MPI Collectives I: Reductions and broadcast

Calculating Pi with a broadcast and reduction
What does “Collective” mean

• Everyone within the communicator (named in the call) must make that call before the call can be effected

• Essentially, a collective call requires coordination among all the processes of a communicator
Some Basic MPI Collective Calls

• **MPI_Barrier(MPI_Comm comm)**
  • Blocks the caller until all processes have entered the call

• **MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)**
  • Broadcasts a message from rank ‘root’ to all processes of the group
  • It is called by all members of group using the same arguments

• **MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)**

• **MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)**
int main(int argc, char **argv) {
    int myRank, numPes;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numPes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    int count, i, numTrials, myTrials;
    if(myRank == 0) {
        scanf("%d", &numTrials);
        myTrials = numTrials / numPes;
        numTrials = myTrials * numPes; // takes care of rounding
    }
    MPI_Bcast(&myTrials, 1, MPI_INT, 0, MPI_COMM_WORLD);

    count = 0;
    srand(myRank);
    double x, y, pi;
for (i=0; i<myTrials; i++) {
    x = (double) random()/RAND_MAX;
    y = (double) random()/RAND_MAX;
    if (x*x + y*y < 1.0)
        count++;
}

int globalCount = 0;

MPI_Allreduce(&count, &globalCount, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

pi = (double)(4 * globalCount) / (myTrials * numPes);

printf("[%d] pi = %f
", myRank, pi);
MPI_Finalize();
return 0;
} /* end function main */
MPI_Reduce

• Often, you want the result of a reduction only on one processors
• E.g. In our Pi program, we may want to calculate pi on the process with rank 0,
  • and other processes to stay silent (i.e. output nothing)
• The system can use a more efficient algorithm if it does not need to convey the result to every process
  • You tell the system so by calling MPI_Reduce instead of MPI_Allreduce
  • Needs one extra parameter to state which process should get the result
for (i=0; i<myTrials; i++) {
    x = (double) random()/RAND_MAX;
    y = (double) random()/RAND_MAX;
    if (x*x + y*y < 1.0)
        count++;
}

int globalCount = 0;

MPI_Reduce(&count, &globalCount, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

pi = (double)(4 * globalCount) / (myTrials * numPes);

printf("[%d] pi = %f\n", myRank, pi);
MPI_Finalize();
return 0;
} /* end function main */
for (i=0; i<myTrials; i++) {
    x = (double) random()/RAND_MAX;
    y = (double) random()/RAND_MAX;
    if (x*x + y*y < 1.0)
        count++;
}

int globalCount = 0;

MPI_Reduce(&count, &globalCount, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if (myRank == 0) {
    pi = (double)(4 * globalCount) / (myTrials * numPes);
    printf("[%d] pi = %f\n", myRank, pi);
}

MPI_Finalize();
return 0; /* end function main */
More Collectives

Gather, Scatter, Allgather, All-to-All, etc.

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Based on content from the MPI standard document
MPI Broadcast

• int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

When Syntax is omitted, this is where you are expected to go to look it up

MPI Standard, 2015
MPI_Gather and MPI_Scatter
MPI Allgather

int MPI_Allgather(const void* sbuf, int scount,
                   MPI_Datatype stype, void* rbuf, int rcount,
                   MPI_Datatype rtype, MPI_Comm comm)
int MPI_Alltoall(const void* sbuf, int scount,
                 MPI_Datatype stype, void* rbuf, int rcount,
                 MPI_Datatype rtype, MPI_Comm comm)
Variable sizes per process

• For some operations, e.g. MPI_Gather, , the amount of data contributed by each process is assumed to be constant

• What if the data for each process is of different size?
  • Variants with “v” at the end handle that case with additional size parameters
  • MPI_Gatherv, MPI_Scatterv, MPI_Alltoallv, ..
More collectives

• We omitted a few collective operations and modes that may be useful
• Reduce_scatter: reduce a vector and scatter portions of it to processes
  • Equivalent of reduce followed by scatter
  • But may have faster implementation
• Nonblocking modes
  • Added recently to the MPI standard
  • Indicated by letter I as with Irecv
  • E.g. MPI_Igather
Do-it-yourself Collective operations

• May be implemented with MPI point-to-point

• Implementations can optimize for small (latency), large (bandwidth), or both.

• Generality of some MPI collective operations can limit performance
  • Routines must assume datatypes are general and non-contiguous
  • Time/memory tradeoffs (for internal temporary buffers)

• So, occasionally, but hopefully rarely, you will want to use custom implementations
Some of the figures and content taken from material from the MPI forum

MPI: A Message-Passing Interface Standard
Version 3.1
Message Passing Interface Forum
June 4, 2015

© 1993, 1994, 1995, 1996, 1997, 2008, 2009, 2012, 2015 University of Tennessee, Knoxville Tennessee. Permission to copy without fee all or part of this material is granted, provided the University of Tennessee copyright notice and the title of this document appear, and notice is given that copying is by permission of the University of Tennessee.
CS484

MPI: sub-communicators

Fox’s Algorithm for Matrix Multiplication

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

© 2018 L. V. Kale at the University of Illinois Urbana
Subcommunicators

• A communicator represents a set of processes, and each member process has a rank in that communicator.

• So far, we used only one communicator, MPI_COMM_WORLD, which has all the processes of the program in it.

• Next, we introduce the ideas of:
  • Sub-communicators, created via mpi_Comm_split, by splitting an existing communicator.
  • Cartesian communicators: “reshape” (reorganize) an existing set of processes in an existing communicator into a multi-dimensional structure (e.g. rectangular): mpi_Cart_create.
Fox’s Algorithm for Matrix Multiplication

• Assume each matrix is decomposed into square tiles among a square number of processors

• It will be convenient to
  • Broadcast submatrices of A along a process row
  • Pass submatrices of B along process column

• Create a communicator for each row and column

![Diagram of matrix multiplication](image)
int main(int argc, char **argv) {
    int myRank, numPes;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numPes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    int dimSize[2] = {NUM_ROWS, NUM_COLS};
    int periodic[2] = {0, 0};
    int myCoords[2];

    MPI_Comm comm2D;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dimSize, periodic, 0,&comm2D);

    int myRow, myCol;
    MPI_Cart_coords(comm2D, myRank, 2, myCoords);
    myRow = myCoords[0]; myCol = myCoords[1];
MPI_Comm commR, commC;
MPI_Comm_split(comm2D, myRow, myCol, &commR);
MPI_Comm_split(comm2D, myCol, myRow, &commC);

int A[DIMX*DIMY], B[DIMX*DIMY], C[DIMX*DIMY];
int tileA[DIMX*DIMY], tileB[DIMX*DIMY];
int tmpA[DIMX*SIZEY], tmpB[SIZEX*DIMY];

MPI_Barrier(MPI_COMM_WORLD);
for(i=0; i<NUM_COLS; i++) {
    if((myRank%NUM_COLS) == i) {
        for(j=0; j<DIMX; j++)
            for(k=0; k<DIMY; k++)
                tileA[j*DIMY + k] = A[j*DIMY+k];
    }
    MPI_Bcast(tileA, DIMX*DIMY, MPI_INT, i, commR);
    for(j=0; j<DIMX; j++)
        for(k=0; k<DIMY; k++)
            tmpA[i*DIMY + j*SIZEY + k] = tileA[j*DIMY+k];
    MPI_Barrier(MPI_COMM_WORLD);
}
for(i=0; i<NUM_ROWS; i++) {
    if((myRank/NUM_COLS) == i) {
        for(j=0; j<DIMX; j++)
            for(k=0; k<DIMY; k++)
                tileB[j*DIMY + k] = B[j*DIMY+k];
    }
    MPI_Bcast(tileB, DIMX*DIMY, MPI_INT, i, commC);
    for(j=0; j<DIMX; j++)
        for(k=0; k<DIMY; k++)
            tmpB[i*DIMX*DIMY + j*DIMY + k] = tileB[j*DIMY+k];
    MPI_Barrier(MPI_COMM_WORLD);
}

for(i=0; i<DIMX; i++)
    for(j=0; j<SIZEY /* SIZEX */; j++)
        for(k=0; k<DIMY; k++)
            C[i*DIMY + k] += tmpA[i*SIZEY + j] * tmpB[j*DIMY + k];

MPI_Finalize();
return 0;
} /* end function main */
Parallel Algorithms in MPI
Prefix Sum
Prefix Sum

• Given array A[0..N-1], produce B[N], such that B[k] is the sum of all elements of A up to A[k]

B[0] = A[0];
for (i=1; i<N; i++)
    B[i] = B[i-1] + A[i];
Prefix Sum Problem

• Given array A[0..N-1], produce B[N], such that B[k] is the sum of all elements of A up to A[k]

\[
A = \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
3 & 1 & 4 & 2 & 5 & 2 & 7 & 4 \\
\end{array}
\]

\[
B = \begin{array}{cccccccc}
3 & 4 & 8 & 10 & 15 & 17 & 24 & 28 \\
\end{array}
\]

Prefix Sum: A good Sequential Algorithm

• Data dependency from iteration to iteration
  • How can this be parallelized at all?

\[
\begin{align*}
B[0] &= A[0]; \\
\text{for } (i=1; i<N; i++) & \ \\
B[i] &= B[i-1] + A[i];
\end{align*}
\]

• It looks like the problem is inherently sequential, but theoreticians came up with a beautiful algorithm called recursive doubling or just parallel prefix
Parallel Prefix: Recursive Doubling

N Data Items
P Processors
N=P

Log P Phases
P additions in each phase
P log P operations
Completes in $O(\log P)$ time
Parallel Prefix: Engineering

• Issue : $N >> P$

• Recursive doubling : Naïve implementation
  • Operation count: $\log(N) \cdot N$

• A better implementation: well-engineered:
  • Take blocking of data into account
  • Each processor calculate its sum, then
    • Participates in a parallel algorithm (with $P$ numbers)
    • to get sum to its left, and then adds to all its elements
  • $N/P + \log(P) + N/P$
    • Only doubling of operation Count

• What did we do?
  • Same algorithm, better parallelization/engineering

The log($P$) step could be done sequentially in a shared memory program, but not in an MPI program
Let's focus on $N=\equiv P$

- I.e. every rank has just one number
- And let's implement the algorithm in MPI
Pseudocode for Parallel Prefix with MPI

\[ i = \text{my rank} \]
\[ p = \text{total number of processes} \]

\[ \text{myValue} = i; \quad // \text{for testing it, this is a simple value to have} \]
\[ \text{distance} = 1; \quad \text{value} = \text{myValue}; \]

while (distance < p) {
    if ( (i+distance) < p) \quad // i.e. the destination exists
        send value to i+distance
    if (i-distance >= 0) {
        recv X from i-distance
        value += X;
    }
    distance = distance*2;
}

Print my rank, myValue, value
```c
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i, p, distance;
    double value, x;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&i);
    MPI_Comm_size(MPI_COMM_WORLD,&p);

    distance = 1;
    value = i;

    while(distance < p) {
        if((i+distance)<p)
            MPI_Send(&value,1,MPI_DOUBLE, i+distance,0,MPI_COMM_WORLD);
        if((i-distance)>=0) {
            MPI_Recv(&x,1,MPI_DOUBLE, i-distance,0,MPI_COMM_WORLD,&status);
            value+=x;  }
        distance*=2;
    }
    printf("%f\n",value);  MPI_Finalize(); return 0;
}
```

Implementation of prefix sum using MPI point-to-point communication
MPI Supports Parallel Prefix as a Collective

- int MPI_Scan(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

- The count is used to do specify an array for the buffers

- The scan operation is performed on each element of the array independently
  - E.g. recvbuf[23] on process 4 will contain the sum of 23\textsuperscript{rd} elements (i.e. sendbuf[23]) from processes 0, 1, 2, 3, and 4.
  - Instead of sum, it can be any op supported by reductions
Uses of parallel prefix (also called scan)

• *Scan* (parallel prefix) is very useful operation in many situations

• Consider the following situation:
  • Each processor $i$ has $K_i$ numbers
  • They are sorted within and across processors
  • But they are unequally distributed
  • How would you balance them, without losing the sorted-ness?
Average = 5

Sum of Previous Items obtained using Prefix Sum
Algorithmic Overhead

- Sometimes, we have to use an algorithm with higher operation count in order to parallelize an algorithm
  - Either the best sequential algorithm doesn’t parallelize at all
  - Or, it doesn’t parallelize well (e.g. not scalable)

- What to do?
  - Choose algorithmic variants that minimize overhead
  - Use two level algorithms

- Examples:
  - Parallel Prefix (Scan)
  - Game Tree Search
Parallelization Overhead: Advice

• Explore alternative algorithms
  • Unless the algorithmic overhead is inevitable!
• Don’t take algorithms that say “We use $f(N)$ processors to solve a problem of size $N$” as they are.
  • Use Clyde Kruskal’s metric:
    • Theoretical complexity must be expressed in terms of $N$ data items, $P$ processors
  • Reformulate accordingly
Cost Model
A simple cost model for message passing

• Messaging cost is $\alpha + n\beta$, where
  • $\alpha$: per message cost
  • $\beta$: Per byte cost

• For the sake of intuition:
  • Assume $\alpha$ is 1,000 times larger than $\beta$
  • Typical: per message cost may be 1 microsecond on a supercomputer, and tens of microseconds on commodity clusters,
    • Per byte cost may be 1 ns (corresponds to 1 Gbyte/s, not Gbit/s, bandwidth)

• What do we mean by “messaging cost”
  • Latency
  • Overhead to Processor
Communication Cost Model

• This model ignores several factors:
  • It assumes there is no contention in the network
  • It assumes latency is independent of distance
    • Which is mostly true, in no-contention scenario
  • We will revisit these issues when we study network topologies
    • But note that for first-order analysis, the $\alpha + n*\beta$ model is good enough
  • Also, it ignores packetization and per-packet overheads
Communication co-processor and CPU overhead

• CPU overhead: how much time is taken away from the CPU to deal with communication
  • Function calls
  • Data copying, if needed
  • Setup / interaction with the network
  • In MPI: tag matching

• Using a co-processor (a modern NIC will do) ”off-loads” some of the communication work from the CPU
  • Especially explicit data copying, in many cases

• For this reason, it is useful to separate overhead and latency
Each component has a per-message cost, and per byte cost

Each cost, for a $n$-byte message

\[ = \alpha + n \beta \]

Important metrics:
- Overhead at processor, co-processor
- Network latency
- Network bandwidth consumed
  - Number of hops traversed
Communication Basics

• Message Latency: time between the application sending the message and receiving it on the other processor

• Send overhead: time for which the sending processor was “occupied” with the message

• Receive overhead: the time for which the receiving processor was “occupied” with the message

• Network latency

• Separating overhead in analysis is useful only when you are using the time until communication completes for some computation, or for setting up another communication
  • Example: consider cost of sending 10 messages to 10 distinct processors
    • The overheads are serialized on the CPU, the latencies are overlapped

• Except when we state otherwise, we will use the simple $\alpha + n*\beta$ model without separating overhead

L.V.Kale
Overall Cost Model

• Execution time (aka completion time) can be modeled, for many applications, as communication cost
  • $T = T_{\text{comp}} + T_{\text{comm}}$

• This assumes
  • All processes are doing the same work
  • There is no overlap of communication and computation
  • With Overlap, $T = T_{\text{comp}} + T_{\text{comm}} - T_{\text{overlap}}$

• Do this for
  • Each processor (if they are doing the same thing)
  • Worst loaded processor, if the loads are imbalanced, or
  • longest chain (critical path) if that dominates

• Typically, to get the completion time expression
Other models: LogP, logGP, etc.

• LogP is an acronym: L: Latency, o: overhead, g: gap, p: processors
  • g: time period for injecting a short fixed size message
    • Maybe accounts for bandwidth, or contention or injection rate
  • Model is meant for theoretical analysis of algorithms with short fixed-size messages

• LogGP: generalized to arbitrary size messages
  • Starts resembling $\alpha + n*\beta$ model, as long as we take cognizance of overhead

• My advice:
  • Write (i.e. model) completion time expression, phase-by-phase if possible, with $\alpha + n*\beta$ model,
    • keeping in mind the characteristics of algorithm (critical path? Load balance? Overlap?)
  • Always measure, and compare with model
Cost Model: Examples

Stencil, Gauss-Seidell

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Sequential Pseudocode:

```java
while (maxError > Threshold) {
    Re-apply Boundary conditions
    maxError = 0;
    for i = 0 to N-1 {
        for j = 0 to N-1 {
            if (|B[i,j]- A[i,j]| > maxError)
                maxError = |B[i,j]- A[i,j]|
        }
    }
    swap B and A
}
```

Decomposition by:
- Row
- Blocks
- Or Column
Performance Estimate

• Computation, $T_{\text{comp}}$, is $t_c \times N \times N/p$, where $t_c$ is the computation cost per cell
  • For either decomposition

• Communication cost: $T_{\text{comm}}$
  • For Row decomposition: 2 messages of size $N$ words (of 8 bytes each)
    • $2(\alpha + 8N\beta)$
  • For tile decomposition: 4 messages of size $N/\sqrt{p}$ words (of 8 bytes each)
    • $4(\alpha + 8N\beta/(\sqrt{p}))$
  • Which one is better? Compare for specific values of $N$, $p$, $\alpha$, and $\beta$
Gauss-Seidel Relaxation

Sequential Pseudocode:

While (maxError > Threshold) {
    Re-apply Boundary conditions
    maxError = 0;
    for i = 0 to N-1 {
        for j = 0 to N-1 {
            old = A[i, j]
                           + A[i+1, j] + A[i-1, j]) ;
            if (|A[i, j]-old| > maxError)
                maxError = |A[i, j]-old|
        }
    }
}
How Do We Parallelize Gauss-Seidel?

• Visualize the flow of values
• Not the control flow:
  • That goes row-by-row
• Flow of dependences: which values depend on which values?
• Does that give us a clue on how to parallelize?
Parallelizing Gauss-Seidel

• Some ideas
  • Row decomposition, with pipelining

• Square over-decomposition
  • Assign many squares to a processor (essentially same?)
Row decomposition with pipelining

# Columns = N/W
# Rows = P

# Of Phases
N/W + (P-1)
Row decomposition, with pipelining

<table>
<thead>
<tr>
<th>Time</th>
<th>Number of Procs Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
</tr>
<tr>
<td>(\frac{N}{W})</td>
<td>(P)</td>
</tr>
<tr>
<td>(\frac{N + P - 1}{W})</td>
<td>(P)</td>
</tr>
</tbody>
</table>

L.V.Kale
Apply the cost model to Gauss-Seidel

• Apply the cost model to Gauss-Seidel, and then find the optimal w

• Completion time:
  • Sum of 3 phases: rising, plateau, falling phase
  • Communication in each phase is 1 message of W words:
One-sided Communication
One-sided Communication

- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or write to this memory
Two-sided Communication Example
One-sided Communication Example
Comparing One-sided and Two-sided Programming

Even the sending process is delayed

Delay in process 1 does not affect process 0

L.V. Kale
Window creation models

• Four models exist
  • MPI_WIN_CREATE
    • You already have an allocated buffer that you would like to make remotely accessible
  • MPI_WIN_ALLOCATE
    • You want to create a buffer and directly make it remotely accessible
  • MPI_WIN_CREATE_DYNAMIC
    • You don’t have a buffer yet, but will have one in the future
  • MPI_WIN_ALLOCATE_SHARED
    • You want multiple processes on the same node share a buffer
    • We will not cover this model today
MPI_WIN_CREATE

• Expose a region of memory in an RMA window
  • Only data exposed in a window can be accessed with RMA ops.

• Arguments:
  • base - pointer to local data to expose
  • size - size of local data in bytes (nonnegative integer)
  • disp_unit - local unit size for displacements, in bytes (positive integer)
  • info - info argument (handle)
  • comm - communicator (handle)

```c
int MPI_Win_create(void *base, MPI_Aint size,
                    int disp_unit, MPI_Info info,
                    MPI_Comm comm, MPI_Win *win)
```
int main(int argc, char ** argv)
{
    int *a;   MPI_Win win;

    MPI_Init(&argc, &argv);

    /* create private memory */
    a = (void *) malloc(1000 * sizeof(int));
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* collectively declare memory as remotely accessible */
    MPI_Win_create(a, 1000*sizeof(int), sizeof(int),
                      MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    /* Array ‘a’ is now accessibly by all processes in
     * MPI_COMM_WORLD */

    MPI_Win_free(&win);
    free(a);
    MPI_Finalize(); return 0;
}
Data movement: \textit{Get}

\texttt{MPI\_Get(}
\begin{itemize}
\item \texttt{origin\_addr, origin\_count, origin\_datatype, target\_rank, target\_disp, target\_count, target\_datatype, win)}
\end{itemize}
\texttt{)}

• Move data \texttt{to} origin, \texttt{from} target

• Separate data description triples for \texttt{origin} and \texttt{target}

Other preparatory calls are needed before data movement calls are made
See \texttt{Put\_and\_get}
Data movement: *Put*

MPI_Put(
    origin_addr, origin_count, origin_datatype,
    target_rank,
    target_disp, target_count, target_datatype,
    win)

- Move data **from** origin, **to** target
- Same arguments as MPI_Get

Other preparatory calls are needed before data movement calls are made
See [Put and get](#)
Data aggregation: Accumulate

• Like MPI_Put, but applies an MPI_Op instead
  • Predefined ops only, no user-defined!

• Result ends up at target buffer

• Different data layouts between target/origin OK, basic type elements must match

• Put-like behavior with MPI_REPLACE (implements $f(a,b)=b$)
  • Atomic PUT

Other preparatory calls are needed before data movement calls are made
See Put_and_get

L.V.Kale
Additional Atomic Operations

• Compare-and-swap
  • Compare the target value with an input value; if they are the same, replace the target with some other value
  • Useful for linked list creations – if next pointer is NULL, do something

• Fetch-and-Op
  • Fetch old value and perform Op at the target with provided value
  • E.g. Add 1 to the target, and get me the old value
Fence: Active Target Synchronization

- Collective synchronization model
- Starts and ends access and exposure epochs on all processes in the window
- All processes in group of “win” do an MPI_WIN_FENCE to open an epoch
- Everyone can issue PUT/GET operations to read/write data
- Everyone does an MPI_WIN_FENCE to close the epoch
- All operations complete at the second fence synchronization

```
MPI_Win_fence(int assert, MPI_Win win)
```
slides from a tutorial by
William Gropp, Pavan Balaji, Torsten Hoefler, and
Martin Schulz
One-sided Communication: example
Balancing Sorted Data, revisited
Balancing Sorted Data (revisited)

• Consider the following situation:
  • Each processor $i$ has $K_i$ numbers
  • They are sorted within and across processors
  • But they are unequally distributed
  • How would you balance them, without losing the sorted-ness?
Exercise: Implement this with two-sided communication..

- It is hard because a process doesn’t know how many messages to receive.
- You can solve that by doing a reduce-scatter (or all-reduce, but that’s overkill) on a vector of count of messages destined for processor i.
Solution using one-sided communication: basic idea

1. Identify where exactly my data goes
2. **Put** it there
3. Bracket this with fences to make sure it's complete
Sometimes you may have to send to just one process (or 0, if destination process $i$ is same as my rank).

Precautions: if $N$ is not a evenly divisible by $P$:
1. Take ceiling($N/P$) to make sure you don’t overload last processor
2. If $N/P$ is smaller than $P$, devise a more careful scheme to avoid leaving empty processors at the end
int myStartGlobal;
int *sorted, *data;
MPI_Win win;

MPI_Exscan(&myDataCount, &myStartGlobal, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
int myEnd = myStartGlobal + myDataCount - 1;

MPI_Win_allocate(dataPerProc*sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &sorted, &win);

MPI_Win_fence(MPI_MODE_NOPRECEDE, win);  // fence - there are no epochs before this
int next = myStartGlobal;
while (next <= myEnd) {
    int dest = next/dataPerProc;
    int displacement = next%dataPerProc;
    int size = std::min( dataPerProc*(dest+1), myEnd+1) - next;
    MPI_Put(&data[next-myStartGlobal], size, MPI_INT, dest, displacement, size, MPI_INT, win);
    next += size;
}
MPI_Win_fence(0, win);  // Mark Epoch done.. After this collective call, data will be available in sorted array
... use sorted and balanced data ...
MPI_Win_fence(MPI_MODE_NOSUCCEED, win); // fence to indicate no more epochs