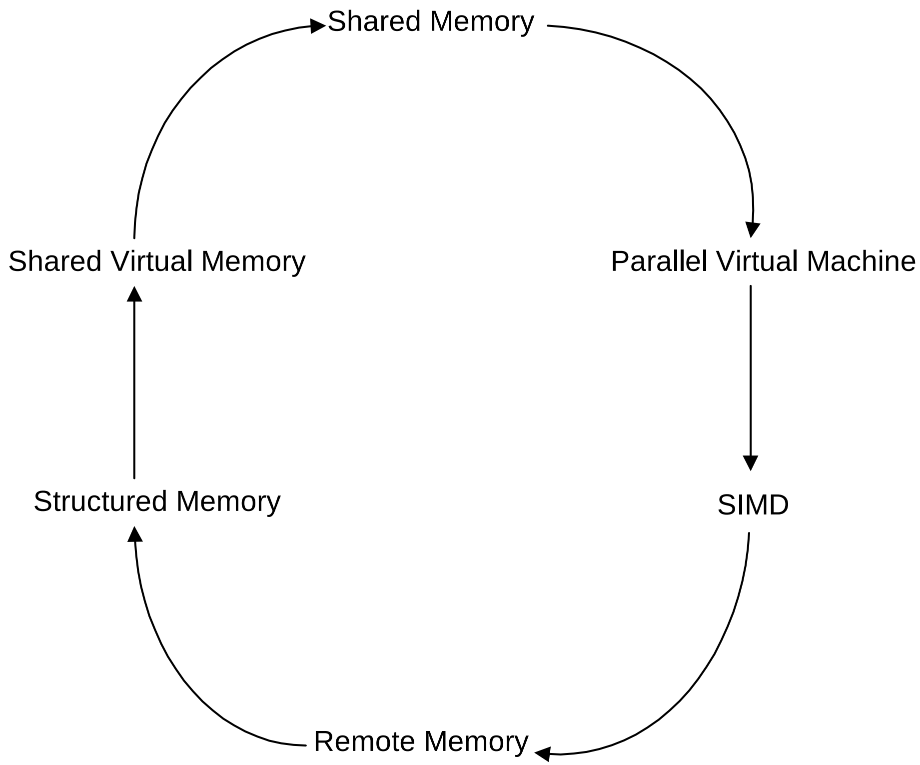
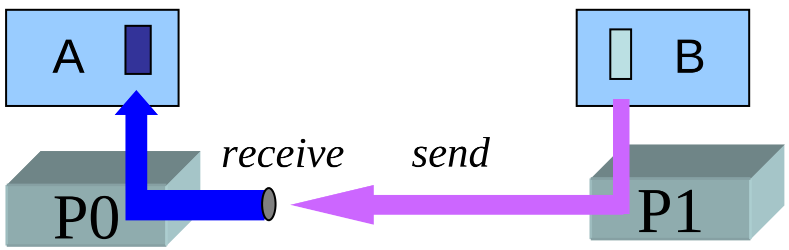
Remote Memory Architectures

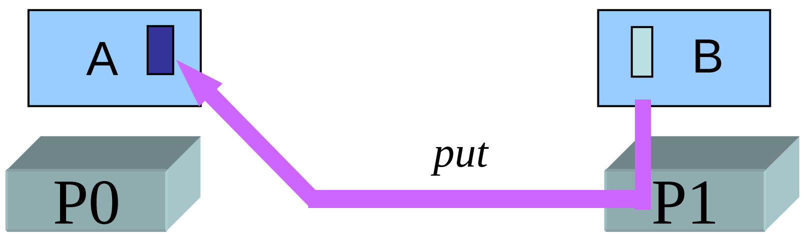
Evolution


Shared Memory
Parallel Virtual Machine
SIMD
Remote Memory
Structured Memory
Shared Virtual Memory

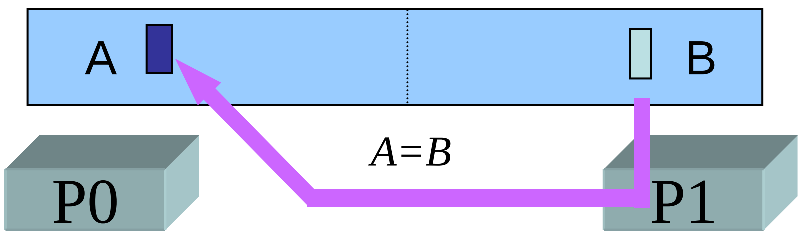
Communication Models



message passing
2-sided model

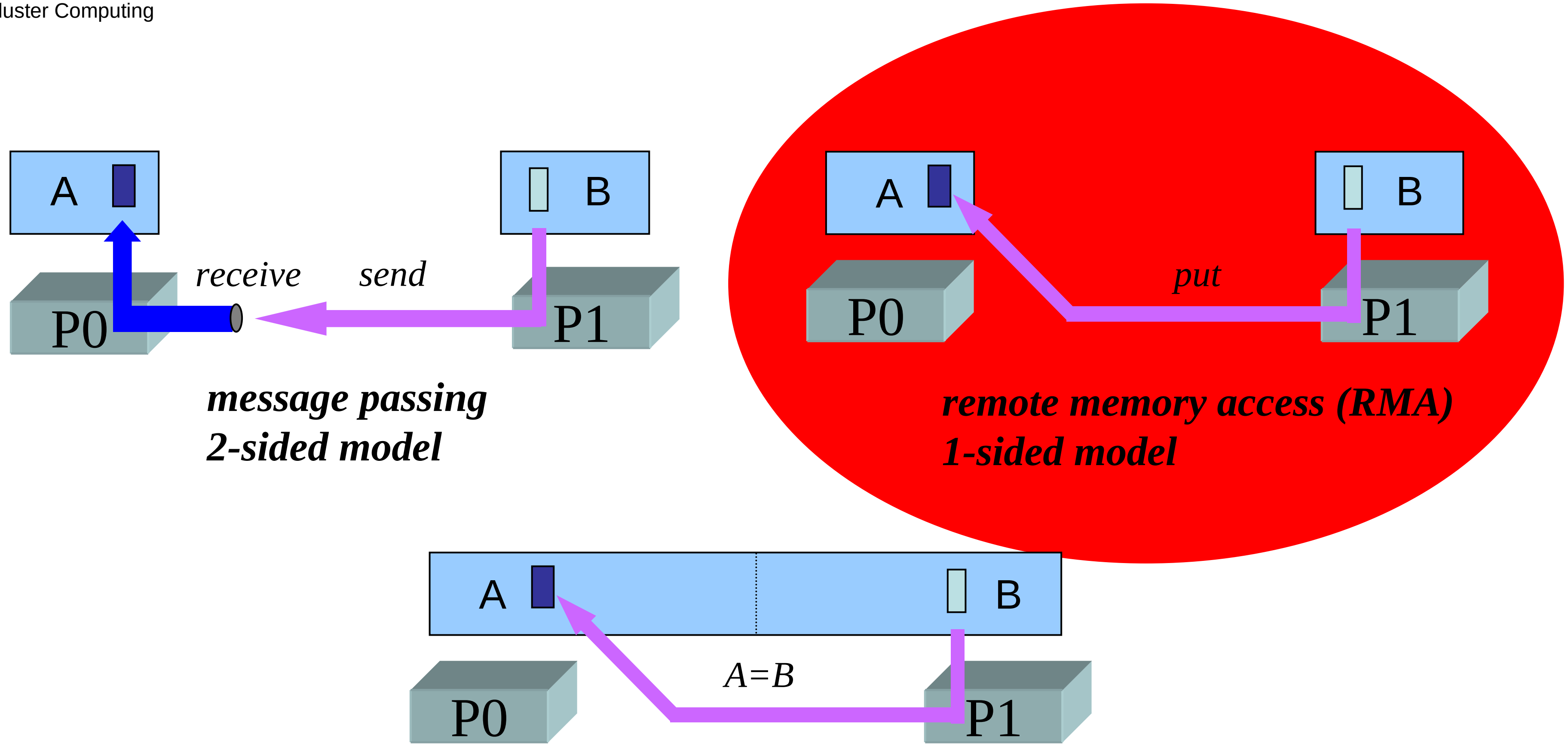


remote memory access (RMA)
1-sided model



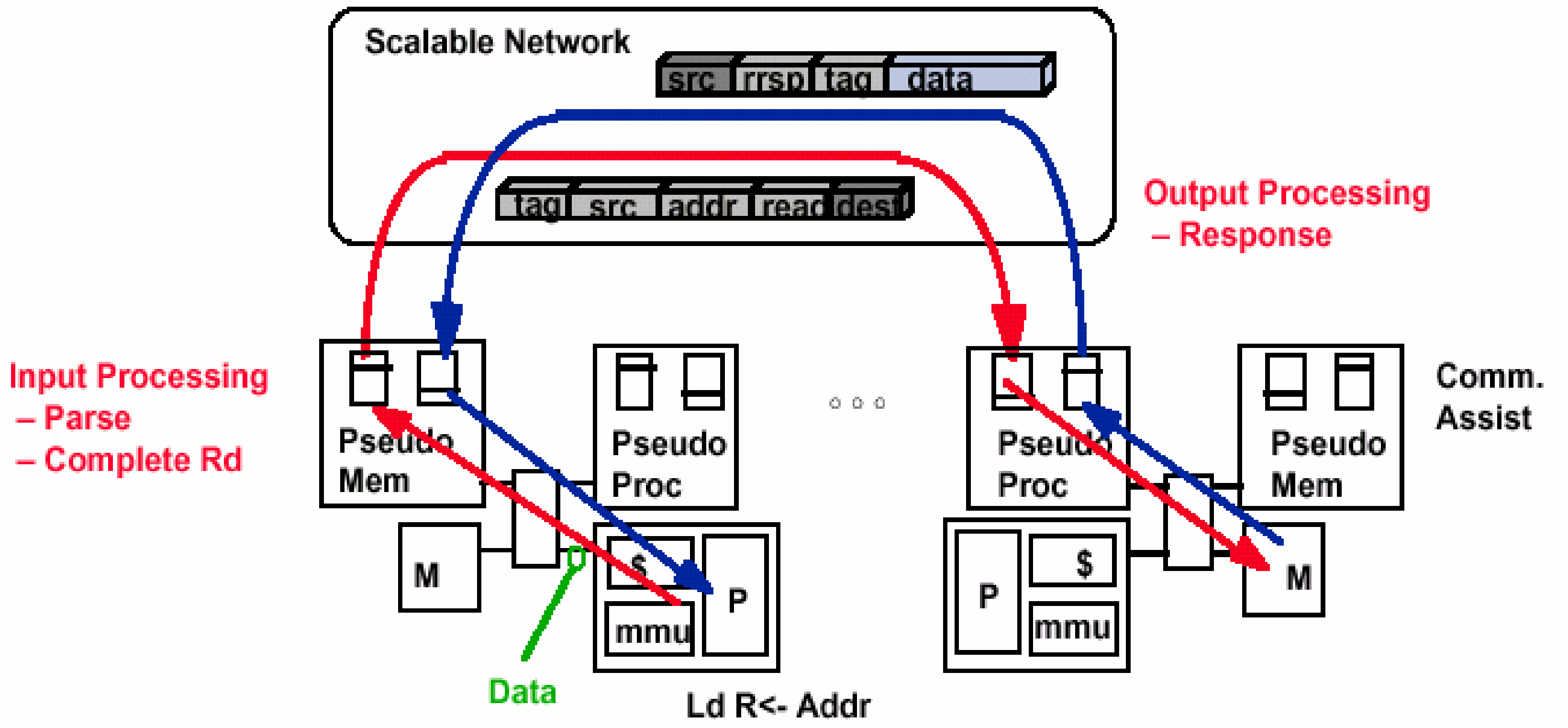
shared memory load/stores
0-sided model

Communication Models



shared memory load/stores
0-sided model

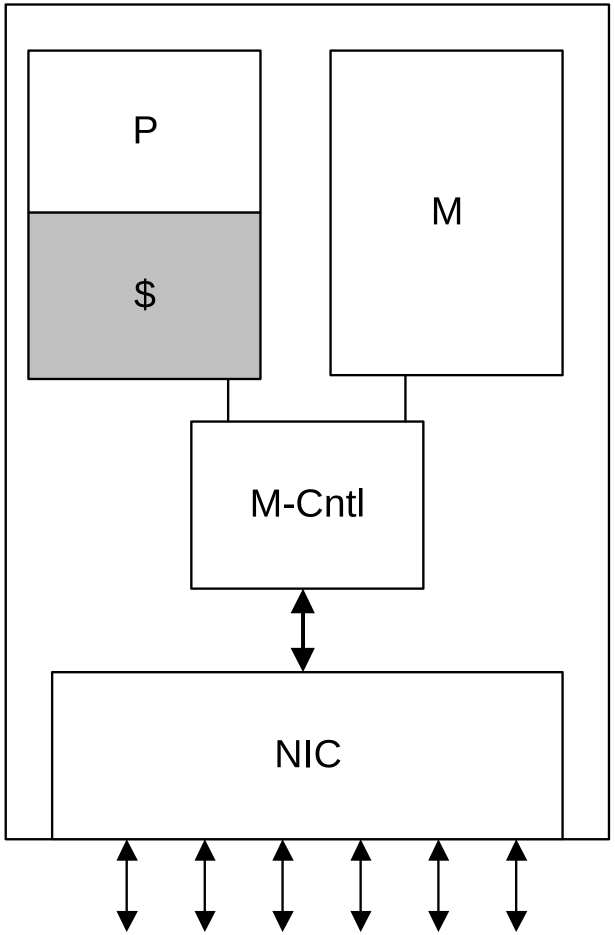
Remote Memory



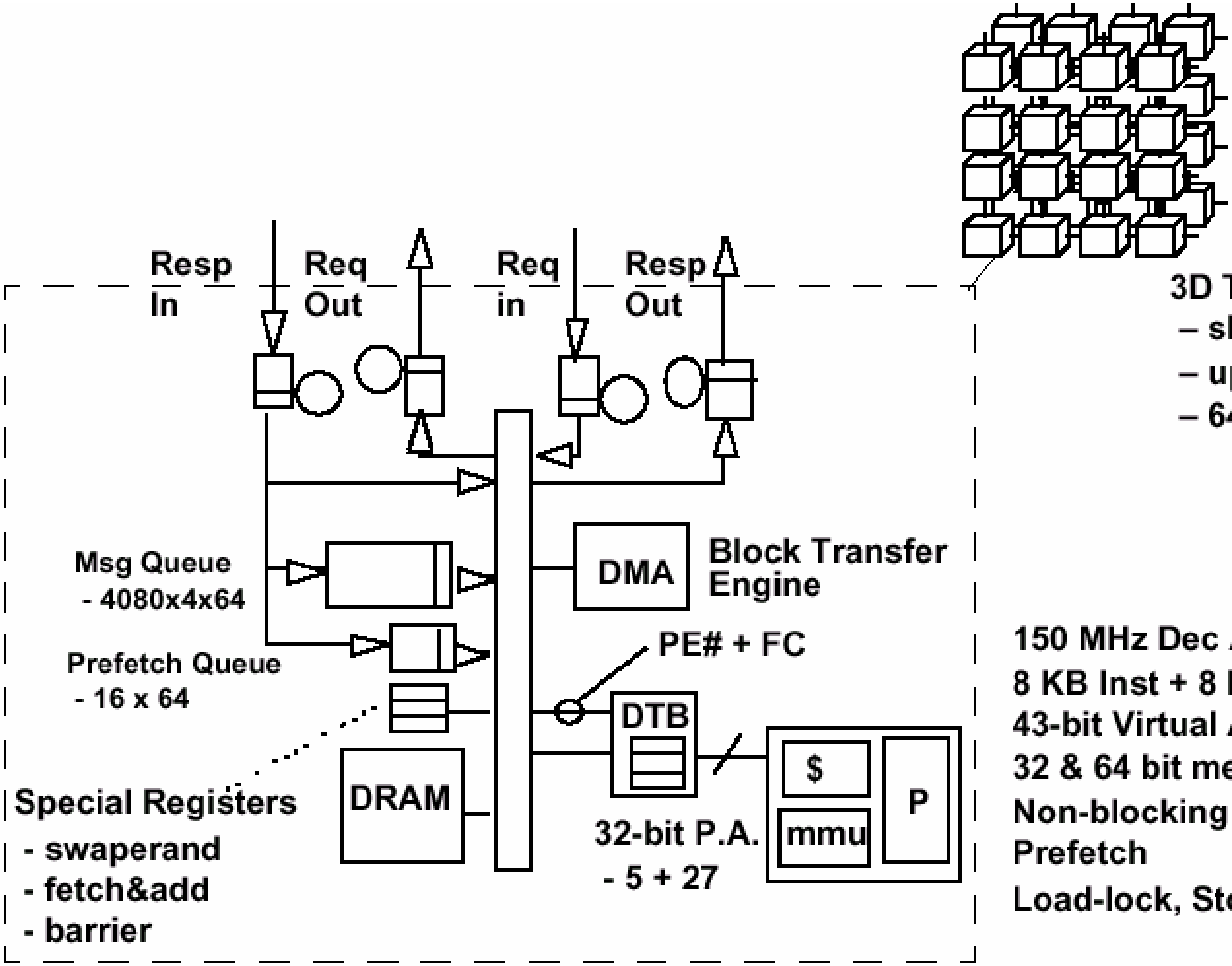
Cray T3D

- Scales to 2048 nodes each with
  - Alpha 21064 150Mhz
  - Up to 64MB RAM
  - Interconnect

Cray T3D Node



Cray T3D



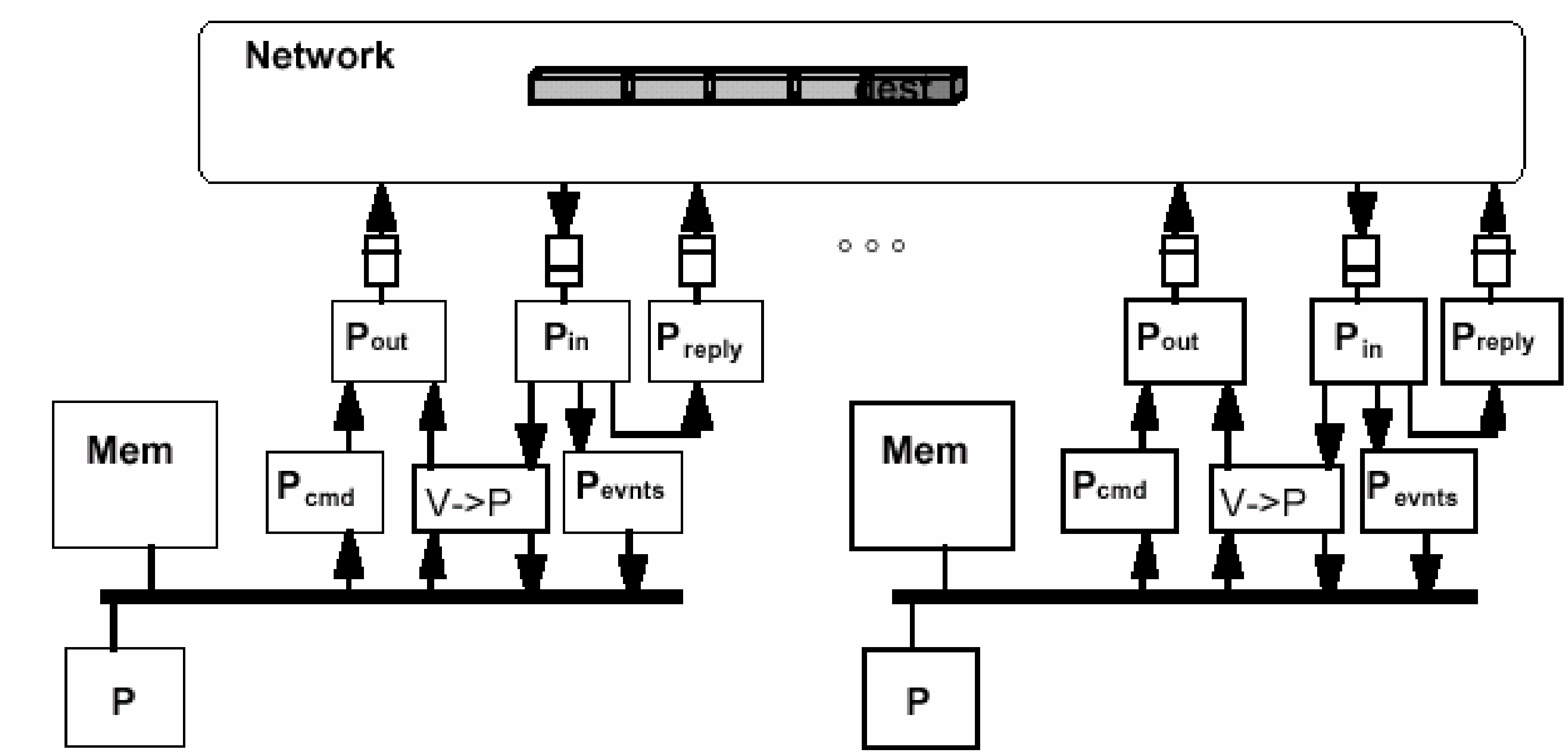
3D Torus of Pair of PEs
- share net & BLT
- upto 2048
- 64 MB each

150 MHz Dec Alpha (64-bit)
8 KB Inst + 8 KB Data
43-bit Virtual Address
32 & 64 bit mem + byte operations
Non-blocking stores + mem-barrier
Prefetch
Load-lock, Store Conditional

Meiko CS-2

- Sparc-10 stations as nodes
- 50 MB/sec interconnect
- Remote memory access is performed as DMA transfers

Meiko-CS2


Network
dest
Pout
Pin
Preply
Pout
Pin
Preply
Mem
Pcmd
V->P
Pevnts
Mem
Pcmd
V->P
Pevnts
P
P

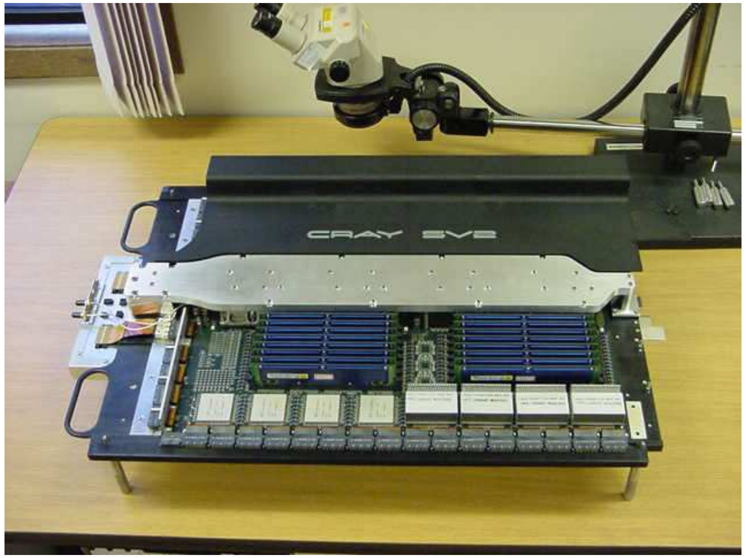
Cray X1E

- 64-bit Cray X1E Multistreaming Processor (MSP); 8 per compute module
- 4-way SMP node

Cray X1: *Parallel Vector Architecture*

Cray combines several technologies in the X1

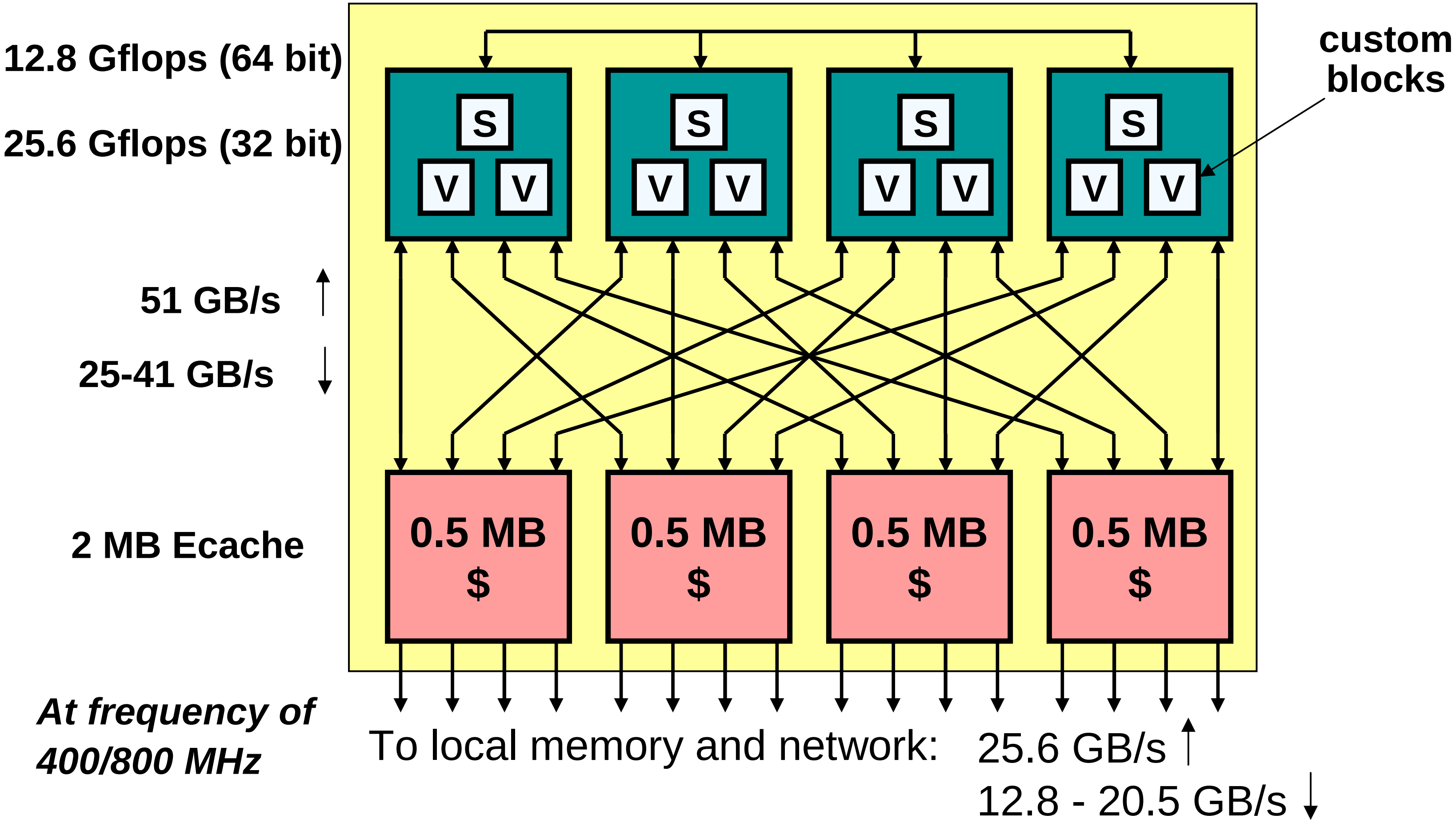
- **12.8 Gflop/s Vector processors (MSP)**
- **Cache (unusual on earlier vector machines)**
- **4 processor nodes sharing up to 64 GB of memory**
- **Single System Image to 4096 Processors**
- **Remote put/get between nodes (faster than MPI)**

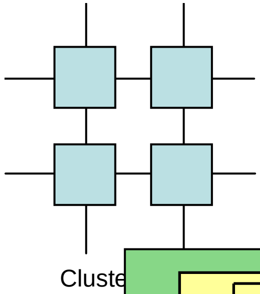


At Oak Ridge National Lab 504 processor machine, 5.9 Tflop/s for Linpack (out of 6.4 Tflop/s peak, 91%)

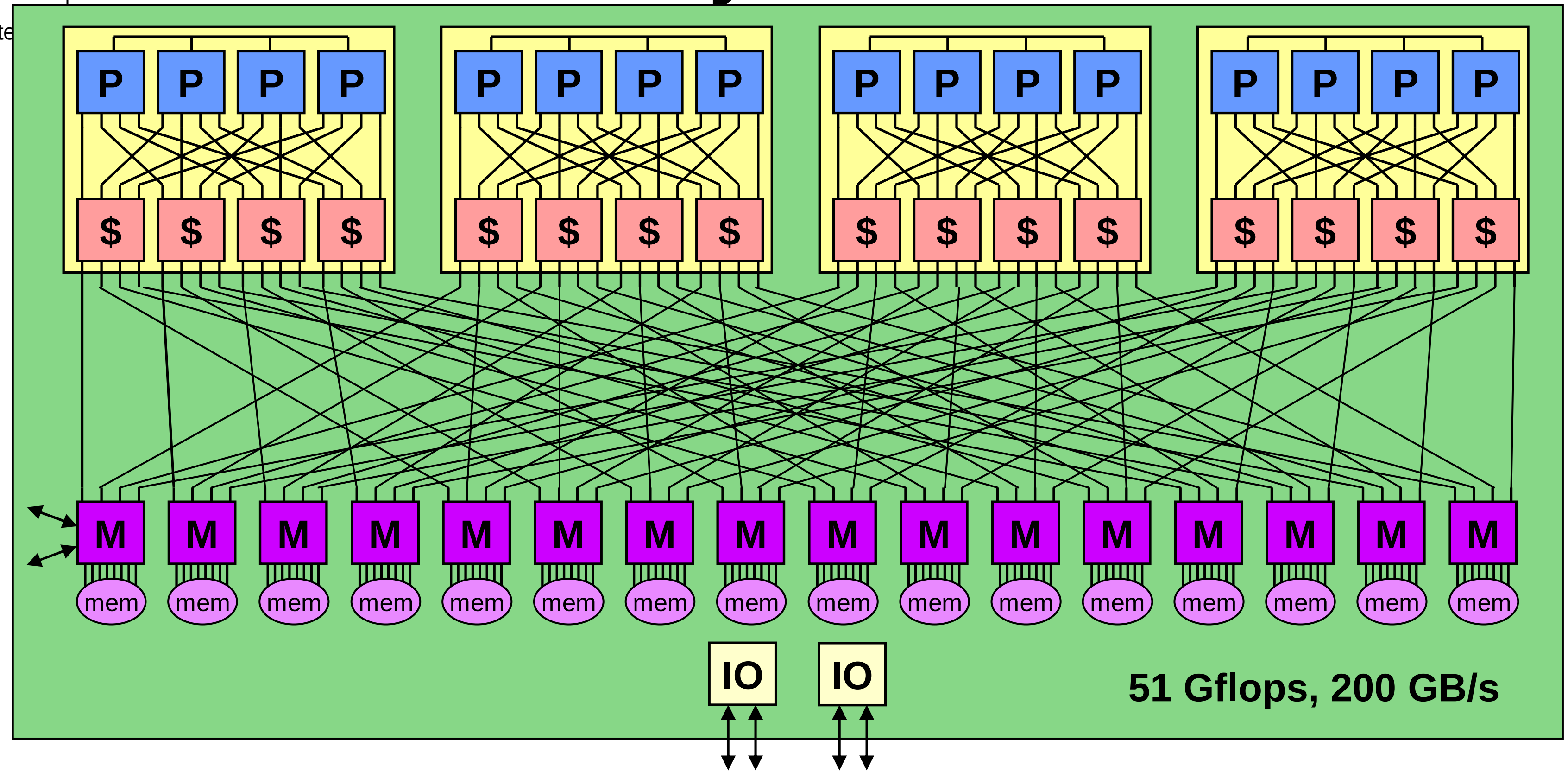
Cray X1 Vector Processor

- Cray X1 builds a larger “virtual vector”, called an MSP
  - 4 SSPs (each a 2-pipe vector processor) make up an MSP
  - Compiler will (try to) vectorize/parallelize across the MSP



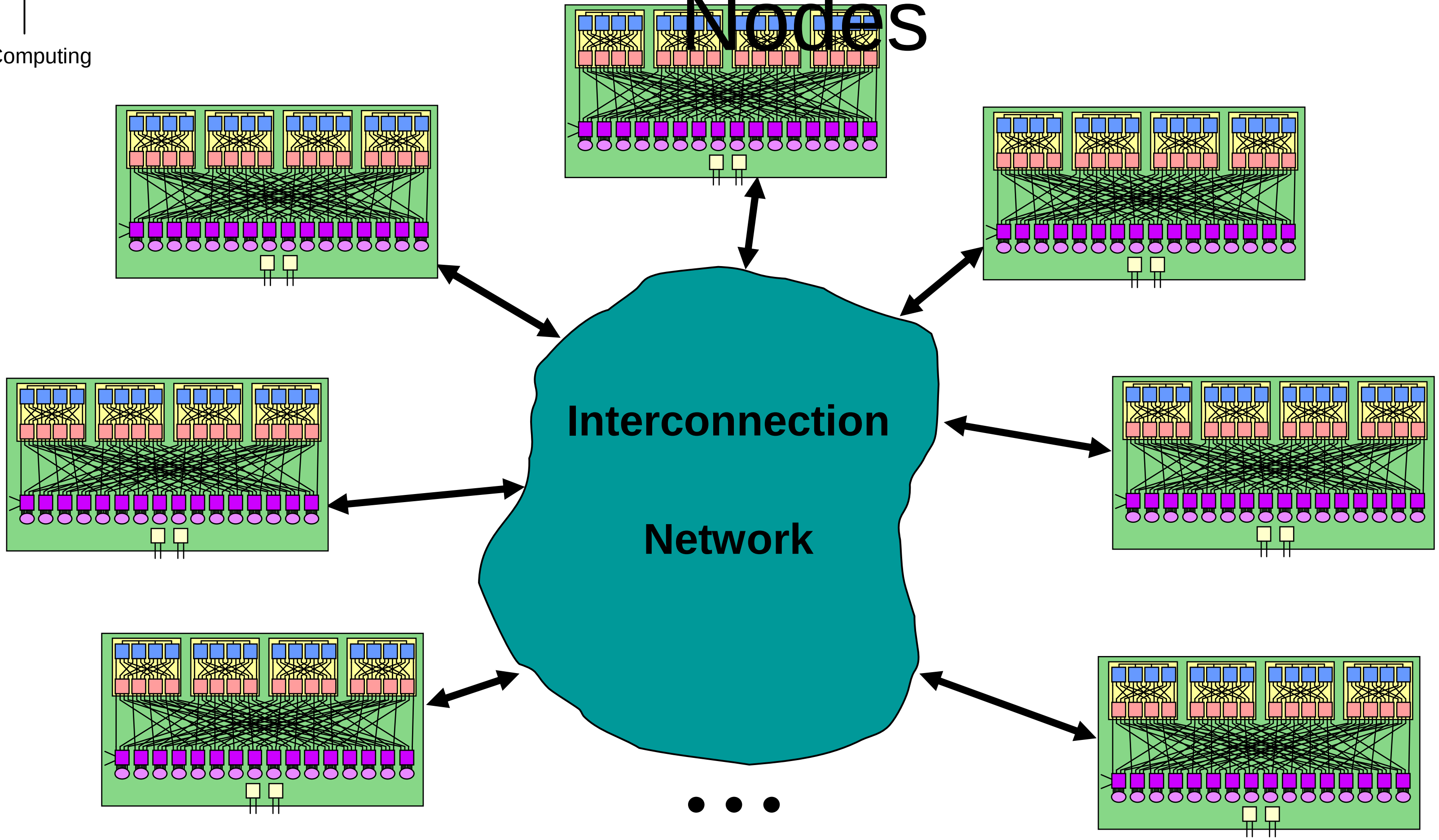


Cray X1 Node



- Four multistream processors (MSPs), each 12.8 Gflops
- High bandwidth local shared memory (128 Direct Rambus channels)
- 32 network links and four I/O links per node

NUMA Scalable up to 1024 Nodes



- 16 parallel networks for bandwidth
- 128 nodes for the ORNL machine

Direct Memory Access (DMA)

- Direct Memory Access (DMA) is a capability provided that allows data to be sent directly from an attached device to the memory on the computer's motherboard.

- The CPU is freed from involvement with the data transfer, thus speeding up overall computer operation

Remote Direct Memory Access (RDMA)

RDMA is a concept whereby two or more computers communicate via Direct memory Access directly from the main memory of one system to the main memory of another .

How Does RDMA Work

- Once the connection has been established, RDMA enables the movement of data from one server directly into the memory of the other server
- RDMA supports “zero copy ,” eliminating the need to copy data between application memory and the data buffers in the operating system.

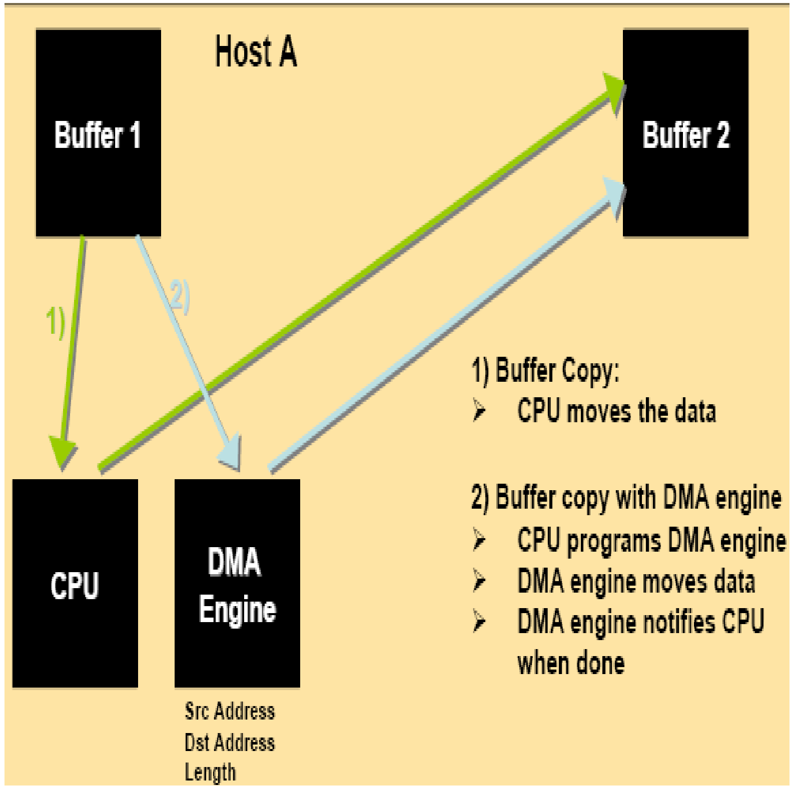
Advantages

- Latency is reduced and applications can transfer messages faster.
- Applications directly issue commands to the adapter without having to execute a Kernel call.
- RDMA reduces demand on the host CPU.

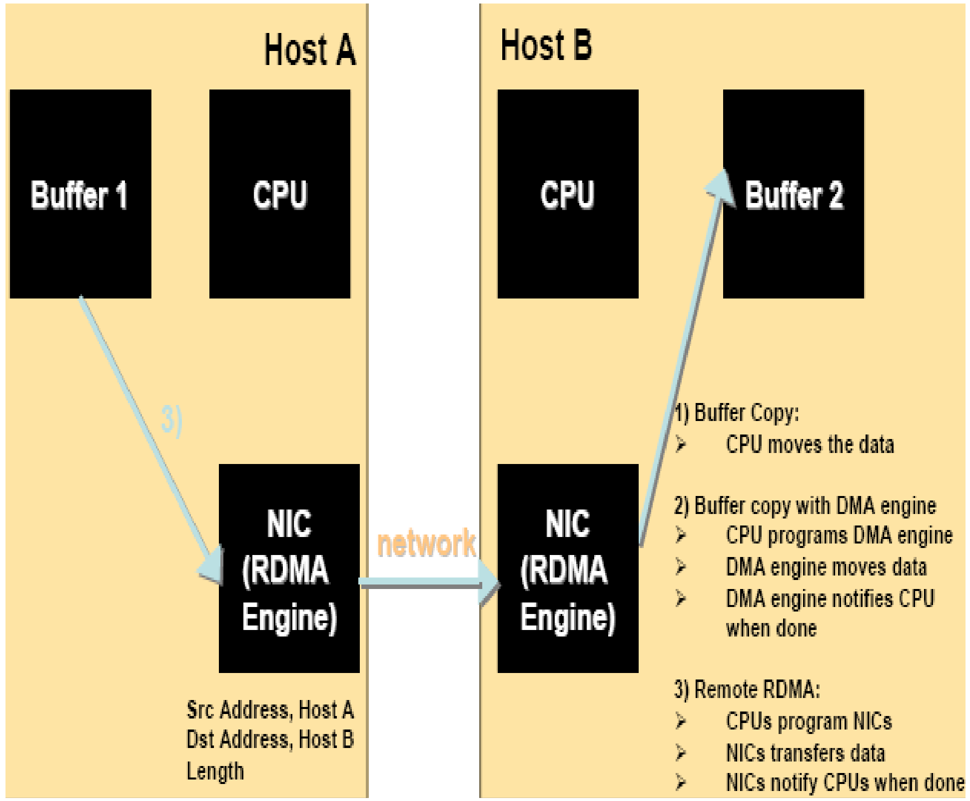
Disadvantages

- Latency is quite high for small transfers
- To avoid kernel calls a VIA adapter must be used

DMA


Host A
Buffer 1
Buffer 2
1)
2)
CPU
DMA Engine
Src Address
Dst Address
Length
1) Buffer Copy:
CPU moves the data
2) Buffer copy with DMA engine
CPU programs DMA engine
DMA engine moves data
DMA engine notifies CPU when done

RDMA


Host A
Buffer 1
CPU
3)
NIC (RDMA Engine)
Src Address, Host A
Dst Address, Host B
Length
network
Host B
CPU
Buffer 2
NIC (RDMA Engine)
1) Buffer Copy:
CPU moves the data
2) Buffer copy with DMA engine
CPU programs DMA engine
DMA engine moves data
DMA engine notifies CPU when done
3) Remote RDMA:
CPUs program NICs
NICs transfers data
NICs notify CPUs when done

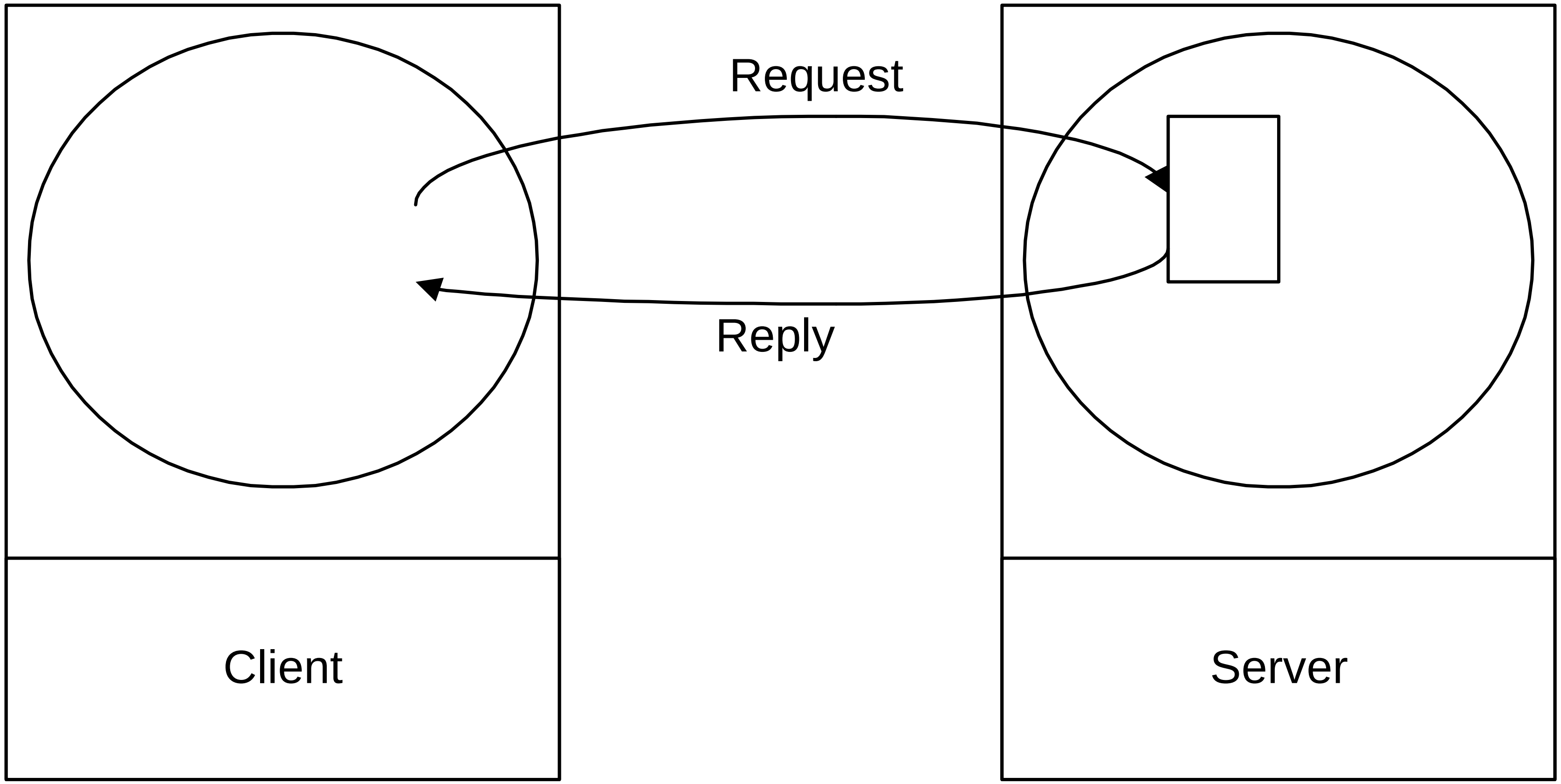
Programming with Remote Memory

RMI/RPC

- Remote Method Invocation/Remote Procedure Call
- Does not provide direct access to remote memory but rather to remote code that can perform the remote memory access
- Widely supported
- Somewhat cumbersome to work with

Cluster Computing

RMI/RPC



RMI

- Setting up RMI is somewhat hard
- Once the system is initialized accessing remote memory is transparent to local object access

Setting up RMI

- Write an interface for the server class
- Write an implementation of the class
- Instantiate the server object
- Announce the server object
- Let the client connect to the object

RMI Interface

```
public interface MyRMIClass extends java.rmi.Remote {
  public void setVal(int value) throws java.rmi.RemoteException;
  public int getVal() throws java.rmi.RemoteException;
}
```

RMI Implementaion

```
public class MyRMIClassImpl
extends UnicastRemoteObject implements MyRMIClass {
    private int iVal;
    public MyRMIClassImpl() throws RemoteException{
        super(); iVal=0;
    }
    public synchronized void setVal(int value) throws java.rmi.RemoteException {
        iVal=value;
    }
    public synchronized int getVal() throws java.rmi.RemoteException {
        return iVal;
    }
}
```

RMI Server Object

```
public class StartMyRMIServer {
 static public void main(String args[])  {
       System.setSecurityManager(new RMISecurityManager());
       try {
Registry reg = java.rmi.registry.LocateRegistry.createRegistry(1099);
           MyRMIClassImpl MY = new MyRMIClassImpl();
           Naming.rebind(”MYSERVER", MY);
       } catch (Exception _) {}
    }
}
```

RMI Client

```
class MYClient {
  static public void main(String [] args){
      String name="//n0/MYSERVER";
      MyRMIClass MY;
      try { MY = (MyRMIClass)java.rmi.Naming.lookup(name);
      } catch (Exception ex) {}
      try {
        System.out.println(”Value is ”+MY.getVal());
        MY.setVal(42);
        System.out.println(”Value is ”+MY.getVal());
    } catch (Exception e){}
  }
}
```

Pyro

- Same as RMI
  - But Python
- Somewhat easier to set up and run

Pyro

```
import Pyro.core
import Pyro.naming
class JokeGen(Pyro.core.ObjBase):
 def joke(self, name):
   return "Sorry "+name+", I don't know any jokes."

daemon=Pyro.core.Daemon()
ns=Pyro.naming.NameServerLocator().getNS()
daemon.useNameServer(ns)
uri=daemon.connect(JokeGen(),"jokegen")
daemon.requestLoop()
```

Pyro

```
import Pyro.core
# finds object automatically if you're running the Name Server.
jokes = Pyro.core.getProxyForURI("PYRONAME://jokegen")
print jokes.joke("Irmen")
```

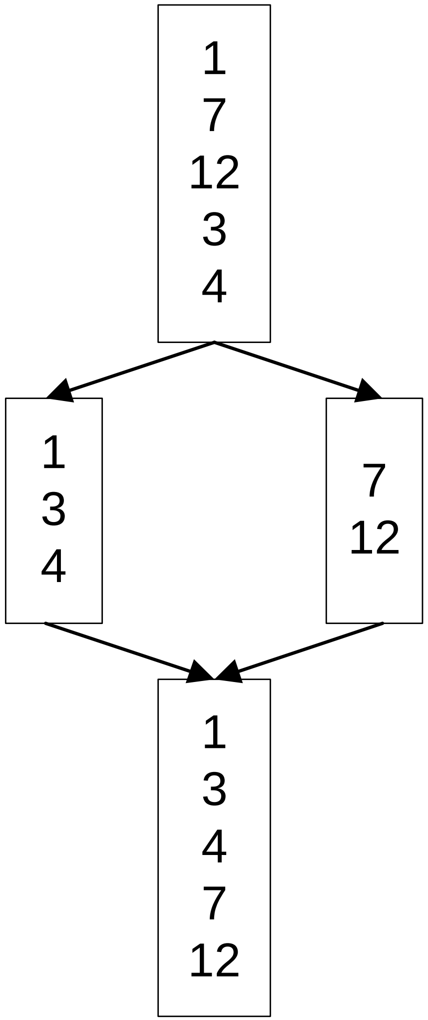
Extend Java Language

- JavaParty : University of Karlsruhe
  - Provides a mechanism for parallel programming on distributed memory machines.
  - Compiler generates the appropriate Java code plus RMI hooks.
  - The remote keywords is used to identify which objects can be called remotely.

JavaParty Hello

```
package examples ;

public remote class HelloJP {

    public void hello() {
        System.out.println(“Hello JavaParty!”) ;
    }

    public static void main(String [] args) {

        for(int n = 0 ; n < 10 ; n++) {
            // Create a remote method on some node
            HelloJP world = new HelloJP() ;

            // Remotely invoke a method
            world.hello() ;
        }
    }
}
```

RMI Example



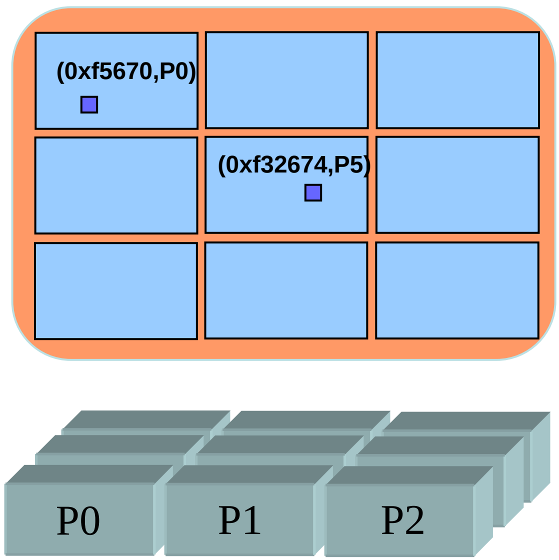
Global Arrays

- Originally designed to emulate remote memory on other architectures – but is extremely popular with actual remote memory architectures

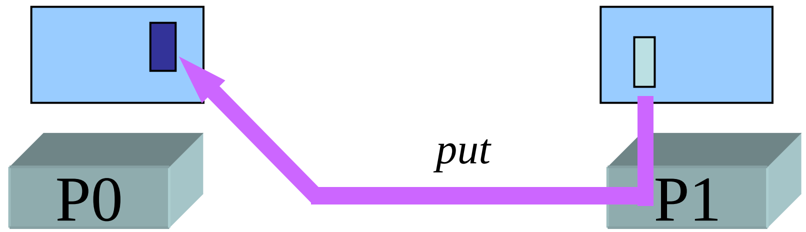
Cluster Computing

Global address space & One-sided communication

collection of address spaces of processes in a parallel job (address, pid)



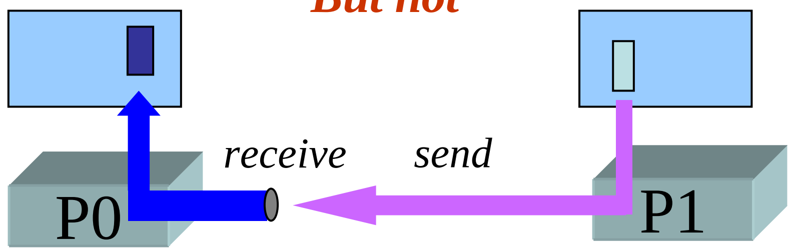
Communication model



one-sided communication

SHMEM, ARMCI, MPI-2-1S

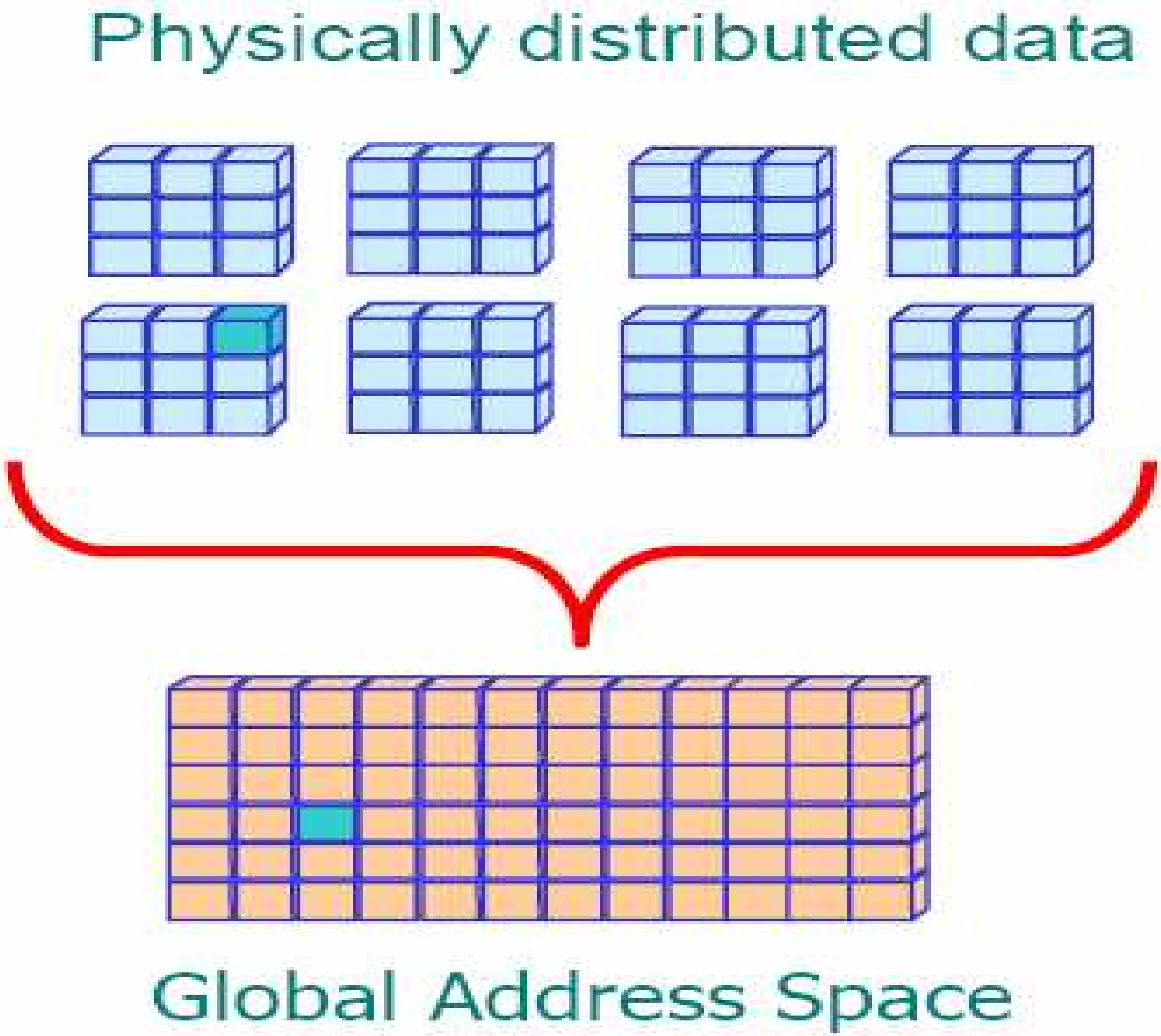
But not



message passing

Cluster Com

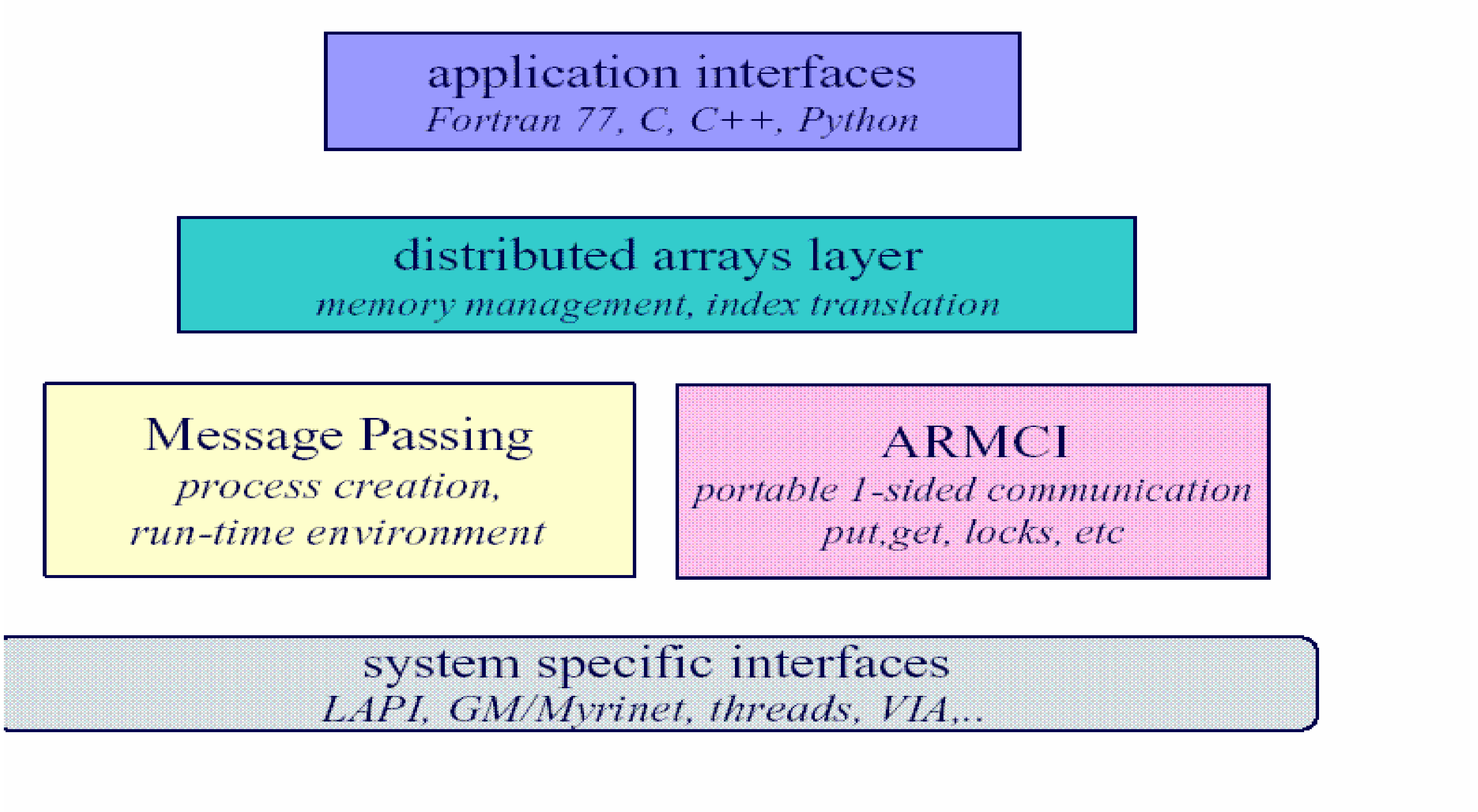
Global Arrays Data Model



Comparison to other models

	Shared memory	Message passing	Global Arrays
Data view	shared	distributed	distributed **or** shared
Access to data	simplest (*a*=*b*)	hard (*send-receive*)	simple (*ga_put/get*)
Data locality information	obscure	explicit	easily available (*ga_disitribution/ ga_locate*)
Scalable performance	limited	very good	very good

Structure of GA



Cluster Computing

GA functionality and Interface

- Collective operations
- One sided operations
- Synchronization
- Utility operations
- Library interfaces

Global Arrays

- Models global memory as user defined arrays
- Local portions of the array can be accessed as native speed
- Access to remote memory is transparent
- Designed with a focus on computational chemistry

Global Arrays

- Synchronous Operations
  - Create an array
  - Create an array, from an existing array
  - Destroy an array
  - Synchronize all processes

Global Arrays

- Asynchronous Operations
  - Fetch
  - Store
  - Gather and scatter array elements
  - Atomic read and increment of an array element

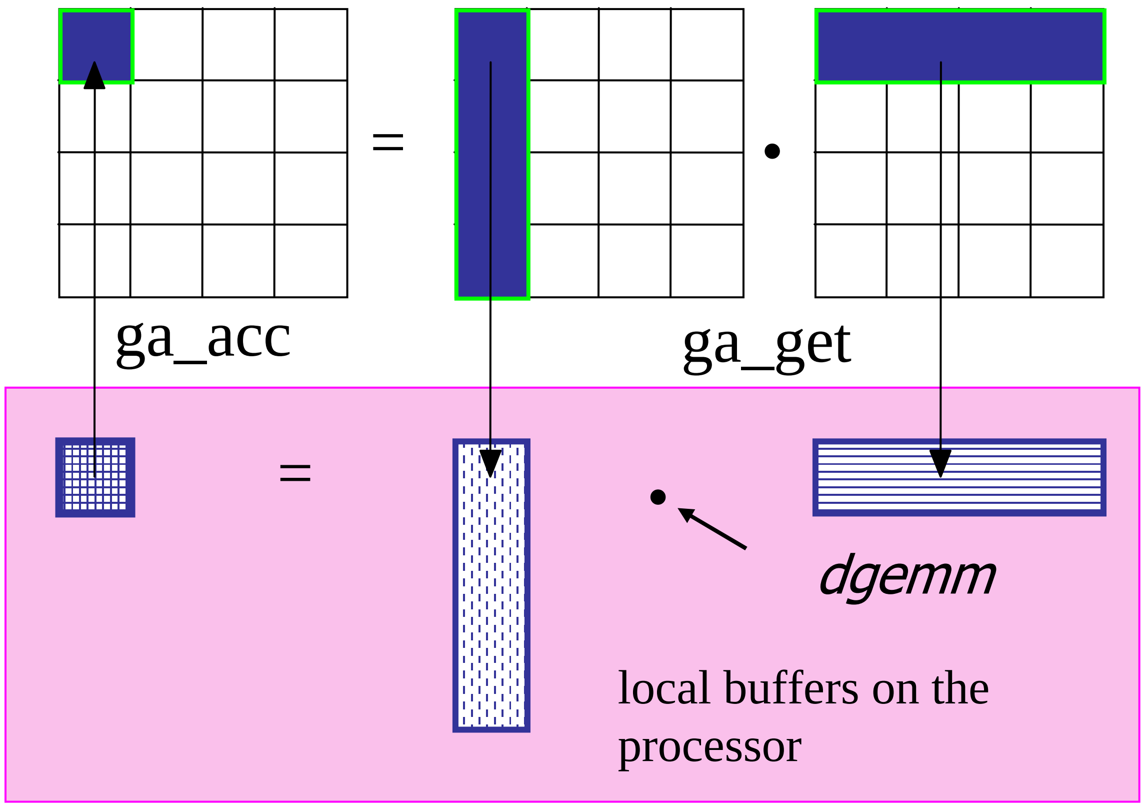
Global Arrays

- BLAS Operations
  - vector operations (dot-product or scale)
  - matrix operations (e.g., symmetrize)
  - matrix multiplication

GA Interface

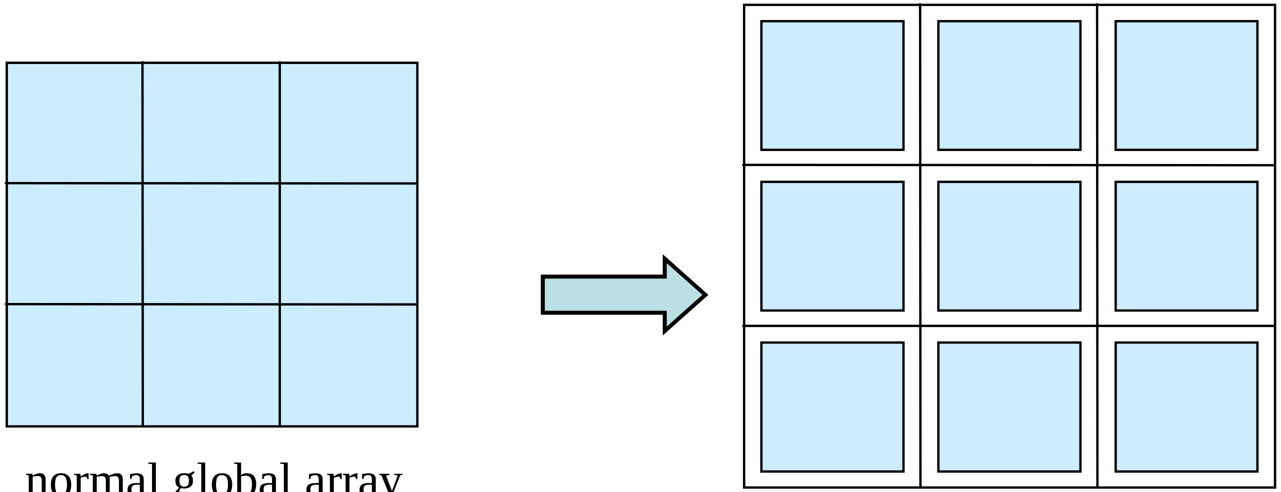
- Collective Operations
  - GA_Initialize, GA_Terminate, GA_Create, GA_Destroy
- One sided operations
  - NGA_Put, NGA_Get
- Remote Atomic operations
  - NGA_Acc, NGA_Read_Inc
- Synchronisation operations
  - GA_Fence, GA_Sync
- Utility Operations
  - NGA_Locate, NGA_Distribution
- Library Interfaces
  - GA_Solve, GA_Lu_Solve

Example: Matrix Multiply



global arrays representing matrices

Ghost Cells



normal global array

global array with ghost cells

- **Operations**

  `NGA_Create_ghosts` - creates array with ghosts cells

  `GA_Update_ghosts` - updates with data from adjacent processors

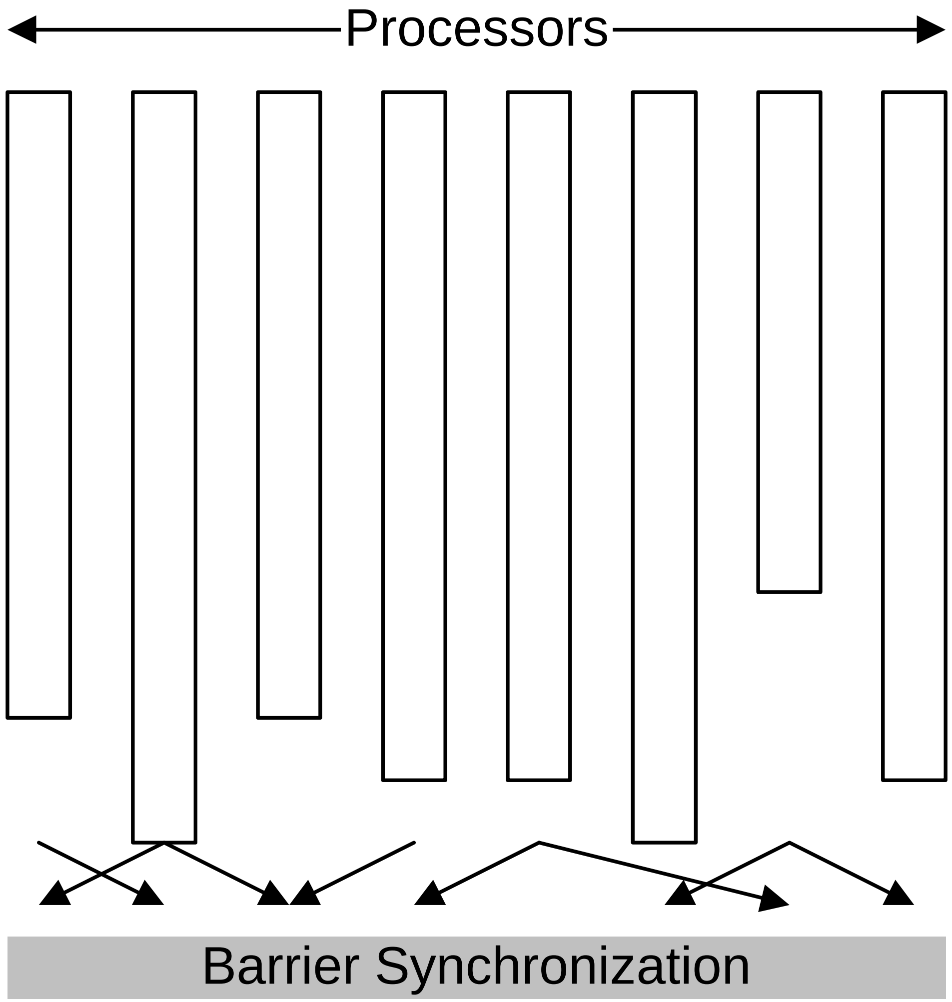
  `NGA_Access_ghosts` - provides access to “local” ghost cell elements

- **Embedded Synchronization - controlled by the user**
- **Multi-protocol implementation to match platform characteristics**
  - **e.g., MPI+shared memory on the IBM SP, SHMEM on the Cray T3E**

BSP

- Bulk Synchronous Parallelism
- Stop ’n Go model similar to OpenMP
- Based on remote memory access
  - Remote memory need not be supported by the hardware

BSP Superstep



BSP Operations

- Initialization
  - bsp_init
  - bsp_start
  - bsp_end
  - bsp_sync
- Misc
  - bsp_pid
  - bsp_nprocs
  - bsp_time

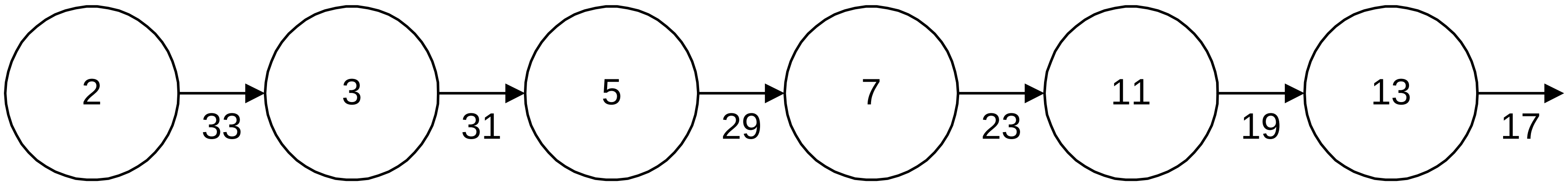
BSP Operations

- DRMA
  - bsp_pushregister
  - bsp_popregister
  - bsp_put
  - bsp_get
- High Performance
  - bsp_hpput
  - bsp_hpget

BSP Operations

- BSMP
  - Bsp_set_tag_size
  - Bsp_send
  - Bsp_get_tag
  - Bsp_move
- High Performance
  - Msb_hpmove

BSP Example



BSP Sieve

```
void bsp_sieve() {
  int i, candidate, prime;
  bsp_pushregister(&candidate,sizeof(int));
  bsp_sync();

  prime=candidate=-1;
  for(i=2; i<100; i++){
    if(bsp_pid()==0)candidate=i;
    else if(prime==-1)prime==candidate;
    if(candidate%prime==0)candidate=-1;
    bsp_put(bsp_pid()+1,&candidate,&candidate,0,sizeof(int));
    bsp_sync();
  }
}
```

MPI-2 and other RMA models

Cray SHMEM

(IBM LAPI, GM, Elan, IBA similar)

Process 0 Process 1

shmem_put

MPI-2 1-Sided “active target”

Process 0 Process 1

MPI_Win_Post

MPI_Win_Start

MPI_Put

MPI_Win_Complete

MPI_Win_Wait

MPI-2 1-Sided “passive target”

Process 0 Process 1

MPI_Win_Lock

MPI_Put

MPI_Win_Unlock

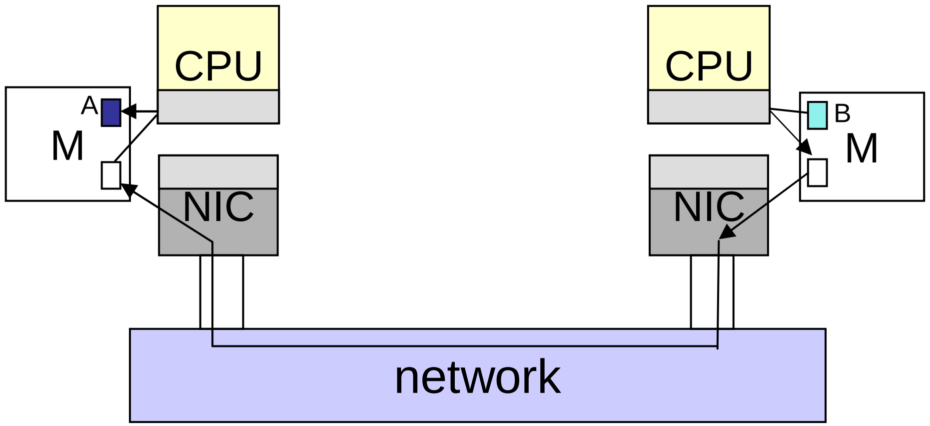
(Note: lock and put can be combined in networks that support active messages like IBM LAPI or sophisticated, user programmable adapters like Quadrics)

———— data transfer

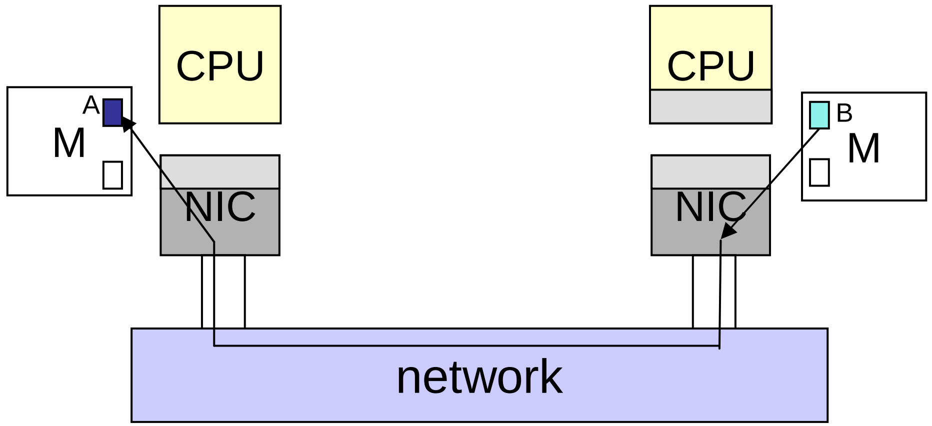
- - - - - - synchronization

- MPI-2 1-sided is more synchronous than native RMA protocols
- Other RMA models decouple synchronization from data transfer

Data Movement



copy-based, high CPU involvement e.g., IBM SP



zero-copy, low CPU involvement e.g., Quadrics

- These are two ends of the spectrum
  - Consider commodity hpc networks (Myrinet, IBA)
    - MPI tries to “register” user buffers with NIC on the fly
      - after handshaking between sender and receiver are zero-copy
      - NIC does handle MPI tag matching and queue management
    - RMA model is more favorable than MPI on these networks
      - once the user registers communication buffer
      - Put/get operations handled by DMA engines on the NIC
      - No need to involve remote CPU