INF-3201 Parallel Programming

Exam Preparation 2016

Camilla Stormoen

1. Load Balancing, chapter 7 (p.)

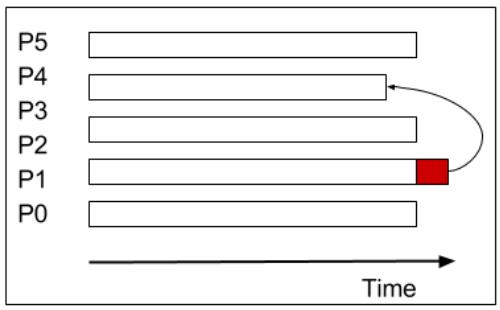
1.1. H15, H13

1.1.1. Give a short explanation of the following:

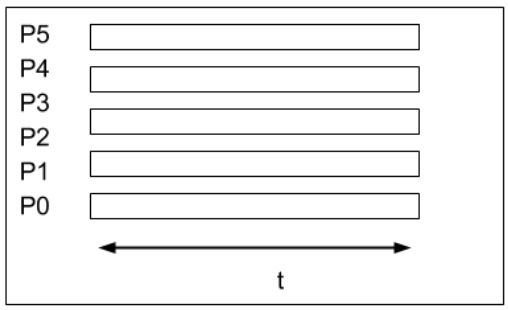
1.1.1.1. Load balancing:

Used to distribute computations fairly between processors in order to obtain the highest-possible execution speed.

Dynamic load balancing is more powerful than static, but it does incur very significant overhead when processes are created.



Imperfect load balancing leading to increased execution time



Perfect load balance

1.1.1.2. Static load balancing

All processes are specified before execution and the system will execute a fixed number of processes. Divide work to all equally.

1.1.1.3. Dynamic load balancing

Processes can be created and their execution initiated during the execution of other processes. Divide random work and ask for new task when done.

In dynamic load balancing, tasks are allocated to processors during the execution of the program. Dynamic load balancing can be classified as one of the following:

- *Centralized*: tasks are handed out from a centralized location. Master-slave structure which the master process controls each of the set of slave process directly.

- *Decentralized*: tasks are passed between arbitrary processes. A collection of worker processes operate upon the problem and interact among themselves, finally reporting to a single process.

1.1.1.4. What separates applications (or algorithms) that benefit from static load balancing vs. ones that benefit from dynamic load balancing?

https://www.quora.com/What-is-the-difference-between-static-balancing-and-dynamic-balancing

- Static load balancing:
  - Round robin algorithm
  - Randomized algorithms
  - Recursive bisection
  - Simulated annealing
  - Genetic algorithm

- Dynamic load balancing:
  - Central Queue Algorithm
  - Local Queue Algorithm
  - Least Connection Algorithm

1.2. H13

1.2.1. Explain and contrast the following terms (p. 106):

1.2.1.1. Data partitioning vs. functional parallelism

- Data partitioning *(domain decomposition)*:
  - Partitioning divides the problem into parts. Partitioning can be applied to the program data (i.e. to dividing the data and operating upon the divided data concurrently.
  - Example: Mandelbrot

- Functional parallelism *(functional decomposition)*:
  - Partitioning can also be applied to the functions of a program (i.e. dividing it into independent functions and executing them concurrently.

2. Message Passing, chapter 2 (p. 42)

2.1. H15, H14, H13

2.1.1. Give a short explanation of what blocking vs. non-blocking communication is in message passing systems and show how the two can be used. You can use MPI as an example (or MPI-like pseudocode).

- *Blocking message-passing:* describe routines that do not allow the process to continue until the transfer is completed. The routines are “blocked” from continuing.

Blocking Send

MPI_Send(buf, count, datatype, dest, tag, comm)
buf - address of send buffer
count - number of items to send
datatype - datatype of each item
dest - rank of destination process
tag - message tag
comm - communicator

MPI_Recv(buf, count, datatype, src, tag, comm, status)
buf - address of receive buffer
count - max number of items to receive
datatype - datatype of each item
dest - rank of destination process
tag - message tag
comm - communicator
status - status after operation

Example: Send an integer x from process 0 to process 1

```
MPI_Comm_Rank(MPI_COMM_WORLD, &myRank); //find rank of this process

if (myRank == 0)
{
        int x;
        MPI_Send(&x, 1, MPI_INT, 1, msgTag, MPI_COMM_WORLD);
}
else if (myRank == 1)
{
        int x;
        MPI_Recv(&x, 1, MPI_INT, 0, msgTag, MPI_COMM_WORLD, status);
}
```

- *Non-blocking message-passing*: describe routines that return whether or not the message had been received.

Non-Blocking Send

MPI_Isend(buf, count, datatype, dest, tag, comm, request)
buf - address of send buffer
count - number of items to send
datatype - datatype of each item
dest - rank of destination process
tag - message tag
comm - communicator

MPI_Irecv(buf, count, datatype, src, tag, comm, status)
buf - address of receive buffer
count - max number of items to receive
datatype - datatype of each item
dest - rank og destination process
tag - message tag
comm - communicator
status - status after operation

Example: Send an integer x from process 0 to process 1 and allow process 0 to continue

```
MPI_Comm_Rank(MPI_COMM_WORLD, &myRank); //find rank of this process

if (myRank == 0)
{
        int x;
        MPI_Isend(&x, 1, MPI_INT, 1, msgTag, MPI_COMM_WORLD, req1);
        compute();
        MPI_Wait(req1, status); //wait until operation completed and returns then
}
else if (myRank == 1)
{
        int x;
        MPI_Irecv(&x, 1, MPI_INT, 0, msgTag, MPI_COMM_WORLD, status);
}
```

2.1.2. What is the difference between asynchronous and synchronous operations?

- *Asynchronous operations:*
  - Routines that do *not wait* for actions to complete before returning.
  - Usually require local storage for messages.
  - More than one version depending upon the actual semantics for returning.

- *Synchronous operations:*
  - Performs two actions
    - Transfer data
    - Synchronize processes
      - A synchronous send routine will wait until the complete message that it has sent has been accepted by the receiving process before returning.
    - Send - returns when message can be accepted by receiver.
    - Receive - returns when message received.

2.1.3. When does a locally blocking send routine in MPI behave as a synchronous routine?

(*PP: 03a_messagepassing.pdf, s 13) "Asynchronous (blocking) routines changing to synchronous routines"*

- Once local actions completed and message is safely on its way, sending process can continue with subsequent work.
- Buffers only of finite length and a point could be reached when send routine held up because all *available buffer space exhausted*.
- Then, send routine will *wait* until storage becomes re-available - i.e then routine behaves as a synchronous routine.

2.2. *From lecture, he said "Exam!!" Synchronized computations*

- *Fully synchronous:* All processes involved in the computation must be synchronized.

- *Locally synchronous*: processes only need to synchronize with a set of locally/logically nearby processes. Not all processes involved in the computation.
  - Example: Heat distribution - Frost Trap.

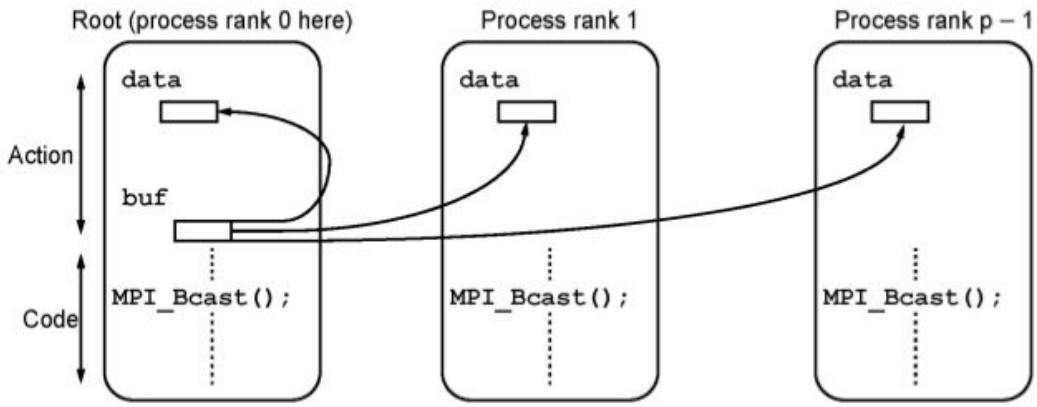
3. Collective/group operations

3.1. H15, H14

3.1.1. Explain the term group (or collective) operations. A short description is enough. Explain the following operations (p. 49):

3.1.1.1. Broadcast

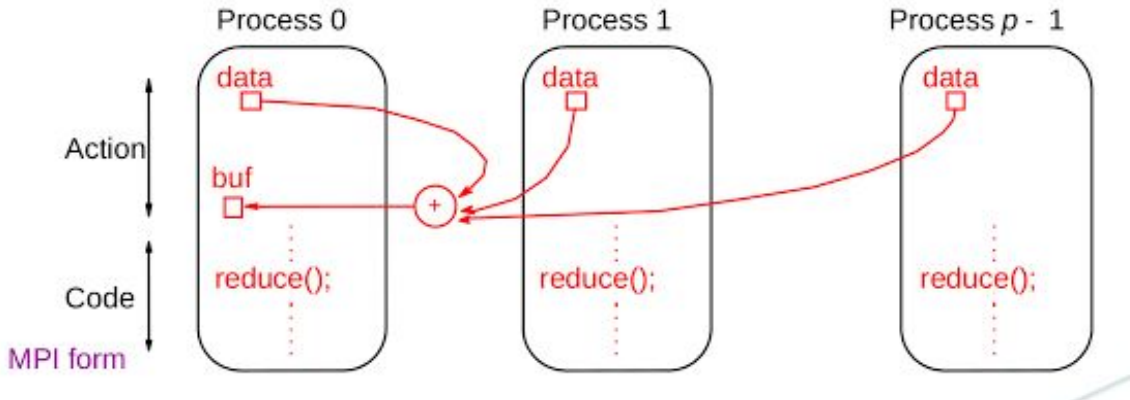
- Broadcast is used to describe sending the same message to all the processes concerned with the problem.
- From root -> all other processes



2a.4

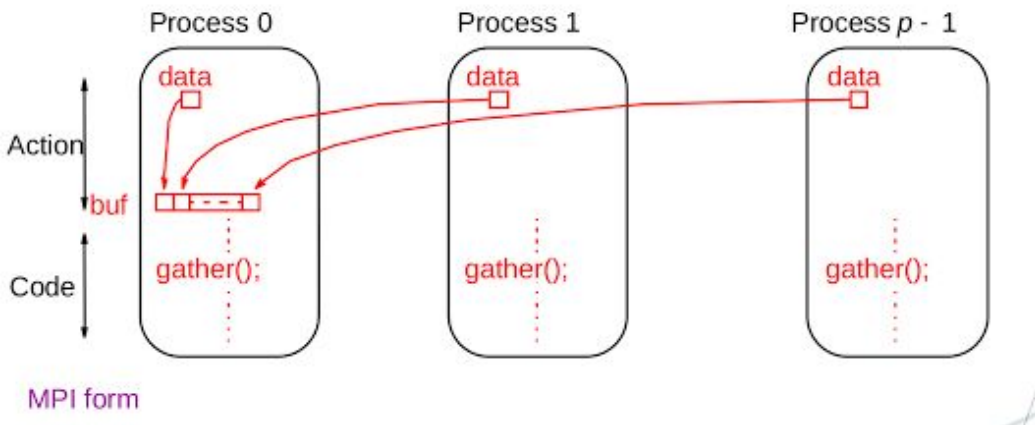
3.1.1.2. Reduce

- Gather operation combined with specific arithmetic/logical operation.
- E.g.: values could be gathered and then added together by the root, which will be done by the reduce operations.
- Combine values on all processes to single value.



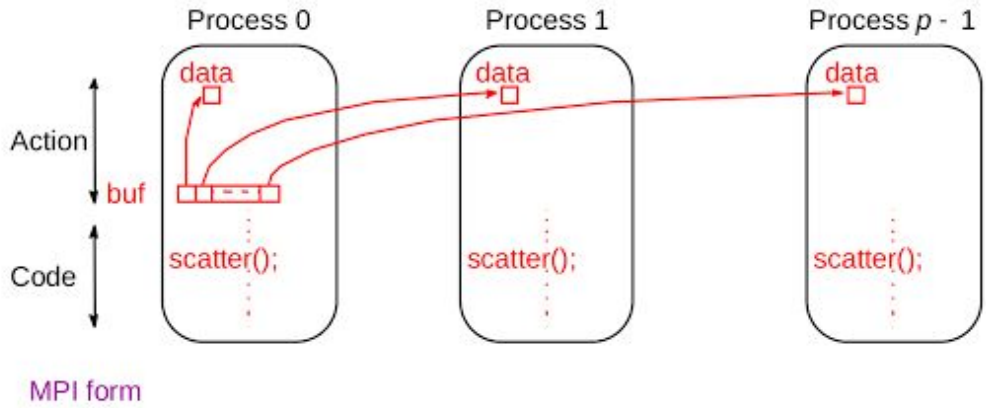
3.1.1.3. Gather

- Having one process collect individual values from set of processes.
- Gather is used to to described having one process collect individual values from a set of processes. Gather is normally used after some computations has been done by these processes.
- Gather values for group of processes.



3.1.1.4. Scatter

- Sending each element of an array in root process to a separate process. Contents of *i*th location of array sent to *i*th process.
- Scatter is used to describe sending each element of an array of data in the root to separate processes.
- Scatter buffer in parts to group of processes



4. GPGPU computing with CUDA

4.1. H15, H14, H13

4.1.1. What is a "Kernel"?

A kernel is a function/functions executed on the GPU. The kernels are executed by many GPU threads in parallel. CUDA C extends C by allowing the programmer to define C functions, called kernels.

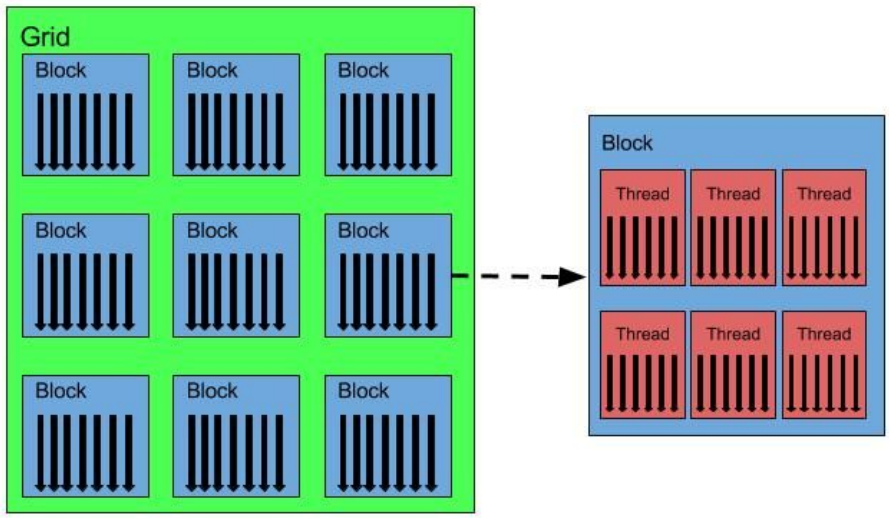
4.1.2. What are "thread blocks", "warps" and "grids"? How are they related?

- *In CUDA we usually launch many threads in groups of thread blocks that form a grid.*

- *Grid*: Parallel code/A kernel is launched as a grid of thread blocks which are completely independent.

- *Warp*: Consists of 32 threads. Instructions are issued per warp. (Threads are single execution units that run kernels in the GPU)

- *Block*: Consists of warps (max 32 warps/block). All the threads in the same block can communicate.



4.1.3. Outline how a program that uses a GPU looks and works. The description should be high level and can use pseudocode. Assume that the program has some data that the GPU should use and that the GPU code should produce some resulting data for the CPU. Don't worry about performance optimisations, we're after a rough outline/template and how the GPU and the CPU cooperate.

CPU/GPU:

- Alloc memory for host(cpu) and device(gpu)
- copy from CPU to GPU
- do some code
- wait for kernel done (sync, kernel is async)
- copy from GPU to CPU
- CPU can do other stuff in between

5. Classification of computations

5.1. H15

```
// Matrix multiplication
void mxmult()
{
        // Given the following arrays (initialised elsewhere)
        int A[N][N];
        int B[N][N];
        int C[N][N];
        ...
        // Simple matrix multiplication C = A*B
        for(int i = 0; i < N; i++) {
                for(int j = 0; j < N; j++) {
                        C[i][j] = 0;
                        for(int k = 0; k < N; k++) {
                                C[i][j] += A[i][k]*B[k][j];
                        }
                }
        }
}
```

```
Mandelbrot:
/*
Mandelbrot divergence test (from mandatory exercise 1)
@param x,y Space coordinates
@returns Number of iterations before convergence
*/
inline int solve(double x, double y) {
        double r=0.0, s=0.0;
        double next_r, next_s;
        int itt=0;
        while((r*r+s*s)<=4.0) {
                next_r=r*r-s*s+x;
                next_s=2*r*s+y;
                r=next_r; s=next_s;
                if(++itt==ITERATIONS)
                        break;
        }
        return
        itt;
}
// somewhere in the main code:
....
//Loops over pixels
for(y=0;y<HEIGHT;y++){
        for(x=0;x<WIDTH;x++){
        ...
        colorMap[y][x]=palette[solve(translate_x(x),translate_y(y))];
        }
}
....
```

5.1.1. Two of the classifications we can use for algorithms are memory bound and CPU/compute bound.

5.1.1.1. What is a memory bound algorithm? Which one of the above algorithms would be memory bound?

Memory bound means the rate at which a process progresses is limited by the amount memory available and the speed of that memory access. A task that processes large amounts of in memory data, for example multiplying large matrices, is likely to be Memory Bound. The best known algorithm that takes advantages of memoization is an algorithm that computes the Fibonacci numbers.

Algorithm above: matrix, need all columns in memory, read from different memory A[], C[]

5.1.1.2. What is a compute bound algorithm? Which one of the above algorithms would be compute bound?

CPU Bound means the rate at which process progresses is limited by the speed of the CPU. A task that performs calculations on a small set of numbers, for example multiplying small matrices, is likely to be CPU bound.

Algorithm above: mandelbrot - heavy computation for each pixel

6. GPU vs CPU

6.1. H15

6.1.1. Given two architectures from one of the papers we looked at this year (for simplicity, we only include some of the numbers):

(*From "Debunking the 100X GPU VS. CPU Myth"*)

	Num. PE	BW (GB/sec)	Peak SP Scalar, GFLOPS
CPU: Core i7-960	4	32	25.6
GPU: GTX 280	30	141	116.6

As a rule of thumb, which of the architectures would you use for algorithms that are:

6.1.1.1. Compute bound

- *GPU*
- Faster computations

6.1.1.2. Memory bound

- CPU/GPU????
- Costly for GPU
- Copy memory from cpu to gpu - no other way
- GPU have gather/scatter from memory

6.1.1.3. Synchronization bound

- *CPU*
- Only synchronization between (intern) blocks in GPU
- GPU struggled with synchronization

Explain why for each. You only need to describe the main reason for each.

7. CSP mechanisms

7.1. H15

```
# Using the old PyCSP 0.3 library
@process
def lazy(out):
        while True:
                time.sleep(5)
                out("kebab")

@process
def eager(out):
        while True:
                time.sleep(1)
                out("beer")

@process
def reader(ch1, ch2):
        alt = Alternative(ch1, ch2)
        while 1:
                ret = alt.select()
                print "Got a", ret()
ch1 = One2OneChannel()
ch2 = One2OneChannel()
Parallel(lazy(ch1.write), eager(ch2.write), reader(ch1.read, ch2.read))
```

The following code is equivalent if you prefer to read Go:

```
func
lazy(out chan string) {
        for{
        time.Sleep(5*time.Second)
        out <- "kebab"
        }
}

func eager(out chan string) {
        for{
                time.Sleep(1*time.Second)
                out <- "beer"
        }
}

func reader(ch1 chan string, ch2 chan string) {
        for{
                select{
                        case s := <- ch1:
                                fmt.Println("Got a", s)
                        case s := <- ch2:
                                fmt.Println("Got a", s)
                }
        }
}

func main() {
        ch1 := make(chan string)
        ch2 := make(chan string)
        go lazy(ch1);
        go eager(ch2);
        go reader(ch1, ch2);
        time.Sleep(30*time.Second) // some arbitrary time
}
```

~~7.1.1. Draw the process network.~~

7.1.2. Why do we need an external choice construct in CSP (called Alternative in PyCSP, select in Go)? What does it solve?

Go's select lets you wait on multiple channel operations.
Here we select across two channels.
We'll use select to await both of these values simultaneously, printing each one as it arrives.
Basic sends and receives on channels are blocking. However, we can use select with a default clause to implement non-blocking sends, receives, and even non-blocking multi-way selects.

7.1.3. What (approximately) does the program print?

Eager will print each second and in the 5 second lazy will print one time and it will continue..
For 30 seconds: Got a beer -> Got a beer -> Got a beer -> Got a beer -> Got a beer -> Got a kebab -> Got a beer -> ...

8. Parallelisation strategies

8.1. H14

8.1.1. Give a short description of the following:

8.1.1.1. Embarrassingly parallel computations (p. 79)

A computation can be divided into a number of completely independent parts, each of which can be executed by a separate processor.
Example: Mandelbrot set

8.1.1.2. Divide and conquer (p. 111)

The divide and conquer approach is characterized by dividing a problem into subproblems that are of the same form as the larger problem.
Example: Sorting (Merge Sort, Quicksort), multiplying large numbers etc.

8.1.1.3. Pipelined computations (p. 145?)

Problem divided into a series of tasks that have to be completed one after the other (the basis of sequential programming). Each task executed by a separate process or processor.
P0->P1->P2->P3->P3->...
Example: Frequency filter or *“Add all the elements of array a to an accumulating sum”*:

```
for(i = 0); i < n; i++)
        sum = sum + a[i];
```

Where can pipelining be used?

- Assuming problem can be divided into a series of sequential tasks, pipelined approach can provide increased execution speed under the following three types of execution:

1. If more than one instance of the complete problem is to be executed
2. If a series of data items must be processed, each requiring multiple operations.
3. If information to start next process can be passed forward before process has completed all its internal operations.

9. Parallel Execution

9.1. H14

```
for (i = 2; i < 6; i++) {
        x = i - 2*i + i*i;
        a[i] = a[x];
}
```

9.1.1. Can you run the iterations in parallel without causing any problems? Explain why.

No, you need to have locks on x.
a[6] will overwrite the first.. need ordering between accesses.

2 - 2*2 + 2*2 = 2
3 - 2*3 + 3*3 = **6**
4 - 2*4 + 4*4 = 12
5 - 2*5 + 5*5 = 20

if you have i <= 6
6 - 2*6 + 6*6 = 30

Hint: Bernstein's conditions are useful for analysing the problem. (s. 250 -> this example in the book)
Bernstein's *conditions do not allow memory to be shared between different processes. For that, some means of enforcing an ordering between accesses is necessary, such as semaphores, barriers or some other synchronization method.*

10. Barriers and concurrency

10.1. H14

10.1.1. What is a barrier? What purpose does it have?

A barrier is a mechanism that prevents any process from continuing past a specified point until all the processes are ready.
Purpose:

The following is a Python program with a barrier of size 2. By calling *run_sync*, we can start *nthr* threads that each try to call *b.wait() nsyncs* times.

```
import threading
import sys
import time

b = threading.Barrier(2)

def syncthr(thid, nsyncs):
        for i in range(nsyncs):
                print(thid, "waiting for barrier round", i)
                b.wait()
                time.sleep(1)

def run_sync(nthr, nsyncs):
        #make 'nthr' worker threads, telling each to sync 'nsync' times
        workers = [threading.Thread(target=syncthr, args=(i, nsyncs)) for i in range(nthr)]

        #start the threads
        for w in workers:
                w.start()

        #wait for every thread to complete
        for w in workers:
                w.join()
```

10.1.2. Which of the following calls will complete? Explain why. It may be easier to draw the execution of each thread on a timeline to help solve the problem.

a) *run_sync(2, 1)*
b) *run_sync(3, 1)*
c) *run_sync(3, 2)*

a and ***c*** will complete.
Barrier need 2 threads to continue.

A look like this:	B looks like this:	C looks like this:
syncthr(1,1): 1 wait 1 syncthr(2,1): 2 wait got two -> GO	syncthr(1, 1): 1 wait 1 syncthr(2,1): 2 wait 1 got two -> GO synchtr(3,1): 3 wait 1 Wait forever! Only 1 thread	syncthr(1,2): 1 wait 1 1 wait 2 got two -> GO syncthr(2,2): 2 wait 1 2 wait 2 got two -> GO synthr(3,2): 3 wait 1 3 wait 2 got two -> GO

11. Speedup and scaling

11.1. H13

11.1.1. Assume that we have a program where 10% of the code must be executed sequentially. What is the maximum potential speedup of the program (ignoring communication overhead) if we execute it on 32 CPU's?

Total time to execute: *T(2) = 0.1 + (1 - 0.1) / 32 = 0.128*

Amdahl's law:

$$S(n) = \frac{T(1)}{T(1)(B + \frac{1}{n}(1-B))} = \frac{32}{32(0.1+\frac{1}{32}(1-0.1))} = 7.80$$

$$S(n) = \frac{1}{(1-0.9)+(\frac{0.9}{32})} = 7.8$$

$S(n)$ *is theoretical speedup*
$T(n)$ *is the time an algorithm takes to finish when running n threads*
B is the fraction of the algorithm that i s strictly serial
(*so* $1-B$ *is how much of the program can be run in parallel*)

11.1.2. What is Amdahl's law?

Calculate how much a computation can be speed up by running part of it in parallel.
A program (or algorithm) which can be parallelized can be split up into two parts:

- A part which cannot be parallelized
- A part which can be parallelized

Amdahl's law: $\frac{1}{(1-P)+\frac{P}{cores}}$

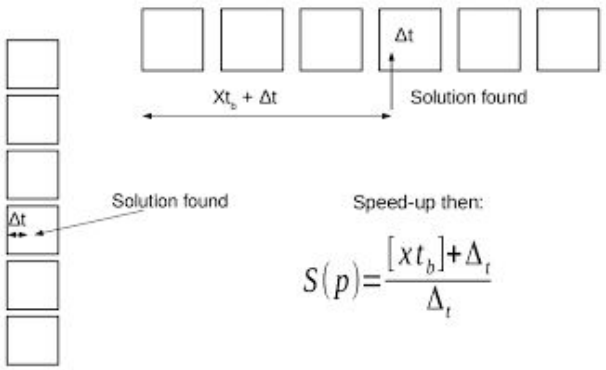
P *is parallel part in* %

11.1.3. What is superlinear speedup?

Superlinear speedup: Sometimes a speedup of more than *A* when using *A* processors is observed.
One common reason for superlinear speedup is extra memory in the multiprocessor system.
Speedup better than sequential code

11.1.4. Why can we observe superlinear speedup in applications such as search applications?

In search problems performed by exhaustively looking for the solution, suppose the solution space is divided among the processors for each one to perform an independent search. In a sequential implementation, the different search spaces are attacked one after another. In parallel implementation, they can be done simultaneously, and one processor might find the solution almost immediately.



12. Pipelines, CSP and MPI

12.1. H13

12.1.1. We are going to make a simple pipelined program with P processes. Each process adds a constant to a received partial sum and passes it to the next process. The first process is the source and the last process is the sink. For simplicity, just assume that the source has a list of numbers of length N and that the last process collects the finished numbers in a result list.

When writing the programs, you can use simple pseudo code. You don't need to remember all of the parameters or exact names for API calls. You can also use a CSP-based language(PyCSP, JCSP, Go or similar pseudocode) instead of syntax from CSP theory.

12.1.1.1. Write pseudocode for a simple CSP program implementing the program.

Like "Sieve of eratosthenes"

```
func main() {
	var c = make(chan int)
	N = 5
	go add(num, c, N)
	res := <-c //receive from c
}

func add (num int, c chan int) {
	sum := make(chan int)
	sum += num
	c <- sum //send sum to c
}
```

12.1.1.2. Write pseudocode for a simple MPI program implementing the program.

"Slides: Pipelined, s. 15"

```
int numTasks, taskId, sum = 0;
int status, msgTag;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numTasks);
MPI_Comm_rank(MPI_COMM_WORLD, &taskId);

if (taskId == 0) {
      //master
      MPI_Send(&number, 1, MPI_INT, P_1, msgTag,
      MPI_COMM_WORLD);
}
else if (taskId == (numTasks - 1)) {
      //last process
      MPI_Recv(&number, 1, MPI_INT, P_1,
      MPI_COMM_WORLD, &status);
      MPI_Reduce();?????
      sum = sum + number;
}
else {
      //all other processes
      MPI_Recv(&number, 1, MPI_INT, P_numTasks-1,
      msgTag, MPI_COMM_WORLD, &status);
      sum = sum + number
      MPI_Send(&sum, 1, MPI_INT, P_numTasks+1,
      msgTag, MPI_COMM_WORLD);
}
```

GATHER/REDUCE/SCATTER

```
MPI_Scatter(&sendData, 1, MPI_INT, &recvData,
1, MPI_INT, 0, MPI_COMM_WORLD);
sum = 0;
for (i = 0; i < numTasks; i++) {
      sum += number[i]
}

MPI_Gather(&sum, 1, MPI_INT, &sum2, 1,
MPI_INT, numTasks-1, MPI_COMM_WORLD);
```

12.1.2. It often makes sense for the *source* and the *sink* to be the same process (the process with the problem is often the one using the solution). This changes the configuration from a line to a circle.

12.1.2.1. This can easily cause deadlocks. Why?

Can wait on each other

12.1.2.2. CSP has a mechanism we can use to avoid deadlocks in this situation. What is it and how does it work? Modify your program to avoid deadlocks using this mechanism.

Hoare: Processes cross the critical area to gain access to the shared data. Before entry to the critical area, all other processes must verify and update the value of the shared variable. Upon exit, the processes must again verify that all processes have the same value.

Another technique to maintain data integrity is through the use of mutual exclusion semaphore or a mutex. A mutex is a specific subclass of a semaphore that only allows one process to access the variable at once. A semaphore is a restricted access variable that serves as the classic solution to preventing race hazards in concurrency. Other processes attempting to access the mutex are blocked and must wait until the current process releases the mutex. When the mutex is released, only one of the waiting processes will gain access to the variable, and all others continue to wait.

12.1.2.3. MPI provides multiple options for avoiding deadlocks with the circle configuration. Describe the one and show how you can change your program to use it.

An MPI program can *deadlock* if one process is waiting for a message from another process that never gets sent.

13. Exam Preparation

- p666 - concepts
- OpenMP!
- Debunking …. -> no kernel! optimization (what is best on gpu/cpu? memory bound algorithm)
- Docs.python:
  - threadpool/mapreduce
    - wait/ask thread -> future
- do something on GPU
  - API
  - MPI_Send(buffer, #, type, …)
  - GPU:
    - Alloc memory for host(cpu) and device(gpu)
    - copy from cpu to gpu
    - do code
    - wait for kernel done (sync, kernel is async)
    - copy from gpu to cpu

- Architects that needs parallel programming
  - multicore
  - GPU
- Divide and conquer
  - Examples
  - Partitioning