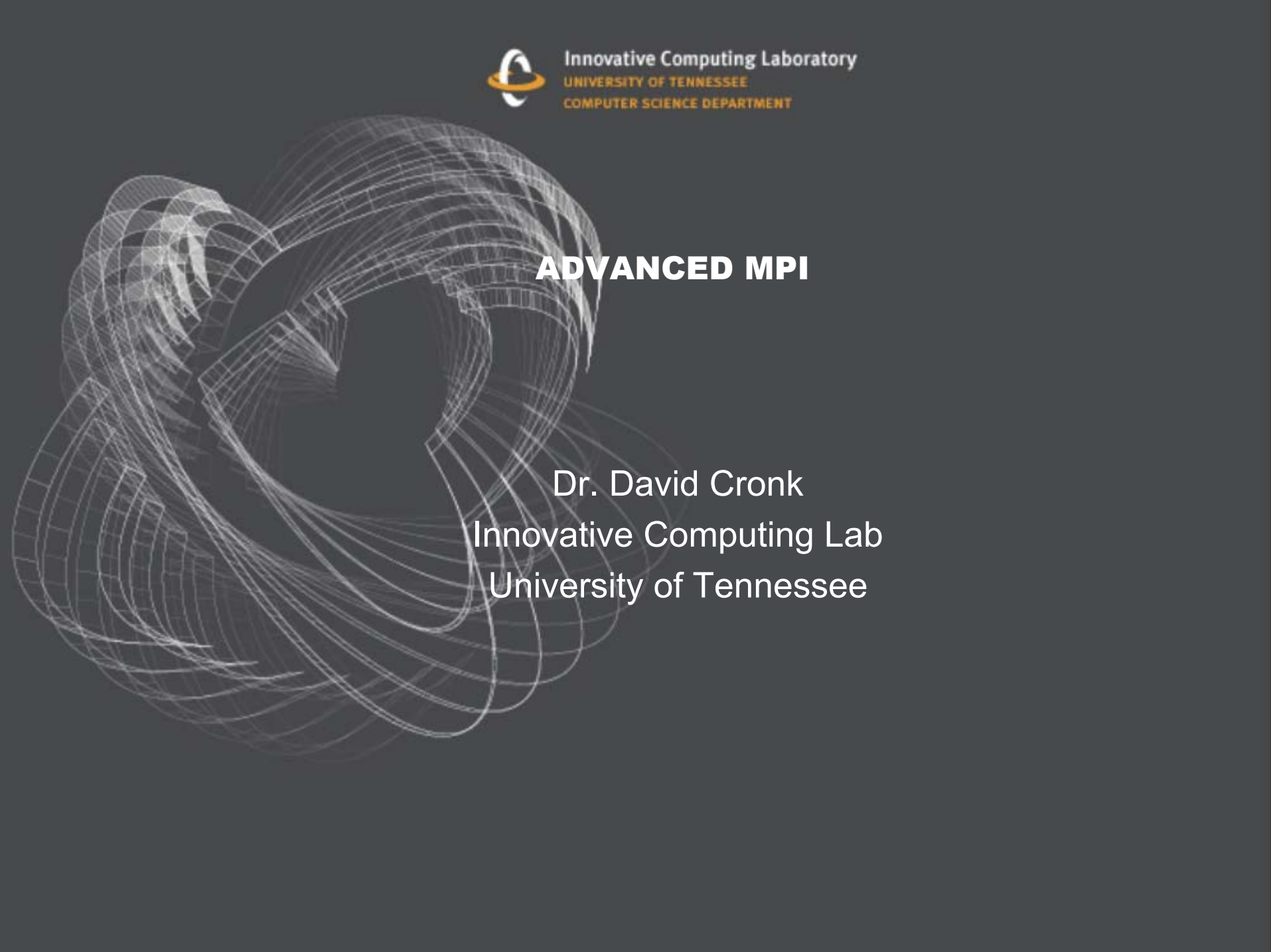


ADVANCED MPI

Dr. David Cronk
Innovative Computing Lab
University of Tennessee

Course Outline

- Day 1
  - Morning - Lecture
    - Communicators/Groups
    - Extended Collective Communication
    - One-sided Communication
  - Afternoon - Lab
    - Hands on exercises demonstrating the use of groups, extended collectives, and one-sided communication

Course Outline (cont)

Day 2

- Morning – Lecture
  - MPI-I/O
- Afternoon – Lab
  - Hands on exercises using MPI-I/O

bCourse Outline (cont)

- Day 3
  - Morning – Lecture
    - Performance Analysis of MPI programs
      - TAU
      - Vampir/VampirTrace
  - Afternoon – Lab
    - Hands on exercises using Vampir and VampirTrace

Communicators and Groups

Introduction

Group Management

Communicator Management

Intercommunicators

Communicators and Groups

- Many MPI users are only familiar with the communicator MPI_COMM_WORLD
- A communicator can be thought of a handle to a group
- A group is an ordered set of processes
  - Each process is associated with a rank
  - Ranks are contiguous and start from zero
- For many applications (dual level parallelism) maintaining different groups is appropriate
- Groups allow collective operations to work on a subset of processes
- Information can be added onto communicators to be passed into routines

Communicators and Groups(cont)

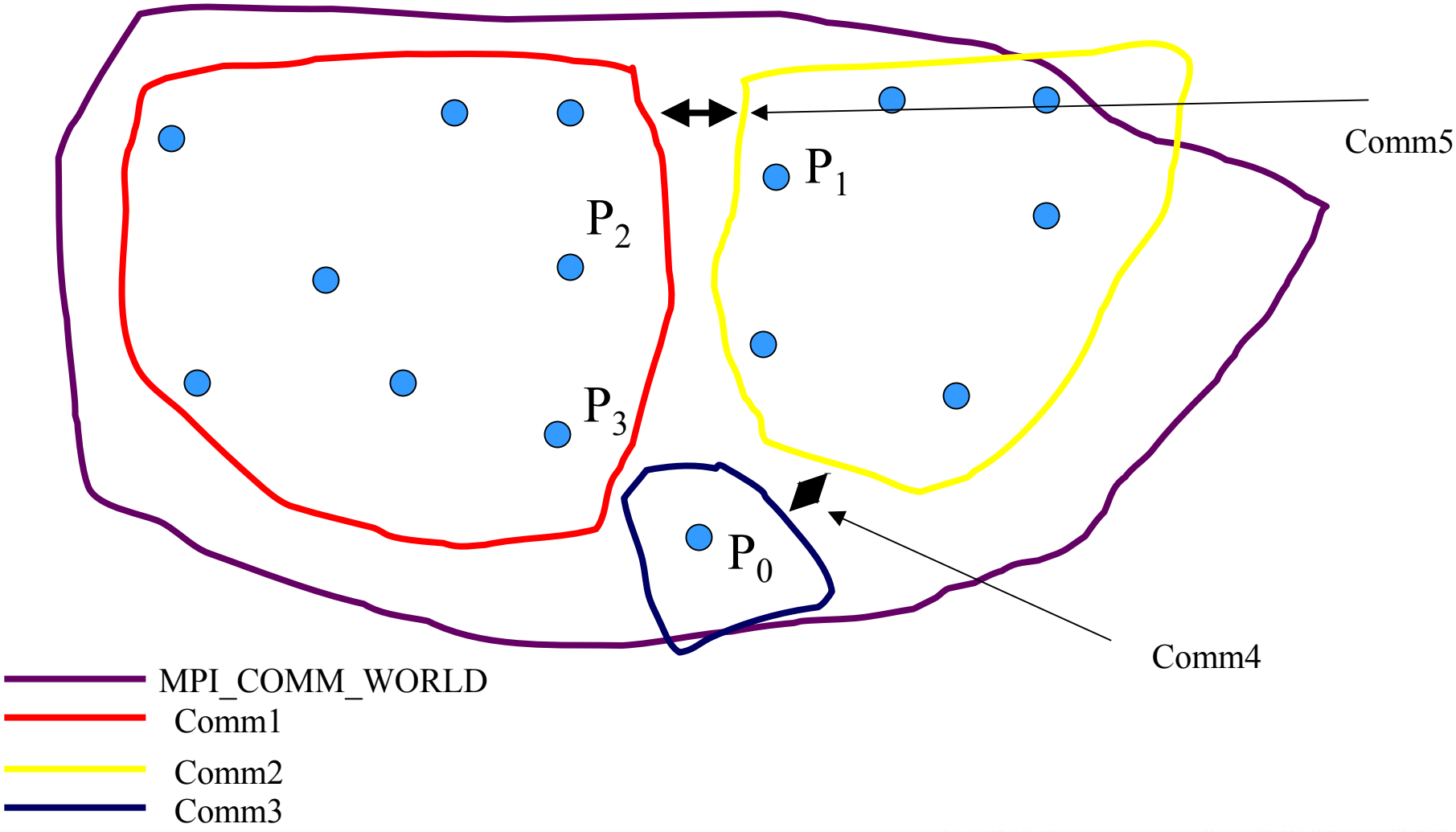
While we think of a communicator as spanning processes, it is actually unique to a process

A communicator can be thought of as a handle to an object (group attribute) that describes a group of processes

An intracommunicator is used for communication within a single group

An intercommunicator is used for communication between 2 disjoint groups

Communicators and Groups(cont)



Communicators and Groups(cont)

Refer to previous slide

- There are 4 distinct groups
- These are associated with intracommunicators MPI_COMM_WORLD, comm1, and comm2, and comm3
- P_3 is a member of 2 groups and may have different ranks in each group(say 3 & 4)
- If P_2 wants to send a message to P_1 it must use MPI_COMM_WORLD (intracommunicator) or comm5 (intercommunicator)
- If P_2 wants to send a message to P_3 it can use MPI_COMM_WORLD (send to rank 3) or comm1 (send to rank 4)
- P_0 can broadcast a message to all processes associated with comm2 by using intercommunicator comm5

Group Management

All group operations are local

As will be clear, groups are initially not associated with communicators

Groups can only be used for message passing within a communicator

We can access groups, construct groups, and destroy groups

Group Accessors

MPI_GROUP_SIZE(group, size, ierr)

- MPI_Group group int size (C)
- INTEGER group, size, ierr (Fortran)
- This routine returns the number of processes in the group

MPI_GROUP_RANK(group, rank, ierr)

- MPI_Group group int rank (C)
- INTEGER group, rank, ierr (Fortran)
- This routine returns the rank of the calling process

Group Accessors (cont)

MPI_GROUP_TRANSLATE_RANKS (group1, n, ranks1, group2, ranks2, ierr)

- MPI_Group group1, group2 int n, *ranks1, *ranks2
- INTEGER group1, n, ranks(), group2, ranks(), ierr
- This routine takes an array of n ranks (ranks1) which are ranks of processes in group1. It returns in ranks2 the corresponding ranks of the processes as they are in group2
- MPI_UNDEFINED is returned for processes not in group2

Groups Accessors (cont)

MPI_GROUP_COMPARE (group1, group2 result, ierr)

- MPI_Group group1, group2 int result
- INTEGER group1, group2, result, ierr (Fortran)
- This routine returns the relationship between group1 and group2
- If group1 and group2 contain the same processes, ranked the same way, this routine returns MPI_IDENT
- If group1 and group2 contain the same processes, but ranked differently, this routine returns MPI_SIMILAR
- Otherwise this routine returns MPI_UNEQUAL

Group Constructors

- Group constructors are used to create new groups from existing groups
- Base group is the group associated with MPI_COMM_WORLD (use mpi_comm_group to get this)
- Group creation is a local operation
  - No communication needed
- Following group creation, no communicator is associated with the group
  - No communication possible with new group
- Each process in a new group MUST create the group so it is identical!
- Groups are created through some communicator creation routines covered later

Group Constructors (cont)

MPI_COMM_GROUP (comm, group, ierr)

- MPI_Comm comm MPI_Group group (c)
- INTEGER comm, group, ierr (Fortran)
- This routine returns in *group* the group associated with the communicator comm

Group Constructors (cont)
Set Operations

MPI_GROUP_UNION(group1, group2, newgroup, ierr)

MPI_GROUP_INTERSECTION(group1, group2, newgroup, ierr)

MPI_GROUP_DIFFERENCE(group1, group2, newgroup, ierr)

MPI_Group group1, group2, *newgroup (C)

INTEGER group1, group2, newgroup, ierr (Fortran)

Group Constructors (cont)
Set Operations

Union: Returns in newgroup a group consisting of all processes in group1 followed by all processes in group2, with no duplication

Intersection: Returns in newgroup all processes that are in both groups, ordered as in group1

Difference: Returns in newgroup all processes in group1 that are not in group2, ordered as in group1

Group Constructors (cont)
Set Operations

Let group1 = {a,b,c,d,e,f,g} and group2 = {d,g,a,c,h,I}

MPI_Group_union(group1,group2,newgroup)

Newgroup = {a,b,c,d,e,f,g,h,I}

MPI_Group_intersection(group1,group2,newgroup)

Newgroup = {a,c,d,g}

MPI_Group_difference(group1,group2,newgroup)

Newgroup = {b,e,f}

Group Constructors (cont)
Set Operations

Let group1 = {a,b,c,d,e,f,g} and group2 = {d,g,a,c,h,I}

MPI_Group_union(group2,group1,newgroup)

Newgroup = {d,g,a,c,h,I,b,e,f}

MPI_Group_intersection(group2,group1,newgroup)

Newgroup = {d,g,a,c}

MPI_Group_difference(group2,group1,newgroup)

Newgroup = {h,i}

Group Constructors (cont)

MPI_GROUP_INCL(group, n, ranks, newgroup, ierr)

- MPI_Group group, *newgroup int n, *ranks
- INTEGER group, n, ranks(), newgroup, ierr
- This routine creates a new group that consists of all the n processes with ranks ranks[0]..ranks[n-1]
- The process with rank i in newgroup has rank ranks[i] in group

Group Constructors (cont)

MPI_GROUP_EXCL(group, n, ranks, newgroup, ierr)

- MPI_Group group, *newgroup int n, *ranks
- INTEGER group, n, ranks(), newgroup, ierr
- This routine creates a new group that consists of all the processes in group after deleting processes with ranks ranks[0]..ranks[n-1]
- The relative ordering in newgroup is identical to the ordering in group

Group Constructors (cont)

MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup, ierr)

- MPI_Group group, *newgroup int n, ranges[][3]
- INTEGER group, n, ranges(*,3), newgroup, ierr)
- Ranges is an array of triplets consisting of start rank, end rank, and stride
- Each triplet in ranges specifies a sequence of ranks to be included in newgroup
- The ordering in newgroup is as specified by ranges

Group Constructors (cont)

MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup, ierr)

- MPI_Group group, *newgroup int n, ranges[][3]
- INTEGER group, n, ranges(*,3), newgroup, ierr)
- Ranges is an array of triplets consisting of start rank, end rank, and stride
- Each triplet in ranges specifies a sequence of ranks to be excluded from newgroup
- The ordering in newgroup is identical to that in group

Group Constructors (cont)

Let group = {a,b,c,d,e,f,g,h,i,j}

n=5, ranks = {0,3,8,6,2}

ranges= {(4,9,2),(1,3,1),(0,9,5)}

MPI_Group_incl(group,n,ranks,newgroup)

newgroup = {a,d,I,g,c}

MPI_Group_excl(group,n,ranks,newgroup)

newgroup = {b,e,f,h,j}

MPI_Group_range_incl(group,n,ranges,newgroup)

newgroup = {e,g,I,b,c,d,a,f}

MPI_Group_range_excl(group,n,ranges,newgroup)

newgroup = {h}

Communicator Management

Communicator access operations are local, thus requiring no interprocess communication

Communicator constructors are collective and may require interprocess communication

All the routines in this section are for intracommunicators, intercommunicators will be covered separately

Communicator Accessors

MPI_COMM_SIZE (MPI_Comm comm, int size, ierr)

- Returns the number of processes in the group associated with comm

MPI_COMM_RANK (MPI_Comm comm, int rank, ierr)

- Returns the rank of the calling process within the group associated with comm

MPI_COMM_COMPARE (MPI_Comm comm1, MPI_Comm comm2, int result, ierr) returns:

- MPI_IDENT if comm1 and comm2 are handles for the same object
- MPI_CONGRUENT if comm1 and comm2 have the same group attribute
- MPI_SIMILAR if the groups associated with comm1 and comm2have the same members but in different rank order
- MPI_UNEQUAL otherwise

Communicator Constructors

MPI_COMM_DUP (MPI_Comm comm, MPI_Comm newcomm, ierr)

This routine creates a duplicate of comm

newcomm has the same fixed attributes as comm

Defines a new communication domain

- A call to MPI_Comm_compare (comm, newcomm, result) would return MPI_IDENT

Useful to library writers and library users

Communicator Constructors

MPI_COMM_CREATE (MPI_Comm comm,MPI_Group group, MPI_Comm newcomm, ierr)

- This is a collective routine, meaning it must be called by all processes in the group associated with comm
- This routine creates a new communicator which is associated with group
- MPI_COMM_NULL is returned to processes not in group
- All group arguments must be the same on all calling processes
- group must be a subset of the group associated with comm

Communicator Constructors

```
CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)
CALL MPI_COMM_SIZE (MPI_COMM_WORLD, size, ierr)
CALL MPI_COMM_GROUP (MPI_COMM_WORLD, wgroup, ierr)

ranges (1,1) = 10
ranges(1,2) = size-1
ranges(1,3) = 1
CALL MPI_GROUP_RANGE_INCL (wgroup, 1, ranges, newgroup, ierr)
CALL MPI_COMM_CREATE (MPI_COMM_WORLD, newgroup, newcom, ierr)
```

newgroup is set to MPI_COMM_NULL in processes 0 through 9 of MPI_COMM_WORLD

Communicator Constructors

```
CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)
CALL MPI_COMM_SIZE (MPI_COMM_WORLD, size, ierr)
CALL MPI_COMM_GROUP (MPI_COMM_WORLD, wgroup, ierr)

ranges (1,1) = 10
ranges(1,2) = size-1
ranges(1,3) = 1
CALL MPI_GROUP_RANGE_INCL (wgroup, 1, ranges, newgroup, ierr)
CALL MPI_COMM_CREATE (MPI_COMM_WORLD, newgroup, newcom, ierr)
CALL MPI_GROUP_RANGE_EXCL (wgroup, 1, ranges, newgroup, ierr)
CALL MPI_COMM_CREATE (MPI_COMM_WORLD, newgroup, newcom, ierr)
```

Communicator Constructors

MPI_COMM_SPLIT(MPI_Comm comm, int color, int key, MPI_Comm newcomm, ierr)

- MPI_Comm comm, newcomm int color, key
- INTEGER comm, color, key, newcomm, err
- This routine creates as many new groups and communicators as there are distinct values of color
- The rankings in the new groups are determined by the value of key, ties are broken according to the ranking in the group associated with comm
- MPI_UNDEFINED is used as the color for processes to not be included in any of the new groups

Communication Constructors

Rank	0	1	2	3	4	5	6	7	8	9	10
Process	a	b	c	d	e	f	g	h	i	j	k
Color	U	3	1	1	3	7	3	3	1	U	3
Key	0	1	2	3	1	9	3	8	1	0	0

Both process a and j are returned MPI_COMM_NULL

3 new groups are created

{i, c, d}

{k, b, e, g, h}

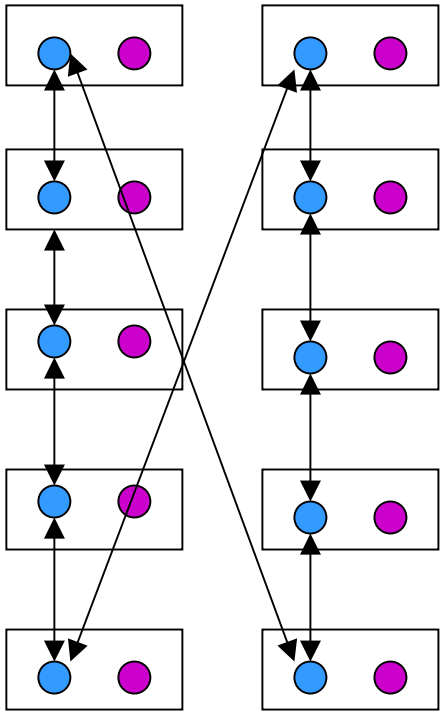
{f}

Communication Constructors

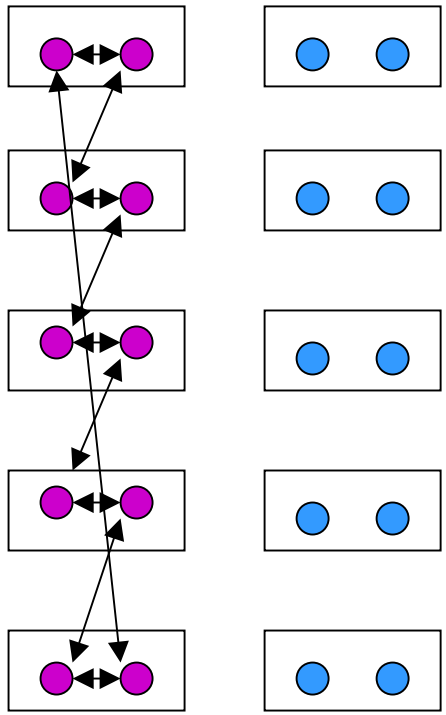
```
CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)

IF (myrank .ge. 10) THEN
  color = 0
ELSE
  color = MPI_UNDEFINED
ENDIF

CALL MPI_COMM_SPLIT (MPI_COMM_WORLD, color, 1, newcomm, ierr)
```

Communication Constructors

```
CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)

IF (myrank .ge. 10) THEN
  color = 0
ELSE
  color = 1
ENDIF

CALL MPI_COMM_SPLIT (MPI_COMM_WORLD, color, 1, newcomm, ierr)
```



- Group 1
- Group 2



Destructors

The communicators and groups from a process' viewpoint are merely handles

Like all handles in MPI, there is a limited number available – YOU CAN RUN OUT

MPI_GROUP_FREE (MPI_Group group, ierr)

MPI_COMM_FREE (MPI_Comm comm, ierr)

Intercommunicators

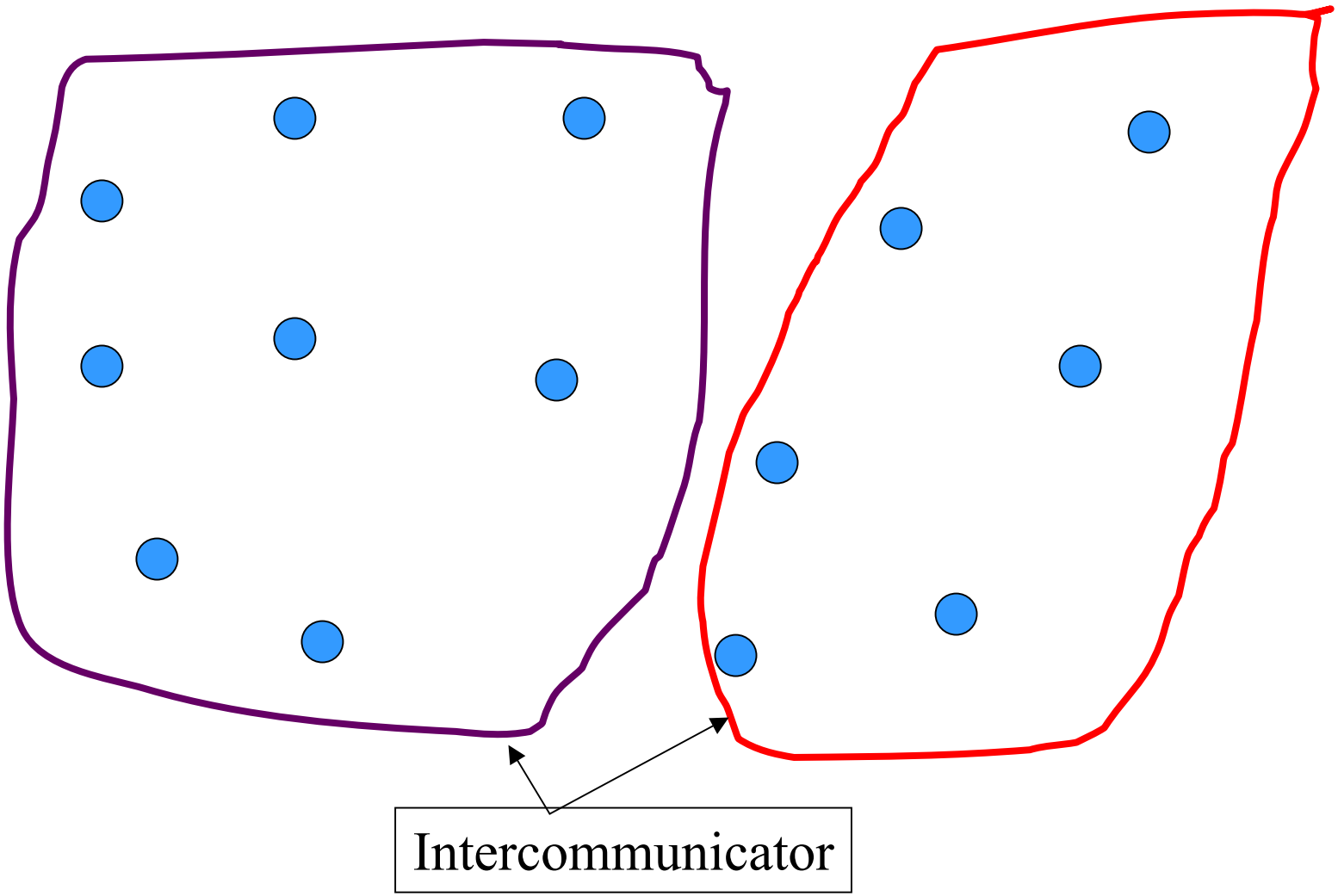
Intercommunicators are associated with 2 groups of disjoint processes

Intercommunicators are associated with a remote group and a local group

The target process (destination for send, source for receive) is its rank in the remote group

A communicator is either intra or inter, never both

Intercommunicators



Intercommunicator Accessors

MPI_COMM_TEST_INTER (MPI_Comm comm, int flag, ierr)

- This routine returns true if comm is an intercommunicator, otherwise, false

MPI_COMM_REMOTE_SIZE(MPI_Comm comm, int size, ierr)

- This routine returns the size of the remote group associated with intercommunicator comm

MPI_COMM_REMOTE_GROUP(MPI_Comm comm, MPI_Groupgroup, ierr)

- This routine returns the remote group associated with intercommunicator comm

Intercommunicator Constructors

The communicator constructors described previously will return an intercommunicator if the are passed intercommunicators as input

- MPI_COMM_DUP: returns an intercommunicator with the same groups as the one passed in
- MPI_COMM_CREATE: each process in group A must pass in group the same subset of group A (A1). Same for group B (B1). The new communicator has groups A1 and B1 and is only valid on processes in A1 and B1
- MPI_COMM_SPLIT: As many new communicators as there are distinct pairs of colors are created

MPI_COMM_CREATE

Intercomm1 is an intercommunicator that relates to groups A = {a,b,c,d,e,f,g,h,I,j} and groups B = {k,l,m,n,o,p,q,r,s,t}

All processes in group A create a new group A' = {f, g, h, I, j}

All processes in group B create a new group B' = {p, q, r, s, t}

All processes in group A call MPI_Comm_create with comm=intercomm1 and group = A'

All processes in group B call MPI_Comm_create with comm=intercomm1 and group = B'

Processes {a,b,c,d,e, and k,l,m,n,o} are each returned newcomm = MPI_COMM_NULL

All processes in A' are returned an intercommunicator with A' as the local group and B' as the remote group

All processes in B' are returned an intercommunicator with B' as the local group and A' as the remote group

MPI_COMM_SPLIT

Rank	0	1	2	3	4	5	6	7	8	9	10
Process	a	b	c	d	e	f	g	h	i	j	k
Color	U	3	3	1	1	7	3	3	1	U	3
Key	0	1	2	3	1	9	3	8	1	0	0

Group A

Rank	0	1	2	3	4	5	6	7	8	9	10
Process	l	m	n	o	p	q	r	s	t	u	v
Color	5	3	1	U	3	3	3	7	1	U	3
Key	0	1	2	3	1	9	3	8	1	0	0

Group B

MPI_COMM_SPLIT

Processes a, j, l, o, and u would all have MPI_COMM_NULL returned in newcomm

newcomm1 would be associated with 2 groups: {e, i, d} and {t, n}

newcomm2 would be associated with 2 groups: {k, b, c, g, h} and {v, m, p, r, q}

newcomm3 would be associated with 2 groups: {f} and {s}

Intercommunicator Constructors

MPI_INTERCOMM_CREATE (local_comm, local_leader, bridge_comm, remote_leader, tag, newintercomm, ierr)

This routine is called collectively by all processes in 2 disjoint groups

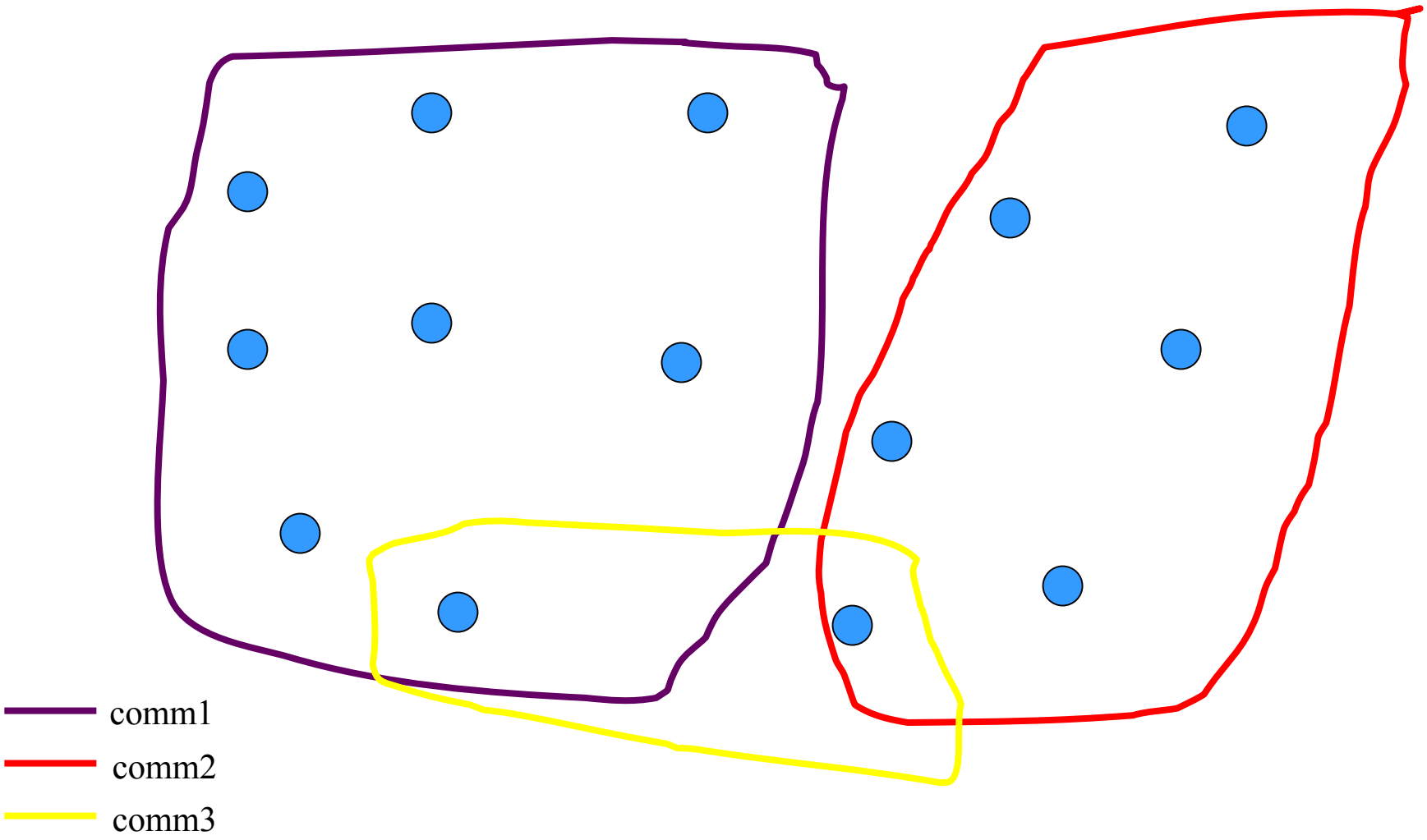
All processes in a particular group must provide matching local_comm and local_leader arguments

The local leaders provide a matching bridge_comm (a communicator through which they can communicate), in remote_leader the rank of the other leader within bridge_comm, and the same tag

The bridge_comm, remote_leader, and tag are significant only at the leaders

There must be no pending communication across bridge_comm that may interfere with this call

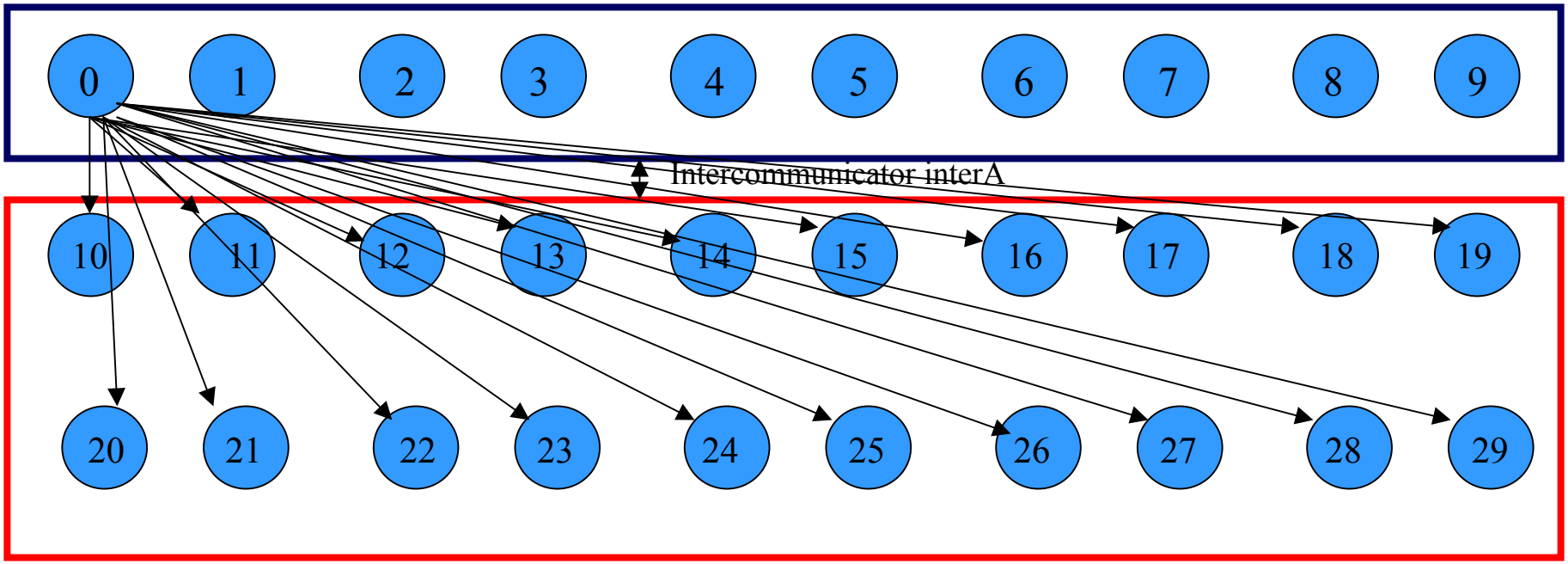
Intercommunicators



Communication Constructors

```
CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)

IF (myrank .ge. 10) THEN
  color = 0
ELSE
  color = 1
ENDIF

CALL MPI_COMM_SPLIT (MPI_COMM_WORLD, color, 1, newcomm, ierr)
CALL MPI_INTERCOMM_CREATE (newcom, 0, MPI_COMM_WORLD,
        0, 111, newintercomm, ierr)
```

Now processes in each group can communicate with the intercommunicator.
For instance, process 0 of MPI_COMM_WORLD can broadcast
a value to all the processes with rank >= 10 in MPI_COMM_WORLD



MPI_COMM_SPLIT (…..)

MPI_INTERCOMM_CREATE(..)

Process 0 call MPI_BCAST with interA….

Group of masters

Group of slaves

Intercommunicators

MPI_INTERCOMM_MERGE (MPI_Comm intercomm, int high, MPI_Comm newintracomm, ierr)

- This routine creates an intracommunicator from a union of the two groups associated with intercomm
- High is used for ordering. All process within a particular group must pass the same value in for high (true or false)
- The new intracommunicator is ordered with the high processes following the low processes
- If both groups pass the same value for high, the ordering is arbitrary

TAKE A BREAK

Extended Collective Communication

- The original MPI standard did not allow for collective communication across intercommunicators
- MPI-2 introduced this capability
- Useful in pipelined algorithms where data needs to be moved from one group of processes to another

Three types of collective

Rooted:

- MPI_Gather and MPI_Gatherv
- MPI_Reduce
- MPI_Scatter and MPI_Scatterv
- MPI_Bcast

All-to-all:

- MPI_Allgather and MPI_Allgatherv
- MPI_Alltoall, MPI_Alltoallv, and MPI_Alltoallw
- MPI_Allreduce, MPI_Reduce_scatter

Other:

- MPI_Scan, MPI_Exscan, and MPI_Barrier

Data movement in extended collectives

Rooted:

- One group (root group) contains the root process while the other group (leaf group) has no root
- Data moves from the root to all the processes in the leaf group (one-to-all) or vice-versa (all-to-one)
- The root process uses MPI_ROOT for its root argument while all other processes in the root group pass MPI_PROC_NULL
- All processes in the leaf group pass the rank of the root relative to the root group

Data movement in extended collectives

All-to-all

Data sent by processes in group A are received by processes in group B while data sent by processes in group B are received by processes in group A

MPI_Barrier (comm, ierr)

Syntactically identical to a situation where all processes are in the same group and call a barrier with the intracommunicator associated with said group

That is, all processes in group A may exit the barrier after all processes in group B have entered the call, and vice-versa

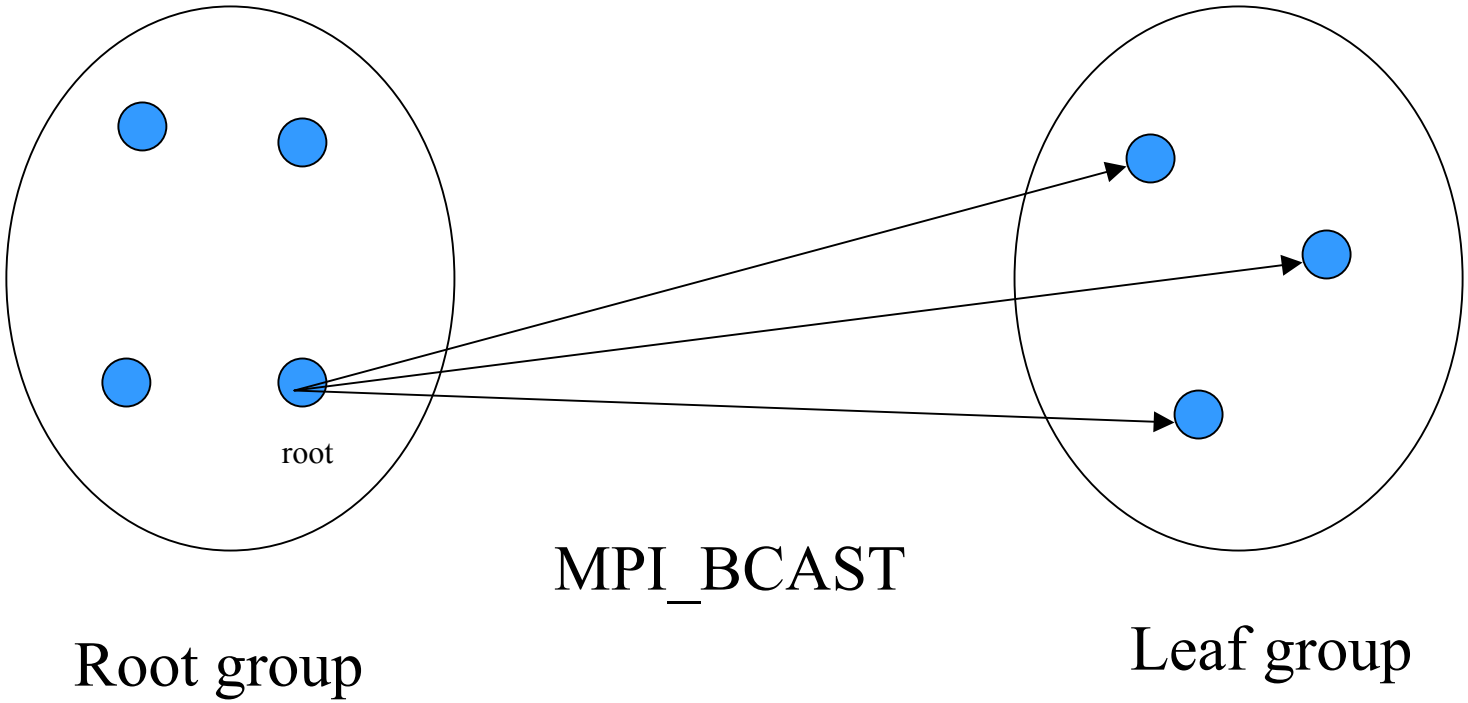
MPI_BCAST (buff, count, dtype, root, comm, ierr)

Data is broadcast from the root to all processes in the leaf group

Root group: Root process passes MPI_ROOT for the *root* argument while others pass MPI_PROC_NULL. B*uff, count, and dtype* are not significant in non-root processes

Leaf group: All processes pass the same argument in *root*, which is the rank of the root process in the root group. *count* and *type* must be consistent with *count* and *type* on the root

MPI_Bcast



MPI_Gather (sbuf, scount, stype, rbuf,rcount,rtype,root, comm,ierr)

Data is gathered in rank order from all the processes in the leaf group into rbuf of the root

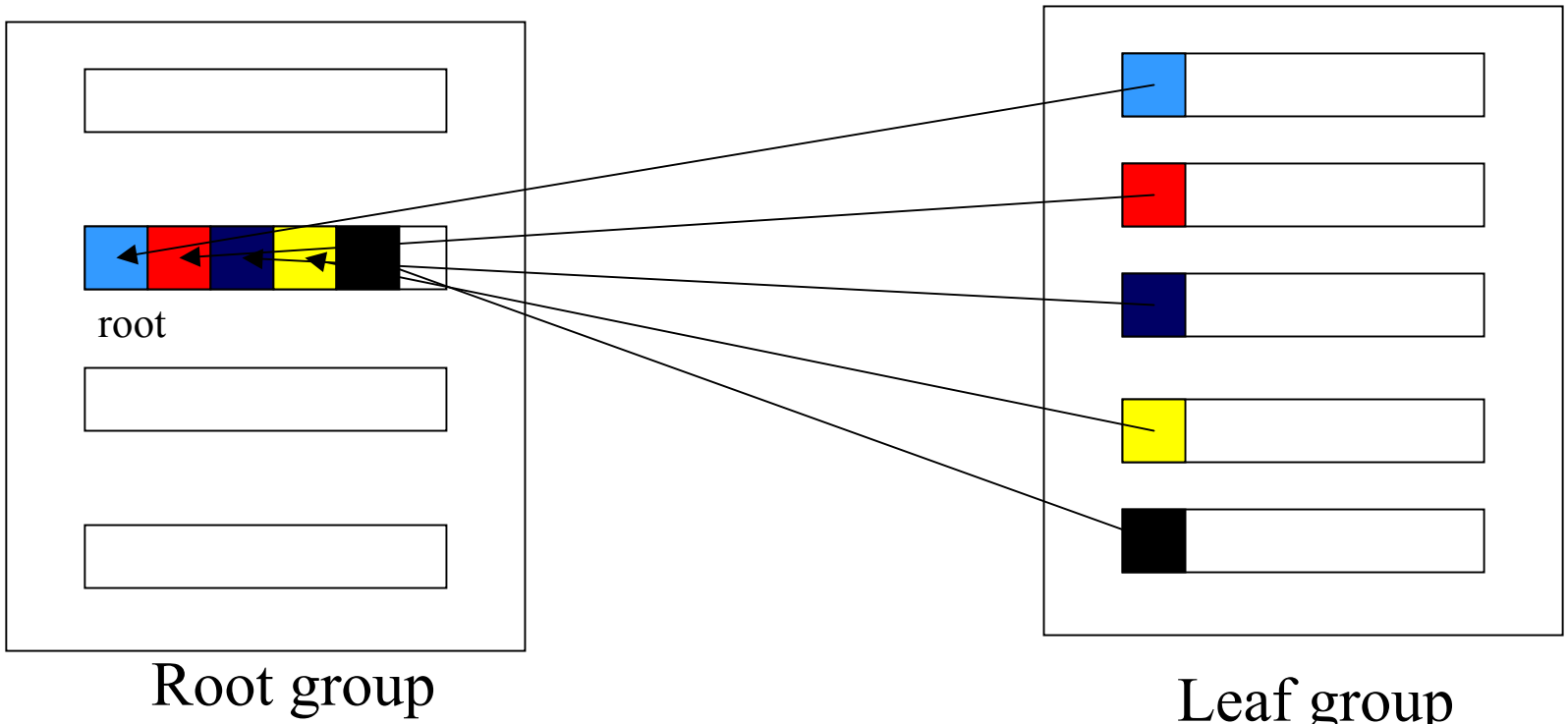
Root group: Root process passes MPI_ROOT for the *root* argument while others pass MPI_PROC_NULL.

Leaf group: All processes pass the same argument in *root*, which is the rank of the root process in the root group. *scount* and *stype* must be consistent with *rcount* and *rtype* on the root

Send arguments are only meaningful at processes in the leaf group

Receive arguments are only meaningful at the root

MPI_GATHER



MPI_Scatter (sbuf, scount, stype, rbuf,rcount,rtype,root, comm,ierr)

Data is scattered in rank order from the root to all the processes in the leaf group

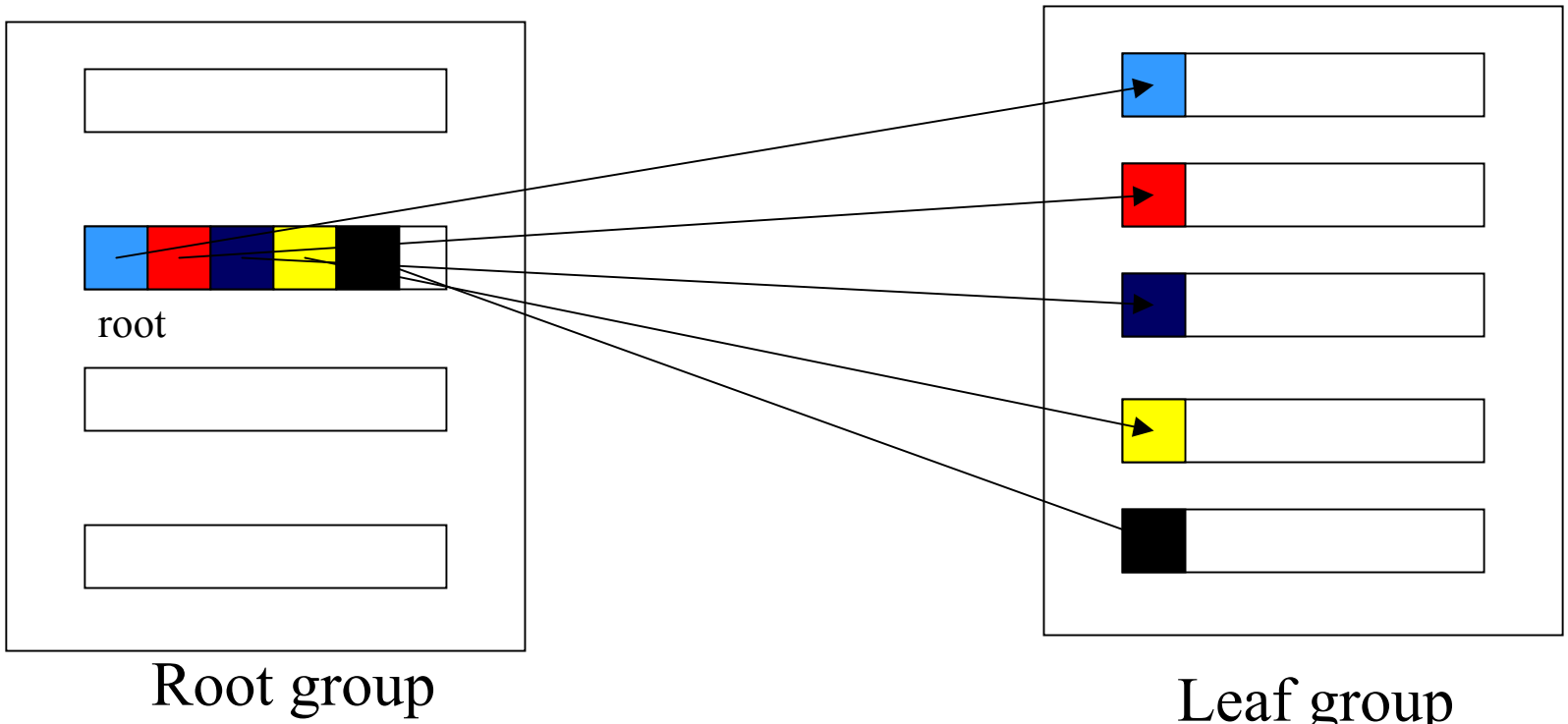
Root group: Root process passes MPI_ROOT for the *root* argument while others pass MPI_PROC_NULL.

Leaf group: All processes pass the same argument in *root*, which is the rank of the root process in the root group. *rcount* and *rtype* must be consistent with *scount* and *stype* on the root

Receive arguments are only meaningful at processes in the leaf group

Send arguments are only meaningful at the root

MPI_SCATTER



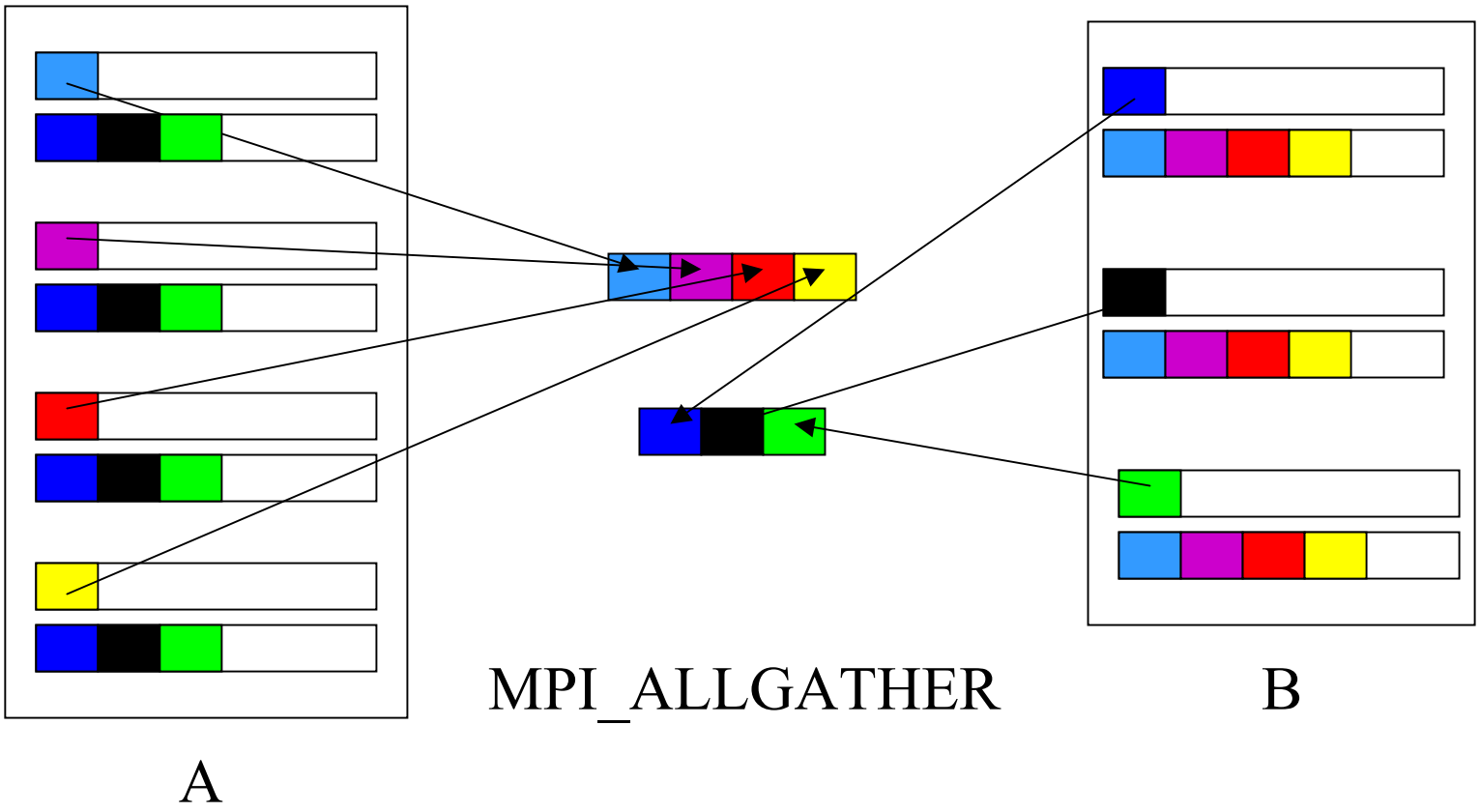
MPI_Allgather (sbuf,scount,stype, rbuf,rcount,rtype, comm,ierr)

All arguments are meaningful at every process

Data from *sbuf* at all processes in group A is concatenated in rank order and the result is stored at *rbuf* of every process in group B and vice-versa

Send arguments in A must be consistent with receive arguments in B, and vice-versa

MPI_ALLGATHER



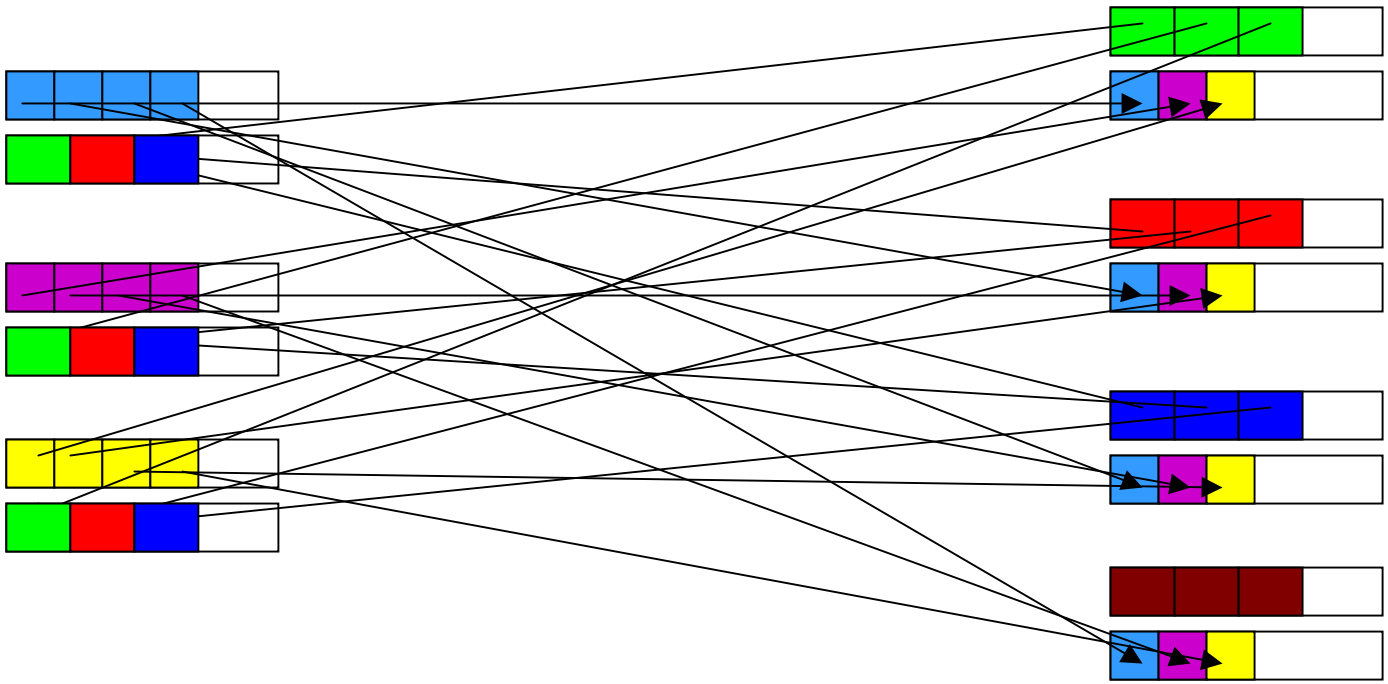
MPI_Alltoall (sbuff, scount, stype, rbuf, rcount, rtype, comm, ierr)

Result is as if each process in group A scatters its *sbuff* to each process in group B and each process in group B scatters its *sbuff* to each process in group A

Data is gathered in *rbuff* in rank order according to the rank in the group providing the data

Each process in group A sends the same amount of data to group B and vice-versa

MPI_ALLTOALL



MPI_ALLTOALL

MPI_Reduce (sbuf, rbuf, count, datatype, op, root, comm, ierr)

Root group: Root process passes MPI_ROOT for the *root* argument while others pass MPI_PROC_NULL

Leaf group: All processes pass the same argument in *root*, which is the rank of the root process in the root group

s*buf* is only meaningful at processes in the leaf group

rbuf is only meaningful at the root

The result is as if the leaf group did a regular reduce except the results are stored at root

count, datatype, and *op* should be meaningless at non-root processes in root group

MPI_Allreduce (sbuf, rbuf, count, datatype, op, comm, ierr)

The result is as if group A did a regular reduce except the results are stored at all the process in group B and vice versa

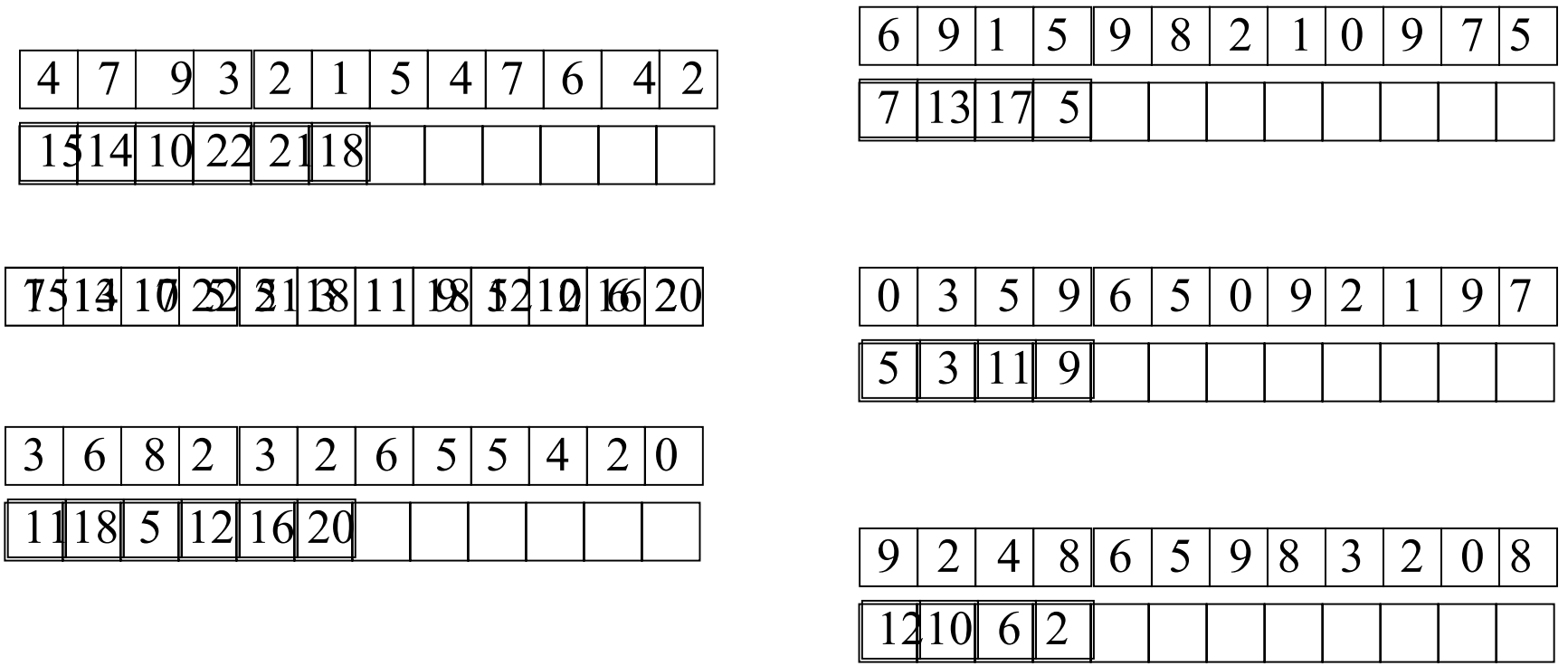
Count should be the same at all processes

MPI_Reduce_scatter (sbuf, rbuf, rcounts, datatype, op, comm, ierr)

The result is as if group A did a regular reduce with count equal to the sum of *rcounts* followed by a scatter to group B, and vice-versa

rcount should be the same at all processes in each group and the sum of all the *rcounts* in group A should equal the sum of all rcounts in group B

MPI_REDUCE_SCATTER



MPI_Scan and MPI_Exscan

There are no extended collective operations for these 2 routines

One Sided Communication

One sided communication allows shmem style gets and puts

Only one process need actively participate in one sided operations

With sufficient hardware support, remote memory operations can offer greater performance and functionality over the message passing model

MPI remote memory operations do not make use of a shared address space

One Sided Communication

By requiring only one process to participate, significant performance improvements are possible

- No implicit ordering of data delivery
- No implicit synchronization

Some programs are more easily written with the remote memory access (RMA) model

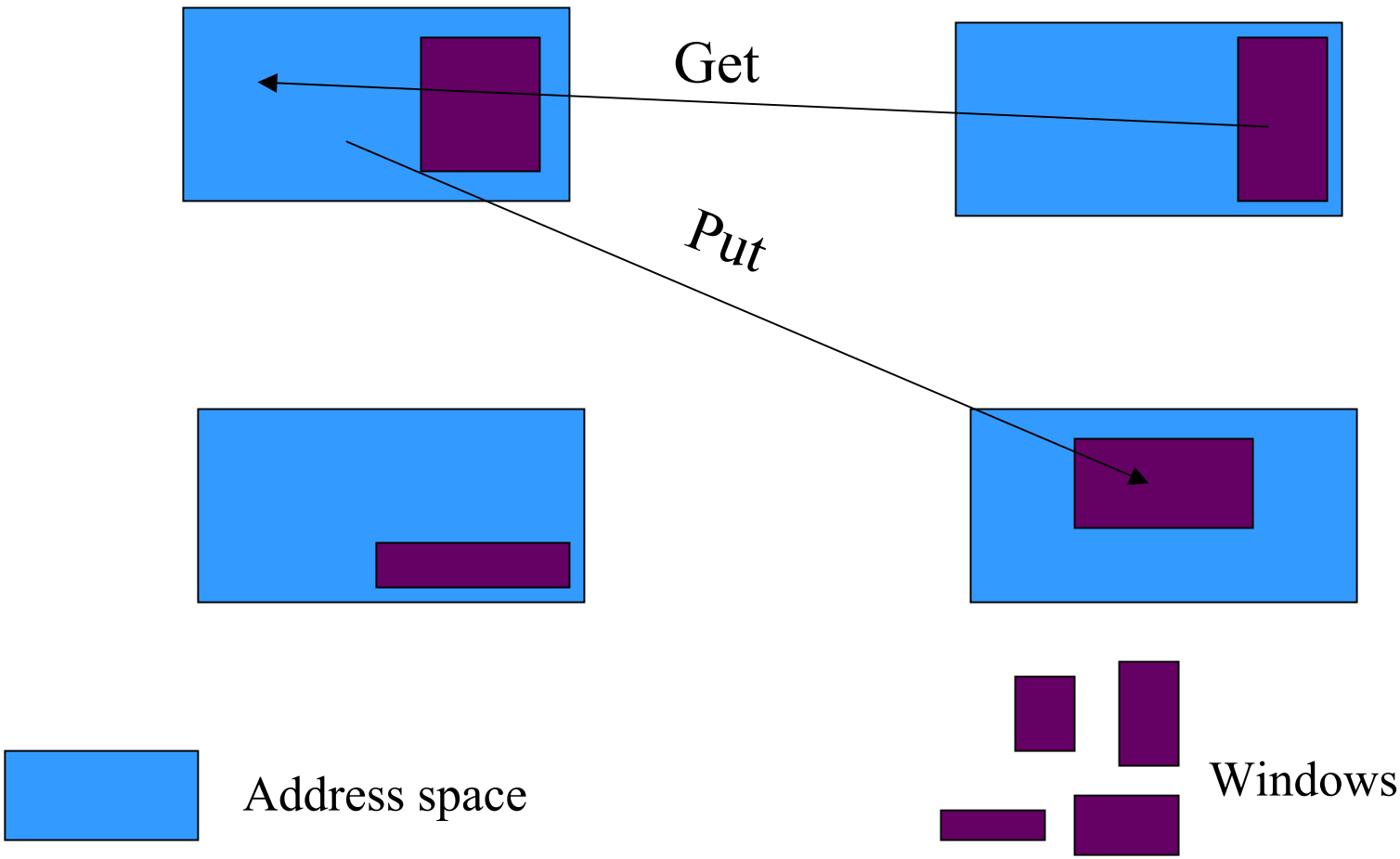
- Global counter

One Sided Communication

RMA operations require 3 steps

1. Define an area of memory that can be used for RMA operations (window)
2. Specify the data to be moved and where to move it
3. Specify a way to know the data is available

One Sided Communication



One Sided Communication

Memory Windows

- A memory window defines an area of memory that can be used for RMA operations
- A memory window must be a contiguous block of memory
- Described by a base address and number of bytes
- Window creation is collective across a communicator
- A window object is returned. This window object is used for all subsequent RMA calls

One Sided Communication

MPI_WIN_CREATE (void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win, ierr)

- base is the base address of the window
- size is the size in bytes of the window
- disp_unit is the displacement unit for data access (1 for bytes)
- info is used for performance tuning
- comm is the communicator over which the call is collective
- win is the window object returned

One Sided Communication

Data movement

- MPI_PUT
- MPI_GET
- MPI_ACCUMULATE

All data movement routines are non-blocking

Synchronization call is required to ensure operation completion

One Sided Communication

MPI_PUT (void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win window, ierr)

- origin_addr is the address in the calling process of the data to be transferred. It need not be within a memory window
- origin_count is the number of elements of type origin_datatype to be transferred
- target_rank is the rank within the window object of the destination process
- target_disp is the offset into the window on the destination process. This is in terms of disp_unit used in window creation on target process
- target_count and target_datatype are similar to count and datatype used in a receive
- window is the window object returned from creation

One Sided Communication

MPI_GET (void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win window, ierr)

- origin_addr is the address in the calling process where the data is to be transferred. It need not be within a memory window
- origin_count is the number of elements of type origin_datatype to be transferred into origin_addr
- target_rank is the rank within the window object of the destination process
- target_disp is the offset into the window on the destination process. This is in terms of disp_unit used in window creation on target process
- target_count and target_datatype are similar to count and datatype used in a send
- window is the window object returned from creation

One Sided Communication

MPI_ACCUMULATE (void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,MPI_Op op,MPI_Win window,ierr)

- All arguments besides op are the same as in get and put
- op is an MPI_Op as in MPI_Reduce
- op can only be a pre-defined operation
- Still a one-sided operation (not collective)
- Combines communication and computation
  - Like a put, but with a computation

One Sided Communication

Completing data transfers

- There are a number of different ways to complete data transfers
- The simplest is a barrier like mechanism (fence)
- This mechanism can also be used to ensure data is available
- The fence operation is collective across all process in the communicator used to create the windows
- Most suitable for data parallel applications
- A fence is used to separate local load/stores and RMA operations
- Multiple RMA operations may be completed with a single call to fence

One Sided Communication

MPI_WIN_FENCE (int assert, MPI_Win win, ierr)

- assert is an integer value used to provide information about the fence that may allow an MPI implementation to do performance optimization
- win is the window object return in the MPI_Win_create call

Point-to-Point Message Passing

```
CALL MPI_COMM_RANK (MPI_COMM_WORLD, rank, ierr)
IF (rank .eq. 0) then
  CALL MPI_ISEND (outbuff, n, MPI_INT, 1, 0,
                  MPI_COMM_WORLD, request, ierr)
ELSE
  CALL MPI_IRECV (inbuff, n, MPI_INT, 0, 0,
                  MPI_COMM_WORLD, request, ierr)
ENDIF
……..
Do other work
……..
CALL MPI_WAIT (request, status, ierr)
```

One Sided Communication

```
CALL MPI_COMM_RANK (MPI_COMM_WORLD, rank, ierr)
CALL MPI_TYPE_SIZE (MPI_INT, size, ierr)
IF (rank .eq. 0) then
  CALL MPI_WIN_CREATE (MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
     MPI_COMM_WORLD, win, ierr)
ELSE
  CALL MPI_WIN_CREATE (inbuf, n*size, size, MPI_INFO_NULL,
     MPI_COMM_WORLD, win, ierr)
ENDIF
CALL MPI_WIN_FENCE (0, win, ierr)
IF (rank .eq. 0) then
  CALL MPI_PUT (outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win, ierr)
ENDIF
……
Do other work
……
CALL MPI_FENCE (0, win, ierr)
CALL MPI_WIN_FREE (win, ierr)
```

One Sided Communication

```
MPI_Win_create (A, …., &win);
MPI_Win_fence (0, win);
If (rank == 0) {
  MPI_Put (….., win);
  MPI_Put (….., win);
  ……
  MPI_Put (….., win);
}
```

```
MPI_Win_fence (0, win);
MPI_Get (…. , win);
MPI_Win_fence (0, win);
A[rank] = 4;
MPI_Win_fence (0, win);
MPI_Put ( … , win);
MPI_Win_fence (0, win);
```

One Sided Communication

Passive target RMA

- Requires synchronization calls by only the process initiating data transfer
- MPI_Win_lock and MPI_Win_unlock define an *access epoch*
- Lock and unlock apply only to the remote memory window, not the entire window object
- A call to unlock ensures all RMA operations performed since the call to lock have completed
- Lock and unlock pairs are required around local access to memory windows as well
- Locks can be shared or exclusive
- Some implementations may require windows to be allocated by MPI_Alloc_mem

One Sided Communication

MPI_WIN_LOCK (int locktype, int rank, int assert, MPI_Win win,ierr)

MPI_WIN_UNLOCK (int rank, MPI_Win win, ierr)

- Locktype can be MPI_LOCK_SHARED or MPI_LOCK_EXCLUSIVE
- Rank is the rank of the process that owns the window to be accessed
- Assert is an integer value used for optimization
- Win is the window object of which the targeted window is part

One Sided Communication

```
If (rank == 0) {
  MPI_Win_lock (MPI_LOCK_SHARED, 1, 0, win);
  MPI_Put (outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win);
  MPI_Win_unlock (1, win);
}
```

One Sided Communication

- Not widely implemented
- MPICH and LAM only support active synchronization
  - Passive synchronization is in development
- May be useful for applications that lend themselves to the get/put programming model
- Evidence of some performance optimization on shared memory machines (and Cray!)
- I have seen no evidence that there is any performance advantage on distributed memory machines. (Other than Cray!)

Course Outline

Day 2

- Morning – Lecture
  - MPI-I/O
- Afternoon – Lab
  - Hands on exercises using MPI-I/O

MPI-I/O

Introduction

- What is parallel I/O
- Why do we need parallel I/O
- What is MPI-I/O

MPI-I/O

- Terms and definitions
- File manipulation
- Derived data types and file views

OUTLINE (cont)

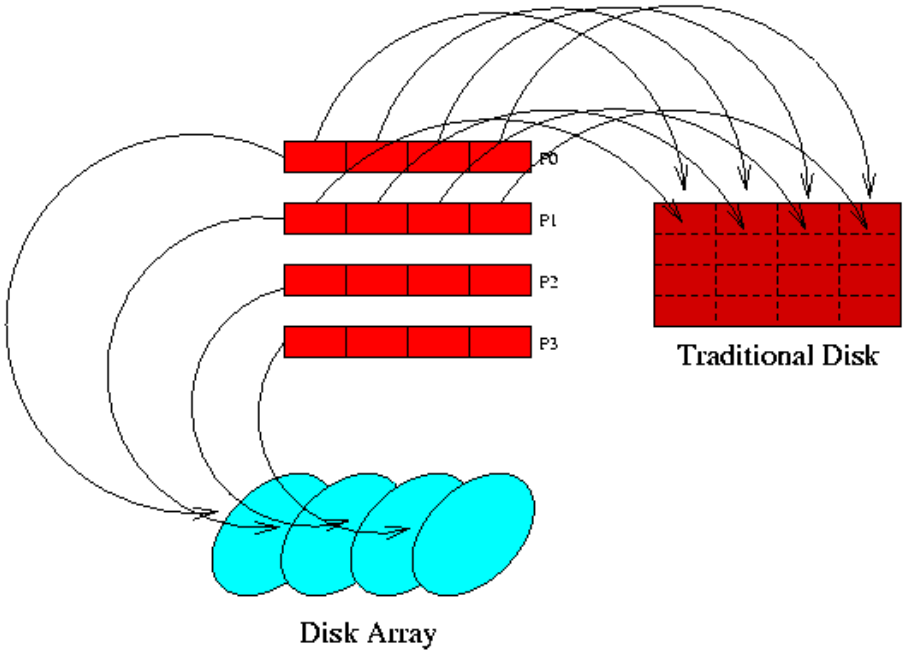
MPI-I/O (cont)

- Data access
  - Non-collective access
  - Collective access
  - Split collective access
- File interoperability
- Gotchas - Consistency and semantics

INTRODUCTION

What is parallel I/O?

› Multiple processes accessing a single file

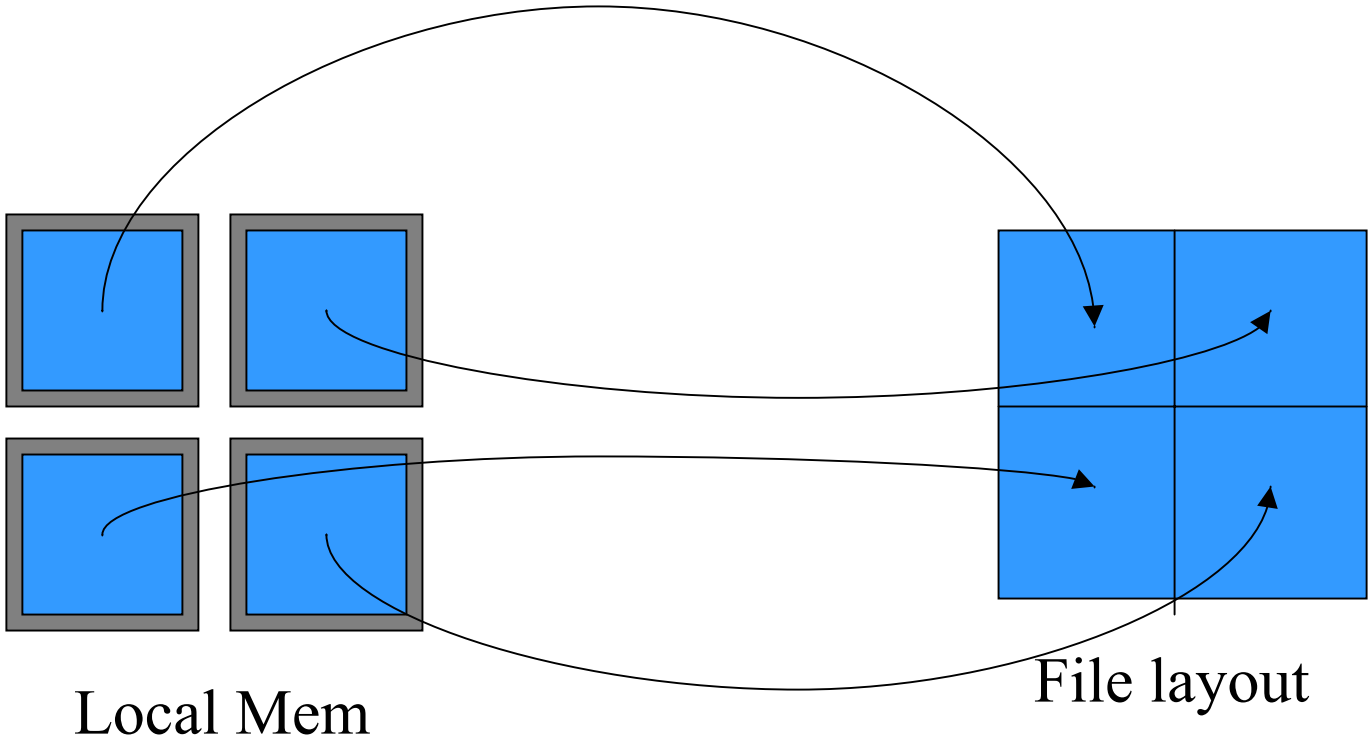


INTRODUCTION

What is parallel I/O?

- Multiple processes accessing a single file
- Often, both data and file access is non-contiguous
  - Ghost cells cause non-contiguous data access
  - Block or cyclic distributions cause non-contiguous file access

Non-Contiguous Access



INTRODUCTION

What is parallel I/O?

- Multiple processes accessing a single file
- Often, both data and file access is non-contiguous
  - Ghost cells cause non-contiguous data access
  - Block or cyclic distributions cause non-contiguous file access
- Want to access data and files with as few I/O calls as possible

INTRODUCTION (cont)

Why use parallel I/O?

- Many users do not have time to learn the complexities of I/O optimization
- Use of parallel I/O can simplify coding
  - Single read/write operation vs. multiple read/write operations
- Parallel I/O potentially offers significant performance improvement over traditional approaches

INTRODUCTION (cont)

Traditional approaches

- Each process writes to a separate file
  - Often requires an additional post-processing step
  - Without post-processing, restarts must use same number of processor
- Result sent to a master processor, which collects results and writes out to disk
- Each processor calculates position in file and writes individually

INTRODUCTION (cont)

What is MPI-I/O?

- MPI-I/O is a set of extensions to the original MPI standard
- This is an interface specification: It does NOT give implementation specifics
- It provides routines for file manipulation and data access
- Calls to MPI-I/O routines are portable across a large number of architectures

MPI-I/O

Terms and Definitions

- Displacement - Number of bytes from the beginning of a file
- etype - unit of data access within a file
- filetype - datatype used to express access patterns of a file
- file view - definition of access patterns of a file
  - Defines what parts of a file are visible to a process

MPI-I/O

Terms and Definitions

- Offset - Position in the file, relative to the current view, expressed in terms of number of etypes
- file pointers - offsets into the file maintained by MPI
  - Individual file pointer - local to the process that opened the file
  - Shared file pointer - shared (and manipulated) by the group of processes that opened the file

FILE MANIPULATION

MPI_FILE_OPEN(MPI_Comm comm, char *filename, int mode, MPI_Info info, MPI_File *fh, ierr)

- Opens the file identified by *filename* on each processor in communicator *Comm*
- Collective over this group of processors
- Each processor must use same value for *mode* and reference the same file
- *info* is used to give hints about access patterns

FILE MANIPULATION

MODES

- MPI_MODE_CREATE
  - Must be used if file does not exist
- MPI_MODE_RDONLY
- MPI_MODE_RDWR
- MPI_MODE_WRONLY
- MPI_MODE_EXCL
  - Error if creating file that already exists
- MPI_MODE_DELETE_ON_CLOSE
- MPI_MODE_UNIQUE_OPEN
- MPI_MODE_SEQUENTIAL
- MPI_MODE_APPEND

Hints

Hints can be passed to the I/O implementation via the info argument

MPI_Info info

MPI_Info_create (&info)

MPI_Info_set (info, key, value)

- key is a string specifying the hint to be applied
- value is a string specifying the value key is to be set to

There are 15 pre-defined keys

The implementation may or may not make use of hints

Hints

striping_factor

The number of I/O devices to be used

striping_unit

The number of bytes per block

collective_buffering

true or false: whether collective buffering should be performed

cb_block_size

Block size to be used for buffering (nodes access data in chunks this size

cb_buffer_size

The total buffer size that should be used for buffering (often block size times # nodes)

FILE MANIPULATION (cont)

MPI_FILE_CLOSE (MPI_File *fh)

- This routine synchronizes the file state and then closes the file
- The user must ensure all I/O routines have completed before closing the file
- This is a collective routine (but not synchronizing)

DERIVED DATATYPES & VIEWS

Derived datatypes are not part of MPI-I/O

They are used extensively in conjunction with MPI-I/O

A filetype is really a datatype expressing the access pattern of a file

Filetypes are used to set file views

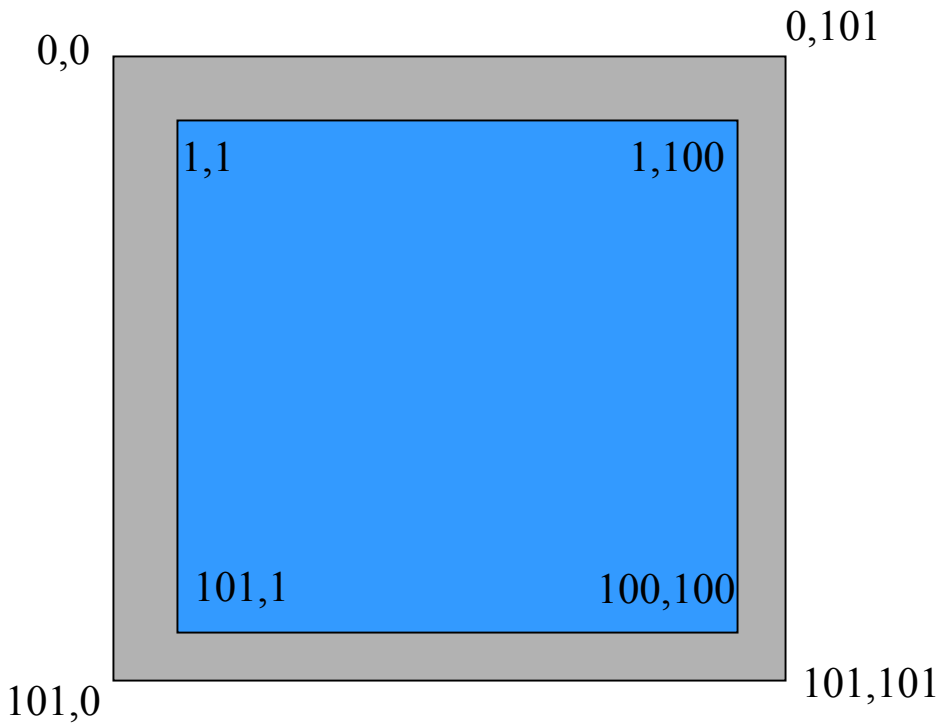
DERIVED DATATYPES & VIEWS

Non-contiguous memory access

MPI_TYPE_CREATE_SUBARRAY

- NDIMS - number of dimensions
- ARRAY_OF_SIZES - number of elements in each dimension of full array
- ARRAY_OF_SUBSIZES - number of elements in each dimension of sub-array
- ARRAY_OF_STARTS - starting position in full array of sub-array in each dimension
- ORDER - MPI_ORDER_(C or FORTRAN)
- OLDTYPE - datatype stored in full array
- NEWTYPE - handle to new datatype

NONCONTIGUOUS MEMORY ACCESS

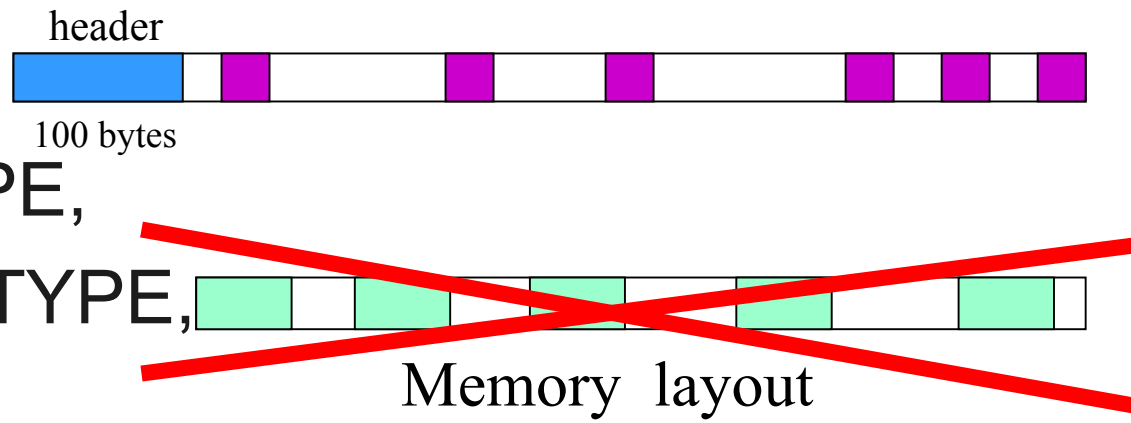


NONCONTIGUOUS MEMORY ACCESS

```
INTEGER sizes(2), subsizes(2), starts(2), dtype, ierr
sizes(1) = 102
sizes(2) = 102
subsizes(1) = 100
subsizes(2) = 100
starts(1) = 1
starts(2)= 1
CALL MPI_TYPE_CREATE_SUBARRAY(2,sizes,subsizes,starts,
    MPI_ORDER_FORTRAN,MPI_REAL8,dtype,ierr)
```

NONCONTIGUOUS FILE ACCESS

```
MPI_FILE_SET_VIEW(
    MPI_File FH,
    MPI_Offset DISP,
    MPI_Datatype ETYPE,
    MPI_Datatype FILETYPE,
    char *DATAREP,
    MPI_Info INFO,
    IERROR)
```



NONCONTIGUOUS FILE ACCESS

The file has ‘holes’ in it from the processor’s perspective

- MPI_TYPE_CONTIGUOUS(NUM,OLD,NEW,IERR)
  - NUM - Number of contiguous elements
  - OLD - Old data type
  - NEW - New data type
- MPI_TYPE_CREATE_RESIZED(OLD,LB,EXTENT, NEW, IERR)
  - OLD - Old data type
  - LB - Lower Bound
  - EXTENT - New size
  - NEW - New data type

‘Holes’ in the file

Memory layout

File layout (2 ints followed by 3 ints)

CALL MPI_TYPE_CONTIGUOUS(2, MPI_INT, CTYPE, IERR)

DISP = 4

LB = 0

EXTENT=5*4

CALL MPI_TYPE_CREATE_RESIZED(CTYPE,LB,EXTENT,FTYPE,IERR)

CALL MPI_TYPE_COMMIT(FTYPE, IERR)

CALL MPI_FILE_SET_VIEW(FH,DISP,MPI_INT,FTYPE,’native’,MPI_INFO_NULL, IERR)

NONCONTIGUOUS FILE ACCESS

The file has ‘holes’ in it from the processor’s perspective

A block-cyclic data distribution

NONCONTIGUOUS FILE ACCESS

The file has ‘holes’ in it from the processor’s perspective

A block-cyclic data distribution

MPI_TYPE_VECTOR(

COUNT - Number of blocks

BLOCKLENGTH - Number of elements per block

STRIDE - Elements between start of each block

OLDTYPE - Old datatype

NEWTYPE - New datatype)

Block-cyclic distribution

File layout (blocks of 4 ints)

CALL MPI_TYPE_VECTOR(3, 4, 16, MPI_INT, FILETYPE, IERR)

CALL MPI_TYPE_COMMIT (FILETYPE, IERR)

DISP = 4 * 4 * MYRANK

CALL MPI_FILE_SET_VIEW (FH, DISP, MPI_INT, FILETYPE, 'native', MPI_INFO_NULL, IERR)

NONCONTIGUOUS FILE ACCESS

The file has ‘holes’ in it from the processor’s perspective

A block-cyclic data distribution

multi-dimensional array access

NONCONTIGUOUS FILE ACCESS

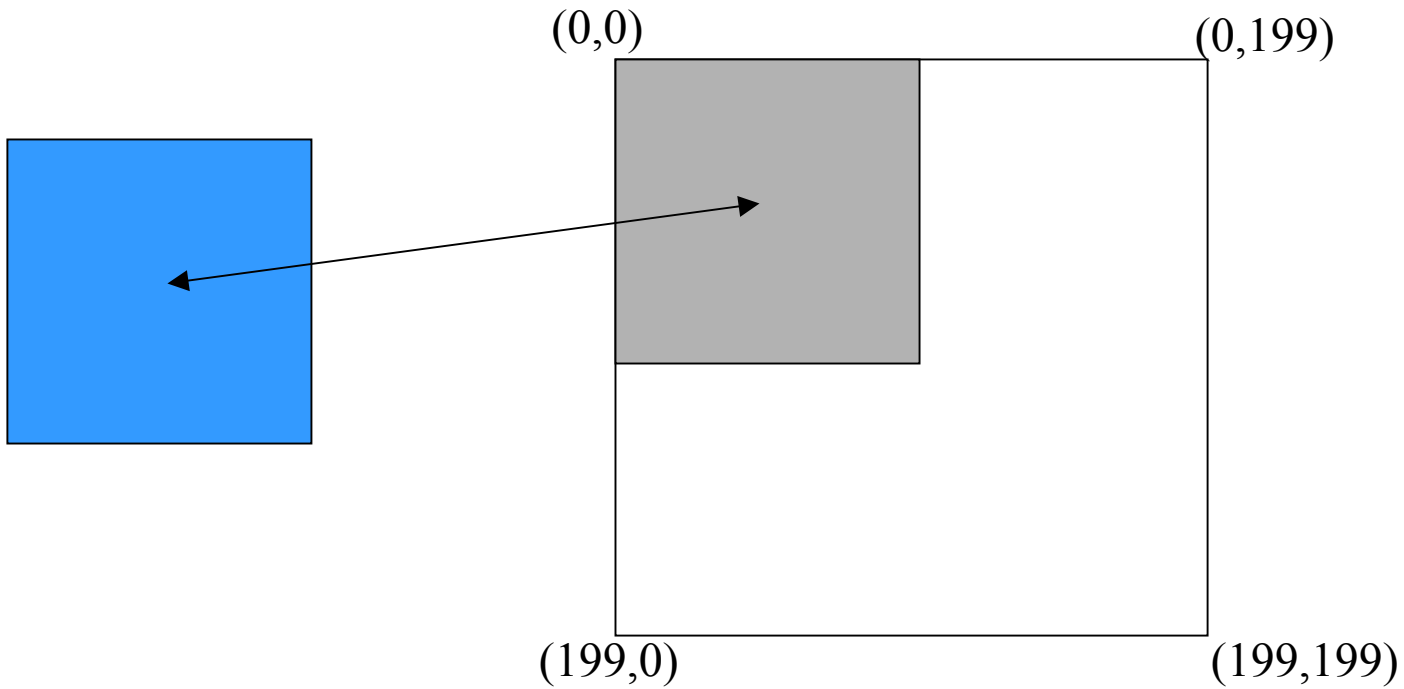
The file has ‘holes’ in it from the processor’s perspective

A block-cyclic data distribution

multi-dimensional array access

MPI_TYPE_CREATE_SUBARRAY()

Distributed array access



Distributed array access

Sizes(1) = 200

sizes(2) = 200

subsizes(1) = 100

subsizes(2) = 100

starts(1) = 0

starts(2) = 0

CALL MPI_TYPE_CREATE_SUBARRAY(2, SIZES, SUBSIZES, STARTS, MPI_ORDER_FORTRAN, MPI_INT, FILETYPE, IERR)

CALL MPI_TYPE_COMMIT(FILETYPE, IERR)

CALL MPI_FILE_SET_VIEW(FH, 0, MPI_INT, FILETYPE, 'NATIVE', MPI_INFO_NULL, IERR)

NONCONTIGUOUS FILE ACCESS

The file has ‘holes’ in it from the processor’s perspective

A block-cyclic data distribution

multi-dimensional array distributed with a block distribution

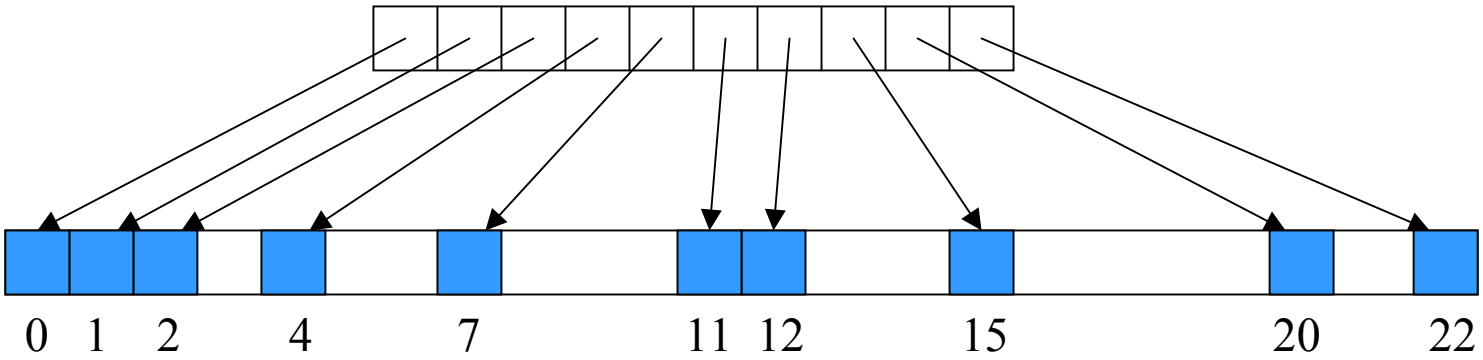
Irregularly distributed arrays

Irregularly distributed arrays

MPI_TYPE_CREATE_INDEXED_BLOCK

- COUNT - Number of blocks
- LENGTH - Elements per block
- MAP - Array of displacements
- OLD - Old datatype
- NEW - New datatype

Irregularly distributed arrays



0	1	2	4	7	11	12	15	20	22

MAP_ARRAY

Irregularly distributed arrays

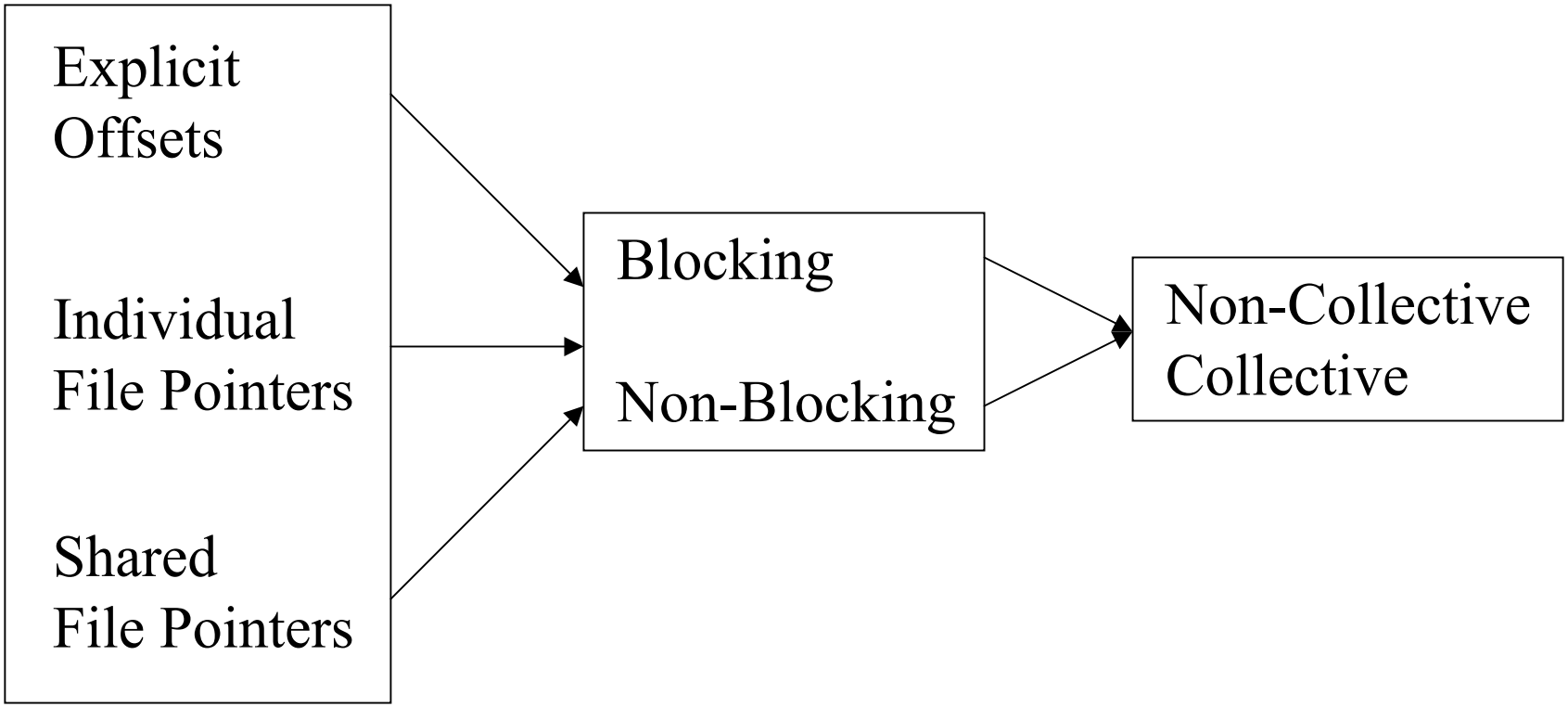
CALL MPI_TYPE_CREATE_INDEXED_BLOCK (10, 1, FILE_MAP, MPI_INT, FILETYPE, IERR)

CALL MPI_TYPE_COMMIT (FILETYPE, IERR)

DISP = 0

CALL MPI_FILE_SET_VIEW (FH, DISP, MPI_INT, FILETYPE, ‘native’, MPI_INFO_NULL, IERR)

DATA ACCESS


Explicit Offsets
Individual File Pointers
Shared File Pointers
Blocking
Non-Blocking
Non-Collective
Collective

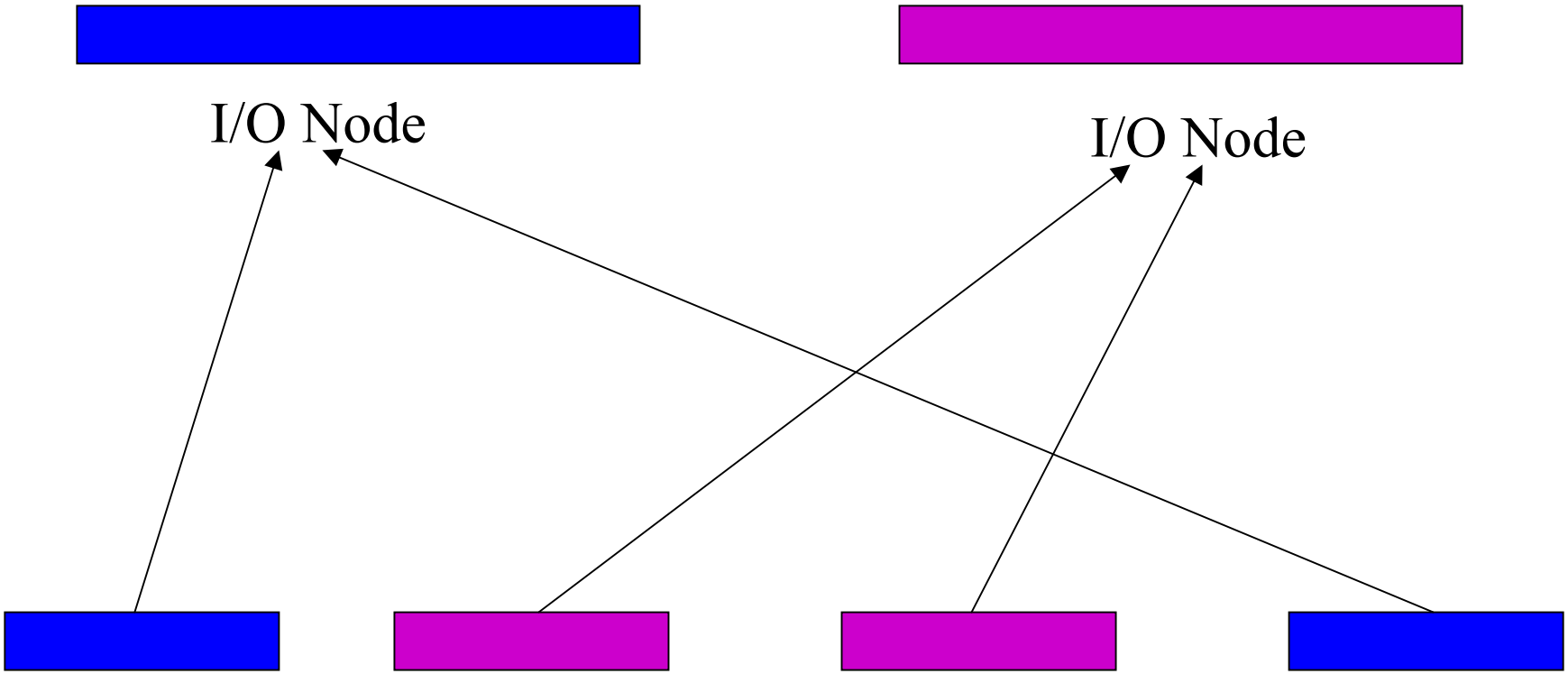
COLLECTIVE I/O

Memory layout on 4 processor

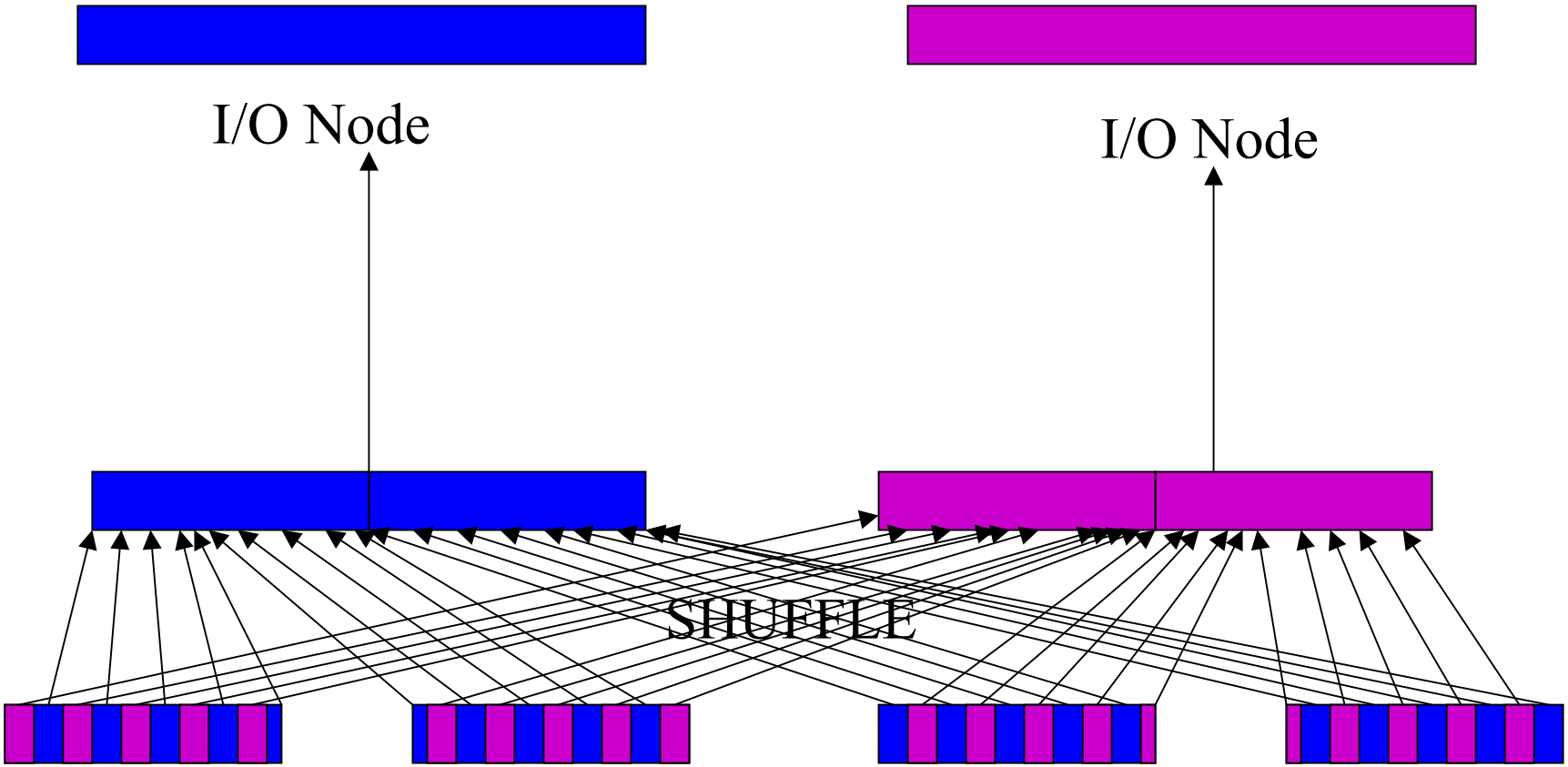
MPI temporary memory buffer

File layout

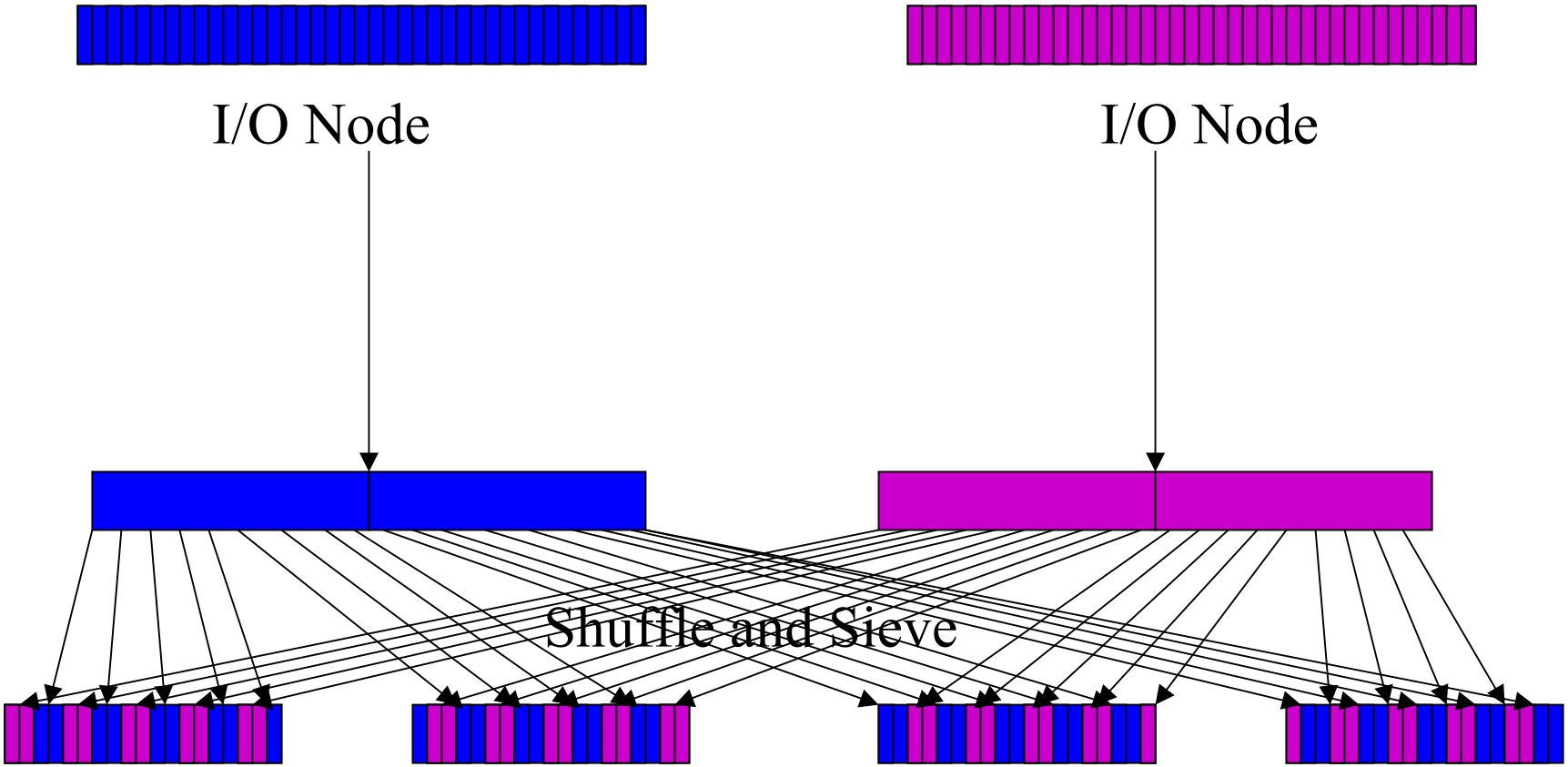
Two-Phase I/O



Two-Phase I/O



Two-Phase I/O with Data Sieving



Collective I/O

Server-based Collective I/O

- Similar to client based, but the I/O nodes collect data in block sizes for file access
- No system buffer space needed on compute nodes

Disk-Directed I/O (DDIO)

Uses server-based collective I/O, but reads data from disk in a manner than minimizes disk head movement. The data is transferred between I/O nodes and compute nodes as they are read/written

DATA ACCESS ROUTINES

EXPLICIT OFFSETS

Parameters

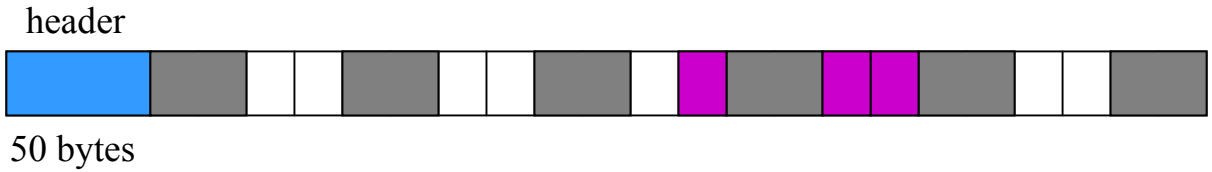
- MPI_File FH - File handle
- MPI_Offset OFFSET - Location in file to start
- void *BUF - Buffer to write from/read to
- int COUNT - Number of elements
- MPI_Datatype DATATYPE - Type of each element
- MPI_Status STATUS - Return status (blocking)
- MPI_Request REQUEST - Request handle (non-blocking,non-collective)

EXPLICIT OFFSETS (cont)

I/O Routines

- MPI_FILE_(READ/WRITE)_AT ()
- MPI_FILE_(READ/WRITE)_AT_ALL ()
- MPI_FILE_I(READ/WRITE)_AT ()
- MPI_FILE_(READ/WRITE)_AT_ALL_BEGIN ()
- MPI_FILE_(READ/WRITE)_AT_ALL_END (FH, BUF, STATUS)

EXPLICIT OFFSETS




```
int buff[3];

count = 5;
blocklen = 2;
stride = 4

MPI_Type_vector (count, blocklen,
     stride, MPI_INT, &ftype);
MPI_Type_commit (ftype);
```

```
disp = 58;
MPI_File_open (MPI_COMM_WORLD, filename,
   MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view (fh, disp, MPI_INT, ftype, "native",
   MPI_INFO_NULL);
MPI_File_write_at (fh, 5, buff, 3, MPI_INT, &status);
MPI_File_close (&fh);
```

IDIVIDUAL FILE POINTERS

Parameters

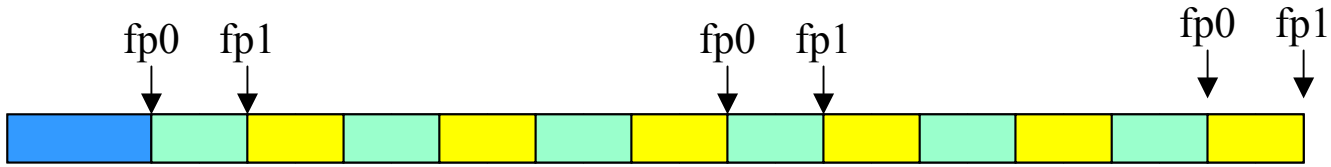
- MPI_File FH - File handle
- void *BUF - Buffer to write to/read from
- int COUNT - number of elements to be read/written
- MPI_Datatype DATATYPE - Type of each element
- MPI_Status STATUS - Return status (blocking)
- MPI_Request REQUEST - Request handle (non-blocking, non-collective)

INDIVIDUAL FILE POINTERS

I/O Routines

- MPI_FILE_(READ/WRITE) ()
- MPI_FILE_(READ/WRITE)_ALL ()
- MPI_FILE_I(READ/WRITE) ()
- MPI_FILE_(READ/WRITE)_ALL_BEGIN()
- MPI_FILE_(READ/WRITE)_ALL_END (FH, BUF, STATUS)

INDIVIDUAL FILE POINTERS




```
int buff[12];

count = 6;
blocklen = 2;
stride = 4

MPI_Type_vector (count, blocklen,
     stride, MPI_INT, &ftype);
MPI_Type_commit (ftype);
```

```
disp = 50 + myrank*8;
MPI_File_open (MPI_COMM_WORLD, filename,
   MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view (fh, disp, MPI_INT, ftype, "native",
   MPI_INFO_NULL);
MPI_File_write(fh, buff, 6, MPI_INT, &status);
MPI_File_write(fh, buff, 6, MPI_INT, &status);
MPI_File_close (&fh);
```

INDIVIDUAL FILE POINTERS

```
int buffA[10];
Int buffB[10];

count = 5;
blocklen = 2;
stride = 4

MPI_Type_vector (count, blocklen,
     stride, MPI_INT, &ftype);
MPI_Type_commit (ftype);
```

```
disp = myrank*8;
MPI_File_open (MPI_COMM_WORLD, filename,
   MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view (fh, disp, MPI_INT, ftype, "native",
   MPI_INFO_NULL);
MPI_File_write(fh, buffA, 10, MPI_INT, &status);
MPI_File_write(fh, buffB, 10, MPI_INT, &status);
MPI_File_close (&fh);
```

INDIVIDUAL FILE POINTERS

```
int buffA[10];
Int buffB[10];

count = 5;
blocklen = 2;
stride = 4

MPI_Type_vector (count, blocklen,
      stride, MPI_INT, &ftype);
MPI_Type_commit (ftype);
```

```
disp = myrank*8;
MPI_File_open (MPI_COMM_WORLD, filename,
   MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view (fh, disp, MPI_INT, ftype, “native”,
   MPI_INFO_NULL);
MPI_File_write(fh, buffA, 10, MPI_INT, &status);
disp = disp + 4*20;
MPI_File_set_view (fh, disp, MPI_INT, ftype, “native”,
   MPI_INFO_NULL);
MPI_File_write(fh, buffB, 10, MPI_INT, &status);
MPI_File_close (&fh);
```

INDIVIDUAL FILE POINTERS

```
int buffA[10];
Int buffB[10];

count = 5;
blocklen = 2;
stride = 4

MPI_Type_vector (count, blocklen,
      stride, MPI_INT, &atype);
extent = count*blocklen*nprocs*4;
MPI_Type_create_resized (atype, 0,
  extent, &ftype);
MPI_Type_commit (ftype);
```

```
disp = myrank*8;
MPI_File_open (MPI_COMM_WORLD, filename,
    MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view (fh, disp, MPI_INT, ftype, "native",
    MPI_INFO_NULL);
MPI_File_write(fh, buffA, 10, MPI_INT, &status);
MPI_File_write(fh, buffB, 10, MPI_INT, &status);
MPI_File_close (&fh);
```

SHARED FILE POINTERS

All processes must have the same view

Parameters

- MPI_File FH - File handle
- void *BUF - Buffer
- int COUNT - Number of elements
- MPI_Datatype DATATYPE - Type of the elements
- MPI_Status STATUS - Return status (blocking)
- MPI_Requests REQUEST - Request handle (Non-blocking, non-collective

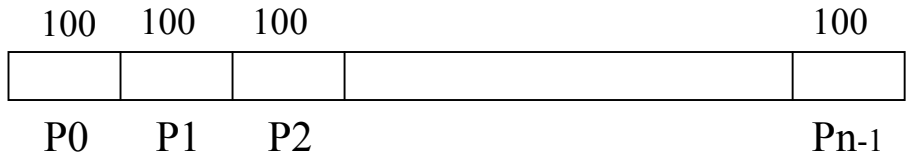
SHARED FILE POINTERS

I/O Routines

- MPI_FILE_(READ/WRITE)_SHARED ()
- MPI_FILE_I(READ/WRITE)_SHARED ()
- MPI_FILE_(READ/WRITE)_ORDERED ()
- MPI_FILE_(READ/WRITE)_ORDERED_BEGIN ()
- MPI_FILE_(READ/WRITE)_ORDERED_END (FH, BUF, STATUS)

SHARED FILE POINTERS

```
comm = MPI_COMM_WORLD;
MPI_Comm_rank (comm, &rank);
amode = MPI_MODE_CREATE |
    MPI_MODE_WRONLY;
…..
MPI_File_open (comm, logfile, amode,
    MPI_INFO_NULL, &fh);
…..
do some computing
if (some event occurred) {
  sprintf(buf, “Process %d: %s\n”, rank, event);
  size = strlen(buf);
  MPI_File_write_shared (fh, buf, size
    MPI_CHAR, &status);
}
MPI_File_close (&fh);
…..
```




```
int buff[100];

MPI_File_open (comm, logfile, amode,
    MPI_INFO_NULL, &fh);
MPI_File_write_ordered (fh, buf, 100,
    MPI_INT, &status);
MPI_File_close (&fh);
```

FILE INTEROPERABILITY

- MPI puts no constraints on how an implementation should store files
- If a file is not stored as a linear byte stream, there must be a utility for converting the file into a linear byte stream
- Data representation aids interoperability

FILE INTEROPERABILITY (cont)

Data Representation

- Native - Data stored exactly as it is in memory.
- Internal - Data may be converted, but may be readable by the same MPI implementation, even on different architectures
- external32 - This representation is defined by MPI. Files written in external32 format can be read by any MPI on any machine

FILE INTEROPERABILITY (cont)

Some MPI-I/O implementations (Romio), created files are no different than those created by the underlying file system.

This means normal Posix commands (cp, rm, etc) work with files created by these implementations

Non-MPI programs can read these files

GOTCHAS - Consistency & Semantics

Collective routines are NOT synchronizing

Output data may be buffered

- Just because a process has completed a write does not mean the data is available to other processes

Three ways to ensure file consistency:

- MPI_FILE_SET_ATOMICITY ()
- MPI_FILE_SYNC ()
- MPI_FILE_CLOSE ()

CONSISTENCY & SEMANTICS

MPI_FILE_SET_ATOMICITY (MPI_File fh, int flag, ierr)

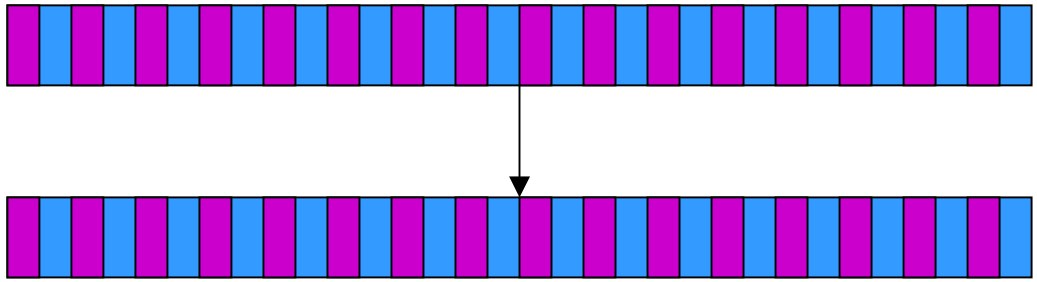
- Causes all writes to be immediately written to disk. This is a collective operation

MPI_FILE_SYNC (MPI_File fh, ierr)

- Collective operation which forces buffered data to be written to disk

MPI_FILE_CLOSE (MPI_File *fh)

- Writes any buffered data to disk before closing the file



Cached

Process 0

Process 1

Read magenta data

Write aqua data

Close file

Read aqua data

GOTCHA!!!

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_SET_ATOMICITY (FH)

CALL MPI_FILE_WRITE_AT (FH, 0, …)

CALL MPI_FILE_READ_AT (FH, 100, …)

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_SET_ATOMICITY (FH)

CALL MPI_FILE_WRITE_AT (FH, 100, …)

CALL MPI_FILE_READ_AT (FH, 0, …)

GOTCHA!!!

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_SET_ATOMICITY (FH)

CALL MPI_FILE_WRITE_AT (FH, 0, …)

CALL MPI_BARRIER ()

CALL MPI_FILE_READ_AT (FH, 100, …)

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_SET_ATOMICITY (FH)

CALL MPI_FILE_WRITE_AT (FH, 100, …)

CALL MPI_BARRIER ()

CALL MPI_FILE_READ_AT (FH, 0, …)

GOTCHA!!!

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_WRITE_AT (FH, 0, …)

CALL MPI_FILE_CLOSE (FH)

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_READ_AT (FH, 100, …)

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_WRITE_AT (FH, 100, …)

CALL MPI_FILE_CLOSE (FH)

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_READ_AT (FH, 0, …)

GOTCHA!!!

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_WRITE_AT (FH, 0, …)

CALL MPI_FILE_CLOSE (FH)

CALL MPI_BARRIER ()

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_READ_AT (FH, 100, …)

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_WRITE_AT (FH, 100, …)

CALL MPI_FILE_CLOSE (FH)

CALL MPI_BARRIER ()

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_READ_AT (FH, 0, …)

GOTCHA!!!

```
CALL MPI_FILE_OPEN (…, FH)
CALL MPI_FILE_WRITE_AT (FH, 0, …)
CALL MPI_FILE_SYNCH (FH)
CALL MPI_BARRIER ()
CALL MPI_FILE_READ_AT (FH, 100, …)
```

```
CALL MPI_FILE_OPEN (…, FH)
CALL MPI_FILE_WRITE_AT (FH, 100, …)
CALL MPI_FILE_SYNCH (FH)
CALL MPI_BARRIER ()
CALL MPI_FILE_READ_AT (FH, 0, …)
```

GOTCHA!!!

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_WRITE_AT (FH, 0, …)

CALL MPI_FILE_SYNCH (FH)

CALL MPI_BARRIER ()

CALL MPI_FILE_SYNCH (FH)

CALL MPI_FILE_READ_AT (FH, 100, …)

CALL MPI_FILE_OPEN (…, FH)

CALL MPI_FILE_WRITE_AT (FH, 100, …)

CALL MPI_FILE_SYNCH (FH)

CALL MPI_BARRIER ()

CALL MPI_FILE_SYNCH (FH)

CALL MPI_FILE_READ_AT (FH, 0, …)

CONCLUSIONS

- MPI-I/O potentially offers significant improvement in I/O performance
- This improvement can be attained with minimal effort on part of the user
  - Simpler programming with fewer calls to I/O routines
  - Easier program maintenance due to simple API

Recommended references

MPI - The Complete Reference Volume 1, The MPI Core

MPI - The Complete Reference Volume 2, The MPI Extensions

USING MPI: Portable Parallel Programming with the Message-Passing Interface

Using MPI-2: Advanced Features of the Message-Passing Interface

Recommended references

http://pdb.cs.utk.edu

Click “View Database”

Go to “Documents”

- MPI_CHECK
- Guidelines for writing portable MPI programs

http://www.cs.utk.edu/~cronk/Using_MPI_IO.pdf

http://www.cs.utk.edu/~cronk/Using_MPI_IO.doc

Course Outline

- Day 3
  - Morning – Lecture
    - Performance Analysis of MPI programs
      - TAU
      - Vampir/VampirTrace
  - Afternoon – Lab
    - Hands on exercises using Vampir and VampirTrace

Performance Analysis

It is typically much more difficult to debug and tune parallel programs

Programmers often have no idea where to begin searching for possible bottlenecks

A tool that allows the programmer to get a quick overview of the program's execution can aid the programmer in beginning this search

Basic Tuning Process

Select “best” compiler flags

Select/interface with “best” libraries

Measure

Validate

Hand-tune (routine/loop-level tuning)

… iterate

Observation: The best way to improve parallel performance is often still to simply improve sequential performance!

Performance Analysis in Practice

Observation: many application developers don't use performance tools at all (or rarely)

Why?

- Learning curve can be steep
- Results can be difficult to understand
- Investment (time) can be substantial
- Maturity/availability of various tools
- Not everyone is a computer scientist

Profiling

Recording of summary information during execution
- inclusive, exclusive time, # calls, hardware statistics, …

Reflects performance behavior of program entities
- functions, loops, basic blocks
- user-defined “semantic” entities

Very good for low-cost performance assessment

Helps to expose performance bottlenecks and hotspots

Implemented through
- sampling: periodic OS interrupts or hardware counter traps
- instrumentation: direct insertion of measurement code

No temporal context

Tracing

Recording of information about significant points (events) during program execution

- entering/exiting code region (function, loop, block, …)
- thread/process interactions (e.g., send/receive message)

Save information in event record

- timestamp
- CPU identifier, thread identifier
- Event type and event-specific information

Event trace is a time-sequenced stream of event records

Can be used to reconstruct dynamic program behavior

TAU Performance System

Tuning and Analysis Utilities (11+ year project effort)

Performance system framework for scalable parallel and distributed high-performance computing

Targets a general complex system computation model

- nodes / contexts / threads
- Multi-level: system / software / parallelism
- Measurement and analysis abstraction

Integrated toolkit for performance instrumentation, measurement, analysis, and visualization

- Portable performance profiling and tracing facility
- Open software approach with technology integration

University of Oregon , Forschungszentrum Jülich, LANL

TAU Instrumentation Approach

- Support for standard program events
  - Routines
  - Classes and templates
  - Statement-level blocks
- Support for user-defined events
  - Begin/End events (“user-defined timers”)
  - Atomic events (e.g., size of memory allocated/freed)
  - Selection of event statistics
- Support definition of “semantic” entities for mapping
- Support for event groups
- Instrumentation optimization

TAU Instrumentation

- Flexible instrumentation mechanisms at multiple levels
  - Source code
    - manual
    - automatic
      - C, C++, F77/90/95 (Program Database Toolkit (*PDT*))
      - OpenMP (directive rewriting (*Opari*), *POMP spec*)
  - Object code
    - pre-instrumented libraries (e.g., MPI using *PMPI*)
    - statically-linked and dynamically-linked
  - Executable code
    - dynamic instrumentation (pre-execution) (*DynInstAPI*)
    - virtual machine instrumentation (e.g., Java using *JVMPI*)

Multi-Level Instrumentation

- Targets common measurement interface
  - *TAU API*
- Multiple instrumentation interfaces
  - Simultaneously active
- Information sharing between interfaces
  - Utilizes instrumentation knowledge between levels
- Selective instrumentation
  - Available at each level
  - Cross-level selection
- Targets a common performance model
- Presents a unified view of execution
  - Consistent performance events

TAU Performance Measurement

- TAU supports profiling and tracing measurement
- Robust timing and hardware performance support using PAPI
- Support for online performance monitoring
  - Profile and trace performance data export to file system
  - Selective exporting
- Extension of TAU measurement for multiple counters
  - Creation of user-defined TAU counters
  - Access to system-level metrics
- Support for callpath measurement
- Integration with system-level performance data
  - Linux MAGNET/MUSE (Wu Feng, LANL)

TAU Measurement Options

Parallel profiling

- Function-level, block-level, statement-level
- Supports user-defined events
- TAU parallel profile data stored during execution
- Hardware counts values
- Support for multiple counters
- Support for callgraph and callpath profiling

Tracing

- All profile-level events
- Inter-process communication events
- Trace merging and format conversion

TAU Analysis

Parallel profile analysis

Pprof

parallel profiler with text-based display

ParaProf

Graphical, scalable, parallel profile analysis and display

Trace analysis and visualization

Trace merging and clock adjustment (if necessary)

Trace format conversion (ALOG, SDDF, VTF, Paraver)

Trace visualization using *Vampir* (Pallas/Intel)

Pprof Output (NAS Parallel Benchmark – LU)

- Intel Quad PIII Xeon
- F90 + MPICH
- Profile
  - Node
  - Context
  - Thread
- Events
  - code
  - MPI

emacs@neutron.cs.uoregon.edu

Buffers Files Tools Edit Search Mule Help

Reading Profile files in profile.*

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	1	3:11.293	1	15	191293269	applu
99.6	3,667	3:10.463	3	37517	63487925	bcast_inputs
67.1	491	2:08.326	37200	37200	3450	exchange_1
44.5	6,461	1:25.159	9300	18600	9157	buts
41.0	1:18.436	1:18.436	18600	0	4217	MPI_Recv()
29.5	6,778	56,407	9300	18600	6065	blts
26.2	50,142	50,142	19204	0	2611	MPI_Send()
16.2	24,451	31,031	301	602	103096	rhs
3.9	7,501	7,501	9300	0	807	jacld
3.4	838	6,594	604	1812	10918	exchange_3
3.4	6,590	6,590	9300	0	709	jacu
2.6	4,989	4,989	608	0	8206	MPI_Wait()
0.2	0.44	400	1	4	400081	init_comm
0.2	398	399	1	39	399634	MPI_Init()
0.1	140	247	1	47616	247086	setiv
0.1	131	131	57252	0	2	exact
0.1	89	103	1	2	103168	erhs
0.1	0.966	96	1	2	96458	read_input
0.0	95	95	9	0	10603	MPI_Bcast()
0.0	26	44	1	7937	44878	error
0.0	24	24	608	0	40	MPI_Irecv()
0.0	15	15	1	5	15630	MPI_Finalize()
0.0	4	12	1	1700	12335	setbv
0.0	7	8	3	3	2893	l2norm
0.0	3	3	8	0	491	MPI_Allreduce()
0.0	1	3	1	6	3874	pintgr
0.0	1	1	1	0	1007	MPI_Barrier()
0.0	0.116	0.837	1	4	837	exchange_4
0.0	0.512	0.512	1	0	512	MPI_Keyval_create()
0.0	0.121	0.353	1	2	353	exchange_5
0.0	0.024	0.191	1	2	191	exchange_6
0.0	0.103	0.103	6	0	17	MPI_Type_contiguous()

--:-- NPB_LU.out (Fundamental)--L8--Top

Terminology – Example

For routine “int main( )”:

Exclusive time

100-20-50-20=10 secs

Inclusive time

100 secs

Calls

1 call

Subrs (no. of child routines called)

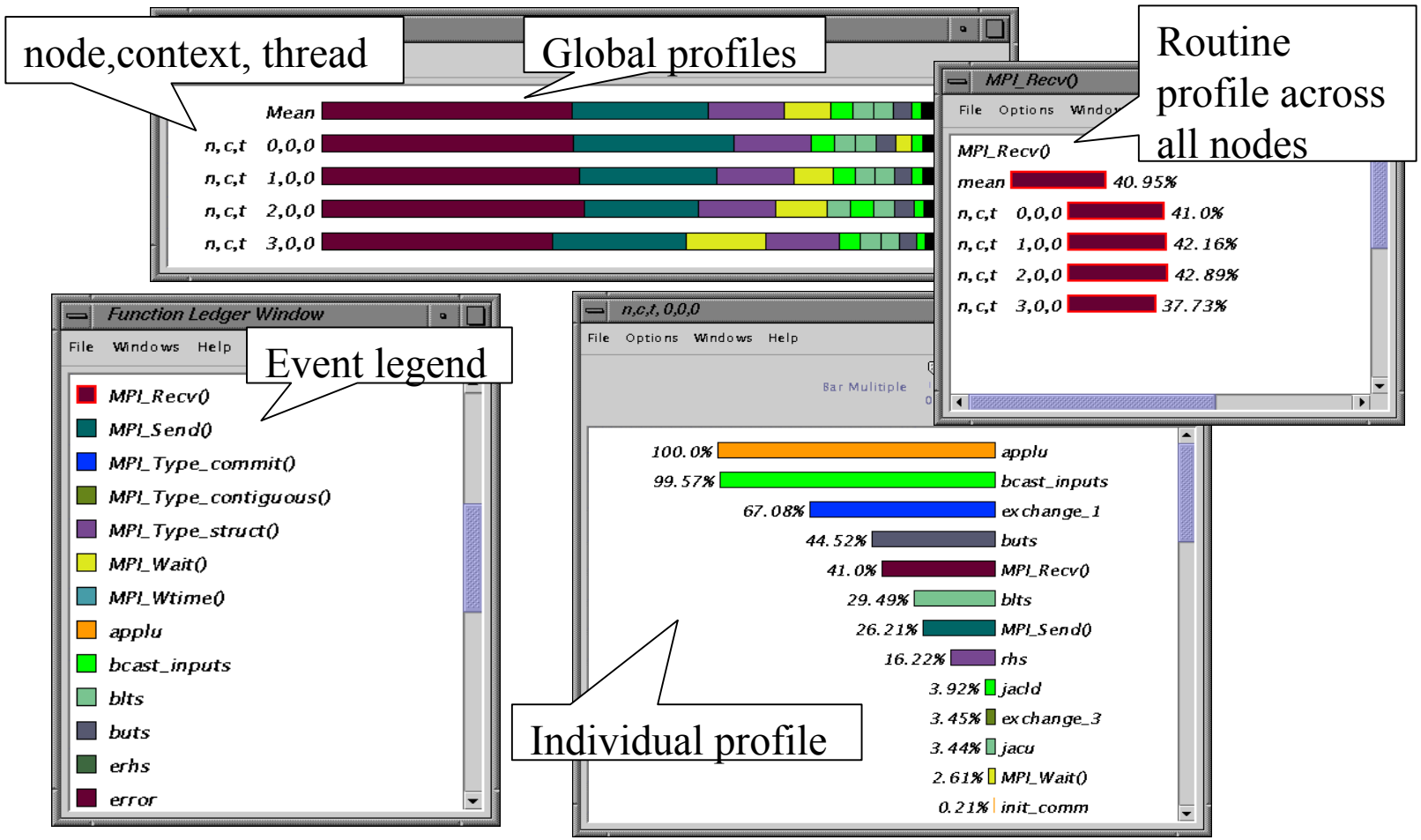
3

Inclusive time/call

100secs

```
int main( )
{ /* takes 100 secs */

  f1(); /* takes 20 secs */
  f2(); /* takes 50 secs */
  f1(); /* takes 20 secs */

  /* other work */
}

/*
Time can be replaced by  counts
from PAPI e.g., PAPI_FP_INS. */
```

ParaProf (NAS Parallel Benchmark – LU)



Using TAU

Install TAU

% configure ; make clean install

Instrument application

TAU Profiling API

Typically modify application makefile

include TAU's stub makefile, modify variables

Set environment variables

directory where profiles/traces are to be stored

Execute application

% mpirun –np <procs> a.out;

Analyze performance data

paraprof, vampir, pprof, paraver …

Description of Optional Packages

PAPI – Measures hardware performance data e.g., floating point instructions, L1 data cache misses etc.

DyninstAPI – Helps instrument an application binary at runtime or rewrites the binary

EPILOG – Trace library. Epilog traces can be analyzed by EXPERT [FZJ], an automated bottleneck detection tool.

Opari – Tool that instruments OpenMP programs

Vampir – Commercial trace visualization tool [formally Pallas, now intelb]

Paraver – Trace visualization tool [CEPBA]

TAU Measurement System Configuration

configure [OPTIONS]

{-c++=<CC>, -cc=<cc>}	Specify C++ and C compilers
{-pthread, -sproc}	Use pthread or SGI sproc threads
-openmp	Use OpenMP threads
-jdk=<dir> (JDK)	Specify Java instrumentation
-opari=<dir> tool	Specify location of Opari OpenMP
-papi=<dir>	Specify location of PAPI
-pdt=<dir>	Specify location of PDT
-dyninst=<dir> Package	Specify location of DynInst
-mpi[inc/lib]=<dir> instrumentation	Specify MPI library
-python[inc/lib]=<dir>	Specify Python instrumentation
-epilog=<dir>	Specify location of EPILOG

TAU Measurement System Configuration

configure [OPTIONS]

-TRACE	Generate binary TAU traces
-PROFILE (default)	Generate profiles (summary)
-PROFILECALLPATH	Generate call path profiles
-PROFILEMEMORY	Track heap memory for each routine
-MULTIPLECOUNTERS	Use hardware counters + time
-COMPENSATE	Compensate timer overhead
-CPUTIME	Use usertime+system time
-PAPIWALLCLOCK	Use PAPI's wallclock time
-PAPIVIRTUAL	Use PAPI's process virtual time
-SGITIMERS	Use fast IRIX timers
-LINUXTIMERS	Use fast x86 Linux timers

Compiling

```
% configure [options]
% make clean install
```

Creates <arch>/lib/Makefile.tau<options> stub Makefile

and <arch>/lib/libTau<options>.a [.so] libraries which defines a single configuration of TAU

Compiling: TAU Makefiles

Include TAU Stub Makefile (<arch>/lib) in the user's Makefile.
Variables:

TAU_CXX	Specify the C++ compiler used by TAU
TAU_CC, TAU_F90	Specify the C, F90 compilers
TAU_DEFS	Defines used by TAU. Add to CFLAGS
TAU_LDFLAGS	Linker options. Add to LDFLAGS
TAU_INCLUDE	Header files include path. Add to CFLAGS
TAU_LIBS	Statically linked TAU library. Add to LIBS
TAU_SHLIBS	Dynamically linked TAU library
TAU_MPI_LIBS	TAU's MPI wrapper library for C/C++
TAU_MPI_FLIBS	TAU's MPI wrapper library for F90
TAU_FORTRANLIBS	Must be linked in with C++ linker for F90
TAU_CXXLIBS	Must be linked in with F90 linker
TAU_INCLUDE_MEMORY	Use TAU's malloc/free wrapper lib
TAU_DISABLE	TAU's dummy F90 stub library

Note: Not including TAU_DEFS in CFLAGS disables instrumentation in C/C++ programs (TAU_DISABLE for f90).

Including TAU Makefile - F90

```
include $PET_HOME/PTOOLS/tau-2.1
F90 = $(TAU_F90)
FFLAGS = -I<dir>
LIBS = $(TAU_LIBS) $(TAU_CXXLIBS)
OBJS = ...
TARGET= a.out
TARGET: $(OBJS)
      $(F90) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
.f.o:
      $(F90) $(FFLAGS) -c $< -o $@
```

TAU Makefile for PDT with MPI

```
include $PET/PTOOLS/tau-2.13.5/rs6000/lib/Makefile.tau-mpi-pdt
FCOMPILE = $(TAU_F90) $(TAU_MPI_INCLUDE)
PDTF95PARSE = $(PDTDIR)/$(PDTARCHDIR)/bin/f95parse
TAUINSTR = $(TAUROOT)/$(CONFIG_ARCH)/bin/tau_instrumentor
PDB=merged.pdb
COMPILE_RULE= $(TAU_INSTR) $(PDB) $< -o $*.inst.f -f sel.dat;\
        $(FCOMPILE) $*.inst.f -o $@;
LIBS = $(TAU_MPI_FLIBS) $(TAU_LIBS) $(TAU_CXXLIBS)
OBJS = f1.o f2.o f3.o …
TARGET= a.out
TARGET: $(PDB) $(OBJS)
        $(TAU_F90) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
$(PDB): $(OBJS:.o=.f)
        $(PDTF95PARSE) $(OBJS:.o=.f) $(TAU_MPI_INCLUDE) -o$(PDB)
# This expands to f95parse *.f -I…/mpi/include -omerged.pdb
.f.o:
        $(COMPILE_RULE)
```

Compensation of Instrumentation Overhead

- Runtime estimation of a single timer overhead
- Evaluation of number of timer calls along a calling path
- Compensation by subtracting timer overhead
- Recalculation of performance metrics to improve the accuracy of measurements
- Configure TAU with –COMPENSATE configuration option

TAU Performance System Status

Computing platforms (selected)

IBM SP / pSeries, SGI Origin 2K/3K, Cray T3E / SV-1 / X1, HP (Compaq) SC (Tru64), Sun, Hitachi SR8000, NEC SX-5/6, Linux clusters (IA-32/64, Alpha, PPC, PA-RISC, Power, Opteron), Apple (G4/5, OS X), Windows

Programming languages

C, C++, Fortran 77/90/95, HPF, Java, OpenMP, Python

Thread libraries

pthreads, SGI sproc, Java,Windows, OpenMP

Compilers (selected)

Intel KAI (KCC, KAP/Pro), PGI, GNU, Fujitsu, Sun, Microsoft, SGI, Cray, IBM (xlc, xlf), Compaq, NEC, Intel

Vampir/VampirTrace

- Vampirtrace is an instrumented MPI library to link with user code for automatic tracefile generation on parallel platforms
- Vampir is a visualization program used to visualize trace data generated by Vampitrace

Vampir/VampirTrace

http://www.pallas.com/e/products/vampir

Version 4.0

Languages and Libraries: C, C++, Fotran77/90/95, HPF, MPI, OpenMP support being worked on

Supported Platforms: Most all HPC platforms (for how long?)

Vampirtrace

Profiling library for MPI applications

Produces tracefiles that can be analyzed with the Vampir performance analysis tool or the Dimemas performance prediction tool.

Merely linking your application with Vampirtrace enables tracing of all MPI calls. On some platforms, calls to user-level subroutines are also recorded.

API for controlling profiling and for defining and tracing user-defined activities.

Vampir Features

- Tool for converting tracefile data for MPI programs into a variety of graphical views
- Highly configurable
- Timeline display with zooming and scrolling capabilities
- Profiling and communications statistics
- Source-code clickback

Running and Analyzing Vampirtrace-instrumented Programs

Programs linked with Vampirtrace are started in the same way as ordinary MPI programs.

Use Vampir to analyze the resulting tracefile.

A configuration file is saved that controls all your default values

An example program

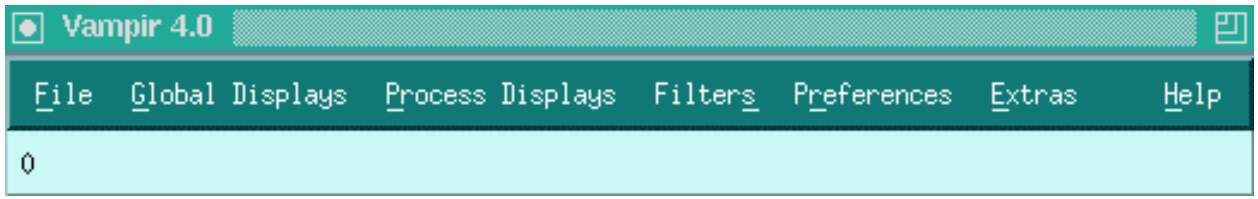
Poisson solver (iterative)

After each iteration, each process must exchange data with both its left and right neighbor

Each process does a sendrecv to its right followed by a sendrecv to its left

Getting Started

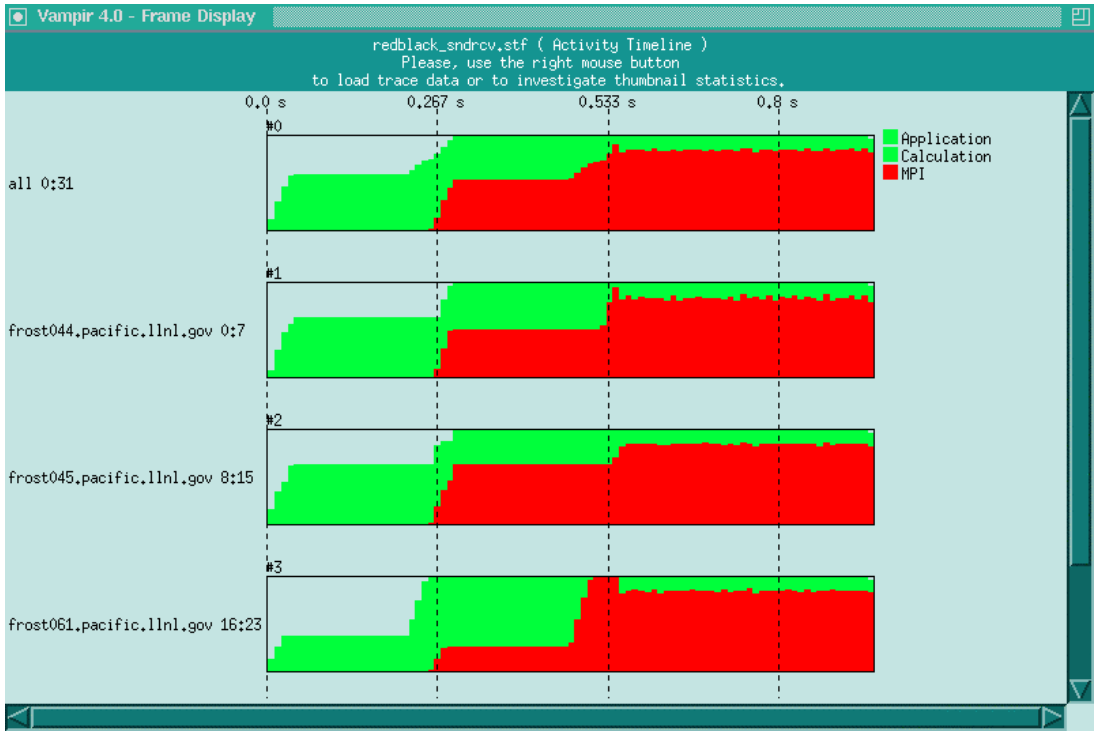
If your path is set up correctly, simply enter “vampir”



To open a tracefile, select “File” followed by “Open Tracefile”
Select tracefile to open or enter a known tracefile

The entire event trace is **not** opened. Rather, metadata
Is read and a frame display is opened. This is a preview
Of the trace

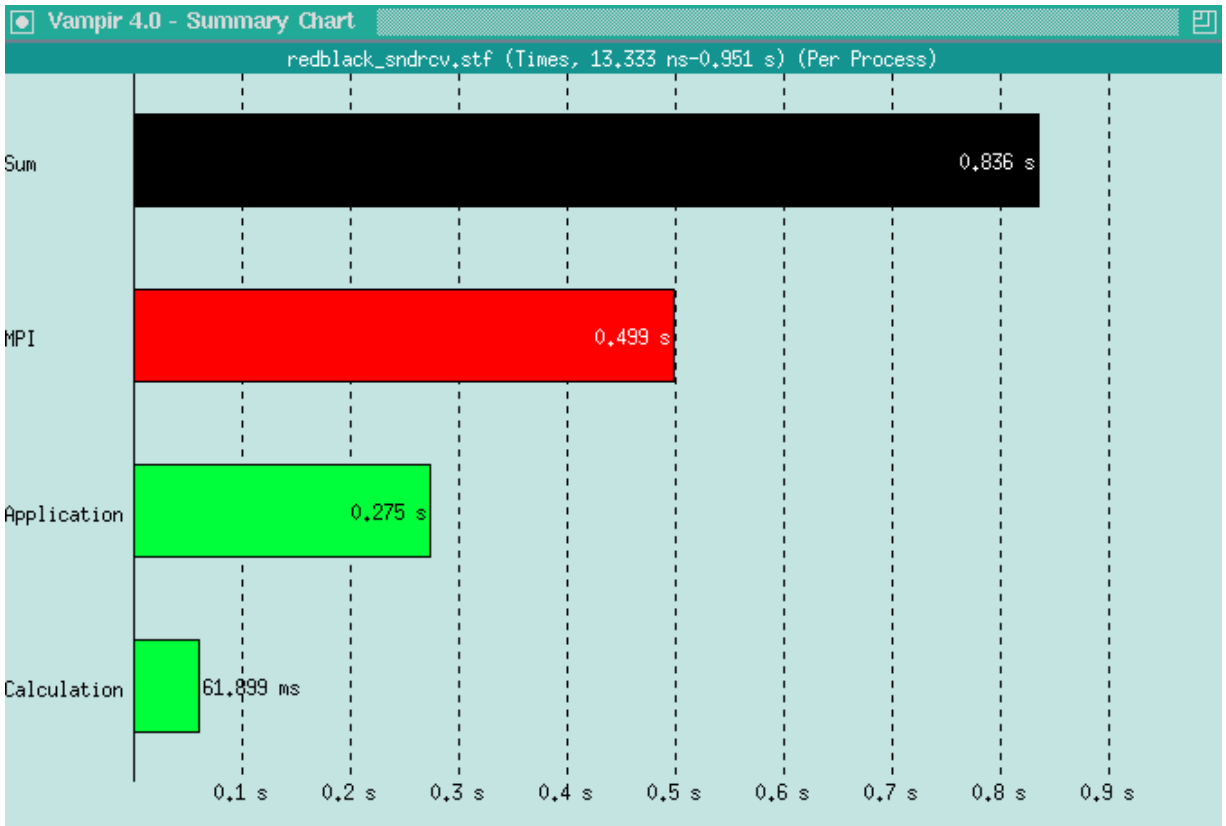
Frame Display



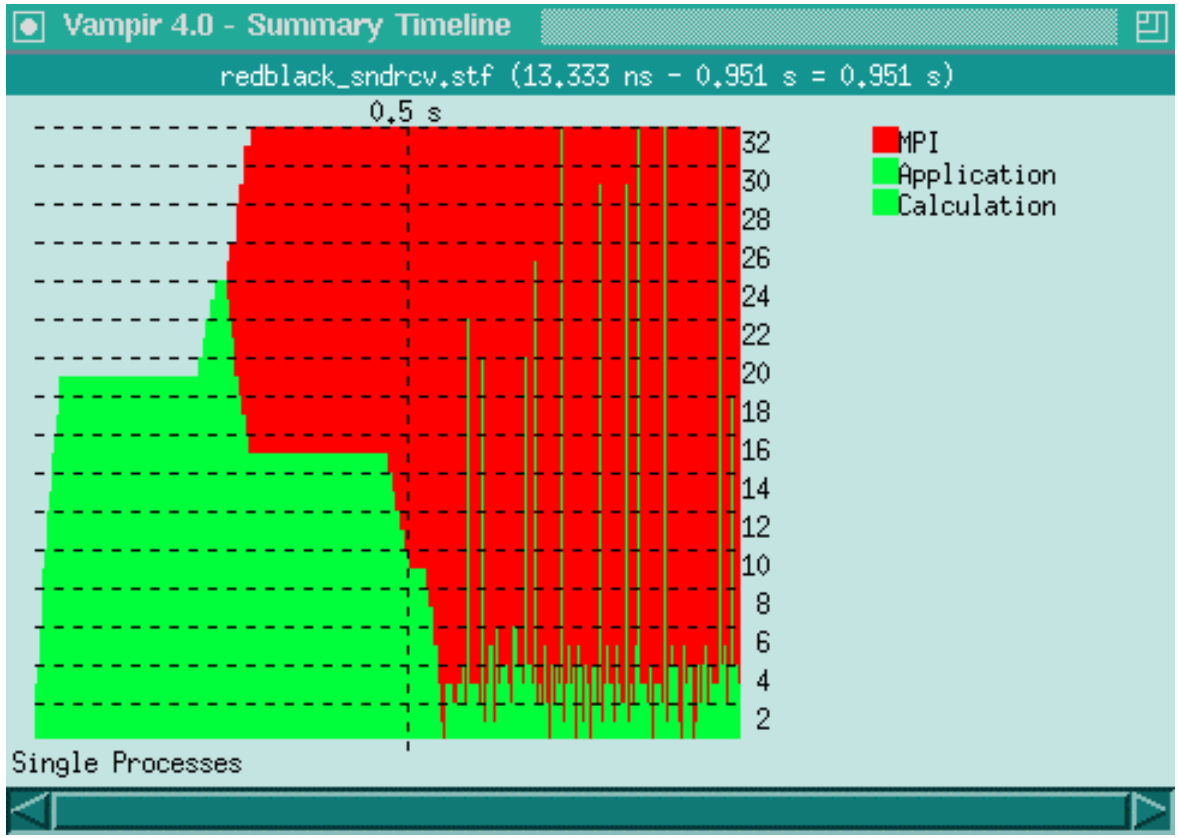
Right click to get a context menu and select load/Whole Trace

Summary Chart

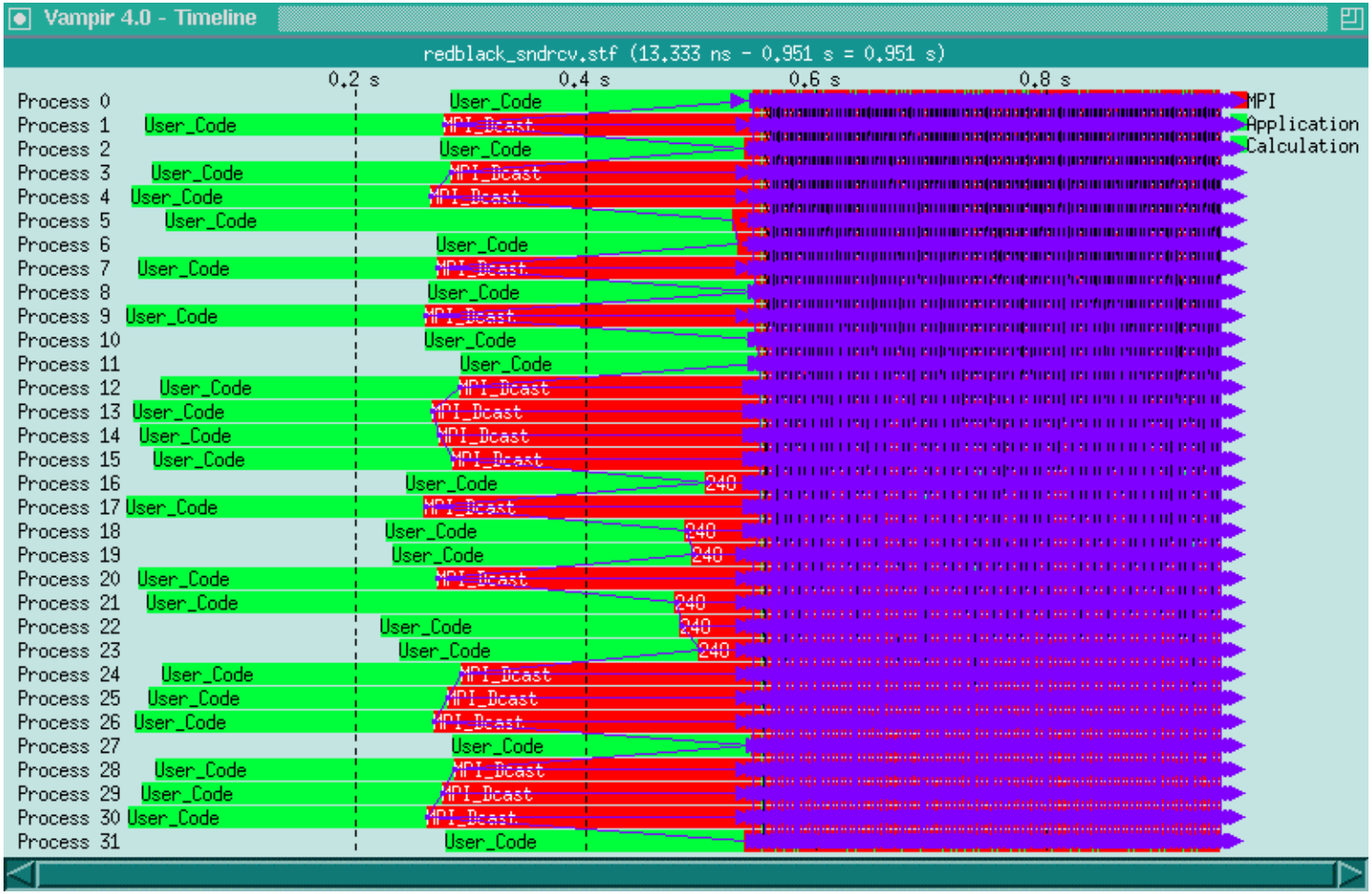
By default, Vampir starts with a summary chart of the entire execution run



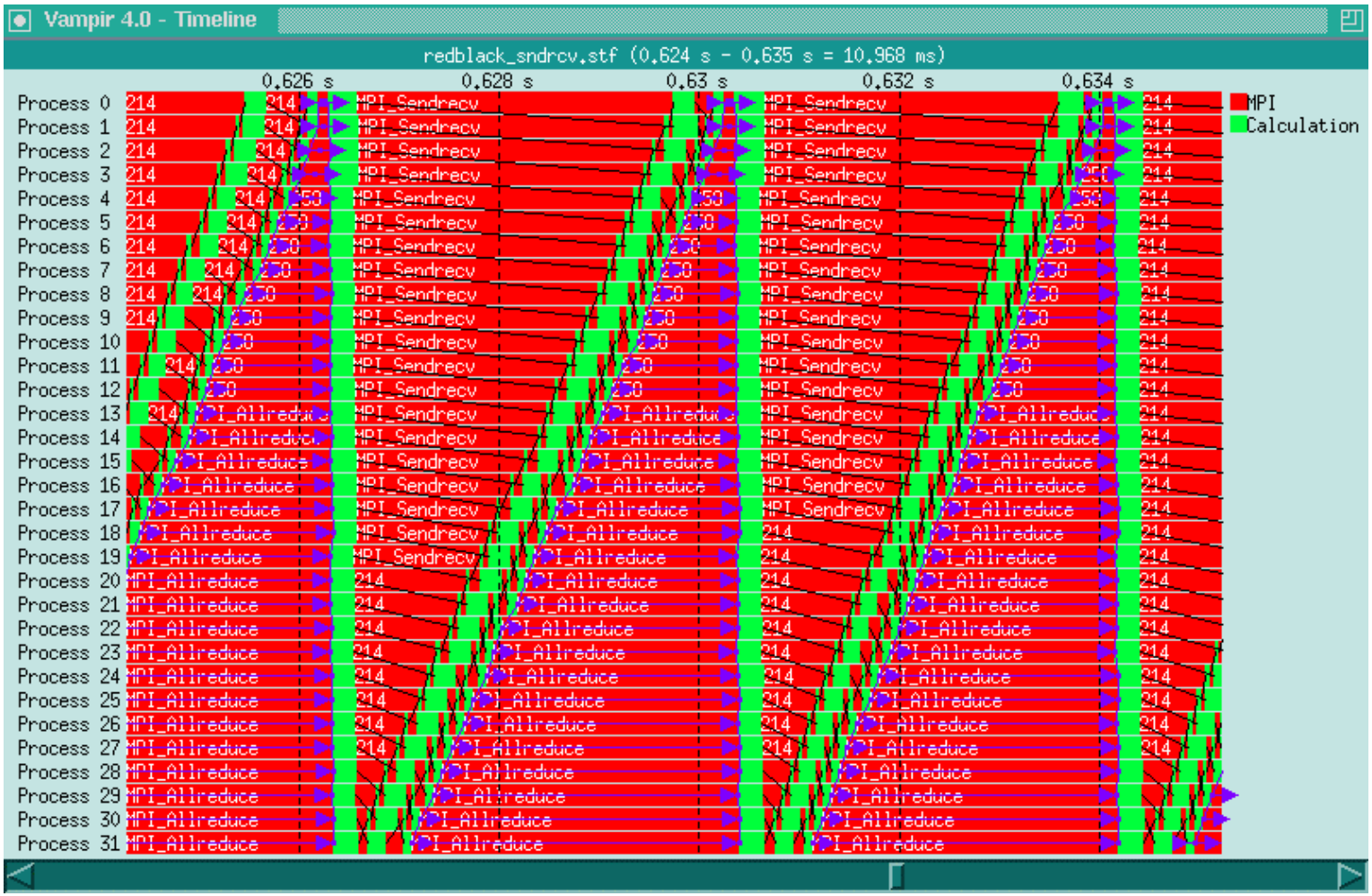
Summary Timeline



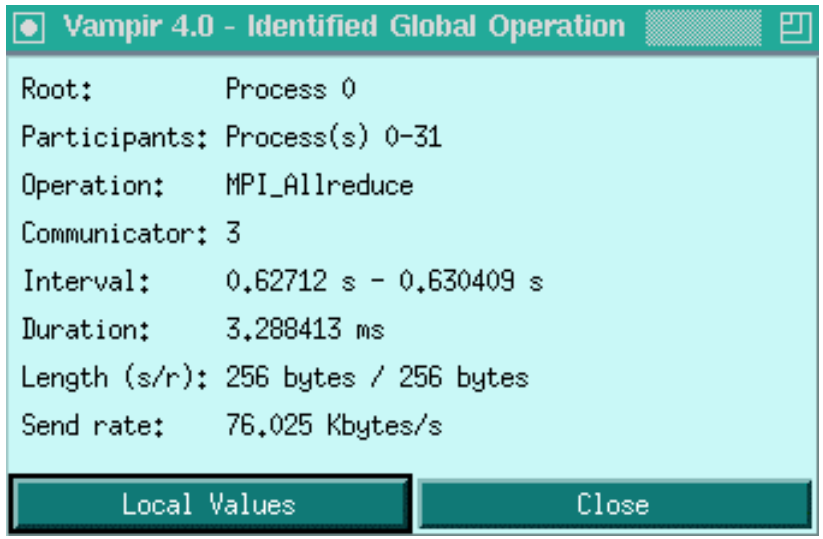
Timeline


Vampir 4.0 - Timeline
redblack_sndrcv.stf (13.333 ns - 0.951 s = 0.951 s)
0.2 s
0.4 s
0.6 s
0.8 s
Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
Process 6
Process 7
Process 8
Process 9
Process 10
Process 11
Process 12
Process 13
Process 14
Process 15
Process 16
Process 17
Process 18
Process 19
Process 20
Process 21
Process 22
Process 23
Process 24
Process 25
Process 26
Process 27
Process 28
Process 29
Process 30
Process 31
User_Code
MPI_Bcast
240
MPI
Application
Calculation

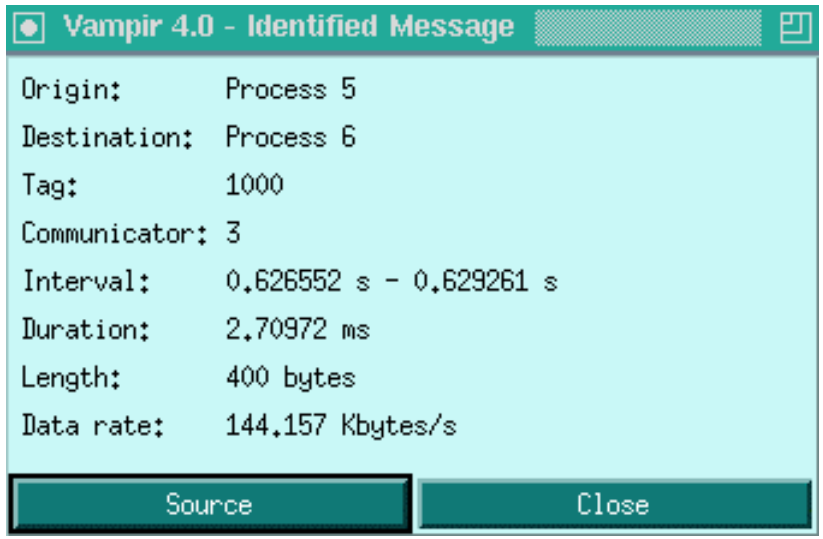
Zoomed Timeline



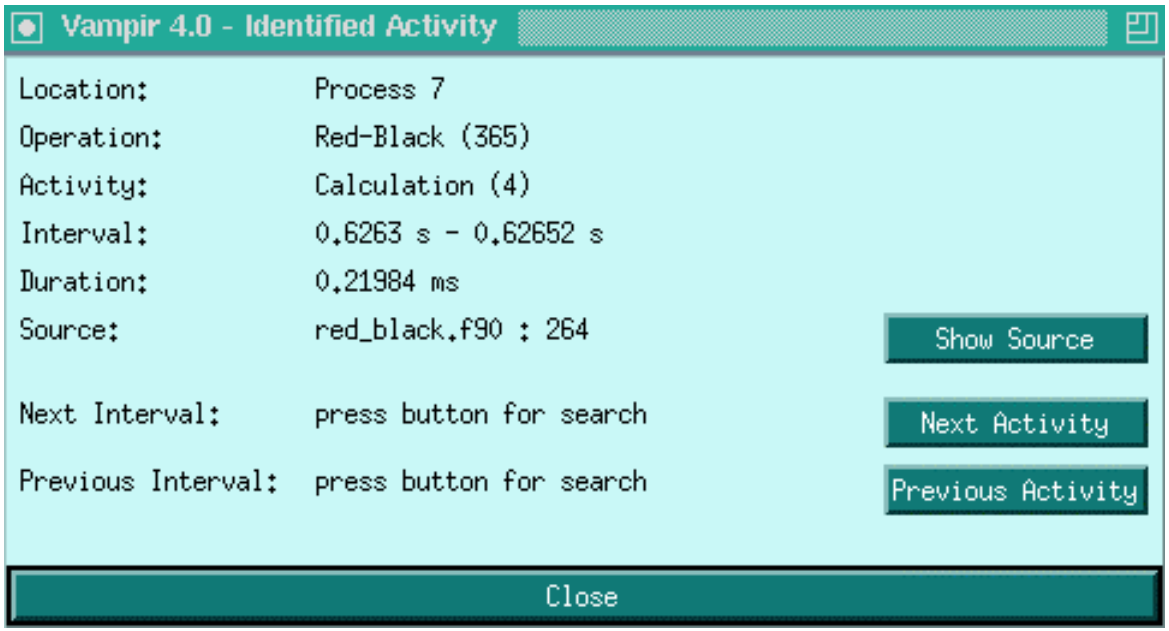
Clicking on an activity



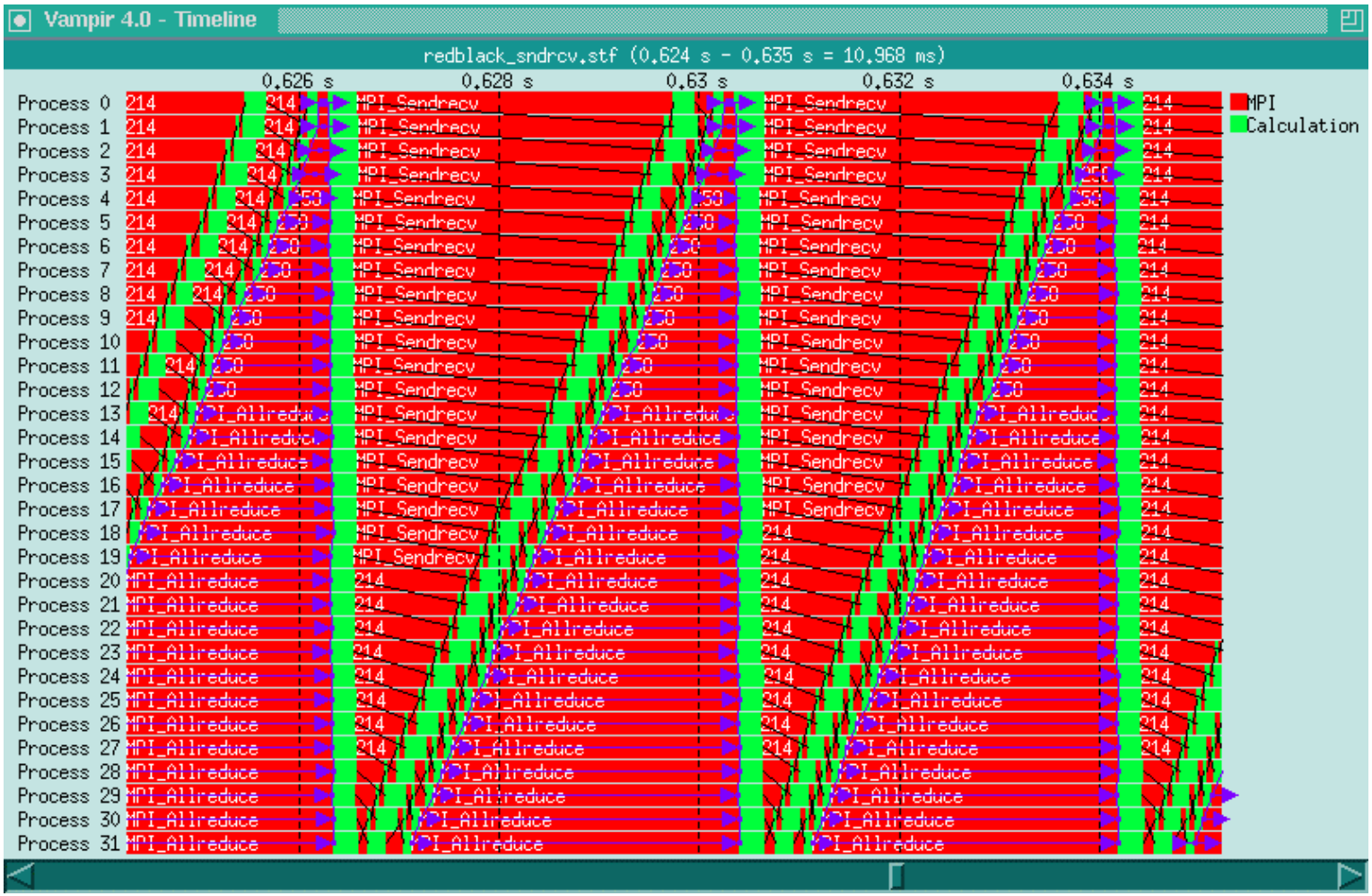
Clicking on an activity



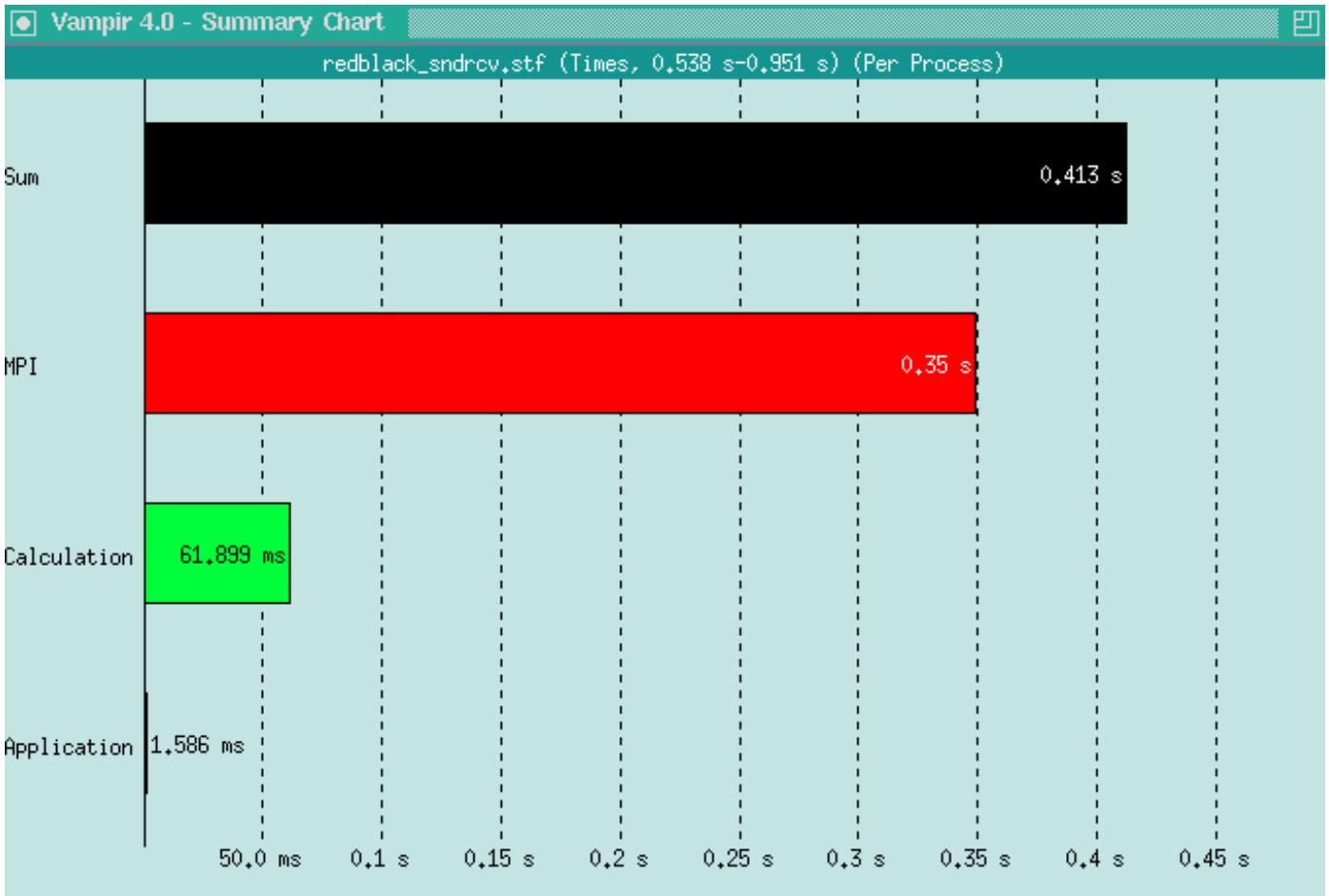
Clicking on an activity



Zoomed Timeline



Zoomed Summary CHart



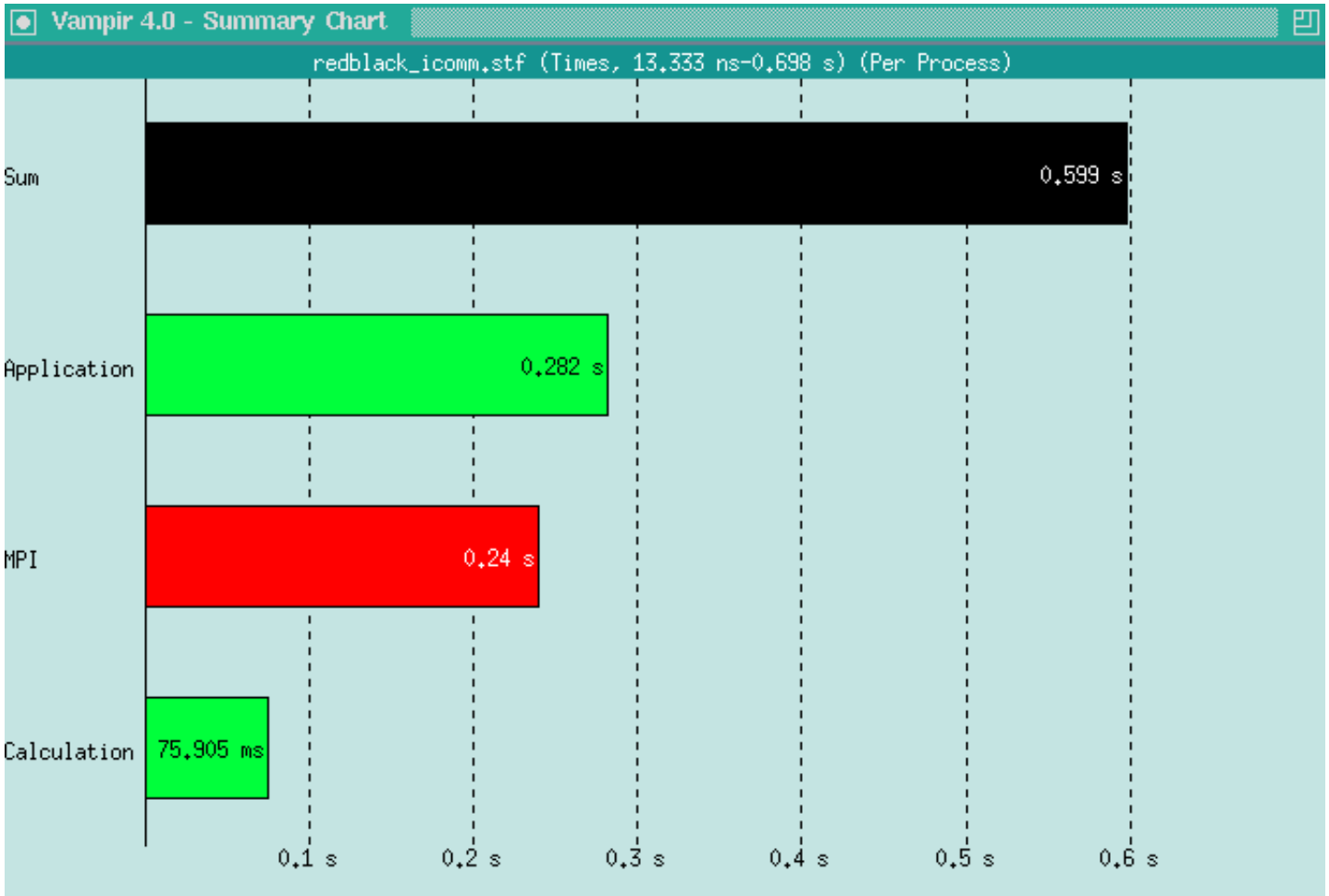
A different approach

Rather than use sendrecv, use non-blocking communication

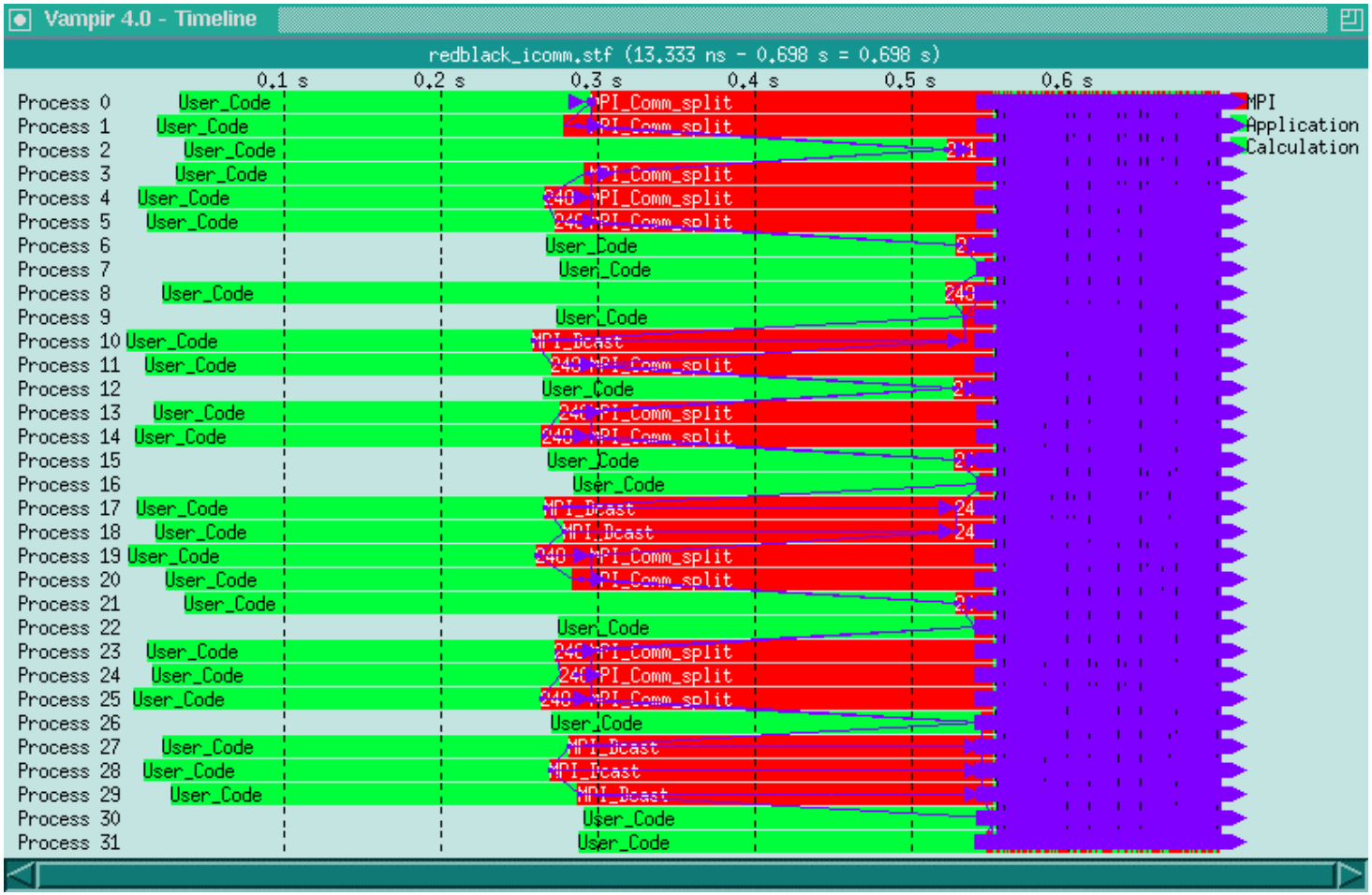
Allows data movement to occur concurrently

Should greatly reduce the amount of time spent waiting for data

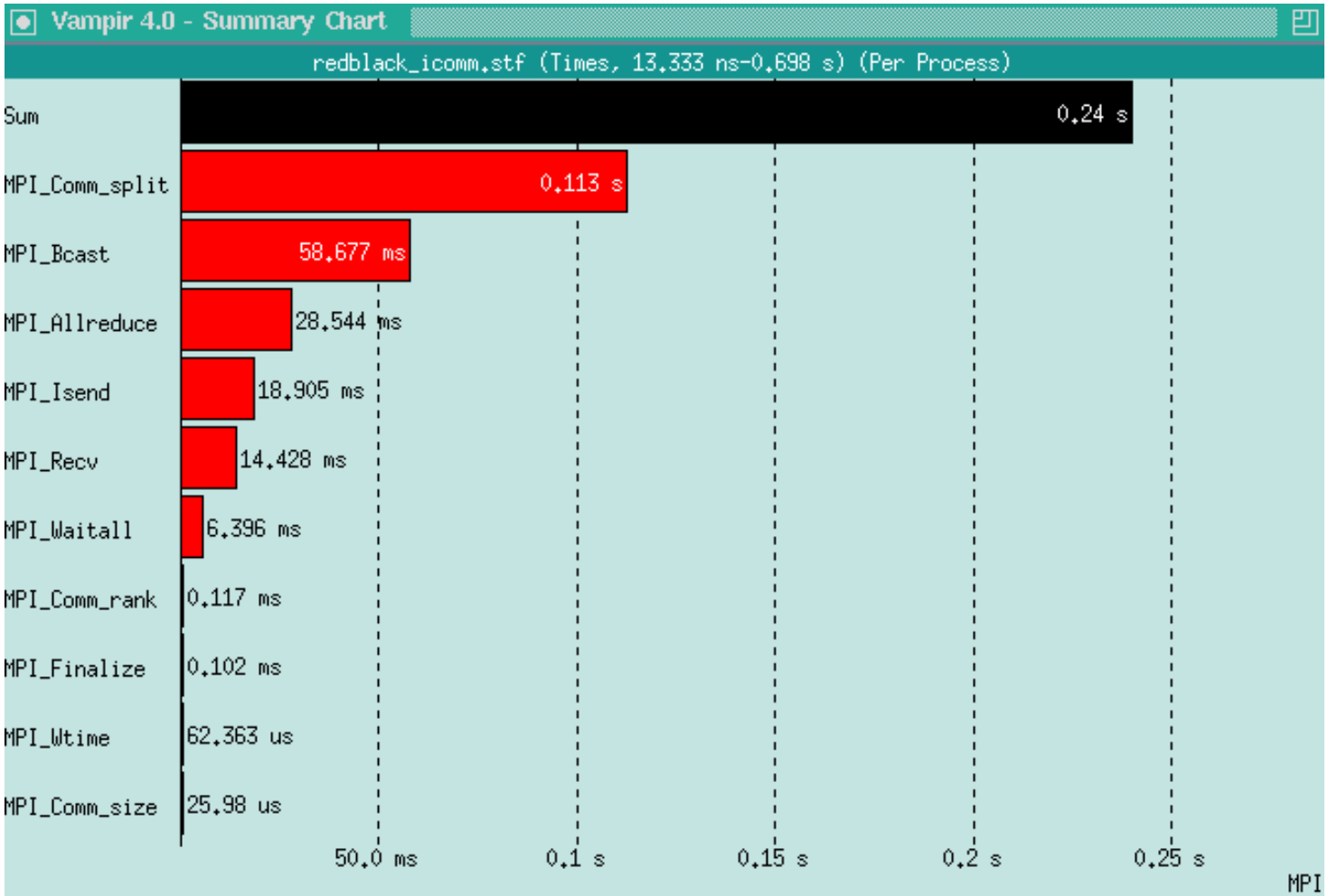
A Different Approach



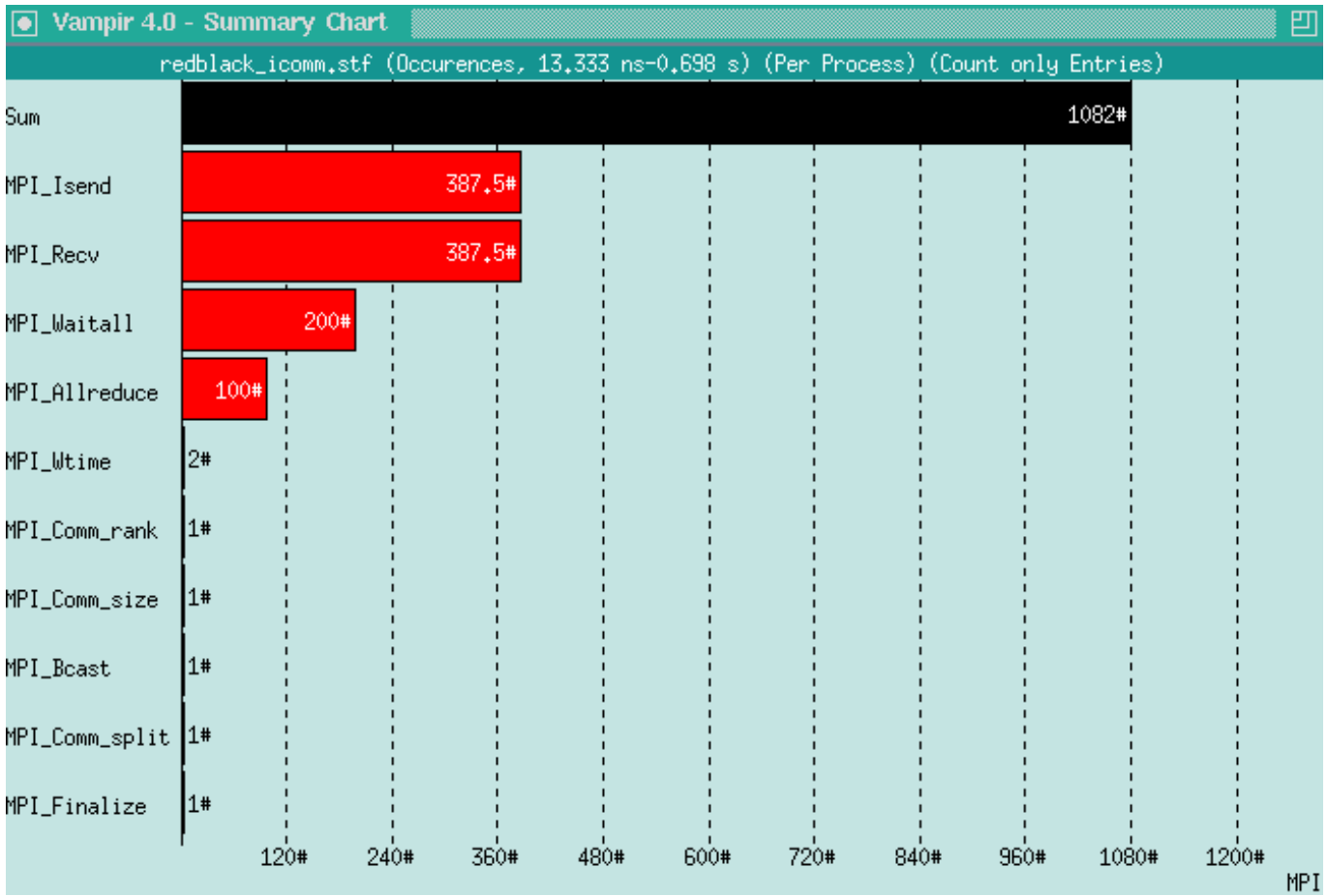
A Different Approach



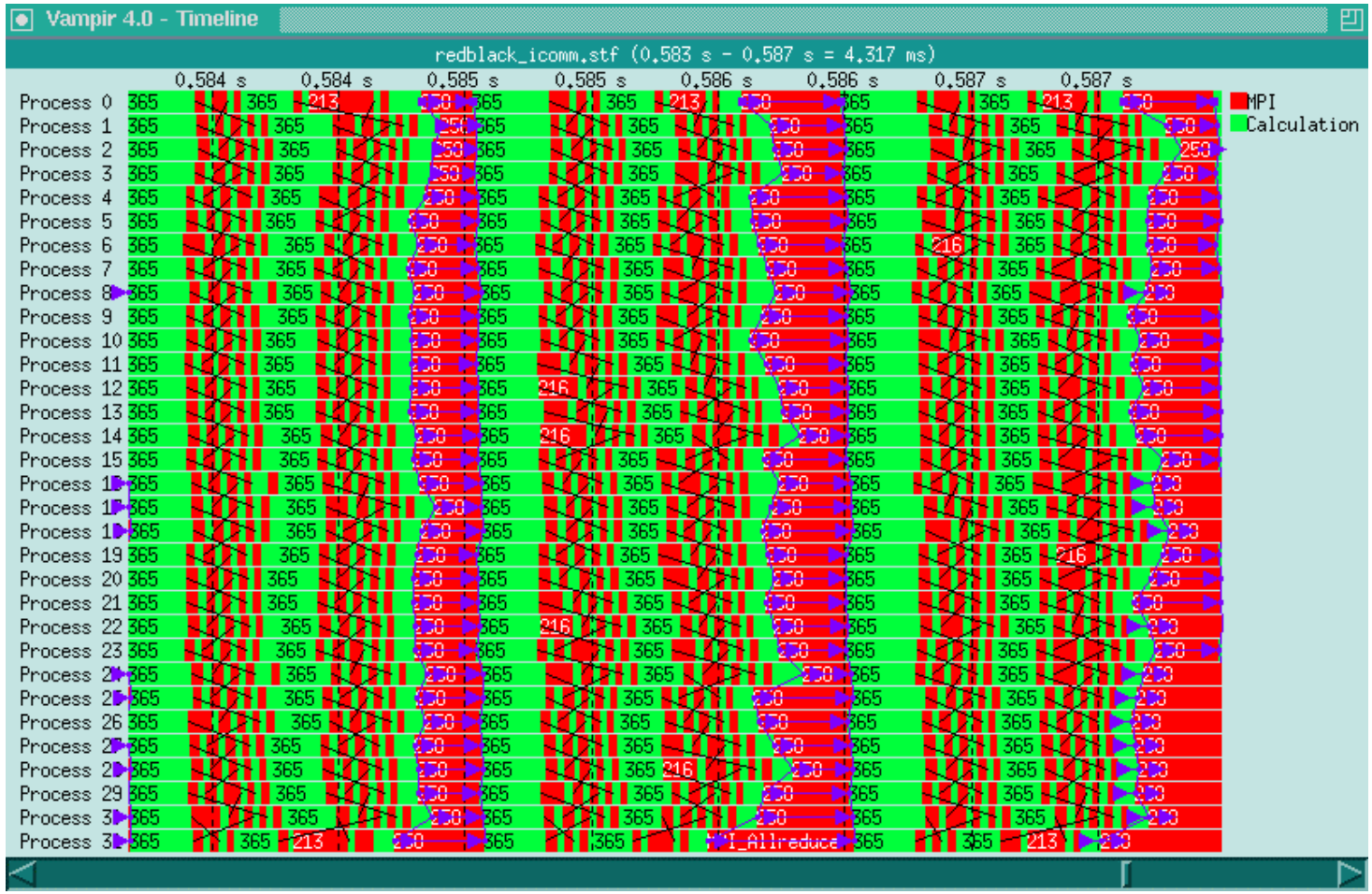
A Different Approach



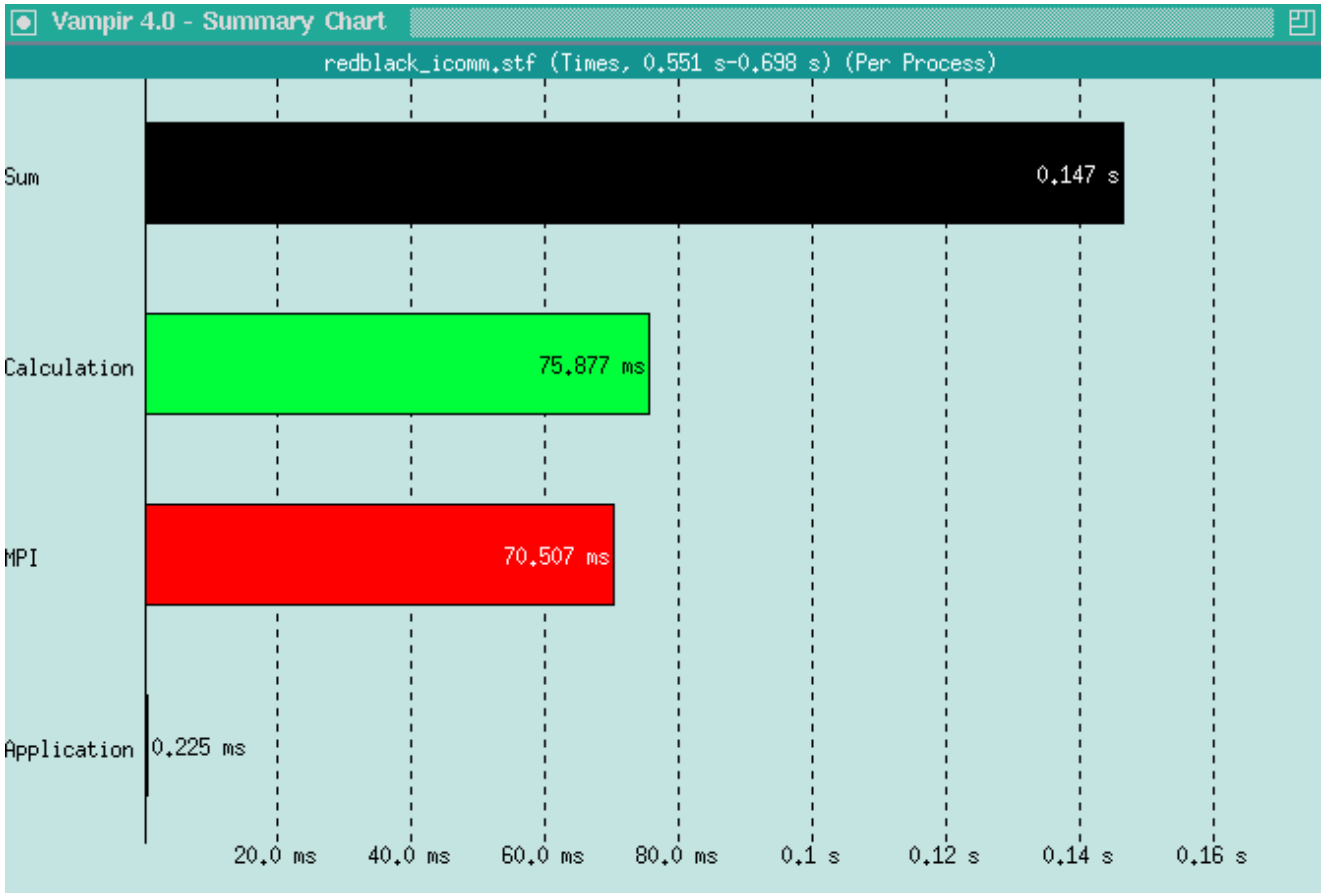
A Different Approach



A Different Approach



A Different Approach



A Different Approach

By switching to non-blocking communication we have reduced the overall execution time.

Much of the remaining time is from start-up

We have eliminated the sever imbalance in wait time

There is still a high ratio of MPI to application

- Probably due to not having a large enough problem size

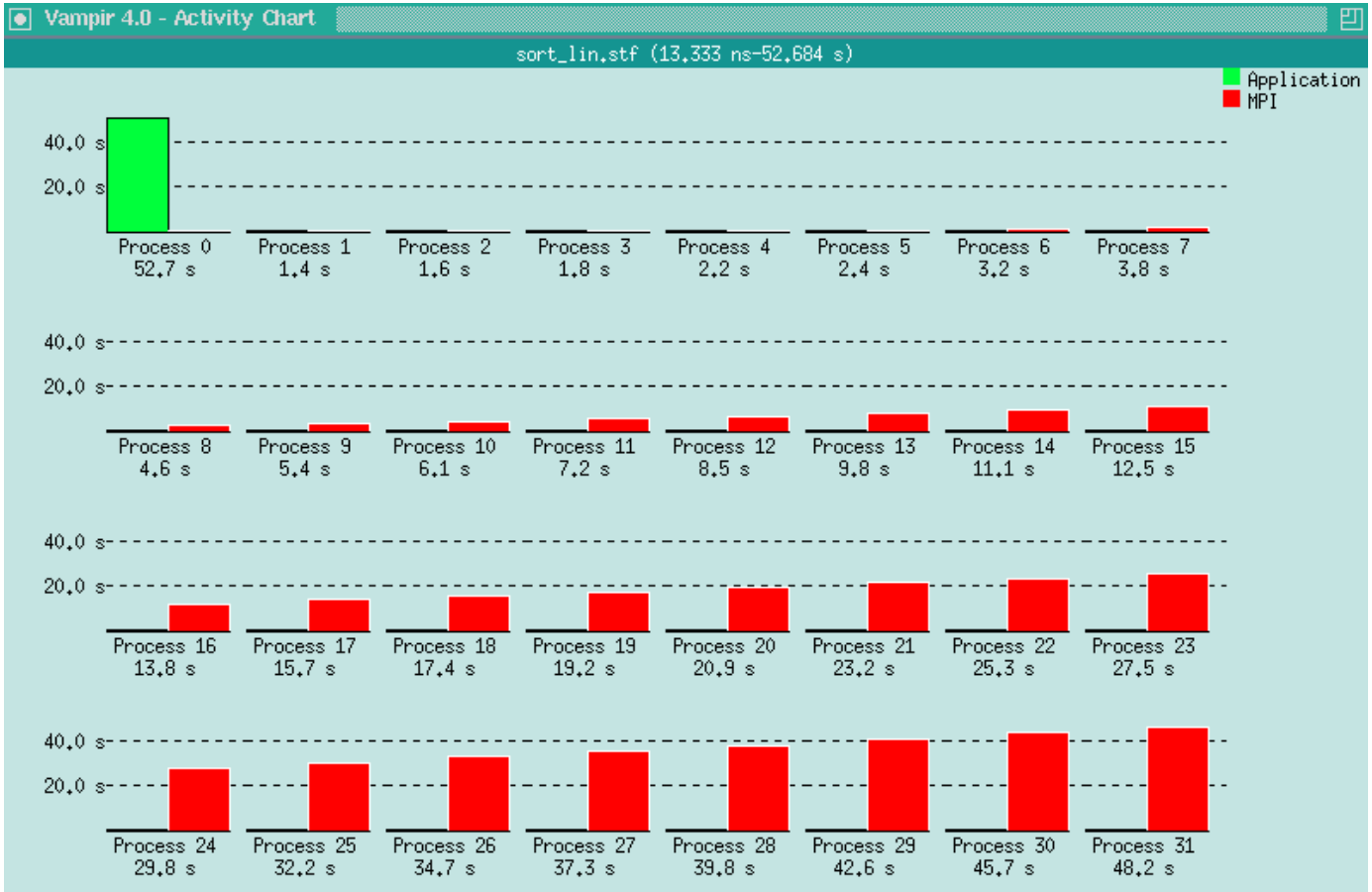
Another example

Parallel sort

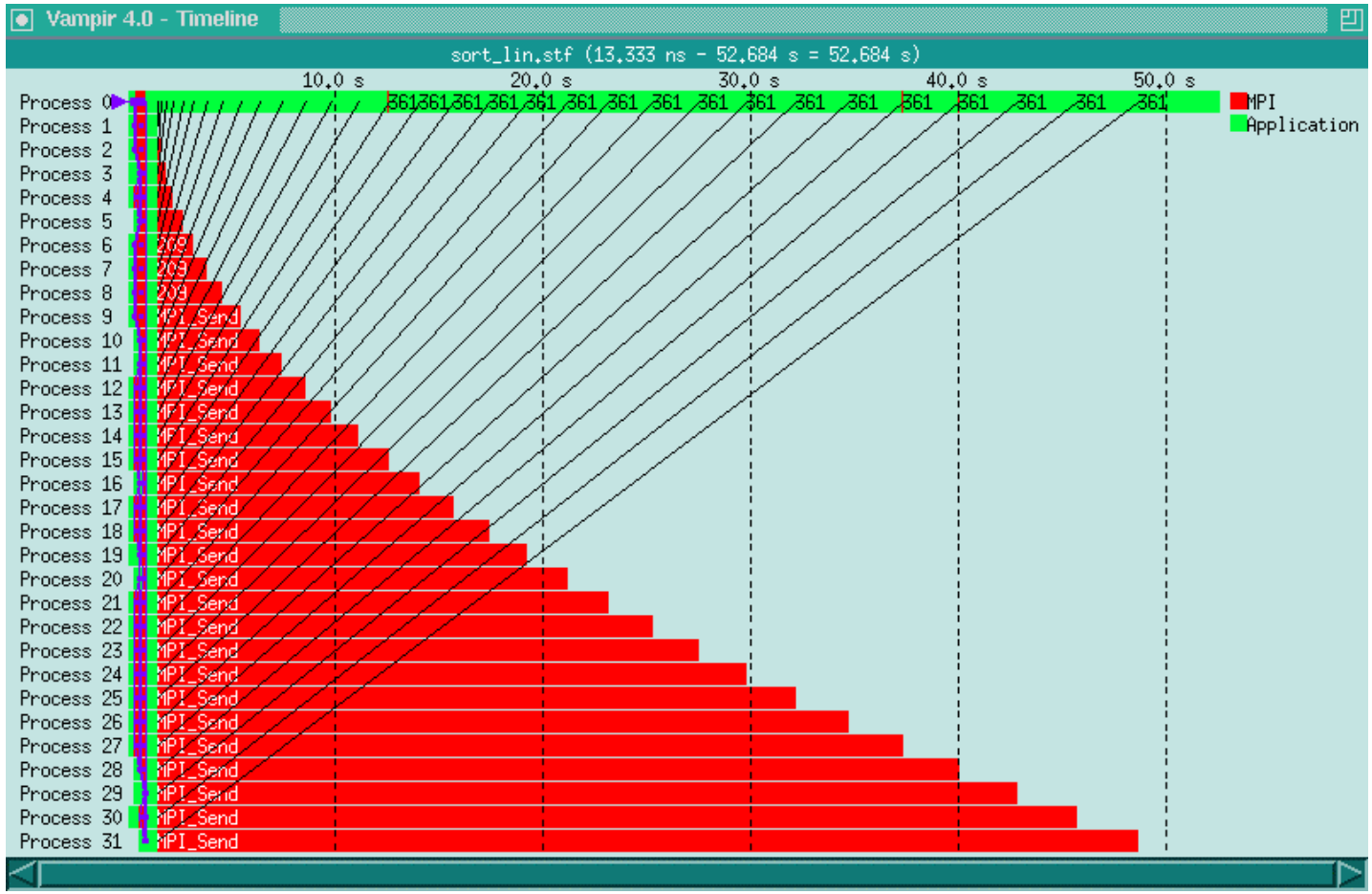
Each process sorts its portion of the data and sends the results to process 0

Process 0 merges the results into a final sort

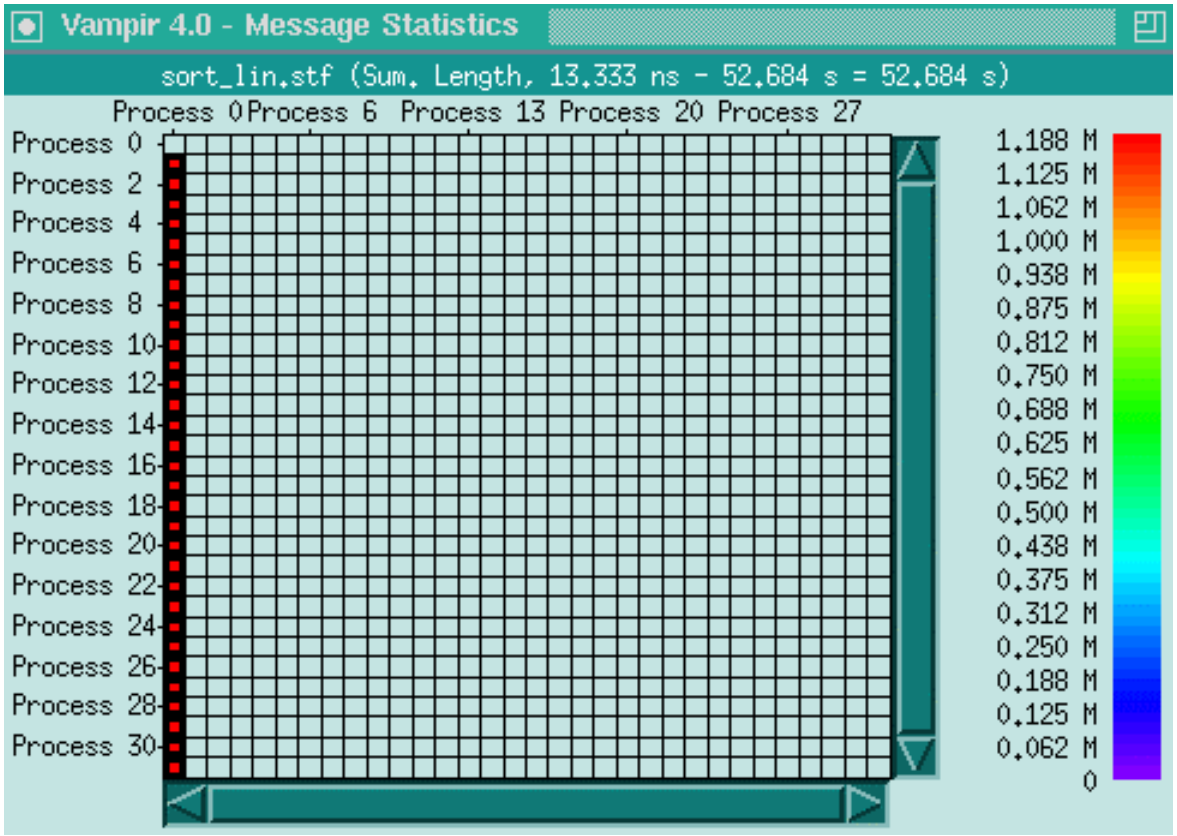
Activity Chart



Timeline


Vampir 4.0 - Timeline
sort_lin.stf (13.333 ns - 52.684 s = 52.684 s)
10.0 s
20.0 s
30.0 s
40.0 s
50.0 s
Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
Process 6
Process 7
Process 8
Process 9
Process 10
Process 11
Process 12
Process 13
Process 14
Process 15
Process 16
Process 17
Process 18
Process 19
Process 20
Process 21
Process 22
Process 23
Process 24
Process 25
Process 26
Process 27
Process 28
Process 29
Process 30
Process 31
MPI_Send
MPI
Application

Message statistics



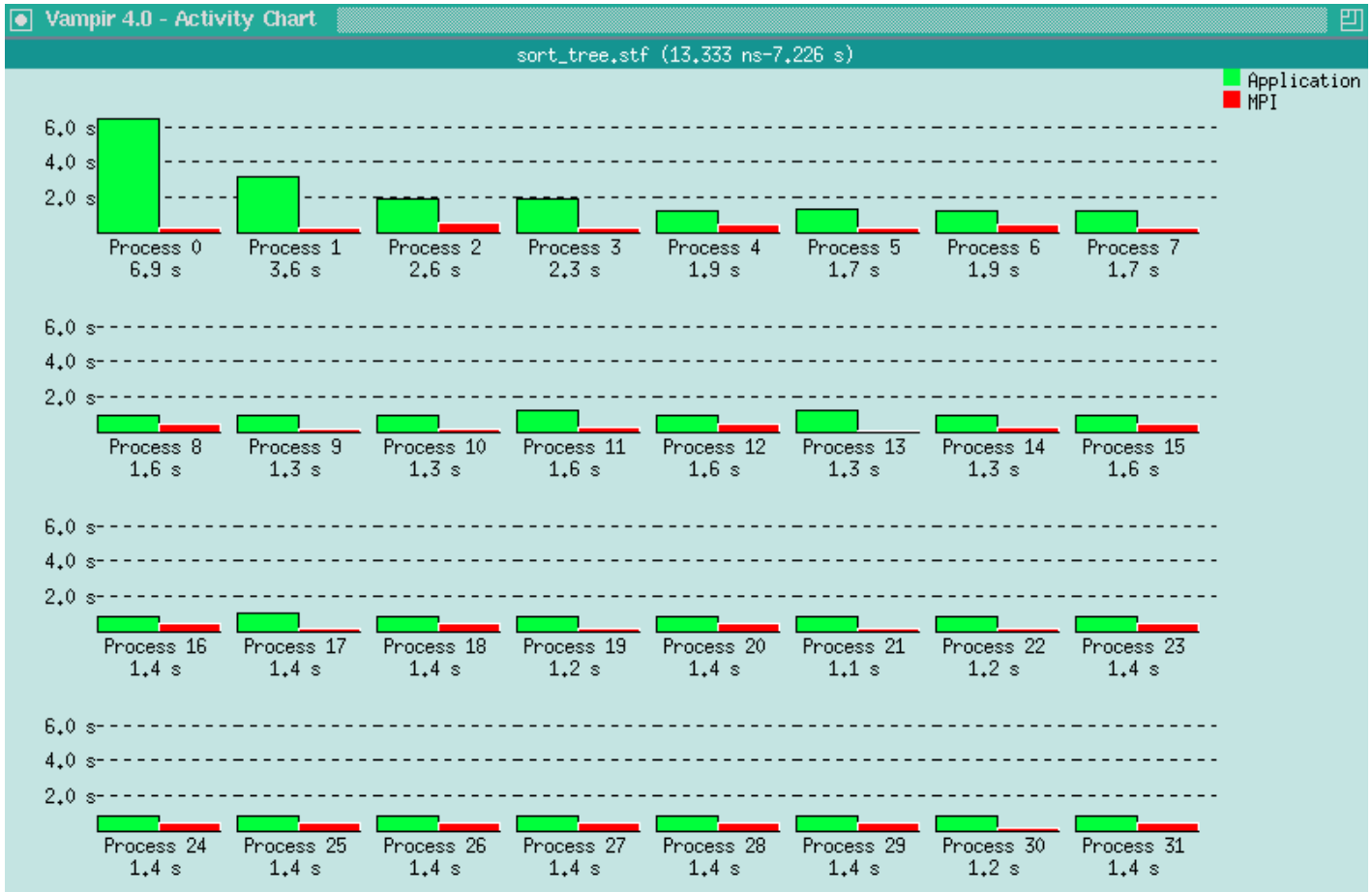
A different approach to sort

Each process still sorts its local data

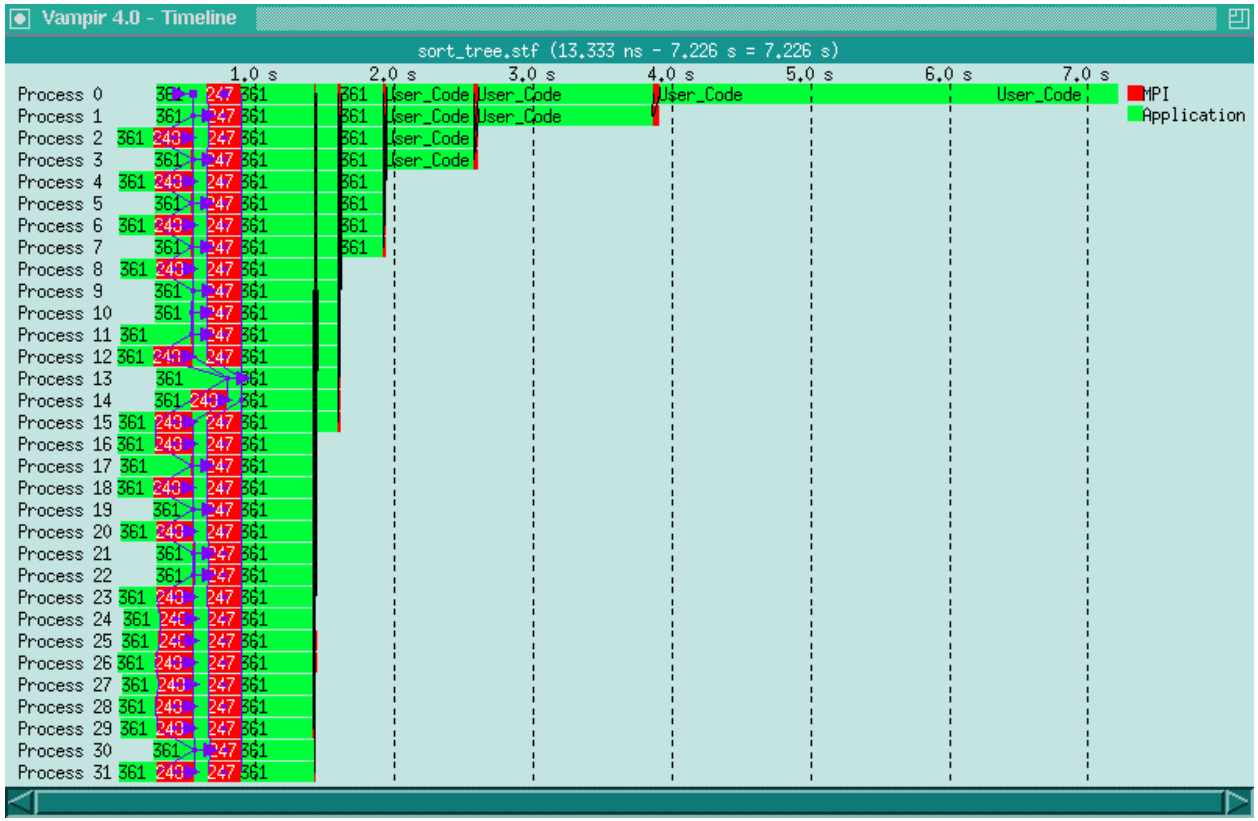
Pass the data based on a tree algorithm, with half the processes receiving data and merging it

Continue up the tree to the root

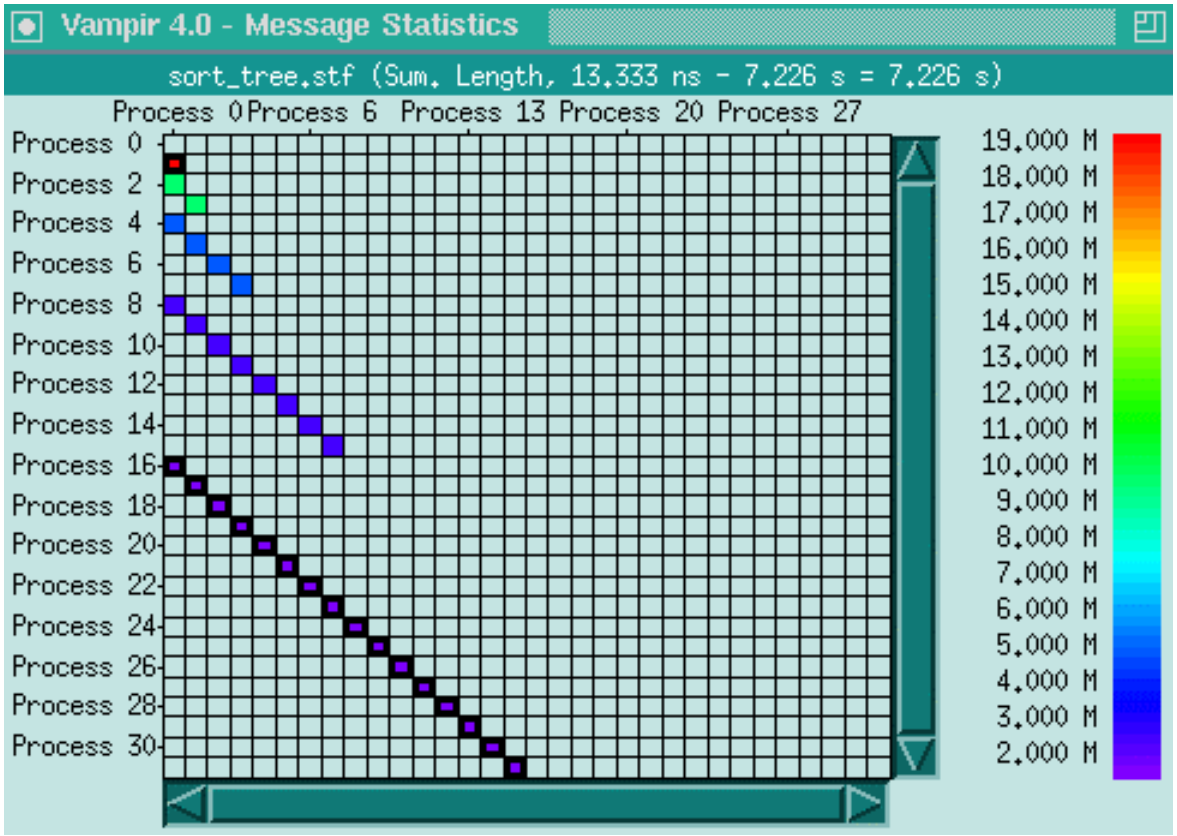
A different approach to sort



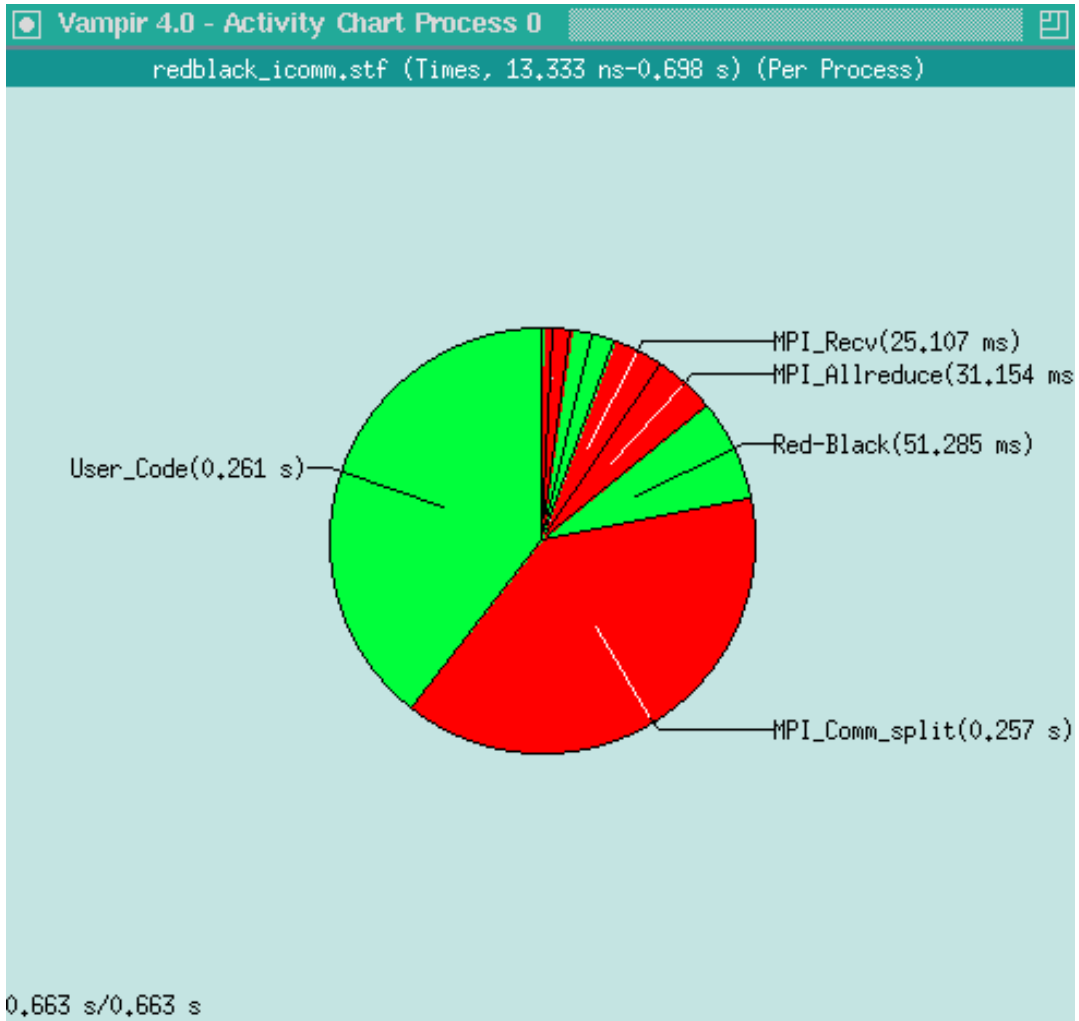
A different approach to sort



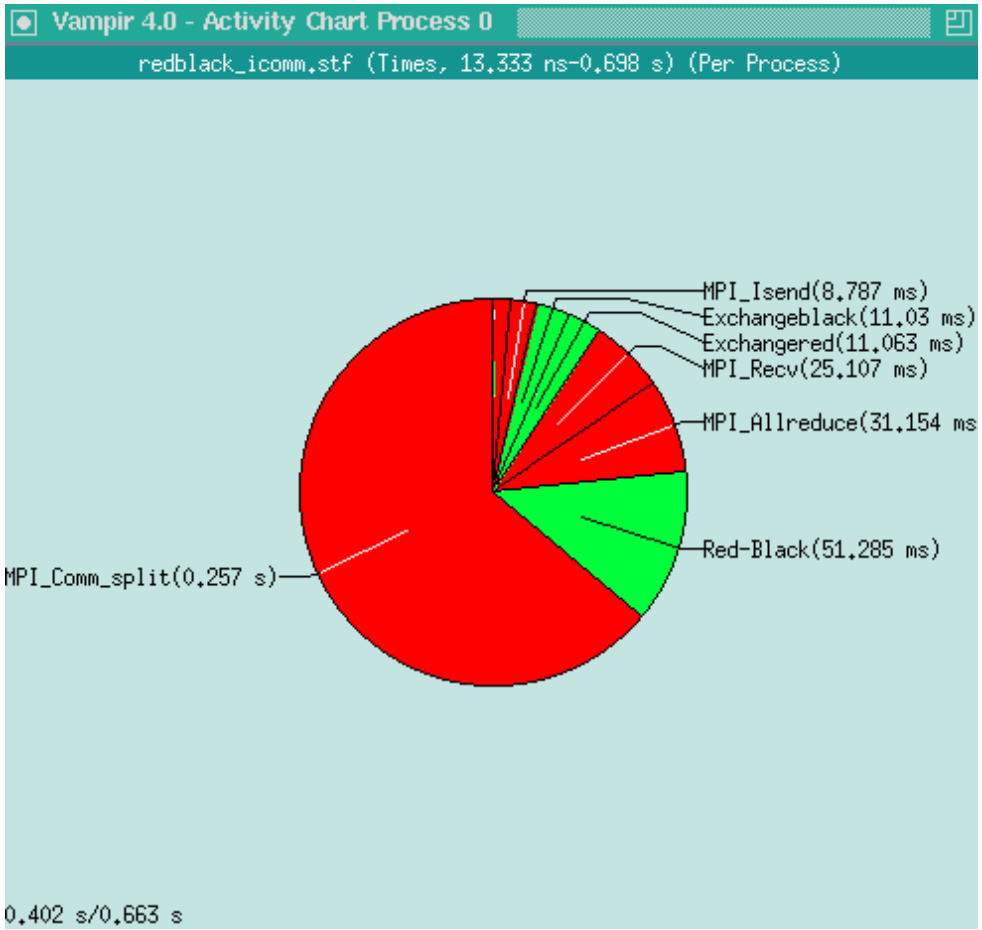
A different approach to sort



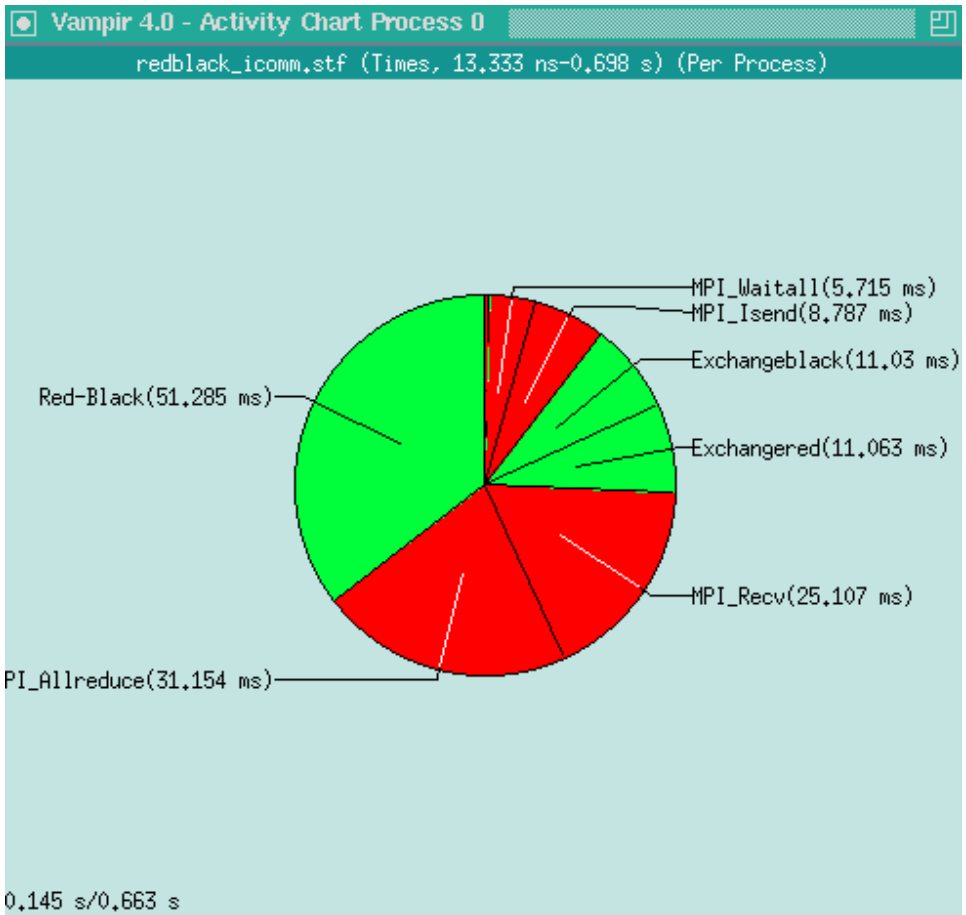
Process Activity Chart Displays



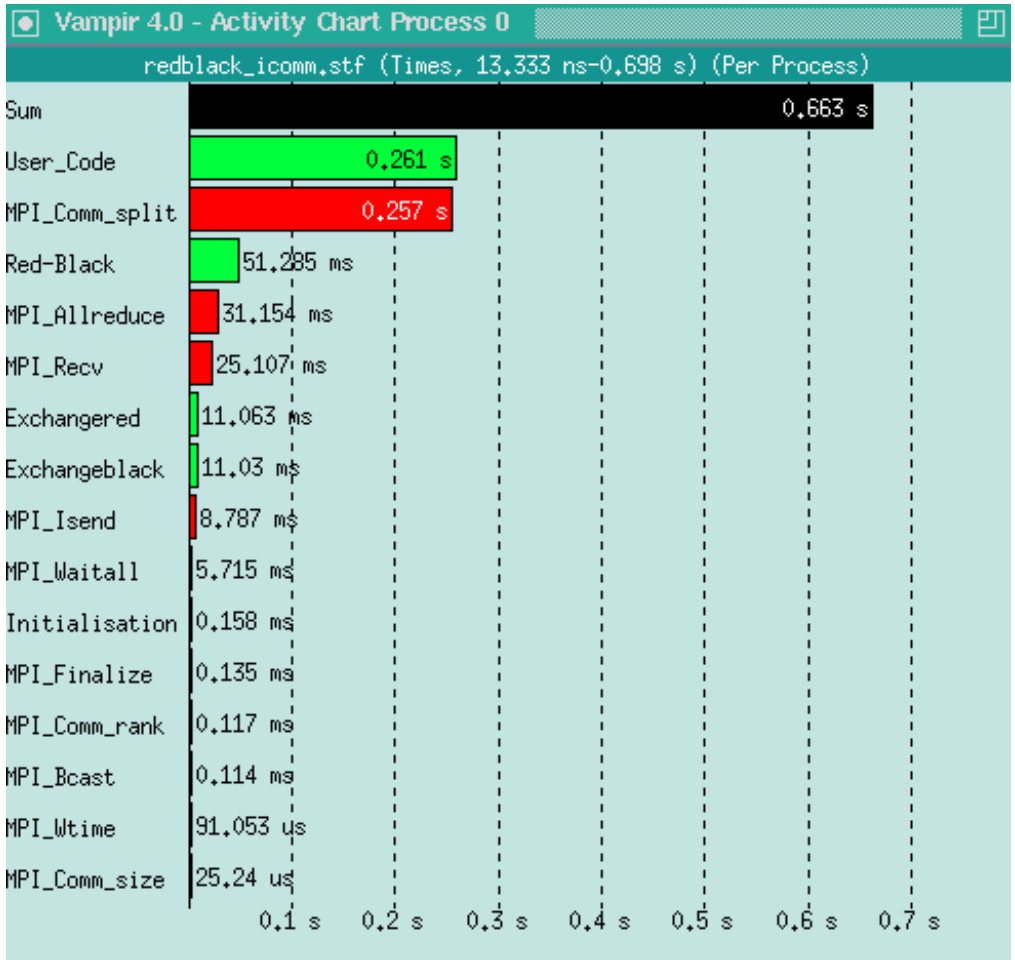
Process Activity Chart Displays



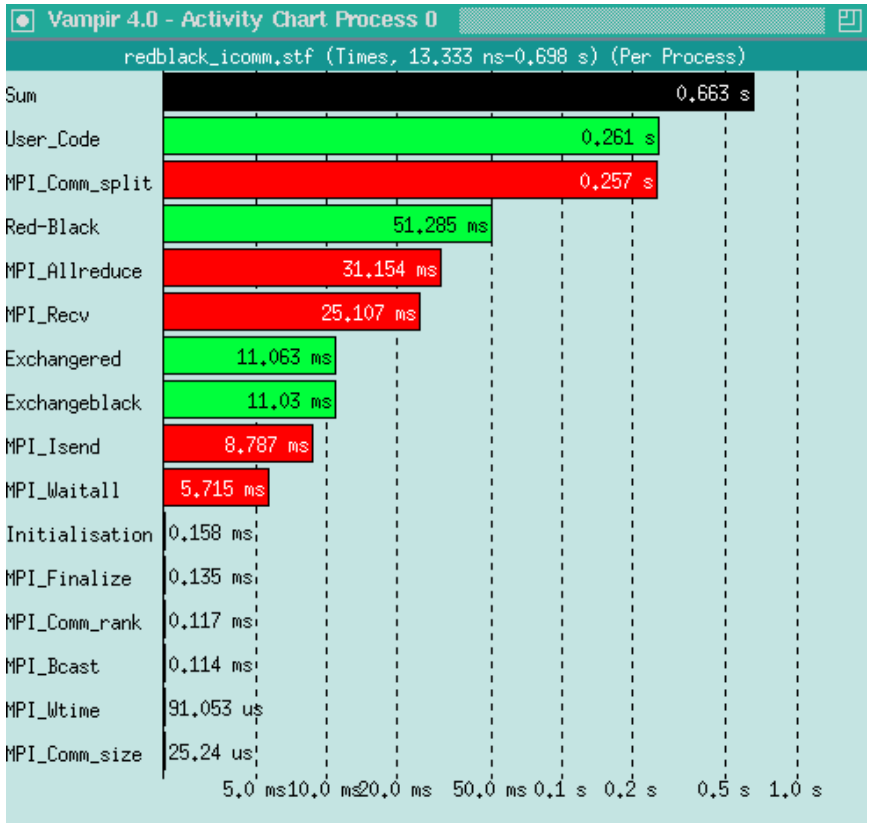
Process Activity Chart Displays



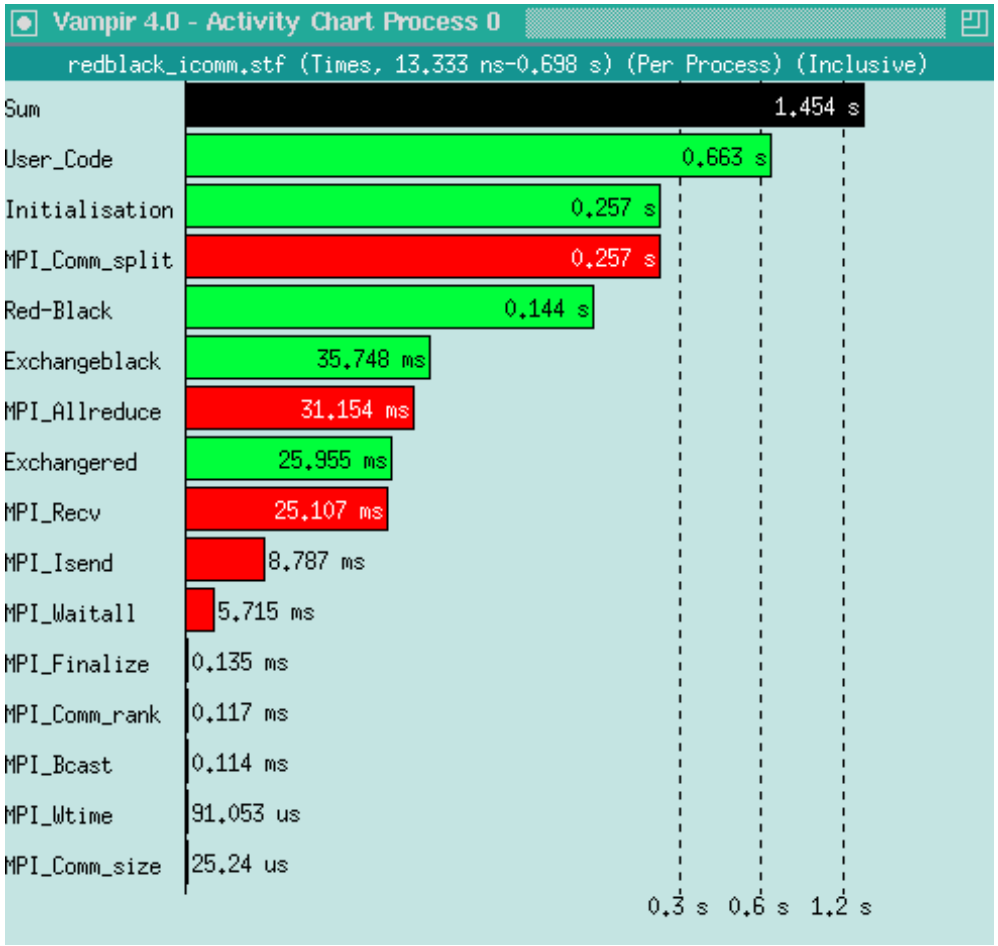
Process Activity Chart Displays



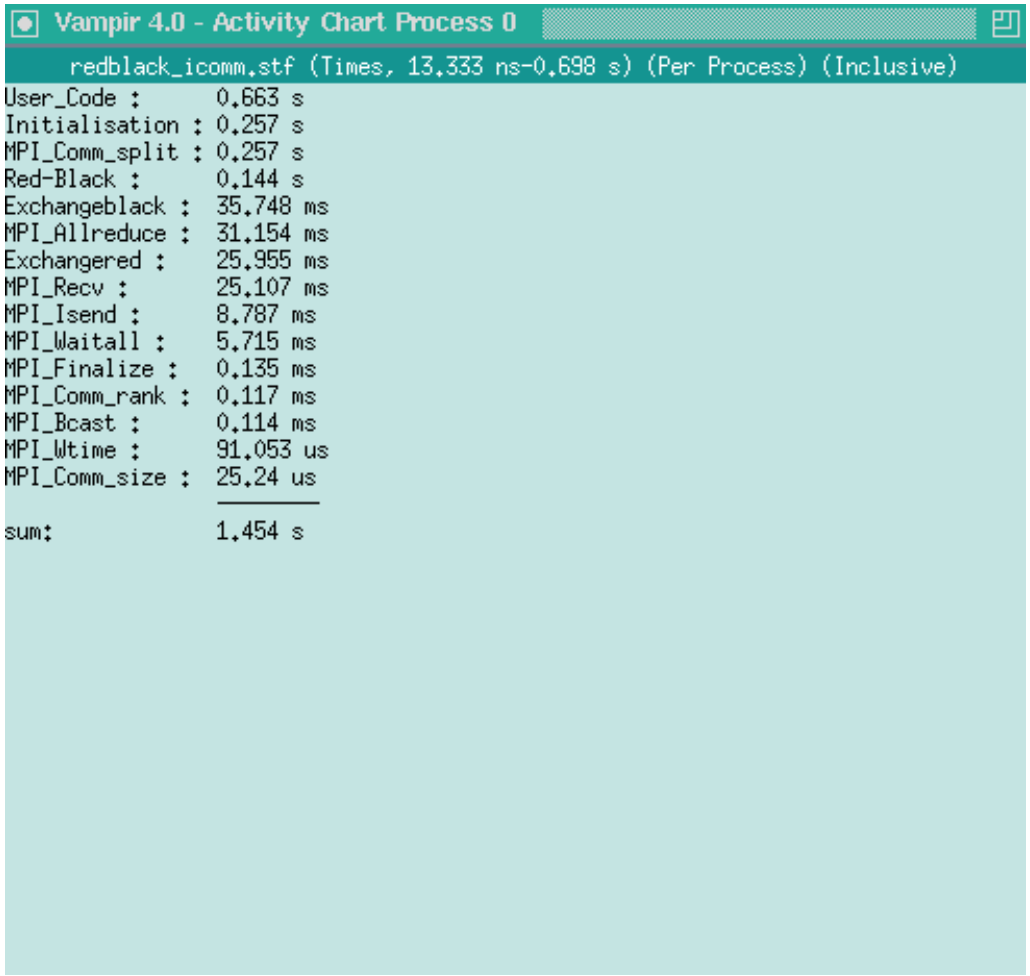
Process Activity Chart Displays



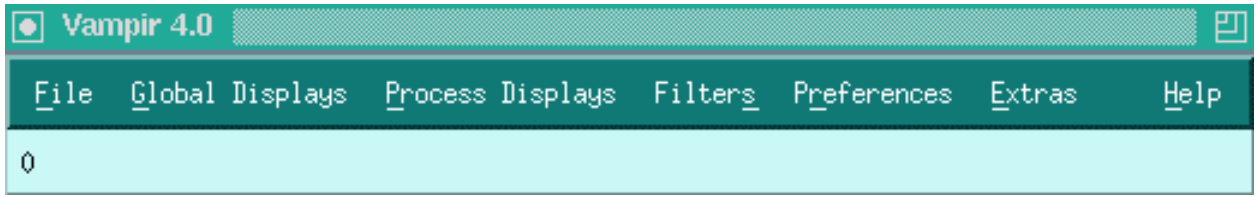
Process Activity Chart Displays



Process Activity Chart Displays



Displays



Timeline
Activity Chart
Summary Chart
Message Statistics
File I/O Statistics

Global Timeline Display

- Context menu is activated with a right mouse click inside any display window
- Zoom in by selecting start of desired region, left click held, drag mouse to end of desired region and release
- Can zoom in to unlimited depth
- Step out of zooms from context menu

Activity Charts

Default is pie chart, but can also use histograms or table mode

Can select different activities to be shown

Can hide some activities

Can change scale in histograms

Summary Charts

Shows total time spent on each activity

Can be sum of all processors or average for each processor

Similar context menu options as activity charts

Default display is horizontal histogram, but can also be vertical histogram, pie chart, or table

Communication Statistics

Shows matrix of comm statistics

- Can show total bytes, total msgs, avg msg size, longest, shortest, and transmission rates
- Can zoom into sub-matrices
- Can get length statistics
- Can filter messages by type (tag) and communicator

Tracefile Size

Often, the trace file from a fully instrumented code grows to an unmanageable size

- Can limit the problem size for analysis
- Can limit the number of iterations
- Can use the vampirtrace API to limit size
  - vttraceoff (): Disables tracing
  - vttraceon(): Re-enables tracing

Performance Analysis and Tuning

First, make sure there is available speedup in the MPI routines

- Use a profiling tool such as VAMPIR
- If the total time spent in MPI routines is a small fraction of total execution time, there is probably not much use tuning the message passing code
  - BEWARE: Profiling tools can miss compute cycles used due to non-blocking calls!

Performance Analysis and Tuning

If MPI routines account for a significant portion of your execution time:

- Try to identify communication hot-spots
  - Will changing the order of communication reduce the hotspot problem?
  - Will changing the data distribution reduce communication without increasing computation?
    - Sending more data is better than sending more messages

Performance Analysis and Tuning

- Are you using non-blocking calls?
  - Post sends/receives as soon as possible, but don't wait for their completion if there is still work you can do!
  - If you are waiting for long periods of time for completion of non-blocking sends, this may be an indication of small system buffers. Consider using buffered mode.

Performance Analysis and Tuning

- Are you sending lots of small messages?
  - Message passing has significant overhead (latency). Latency accounts for a large proportion of the message transmission time for small messages.
    - Consider marshaling values into larger messages if this is appropriate
    - If you are using derived datatypes, check if the MPI implementation handles these types efficiently
    - Consider using MPI_PACK where appropriate
      - dynamic data layouts or sender needs to send the receiver meta-data.

Performance Analysis and Tuning

- Use collective operations when appropriate
  - many collective operations use mechanisms such as broadcast trees to achieve better performance
- Is your computation to communication ratio too small?
  - You may be running on too many processors for the problem size

MPI_CHECK

Tool developed at the University of Iowa for debugging MPI programs written in free or fixed format Fortran 90 and Fortran 77

You can download your own free copy of the software and license at http://www.hpc.iastate.edu/MPI-CHECK.htm

MPI-CHECK does both compile-time and run-time error checking

Compile Time Error Checking

Checks for consistency in the data type of each argument

Checks the number of arguments

Checks the little used intent of each argument

Run-Time Error Checking

- Buffer data type inconisistency
  - This error is flagged if the Fortran data type of the send or receive buffer of an MPI send or receive call is inconsistent with the declared datatype in the MPI call
- Buffer out of bounds
  - This error is flagged if either the starting or ending address of a send or receive buffer is outside the declared bounds of the buffer
- Improper placement of MPI_Init or MPI_Finalize

Run-Time Error Checking

Illegal message length

Invalid MPI Rank

Actual or potential deadlock

- Any cycle of blocking send calls creates a potential for deadlock. While this deadlock may not be manifest on all machines, MPI-CHECK will detect if the potential for deadlock exists.

Using MPI-CHECK

Programs are compiled the same way as normal, except mpicheck is the first command on the command line:

f90 –o a.out –O3 main.f90 sub1.f90 sub2.f90 –lmpi

Becomes

mpicheck f90 –o a.out –O3 main.f90 sub1.f90 sub2.f90 –lmpi

Source files are required, rather than object files

Programs are ran just as without MPI-CHECK

Remarks

While MPI-CHECK does not flag all possible MPI errors, and it may flag some instances of correct usage as potential errors, it has been shown to be very useful in discovering many subtle, yet common, MPI programming errors. It is easy to use and adds little overhead to the execution times of programs.

More information on MPI-CHECK and MPI-CHECK2 (deadlock detection) can be found at:

http://www.hpc.iastate.edu/Papers/mpicheck/mpicheck1.htm

and

http://www.hpc.iastate.edu/Papers/mpicheck2/mpicheck2.htm