c", "rs/6000", etc */
  int kbytes; /* number of kbytes of virtual memory */
  int gbytes; /* number of gbytes of physical memory */
  int avenrun[3]; /* from avenrun stats */
  int lastmouse; /* last mouse */
  int lastkbd; /* last keyboard */
  };
  ```

- A list of hosts:
  ```c
  struct hostlist
  {
  int numhosts;
  struct host **hostarr; /* array of pointers to host entries */
  };
  ```

- A hostlist can be sparse.
Basic functions

- Many return a hostlist
- Either:
  Take a hostlist as an arg, for filtering
  Filter to refine the hostlist, to select only certain hosts (via regex or load or ...)
  Add information to the list (e.g. given a set of names, fill in the IP addresses)

Netexec

- Used to manage aggregates of remote programs

```
struct netexec
{
    struct sockaddr_in s;
    char *hostname;
    char *argv, *envp;
    int arglen, argc;
    int envlen, envc;
    int results;
    int fd0, fd1, fd2;
};
```

Vector RPC

- Used to efficiently call RPC for aggregates of hosts
- hostlists library can generate vector RPC structures and perform vector RPC calls starting with a hostlist

Simple example: alloc a hostlist, filter by RE, run /bin/date on those hosts

```
struct hostlist *hl;
struct netexec *pn;
char *name, *pass, *cmd = "/bin/date";
char *linuxclusterhosts[] = {"c0[012345].", "c06[0123]", ... linuxclusterhosts);nprocs = hl->numhosts;res = rexecl(hl, &nx, &nprocs, &name, &pass, cmd, /* numcmds */ 1,
exec_port, use_priv_port, /* no stderr */ 0);
```
Details

- We only show one command can have arrays of commands e.g. run odd/even hosts w/different nprocs
- nprocs is filled in with successful procs
- nx is filled in with useable netexec struct
- We found with more than 40 or so machines that hostname lookup hurt scaling, so:

Compiled-in host-IP mappings

- Host->IP address lookup was killing scaleup
- IP addresses in most environments are static for years (esp. true for cluster)
- so:
  hl = hostlistfromhosts(); /* create a hostlist */
  selecthostbynamehost(hl, selections); /* select hosts we want */
  gencode(file, "allhosts", hl); /* gen code for inclusion at compile-time */

Gencode output

```c
struct host allhostshosts[] = {
    "p0", {2, 0, (0xb80a2182)}, "NOTYPE", 0, 0,
    {0, 0, 0}, 0, 0,
    "p1", {2, 0, (0xb90a2182)}, "NOTYPE", 0, 0,
    {0, 0, 0}, 0, 0,
    /* etc. */
    
    Typical use: hl = clonehl(allhostshosts);
    Then filter hl
    Reduces lookup time for 100 hosts from 5-12 seconds to "zero"
    Removes hostname lookup as a factor

VEX: making it easy

- VEX encapsulates hostlist and netexec in one object
- Code looks like this:
  vex = vexalloc();
  vexaddpplist(vex, argv[0], /* regular expression ? 1 */ 1);
  if (vexexec(vex) <= 0)  exit(1);
  while (vex->succ) /* vex->succ is number of active remote execs */
    {
      vexioloop(vex); /* handles stdin->remote and remote->stdout */
    }
```
Things we don’t do

- Signal propagation
  This is for efficient remote exec: we didn’t want to pay the overhead
  We can revisit this decision ...

Conclusions

- Successful cluster computing on a large scale requires low-overhead, process-oriented support systems
- Clusters aren’t multiprocessors
- We have shown three such systems
- Two (ZOUNDS, hostlist) have been in use for years for real work
- Private name spaces is just coming into use, but solves major problems we have experienced
- This is all open source