SP Architecture

Uniprocessor node
– Power 2 (peak 256 Mflop/s)
Custom high-performance switch
– 80 MB/s duplex, usec latency
Distributed memory architecture
– single Unix (AIX) kernel per node
Scalable
– 2 - 512 nodes
Successful in the marketplace
– more than 500 systems sold
Scalable Parallel Computer (a la SP) = Cluster of Workstations +

Package
High performance communication subsystem
- h/w (switch, adapter)
- s/w (user-space message passing)

Single system image
- single point of control (physical - console, and logical - config)
- global services (system management, job management)

Parallel operating environment
- (parallel system interfaces, libraries, tools, compiler)

High-availability framework
Application subsystems
- technical packages
- commercial subsystems (DB, OLTP, monitor)

Applications
Building on Workstation technology
Why Commodity Technology

- Necessary in relatively small, cost-conscious market.
- Leverage: lower development cost, time to the market.
- Support sequential loads with no modifications.
- Advantage of standard, open interfaces.
- Better parallel performance can be achieved by tighter coupling of microprocessor technology, interconnect technology, kernel technology, compiler technology, etc.

"Mostly commodity" approach has created a viable market for clusters and scalable computers today.

As market grows, there is more room for development of specific h/w and s/w.

Q: What are the critical custom technologies for scalable computers today and tomorrow?
Scalable Parallel Computer = General Purpose High-End Server

Wide performance range (2w - 512w now)
Configuration flexibility
- Processor, memory, I/O, connectivity,...
Wide range of applications
- I/O intensive ("commercial")
- Compute intensive ("technical")
- Communication intensive ("network centric")
Wide spectrum of requirements
- loosely/tightly coupled
  communication bandwidth, granularity
- low/high sharing
- low/high availability

One platform?
Mar
c
Sni
r
Multipl
e
Platform
s
un
i
SP
C
scalability
availability
commodity
sharing
coupling
SM
P
SP Evolution

Faster node
- increased single node performance
- SMP node (package efficiency)
  uniform access shared memory

Distributed O/S
- single kernel per node (scalability problem)

No single point of failure

Wider application spectrum
- possibilities: data mining, unstructured search engine, web server, mail server, video server, ...

What else?
Multiple Paths between any two nodes
Network Scales with each added Node
80 MB/s duplex, 0.5 usec latency per board now
B/w can be scaled with node performance for 2 generations, with no major change in technology
Custom vs Commodity Switch

Increased similarity in function
- switched (not shared)
- Packet-switching, small packets

2-3 years gap in performance
- Performance gated by cost, not by physics

Different requirements
- b/w, reliability, package, distance, protocols

Custom switch can and will maintain performance differential
Custom switch may use components of commodity (ATM?) technology
I/O Bus vs Memory Bus

I/O bus is
- standard, stable interface
- supports many loads
- built for lower performance
- built for long DMA transfers in non-paged memory

Memory bus
- proprietary, and changing
- built for high performance
- supports few loads
- built for short cache line transfers and cache coherence protocols

Direct connection to memory bus will differentiate SPC’s from clusters.
Standard W/S I/O slots will not be fast enough for SPC and will not have right functional interface for smart, deep adapter
Effective communication time
= communication latency + 
scheduling_time x Prob[recv is not 
running]

Effective broadcast time
= broadcast latency + 
scheduling_time x Prob[some 
recv is not running]

Up to 40% performance loss on a 
128 node system, if not taken care 
of!

Solutions:
0. Gang scheduling + nonblocking 
communication; or
1. Efficient message-driven thread 
scheduling
User and kernel paths
Message-passing library
- MPI, MPL, PVM/e
35-48 MB/s, 40 usec
Protocol stack executed by main compute engine
One copy protocol
  (no cache coherent DMA)
Bandwidth bottlenecks:
- switch (< 80 MB/s)
- uchannel (< 160 MB/s)
- s/w (<255B packet)
Latency bottlenecks:
- uchannel
- no autonomous comm coprocessor
Higher bandwidth, lower latency, increased overlap
– direct memory connection
– higher link b/w
– larger packets
– h/wcommassist (CRC, packaging, flow control,...)
– dedicated commproc(?)
Zero copy protocol
Multiple protection domains
Thread compatibility
Virtual DMA support
Evolution -- Communication Paradigms

Integration of communication and process/thread scheduling
- message-driven threadscheduling
- remotethreadspawn
- activemessage paradigm

Impediments:
- heavythreadcontext
- error isolation
- protection

Integration of internal and external communication

Support to shared memory programming models
Shared Memory Programming Model (1)

Processor to memory comm, rather than proc to proc

producer \hspace{1cm} \text{send} \hspace{1cm} \text{recv} \hspace{1cm} \text{consumer}

vs

producer \hspace{1cm} \text{write} \hspace{1cm} \text{mem} \hspace{1cm} \text{read} \hspace{1cm} \text{consumer}

+ Better fit to dynamic, irregular codes
+ Can be implemented with less overhead (?)
– No buffering, no synchronization in communication
Memory to Memory Copy

Comm subsystem:
- adapter + microcontroller (all h/w implementation)
- or adapter + dedicated processor (polling)
- or adapter + commprocess (interrupt)
Basic Communication Mechanisms

Put/get
    – put(local_address, remote_proc, remote_address, count, local_flag, remote_flag)

Accumulate
    same as put, specifies update operation (sum, max,...)

Read_Modify_Write
    same as accumulate, returns previous value

Enqueue/dequeue
    – Enqueue(local_address, remote_proc, remote_queue, local_flag)

    nonblocking
    general data layouts (scatter/gather?)
    addressing (direct/indirect?)
    coordination mechanisms
    interrupt vs polling (enqueue/dequeue)
Shared Memory Support

0. Shared name space: data distribution does not "intrude" in data naming
1. Caching: a directory based scheme to keep track of local copies and their status
2. Communication occurs as side-effect of load/store: no need for special communication commands (still need synchronization commands)
   (1) can be easily supported in s/w on top of put/get
   (2) can be supported (with no preemption)-- requires active messages
   (3) implemented via page exception (low performance) or requires h/w

   Issue (for 3): H/w assis for shared memory that
   – avoids use of page fault exception for comm
   – does not require global virtual memory manager
   – does not slow local memory accesses

   How important is 3?
Parallel Programming Paradigms

Parallelism and communication have to be handled as part of algorithm design.

Ideal situation:
– Implicit parallelism: user optimizes for coarse grain; system (compiler+run-time+h/w) map computations onto processors, with low synchronization cost.
– Implicit communication: user optimizes for (spatial, temporal) locality; system maps data onto processors with low communication cost.

Reality:
– Complex tradeoffs between total computation work, level of parallelism, granularity, and communication.
– Limited investments in compiler/run-time/hw technologies
– Large investments in existing programming languages
Implicit Parallelism

Standard sequential program + compiler magic
   – far future (or science fiction)
Standard sequential program + annotations
   – Program can compile and run on uniprocessor
   – Annotations specify mapping (of computation and data) to processors, but do not change program semantics

Data parallelism:
   – User specifies data distribution; distribution of computation derived from data distribution
     specification usually declarative, static, regular

Control parallelism:
   – User specifies computation distribution: data migrates where needed
     specification usually executable, dynamic

The two approaches are equivalent if executable, dynamic distribution is allowed
Dynamic redistribution requires shared-memory like communication for efficient support
Parallel FORTRAN

FORTRAN
- Support of High Performance FORTRAN
- Support for automatic data partition
- Support for control parallelism (loop parallelism/task parallelism)
  HPF2 effort, "on" directive.

Same language supported on all (IBM) platforms (Uni, SMP, SP): FORTRAN 90 + directives
Easy port of "shared-memory" codes (Cray, SGI directives)
Performance on large-scale parallel computers will still require non-trivial algorithm development and tuning.

Remote memory copy and "shared memory" runtime solutions will improve performance and facilitate common shared-memory/distributed-memory compilation.
Compilation

\$HPF  INDEPENDENT
DO  I=1, N
  IF  COND(I)
    THEN  CALL  FOO1(I)
    ELSE  CALL  FOO2(I)
  END IF
END  DO

Message passing compilation results in sequential execution (or wasteful communication)
Use of put/get results in shared-memory style compilation (access on demand)
Caching may further improve performance
$HPF$ INDEPENDENT
DO I=1, N
    A(I) = B(MAP1(I)) + B(MAP2(I))
END DO

Message passing compilation uses inspector/executor
– requires collective, expensive operation
– does not allow load balancing
– does not work if access patterns dependent on data computed within same loop (e.g. N-body code)
Parallel C++

Ongoing efforts:
- ABC++
  activeobjects (tasks)
  futures
  parametric regions
- Academic collaborations (CC++, pC++,...)

Please give us a standard

Ongoing research: SOM (Corba) encapsulation of parallel objects.
- interface for resource management
- high-performance SOM run-time
- IDL extensions (?)
Why not Conventional Shared Memory?

Virtual Memory Management does not scale
Error isolation is hard
L2 interface does not scale
  – small cache line
  – small number of pending memory accesses
  – small TLB
  – expensive L2 and TLB coherence protocol
  – small L2
    Need to use local memory as additional level in memory hierarchy (L3)
Parallel System Services

Each node is controlled by separate kernel
Application processes are tightly coupled
  – coordinated parallel computation
Kernels and services are not coupled!
  – coordinated management of resources used by parallel application
  – parallel system interfaces

appl. process

system libs:
I/O, comm,...

kernel:
swapper, VMM,...

appl. process

system libs:
I/O, comm,...

kernel:
swapper, VMM,...

appl. process

system libs:
I/O, comm,...

kernel:
swapper, VMM,...
The Microkernel Approach

**ISSUES**

Parallelism within the kernel
- Performance
- Scalability

Parallel interface to parallel application
- Parallel system call interface
- Coordinated management of resources used by multiprocess parallel application
Serial vs Parallel System Interface

- Sequential applications
- Parallel system

vs

- Parallel application
- Parallel system
Parallel I/O Server

Parallel client

Parallel server

PIOFS Client

Parallel Application

PIOFS Server

Switch

CPU

Mem

adptr

adptr

adptr

CPU

Mem
**Conventional Invocation Model**

Conventional model - sequential server
- each process invokes I/O server separately
- all invocations are channeled to one server

Conventional model - parallel server
- each process invokes I/O server separately
- I/O server is parallel subsystem
- parallel synchronization code within parallel server enforces atomicity (kernel locks or monitors)
**Parallel Invocation Models**

**Improved conventional model:**
- server has policies that take advantage of correlation between requests of parallel client processes
  - e.g. back-end caching vs front-end caching

**Parallel model**
- system call is a collective call, executed in a loosely synchronous manner by all clients in a group
  - e.g. collective I/O: read-broadcast, readscatter, write-gather, etc.

  **Thesis:** more efficient performance can be achieved by direct support for collective calls (true for message passing)

  Third-party protocols required for efficient implementation of collective I/O
  (a process requires a system service on behalf of another process)
What mechanisms are available for collective invocation of (local, remote) parallel procedures (objects, methods)? Are these mechanisms compatible with distributed object frameworks such as CORBA (OLE, Java,...)?

What information is available at the interface?
Sequential Interface

Best encapsulation
- "knowledge" about parallelism is internal to each module

Easy fit in existing object frameworks
- each parallel object is "represented" by a sequential stub
- sequential stub may negotiate for parallel resource allocation, if needed

Worst performance
- Each invocation is a serial bottleneck
Sequential Control - Parallel Data

Reasonable encapsulation
- data path can be negotiated at binding time

- Fits existing object frameworks (?)
  - may need extensions for dynamic data path binding

Good performance
- trivially so with shared memory
Important special case: parallel collective invocation resolves into multiple, loosely synchronous, local invocations.

Generalization to remote invocation?
- remote collective invocation has same interface as local collective invocation
- efficient
- requires extensions to existing object frameworks

All Parallel Invocation
Parallel Server Models

**Parallel-Sequential**
- Multiple clients connecting to multiple servers

**Parallel-Parallel**
- Multiple clients connecting to a single server

**Sequential-Parallel**
- Multiple clients connecting to multiple servers

**Collective I/O Library**
- Most useful with (loosely) synchronous programming model
- Takes advantage of correlations between system activities at processes of parallel application
Vesta Design

Support parallel I/O to one file from multiple compute nodes to multiple storage nodes
- Flexible allocation of I/O bandwidth to jobs
- Flexibility in h/w configuration
- Need not collocate files and processes
- Can offload system calls

Scalable design
- No single point of access for metadata
- No locking for atomicity control
- $2^{64}$ byte per file, $2^{54}$ files

User control of data layout
- Control of physical layout at creation time
- Access to user defined subfiles

Support for shared offset
Support of Posix interfaces
Support for import/export functions
Vesta - Salient Features

User control of layout
- file is 2D array of records
  - cols are mapped round-robin
  - file struct and map specified at creation
- subfile is a rectilinear file decomposition
  - different processes can access same or different subfiles of same file;
  - subfile defined when opened

Accesses are atomic and serializable
- even if they span multiple store nodes

Minimal number of accesses
- commoccurs only between client and servers that hold data
- Client has full knowledge of data distribution after file is opened
- atomicity controle done by combtw servers
Vesta - Future Activities

Evolution to product (95)
  – Posix compatibility
  – protection, recovery

MPI-IO -- MPI library for parallel IO
  – Jointly defined and implemented with NASA Ames

Scalable IO Initiative
  – IO benchmarks and performance analysis
    Performance tools for IO
  – Enhanced system support for parallel IO
  – Compiler support for parallel IO
    Integration with HPF

Optimizations and function enhancements
  – Collective IO
  – Checkpoint/restart
MPI-IO

IO = communication with (logical) IO server
   – point-to-point
   – collective
   – blocking/nonblocking

Use of MPI datatypes
   – for layout of data at client memory
   – for layout of data on file
     superset of vesta partitions

Use of MPI communicators
   – file access group
   – error handling
Problem: processes in parallel partition need be coscheduled
- up to 40% performance deterioration on 128 node benchmark, due to <3% random system background activity!

Current solution
- dedicated fixed partition
- synchronization of scheduling slots for background processing

Future:
- gang scheduling
  - time sharing of virtual "dedicated" partitions
- variable size partition
Scheduling and Communication Infrastructure

Infrastructure
- Control network
  - Coordinate global changes in comm state (global swap, addition/deletion)

Policy
- Parallel job time sharing policy
- Application driven scheduling
- Event-driven scheduling

Issues
- Need "parallel events" (collective calls) for event-driven scheduling
- Need more information on system overheads
- Interaction with memory management
Problem

Parallel code does not mix easily with sequential code.
- cannot integrate parallel code modules in existing software systems
- cannot integrate parallel servers into existing distributed systems

No good parallel interfaces between parallel modules
- E.g., from DB2 to Quest

Parallel code runs efficiently only in batch.
- no interactive, real-time, embedded, ..., parallelism

Possible solution:
- Encapsulate parallelism within objects
- Provide SOM support for parallel objects

Develop system where
- Sequential objects can be replaced with parallel objects, with no change in the remaining system.
- Parallel objects interact with parallel objects, with no sequential bottlenecks
DSOM (aka CORBA)

- IDL specification of remote object
- IDL Compiler
  - User client code with remote object reference
  - Glue code generated by IDL compiler
  - Object request broker
- Distributed server objects
PARSOM

IDL specification of remote object

IDL Compiler

Parallel object
(MPI, HPF, CC++)

User client code with remote object reference

glue code generated by IDL compiler

object request broker
Some Technical Issues

Data needs to be distributed
- IDL: specification of distribution
- run-time for data (re)distribution
  extension of activity on run-time for parallel languages

Parallel object may need to be instantiated
- Need dynamic partition allocation and protocol between ORB and resource manager
  (rather than local ORB daemon)
Invocation Mechanisms
Relevant Activities

IBM
– SOM, DSOM
IBM Research (my area)
– Dynamic process groupscheduling
– Persistent object storage
– Milliways

Pasadena II Workshop:
"A research program should be established to investigate and shape the interaction between distributed processing and HPC. This should take the form of research into, and implementation of, an extended CORBA-like infrastructure that supports parallel processing."

Ongoing (starting) efforts by CRPC members (with my lobbying)

Providing CORBA interfaces from CC++ (Chandi/Kesselman, Caltech)

IDL specification for data parallelism (Gannon, Indiana)
Common interface and run-time support for parallel objects (Saltz, Maryland)

Use common object interfaces to link
– sequential client to parallel server (e.g., computational server, EOS server)
– multiple parallel modules (e.g. multidisciplinary applications)

Also, interest at LANL, NRL,..
SIMD and SPMD Model

- SIMD Code
  - data parallel operations

- SPMD Code
  - synchronous procedure calls
Parallel C++ libraries for matrix operations (A+/P+), linear algebra (Scalapack++), irregular data structures (Chaos++), adaptive grid (LPARX),...

Parallel code is identical to sequential code
– All processes execute same sequential code (including library calls)
– All communication is buried into library code
– Information on data distribution is buried in object descriptor.

Good match to HPF
Extendible approach (hook your own library with your messy message-passing code)
In need of standardization
Needs many additions (file I/O, graphics,...)

Pasadena II workshop:
"We recommend to convene within few months a forum for implementers of OO scientific libraries in view of standardizing the interfaces for these libraries... It will aim at defining within less than a year portable descriptions for distributed dense and sparse arrays, grids, etc..."
Infrastructure for Parallel Objects

Mechanisms for dynamic parallel object instantiation
- 1-n, n-1, n-m
- same or distinct processes

Mechanisms for redistribution of data
- HPF rectilinear distributions
- user-defined distributions

Parallel persistent object repository (aka PIOFS)
Conclusion

Scalable parallel servers are here to stay
- intermediate level of integration between clusters and tightly coupled, single kernel SMPs.
- significant reuse of commodity technology

A reasonable path can be outlined for h/w evolution
- including support for shared memory programming model

Much thinking still needed on system structure

What is a scalable Operating System?
What should be system interfaces to parallel applications?

How does one build hierarchical resource management services?

Much ground for useful and interesting research
- But designing beautiful, nicely integrated solutions from scratch is an exercise in futility