Lecture 20 – Parallel Performance and MPI

Prof. Brendan Kochunas

NERS/ENGR 570 - Methods and Practice of Scientific Computing (F20)

COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Outline

- Parallel Performance
- The fundamental concepts in MPI
- Point-to-point communication
- Collective communication

Learning Objectives: By the end of Today's Lecture you should be able to

- (*Skill*) Conducting a strong scaling study
- (*Skill*) Conduct a weak scaling study
- (*Skill*) Compute the parallel speedup and efficiency
- (*Knowledge*) explain the types of communication in MPI?

Parallel Performance

Example: OpenMP

Threads	Time	
1		
2		
4		
8		
16		
24		
32		

Parallel Performance Metrics

- Strong Scaling: fixes problem size and increases number of processors.
  - Provides insight into how finely grained an algorithm can be parallelized and how much parallel overhead there is relative to useful computation
- Weak Scaling: fixes problem size *per process* and increases number of processors.
  - Provides insight into whether the parallel overhead varies faster or slower than the amount of work as the problem size is increased.
- Speedup and Efficiency:
- Good efficiency does not necessarily mean you have fast code
  - It could mean you have terrible serial performance

$$S\left(P_{size}, N_p\right) \equiv \frac{T\left(P_{size}, 1\right)}{T\left(P_{size}, N_p\right)}$$

$$E_{strong}\left(P_{size}, N_p\right) \equiv \frac{T\left(P_{size}, 1\right)}{N_p \times T\left(P_{size}, N_p\right)}$$

$$E_{weak}\left(P_{size}, N_p\right) \equiv \frac{T\left(P_{size}, 1\right)}{T\left(P_{size} \times N_p, N_p\right)}$$

Parallel Execution Time Models

Moving from serial to parallel

- Serial Latency based model

$$T_{serial} = Ft_F + \alpha_1 L + \sum_{j=1}^{\kappa-1} \left(\alpha_{j+1} - \alpha_j\right) M_j + \left(\alpha_{mem} - \alpha_\kappa\right) M_\kappa$$

- Parallel Model

$$T_{parallel}\left(N_p\right) = \frac{T_{serial}}{N_p} + T_{overhead}\left(N_p\right)$$

- Difficult to develop exact expressions,
  - Alternatively measure realistic average values based on microbenchmarks.

Canonical Execution Time Models

- Distributed Memory Computing
  - Point-to-Point Communication Time

$$T_{comm} = \underbrace{\alpha_{network}}_{\text{Latency}} + \underbrace{\beta_{network}}_{\text{1/Bandwidth}} \underbrace{N}_{\text{Amount of data}}$$

  - Collective operations have their own (depends on algorithm implemented in library)

$$T_{\text{All\_reduce,small}} = \lceil \log p \rceil \left(\alpha_{network} + \beta_{network} \times N + \gamma \times N\right)$$

$$T_{\text{All\_reduce,large}} = 2 \log p \, \alpha_{network} + \frac{p-1}{p}\left(2\beta_{network} \times N + \underbrace{\gamma}_{\substack{\text{Time to perform reduce operation} \\ \text{(e.g. sum, max, multiply, etc.)}}} \times N\right)$$

Fundamentals of getting good parallel performance

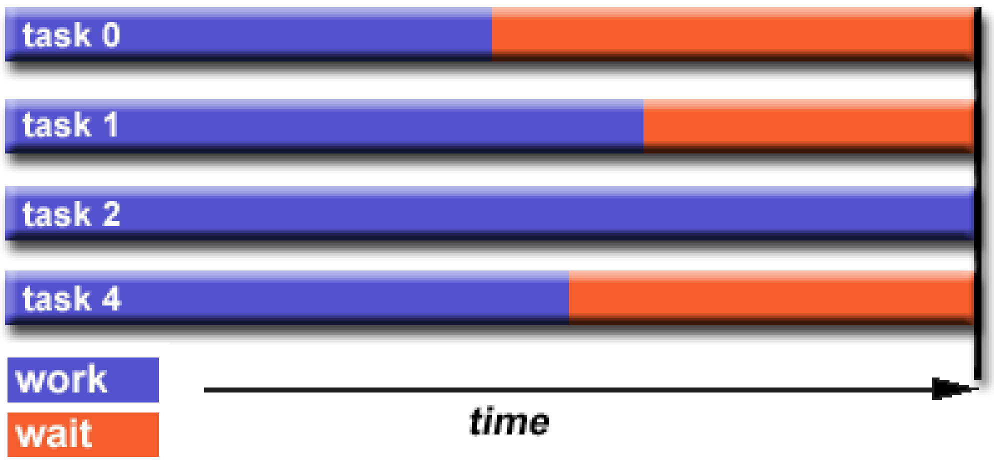
- Maximize amount of work that can be parallelized.
- Minimize overhead.
- Usually this means
  - Balance work loads among processors
  - Avoid synchronization
    - Especially for shared memory
  - use non-blocking communication
    - Primarily in distributed memory models
- Make sure the serial code is optimized.

$$T_{parallel}\left(N_p\right) = \underbrace{T_{non-parallel}}_{\text{Minimize}} + \overbrace{\frac{T_{serial}}{N_p}}^{\text{Assumes perfect load balance}} + \underbrace{T_{overhead}\left(N_p\right)}_{\text{Minimize}}$$

Load Balance & Idle Time

Problem

- Poor strong scaling
- Poor parallel efficiency
- Increase cores by factor of 2x, do not observe 2x speedup.



Solution

- Determine what the load balance/imbalance is
  - Need to assign a value of "work" to each subdomain.
  - What is the maximum to minimum workload for all domains.
- Change partitioning to improve load balancing
- Change parallel algorithm

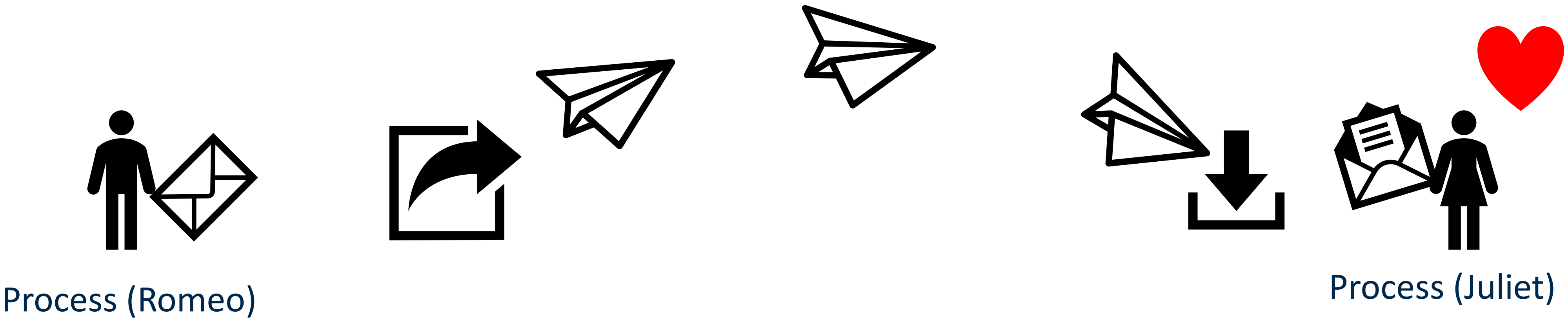
MPI Concepts & Basics

Motivation

- For a variety of reasons, scientific computing platforms have evolved into the HPC architectures of today.
  - These machines allow scientists to complement theory and experiment with simulation and advance our understanding.
  - As we saw in lecture 16 on Parallel programming models & algorithms, there are a variety of ways to *think* about how our algorithms can be implemented in parallel, but for them to be realized they must ultimately get implemented on a machine.
- Why Message Passing?
  - The message passing model is (relative to other models) **universal** and may be realized on a variety of computer architectures.
  - The message passing model has been found to be a useful and complete model in which to **express** parallel algorithms. It places key elements of expressing the parallel algorithm in the hands of the programmer, not the compiler.
  - Message passing allows for good **performance**. Explicit management of data with processors allows compilers to be able to do the best job of fully utilizing memory hierarchy and processor

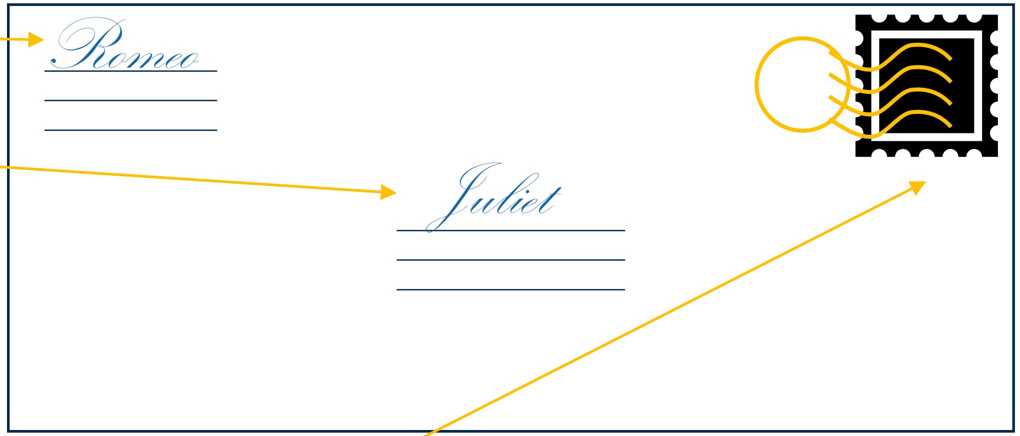
Message Passing Model

- "Postal Analogy"
  - Model assumes different processes executing simultaneously (Living humans) have separate memory spaces (brains) and message passing is cooperative
  - (e.g. man sends letter, woman receives letter)



MPI Concepts: Messages

- Sender
- Receiver
- Contents of message



- For MPI, there's also
  - messages need an identifier called a ***tag***
    (I get lots of letters from you Romeo, which one are you talking about?!)
  - ***Type*** of message (e.g. letter, flowers, candy)
  - ***Size*** of message (e.g. 10 pages, a dozen roses, a box of chocolates)

MPI Concepts: Communicators

- In postal analogy this would be like some notion of the post office/postal system
  - There is not a “good” analogy here.
- Suggestion: think about it like an application programmer
  - How can MPI keep track of which unique processes are a part of your execution?
    - These must be named/identified in some way
  - What if your application uses a library built on MPI?
    - There should exist a reliable way to separate messages you implement from ones that the library implements.
- Communicators solve the problem of organizing groups and contexts
  - Groups name processes
  - Contexts are like systems of post offices (think different countries, states, zip codes)
    - These facilitate the use of software libraries

The Fundamental MPI Routines: Send/Recv

```
MPI_Send(variable_address, size, datatype, destination, tag, communicator)
```

```
MPI_Recv(variable_address, max_size, datatype, source, tag, communicator, status)
```

status: needed for knowing what happened (e.g. did it work?)

MPI Program Basics (The original 6)

Routine	Purpose
MPI_Init	Initialize MPI
MPI_Comm_size	Find out how many processes there are
MPI_Comm_rank	Find out which process I am
MPI_Send	Send a message
MPI_Recv	Receive a message
MPI_Finalize	Terminate MPI

The MPI Library conventions

- Naming convention:
  - Everything starts with `MPI_`
  - Compile time constants appear in all caps (e.g. MPI_COMM_WORLD)
  - Routines named as:
    - `MPI_<Operation>` (e.g. MPI_Send, MPI_Barrier, etc.)
    - `MPI_<Class>_<action>_[<subset>]` (e.g. MPI_Comm_size, MPI_Group_split, MPI_Comm_get_errhandler)
  - Fortran interfaces include extra ierr argument
    - C Interface `ierr = MPI_<Class>_<action>(arg1,arg2,…,argN)`
    - Fortran interface `MPI_<class>_<action>(arg1,arg2,…,argN,ierr)`

Compiling and Running MPI Programs

Compiling

- MPI installs with "compiler wrappers"
  - These are simple programs that call your normal compilers with the extra options for compiling and linking against the MPI library.
- These are being standardized

Wrapper	Compiler
mpicc	C
mpicxx, mpic++, mpiCC	C++
mpifort, mpif77, mpif90	Fortran

Running

- `mpiexec [options] <executable>`
  - `-np` <number_of_processors>
  - `-f` <machinefile>
- Try to avoid
  - `mpirun` (deprecated)
  - `mpif77` and `mpif90` (deprecated in OpenMPI)
  - `mpiCC` (some file systems are not case sensitive)

Common MPI Distributions

- MPICH (ANL/UIUC)
  - This is **THE** reference implementation, often supports newest features first. Very high quality.
  - Does not support infiniband networks
  - Basis for many other implementations
- MVAPICH (OSU)
  - Derivative of MPICH supporting high speed networks
- OpenMPI
  - Competitor with MPICH, has good process control, slower on feature support
  - Generally has more bugs than MPICH
- Vendor implementations
  - Built on MPICH but swap routines/functions for code specific for their machines
  - Intel, HP, Cray, SGI, IBM, and probably others.

Point-to-Point Communication

MPI Point-to-Point Communication Routines

- Point-to-Point communication involves 2 processors.
- Basic calls:

```
MPI_Send(variable_address,
         size,
         datatype,
         destination,
         tag, communicator)

MPI_Recv(variable_address,
         max_size,
         datatype,
         source,
         tag, communicator, status)
```

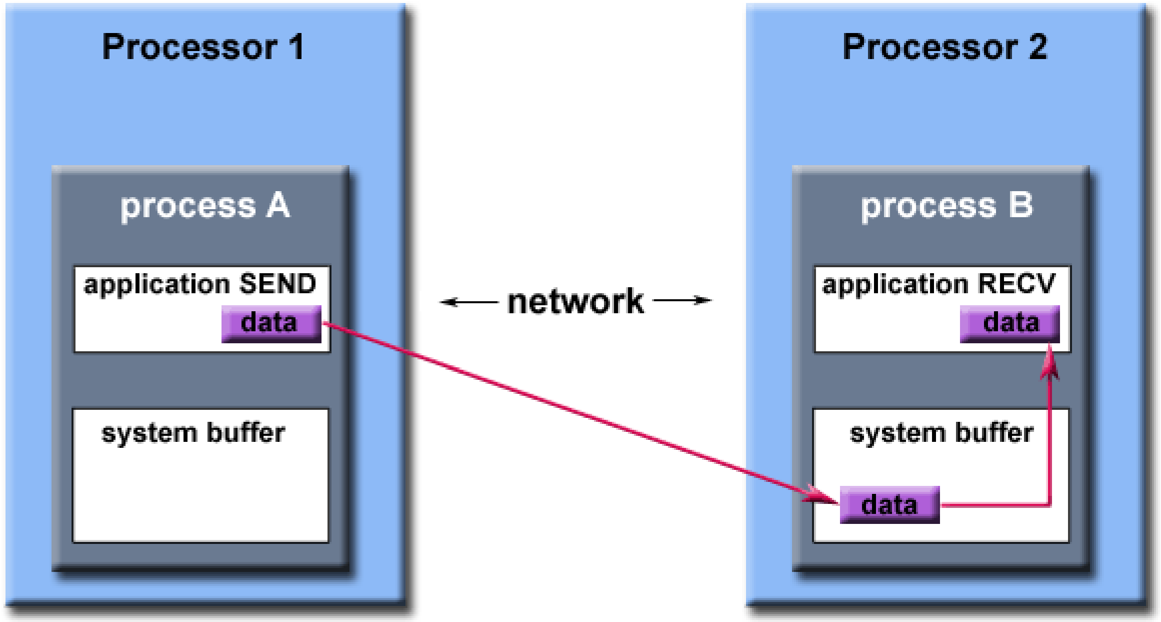
- Many variations (communication modes):
  - **Standard** mode – a send will not block even if a receive for that message has not occurred (except for lack of resources, e.g. out of buffer space at sender or receiver)
  - **Buffered** mode (MPI_Bsend) – same as standard mode, except return is always immediate, i.e., returns an error code as opposed to waiting for resources)
  - **Synchronous** mode (MPI_Ssend) – will only return when matching receive has started. No extra buffer copy needed, but can't do any computation while waiting.
  - **Ready** mode (MPI_Rsend) – will only work if matching receive is already waiting. Best performance, but can fail badly if not synchronized.
  - **Immediate** mode (MPI_Isend, etc.) – starts a standard-mode send but returns immediately. No extra buffer copy needed, but the sender should not modify any part of the send buffer until the send completes.
  - Also a combined **sendrecv**

MPI Point-to-point communication

- So many choices, which one is best?
  - The standard `send` and `recv` are good for learning MPI, but are generally not used in production application codes.
  - Buffered `send` and `recv` require more effort on the part of the application programmer to manage the buffer.
  - Synchronous send and recv are often same as standard send and recv
- Some form of non-blocking `send` and `recv` is often best for performance.
  - `MPI_Isend` and `MPI_Irecv`
  - Does require additional checking for completion
  - There are limits to the number of simultaneous messages

Message Buffers

- An MPI implementation may (not the MPI standard) decide what happens to data in these types of cases.
  - Typically, a **system buffer** area is reserved to hold data in transit.
- System buffer space is:
  - Opaque to the programmer and managed entirely by the MPI library
  - A finite resource that can be easy to exhaust
  - Often mysterious and not well documented
  - Able to exist on the sending side, the receiving side, or both
  - Something that may improve program performance because it allows send - receive operations to be asynchronous.



Path of a message buffered at the receiving process

Collective Communication

MPI Collectives (1)

- These involve all MPI processes in a *communicator*
- Collectives can always be implemented with point-to-point routines
  - But it is often better to use the routines provided by MPI
- Common collective operations include:
  - Broadcast
  - Reduce
  - Scatter
  - Gather
  - Scan
  - Alltoall

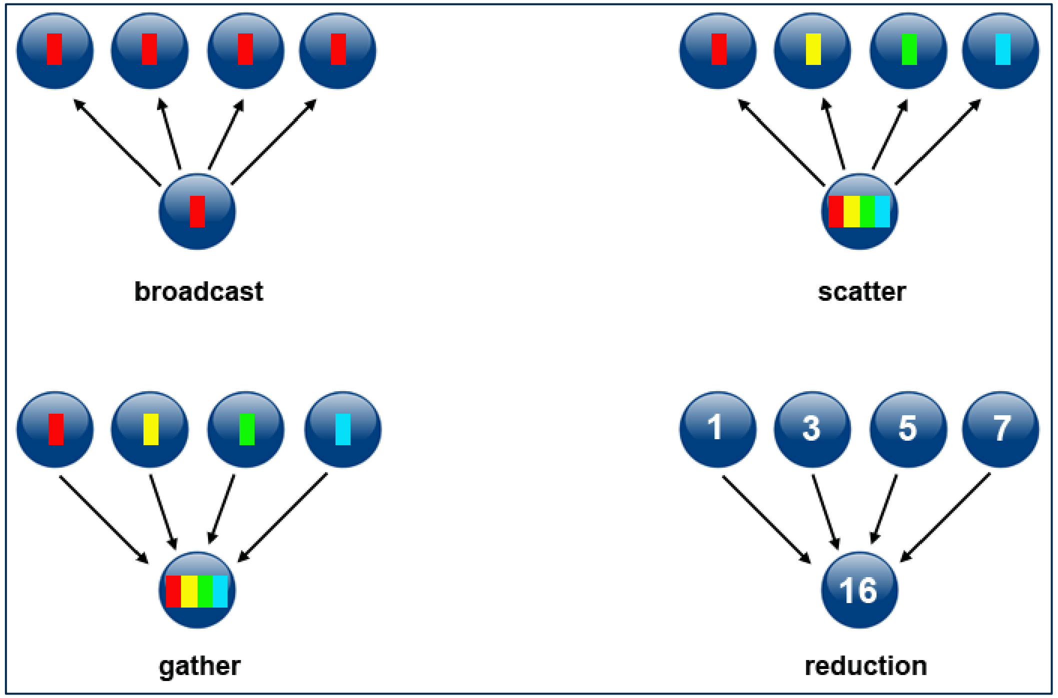


Figure from: https://computing.llnl.gov/tutorials/parallel_comp/

MPI Collectives (2)

Notable Variations

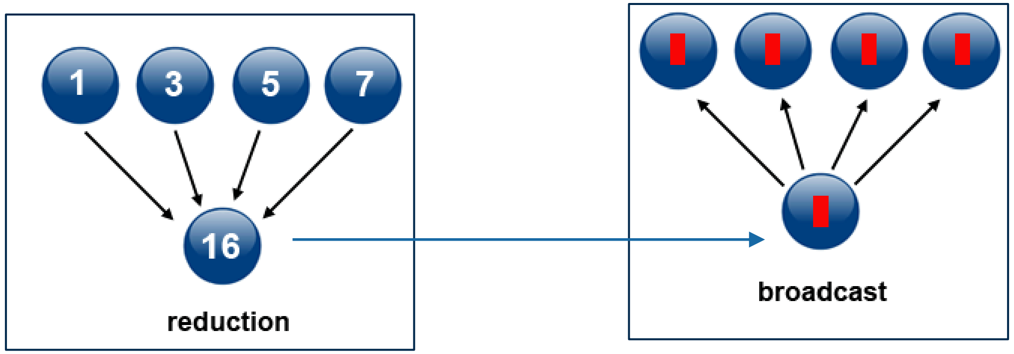
- The "`v`" suffix
  - Stands for vector
  - Means the *<u>size of data may be different</u>* for different processors
  - Gatherv & Scatterv, Alltoallv
- The "`All`" prefix
  - Means the *<u>result of the operation is the same for all processors</u>* in communicator
  - Allreduce & Allgather

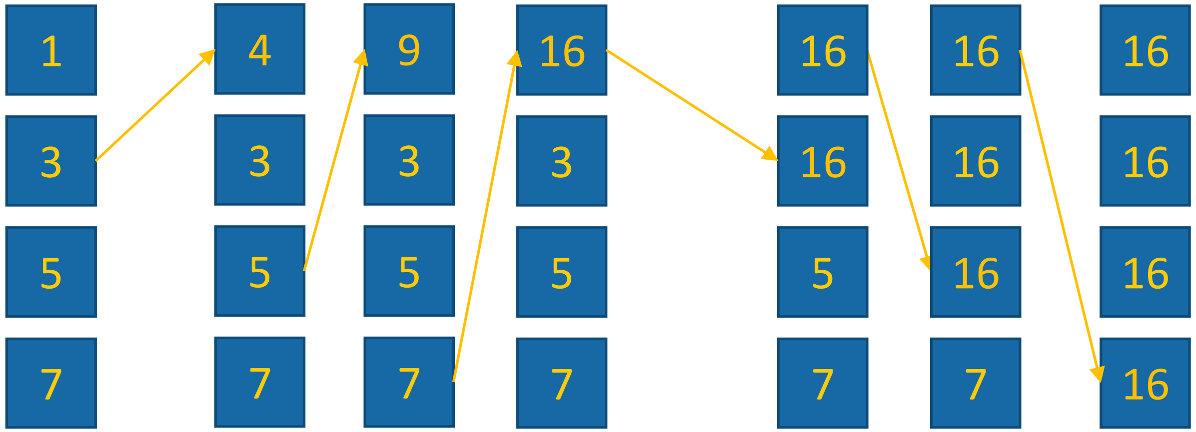
Types of reduction operations

- Arithmetic
  - MPI_SUM
  - MPI_PROD
- Relation Operators (Mins & Maxes)
  - MPI_MAX
  - MPI_MIN
  - MPI_MAXLOC
  - MPI_MINLOC
- Logical Operators
  - MPI_LAND
  - MPI_LOR
  - MPI_LXOR
- Bit-wise operators also supported

Example: MPI_Allreduce Algorithm

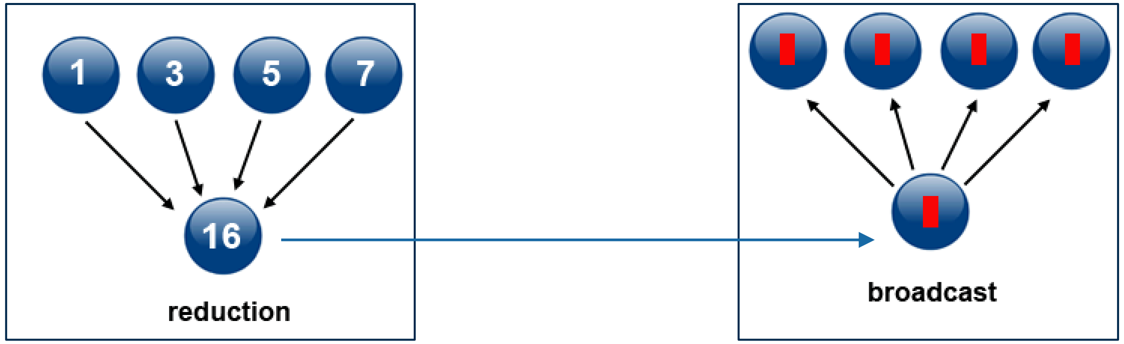
- Reduce + broadcast
- Reduce performed sequentially
  - P-1 steps
- Broadcast performed sequentially
  - Also P-1 steps
- Total of 6 steps

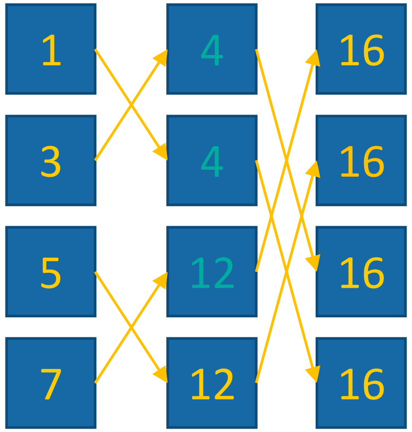




Example:Better Allreduce

- Use a binomial tree
  - Completed in $\lceil \log p \rceil$ steps
- Scales much better to higher number of processors





Even More Advanced Allreduce

- What about long messages?
  - Reduce_scatter + Allgather
- Different algorithms perform better under certain conditions

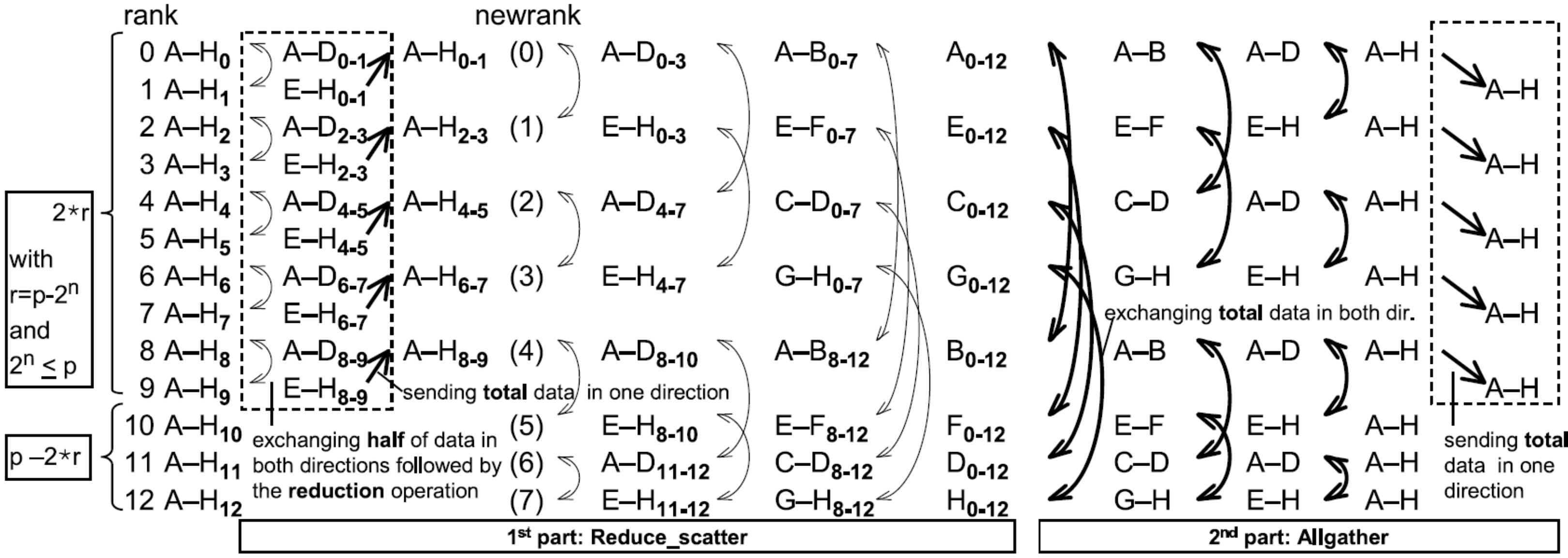


Figure 12: Allreduce using the recursive halving and doubling algorithm. The intermediate results after each communication step, including the reduction operation in the reduce-scatter phase, are shown. The dotted frames show the additional overhead caused by a non-power-of-two number of processes.

Source: http://www.mcs.anl.gov/~thakur/papers/ijhpca-coll.pdf

Summary of Collectives

- Provided as a convenience to the programmer
  - Collectives perform “common” operations that arise in programming
  - Often implemented with more complex and higher performing algorithms
    - Than what a beginner would implement.
- They represent a synchronization point in the program
- Always, always, always involves all processors *within communicator*
  - Otherwise, it causes a deadlock