

---

# Integrating Scalable Process Management into Component- Based Systems Software

---

Rusty Lusk

(with Ralph Butler, Narayan Desai, Andrew Lusk)

Mathematics and Computer Science Division  
Argonne National Laboratory  
[lusk@mcs.anl.gov](mailto:lusk@mcs.anl.gov)



# Outline

---

- Context
  - Early clusters, PVM and MPI (MPICH), production clusters, evolving scale of systems software
- A component approach to systems software
  - The Scalable Systems Software Project
- Defining an abstract process management component
- A stand-alone process manager for scalable startup of MPI programs and other parallel jobs
  - MPD-2
- An MPD-based implementation of the abstract definition
- Experiments and experiences with MPD and SSS software on a medium-sized cluster

# Context

---

- This conference has accompanied, and contributed to, the growth of clusters from experimental to production computing resources
  - The first Beowulf ran PVM
  - Department-scale machines (often one or two apps)
  - Apps in both MPI and PVM
- Now clusters can be institution-wide computing resources
  - Many users and applications
  - Large clusters become central resources with competing users
  - Higher expectations
- Systems software is required for
  - Reliable management and monitoring (hardware and software)
  - Scheduling of resources
  - Accounting

# Current State of Systems Software for Clusters

---

- Both proprietary and open-source systems
  - PBS, LSF, POE, SLURM, COOAE (Collections Of Odds And Ends), ...
- Many are monolithic “resource management systems,” combining multiple functions
  - Job queuing, scheduling, process management, node monitoring, job monitoring, accounting, configuration management, etc.
- A few established separate components exist
  - Maui scheduler
  - Qbank accounting system
- Many home-grown, local pieces of software
- Process Management often a weak point

# Typical Weaknesses of Process Managers

---

- Process startup not scalable
- Process startup not even parallel
  - May provide list of nodes and just start script on first one
  - Leaves application to do own process startup
- Parallel process startup may be restricted
  - Same executable, command-line arguments, environment
- Inflexible and/or non-scalable handling of stdin, stdout, stderr.
- Withholds useful information from parallel library
  - Doesn't help parallel library processes find one another
- No particular support for tools
  - Debuggers, profilers, monitors
- And they are all different!

# Background – The MPD Process Manager

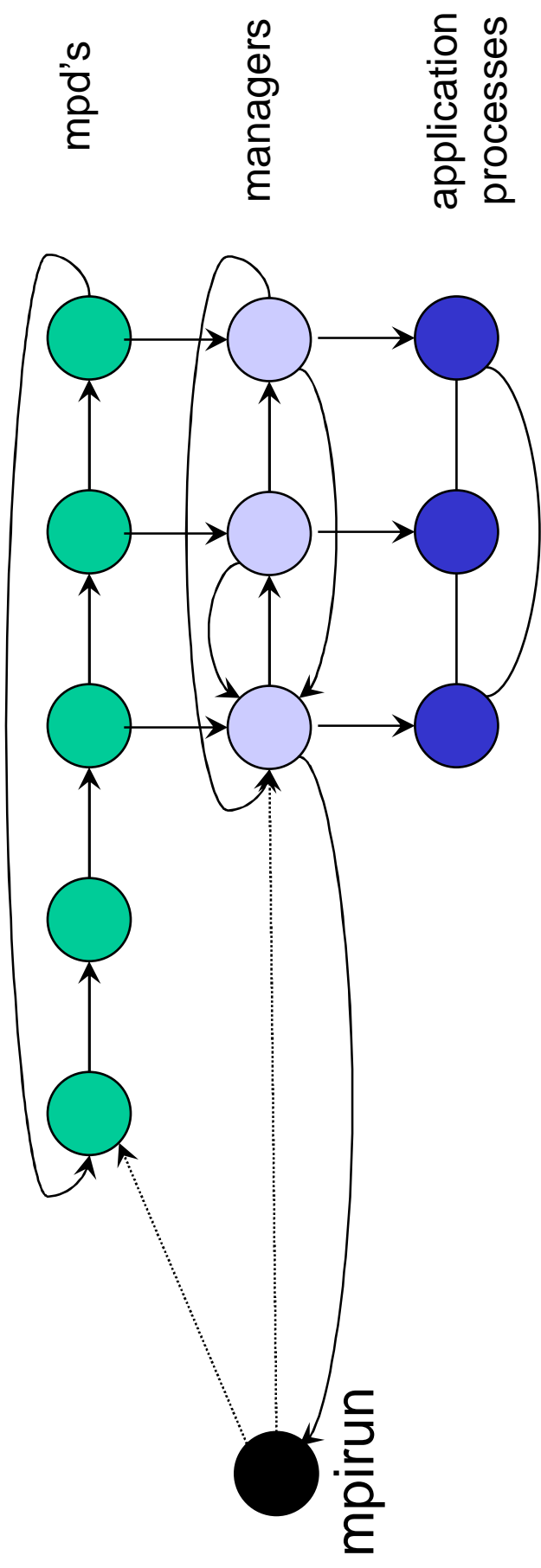
---

- Described at earlier EuroPVM/MPI conferences
- Primary research goals:
  - Fast and scalable startup of parallel jobs (especially MPICH)
  - Explore interface needed to support MPI and other parallel libraries
    - Helping processes locate and connect to other processes in job, in scalable way (the BNR interface)
- Part of MPICH-1
  - `ch_p4mpd` device
- Established that MPI job startup could be very fast
  - Encouraged interactive parallel jobs
  - Allowed some system programs (e.g. file staging) to be written as MPI programs (See Scalable Unix Tools, EuroPVM/MPI-8)

# MPD-1

---

## Architecture of MPD:



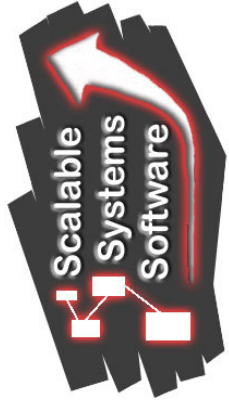
# Recent Developments

---

- Clusters get bigger, providing a greater need for scalability
- Large clusters serve many users
  - Many issues the same for “non-cluster” machines
- MPI-2 functionality puts new demands on process manager
  - `MPI_Comm_spawn`
  - `MPI_Comm_connect`, `MPI_Comm_accept`, `MPI_Comm_join`
- MPICH-2 provides opportunity to redesign library/process manager interface
- Scalable Systems Software SciDAC project presents an opportunity to consider Process Manager as a separate component participating in a component-based systems software architecture
- New requirements for systems software on research cluster at Argonne

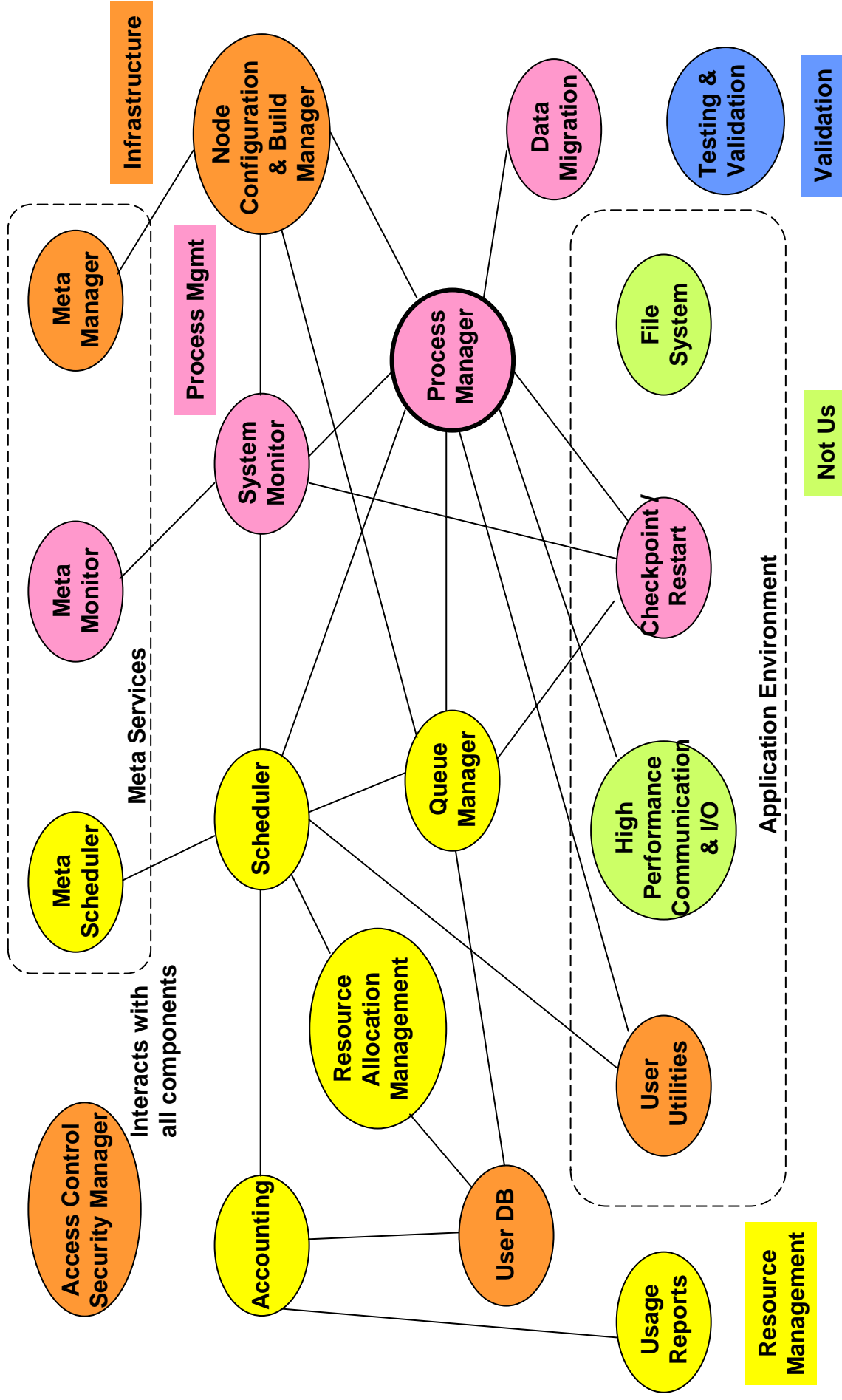
# The Scalable Systems Software SciDAC Project

---



- Multiple Institutions (most national labs, plus NCSA)
- Research goal: to develop a component-based architecture for systems software for scalable machines
- Software goal: to demonstrate this architecture with some prototype open-source components
- One powerful effect: forcing rigorous (and aggressive) definition of what a process manager should do and what should be encapsulated in other components
- <http://www.scidac.org//ScalableSystems>

# System Software Components



# Defining Process Management in the

## Abstract

---

- Define functionality of process manager component
- Define interfaces by which other components can invoke process management services
- Try to avoid specifying how system will be managed as a whole
- Start by deciding what should be included and not included

# Not Included

---

- **Scheduling**
  - Another component will either make scheduling decisions (selection of hosts, time to run), or explicitly leave host selection up to process manager
- **Queueing**
  - A job scheduled to run in the future will be maintained by another component; the process manager will start jobs immediately
- **Node monitoring**
  - The state of a node is of interest to the scheduler, which can find this out from another component
- **Process monitoring**
  - CPU usage, memory footprint, etc, are attributes of individual processes, and can be monitored by another component. The process manager can help by providing job information (hosts, pids)
- **Checkpointing**
  - Process manager can help with signals, but CP is not its job

# Included

---

- Starting a parallel job
  - Can specify multiple executables, arguments, environments
- Handling studio
  - Many options
- Starting co-processes
  - Tools such as debuggers and monitors
- Signaling a parallel job
- Killing a parallel job
- Reporting details of a parallel job
- Servicing the parallel job
  - Support MPI implementation, other services
- In context of Scalable Systems Software suite, register so that other components can find it, and report events

# The SSS Process Manager

---

- Provides previously-listed functions
- Communicates with other SSS components using XML messages over sockets (like other SSS components do)
- Defines syntax and semantics of specific messages:
  - Register with service directory
  - Report events like job start and termination
  - Start job
  - Return information on a job
  - Signal job
  - Kill job
- Uses MPD-2 to carry out its functions

# Second-Generation MPD

---

- Same basic architecture as MPD-1
- Provides new functionality required by SSS definition
  - E.g., separate environment variables for separate ranks
- Provides new interface for parallel library like MPICH-2
  - PMI interface extends, improves, generalizes BNR
    - Multiple key-val spaces
    - Put/get/fence interface for scalability
    - Spawn/accept/connect at low level to support MPI-2 functions
- Maintains scalability features of MPD
- Improved fault-tolerance

# Testing the MPD Ring

---

- Here the ring of MPD's had 206 hosts
- Simulated larger ring by sending message around ring multiple times

Times around the ring	Time in seconds
1	.13
10	.89
100	8.93
1000	89.44

- Linear, as expected
- But fast: > 2000 hops/sec

# Running Non-MPI Jobs

---

- Ran hostname on each node
- Creates stdio tree and collects output from each node
- Sublinear

Number of hosts	Time in seconds
1	.83
4	.86
8	.92
16	1.06
32	1.33
64	1.80
128	2.71
192	3.78

# Running MPI Jobs

---

- Ran cpi on each node (includes I/O, Bcast, Reduce)
- Compared MPICH-1 (ch\_p4 device) with MPICH-2 with MPD-2
- Better!

Number of Processes	Old Time	New Time
1	.4	.63
4	5.6	.67
8	14.4	.73
16	30.9	.86
32	96.9	1.01
64		1.90
128		3.50

# SSS Project Issues

---

- Put minimal constraints on component implementations
  - Ease merging of existing components into SSS framework
    - E.g., Maui scheduler
  - Ease development of new components
  - Encourage multiple implementations from vendors, others
- Define minimal global structure
  - Components need to find one another
  - Need common communication method
  - Need common data format at some level
    - Each component will compose messages others will read and parse
  - Multiple message-framing protocols allowed

# SSS Project Status – Global

---

- Early decisions on inter-component communication
  - Lowest level communication is over sockets (at least)
  - Message content will be XML
    - Parsers available in all languages
  - Did not reach consensus on transport protocol (HTTP, SOAP, BEEP, assorted home grown), especially to cope with local security requirements
- Early implementation work on global issues
  - Service directory component defined and implemented
  - SSSlib library for inter-component communication
    - Handles interaction with SD
    - Hides details of transport protocols from component logic
    - Anyone can add protocols to the library
    - Bindings for C, C++, Java, Perl, and Python

# SSS Project Status – Individual Component Prototypes

---

- Precise XML interfaces not settled on yet, pending experiments with component prototypes
- Both new and existing components
- Maui scheduler is existing full-featured scheduler, having SSS communication added
- QBank accounting system is adding SSS communication interface
- New Checkpoint Manager component being tested now
  - System-initiated checkpoints of LAM jobs

# SSS Project Status – More Individual Component Prototypes

---

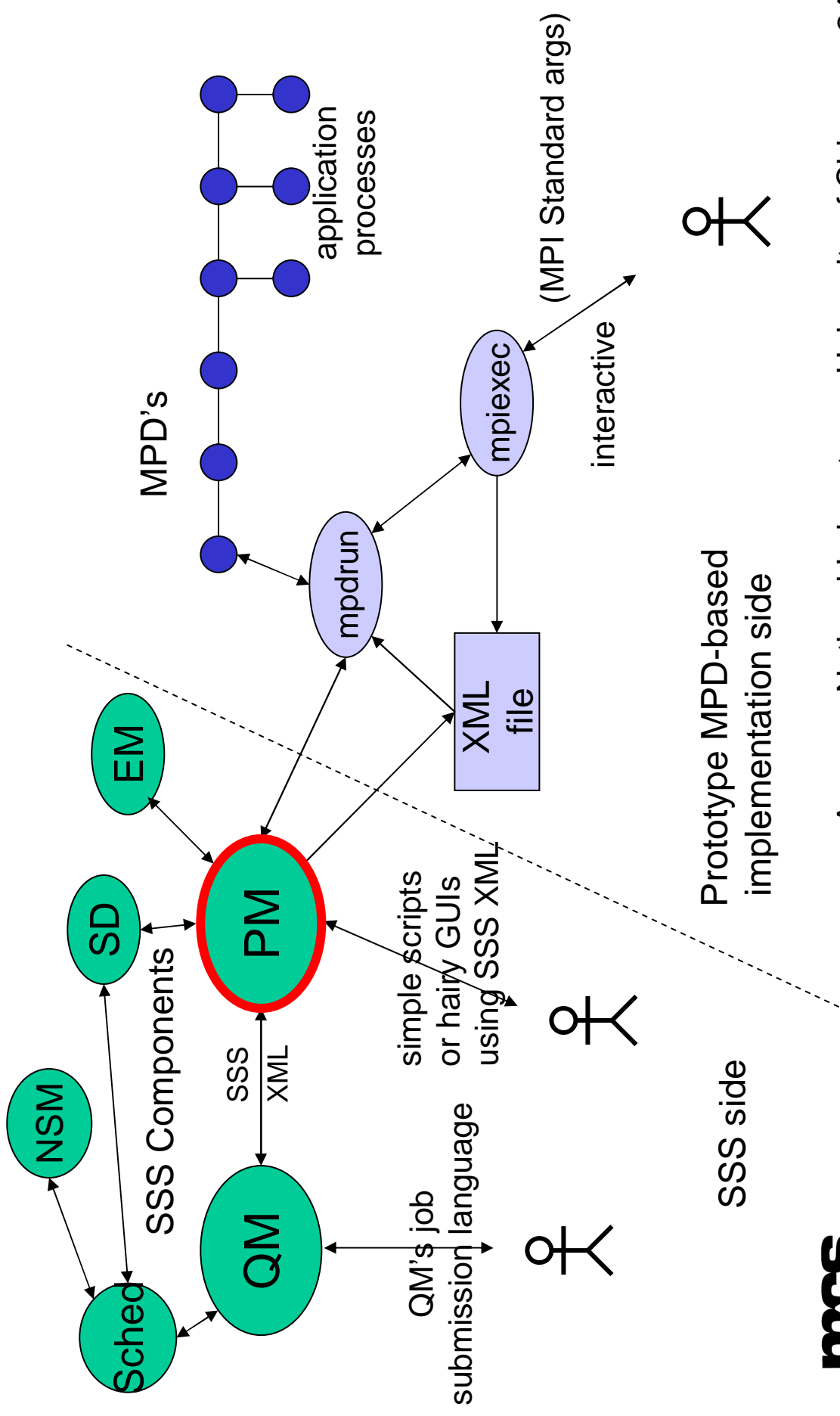
- New Build-and-Configuration Manager completed
  - Controls how nodes are, well, configured and built
- New Node State Manager
  - Manages nodes as they are installed, reconfigured, added to active pool
- New Event Manager for asynchronous communication among components
  - Components can register for notification of events supplied by other components
- New Queue Manager mediates among user (job submitter), Job Scheduler, and Process Manager
- Multiple monitoring components, both new and old

# SSS Project Status – Still More Individual Component Prototypes

---

- New Process Manager component provides SSS interface to MPD-2 process manager
  - Speaks XML through SSSlib to other SSS components
  - Invokes MPD-2 to implement SSS process management specification
  - MPD-2 itself is not an SSS component
  - Allows MPD-2 development, especially with respect to supporting MPI and MPI-2, to proceed independently
  - SSS Process Manager abstract definitions have influenced addition of MPD-2 functionality beyond what is needed to implement **mpiexec** from MPI-2 standard
  - E.g. separate environment variables for separate processes

# Schematic of Process Management Component in Scalable Systems Software Context



# Chiba City

---

- Medium-sized cluster at Argonne National Laboratory
  - 256 dual-processor 500MHz PIII's
  - Myrinet
  - Linux (and sometimes others)
  - No shared file system, for scalability
- Dedicated to Computer Science scalability research, not applications
- Many groups use it as a research platform
  - Both academic and commercial
- Also used by friendly, hungry applications
- New requirement: support research requiring specialized kernels and alternate operating systems, for OS scalability research

# New Challenges

---

- Want to schedule jobs that require node rebuilds (for new OS's, kernel module tests, etc.) as part of “normal” job scheduling
- Want to build larger virtual clusters (using VMware or User Mode Linux) temporarily, as part of “normal” job scheduling
- Requires major upgrade of Chiba City systems software

# Chiba Commits to SSS

---

- Fork in the road:
  - Major overhaul of old, crufty, Chiba systems software (open PBS + Maui scheduler + homegrown stuff), OR
  - Take leap forward and bet on all-new software architecture of SSS
- Problems with leaping approach:
  - SSS interfaces not finalized
  - Some components don't yet use library (implement own protocols in open code, not encapsulated in library)
  - Some components not fully functional yet
- Solutions to problems:
  - Collect components that are adequately functional and integrated (PM, SD, EM, BCM)
  - Write “stubs” for other critical components (Sched, QM)
  - Do without some components (CKPT, monitors, accounting) for the time being

# Features of Adopted Solution

---

- Stubs quite adequate, at least for time being
  - Scheduler does FIFO + reservations + backfill, improving
  - QM implements “PBS compatibility mode” (accepts user PBS scripts) as well as asking Process Manager to start parallel jobs directly
- Process Manager wraps MPD-2, as described above
  - Single ring of MPD’s runs as root, managing all jobs for all users
  - MPD’s started by Build-and-Config manager at boot time
- An MPI program called MPIISH (MPI Shell) wraps user jobs for handling file staging and multiple job steps
- Python implementation of most components
- Demonstrated feasibility of using SSS component approach to systems software
  - Running normal Chiba job mix for over a month now
  - Moving forward on meeting new requirements for research support

# Summary

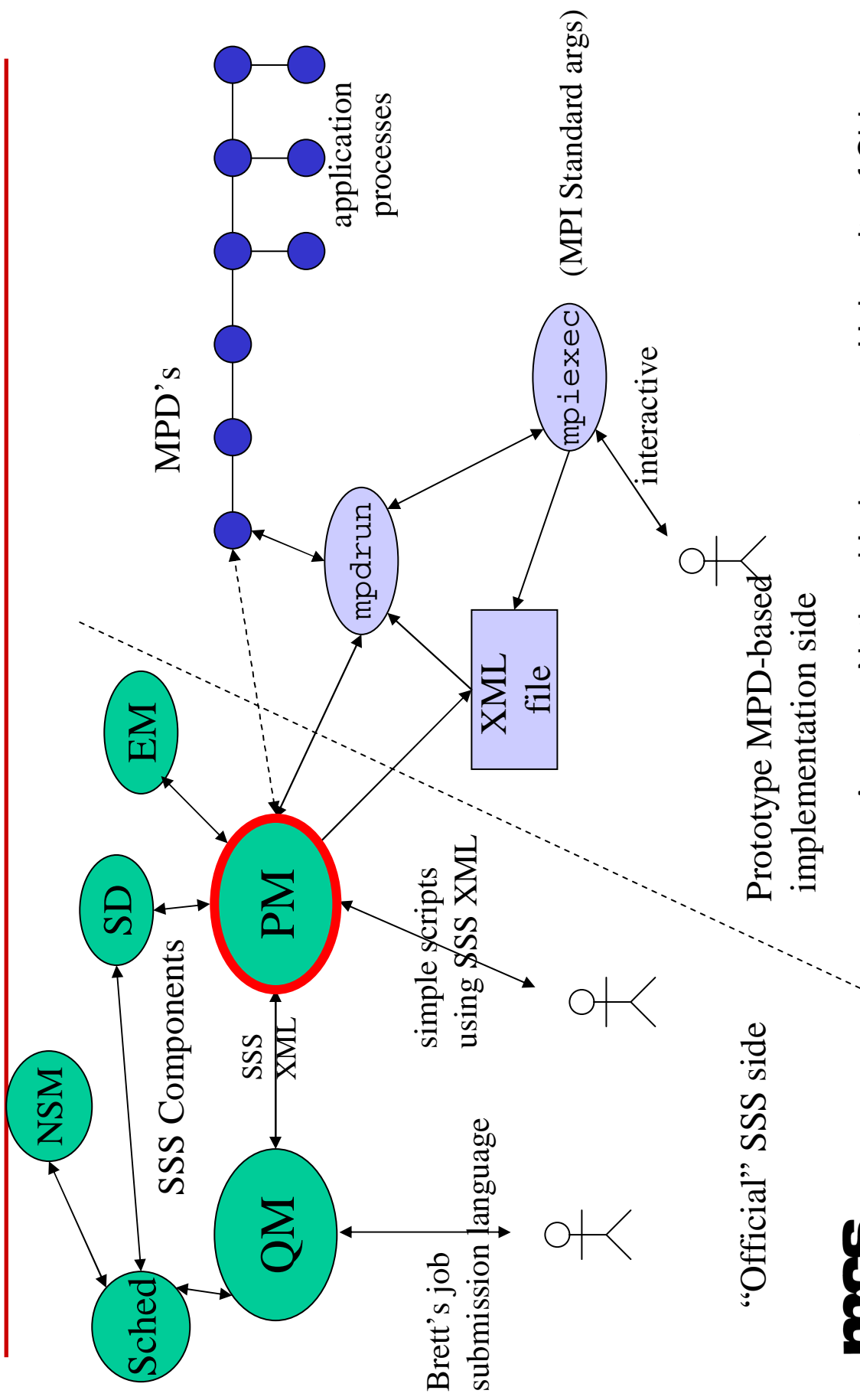
---

- Scalable process management is a challenging problem, even just from the point of view of starting MPI jobs
- Designing an abstract process management component as part of a complete system software architecture helped refine the precise scope of process management
- Original MPD design was adopted to provide core functionality of an SSS process manager without giving up independence (can still start MPI jobs with `mpirexec`, without using SSS environment)
- This Process Manager, together with other SSS components, has demonstrated the feasibility and usefulness of a component-based approach to advanced systems software for clusters and other parallel machines.

# Beginning of Meeting slides

---

# Schematic of Process Management Component in Context



# How should we proceed?

---

- Proposal: voting should actually be on an explanatory document that includes
  - Descriptions – text and motivations
  - Examples – for each type of message, both simple and complicated
  - Details – XML schemas
- What follows is just input to this process

# The Process Manager Interface

---

- The “other end” of interfaces to other components
  - Service Directory
  - Event Manager
- The commands supported, currently tested by interaction with both the SSS Queue Manager and standalone interactive scripts
  - Create-process-group
  - Kill-process-group
  - Signal-process-group
  - Get-process-group-info
  - Del-process-group-info
  - Checkpoint-process-group

# Some Examples - 1

---

```
<create-process-group submitter='desai' totalprocs='32'  
  output='discard'>  
  <process-spec exec='/bin/foo' cwd='/etc' path='/usr/sbin'  
    range='1-32' co-process='tv-server'>  
    <arg idx='1' value='-v' />  
  </process-spec>  
  <host-spec>  
    node1  
    node2  
  </host-spec>  
</create-process-group>  
  
yields:  
  
<process-group pgid='1' />
```

# Some Examples - 2

---

```
<get-process-group-info>
  <process-group pgid='1' />
</get-process-group-info>
```

**yields:**

```
<process-groups>
  <process-group submitter="desai" pgid='1' totalprocs="2" >
    <process-spec cwd="/home/desai/dev/sss/clients"
      exec="/bin/hostname"
      path="/opt/bin:/home/desai/bin:/opt/bin:/usr/local/bin:
        /usr/bin/X11:/usr/games" />
  </process-spec>
</process-groups>

<host-spec>
  topaz
  topaz
</host-spec>

<output>
  topaz
  topaz
</output>

</process-group>
</process-groups>
```

# Some Examples - 3

---

Things like signal and kill process group work the same:

```
<kill-process-group>  
  <process-group pgid='1' submitter='*' />  
</kill-process-group>  
  
yields  
  
<process-groups>  
  <process-group pgid='1' submitter='desai' />  
</process-groups>
```

# Input Schema - 1

---

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xml:lang="en">
  <xsd:annotation>
    <xsd:documentation>
      Process Manager component inbound schema
      SciDAC SSS project, 2002 Andrew Lusk alusk@mcs.anl.gov
    </xsd:documentation>
  </xsd:annotation>

  <xsd:include schemaLocation="pm-types.xsd"/>

  <xsd:complexType name="createpgType">
    <xsd:choice minOccurs="1" maxOccurs="unbounded">
      <xsd:element name="process-spec" type="pg-spec"/>
      <xsd:element name="host-spec" type="xsd:string"/>
    </xsd:choice>
    <xsd:attribute name="submitter" type="xsd:string" use="required"/>
    <xsd:attribute name="totalprocs" type="xsd:string" use="required"/>
    <xsd:attribute name="output" type="xsd:string" use="required"/>
  </xsd:complexType>
```

# Input Schema - 2

---

```
<xsd:element name="create-process-group" type="createPgType"/>

<xsd:element name="get-process-group-info" >
  <xsd:complexType>
    <xsd:choice minOccurs="1" maxOccurs="unbounded" >
      <xsd:element name="process-group" type="pgRestrictionType"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

<xsd:element name="del-process-group-info" >
  <xsd:complexType>
    <xsd:choice minOccurs="1" maxOccurs="unbounded" >
      <xsd:element name="process-group" type="pgRestrictionType"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

# Input Schema - 3

---

```
<xsd:element name="signal-process-group" >
  <xsd:complexType>
    <xsd:choice minOccurs="1" maxOccurs="unbounded" >
      <xsd:element name="process-group" type="pgRestrictionType"/>
    </xsd:choice>
    <xsd:attribute name="signal" type="xsd:string" use="required"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="kill-process-group" >
  <xsd:complexType>
    <xsd:choice minOccurs="1" maxOccurs="unbounded" >
      <xsd:element name="process-group" type="pgRestrictionType"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

<xsd:element name="checkpoint-process-group" >
  <xsd:complexType>
    <xsd:choice minOccurs="1" maxOccurs="unbounded" >
      <xsd:element name="process-group" type="pgRestrictionType"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

# Output Schema - 1

---

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xml:lang="en" >
  <xsd:annotation>
    <xsd:documentation>
      Process Manager component outbound schema
      SciDAC SSS project, 2002 Andrew Lusk alusk@mcs.anl.gov
    </xsd:documentation>
  </xsd:annotation>

  <xsd:include schemaLocation="pm-types.xsd"/>
  <xsd:include schemaLocation="sss-error.xsd"/>

  <xsd:element name="process-groups">
    <xsd:complexType>
      <xsd:choice minOccurs='0' maxOccurs='unbounded'>
        <xsd:element name="process-group" type="pgType"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="process-group" type="pgRestrictionType"/>
  <xsd:element name="error" type="SSSError"/>
</xsd:schema>
```

# Types Schema - 1

---

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xml:lang="en" >
  <xsd:annotation>
    <xsd:documentation>
      Process Manager component schema
      SciDAC SSS project, 2002 Andrew Lusk alusk@mcs.anl.gov
    </xsd:documentation>
  </xsd:annotation>

  <xsd:complexType name="argType" >
    <xsd:attribute name="idx" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="envType" >
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string" use="required"/>
  </xsd:complexType>
```

# Types Schema - 2

---

```
<xsd:complexType name="pg-spec">
  <xsd:choice minOccurs='0' maxOccurs='unbounded'>
    <xsd:element name="arg" type="argType"/>
    <xsd:element name="env" type="envType"/>
  </xsd:choice>
  <xsd:attribute name="range" type="xsd:string"/>
  <xsd:attribute name="user" type="xsd:string"/>
  <xsd:attribute name="co-process" type="xsd:string"/>
  <xsd:attribute name="exec" type="xsd:string" use="required"/>
  <xsd:attribute name="cwd" type="xsd:string" use="required"/>
  <xsd:attribute name="path" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="procType">
  <xsd:attribute name="host" type="xsd:string" use="required"/>
  <xsd:attribute name="pid" type="xsd:string" use="required"/>
  <xsd:attribute name="exec" type="xsd:string" use="required"/>
  <xsd:attribute name="session" type="xsd:string" use="required"/>
</xsd:complexType>
```

# Types Schema - 3

---

```
<xsd:complexType name="procRestrictionType">
  <xsd:attribute name="host" type="xsd:string"/>
  <xsd:attribute name="pid" type="xsd:string"/>
  <xsd:attribute name="exec" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="pgType">
  <xsd:choice minOccurs="1" maxOccurs="unbounded">
    <xsd:element name="process" type="procType"/>
  </xsd:choice>
  <xsd:choice minOccurs="0" maxOccurs="1">
    <xsd:element name='output' type='xsd:string' />
  </xsd:choice>
  <xsd:attribute name="pgid" type="xsd:string" use="required"/>
  <xsd:attribute name="submitter" type="xsd:string" use="required"/>
  <xsd:attribute name="totalprocs" type="xsd:string" use="required"/>
  <xsd:attribute name="output" type="xsd:string" use="required"/>
</xsd:complexType>
```

# Types Schema - 4

---

```
<xsd:complexType name="pgRestrictionType" >
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="process" type="procRestrictionType"/>
  </xsd:choice>
  <xsd:attribute name="pgid" type="xsd:string"/>
  <xsd:attribute name="submitter" type="xsd:string"/>
  <xsd:attribute name="totalprocs" type="xsd:string"/>
</xsd:complexType>
</xsd:schema>
```

# Error Schema - 1

---

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xml:lang="en" >
  <xsd:annotation>
    <xsd:documentation>
      Service Directory error schema
      ScIDAC SSS project
      2003 Narayan Desai desai@mcs.anl.gov
    </xsd:documentation>
  </xsd:annotation>
  <xsd:simpleType name="ErrorType" >
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="Validation|Semantic|Data"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="SSSError" >
    <xsd:attribute name="type" type="ErrorType" use="optional"/>
  </xsd:complexType>
</xsd:schema>
```

# Beginning of Linz slides

---

# Outline

---

- Scalable process management
  - What is process management and where does it fit in with systems software and middleware architecture?
  - An experimental scalable process management system: MPD
- Some new directions
  - Process management in context of Scalable System Software Project
    - The SSS project: components and interfaces
    - The Process Management component
      - Role of MPD
  - Process management and tools
    - How process management can help tools
    - Some examples

# Outline (cont.)

---

- New activities in scalable process management (cont.)
  - **Formal Verification Techniques and MPD**
    - ACL2
    - SPIN/Promela
    - Otter theorem proving
  - **Scalable Process Management in upcoming large-scale systems**
    - YOD/PMI/MPICH on ASCI Red Storm at Sandia
    - MPD as process manager for IBM's BG/L

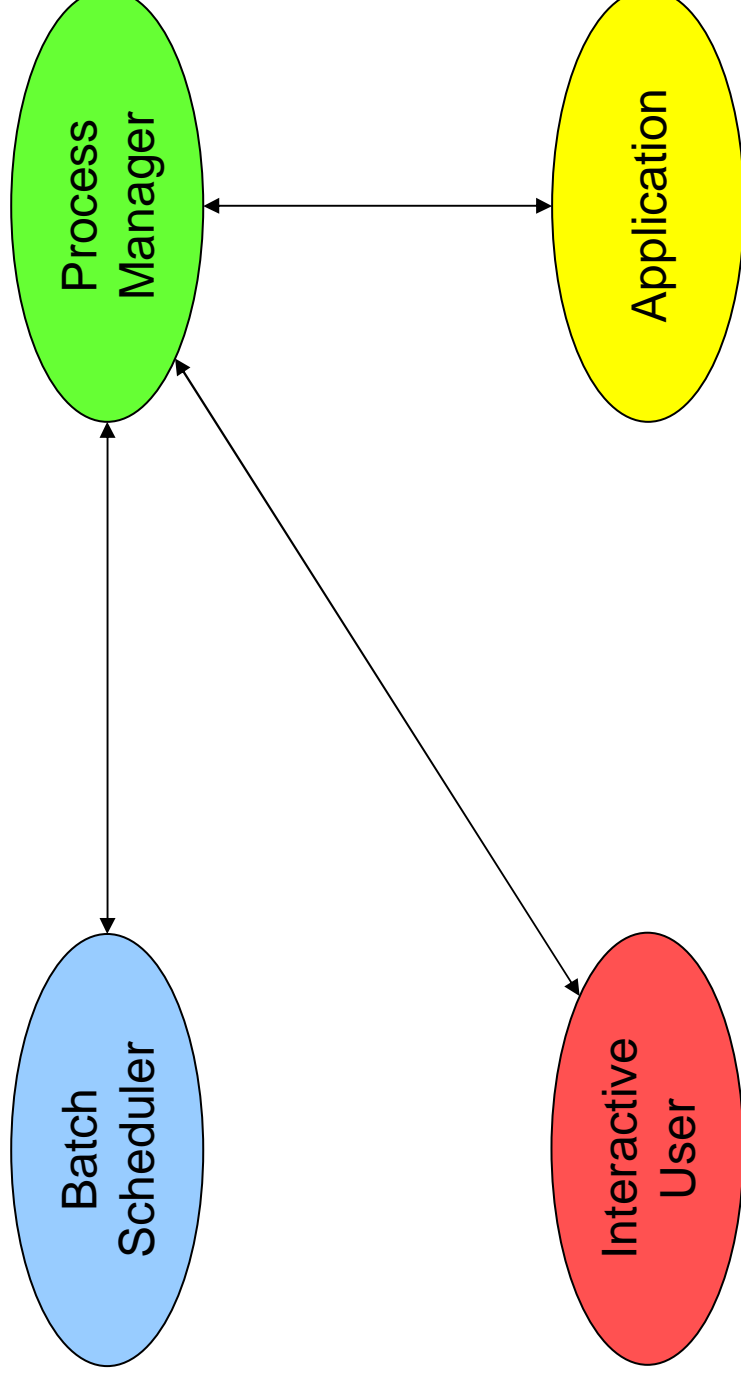
# What is Process Management?

---

- A *process management system* is the software that starts user processes (with command line arguments and environment), ensures that they terminate cleanly, and manages I/O
- For simple jobs, this can be the shell
- For parallel jobs, more is needed
- Process management is different from scheduling, queuing, and monitoring

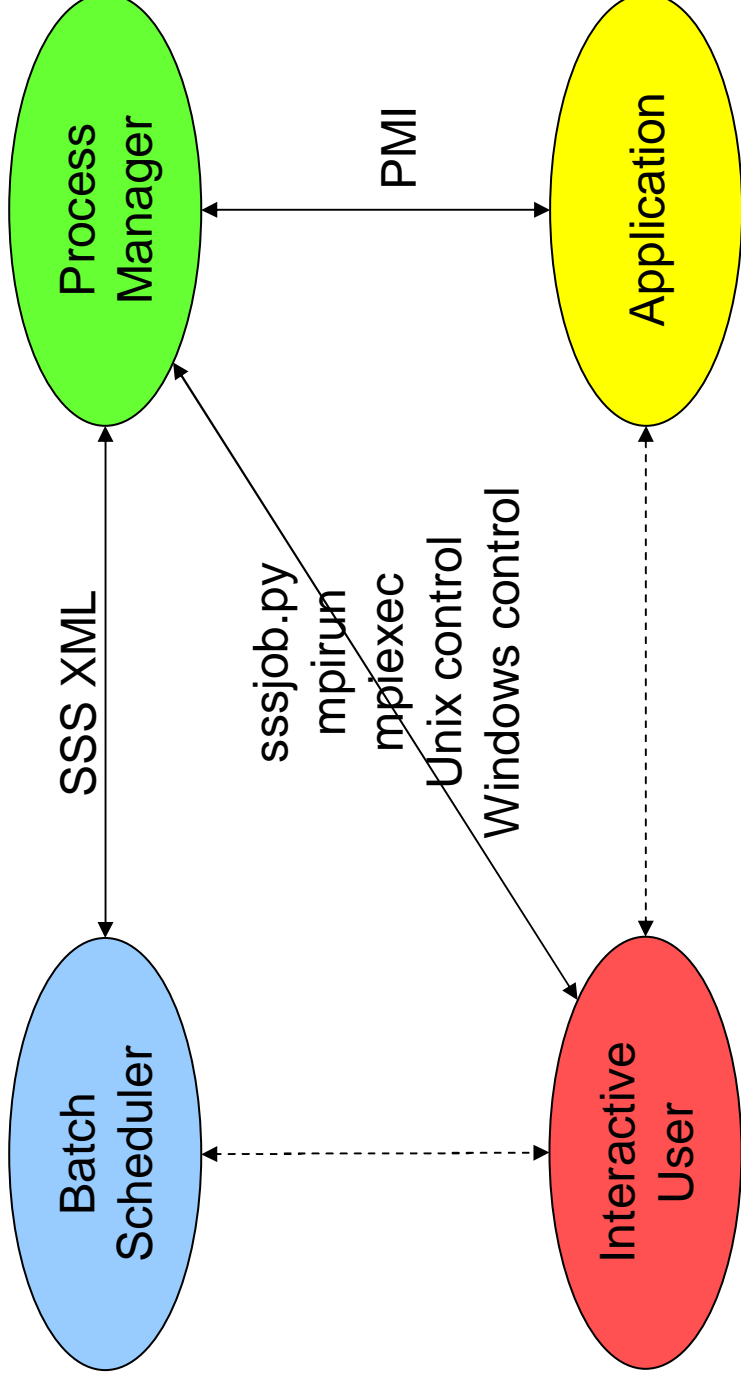
# The Three “Users” of a Process Manager

---



# Interfaces Are the Key

---



# Process Manager Research Issues

---

- Identification of proper process manager functions
  - Starting (with arguments and environment), terminating, signaling, handling stdio, ...
- Interface between process manager and communication library
  - Process placement and rank assignment
  - Dynamic connection establishment
  - MPI-2 functionality: Spawn, Connect, Accept, Singleton Init
- Interface between process manager and rest of system software
  - Cannot be separated from system software architecture in general
  - Process manager is important component of component-based architecture for system software, communicating with multiple other components
- Scalability
  - A problem even on existing large systems
  - Some new systems coming present new challenges
  - Interactive jobs (such as Scalable Unix Tools) need to start fast

# Requirements on Process Manager from Message-Passing Library

---

- Individual process requirements
  - Same as for sequential job
  - To be brought into existence
  - To receive command-line arguments
  - To be able to access environment variables
- Requirements derived from being part of a parallel job
  - Find size of job: `MPI_Comm_size( MPI_COMM_WORLD, &size )`
  - Identify self: `MPI_Comm_rank( MPI_COMM_WORLD, &myrank )`
  - Find out how to contact other processes: `MPI_Send( ... )`

# Finding the Other Processes

---

- Need to identify one or several ways of making contact
  - Shared memory (queue pointer)
  - TCP (host and port for connect)
  - Other network addressing mechanisms (Infiniband)
  - (x,y,z) torus coordinates in BG/L
- Depends on target process
- Only process manager knows where other processes are
- Even process manager might not know everything necessary (e.g. dynamically obtained port)
- “Business Card” approach

# Approach

---

- Define interface from parallel library (or application) to process manager
  - Allows multiple implementations
  - MPD is a scalable implementation (used in MPICH ch\_p4mpd device)
- PMI (Process Manager Interface)
  - Conceptually: access to spaces of key=value pairs
  - No reserved keys
  - Allows very general use, in addition to “business card”
  - Basic part: for MPI-1, other simple message-passing libraries
  - Advanced part: multiple keyval spaces for MPI-2 functionality, grid software
- Provide scalable PMI implementation with fast process startup
- Let others do so too

# The PMI Interface

---

- PMI\_Init
- PMI\_Get\_size
- PMI\_Get\_rank
- PMI\_Put
- PMI\_Get
- PMI\_Fence
- PMI\_End
  
- More functions for managing multiple keyval spaces
  - Needed to support MPI-2, grid applications

# Multiple PMI Implementations

---

- MPD
  - MPD-1, in C, distributed in MPICH 1.2.4 (ch\_p4mpd device)
  - MPD-2, in Python, part of MPICH-2, matches Scalable System Software Project requirements
- “Forker” for MPICH-2 code development
  - mpirun forks the MPI processes
  - Fast and handy for development and debugging on a single machine
- WinMPD on Windows systems
  - NT and higher, uses single keyval space server
- Others possible (YOD?)
  - Clean way for system software implementors to provide services needed by MPICH, other libraries

# Process Manager Research at ANL

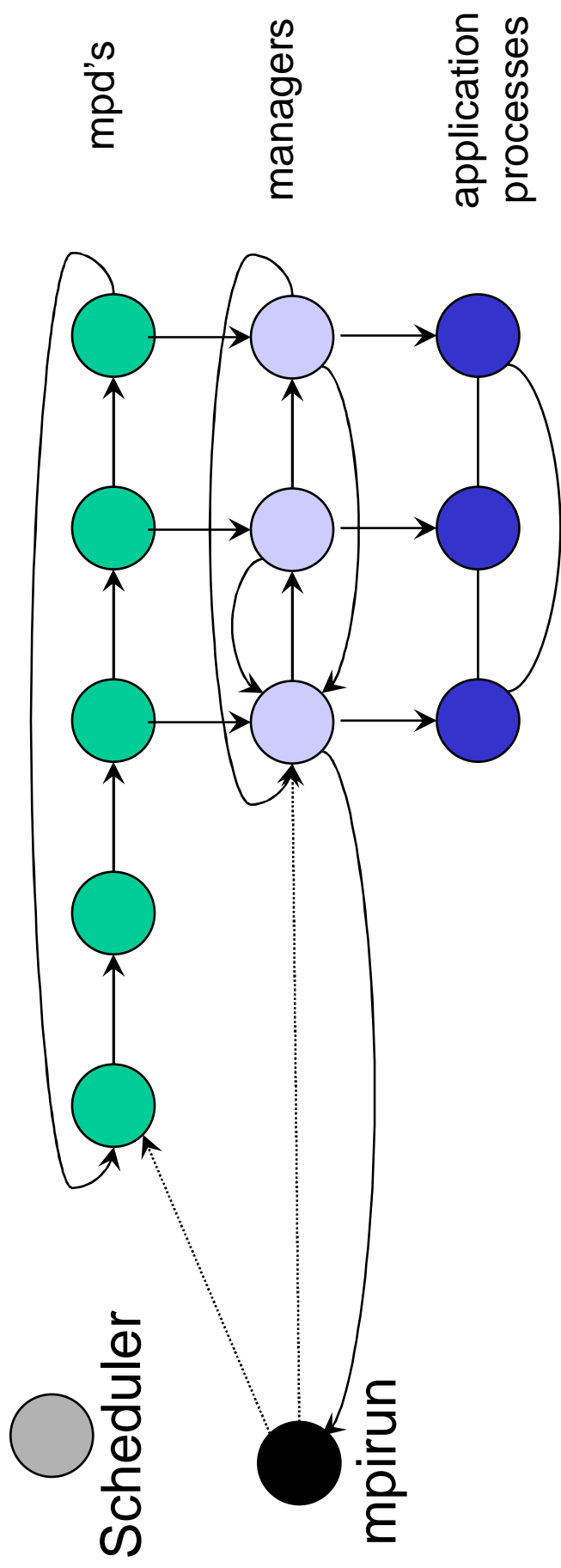
---

- MPD – prototype process management system
- Original Motivation: faster startup of interactive MPICH programs
- Evolved to explore general process management issues, especially in the area of communication between process manager and parallel library
- Laid foundation for scalable system software research in general
- MPD-1 is part of current MPICH distribution
  - Much faster than earlier schemes
  - Manages stdio scalably
  - Tool-friendly (e.g. supports TotalView)

# MPD

---

## Architecture of MPD:



# Interesting Features

---

- **Security**
  - “Challenge-response” system, using passwords in protected files and encryption of random numbers
  - Speed not important since daemon startup is separate from job startup
- **Fault Tolerance**
  - When a daemon dies, this is detected and the ring is reknit => minimal fault tolerance
  - New daemon can be inserted in ring
- **Signals**
  - Signals can be delivered to clients by their managers

# More Interesting Features

---

- Uses of signal delivery
  - signals delivered to a job-starting console process are propagated to the clients
    - so can suspend, resume, or kill an mpirun
    - one client can signal another
    - can be used in setting up connections dynamically
    - a separate console process can signal currently running jobs
    - can be used to implement a primitive gang scheduler
- Mpirun also represents parallel job in other ways
  - totalview mpirun -np 32 a.out
  - runs 32-process job under TotalView control

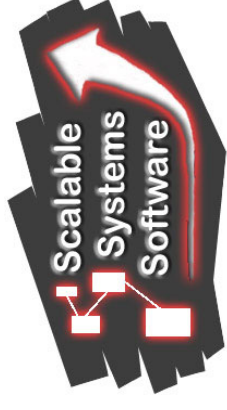
# More Interesting Features

---

- Support for parallel libraries
  - implements the **PMI process manager interface**, used by **MPICH**.
  - Distributed keyval spaces maintained in the managers
  - put, get, fence, spawn
  - solves “pre-communication” problem of startup
  - makes MPD independent from MPICH while still providing needed features

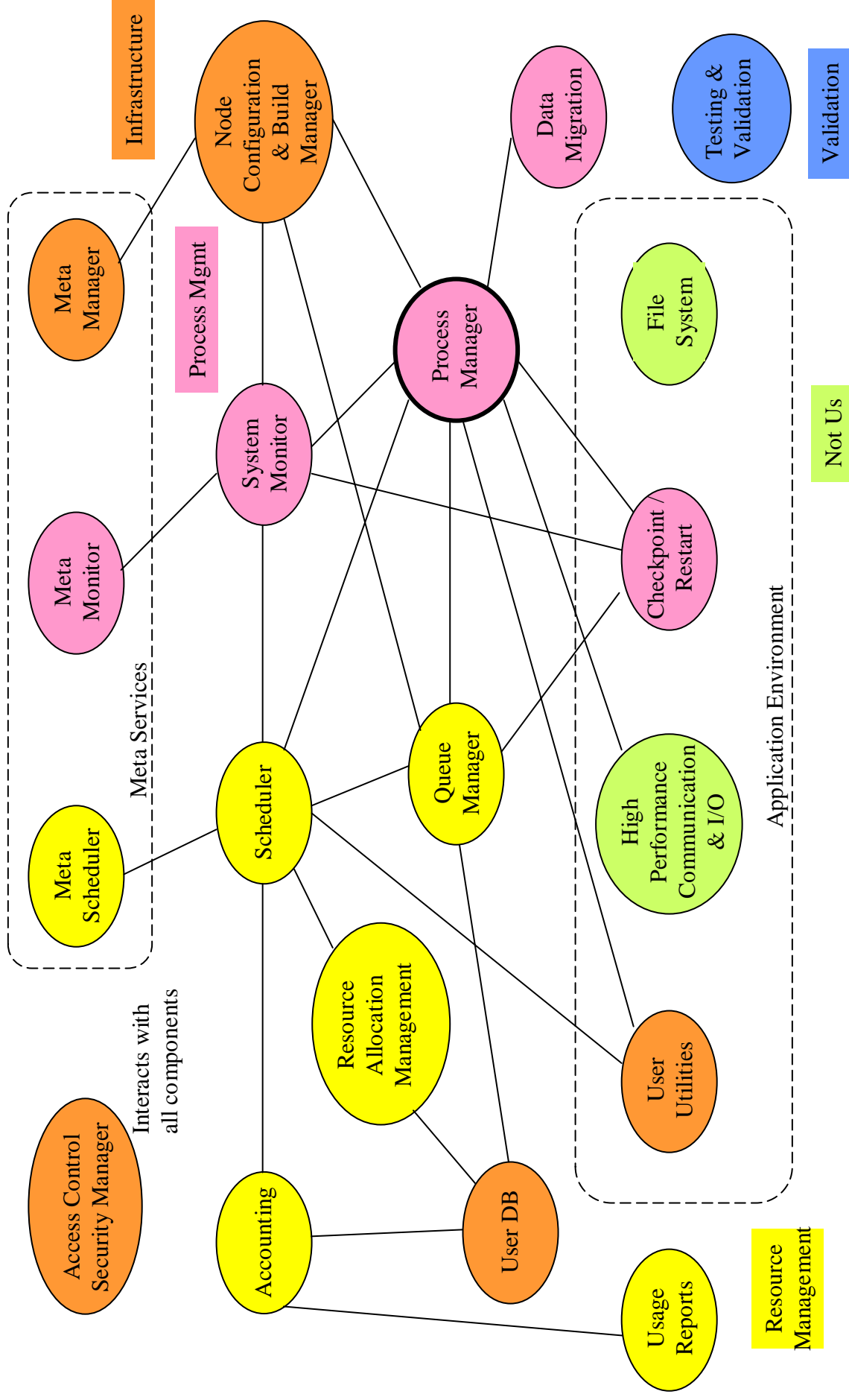
# The Scalable Systems Software SciDAC Project

---



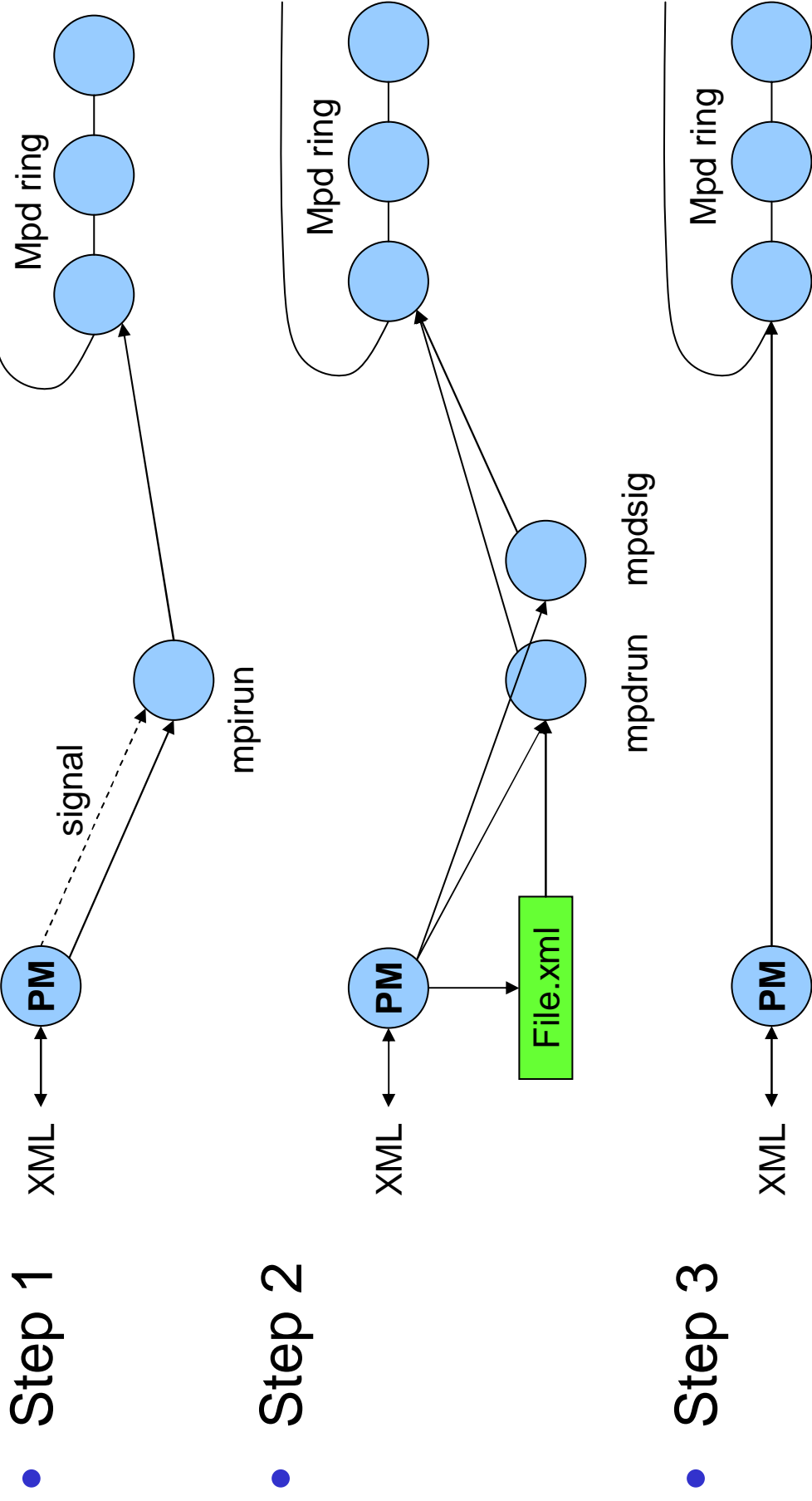
- Multiple Institutions (most national labs, plus NCSA)
- Research goal: to develop a component-based architecture for systems software for scalable machines
- Software goal: to demonstrate this architecture with some prototype components
- Currently using XML for inter-component communication
- Status
  - Inter-component communication library released across project, some components in use at Argonne on Chiba City cluster
  - Detailed XML interfaces to several components
- One powerful effect: forcing rigorous (and aggressive) definition of what a process manager should do and what should be encapsulated in other components
  - Start (with arguments and environment variables), terminate, cleanup
  - Signal delivery
  - Interactive support (e.g. for debugging) – requires studio management
- <http://www.scidac.org//ScalableSystems>

# System Software Components



# Using MPD as a Prototype Project Component

---



# Process Management and Tools

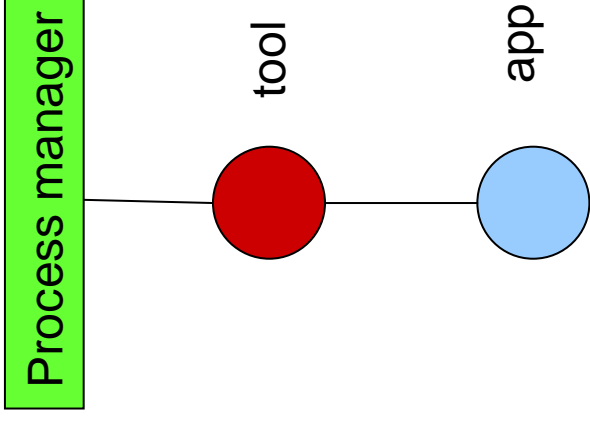
---

- Tools (debuggers, performance monitors, etc.) can be helped by interaction with process manager
- Multiple types of relationship

## (At Least) Three Ways Tool Processes Fit In

---

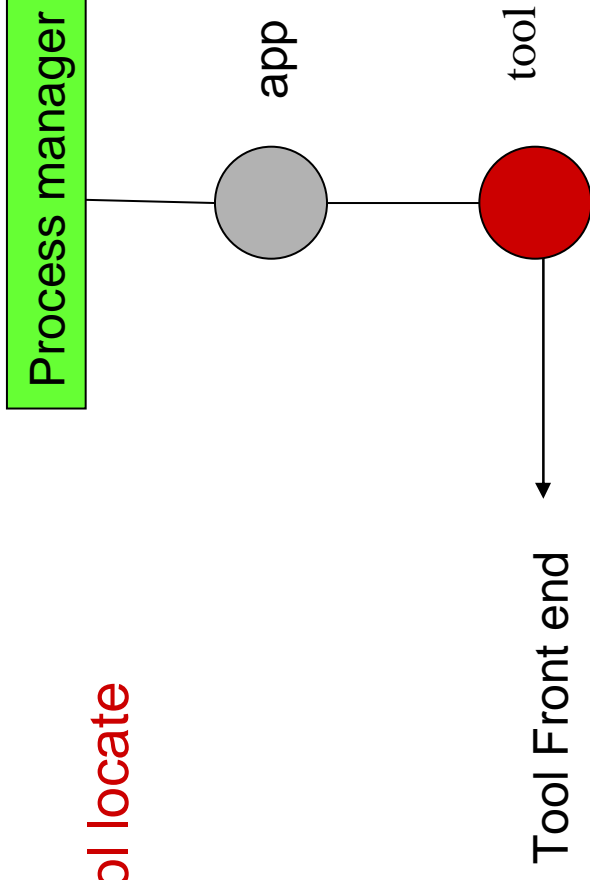
- Tool on top
  - Tool starts app
- Currently in use with MPD for
  - gdb-based debugger
  - Managing stdio
  - “transparent” tools



# Tool Attaches Later

---

- Tool on bottom
  - Process manager helps tool locate processes

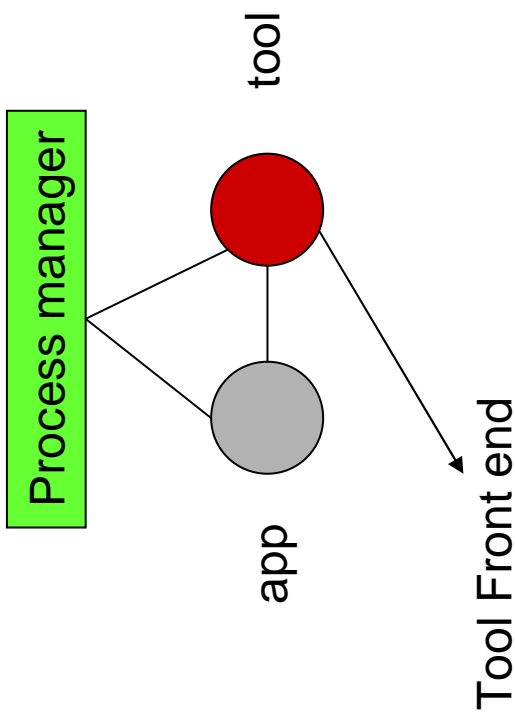


- Currently in use with MPD for
  - Totalview

# Tool Started Along With Application

---

- Tool on the side
  - Process manager starts tool at same time as app for faster, more scalable startup of large parallel job



- Currently used in MPD for
  - Simple monitoring
  - Experimental version of managing studio

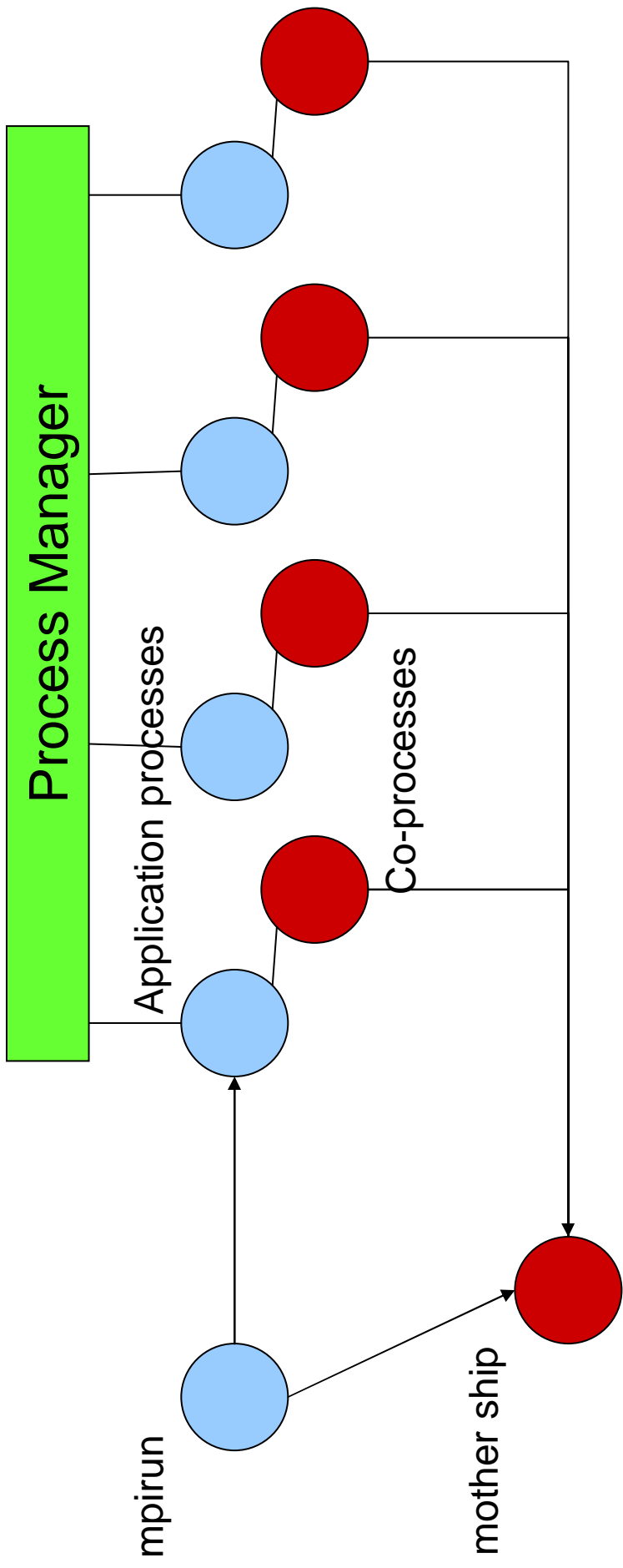
# Co-processes

---

- A generalization of specific approaches to debugging and monitoring
- Basic Idea: several types of “co” processes want to attach to / monitor / take output from application processes
- Often run on same host; need application pid
- Can be started scalably by process manager and passed pid of application process
- Sometimes need to communicate with “mother ship”
- Process manager can start mother ship, pass arguments to both mother ship and applications, perform synchronization
- Being added to XML interface for process manager component in Scalable Systems Software Project, and implemented by MPD
- Exploring more general PM/tool interface with several tool groups

# Co-processes

---



# Formal Methods and MPD

---

- Joint work with Bill McCune and Olga Matlin at Argonne
- Traditional problems with formal methods
  - Require special languages
  - Can not work on large codes
  - Effort not worth the payoff for small sequential programs
- Why MPD is a promising target for formal methods
  - Code is actually quite small
    - Complexity comes from parallelism
  - Parallelism makes debugging difficult
    - And confidence shaky, even after debugging
  - Importance of correctness
    - Critical nature of this component makes verification worth the effort

# General Issues in Using Formal Methods to Certify Correctness of Code

---

- Mismatch of actual code to model
  - System verifies model
  - Actual code will be different to some degree
  - Maintenance of certification as code changes is an issue
- Expressivity of Languages
  - Lisp, Promela, FOL
- Efficiency and scalability of underlying computational system
- Usability in general
  - Pain vs. gain

# We Tried Three Approaches

---

- ACL2
  - Venerable Boyer-Moore lisp-based program verification system
  - Can formulate and interactively prove theorems about code, types, data structures
  - Can execute lisp code with run-time type checking, assertions.
- Spin
  - Well-engineered, user-friendly system for verifying parallel systems
  - Uses special language (Promela) and multiprocess, nondeterministic execution model
  - Explores state space of multiple process/memory states, also runs simulations
- Otter
  - Classical theorem prover
  - First-order logic
  - Can be used to generate state space

# The Test Problems

---

- Typical MPD activities
  - Ring creation
  - Repair when daemon dies
  - Barrier
  - Job startup/rundown
- A typical handler:
  - Read incoming message
  - Parse
  - Modify some variables
  - Send outgoing message(s)
- What MPD code looks like:

```
While (1) {  
    select(...);  
    for all active fd's {  
        handle activity on fd  
    }  
}
```
- Code fragments are simple
- Interaction of handlers in multiple daemons difficult to reason about

# Experiences

---

- ACL2
  - Lisp fully expressive
  - Implemented micro language to help match C code
  - Simulated global Unix and daemon execution with assertions and type checking
  - Very slow
  - Difficult to formulate theorems in sufficient detail to prove
- Spin
  - Promela good match to C code (see next slide)
  - Nice user interface (see later slide)
  - Not scalable (could only handle < 10 processes)
  - Memory limited, not speed limited
- Otter
  - 4<sup>th</sup> generation ANL theorem-proving system
  - Input is first-order logic: `if State(...) & Event(...) then State(...)`
  - Many tools, but bad match to code in general
  - Fast and memory-efficient

# Promela vs. C

---

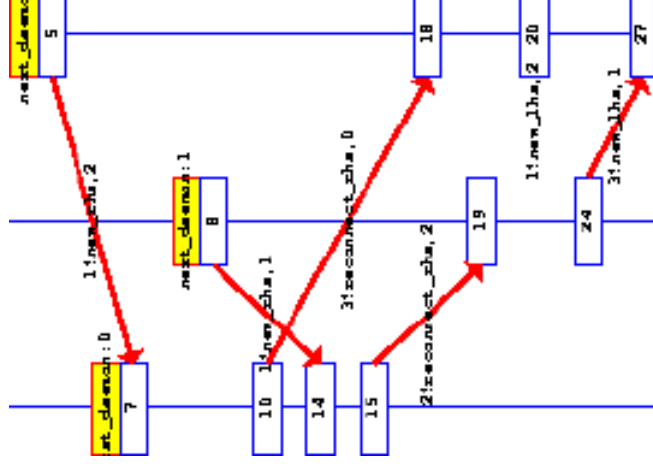
```
:: (msg.cmd == barrier_in) ->
if
:: (IS_1(client_barrier_in,_pid)) ->
if
:: (_pid == 0) ->
make_barrier_out_msg;
find_right(fd,_pid);
write(fd,msg)
:: else ->
make_barrier_in_msg;
find_right(fd,_pid);
write(fd,msg)
fi
:: else ->
SET_1(holding_barrier_in,_pid)
fi

if ( strcmp( cmdval, "barrier_in" ) == 0 ) {
if ( client_barrier_in ) {
if ( rank == 0 ) {
sprintf( buf,
"cmd=barrier_out dest=anyone src=%s\n",
myid );
write_line( buf, rhs_idx );
}
else {
sprintf( buf,
"cmd=barrier_in dest=anyone src=%s\n",
origin );
write_line( buf, rhs_idx );
}
}
else {
holding_barrier_in = 1;
}
}
}
```

# Time and Message Diagrams from SPIN

---

- SPIN can run in simulation mode, with random or directed event sequences
- Produces nice traces
- Can explore entire state space
  - If not too big
- Debugging mode:
  - Explore entire space until assertion violated
  - Rerun, directed by trace, to see sequence of events that led up to bug appearance
  - Perfect form of parallel debugging
  - Worked (found bugs not caught by testing)



# Sample Otter Input

---

State(S), PID(X), \$TRUE(barrier\_in\_arrived(S,X)), \$TRUE(client\_fence\_request(S,X)) ->  
State(assign\_barrier\_here(receive\_message(S,X),X,1)).

State(S), PID(X), \$TRUE(barrier\_in\_arrived(S,X)), \$NOT(client\_fence\_request(S,X)), \$ID(X,0) ->  
State(send\_message(receive\_message(S,X),next(X),barrier\_out)).

State(S), PID(X), \$TRUE(barrier\_in\_arrived(S,X)), \$NOT(client\_fence\_request(S,X)), \$LNE(X,0) ->  
State(send\_message(receive\_message(S,X),next(X),barrier\_in)).

State(S), PID(X), \$TRUE(barrier\_out\_arrived(S,X)), \$ID(X,0) ->  
State(assign\_client\_return(receive\_message(S,X),X,1)).

State(S), PID(X), \$TRUE(barrier\_out\_arrived(S,X)), \$LNE(X,0) ->  
State(assign\_client\_return(send\_message(receive\_message(S,X),next(X),barrier\_out),X,1)).

State(S), \$AND(\$NOT(no\_clients\_released(S)),  
\$OR(\$NOT(all\_clients\_fenced(S)),  
\$NOT(none\_hold\_barrier\_in(S)))) -> Bad\_state(S).

# Use of MPD/PMI in Upcoming Large Systems

---

- Using PMI to interface MPICH to existing process manager
  - Red Storm at Sandia National Laboratory
  - YOD scalable process manager
- Using MPD at large scale
  - IBM BG/L machine at Livermore
  - 64,000 processors
  - MPD used to implement 2-level scheme for scalability
  - Interaction with LoadLeveler, MPD running as root.

# LoadLeveler and MPD for BG/L

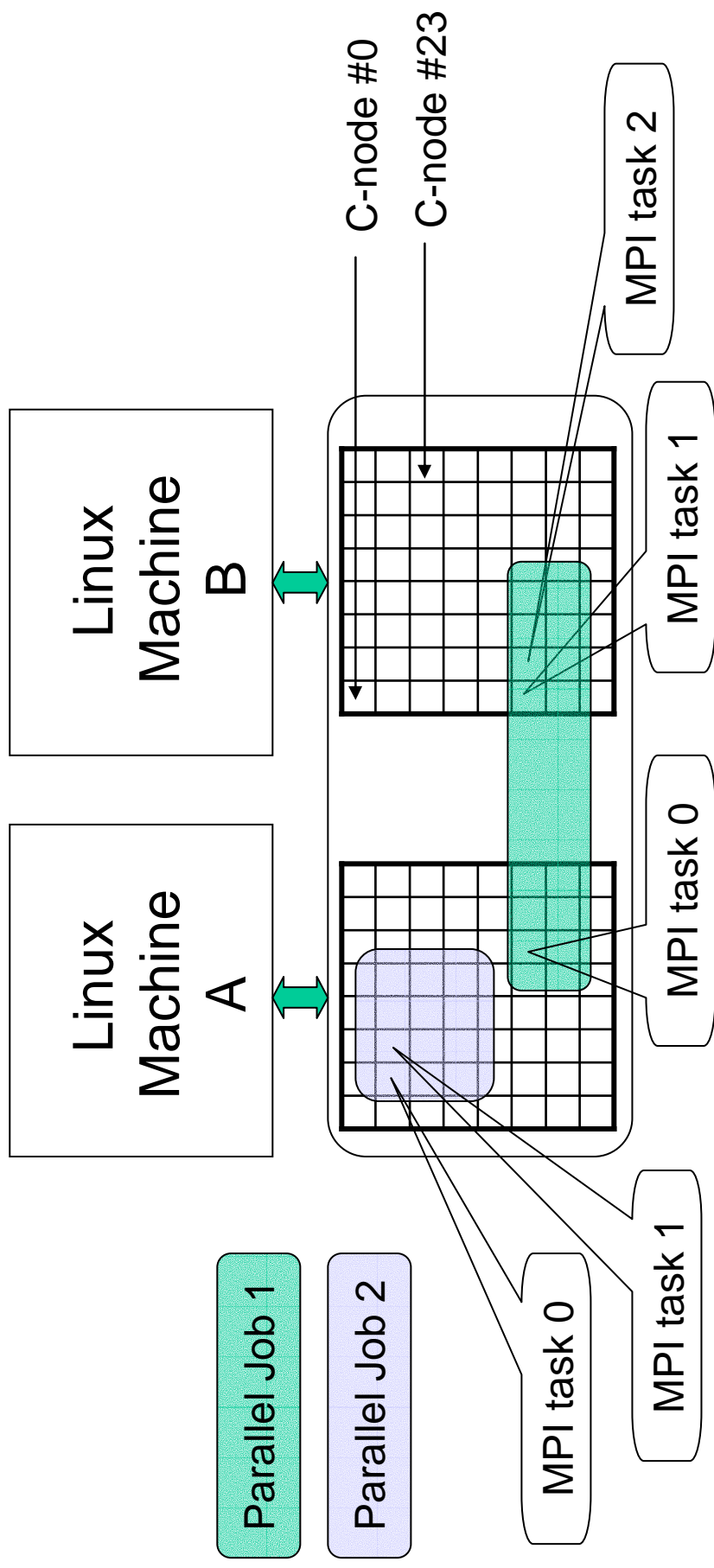
---

- Goals
  - Provide functional and familiar job submission, scheduling, and process management environment on BG/L
  - Change existing code base (LL, MPICH, MPD) as little as possible
- Current Plan: Run MPD's as root and have LL submit job to MPD's to start user job as user
- LL can schedule set of nodes for user to use interactively; then user can use mpirun to run series of short interactive jobs on subsets of allocated nodes
  - Ensure that user can only use scheduled nodes
- Build foundation for development of other scheduling and process management approaches

# BG/L Architecture

---

- Example : 2 I/O nodes, each with 64 compute nodes



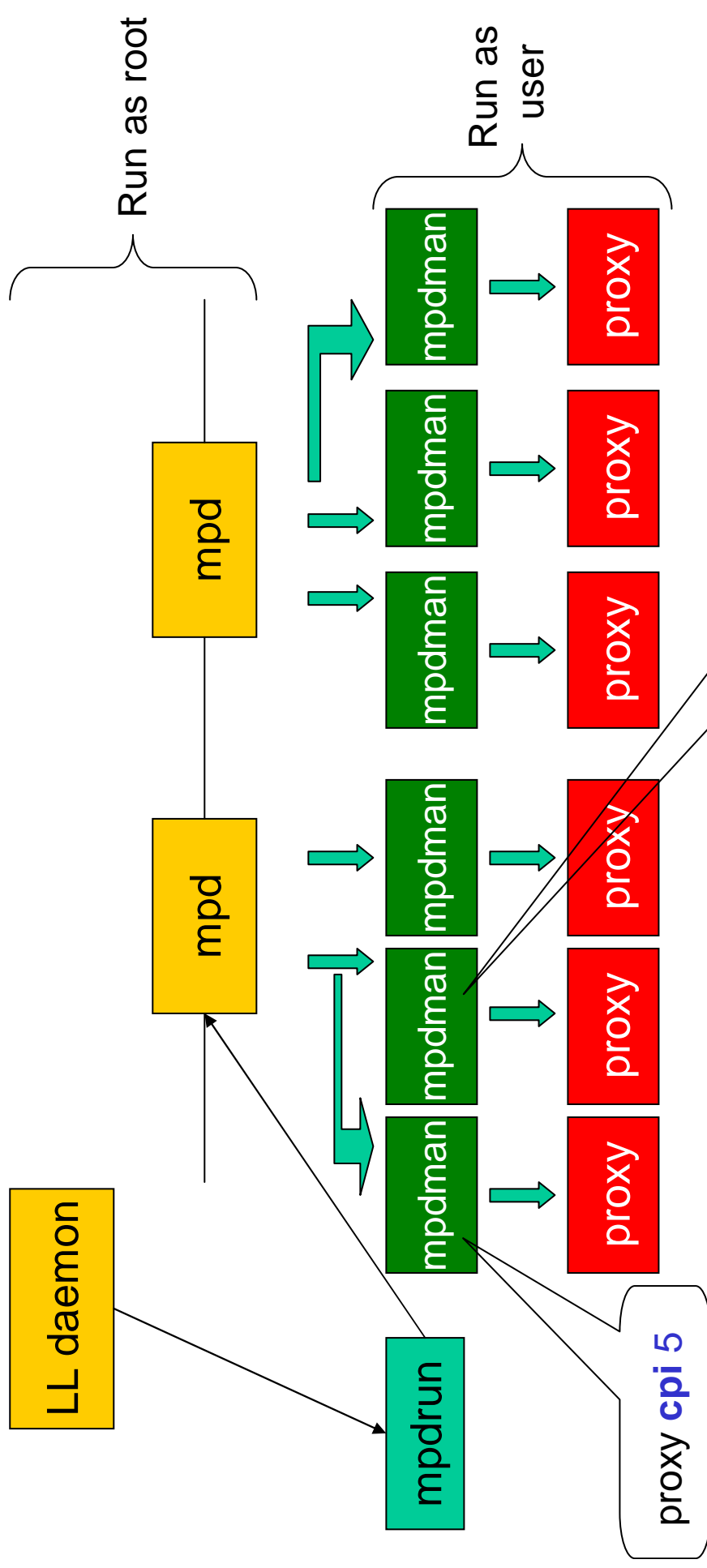
# Proxy processes

---

- A proxy process (Linux process) is created for each MPI task
- The task is not visible to the operating-system scheduler
- The proxy interfaces between the operating-system and the task, passing signals, messages etc...
- It provides transparent communication with the MPI task
- MPD will start these proxy processes
  - **Need to be able to pass separate arguments to each**

# Running the Proxies on the Linux Nodes

---



# Summary

---

- Process management is an important component of the software environment for parallel programs
- MPD is playing a role in helping to define the interface to both parallel program libraries (like MPI implementations) and scalable system software collections (like SSS).
- Formal methods may have something to contribute in the area of parallel systems software
- Tools are an important consideration for process management
- New large-scale systems are taking advantage of these ideas.



The End