Programming Distributed Memory Machines in MPI and UPC

Kenjiro Taura

University of Tokyo
Distributed memory machines

- Cache coherent shared memory is difficult to scale
- Current multicore machines: 64 - hundreds hardware threads
- Larger machines (virtually all supercomputers) consist of a collection of nodes without hardware-supported shared memory; i.e. they are distributed memory machines
the choice is how much to expose the hardware to the programmer

- **message passing models**: directly expose the fact that there are no shared memory

- **global address space models**: build a software layer that looks like a shared memory, but does not necessarily cache data
Today’s topics

1. Introduction
2. MPI
   - MPI overview
   - Point-to-point communication
   - Quick performance tests
   - Subtleties in semantics and pitfalls
   - Collective communication
3. UPC
   - UPC overview
   - UPC’s shared memory
   - Quick performance tests
   - UPC’s parallel loop
   - Synchronization
   - Collective Communication
4. Summary
• **MPI** = Message Passing Interface
• *de fact* standard model for programming distributed memory machines (esp. supercomputers)
• C/C++/Fortran + library APIs
Reference

- http://www.mpi-forum.org/
- section numbers below refer to those in MPI-2.1 document
Implementations

- popular open source implementations:
  - OpenMPI: http://www.open-mpi.org/
  - MVAPICH: http://mvapich.cse.ohio-state.edu/
- many Linux distributions provide binary packages for them
- supercomputers often provide vendor implementations (e.g. Fujitsu)
- unless otherwise stated, this tutorial uses OpenMPI library
Compiling and running MPI programs

- compile with \{mpicc,mpicxx,mpic++\} driver, which invokes a regular compiler (e.g. gcc/g++)

  $ mpicc -Wall program.c

- run with \{mpiexec,mpirun\} launcher

  $ mpirun -n 1 ./a.out
  $ mpirun --hostfile hosts -n 4 ./a.out

- command names and options may differ between implementations and environments
Hello MPI world

```c
#include <stdio.h>
#include <mpi.h>
#include <unistd.h>

int main(int argc, char ** argv) {
    int rank, n_procs;
    char hostname[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
    gethostname(hostname, sizeof(hostname));
    printf("Hello MPI, I am %s, %d-th of %d procs\n", hostname, rank, n_procs);
    MPI_Finalize();
}
```
And run it

```
oooka000:ex% mpicc hello_mpi.c
ooooka000:ex% mpirun -n 1 ./a.out
Hello MPI, I am oooka000, 0-th of 1 procs
ooooka000:ex% mpirun -n 3 ./a.out
Hello MPI, I am oooka000, 0-th of 3 procs
Hello MPI, I am oooka000, 1-th of 3 procs
Hello MPI, I am oooka000, 2-th of 3 procs
ooooka000:ex% cat hosts
ooooka000
ooooka001
ooooka002
ooooka003
ooooka000:ex% mpirun -n 3 --hostfile hosts ./a.out
Hello MPI, I am oooka000, 0-th of 3 procs
Hello MPI, I am oooka001, 1-th of 3 procs
Hello MPI, I am oooka002, 2-th of 3 procs
```
And run it

```bash
oooka000:ex% mpirun -n 6 --hostfile hosts ./a.out
Hello MPI, I am oooka000, 0-th of 6 procs
Hello MPI, I am oooka003, 3-th of 6 procs
Hello MPI, I am oooka001, 1-th of 6 procs
Hello MPI, I am oooka002, 2-th of 6 procs
Hello MPI, I am oooka000, 4-th of 6 procs
Hello MPI, I am oooka001, 5-th of 6 procs
```

More about `--hostfile` option later
**MPI programming model basics**

- **mpirun** launches **many instances of the same program**
  - often called Single Program Multiple Data (SPMD) model
- you can obtain
  - the number of processes (**MPI_Comm_size**)
  - the *rank* (unique serial id) of the calling process (**MPI_Comm_rank**)
- you can communicate with other processes via messages, using ranks as message destinations

```
rank=0 rank=1 rank=2
```
it represents the set of processes consisting of all participating processes
  - in MPI, such a set of processes is called a *communicator*
you may create a communicator by subsetting an existing communicator (§6.4)
all communication routines take a communicator argument
but in this tutorial, we use only MPI_COMM_WORLD
Communication among processes

- point-to-point communication (§3)
- collective communication (§5)
MPI_Send

```c
int MPI_Send(buf, n, item_type, dst, tag, comm)
```

- sends a message of `n` items of `item_type` stored from address `buf` to process `dst` in communicator `comm`, with `tag`
  - `tag` is a small integer with which the receiver can select a message it wants to receive at this time
  - `item_type` is a small integer such as MPI_BYTE, MPI_INT, etc. it’s simple to stick to MPI_BYTE
MPI_Recv

```c
int MPI_Recv(buf, n, item_type, src, tag, comm, status_p)
```

- receives a message of up to `n` items of `item_type` with `tag` from process `from` in communicator `comm` and obtain its status in `status_p`
- `status_p` is a pointer to MPI_Status type
- you may supply MPI_STATUS_IGNORE when unimportant
MPI_Send/MPI_Recv example

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char ** argv) {
    int rank, n_procs;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
    if (rank == 0) {
        MPI_Send("hello", 6, MPI_BYTE, 1, 918, MPI_COMM_WORLD);
    } else if (rank == 1) {
        char msg[6];
        MPI_Recv(msg, 7, MPI_BYTE, 0, 918, MPI_COMM_WORLD,
                  MPI_STATUS_IGNORE);
        printf("received %s\n", msg);
    }
    MPI_Finalize();
}
```
Receiving a message of multiple sources/tags/sizes

• the API suggests the receiver must know which message will arrive next (source, tag, item_type, and the number of elements)

• but what if there are multiple possibilities?
  • MPI_ANY_SOURCE is a source that matches any
  • MPI_ANY_TAG is a tag that matches any
  • MPI_Probe (§3.8) provides a means to wait for a message without receiving it
MPI_Probe example

if (rank < 2) {
    char * msg = (rank == 0 ? "good morning" : "good afternoon");
    MPI_Send(msg, strlen(msg)+1, MPI_BYTE, 2, 918+rank, MPI_COMM_WORLD);
} else if (rank == 2) {
    int i;
    for (i = 0; i < 2; i++) {
        MPI_Status s; int n;
        MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &s);
        MPI_Get_count(&s, MPI_BYTE, &n);
        char * a = malloc(n);
        MPI_Recv(a, n, MPI_BYTE, s.MPI_SOURCE, s.MPI_TAG, MPI_COMM_WORLD, &s);
        printf("received %s (tag=%d) from %d\n", a, s.MPI_TAG, s.MPI_SOURCE);
    }
}
A quick overhead/latency test

- it heavily depends on several factors
  - MPI implementation
  - software stack (e.g. bypass kernel or not)
  - network fabric (e.g. Ethernet or Infiniband)
- test code

```c
if (rank == 0) {
    for (i = 0; i < n; i++) {
        MPI_Send(msg, sz, MPI_BYTE, 1, 918, ...);
        MPI_Recv(msg, sz, MPI_BYTE, ...);
    }
} else if (rank == 1) {
    for (i = 0; i < n; i++) {
        MPI_Recv(msg, sz, MPI_BYTE, ...);
        MPI_Send(msg, sz, MPI_BYTE, 0, 918, ...);
    }
}
```
A quick overhead/latency test

- results with 4 bytes messages

<table>
<thead>
<tr>
<th>MPI</th>
<th>stack</th>
<th>network</th>
<th>1-way latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMPI</td>
<td>TCP/IP</td>
<td>10G Ethernet</td>
<td>57175 ns</td>
</tr>
<tr>
<td>OpenMPI</td>
<td>OpenFabric</td>
<td>Infiniband Tofu</td>
<td>1790 ns</td>
</tr>
<tr>
<td>Fujitsu MPI</td>
<td>within a node</td>
<td></td>
<td>1648 ns</td>
</tr>
<tr>
<td>OpenMPI</td>
<td></td>
<td></td>
<td>1045 ns</td>
</tr>
</tbody>
</table>

- do you remember latencies on shared memory?
- 2 workers: worker 0 on CPU #0

<table>
<thead>
<tr>
<th>CPU for worker 1</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>≈ 20 ns</td>
</tr>
<tr>
<td>4,8,12,16,20,28,32,36,40,44 others</td>
<td>≈ 130 ns</td>
</tr>
<tr>
<td>others</td>
<td>≈ 500 ns</td>
</tr>
</tbody>
</table>
Bandwidth

- 4 bytes / 1790 ns \(\approx\) 2 MB/sec \(\ll\) network capability
  - GigE : 1 Gbps \(\approx\) 100MB/sec
  - 10 GE : 10 Gbps \(\approx\) 1GB/sec
  - Infiniband : 2-168 Gbps

- getting large bandwidth need large messages
- i.e. *sending a single large message is faster than sending many small messages*
Bandwidth

results with OpenMPI/OpenFabric/Infiniband (4x QDR):

<table>
<thead>
<tr>
<th>msg size (bytes)</th>
<th>transfer rate (MB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.23</td>
</tr>
<tr>
<td>40</td>
<td>20.83</td>
</tr>
<tr>
<td>400</td>
<td>102.31</td>
</tr>
<tr>
<td>4K</td>
<td>538.62</td>
</tr>
<tr>
<td>40K</td>
<td>991.15</td>
</tr>
<tr>
<td>400K</td>
<td>1953.30</td>
</tr>
<tr>
<td>4M</td>
<td>2330.23</td>
</tr>
<tr>
<td>40M</td>
<td>2500.95</td>
</tr>
<tr>
<td>400M</td>
<td>2462.58</td>
</tr>
</tbody>
</table>
Subtleties in MPI\_Send/MPI\_Recv semantics

**MPI\_Send** may or may not wait for the receiver to receive it

- *not guaranteed*
  - after MPI\_Send returns, you cannot assume the receiver reached MPI\_Recv
  - you cannot assume MPI\_Send returns until the receiver reached MPI\_Recv

- *guaranteed:*
  - after MPI\_Send returns, you may assume the message has been secured; you may modify/reuse the message buffer
  - messages between the same two processes arrive in the same order as they are sent (*non-overtaking*)
Motivation behind the weak semantics

upon `MPI_Send`, we like to:

- avoid handshake overhead/latency for small messages → inject or buffer the message and immediately

- avoid copying large messages → handshake with the receiver and move the data directly from the sender-supplied region to the receiver-supplied region
Consequences (1) — deadlock

processes sending to each other may deadlock (*send deadlock*)

\[
\begin{align*}
\text{rank 0:} & \\
&MPI\_Send \rightarrow 1; \\
&MPI\_Recv;
\end{align*}
\]

\[
\begin{align*}
\text{rank 1:} & \\
&MPI\_Send \rightarrow 0; \\
&MPI\_Recv;
\end{align*}
\]
Consequences (2)

A sent message may be overtaken by another message sent later through a different path.

<table>
<thead>
<tr>
<th>rank 0:</th>
<th>rank 1:</th>
<th>rank 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send to 2;</td>
<td>MPI_Recv;</td>
<td>MPI_Recv;</td>
</tr>
<tr>
<td>MPI_Send to 1;</td>
<td>MPI_Send to 2;</td>
<td>MPI_Recv;</td>
</tr>
</tbody>
</table>

Diagram:

```
rank 0    rank 1    rank 2
```

```
  ↓                  ↓                  ↓
  ⏯                  ⏯                  ⏯
  ⏯                  ⏯                  ⏯
  ⏯                  ⏯                  ⏯
```
Communication modes (§3.4)

There are other communication modes having stronger guarantees:

- **buffered** send (MPI_Bsend):
  guarantees it returns without waiting for MPI_Recv to be issued; you need to allocate a buffer (§3.6)

- **synchronous** send (MPI_Ssend):
  guarantees it returns only after the matching MPI_Recv has been issued

- **ready** send (MPI_Rsend)
  if MPI_Recv has not been issued when it is called, it returns an error
Non-blocking communication (§3.7)

Orthogonal to the communication modes, the send/recv can be made non-blocking

- initiate sends or receives and later wait for their completion (MPI_Wait)
  - MPI_I{I,B,S,R}send
  - MPI_Irecv
- send initiation returns without securing the message; you may not modify the message buffer until a send completes
Collective communication (§5)

- MPI provides several operations that involve many processes
  - 1-N: broadcast (§5.4), scatter(v) (§5.6)
  - N-1: reduction (§5.9), gather(v) (§5.5)
  - N-N: alltoall(v) (§5.8)
  - N-1-N: allgather (§5.7), allreduce (§5.7), reduction-scatter(-block) (§5.10)
  - barrier (§5.3)
  - scan (§5.11)

- common to all operations: *all processes in a communicator must call the same procedure*

- we’ll see reduction in detail as an example
Logical flow of data

broadcast
scatter
reduction
gather
alltoall

Note: actual dataflow may be different (optimized)
Reduction

int MPI_Reduce(sbuf, rbuf, n, type, op, root, comm);

- reduce each of the \( n \) elements of type \( \text{type} \) in \( \text{sbuf} \) of all processes in communicator \( \text{comm} \) by operation \( \text{op} \), and put the results in \( \text{rbuf} \) of process \( \text{root} \)
- \( \text{op} \) is an integer of type \( \text{MPI}_{\text{Op}} \) (§5.9.2)
Message passing vs. shared memory — why MPI prevails?

- despite its programming difficulty, MPI dominates programming “at scale”
- in message passing, the programmer specifies “who needs this data” (and send data to only those who need them)
- in shared memory, the programmer does not give such information, and the system must be prepared for all possibilities in future
  - it may be read by all threads, just one, or even none
- yet it must always invalidate stale data in others’ caches
  - option 1: invalidate all caches
  - option 2: keep track of who might have cached it
- either option generates more traffic than the application really needs
UPC

- UPC = Unified Parallel C
- shared memory-like programming on distributed memory machines
- extension to C language
Reference

- http://upc.gwu.edu/
- section numbers below refer to those in UPC Manual http://upc.gwu.edu/downloads/Manual-1.2.pdf
- book: “UPC: Distributed Shared-Memory Programming”
Implementations

- several implementations
  - Berkeley UPC (UPC → C translator)
  - GNU UPC
- this tutorial uses Berkeley UPC 2.14.2 http://upc.lbl.gov/
  - note command names are slightly different from the UPC manual
  - see http://upc.lbl.gov/docs/ for how to use
- a single installation can run on top of various network stack
  - UDP (udp)
  - MPI (mpi)
  - Infiniband (ibv)
Compiling and running UPC programs

- compile with **upcc** frontend

  ```
  $ upcc program.c
  $ upcc -network={mpi,ibv,udp} program.c
  $ upcc -conf=config program.c
  ```

- run with **upcrun** launcher

  ```
  $ upcrun -n 2 ./a.out
  ```

- AFAIK, **mpirun** is necessary to specify hosts

  ```
  $ UPC_PTHREADS_PER_PROC=3 mpirun -n 4 --hostfile hosts ./a.out
  ```

  will launch $4 \times 3 = 12$ threads
Hello UPC world

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char ** argv) {
    char hostname[100];
    gethostname(hostname, sizeof(hostname));
    printf("Hello UPC, I am %s, %d-th of %d procs\n", hostname, MYTHREAD, THREADS);
}
```
And run it

```
ookia000:ex% upcc hello_upc.c
ookia000:ex% upcrun -n 1 ./a.out
UPCR: UPC thread 0 of 1 on ookia000 (...)
Hello UPC, I am ookia000, 0-th of 1 threads
ookia000:ex% mpirun --hostfile hosts -n 2 ./a.out
UPCR: UPC thread 0 of 1 on ookia000 (...)
UPCR: UPC thread 0 of 1 on ookia001 (...)
Hello UPC, I am ookia000, 0-th of 1 threads
Hello UPC, I am ookia001, 0-th of 1 threads
```
UPC programming model basics

- SPMD model. similar to MPI, or, OpenMP with `#pragma omp parallel` applied to `main()`
- UPC’s thread ≈ MPI’s process or OpenMP’s thread
- you can obtain
  - the number of threads (THREADS)
  - the unique serial id of the calling thread (MYTHREAD)
- by default, regions (variables/arrays/dynamically allocated memory) are private to a thread
- regions can be declared shared among threads
- you can communicate with other threads via shared variables
UPC’s shared memory: *Partitioned Global Address Space*

- Participating nodes cooperate in providing shared memory.
- Simple/predictable rules determine how variables/array elements are distributed to threads.
- There are *no caches*.
UPC’s shared memory and communication

- The rule is simple: access shared data provided by thread $x$ → communicate with the node thread $x$ is running

  ```
  a[0] = 10;  // send to whoever provides a[0]
  s = a[1];   // send to and recv from whoever provides a[1]
  ```

- In contrast, communication implied by *cache-coherent* shared memory is much more complex
  - whether you read from/wrote to it recently?
  - whether anybody else, and how many read/wrote it after you?
the canonical distribution is *block-cyclic* distribution

*block-cyclic distribution of block size* $S$ cyclically puts $S$ consecutive elements (blocks) to all threads (i.e. with $T$ threads, $0, 1, 2, \ldots, T - 1, 0, 1, 2, \ldots$)

*cyclic* distribution $\equiv$ block cyclic with block size 1

*block* distribution $\equiv$ block cyclic in which each thread has a single block

block-cyclic distribution of block size 3 to 4 threads
Global variables/arrays can be made shared via the following syntax (§3.2)

- **shared int a;** (scalars on 0)
  - [Diagram showing scalar distribution]

- **shared int b[2*THREADS];** (arrays are cyclic by default)
  - [Diagram showing cyclic array distribution]

- **shared [3] int c[6*THREADS];** (specify block size = 3)
  - [Diagram showing block distribution]

- **shared [*] int d[10*THREADS];** (block distribution)
  - [Diagram showing block distribution]

- **shared [] int e[12];** (all elements on 0)
  - [Diagram showing distribution]
Shared memory (2) — dynamic allocation

Shared memory can be dynamically allocated via the following APIs (§6)

- **upc_alloc(size)**: allocate size bytes on the calling thread
  
  ```c
  size
  ```

- **upc_global_alloc(nb, bs)**: block-cyclic-allocate bs bytes \(\times\) nb blocks
  
  ```c
  \approx\text{shared char}\ [nb]\ a[nb \times bs];
  ```

- **upc_all_alloc(nb, bs)**: similar **upc_global_alloc(nb, bs)**, but it is a collective operation (must be called by all threads with the same arguments) and everybody gets the same pointer
A quick overhead/latency test

code:

```c
// allocate n ints on thread 0
shared [] char* a= upc_all_alloc(1, n);
// a single thread accesses it
if (MYTHREAD == ??)
  for (i = 0; i < n; i++) s += a[i];
```

result (note they are round-trip times)

<table>
<thead>
<tr>
<th>MPI</th>
<th>stack</th>
<th>network</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMPI</td>
<td>UDP</td>
<td>Infiniband</td>
<td>73652.03 ns</td>
</tr>
<tr>
<td>OpenMPI</td>
<td>MPI</td>
<td>Infiniband</td>
<td>4872.39 ns</td>
</tr>
<tr>
<td>OpenMPI</td>
<td>OpenFabric</td>
<td>Infiniband</td>
<td>2858.71 ns</td>
</tr>
<tr>
<td>OpenMPI</td>
<td>within a node</td>
<td>Infiniband</td>
<td>8.57 ns</td>
</tr>
</tbody>
</table>
Bulk transfer

- same as MPI, sending a single large message is faster than sending many small messages
- current UPC implementation generates a message for each remote access
- there are APIs to bulk-transfer large data
  - `upc_memget(dst, src, sz);` : shared → local
  - `upc_memput(dst, src, sz);` : local → shared
  - `upc_memcpy(dst, src, sz);` : shared → shared
Using bulk transfer

- **code:**

```c
shared [] int* a = upc_all_alloc(1, sizeof(int)*n);
int * la = calloc(1, sizeof(int)*n);
if (MYTHREAD == ???) {
    // a single large transfer
    upc_memget(la, a, sizeof(int)*n);
    for (i = 0; i < n; i++) s += la[i];
}
```

- **results:**

<table>
<thead>
<tr>
<th>msg size (bytes)</th>
<th>transfer rates (MB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.38</td>
</tr>
<tr>
<td>40</td>
<td>13.56</td>
</tr>
<tr>
<td>400</td>
<td>123.92</td>
</tr>
<tr>
<td>4K</td>
<td>822.17</td>
</tr>
<tr>
<td>40K</td>
<td>2288.25</td>
</tr>
<tr>
<td>400K</td>
<td>Caught a signal: SIGTERM(15)?</td>
</tr>
</tbody>
</table>

Caught a signal: SIGTERM(15)?
upc forall: UPC's work-sharing for loop (§4)

- syntax:
  \[ \text{upc forall} \ (\text{init};\ \text{cond};\ \text{incr};\ \text{affinity}) \ \text{statement} \]

- basic semantics:
  - all threads must encounter the same for loop and they cooperatively execute iterations
  - \textit{affinity} specifies who executes the iteration
  - it is of type int or a pointer to shared memory
    - if it's an int, iterations whose \textit{affinity} are evaluated to \( t \) will be executed by thread \( (t \mod \text{THREADS}) \)
    - if it's a pointer to shared memory, iterations whose \textit{affinity} are evaluated to \( p \) will be executed by the thread that owns address \( p \)
  - no equivalent of OpenMP's \texttt{schedule(dynamic)}
upc forall example

```c
upc forall (i = 0; i < n; i++; i) {
    printf("iteration %d by thread %d\n", i, MYTHREAD);
}

shared [3] int a[N];
int main () {
    upc forall (i = 0; i < N; i++; &a[i]) {
        a[i] = 10;
    }
}
```
Parallelizing nested loops

- basically, only the outermost `upc_forall` is partitioned; inner `upc_forall`’s are executed by all threads
- outer `upc_forall` loops may specify `continue` as their affinity expression, in which case all iterations of them are executed by all threads
- ex.

```plaintext
upc_forall (i = 0; i < n; i++; continue) {
    upc_forall (j = 0; j < n; j++; &a[i][j]) {
        a[i][j] = ...;
    }
}
```
Producer-consumer synchronization

- let’s say we like to sum up all values in a shared array,
- by the usual strategy:
  - each thread computes the sum of its part
  - a master thread computes the global sum
- something like this ??

```c
b[MYTHREAD] = 0.0;
upc_forall(i = 0; i < n; i++; &a[i])
    b[MYTHREAD] += a[i];
if (MYTHREAD == 0) {
    s = 0.0;
    for (i = 0; i < THREADS; i++) s += b[i];
}
```
doesn’t work because there are no guarantees that thread 0 reads
b[i]’s after other threads have written to them

→ barrier synchronization
upc_barrier: UPC’s barrier synchronization

```c
b[MYTHREAD] = 0.0;
upc_forall(i = 0; i < n; i++; &a[i])
    b[MYTHREAD] += a[i];
upc_barrier;
if (MYTHREAD == 0) { ... }
```

```
thread 0       thread 1       thread 2       thread 3
b[0] += ...
|            |            |            |            |
| b[0] += ...
|            |            |            |            |
| b[0] += ...
|            |            |            |            |
| b[0] += ...

upc_forall

b[2] += ...
|            |            |            |            |
| b[2] += ...
|            |            |            |            |
| b[2] += ...
|            |            |            |            |
| b[2] += ...

upc_barrier;

s += b[0];
|            |            |            |            |
| s += b[1];
|            |            |            |            |
| s += b[2];
|            |            |            |            |
| s += b[3];
```

```c
s += b[0];
```

```c
s += b[1];
```

```c
s += b[2];
```

```c
s += b[3];
```
Note: why didn’t we need it in OpenMP?

- any shared memory system has the same problem, including OpenMP
- OpenMP, however, ensures some orderings automatically
  - a barrier after `#pragma omp for`
  - for `#pragma omp parallel`, natural orderings before parallel region → inside parallel region → after parallel region
- note: it’s merely the language spec that makes the difference, not the underlying platform (e.g. hardware vs. software shared memory)
Collective communications

- UPC also provides a similar set of APIs for collective communications (§7.1.1)
- they generally shuffle data on shared memory
- reduction can take user-defined functions too
  - **upc_all_broadcast**
  - **upc_all_scatter**
  - **upc_all_gather**
  - **upc_all_gather_all**
  - **upc_all_exchange**
  - **upc_all_permute**
  - **upc_all_reduce**
  - **upc_all_sort**
Summary

- Common:
  - large messages are network friendly

- MPI:
  - SPMD; you can communicate via messages
  - aware of blocking semantics
  - may utilize collective communication

- UPC:
  - SPMD; you can communicate via shared variables/arrays
  - insert barriers when necessary
  - know which node you are talking to