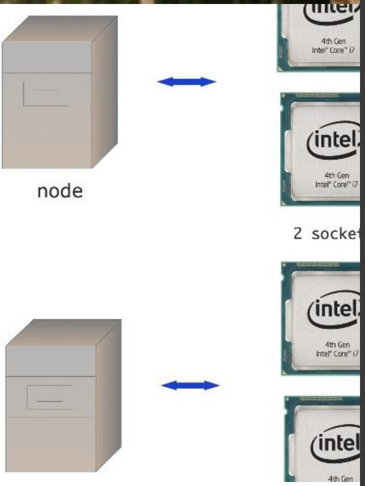
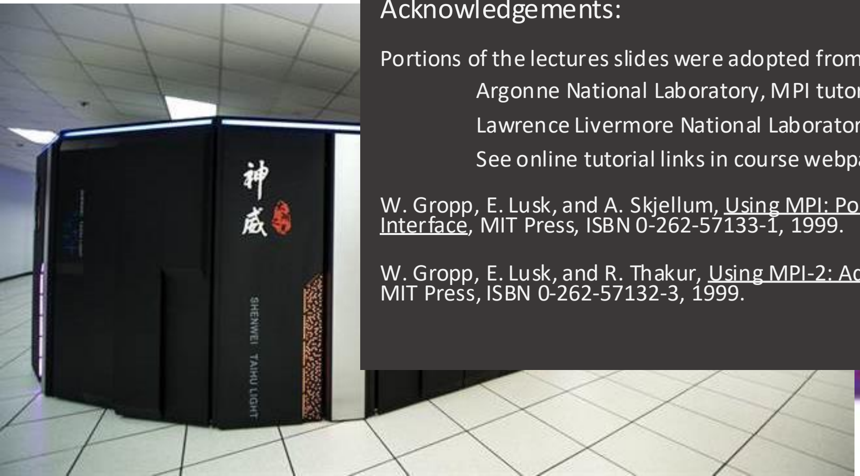
Parallelism at Scale: MPI

cs378h





Outline for Today

- Exam Redux
- Project
- 2PC review
- Scale
- MPI

Acknowledgements:

Portions of the lectures slides were adopted from:

Argonne National Laboratory, MPI tutorials.

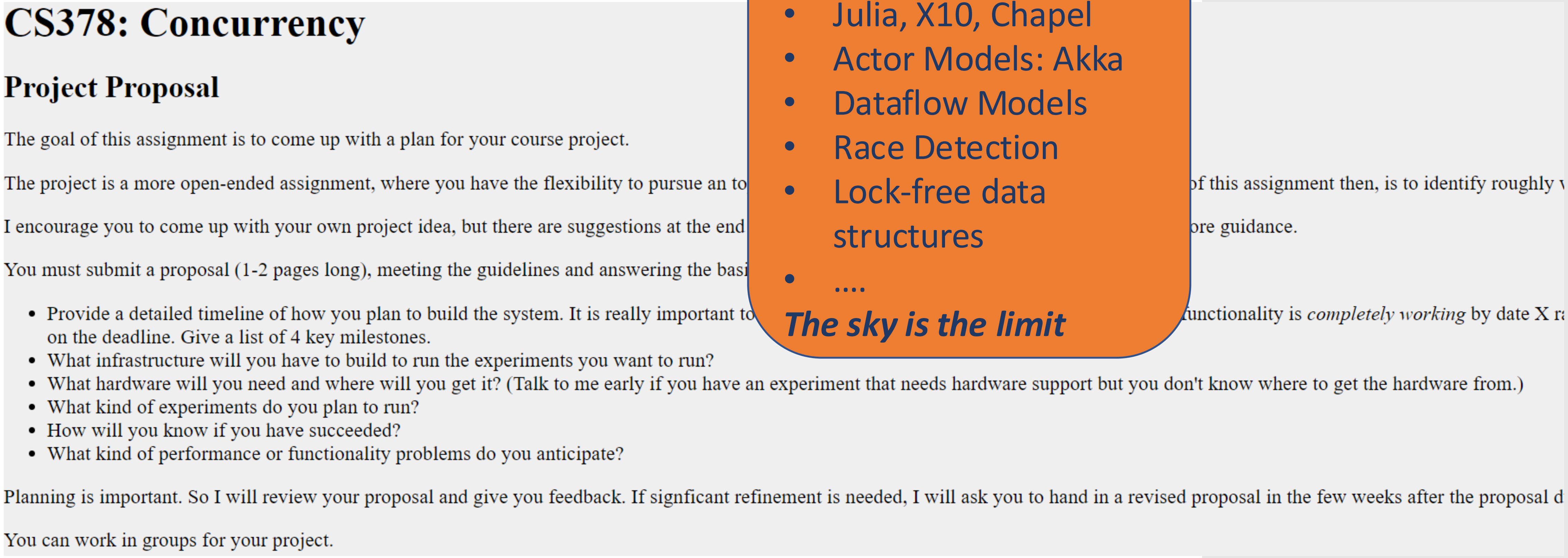
Lawrence Livermore National Laboratory, MPI tutorials

See online tutorial links in course webpage

W. Gropp, E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, MIT Press, ISBN 0-262-57133-1, 1999.

W. Gropp, E. Lusk, and R. Thakur, Using MPI-2: Advanced Features of the Message Passing Interface, MIT Press, ISBN 0-262-57132-3, 1999.

Project Proposal



CS378: Concurrency

Project Proposal

The goal of this assignment is to come up with a plan for your course project.

The project is a more open-ended assignment, where you have the flexibility to pursue an to of this assignment then, is to identify roughly

I encourage you to come up with your own project idea, but there are suggestions at the end ore guidance.

You must submit a proposal (1-2 pages long), meeting the guidelines and answering the basi

- Provide a detailed timeline of how you plan to build the system. It is really important to functionality is *completely working* by date X r on the deadline. Give a list of 4 key milestones.
- What infrastructure will you have to build to run the experiments you want to run?
- What hardware will you need and where will you get it? (Talk to me early if you have an experiment that needs hardware support but you don't know where to get the hardware from.)
- What kind of experiments do you plan to run?
- How will you know if you have succeeded?
- What kind of performance or functionality problems do you anticipate?

Planning is important. So I will review your proposal and give you feedback. If signficant refinement is needed, I will ask you to hand in a revised proposal in the few weeks after the proposal d

You can work in groups for your project.

Ideas:

- Heterogeneity
- Transactional Memory
- Julia, X10, Chapel
- Actor Models: Akka
- Dataflow Models
- Race Detection
- Lock-free data structures
- ….

The sky is the limit

- A very good example

Questions?

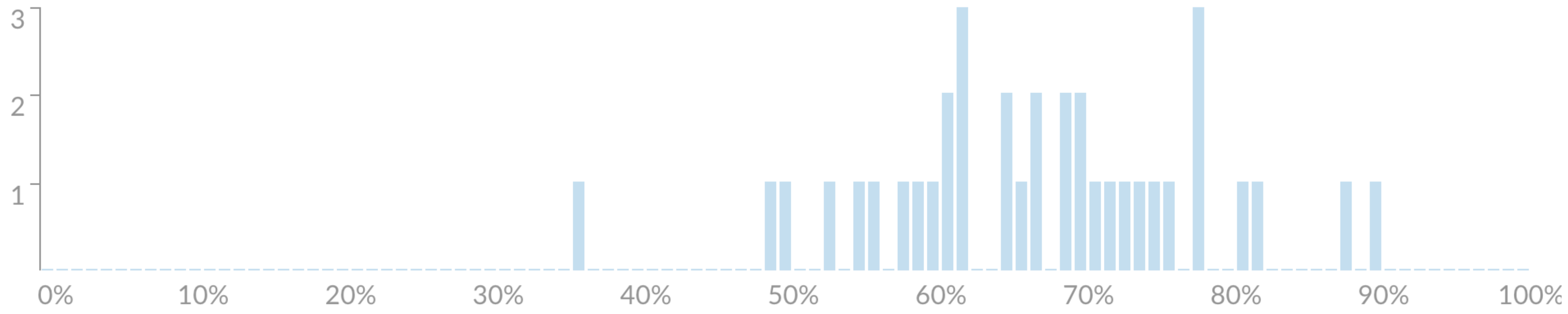
Exam 1 Stats

Average Score
67%

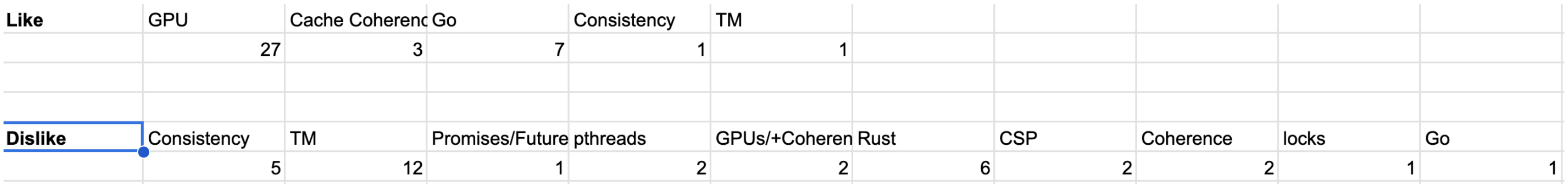
High Score
90%

Low Score
36%

Standard Deviation
11.6



Likes/Dislikes



Like	GPU	Cache Coherenc	Go	Consistency	TM					
	27	3	7	1	1					
Dislike	Consistency	TM	Promises/Future	pthreads	GPUs/+Coheren	Rust	CSP	Coherence	locks	Go
	5	12	1	2	2	6	2	2	1	1

Exam Q*: Uniprocessors/Concurrency

1. In a uniprocessor system concurrency control is best implemented with

   [✓ (a) Semaphores]

   (b) Spinlocks

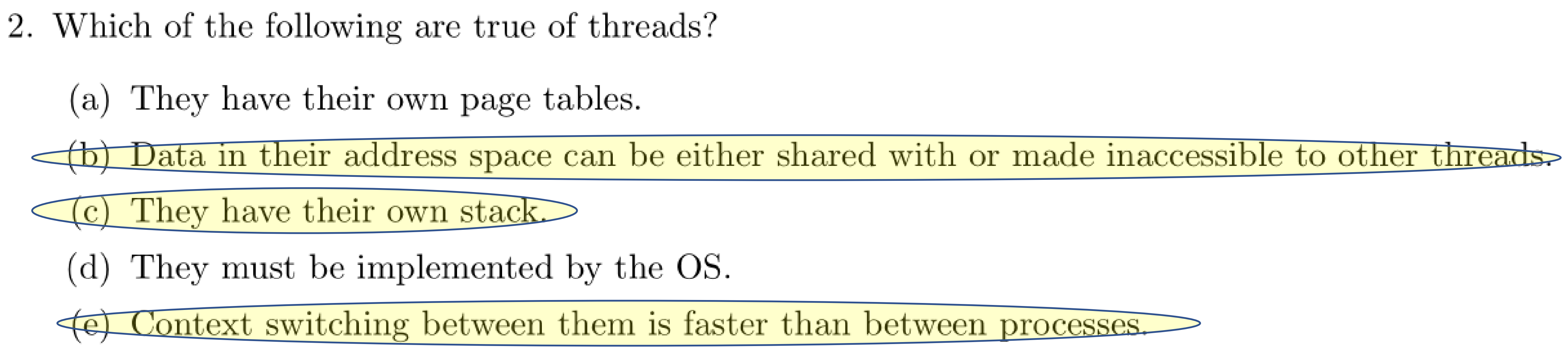
   [✓ (c) Interrupts]

   (d) Atomic instructions

   (e) Bus locking

   (f) Processes and threads

Exam Q*: Threads and Address Spaces



2. Which of the following are true of threads?

   (a) They have their own page tables.

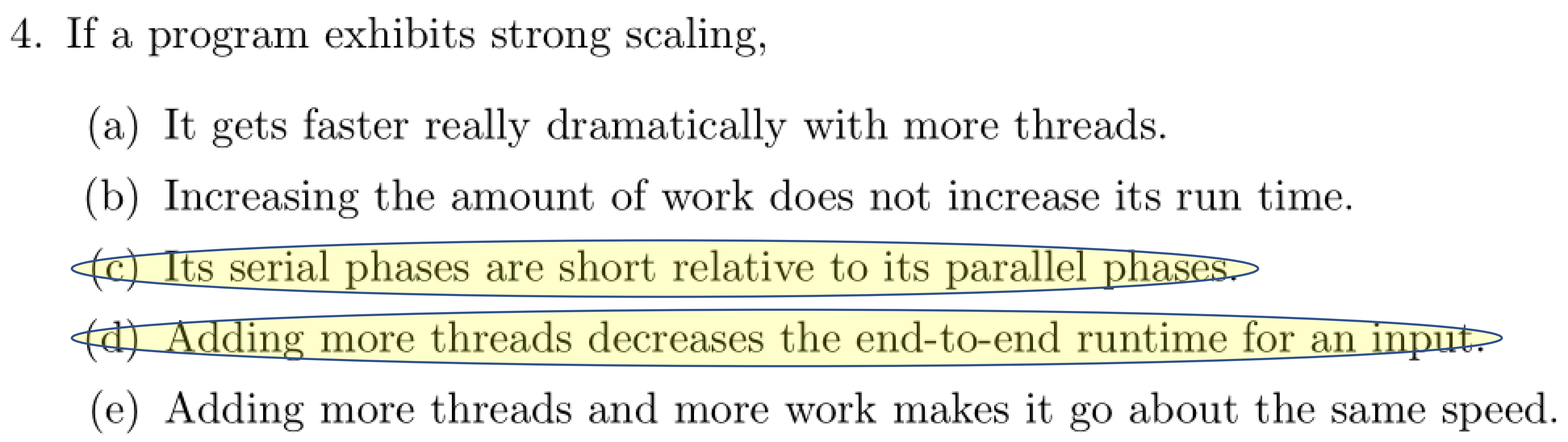
   [✓ (b) Data in their address space can be either shared with or made inaccessible to other threads.]

   [✓ (c) They have their own stack.]

   (d) They must be implemented by the OS.

   [✓ (e) Context switching between them is faster than between processes.]

Exam Q*: Scaling



4. If a program exhibits strong scaling,

   (a) It gets faster really dramatically with more threads.

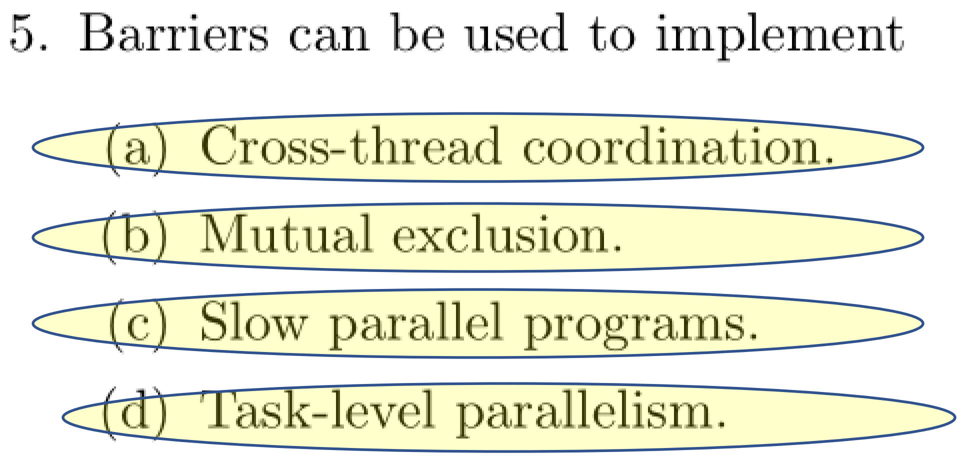
   (b) Increasing the amount of work does not increase its run time.

   [✓ (c) Its serial phases are short relative to its parallel phases.]

   [✓ (d) Adding more threads decreases the end-to-end runtime for an input.]

   (e) Adding more threads and more work makes it go about the same speed.

Exam Q*: Barrier generality



5. Barriers can be used to implement

   [✓ (a) Cross-thread coordination.]

   [✓ (b) Mutual exclusion.]

   [✓ (c) Slow parallel programs.]

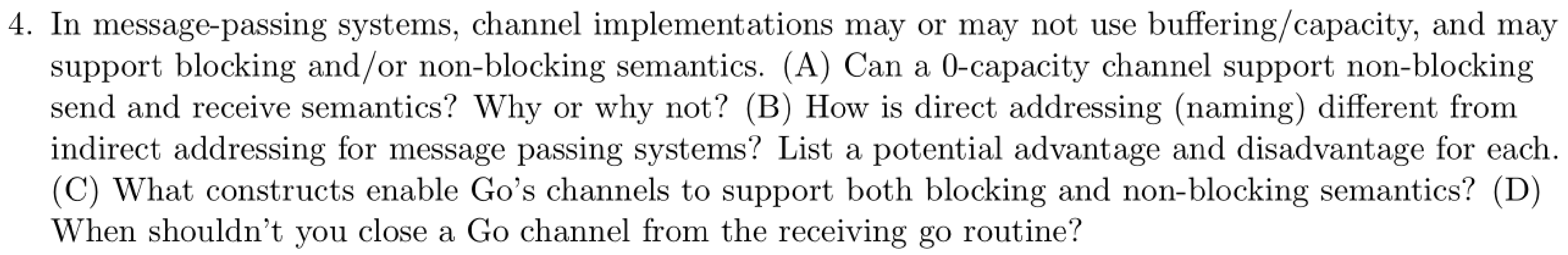
   [✓ (d) Task-level parallelism.]

Exam Q*: Formal properties and TM

Paraphrased: Do <safety, liveness, bounded wait, failure atomicity> suffice to define correctness for TM?

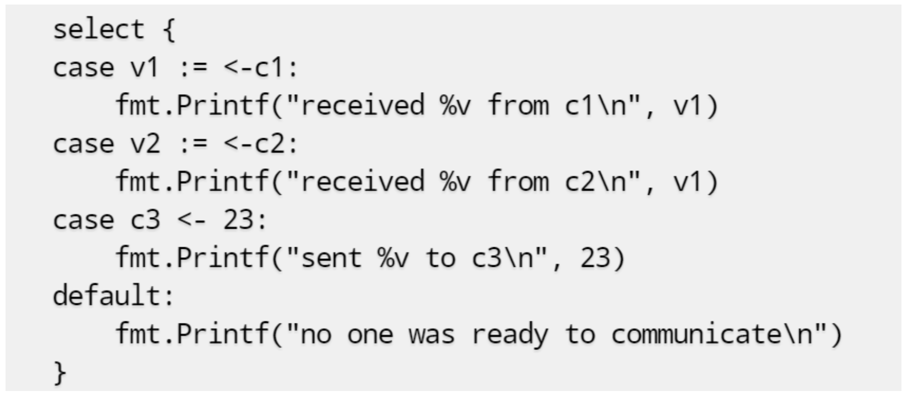
- The point: ***TM can violate single-writer invariant***
- Not the point: ***ACID***

Exam Q*: CSP models and Go



4. In message-passing systems, channel implementations may or may not use buffering/capacity, and may support blocking and/or non-blocking semantics. (A) Can a 0-capacity channel support non-blocking send and receive semantics? Why or why not? (B) How is direct addressing (naming) different from indirect addressing for message passing systems? List a potential advantage and disadvantage for each. (C) What constructs enable Go's channels to support both blocking and non-blocking semantics? (D) When shouldn't you close a Go channel from the receiving go routine?

- A) In general no, but receiver can poll
- C) Select!



```
select {
case v1 := <-c1:
    fmt.Printf("received %v from c1\n", v1)
case v2 := <-c2:
    fmt.Printf("received %v from c2\n", v1)
case c3 <- 23:
    fmt.Printf("sent %v to c3\n", 23)
default:
    fmt.Printf("no one was ready to communicate\n")
}
```

Exam Q: GPUs + Locks + Divergence

```
double atomicAdd(double *data, double val) {

    while(atomicExch(&locked, 1) != 0)
        ;  // spin

    double old = *data;
    *data = old + val;
    locked = 0;
    return old;
}
```

A) divergence

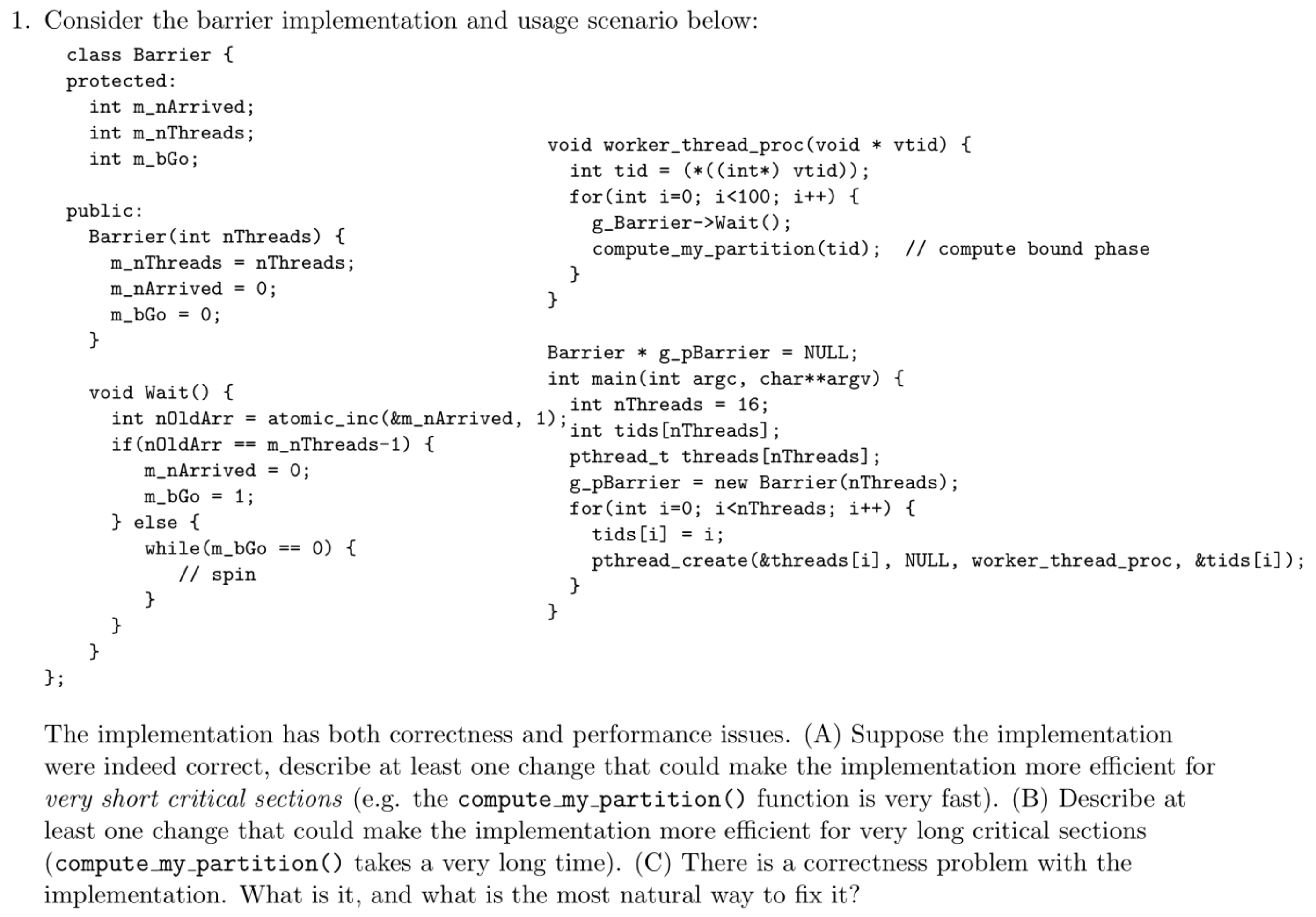
B) at least 1 block, N threads

C) N blocks, 1 thread/block

D) CAS loop is OK,

- *All threads just can't get the lock!*

Exam Q: Barriers



1. Consider the barrier implementation and usage scenario below:

```
class Barrier {
protected:
  int m_nArrived;
  int m_nThreads;
  int m_bGo;

public:
  Barrier(int nThreads) {
    m_nThreads = nThreads;
    m_nArrived = 0;
    m_bGo = 0;
  }

  void Wait() {
    int nOldArr = atomic_inc(&m_nArrived, 1);
    if(nOldArr == m_nThreads-1) {
       m_nArrived = 0;
       m_bGo = 1;
    } else {
       while(m_bGo == 0) {
          // spin
       }
    }
  }
};
```

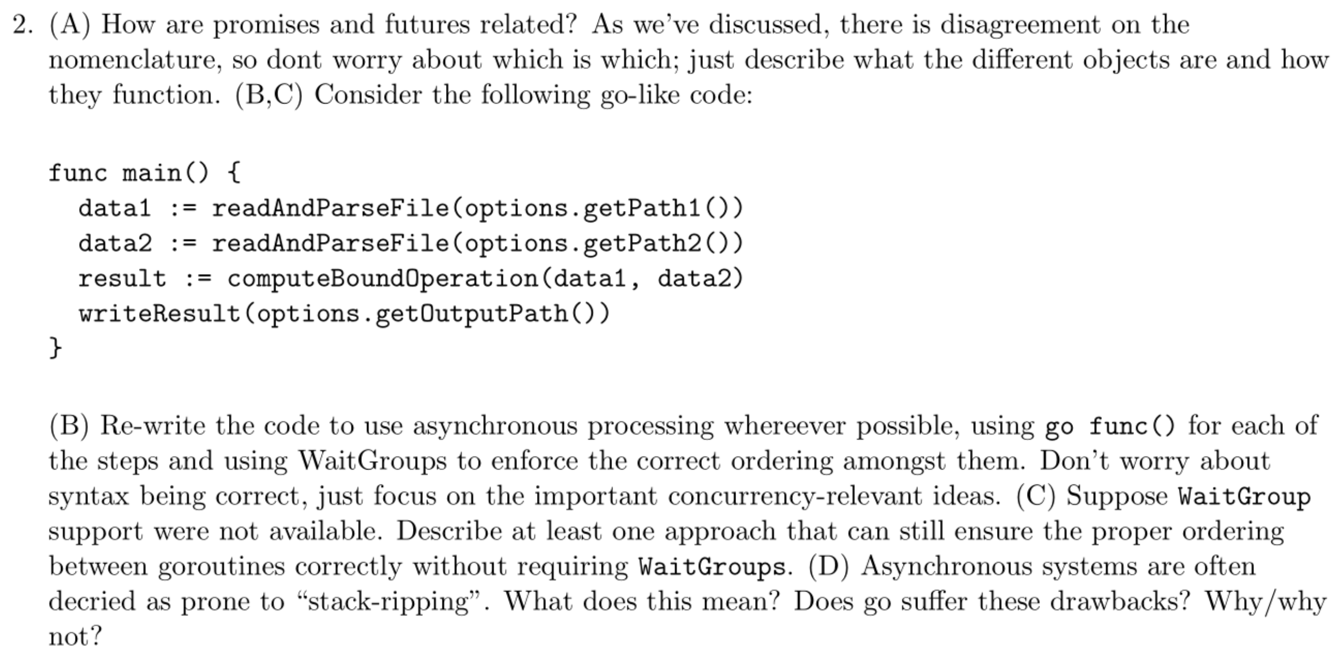
```
void worker_thread_proc(void * vtid) {
  int tid = (*((int*) vtid));
  for(int i=0; i<100; i++) {
    g_Barrier->Wait();
    compute_my_partition(tid);  // compute bound phase
  }
}

Barrier * g_pBarrier = NULL;
int main(int argc, char**argv) {
  int nThreads = 16;
  int tids[nThreads];
  pthread_t threads[nThreads];
  g_pBarrier = new Barrier(nThreads);
  for(int i=0; i<nThreads; i++) {
    tids[i] = i;
    pthread_create(&threads[i], NULL, worker_thread_proc, &tids[i]);
  }
}
```

The implementation has both correctness and performance issues. (A) Suppose the implementation were indeed correct, describe at least one change that could make the implementation more efficient for *very short critical sections* (e.g. the `compute_my_partition()` function is very fast). (B) Describe at least one change that could make the implementation more efficient for very long critical sections (`compute_my_partition()` takes a very long time). (C) There is a correctness problem with the implementation. What is it, and what is the most natural way to fix it?

- A) spin on local go flag
- B) some kind of blocking
- C) barrier doesn't reset (8), some strategy to make it reset (4)

Exam Q*: P+F



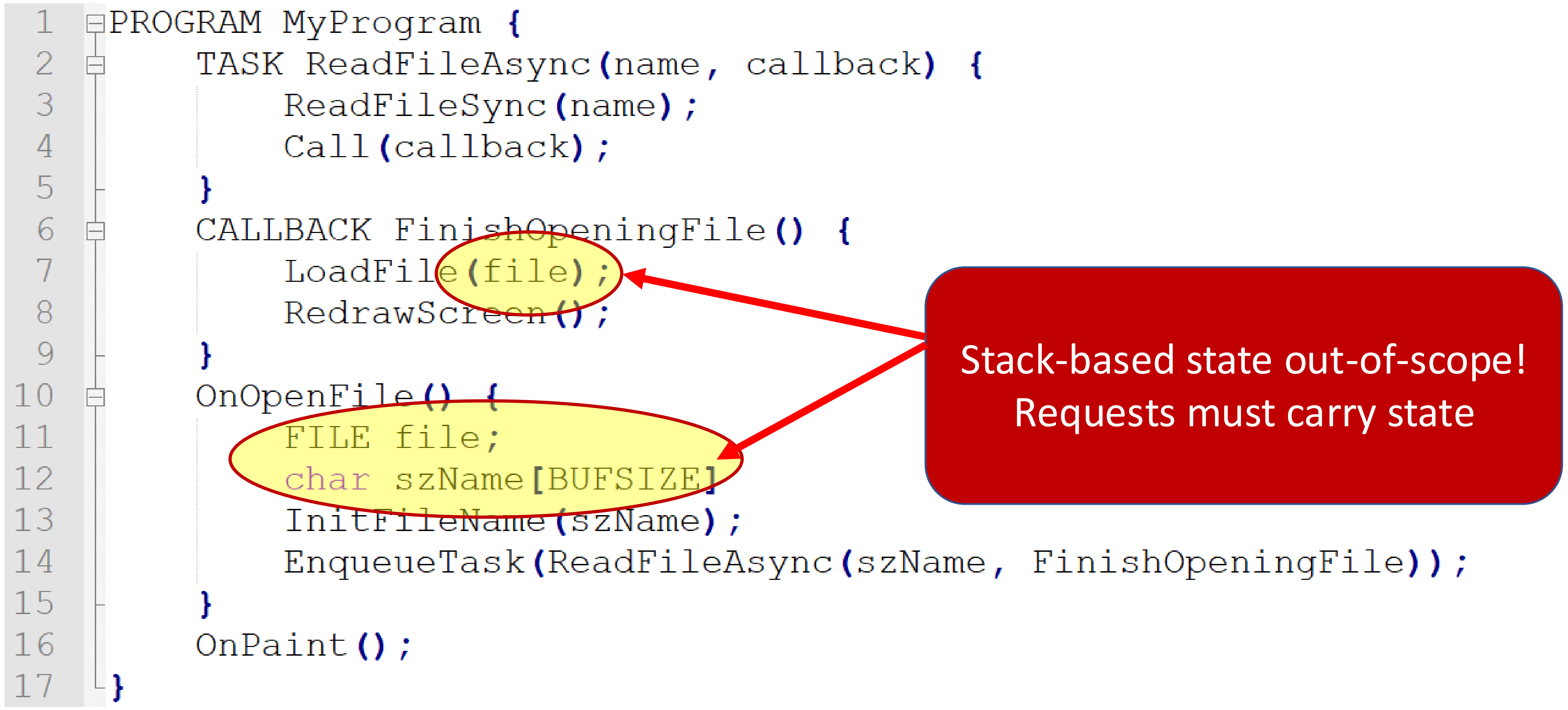
2. (A) How are promises and futures related? As we've discussed, there is disagreement on the nomenclature, so dont worry about which is which; just describe what the different objects are and how they function. (B,C) Consider the following go-like code:

```
func main() {
  data1 := readAndParseFile(options.getPath1())
  data2 := readAndParseFile(options.getPath2())
  result := computeBoundOperation(data1, data2)
  writeResult(options.getOutputPath())
}
```

(B) Re-write the code to use asynchronous processing whereever possible, using `go func()` for each of the steps and using WaitGroups to enforce the correct ordering amongst them. Don't worry about syntax being correct, just focus on the important concurrency-relevant ideas. (C) Suppose `WaitGroup` support were not available. Describe at least one approach that can still ensure the proper ordering between goroutines correctly without requiring `WaitGroups`. (D) Asynchronous systems are often decried as prone to "stack-ripping". What does this mean? Does go suffer these drawbacks? Why/why not?

- A) something about futures and promises
- B) pretty much anything with go func()
- C) Channels!
- D) Stack-ripping → some creative responses
  - (next slide)

Stack-Ripping

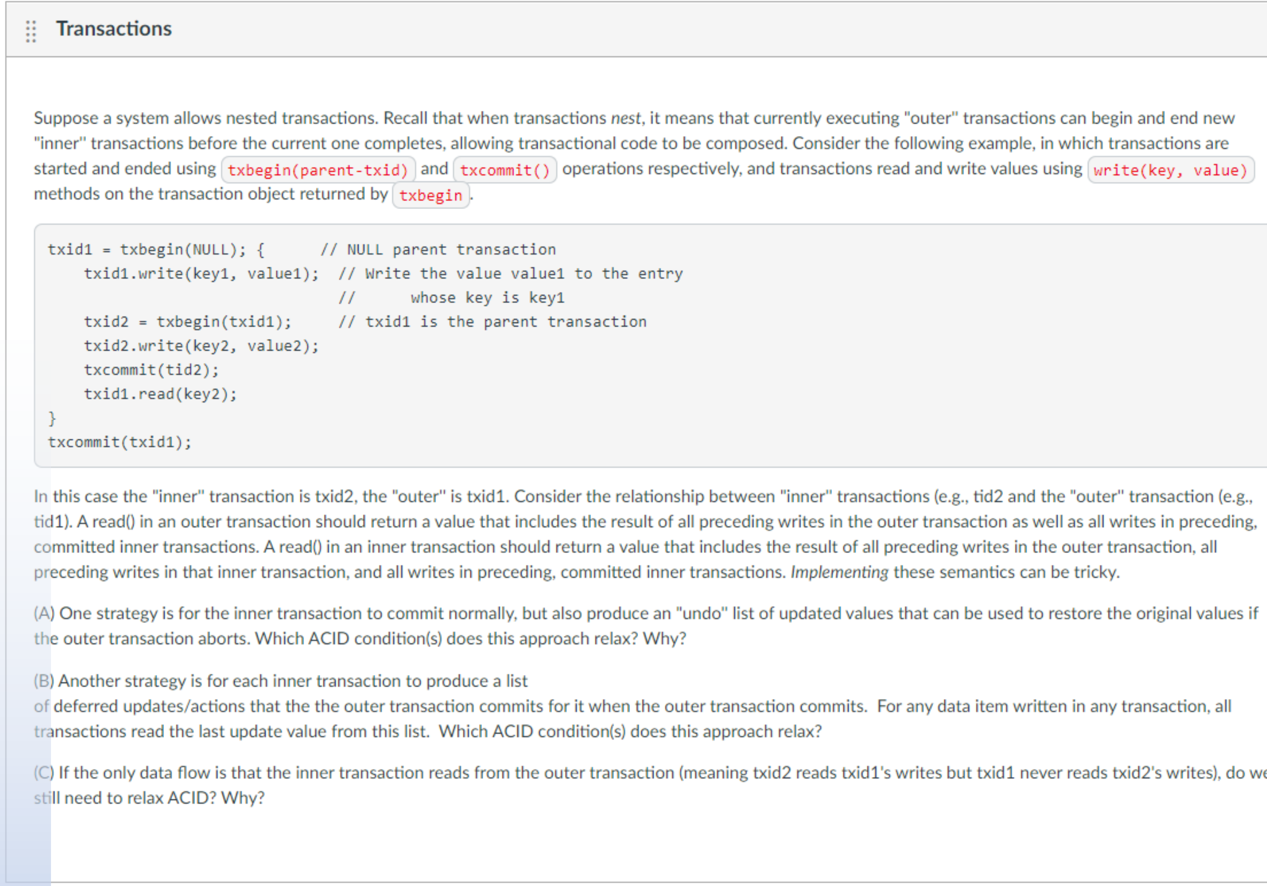


```
PROGRAM MyProgram {
    TASK ReadFileAsync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }
    CALLBACK FinishOpeningFile() {
        LoadFile(file);
        RedrawScreen();
    }
    OnOpenFile() {
        FILE file;
        char szName[BUFSIZE]
        InitFileName(szName);
        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
    }
    OnPaint();
}
```

Stack-based state out-of-scope!
Requests must carry state

Exam Q*: Transactions

- A) Isolation, Atomicity, Durability
  - A) ***I***: other tx see "in-flight" state
  - A) ***A***: some of outer is available without all being available
  - A) ***D***: other tx see state that rolls back
- B) Isolation – all txs see writes of deferred actions (text is subtle)
  - B) Not ***C*** – all txs see writes in order
- C) No relaxation required
  - data only flows outer → inner
  - no uncommitted inner writes observed



Transactions

Suppose a system allows nested transactions. Recall that when transactions *nest*, it means that currently executing "outer" transactions can begin and end new "inner" transactions before the current one completes, allowing transactional code to be composed. Consider the following example, in which transactions are started and ended using `txbegin(parent-txid)` and `txcommit()` operations respectively, and transactions read and write values using `write(key, value)` methods on the transaction object returned by `txbegin`.

```
txid1 = txbegin(NULL); {      // NULL parent transaction
    txid1.write(key1, value1);  // Write the value value1 to the entry
                                //      whose key is key1
    txid2 = txbegin(txid1);     // txid1 is the parent transaction
    txid2.write(key2, value2);
    txcommit(tid2);
    txid1.read(key2);
}
txcommit(txid1);
```

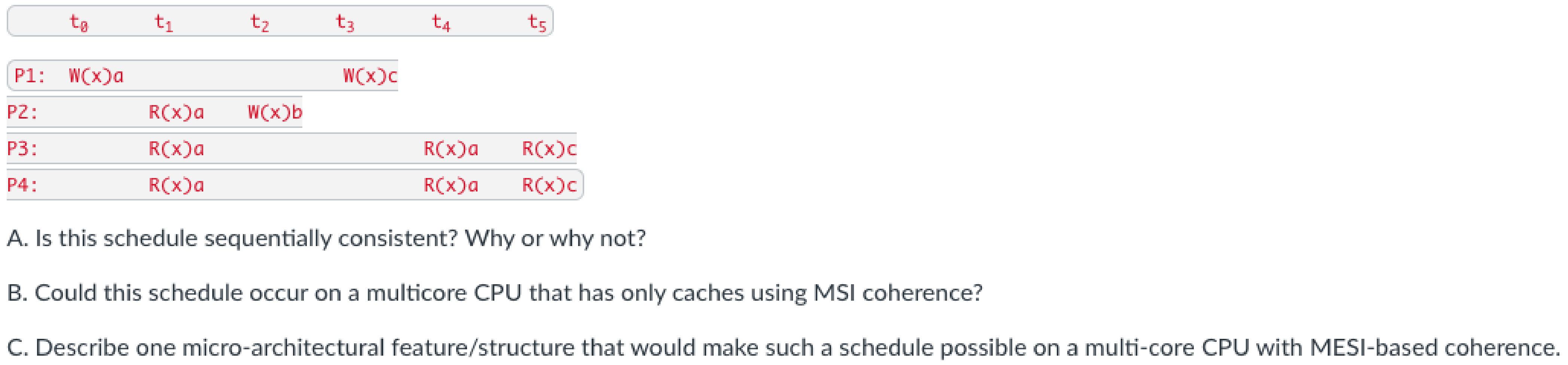
In this case the "inner" transaction is txid2, the "outer" is txid1. Consider the relationship between "inner" transactions (e.g., tid2 and the "outer" transaction (e.g., tid1). A read() in an outer transaction should return a value that includes the result of all preceding writes in the outer transaction as well as all writes in preceding, committed inner transactions. A read() in an inner transaction should return a value that includes the result of all preceding writes in the outer transaction, all preceding writes in that inner transaction, and all writes in preceding, committed inner transactions. *Implementing* these semantics can be tricky.

(A) One strategy is for the inner transaction to commit normally, but also produce an "undo" list of updated values that can be used to restore the original values if the outer transaction aborts. Which ACID condition(s) does this approach relax? Why?

(B) Another strategy is for each inner transaction to produce a list
of deferred updates/actions that the the outer transaction commits for it when the outer transaction commits. For any data item written in any transaction, all transactions read the last update value from this list. Which ACID condition(s) does this approach relax?

(C) If the only data flow is that the inner transaction reads from the outer transaction (meaning txid2 reads txid1's writes but txid1 never reads txid2's writes), do we
still need to relax ACID? Why?

Sequential Consistency



	t_0	t_1	t_2	t_3	t_4	t_5
P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)a	R(x)c
P4:		R(x)a			R(x)a	R(x)c

A. Is this schedule sequentially consistent? Why or why not?

B. Could this schedule occur on a multicore CPU that has only caches using MSI coherence?

C. Describe one micro-architectural feature/structure that would make such a schedule possible on a multi-core CPU with MESI-based coherence.

- A. Yes it is SC, program order, same total order
- B. No, cache-only $\rightarrow$ SC, t4 reads should return c
- C. Store buffers

Two-phase commit

- N participants agree or don’t (atomicity)
- Phase 1: everyone “prepares”
- Phase 2: Master decides and tells everyone to actually commit
- What if the master crashes in the middle?

2PC: Phase 1

1. Coordinator sends REQUEST to all participants
2. Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT to local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Example—move: C→S1: delete foo from /, C→S2: add foo to /

Failure case:
S1 writes rm /foo, VOTE_COMMIT to log
S1 sends VOTE_COMMIT
S2 decides permission problem
S2 writes/sends VOTE_ABORT

Success case:
S1 writes rm /foo, VOTE_COMMIT to log
S1 sends VOTE_COMMIT
S2 writes add foo to /
S2 writes/sends VOTE_COMMIT

2PC: Phase 2

- Case 1: receive VOTE_ABORT or timeout
  - Write GLOBAL_ABORT to log
  - send GLOBAL_ABORT to participants
- Case 2: receive VOTE_COMMIT from all
  - Write GLOBAL_COMMIT to log
  - send GLOBAL_COMMIT to participants
- Participants receive decision, write GLOBAL_* to log

2PC corner cases

Phase 1

1. Coordinator sends REQUEST to all participants
2. X Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT to local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Phase 2

- Y Case 1: receive VOTE_ABORT or timeout
  - Write GLOBAL_ABORT to log
  - send GLOBAL_ABORT to participants
- Case 2: receive VOTE_COMMIT from all
  - W Write GLOBAL_COMMIT to log
  - send GLOBAL_COMMIT to participants
- Z Participants recv decision, write GLOBAL_* to log

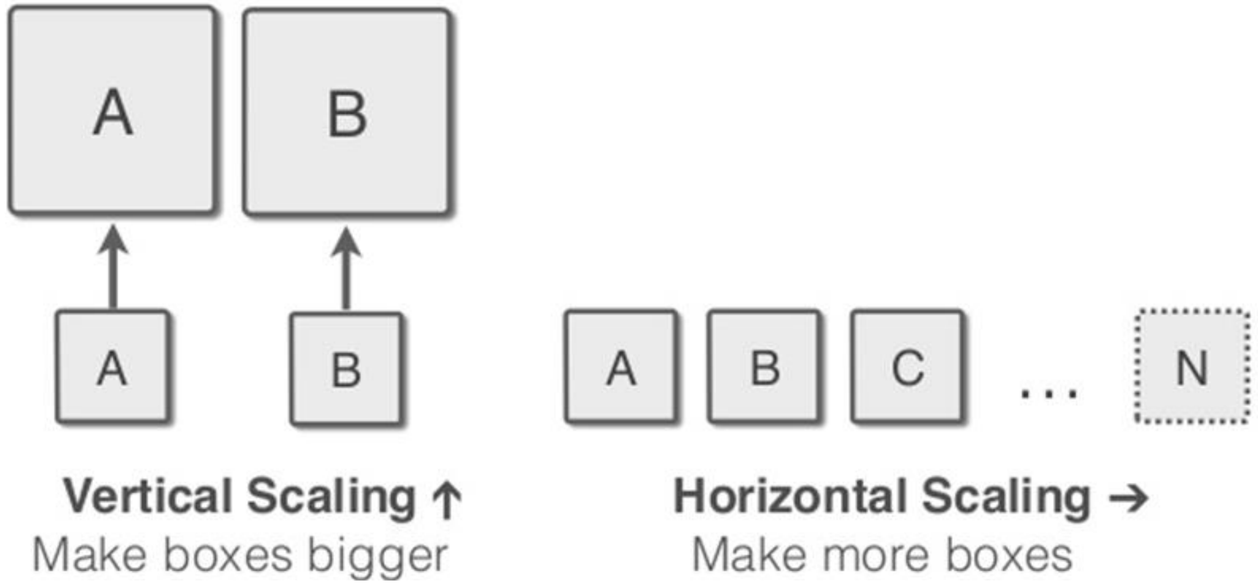
- What if participant crashes at X?
- Coordinator crashes at Y?
- Participant crashes at Z?
- Coordinator crashes at W?

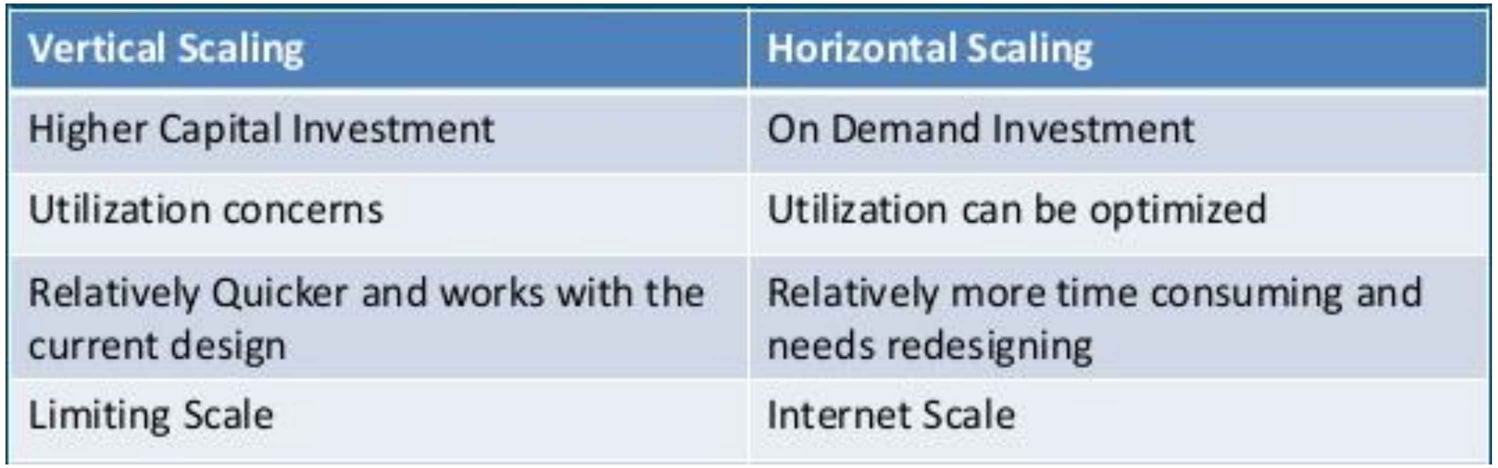
2PC limitation(s)

- Coordinator crashes at W, never wakes up
- All nodes block forever!
- Can participants ask each other what happened?
- 2PC: always has risk of indefinite blocking
- Solution: (yes) 3 phase commit!
  - Reliable replacement of crashed “leader”
  - 2PC often good enough in practice

Questions?

Scale Out vs Scale Up





Vertical Scaling	Horizontal Scaling
Higher Capital Investment	On Demand Investment
Utilization concerns	Utilization can be optimized
Relatively Quicker and works with the current design	Relatively more time consuming and needs redesigning
Limiting Scale	Internet Scale

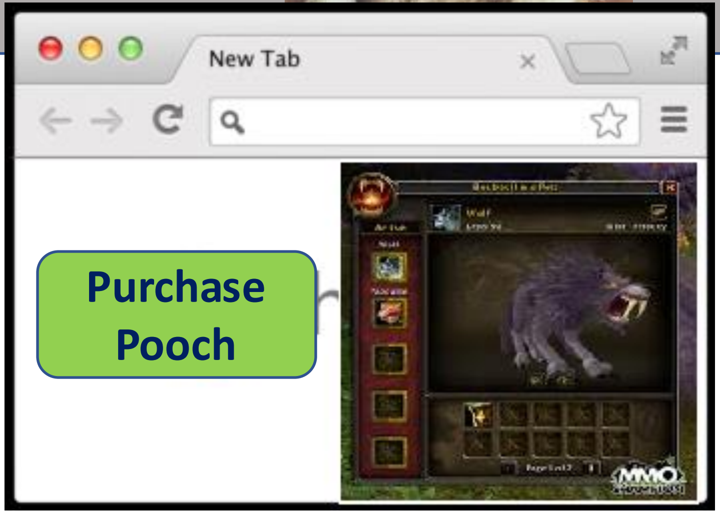
Parallel Systems Architects Wanted

Hot Startup Idea:

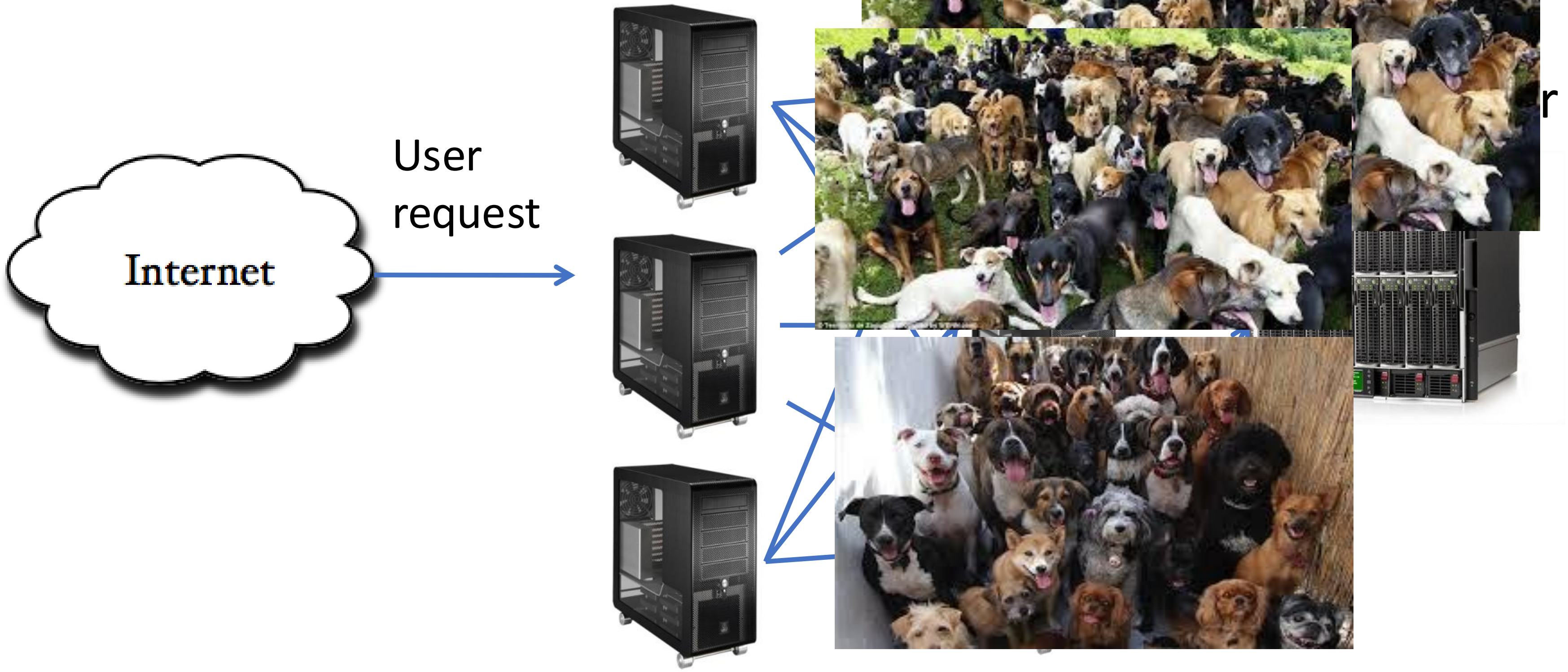
www.purchase-a-pooch.biz

lete request

How to handle lots and lots of dogs?



3 Tier architecture



Web Servers (Presentation Tier) and App se

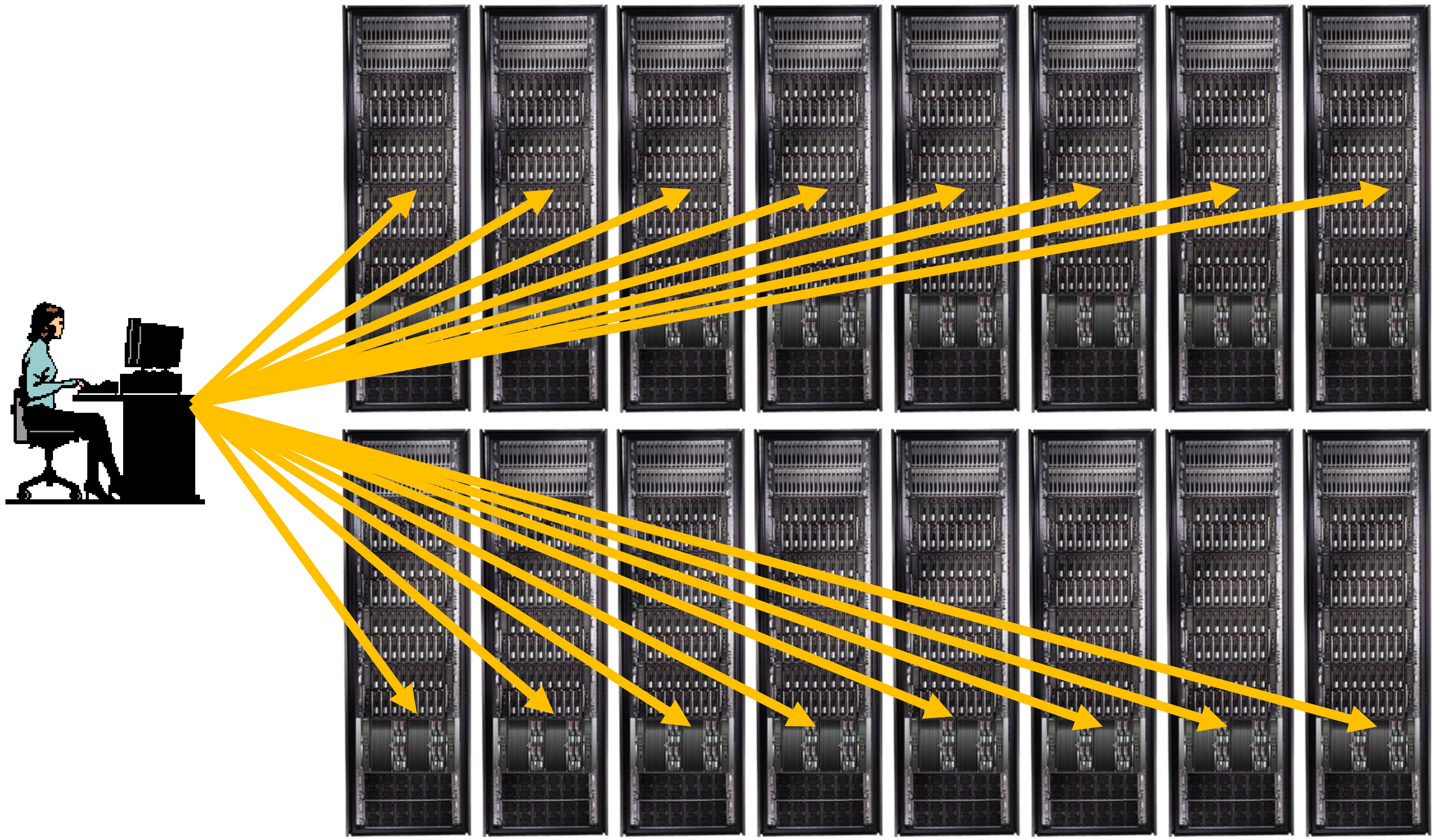
Database Server → scales *vertically*

Horizontal Scale → "Shared Nothing"

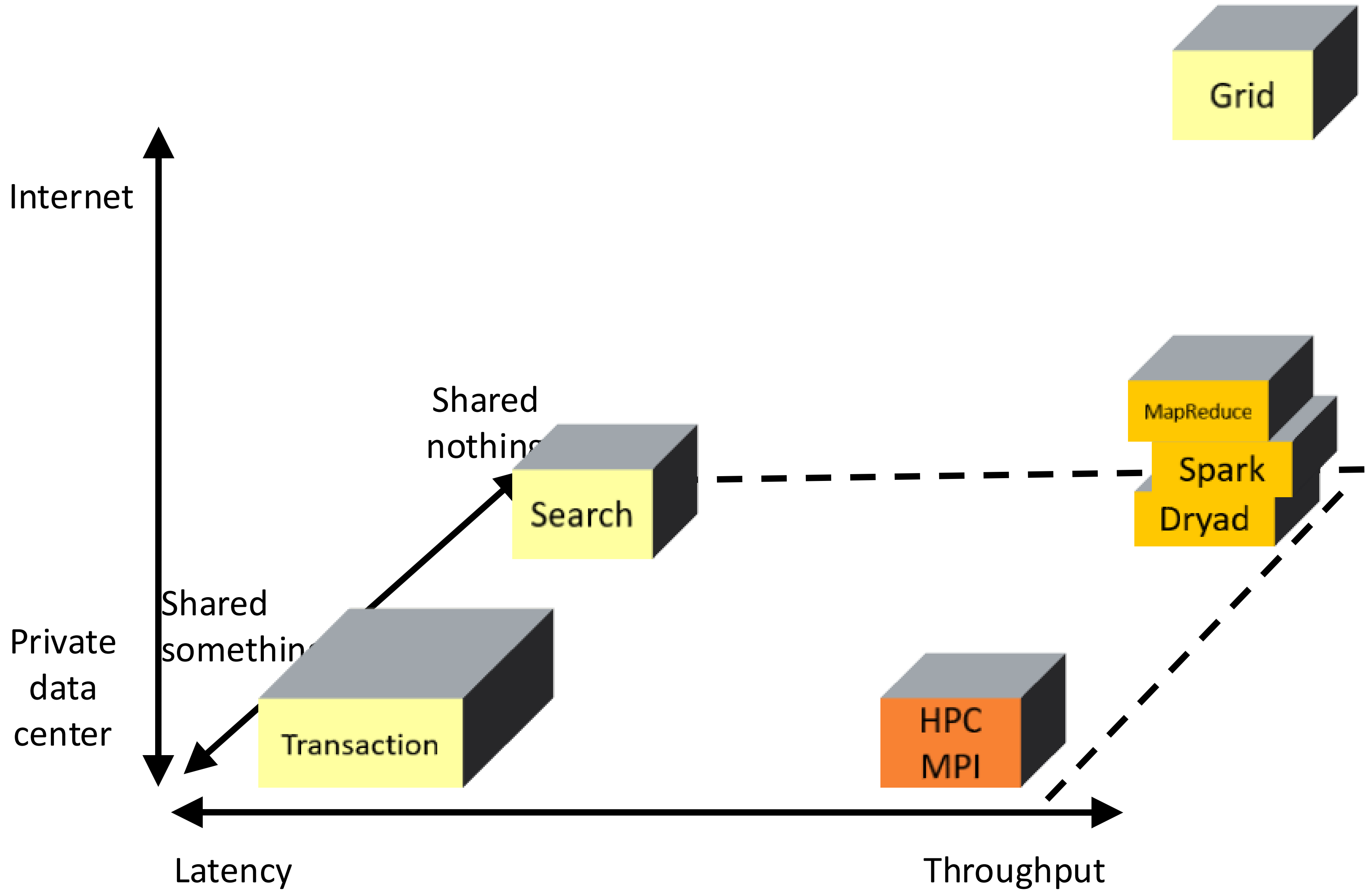
Why is this a good arrangement?

Vertical scale gets you a long way, but there is always a bigger problem size

Horizontal Scale: Goal



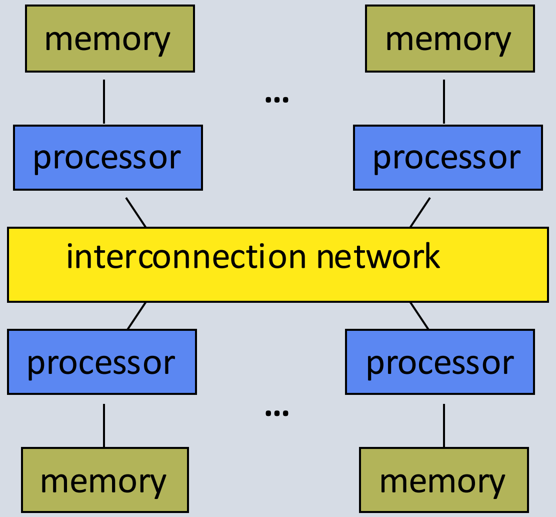
Design Space



Parallel Architectures and MPI

Distributed Memory Multiprocessor

Messaging between nodes

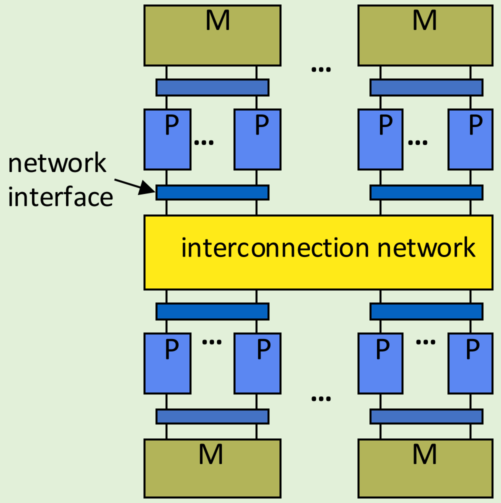


Massively Parallel Processor (MPP)

Many, many processors

Cluster of SMPs

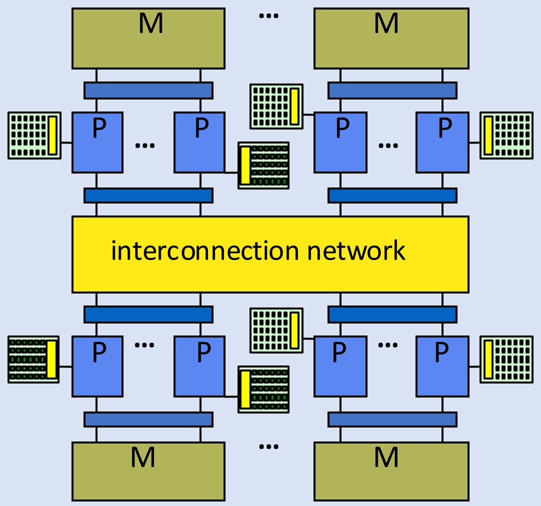
- Shared memory in SMP node
- Messaging ←→ SMP nodes



- also regarded as MPP if processor # is large

Multicore SMP+GPU Cluster

- Shared mem in SMP node
- Messaging between nodes



- GPU accelerators attached

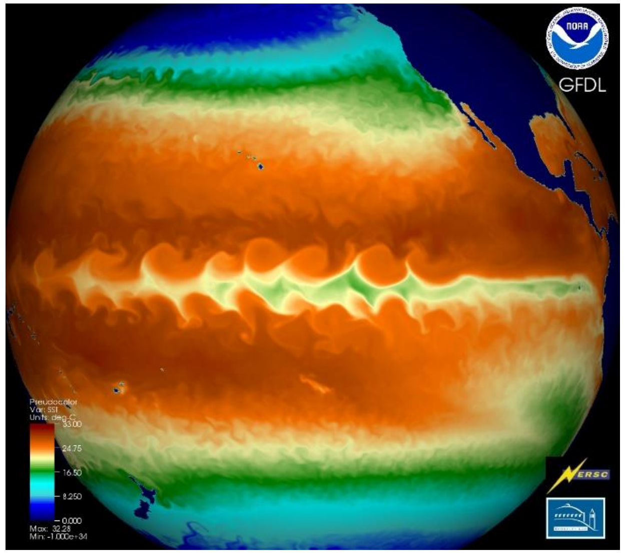
What requires extreme scale?

Simulations—why?

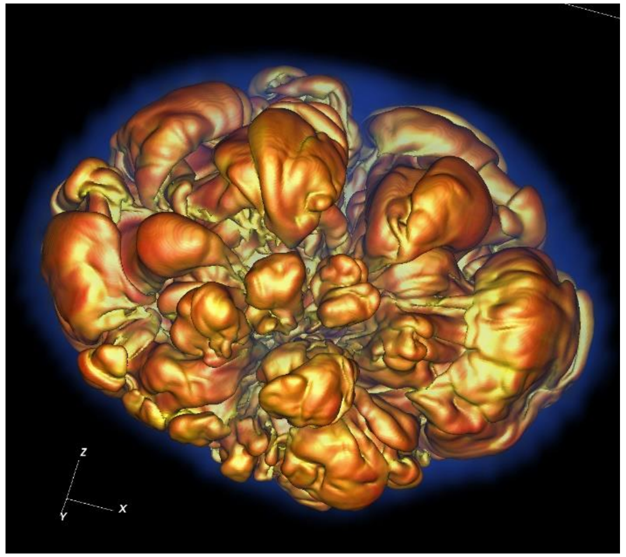
- Simulations are sometimes more cost effective than experiments

Why extreme scale?

- More compute cycles, more memory, etc, lead for faster and/or more accurate simulations



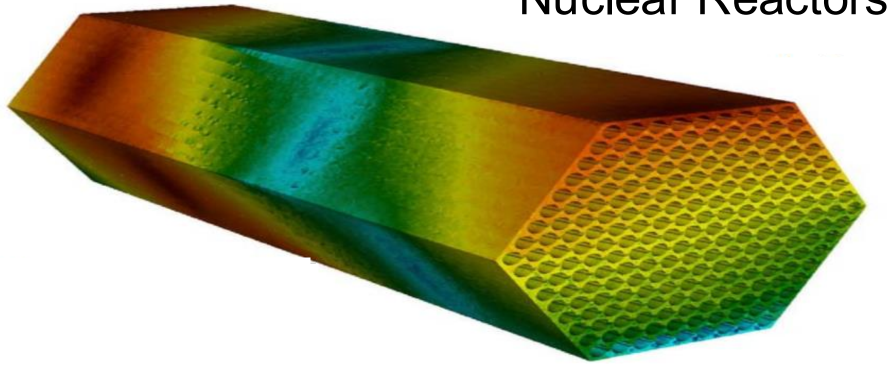
Climate Change



Astrophysics

Image credit: Prabhat, LBNL

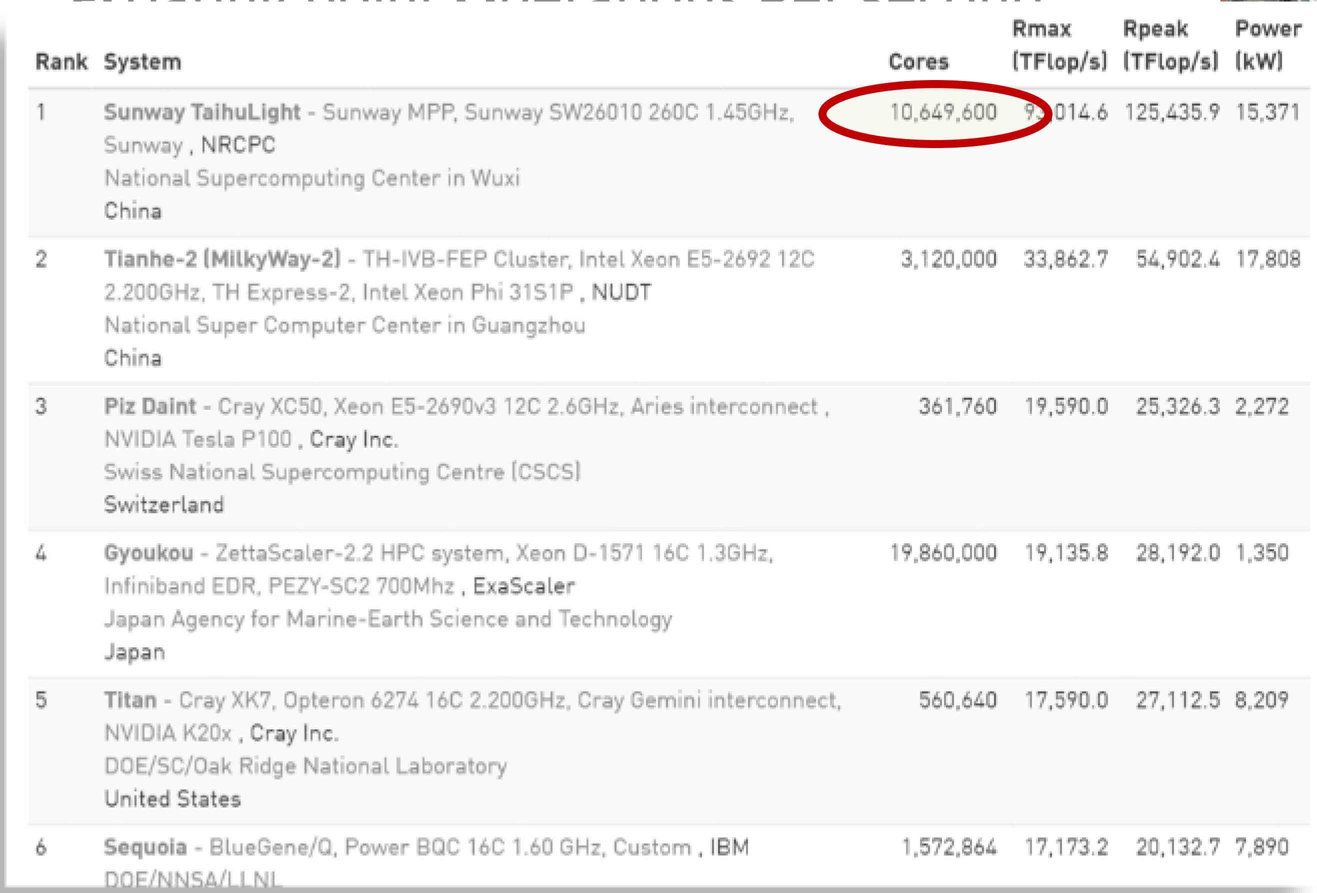
Nuclear Reactors



How big is “extreme” scale?

Measured in FLOPs

Floating point Operations Per second

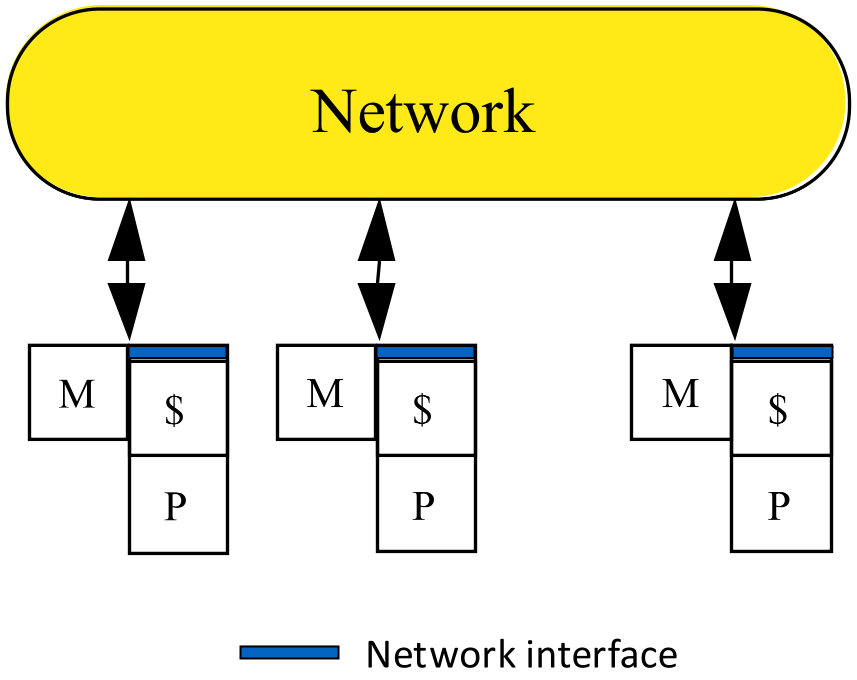


Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
2	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P , NUDT National Super Computer Center in Guangzhou China	3,120,000	33,862.7	54,902.4	17,808
3	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	361,760	19,590.0	25,326.3	2,272
4	**Gyoukou** - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz , ExaScaler Japan Agency for Marine-Earth Science and Technology Japan	19,860,000	19,135.8	28,192.0	1,350
5	**Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc. DOE/SC/Oak Ridge National Laboratory United States	560,640	17,590.0	27,112.5	8,209
6	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom , IBM DOE/NNSA/LLNL	1,572,864	17,173.2	20,132.7	7,890

Performance of over 10 Peta
floating point number operations per second
(10 Peta=10,000,000,000,000,000)
K computer
IKEN K / Kei computer
4 on Top500.org, 10PFLOPs

Distributed Memory Multiprocessors

- Each processor has a local memory
  - Physically separated address space
- Processors communicate to access non-local data
  - Message communication
    - *Message passing architecture*
  - Processor interconnection network
- Parallel applications partitioned across
  - Processors: execution units
  - Memory: data partitioning
- Scalable architecture
  - Incremental cost to add hardware (cost of node)



- Nodes: complete computer
  - Including I/O
- Nodes communicate via network
  - Standard networks (IP)
  - Specialized networks (RDMA, fiber)

Performance: Latency and Bandwidth

Bandwidth

- Need high bandwidth in communication
- Match limits in network, memory, and proce
- Network interface speed vs. network bisecti bandwidth

Latency

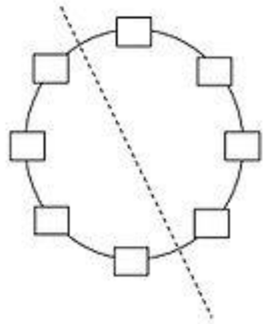
- Performance affected: processor may have to wait
- Hard to overlap communication and computation
- Overhead to communicate: a problem in many machines

Latency hiding

- Increases programming system burden
- E.g.: communication/computation overlap, prefetch

Wait…bisection bandwidth?

if network is **bisected**, **bisection bandwidth == bandwidth** between the two partitions



Is this different from metrics we've cared about so far?

Ostensible Advantages of Distributed Memory Architectures

Hardware simpler (especially versus NUMA), more scalable

Communication explicit, simpler to understand

Explicit communication →

focus attention on costly aspect of parallel computation

Synchronization →

naturally associated with sending messages

reduces possibility for errors from incorrect synchronization

Easier to use sender-initiated communication →

some advantages in performance

Can you think of any *disadvantages*?

Running on Supercomputers

- Programmer plans a ***job***; job ==
  - parallel binary program
  - "input deck" (specifies input data)
- Submit job to a ***queue***
- Scheduler allocates resources when
  - resources are available,
  - (or) the job is deemed "high priority"

Sometimes 1 job takes whole machine

These are called "hero runs"…

Sometimes many smaller jobs

Supercomputers used continuously

Processors: "scarce resource"

jobs are "plentiful"

- Scheduler runs scripts that initialize the environment
  - Typically done with environment variables
- At the end of initialization, it is possible to infer:
  - What the desired job configuration is (i.e., how many tasks per node)
  - What other nodes are involved
  - How your node's tasks relates to the overall program
- MPI library interprets this information, hides the details

The Message-Passing Model

Process: a program counter and address space

Proc

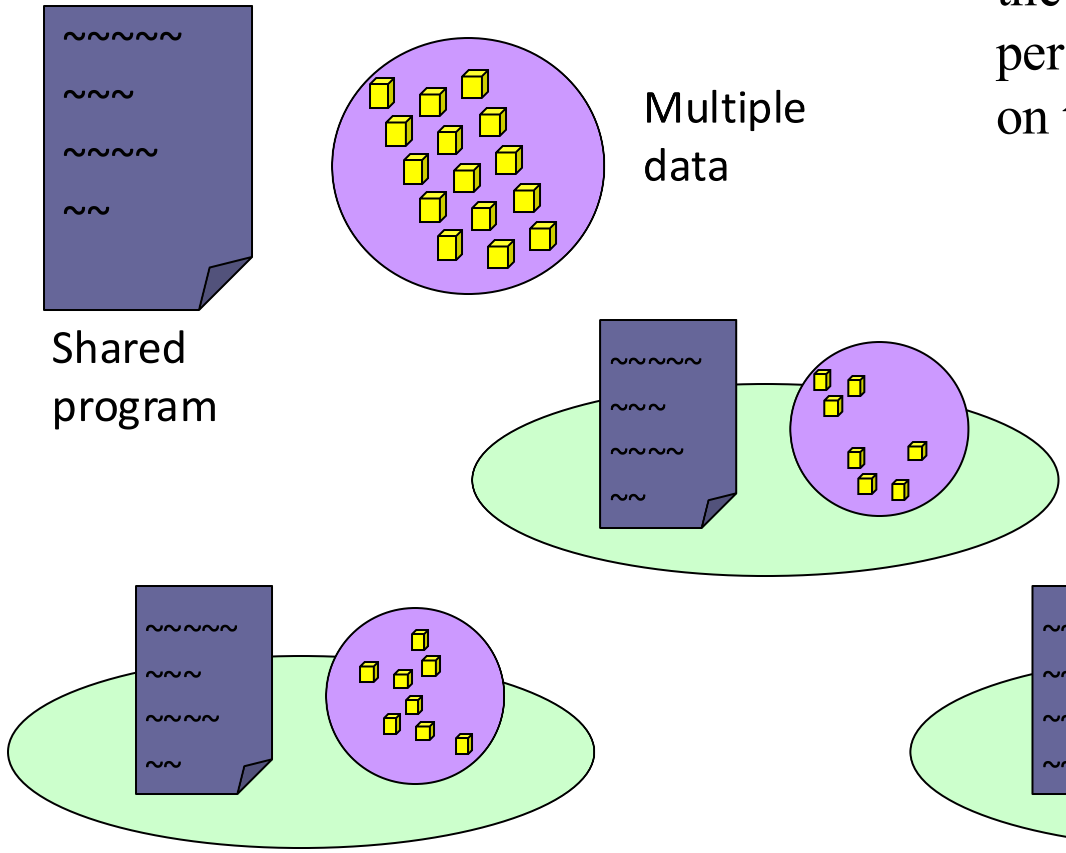
MPI

Inte

- MPI == ***Message-Passing Interface specification***
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
- Specified in C, C++, Fortran 77, F90
- Message Passing Interface (MPI) Forum
  - http://www.mpi-forum.org/
  - http://www.mpi-forum.org/docs/docs.html

- Two flavors for communication
  - Cooperative operations
  - One-sided operations

SPMD

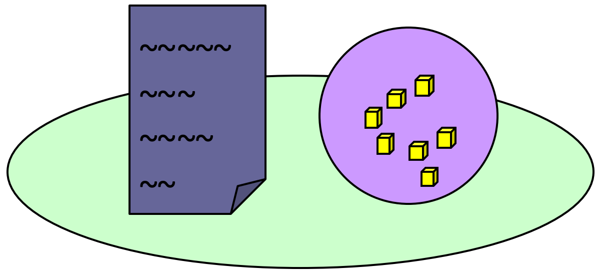


~~~~~
~~~
~~~~
~~
Multiple data
Shared program

"Owner compute" rule:
Process that "owns" the data (local data) performs computations on that data

Data distributed across processes

Not shared → shared nothing


~~~~~

Cooperative Operations

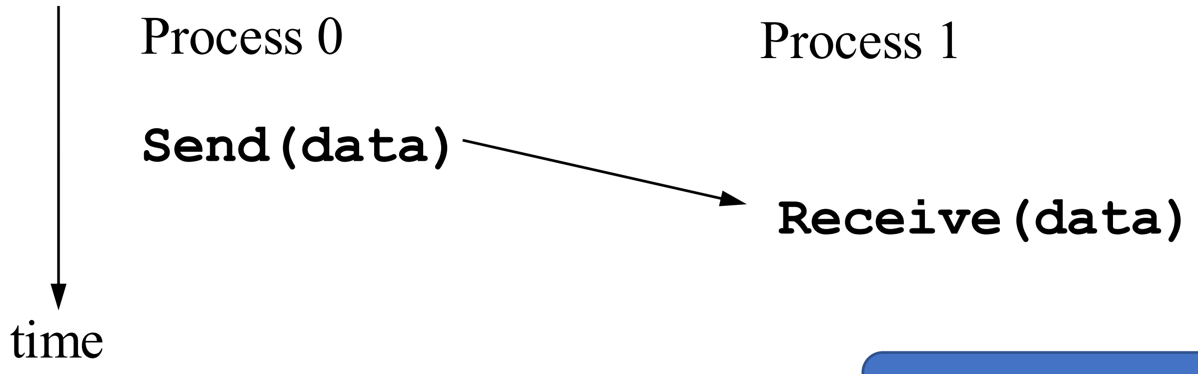
Data is cooperatively exchanged in message-passing

Explicitly sent by one process and received by another

Advantage of local control of memory

- Change in the receiving process's memory made with receiver's explicit participation

Communication and synchronization are combined



Familiar argument?

One-Sided Operations

Are 1-sided operations better for performance?

One-sided operations between processes

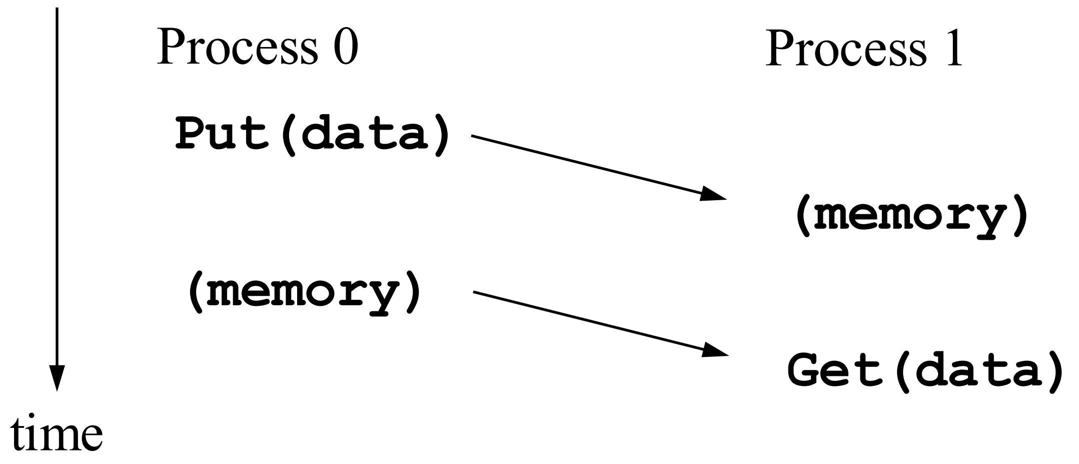
- Include remote memory reads and writes

Only one process needs to explicitly participate

- There is still agreement implicit in the SPMD program

Implication:

- Communication and synchronization are decoupled



A Simple MPI Program

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

MPI_Init

Hardware resources allocated

- MPI-managed ones anyway…

Start processes on different nodes

- Where does their executable program come from?

Give processes what they need to know

- Wait…what do they need to know?

Configure OS-level resources

Configure tools that are running with MPI

…

MPI_Finalize

Executive Summary

- Undo all of init
- Be able to do it on success or failure exit

Why do we need to finalize MPI?

What is necessary for a “graceful” MPI exit?

Can bad things happen otherwise?

Suppose one process exits…

How do resources get de-allocated?

How to shut down communication?

What type of exit protocol might be used?

- By default, an error causes all processes to abort
- The user can cause routines to return (with an error code)
  - In C++, exceptions are thrown (MPI-2)
- A user can also write and install custom error handlers
- Libraries may handle errors differently from applications

Running MPI Programs

MPI-1 does not specify how to run an MPI program

Starting an MPI program is dependent on implementation

- Scripts, program arguments, and/or environment variables

% mpirun -np <procs> a.out

- **For MPICH under Linux**

mpiexec <args>

- Recommended part of MPI-2, as a recommendation
- **mpiexec** for MPICH (distribution from ANL)
- **mpirun** for SGI’s MPI

Finding Out About the Environment

- Two important questions that arise in message passing
  - How many processes are being use in computation?
  - Which one am I?
- MPI provides functions to answer these questions
  - **MPI_Comm_size** reports the number of processes
  - **MPI_Comm_rank** reports the rank
    - number between 0 and size-1
    - identifies the calling process

Hello World Revisited

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

❒ What does this program do?

Comm?
“Communicator”

Basic Concepts

Processes can be collected into *groups*

Each message is sent in a *context*

- Must be received in the same context!

A group and context together form a *communicator*

A process is identified by its *rank*

- With respect to the group associated with a communicator

There is a default communicator MPI_COMM_WORLD

- Contains all initial processes

MPI Basic (Blocking) Send

`MPI_SEND (start, count, datatype, dest, tag, comm)`

The message buffer is described by:

- **`start`**, **`count`**, **`datatype`**

The target process is specified by **`dest`**

- Rank of the target process in the communicator specified by **`comm`**

Process blocks until:

- Data has been delivered to the system
- Buffer can then be reused

Message may not have been received by target process!

MPI Datatypes

Message data (sent or received) is described by a triple

address, count, datatype

An MPI *datatype* is recursively defined as:

Predefined data type from the language

A contiguous array of MPI datatypes

A strided block of datatypes

An indexed array of blocks of datatypes

- Enables heterogeneous communication
  - Support communication between processes on machines with different memory representations and lengths of elementary datatypes
  - MPI provides the representation translation if necessary
- Allows application-oriented layout of data in memory
  - Reduces memory-to-memory copies in implementation
  - Allows use of special hardware (scatter/gather)

MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*
  - Assist the receiving process in identifying the message
- Messages can be screened at receiving end by specifying specific tag
  - **MPI_ANY_TAG** matches any tag in a receive
- Tags are sometimes called “message types”
  - MPI calls them “tags” to avoid confusion with datatypes

MPI with Only Six Functions

Many parallel programs can be written using:

- **MPI_INIT()**
- **MPI_FINALIZE()**
- **MPI_COMM_SIZE()**
- **MPI_COMM_RANK()**
- **MPI_SEND()**
- **MPI_RECV()**

Why have any other APIs (e.g. broadcast, reduce, etc.)?

Point-to-point (send/recv) isn’t always the most efficient...

- Add more support for communication

Excerpt: Count 3s

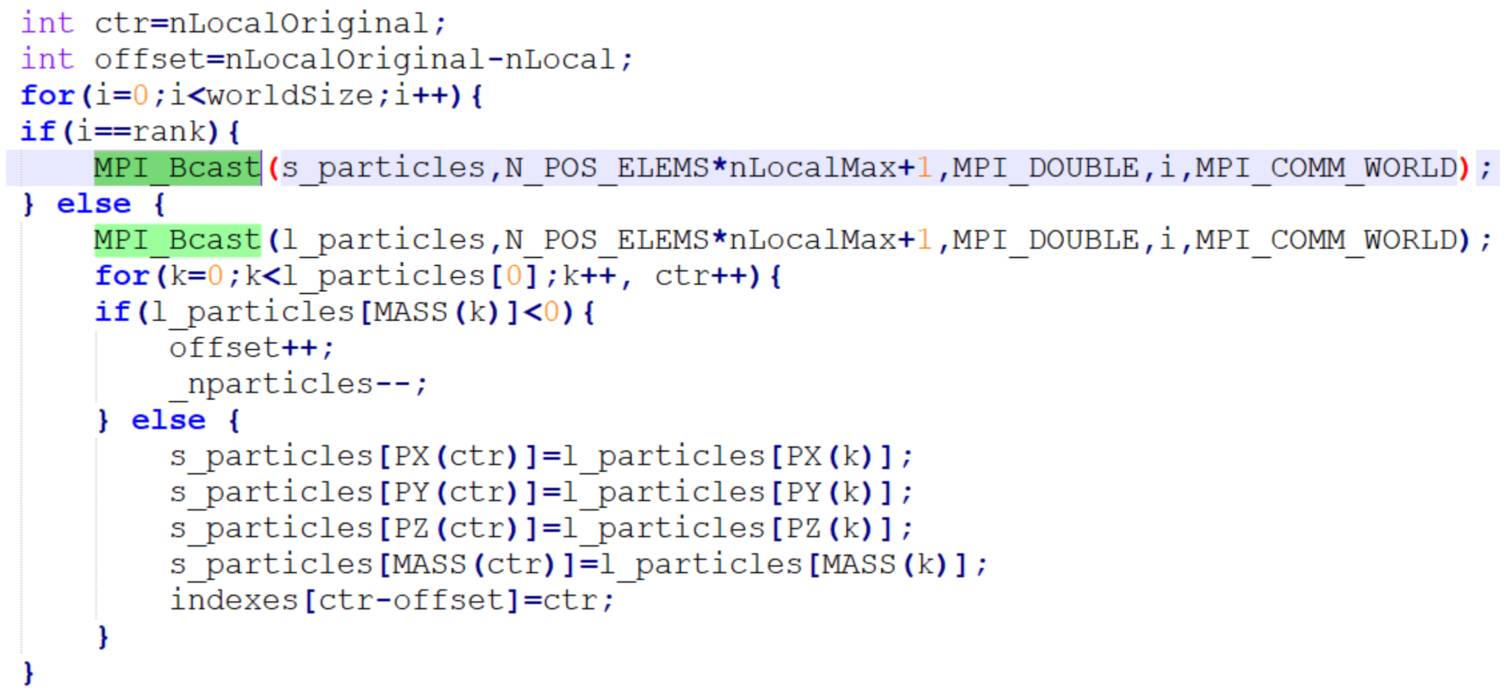
```
if(myID == RootProcess) {

    FILE * fp = fopen(argv[1], "r");
    res = fscanf(fp, "%d\n", &nnums);
    for(int p=0; p<num_procs-1; p++) {
        for(int i=0; i<shard_length; i++) {
        res = fscanf(fp, "%d\n", &values[i]);
        assert(res != EOF);
        }
        //_info("ID-%d: sending shard_length:%d to ID-%d\n", RootProcess, shard_length, p+1);
        MPI_Send(&shard_length, 1, MPI_INT, p+1, stag++, MPI_COMM_WORLD);
        //_info("ID-%d: sending shard:%d to ID-%d\n", RootProcess, shard_length, p+1);
        MPI_Send(values, shard_length, MPI_INT, p+1, stag++, MPI_COMM_WORLD);
    }
    /* ... */

} else {

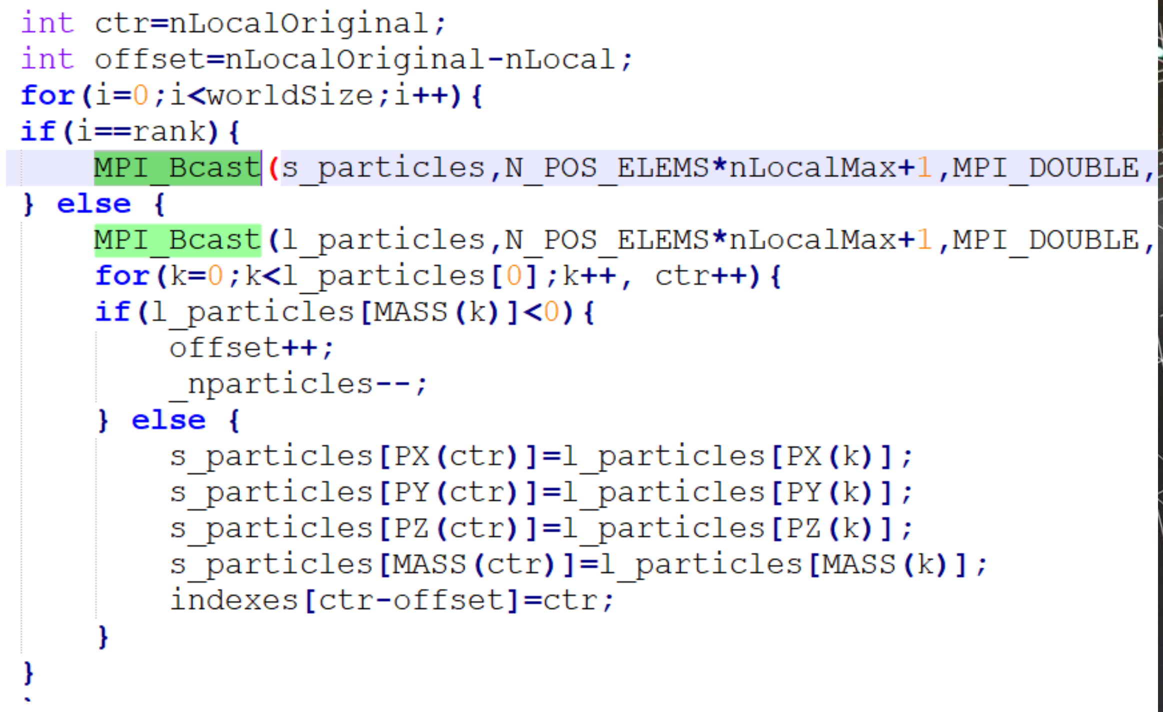
    MPI_Recv(&shard_length, 1, MPI_INT, RootProcess, tag, MPI_COMM_WORLD, &status);
    values = (int*)calloc(shard_length,sizeof(int));
    MPI_Recv(values, shard_length, MPI_INT, RootProcess, tag, MPI_COMM_WORLD, &status);
    mylength = shard_length;
}
```

Excerpt: Barnes-Hut

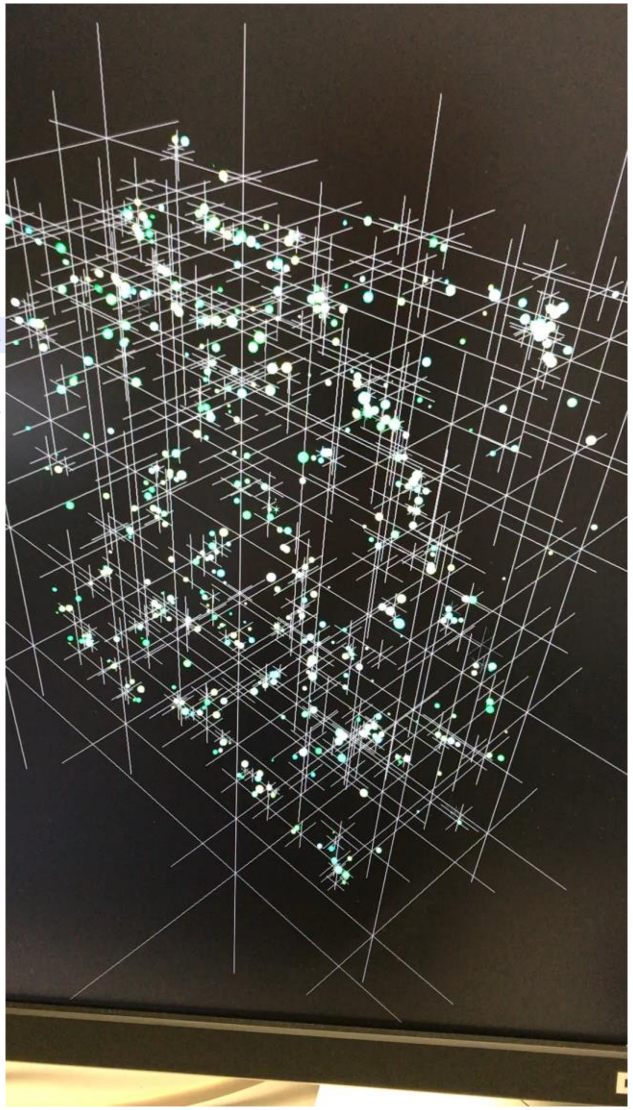


```
int ctr=nLocalOriginal;
int offset=nLocalOriginal-nLocal;
for(i=0;i<worldSize;i++){
if(i==rank){
    MPI_Bcast(s_particles,N_POS_ELEMS*nLocalMax+1,MPI_DOUBLE,i,MPI_COMM_WORLD);
} else {
    MPI_Bcast(l_particles,N_POS_ELEMS*nLocalMax+1,MPI_DOUBLE,i,MPI_COMM_WORLD);
    for(k=0;k<l_particles[0];k++, ctr++){
    if(l_particles[MASS(k)]<0){
        offset++;
        _nparticles--;
    } else {
        s_particles[PX(ctr)]=l_particles[PX(k)];
        s_particles[PY(ctr)]=l_particles[PY(k)];
        s_particles[PZ(ctr)]=l_particles[PZ(k)];
        s_particles[MASS(ctr)]=l_particles[MASS(k)];
        indexes[ctr-offset]=ctr;
    }
}
```

Excerpt: Barnes-Hut



```
int ctr=nLocalOriginal;
int offset=nLocalOriginal-nLocal;
for(i=0;i<worldSize;i++){
if(i==rank){
    MPI_Bcast(s_particles,N_POS_ELEMS*nLocalMax+1,MPI_DOUBLE,
} else {
    MPI_Bcast(l_particles,N_POS_ELEMS*nLocalMax+1,MPI_DOUBLE,
    for(k=0;k<l_particles[0];k++, ctr++){
    if(l_particles[MASS(k)]<0){
        offset++;
        _nparticles--;
    } else {
        s_particles[PX(ctr)]=l_particles[PX(k)];
        s_particles[PY(ctr)]=l_particles[PY(k)];
        s_particles[PZ(ctr)]=l_particles[PZ(k)];
        s_particles[MASS(ctr)]=l_particles[MASS(k)];
        indexes[ctr-offset]=ctr;
    }
}
```



Reduce/Allreduce

A0	B0	C0
A1	B1	C1
A2	B2	C2

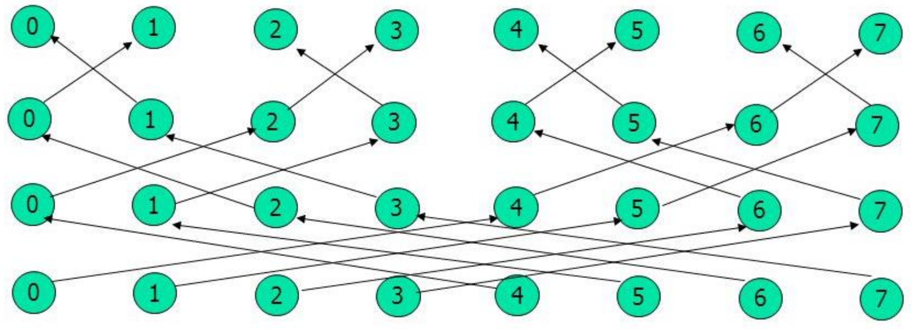
reduce $\Longrightarrow$

A0+A1+A2	B0+B1+B2	C0+C1+C2

llreduce $\Longrightarrow$

A0+A1+A2	B0+B1+B2	C0+C1+C2
A0+A1+A2	B0+B1+B2	C0+C1+C2
A0+A1+A2	B0+B1+B2	C0+C1+C2

```
Int MPI::COMM_WORLD.Allreduce(
                void*           operand   /* in */,
                void*           result    /* out */,
                int             count     /* in */,
                MPI::Datatype   datatype  /* in */,
                MPI::Op         operator  /* in */)
```



MPI_Reduce

- ***MPI_Reduce*** (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  - IN sendbuf (address of send buffer)
  - OUT recvbuf (address of receive buffer)
  - IN count (number of elements in send buffer)
  - IN datatype (data type of elements in send buffer)
  - IN op (reduce operation)
  - IN root (rank of root process)
  - IN comm (communicator)

- MPI_Reduce combines elements specified by send buffer and performs a **reduction operation** on them.
- There are a number of predefined reduction operations: MPI_MAX, MPI_MIN, MPI_SUM, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC

Reduce_scatter/Scan

A0	B0	C0
A1	B1	C1
A2	B2	C2

reduce-scatter

A0+A1+A2		
B0+B1+B2		
C0+C1+C2		

A0	B0	C0
A1	B1	C1
A2	B2	C2

scan

A0	B0	C0
A0+A1	B0+B1	C0+C1
A0+A1+A2	B0+B1+B2	C0+C1+C2

MPI_Scan

- *MPI_Scan* (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
  - IN sendbuf (address of send buffer)
  - OUT recvbuf (address of receive buffer)
  - IN count (number of elements in send buffer)
  - IN datatype (data type of elements in send buffer)
  - IN op (reduce operation)
  - IN comm (communicator)

- Note: **count** refers to total number of elements that will be recveived into receive buffer after operation is complete

To use or not use MPI?

- USE
  - You need a portable parallel program
  - You are writing a parallel library
  - You have irregular or dynamic data relationships
  - You care about performance
- NOT USE
  - You don’t need parallelism at all
  - You can use libraries (which may be written in MPI) or other tools
  - You can use multi-threading in a concurrent environment
    - You don’t need extreme scale