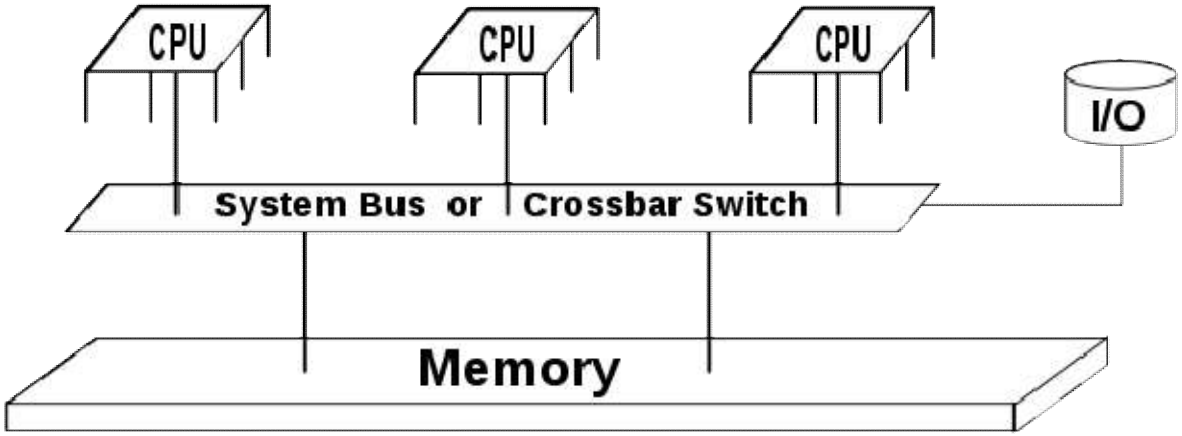
# High Performance Computing

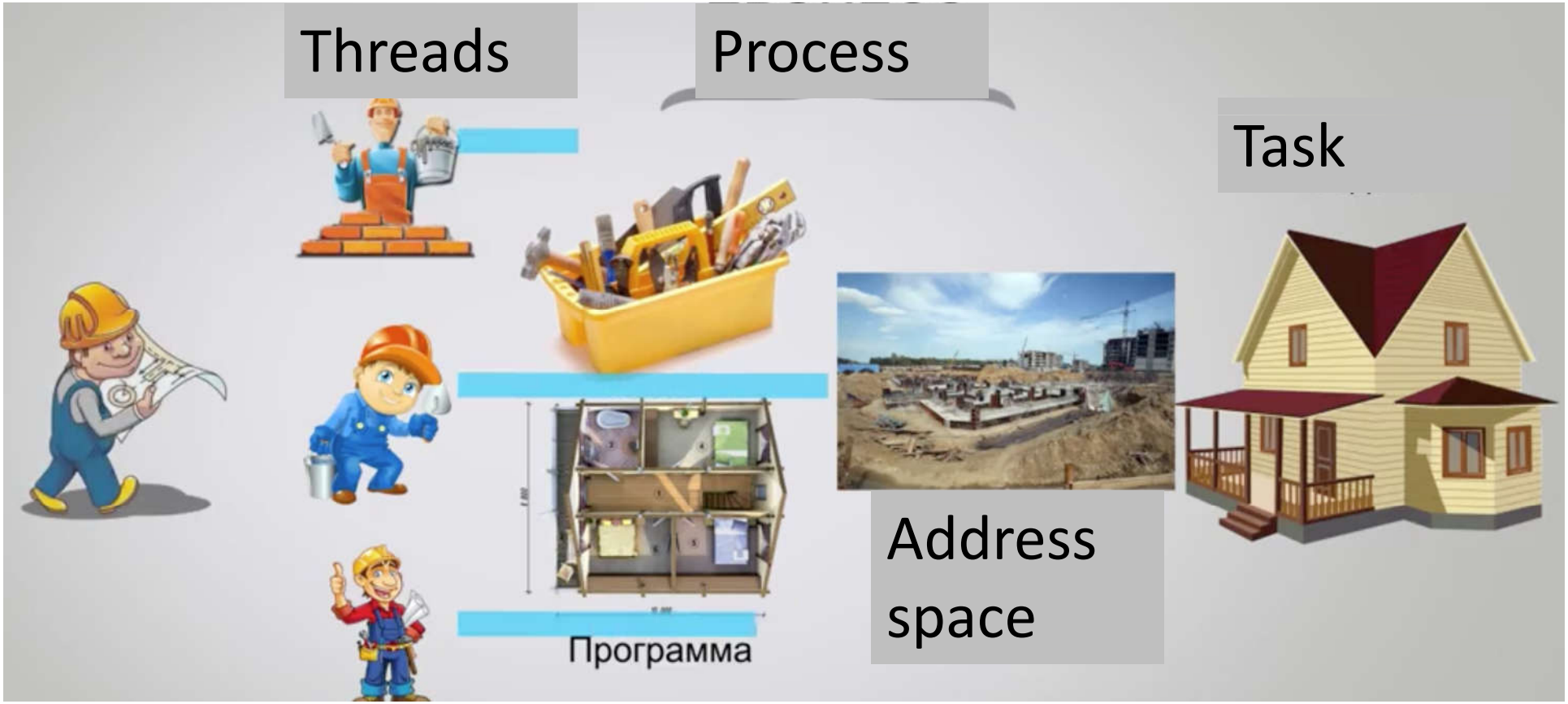
## Term 4 2018/2019

Lecture 3

# Shared memory and distributed memory

**shared memory**



# OpenMP

OpenMP is an API for multithreaded, shared memory parallelism.

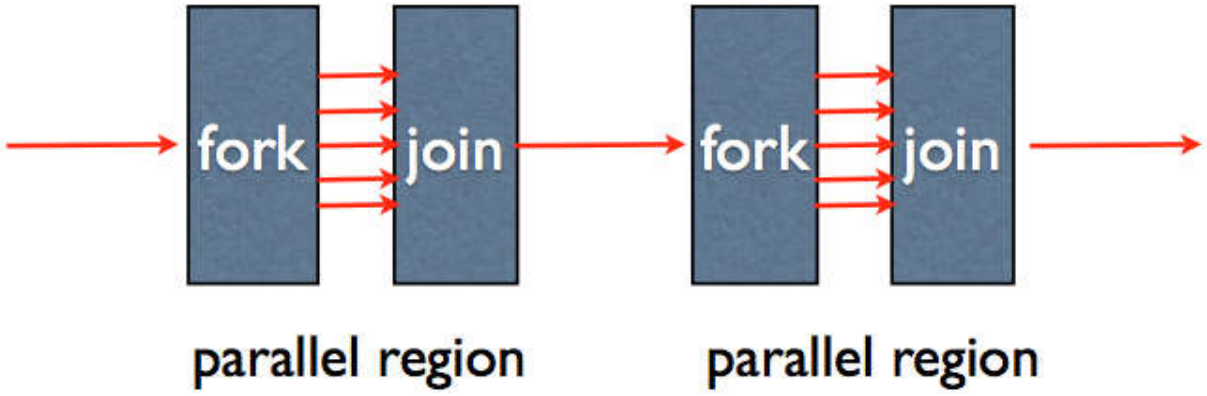
- A set of compiler directives inserted in the source program
  - pragmas in C/C++ (pragma = compiler directive external to prog. lang. for giving additional info., usually non-portable, treated like comments if not understood)
  - (specially written) comments in fortran
- Library functions
- Environment variables

Goal is standardization, ease of use, portability. Allows incremental approach. Significant parallelism possible with just 3 or 4 directives.

# Fork-Join concept

Explicit programmer control of parallelization using fork-join model of parallel execution

- all OpenMP programs begin as single process, the master thread, which executes until a parallel region construct encountered
- FORK: master thread creates team of parallel threads
- JOIN: When threads complete statements in parallel region construct they synchronize and terminate, leaving only the master thread. (similar to fork-join of Pthreads)



# Fork-Join concept

- User inserts directives telling compiler how to execute statements
  - which parts are parallel
  - how to assign code in parallel regions to threads
  - what data is private (local) to threads
  - `#pragma omp` in C and `!$omp` in Fortran
- Compiler generates explicit threaded code
- Rule of thumb: One thread per core (2 or 4 with hyperthreading)
- Dependencies in parallel parts require synchronization between threads

# Simple example

```
#include <omp.h>
#include <stdio.h>

int main() {

#pragma omp parallel
printf("Hello world from thread %d\n",omp_get_thread_num());

return 0;
}
```

Compile line:

```
gcc -fopenmp helloWorld.c
```

# Setting number of threads

## Environment Variables:

`setenv OMP_NUM_THREADS 2` (cshell)

`export OMP_NUM_THREADS=2` (bash shell)

## Library call:

`omp_set_num_threads(2)`

```
#include <omp.h>
#include <stdio.h>

int main() {

  omp_set_num_threads(2);

#pragma omp parallel
  printf("Hello world from thread %d\n",omp_get_thread_num());

return 0;
}
```

# Parallel construct

```
#include <omp.h>

int main(){
  int var1, var2, var3;

  ...serial Code

  #pragma omp parallel private(var1, var2) shared (var3)
 {
    ...parallel section
 }

 ...resume serial code

}
```

# Parallel construct

- When a thread reaches a PARALLEL directive, it becomes the master and has thread number 0.
- All threads execute the same code in the parallel region (Possibly redundant, or use work-sharing constructs to distribute the work)
- There is an implied barrier* at the end of a parallel section. Only the master thread continues past this point.
- If a thread terminates within a parallel region, all threads will terminates, and the result is undefined.

barrier - all threads wait for each other; no thread proceeds until all threads have reached that point

# Parallel construct

- If program compiled serially, openMP pragmas and comments ignored, stub library for omp library routines
- easy path to parallelization
- One source for both sequential and parallel helps maintenance.

# Work sharing constructs

- work-sharing construct divides work among member threads. Must be dynamically within a parallel region.
- No new threads launched. Construct must be encountered by all threads in the team.
- No implied barrier on entry to a work-sharing construct; Yes at end of construct.

3 types of work-sharing construct (4 in Fortran - array constructs):

- for loop: share iterates of `for` loop ("data parallelism") iterates must be independent
- sections: work broken into discrete section, each executed by a thread ("functional parallelism")
- single: section of code executed by one thread only

# For schedule example

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

#define N 20

int main()
{
    int sum=0;
    int a[N], i;

    #pragma omp parallel for
    for(i=0;i<N;i++)
    {
        a[i]=i;
        printf("iterate i=%3d by thread %3d\n",i, omp_get_thread_num());
    }
    return 0;

}
```

# OpenMP directives

All directives:

```
#pragma omp directive  [clause ...]
                        if (scalar_expression)
                        private (list)
                        shared (list)
                        default (shared | none)
                        firstprivate (list)
                        reduction (operator: list)
                        copyin (list)
                        num_threads (integer-expression)
```

Directives are:

- Case sensitive (not for Fortran)
- Only one directive-name per statement
- Directives apply to at most one succeeding statement, which must be a structured block.
- Continue on succeeding lines with backslash ( "\" )

# OpenMP for

```
#pragma omp for [clause ...]
                    schedule (type [,chunk])
                    private (list)
                    firstprivate(list)
                    lastprivate(list)
                    shared (list)
                    reduction (operator: list)
                    nowait
```

**SCHEDULE**: describes how to divide the loop iterates

- **static** = divided into pieces of size *chunk*, and statically assigned to threads. Default is approx. equal sized chunks (at most 1 per thread)
- **dynamic** = divided into pieces of size *chunk* and dynamically scheduled as requested. Default chunk size 1.
- **guided** = size of chunk decreases over time. (Init. size proportional to the number of unassigned iterations divided by number of threads, decreasing to *chunk size*)
- **runtime**=schedule decision deferred to runtime, set by environment variable OMP_SCHEDULE.

# OpenMP for example

```
#pragma omp parallel shared(n,a,b,x,y), private(i)
{ // start parallel region

    #pragma omp for nowait
    for (i=0;i<n;i++)
      b[i] = += a[i];

    #pragma omp for nowait
    for (i=0;i<n;i++)
      x[i] = 1./y[i];

} // end parallel region  (implied barrier)
```

Spawning tasks is expensive: reuse if possible.
*nowait* clause: minimize synchronization.

# OpenMP sections example

```
#pragma omp sections [clause ...]
                     private (list)
                     firstprivate(list)
                     lastprivate(list)
                     reduction (operator: list)
                     nowait
{
 #pragma omp section
   structured block
 #pragma omp section
   structured block
}
```

- implied barrier at the end of a SECTIONS directive, unless a NOWAIT clause used
- for different numbers of threads and SECTIONS some threads get none or more than one
- cannot count on which thread executes which section
- no branching in or out of sections

# OpenMP sections example

```
#pragma omp single [clause ...]
                   private (list)
                   firstprivate(list)
                   nowait
 structured block
```

- SINGLE directive says only one thread in the team executes the enclosed code
- useful for code that isn't thread-safe (e.g. I/O)
- rest of threads wait at the end of enclosed code block (unless NOWAIT clause specified)
- no branching in or out of SINGLE block

# OpenMP firstprivate example

What is wrong with this code snippet?

```
#pragma omp parallel for
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

# OpenMP firstprivate example

What is wrong with this code snippet?

```
#pragma omp parallel for
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

By default, `x` is shared variable (i is private).

Could have: Thread 0 set `x` for some `i`.
Thread 1 sets `x` for different `i`.
Thread 0 uses `x` but it is now incorrect.

# OpenMP firstprivate example

Instead use:

```
#pragma omp parallel for private(x)
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

What about `i, dx, y`?

By default `dx, n, y` shared.
`dx, n` used but not changed. `y` changed, but independently for each `i`

# OpenMP firstprivate example

What is wrong with this code?

```
dx = 1/n.;
#pragma omp parallel for private(x,dx)
for (i=0;i<n;i++){
     x = i*dx
     y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

Specifying `dx` private creates a new private variable for each thread, but it is not initialized.

firstprivate clause creates private variables and initializes to the value from the master thread before the loop.

lastprivate copies last value computed by a thread (for *i=n*) to the maser thread copy to continue execution.

# Clauses

These clauses not strictly necessary but may be convenient (and may have performance penalties too).

- lastprivate private data is undefined after parallel construct. this gives it the value of last iteration (as if sequential) or sections construct (in lexical order).
- firstprivate pre-initialize private vars with value of variable with same name before parallel construct.
- default (`none | shared`). In fortran can also have private. Then only need to list exceptions. (none is better habit).
- nowait suppress implicit barrier at end of work sharing construct. Cannot ignore at end of parallel region. (But no guarantee that if have 2 `for` loops where second depends on data from first that same threads execute same iterates)

# More clauses

- if (logical expr) true = execute parallel region with team of threads; false = run serially (loop too small, too much overhead)
- reduction for assoc. and commutative operators compiler helps out; reduction variable is shared by default (no need to specify).

```
#pragma omp parallel for default(none) \
                         shared(n,a) \
                         reduction(+:sum)
  for (i=0;i<n;i++)
    sum += a[i]
  /* end of parallel reduction  */
```

Also other arithmetic and logical ops., `min`,`max` instrinsics in Fortan only.

# Race condition example

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

int main(){

  int x = 2;

  #pragma omp parallel shared(x) num_threads(2)
  {
    if (1 == omp_get_thread_num()){
      x = 5;
    }
    else {
      printf("1: Thread %d has x = %d\n",omp_get_thread_num(),x);
    }

    #pragma omp barrier

    if (0 == omp_get_thread_num()) {
      printf("2: thread %d has x = %d\n",omp_get_thread_num(),x);
    }
    else {
      printf("3: thread %d has x = %x\n",omp_get_thread_num(),x);
    }

  }
  return 0;
}
```

# Synchronization in OpenMP

- Implicit barrier synchronization at end of parallel region (no explicit support for synch. subset of threads). Can invoke explicitly with `#pragma omp barrier`. All threads must see same sequence of work-sharing and barrier regions .
- critical sections: only one thread at a time in critical region with the same name. `#pragma omp critical [(name)]`
- atomic operation: protects updates to individual memory loc. Only simple expressions allowed. `#pragma omp atomic`
- locks: low-level run-time library routines (like mutex vars., semaphores)
- flush operation - forces the executing thread to make its values of shared data consistent with shared memory
- master (like single but not implied barrier at end), *ordered*, ...

At all these (implicit or explicit ) synchronization points OpenMP ensures that threads have consistent values of shared data.

# Atomic example

```
int sum = 0;
#pragma omp parallel for shared(n,a,sum)
{
  for (i=0; i<n; i++){
  #pragma omp atomic
     sum = sum + a[i];
  }
}
```

Better to use a *reduction* clause:

```
int sum = 0;
#pragma omp parallel for shared(n,a)   \
                    reduction(+:sum)
{
  for (i=0; i<n; i++){
     sum += a[i];
  }
}
```

# Locks

Locks control access to shared resources. Up to implementation to use spin locks (busy waiting) or not.

- Lock variables must be accessed only through locking routines:

  `omp_init_lock` `omp_destroy_lock`
  `omp_set_lock` `omp_unset_lock` `omp_test_lock`

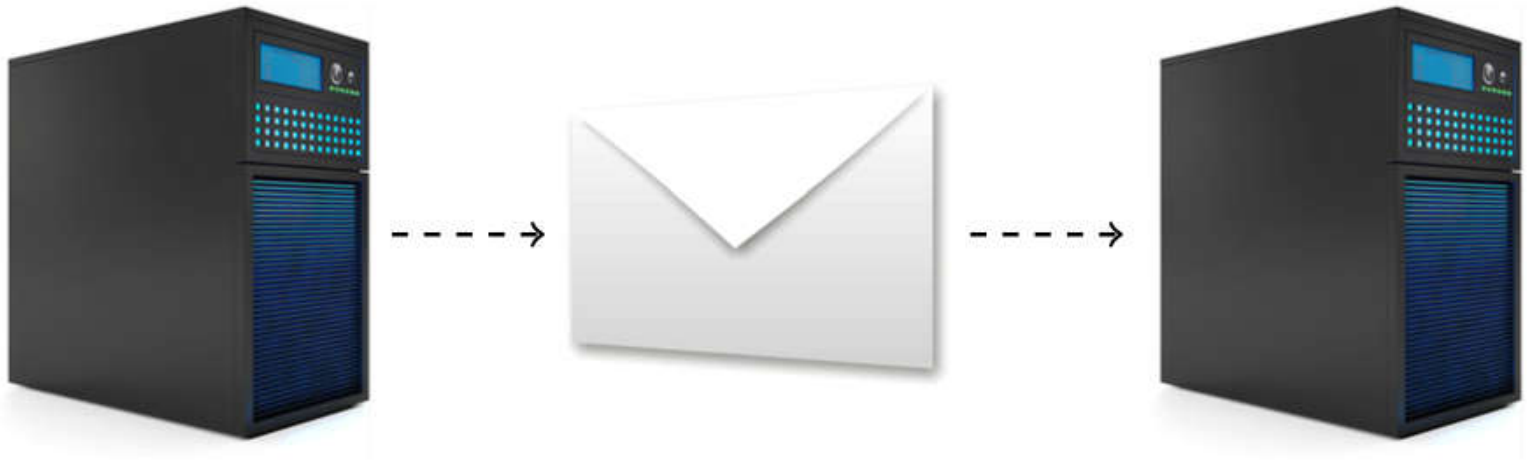
- In C, lock is a type `omp_lock_t` or `omp_nest_lock_t` (In Fortran lock variable is integer)
- initial state of lock is unlocked.
- `omp_set_lock(omp_lock_t *lock)` forces calling thread to wait until the specified lock is available. (Non-blocking version is `omp_test_lock`

# Locks

```
#include <stdio.h>
#include <omp.h>

omp_lock_t my_lock;

int main() {
   omp_init_lock(&my_lock);

   #pragma omp parallel
   {
      int tid = omp_get_thread_num( );
      int i, j;

      for (i = 0; i < 5; ++i) {
         omp_set_lock(&my_lock);
         printf("Thread %d - starting locked region\n", tid);

         printf("Thread %d - ending locked region\n", tid);
         omp_unset_lock(&my_lock);
      }
   }

   omp_destroy_lock(&my_lock);
}
```

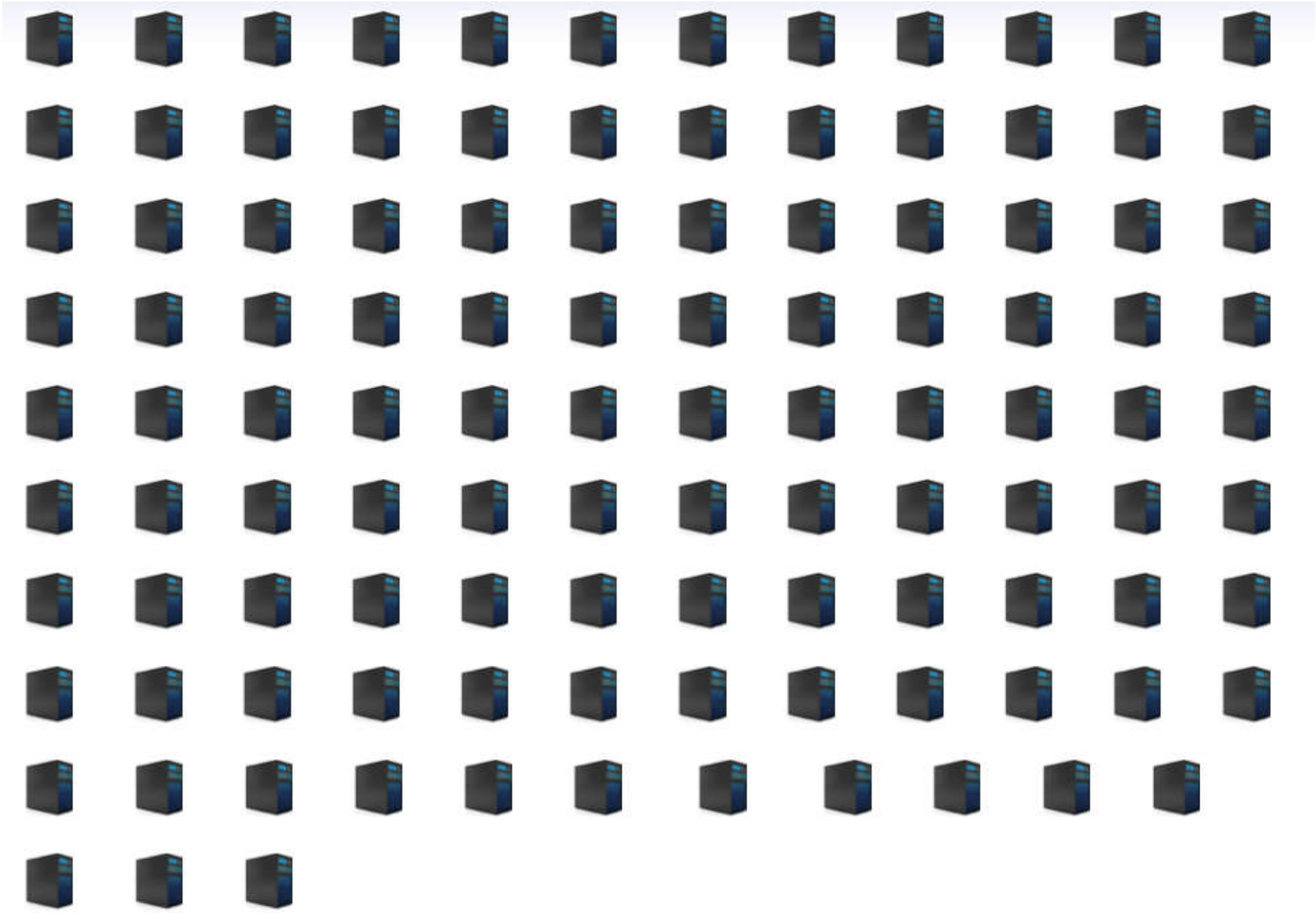
# MPI

**M**essage **P**assing **I**nterface:

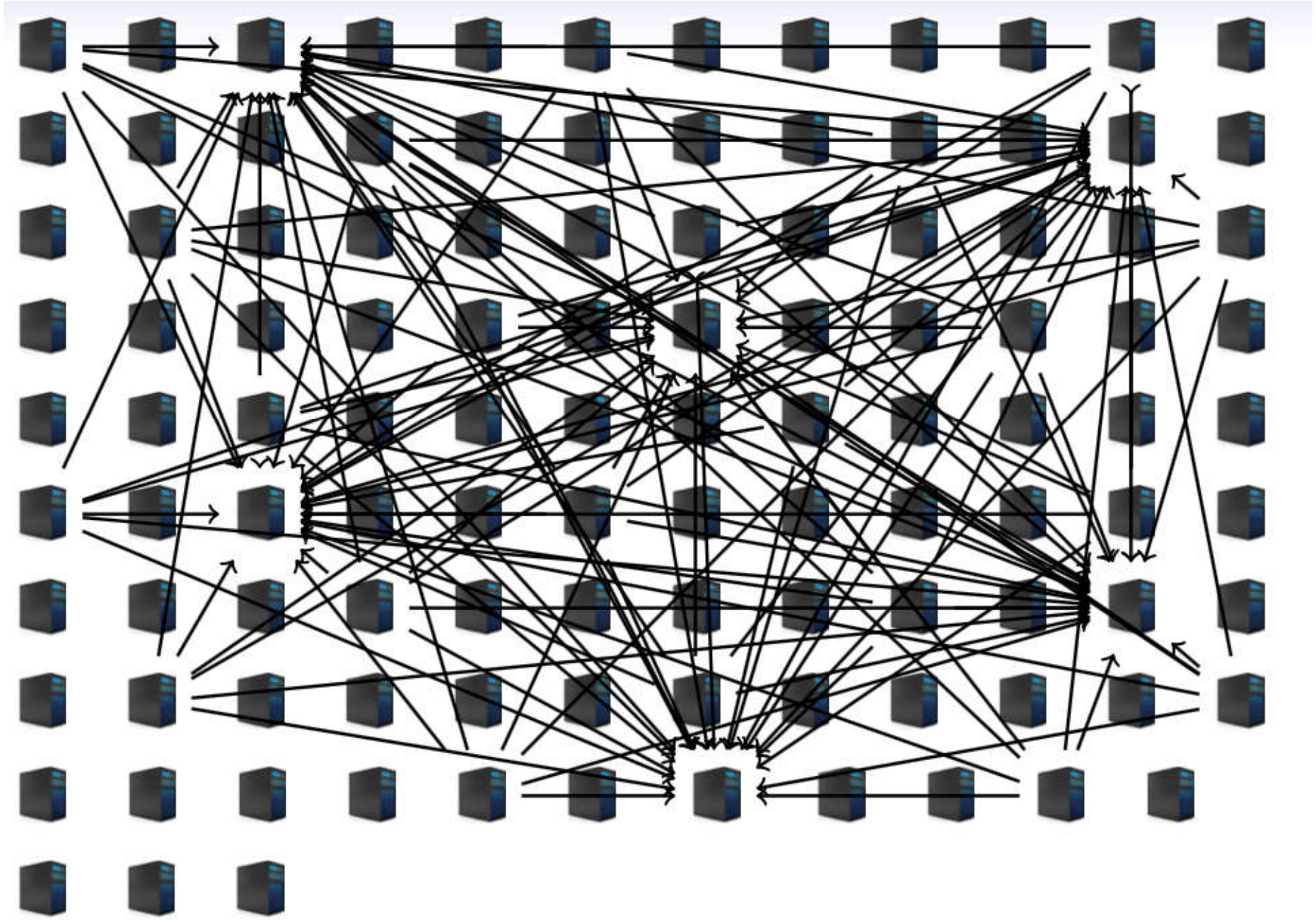


http://mpitutorial.com/

# MPI

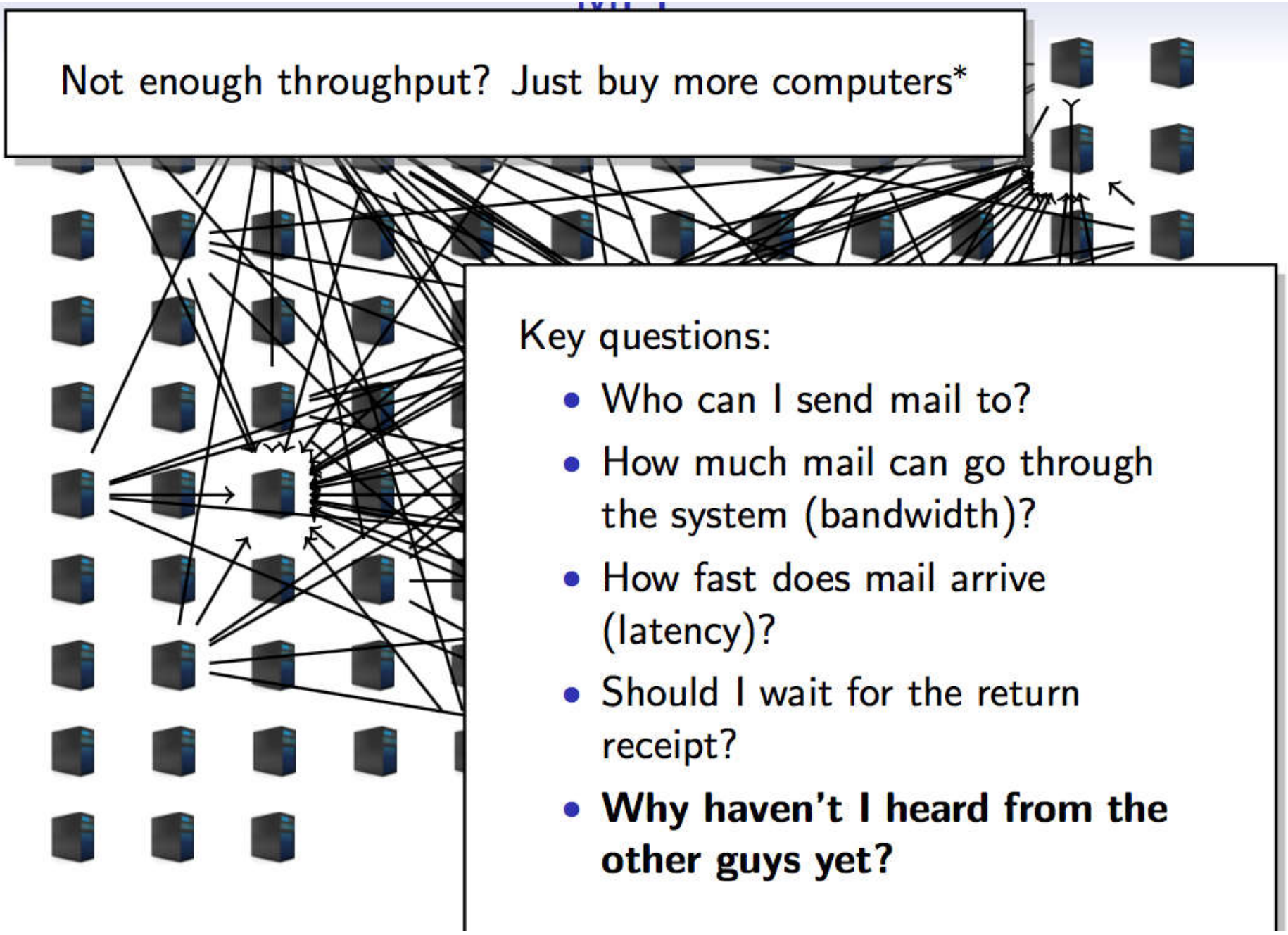


# MPI



# MPI

Not enough throughput? Just buy more computers*



Key questions:

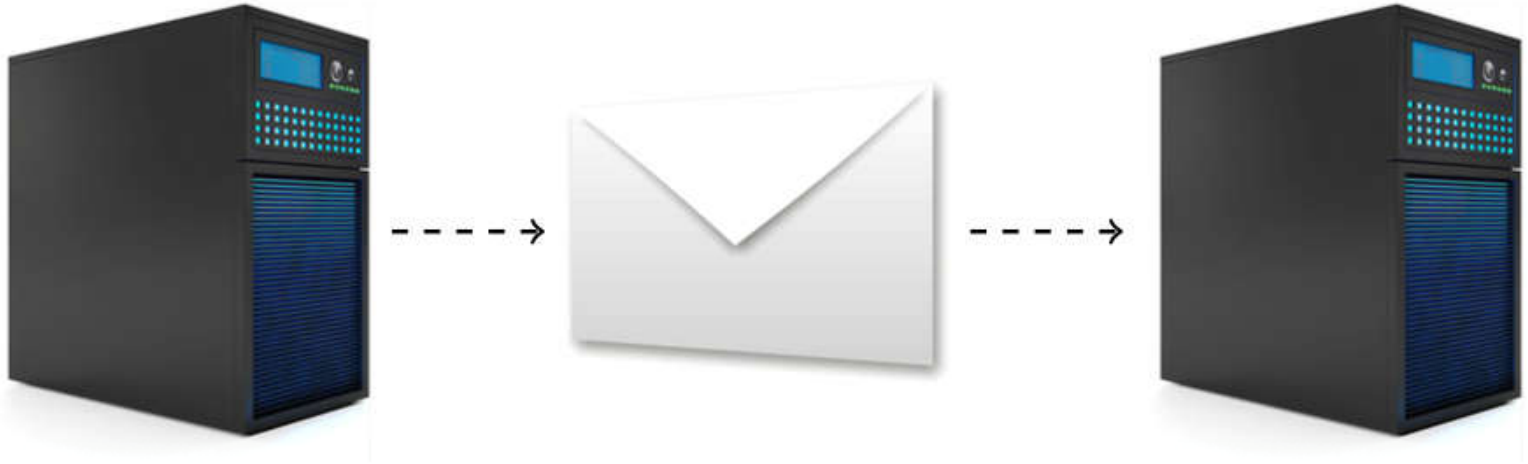
- Who can I send mail to?
- How much mail can go through the system (bandwidth)?
- How fast does mail arrive (latency)?
- Should I wait for the return receipt?
- **Why haven't I heard from the other guys yet?**

# MPI hello world

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

int main(int argc, char ** argv)
{
    int psize;
    int prank;
    MPI_Status status;

    int ierr;

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &prank);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &psize);

    if (prank == 0)
    {
        printf("The number of processes available is %d\n", psize);
    }

    printf("Hello world from process[%d]\n", prank);

    ierr = MPI_Finalize();
    return 0;
}
```

# MPI. Point-to-point communication

**M**essage **P**assing **I**nterface:



```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

```
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

# MPI Datatypes

| MPI datatype | C equivalent |
|---|---|
| MPI_SHORT | short int |
| MPI_INT | int |
| MPI_LONG | long int |
| MPI_LONG_LONG | long long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_UNSIGNED_LONG_LONG | unsigned long long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | char |

# MPI Send&Recv example

```
// Find out rank, size
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int number;
if (world_rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",
           number);
}
```