CPD Notes

Introduction

Opportunities for parallelism

- Functional, different tasks but same data
- Data, different data but same task
- Pipelining, set of taks running in parallel only syncronized at a given point

Speeedup

$$S = \frac{t_{serial}}{t_{parallel}}$$

For P processors the ideal is S = P, but the xpected is S < P.
If S>P, we get superlinear speedup.

What is in the way?

- Data transfers
- Task startup/finalize
- Load balancing
- Serial portions of execution

The parallel execution time is

$$t_{parallel} = ft_{serial} + (1-f)\frac{t_{serial}}{p}$$

where f is the fraction of time in serial execution.

$$S = \frac{t_{serial}}{t_{parallel}} = \frac{1}{f + \frac{1-f}{p}}$$

When p tends to infinite, S = 1/f.

Units of measure in High-Performance Computing

- Flop, floating point operation
- Flop/s
- Bytes

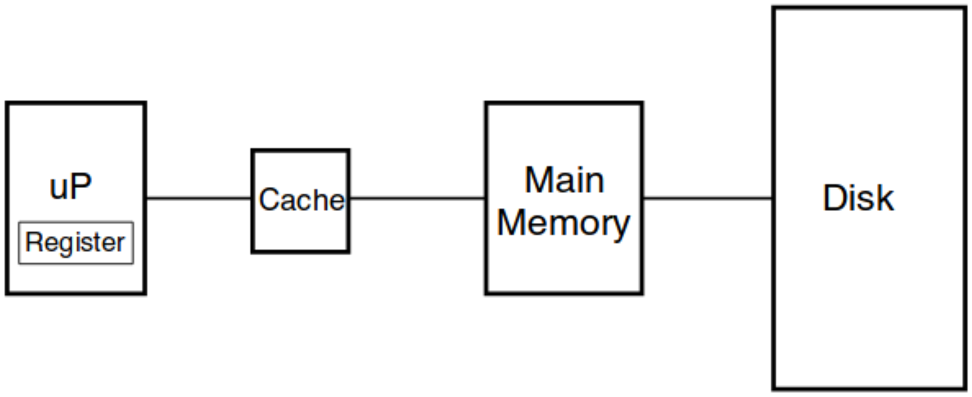
Classification of computer architectures (Flynn' Taxonomy)

- Single Instruction, Single Data (SISD)
- Single Instruction, Multiple Data (SIMD), same instruction executed in all processing elements but each on a different set of data.
- Multiple Instruction, Single Data (MISD), each element executes a different instruction but all on the same data (~pipelining).
- Multiple Instruction, Multiple Data (MIMD), can be on shared memory or on distributed memory.

Uniform Memory Access (UMA) architecture (aka SMP)

Solution for Memory Hierarchy

-----> (bigger/cheaper)
<----- (faster)



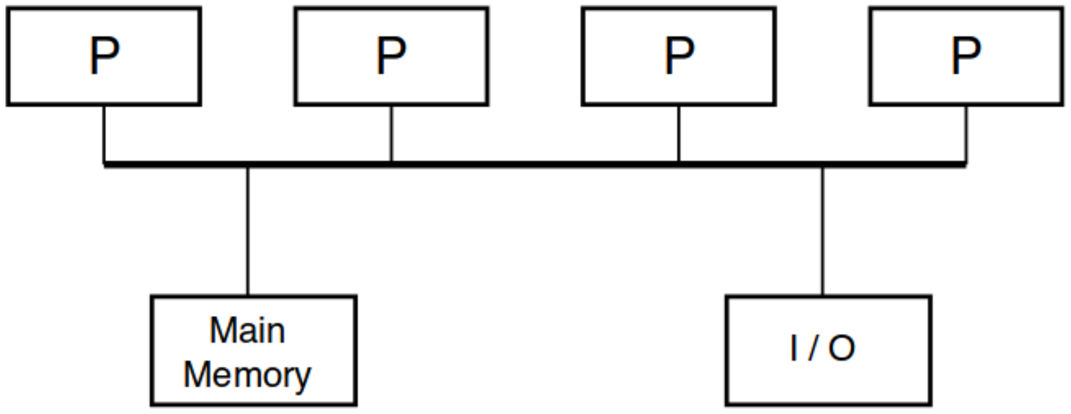
Why does this work?
Principle of Locality.

- Temporal Locality: If a given address is accessed, it will probably be accessed in the near future. Cache stores most recently accessed memory addresses. Virtual Memory keeps in physical memory the most recently accessed virtual address.

- Spatial Locality: If a given address is accessed, adjacent memory addresses will probably be accessed in the near future. Cache brings a set of adjacent memory positions (4 to 16 memory locations). Virtual memory reads from disk a set of adjacent memory positions (4k to 16k memory pages).

Shared Memory Systems

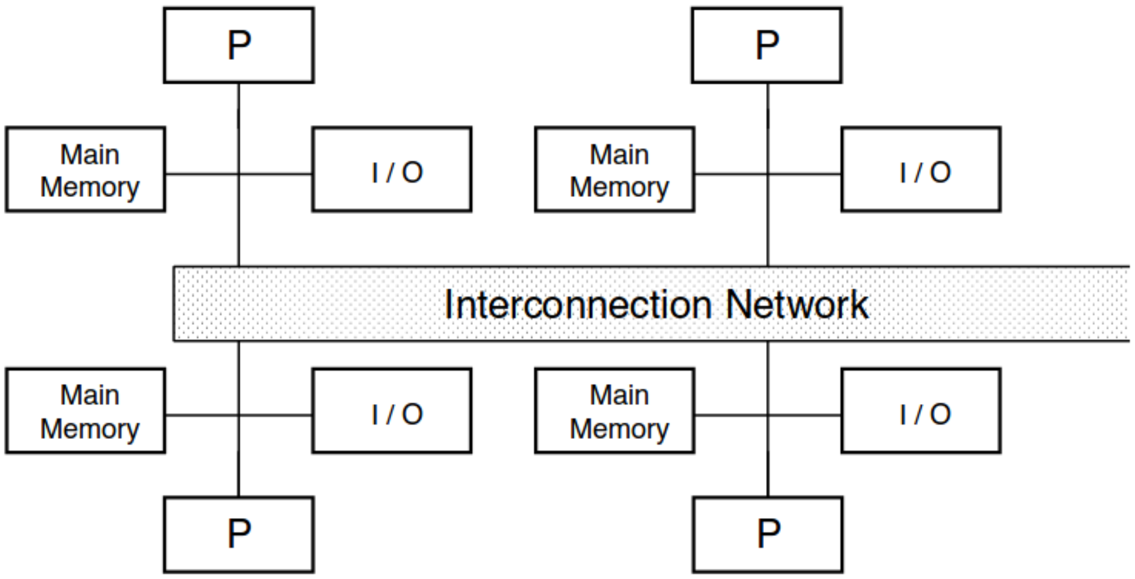
Uniform Memory Access architectures, also known as Symmetric Share Memory Multiprocessors (SMP).



Caches typically write-back to reduce the number of accesses to main memory (write to cache and later flush to main memory) instead of write-through (write to main memory through cache).

Distributed Memory Systems

Non-Uniform Memory Access (NUMA) architectures or Multicomputers.



Shared	Distributed
Data sharing is much easier	Data needs to migrate using messages
Tasks are threads (simpler, less startup/terminate overhead, finer-grain parallelism)	Tasks are processes
Uniform data-access time	Variable
Easier to program	Harder to program
Contention in memory limits scalability	Effective way to increase memory bandwidth
Communication is implicit (harder to debug)	It’s explicit
More complex hardware	Simpler

Memory Coherence

A memory system is coherent if

- A read of M[x] by P that follow a write by P to M[x], and with no writes to M[X] in between,

always returns the value written by P.
- A read of M[X] by P_i that follows a write of M[x] by P_j always returns the value written by P_j if the read and write are sufficiently separated in time and no other writes occur between the two accesses.
- Two writes to the same location by any two processors are seen in the same order by all processors.

Cache Coherence Protocols

Keep track of state of shared blocks of data.

- Directory based (the state is kept in a centralized location)
- Snooping (snooping the bus to determine which status the caches should modify)

Directory based

Block states

- Uncached, no processor has a copy of the data block
- Shared, one or more processors does and the value stored is up to date.
- Exclusive, only one processor has a copy of the data and it has been modified. It’s out of date.

Directory access can be a bottleneck, the solution is a distributed directory.

Snooping

- Invalidate protocols, invalidate value when it’s a updated in memory
- Update protocols, broadcast the new value

Invalidate VS Update

- Multiple writes to the same address cause multiple broadcasts, but only one invalidation.
- Each word written in a given cache block causes broadcast but only the first written causes invalidation
- Broadcasting leds to a smaller delay when reading a word
- Write invalidate are the most used due to lower bandwidth

Memory Consistency

A memory system is coherent if it always reads the most recent value written to the memory being

read.
Coherence defines what values are read after a write.
Consistency defines when values are read after being written.

Sequential consistency

- Every processor sees the write operations in the same order.
- Each write causes the processor to wait until all processors hve been notified.

Non-Uniform Memory Access (NUMA) architecture and Multicomputers

Two approaches:

- NUMA architecture, memories are physically separated but can be addressed as one logically shared addressed space, Distributed Shared Memory (DSM). It entails synchronous writes, processor is blocked until confirmation of write operation.
- Multicomputers, each processor has its own private space address, not accessible by others. Data sharing requires explicit messages among processors. The multicomputers can be asymmetrical (one distributes the workload) or symmetrical.

NUMA (DSM) is easier to program and no compiler libraries are necessary. However, multicomputers have simpler hardware, explicit communication and are able to emulate DSM.

Interconnection Networks

- Packet switching, messages are sent one at a times over a shared medium that connects all processors. The message is divided into packets.
- Circuit switching
  - Bus
  - Ring
  - 2D grid
  - 2D Torus (Rings)
  - 4D Hypercube
  - Butterfly
  - Tree/Fat Tree

Circuit switching evaluations metrics:

- Bisection bandwidth: minimum number of links that need to be cut to divide the network in two halves.
- Diameter: largest number of switches in the path between any nodes.
- Cost: required hardware and wires.
- Scalability: how the above parameters groe with the number of processors.

Example,

Type	Switches	Diameter	Bisection Bandwidth	Ports
2D Mesh	n	$2(\sqrt{n}-1)$	$\sqrt{n}$	4
Tree	2n - 1	2 log(n)	1	3

In order for the programmer to achieve better performance, in some cases asynchronous messages can be sent/received. Executing the read before data is needed.

OpenMP

Directives (#pragma omp directive)

- Parallelization
    - parallel region (Implicit barrier at the end)
    - parallel for (Implicit barrier at the end. Data parallelism.)
    - parallel sections (Funtional Parallelism)
    - taks
- Data environment (shared, private, threadprivate, reduction, firstprivate, lastprivate, …)
- Synchronization (barrier, critical, atomic, single/master, …)

Other functions

- omp_get_thread_num()
- omp_get_num_threads()
- omp_set_num_threads()
- omp_get_num_procs()
- omp_set_lock() and respectives

All variables are by default shared, except the loop variable of a parallel for, stack variables in called subroutines.

#pragma omp parallel for private (list) makes a private copy for each thread for each variable in the list.

Values of private variables are undefined on entry and exit.

- firstprivate (list) initialiazes the variables in the list with the original values before entering the parallel construct.
- When using lastprivate (list) the thread that executes the last iteration/section updates the values of the variables in the list.
- threadprivate variables are global variables that are private throughout the execution of the program

Atomic applies to simple operations. The critical directive implments mutual exclusion in terms of execution anda tomic in terms of access to data.

#praga omp parallel if (expression) allows parallelism when it's beneficial.

#pragma omp parallel for reduction(+:result) sums the results from each thread.

The schedule directive allows work distribution among threads through

- schedule (static | dynamic | guided [,chunk])
- schedule (auto | runtime)

chunk divides iterations into chunks and distributes them in a reound robin fashion.
auto delegates the scheduling to the compiler. runtime delegates to the runtime through environment.

On #pragma omp task, tasks are added to a pool and eventually executed.

Scheduling Options

- Static scheduling, lower overhead but imbalance
- Dyanmic, larger overhead, useful for asymmetric workload
- Chunks, larger chunks reduce overhead and increase cache hit rate. However, small chunks allow finer balancing.

Static makes all tasks start at the beggining. Dynamic makes them start as needed.

Performance analysis and Debugging of parallel programs

The performance is dependent on

- Fraction of parallel runtime
- Thread management
- Data organization
- Load Balancing

To optimize OpenMP

- Minimize forks/joins
- Minimize synchronization
- Maximize private data

False sharing, caches are organized into blocks of contiguous memory locations (because of spatial locality), therefore it's possible for 2 processors to share the same cache block and not the same memory location within the block.
If one processor writes to its own block, it then causes the other processor's entire block to get updated/invalidated, which can affect performance.

To increase parallel fraction of work, it is best to parallelize the outermost loop of a nested loop.

Static loop scheduling in large chunks promotes cache and page locality but may not achieve load balancing.
Dynamic and interleaved scheduling can achieve good load balancing but cause poor locality.

Major overheads reported by ompp

- Synchronization, ex: waiting times to enter critical sections or acquire locks.
- Imbalance
- Limited Parallelism
- Thread management, time for creation/destruction of threads in parallel regions and overhead for signaling locks and critical sections.

Shared Memory Parallel

Possible conflicts

- Deadlocks, locked resource will never become free
- Livelocks, threads working on tasks they cant finish
- Race conditions, the output depends on the timing of the threads

Monitor Concept

When a thread arrives at the beggining of the monitor region, it is placed jnto an entry set for the associated monitor.
When it finishes, it exists and releases the monitor.
When executing a wait command, the thread releases the monitor and enters a wait set.
It will stay in the set until notified otherwise.
When a thread executes a notify, it keeps holding the monitor until it wants.

Transactions

Operations in a transaction either all occur or none occurs. In case of conflict, the program roll back to the state it was before the thread entered.

Shared Transactional Memory

- Increases concurrency (threads are not blocked)
- Conflicts only arise when more tha one thread asks access to the same memory position
- Conflicts are rare (small nmber of roll backs)

#pragma omp atomic makes up a transaction. It can depend on a condition (Conditional Critical Region - STM, Shared Transactional Memory). If the condition is not satisfied, the thread will be blocked until a commit has been made that affects the condition.

It has some issues,

- Overhead for conflict detection, computational and memory
- Overhead from commit
- Cannot be used in operations that can’t be undone

Foster’s Methodology

Task/Channel Model

Parallel computations is represented as a set of tasks that may interact with each other by sending messages through channels. Receiving is synchrnous but sending is asynchronous.

Foster’s Design

It entails for stages.

1) Partitioning

Divide the work into smaller primitive tasks.

2) Communication

Identify the communication between those tasks.

- Local, values shared by a small number of tasks
- Gloabl, values are requires by a big number of tasks

3) Agglomeration

Join primitive tasks into larger tasks.

4) Mapping

Assign tasks to processors.

Parallel Performance Analysis

$$Speedup = \frac{Sequential\ program}{Same\ program\ but\ paralllel}$$

$$\sigma = sequential\ computations$$

$$\phi = parallel\ computations$$

$$k = communication$$

$$Speedup <= \frac{\sigma + \phi}{\sigma + \phi/p + k}$$

$$Efficiency = speedup/processors$$

Amdahl’s Law

$$f = \frac{\sigma}{\sigma + \phi}$$

f is the fractions of operations in a computation that must be performed sequentially.

$$Speedup <= \frac{1}{f + \frac{1-f}{p}}$$

It does not take into account k. So it overestimates.
It’s focused only on decreasing the execution time, and not the size of the problem.

Gustafson-Barsis’ Law

$$f = \frac{\sigma}{\sigma + \phi/p}$$

f is the fraction of sequential computation in the parallel program.

$$Speedup <= p + (1 - p) * f$$

Predicts scaled speedup. In function of the problem size.
Still ignores k.

Karp-Flatt Metric

Experimentally determined serial fraction

$$e = \frac{\sigma + k}{\sigma + \phi}$$

e is the fraction of the original program that can’t be parallelized.

$$e = \frac{1/Speedup - 1/p}{1 - 1/p}$$

Isoefficieny

It's a way to measure scalibility.

Execution time of parallel program in p processors.

$$T(n,p) = \sigma(n) + \phi(n)/p + k(n,p)$$

Total overhead increasing with p. Time the rest of the processes are idle while one executes the sequential part. Plus communication overhead.

$$T_0(n,p) = (p-1) * \sigma(n) + p * k(n,p)$$

Isoefficiency relation.

$$T(n,1) >= Const * T_0(n,p)$$

As p increases, the work to be done (n) must increase in order to mantain the relation above.

$$C = \frac{\varepsilon(n,p)}{1-\varepsilon(n,p)}$$

$$\varepsilon(n,p) = Speedup/p$$

Scalibility Function

Suppose isoefficiency relation is n >= f(p)

$$\frac{M(f(p))}{p}$$

This is the memory usage by processor - scalibility function.
To mantain efficiency when increasing p, we must increase n.

When the scalibilty function complexity is constant, it's perfectly scalable.

Load Balancing

Mapping decisions:

1. Static number of tasks
    1. Structured Communication
        1. Constant computation time
           One task per processor (agglomerate tasks to minimize communication)
        2. Varies
           Cyclically map tasks to processors
    2. Unstructured Communication
       Use a static load balancing algorithm
2. Dynamic number of tasks
    1. Frequent communication
       Use a decentralized dynamic load balancing aloritm
    2. Little communication, many tasks
       Use a centrelized one.

Work pool or **processor farm** on centralized dynamic load balancing

- Master holds all tasks
- New tasks may be generated during execution
- When idle, slave requests a task to master
- Master selects and sends the tasks
- Specialized slaves can be considered
- Master can hold global data

The master can become a bottleneck as it only issues one task at a time. Ok if few slaves and/or intense tasks. Bad for finer grained tasks and/or many slaves.

Termination Detection

1. No messages are in transit at time t
2. How long till assuming it has finished???

Solution

- Send/Receive acknowledgement messages that determine active/inactive states.
- Ring termination

Combinational Search

Finding one or more optimal or suboptimal solutions in a defined problem space.

Backtrack search

Uses depth first search to consider alternative solutions to combination seach problems.
At level k, we have b^k nodes. Up to level k, we have

$$\frac{b^{k+1} - b}{b - 1}$$

nodes. b is the branching factor.

On parallel backtracking, if $p = b^k$, each process is assigned a subtree. Else, define a level m up to which all processes execute redundant search and at level m, assign new subtrees.

Branch and Bound

Performs depth first search but is able to prune its search by using upper and lower estimated bounds of the cost function.

Estimate a bound of moves to make. If the depth+bound are higher than an existing solution discard the subtree.

On parallel branch and bound may examine unnecessary nodes because each process is seaching locally its subtree.
It reduces the amount of wasted work but increases communication overhead

Monte Carlo Method

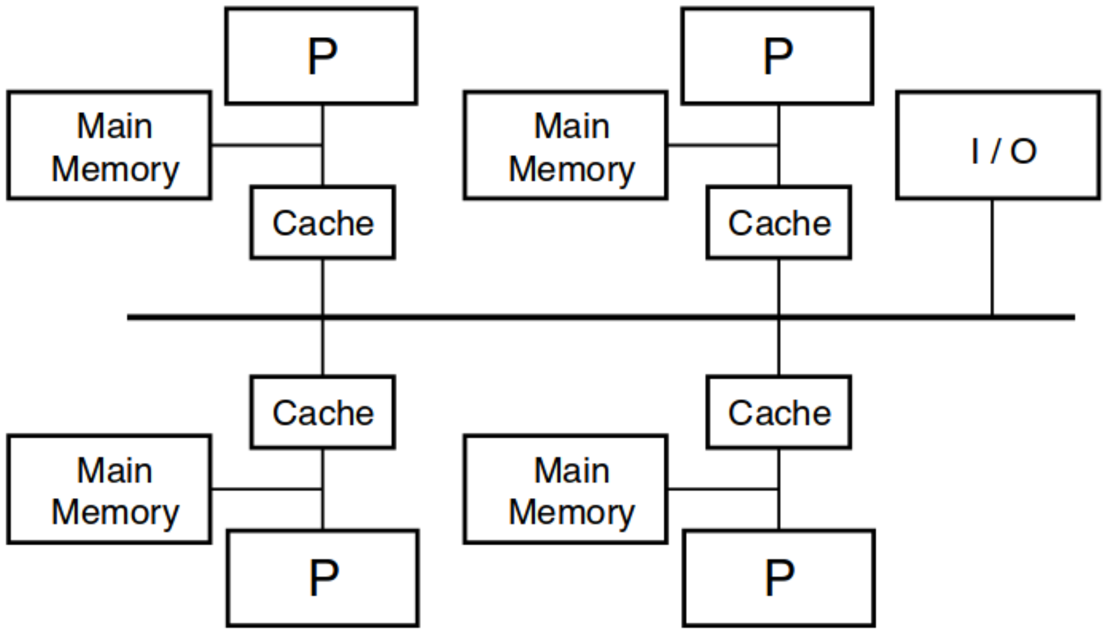
Solve a problem using statistical sampling.

1. Define a domain of possible inputs.
2. Generate inputs randomly from a probability distribution over the domain
3. Perform deterministic computation on the inputs
4. Aggregate results

Accuracy increases at a rate of $1/\sqrt{n}$. n being the inputs.

ccNUMA

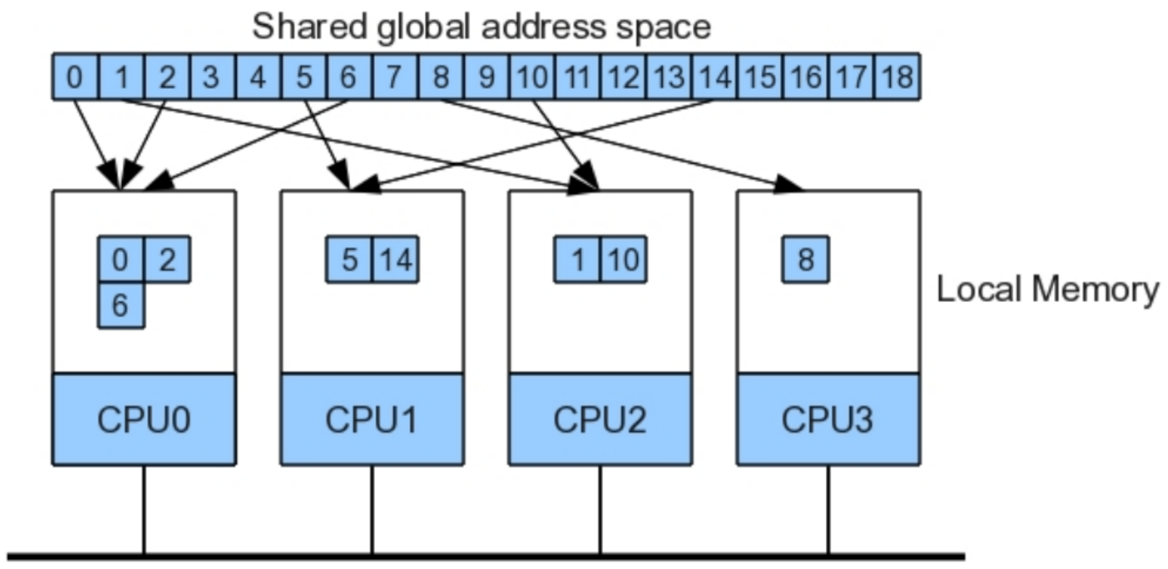
UMA is limited on scalibility, due to contention on accessing memory. Multicomputers have high communication overheads. **Cache-Coherent** NUMA offers an intermidiate solution.



- Highly Scalable
- Memory bandwidth with computational power
- Cache coherent possible due to shared global bus
- Memory is physically distributed throughout the system memory and peripherals are globally addressable.
- Local memory accesses are faster than remote accesses.
- Local accesses on different nodes do not interfere with each other.

Directory-based protocols can become a bottleneck, ccNUMA is distributed.

It can be implemented in a similar fashion to the virtual memory scheme. Logic addresses are divided into pages.



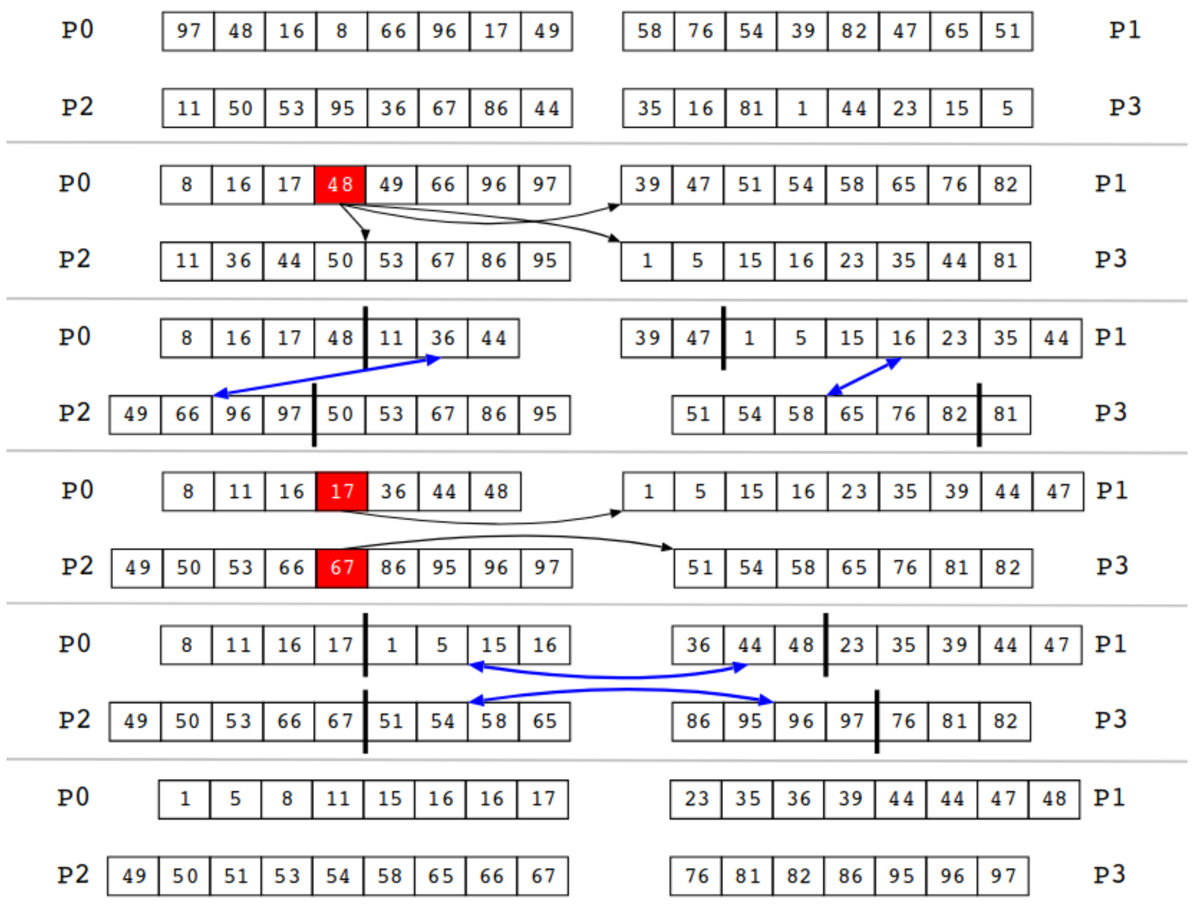
Parallel Sort

Quicksort balances the list sizes very poorly.

Hyperquicksort

Sorts the elements before broadcasting pivot.

1. Sort elements in each process
2. Select a median as pivot and broadcast it.
3. Each process n the upper half swaps with a partner in the lower half
4. Recurse on each half



Initial quicksort step O(log (n/p) * n/p)

Each remaining sort O(n/p)

Pivot broadcast O(log p)

Array exchange O(n/p)

Limitations

It’s assumed lists remain balanced. As p increases, each list decreases.

Thus, the likelihood of lists being unbalanced is larger.

Unbalanced lists lower efficiency.

A better solution is to sample values from all processes before choosing the pivot.

Parallel Sorting by Regular Sampling

- Keeps list sizes balanced
- Avoids repeated communication of keys
- Doesn’t require the number of processors to be a power of 2

1. Each process sorts its elements
2. Each process selects regular samples of their list, in a total of p^2 samples
3. One process gathers all of these, sorts and chooses p-1 pivot values and broadcasts
4. Each process partitions its list into p pieces using the pivot values.
5. Each process sens its partitions to other processes
6. Each process merges and sorts its partitions

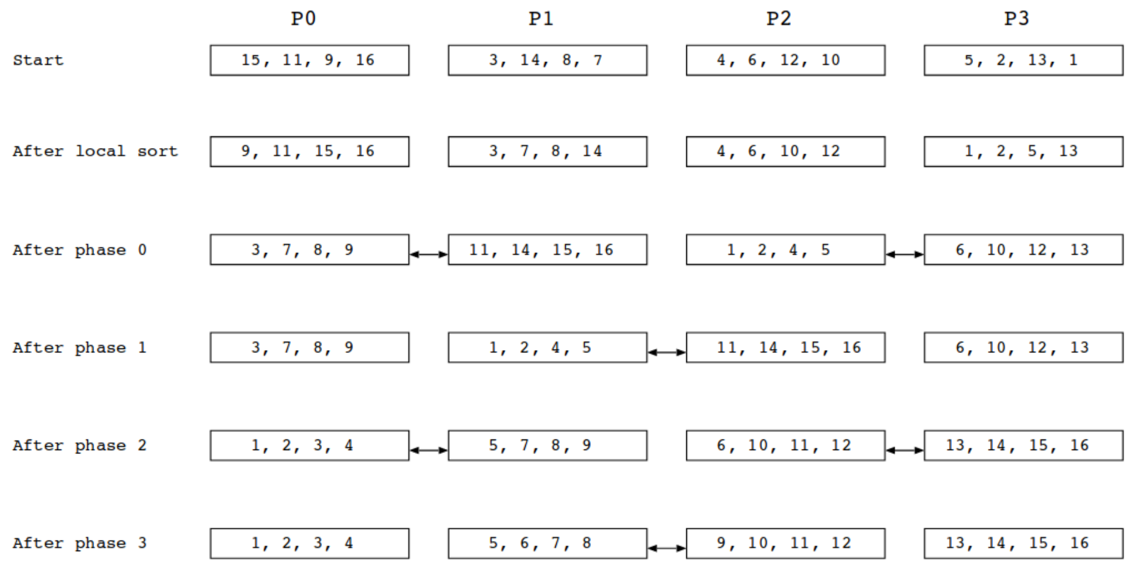
Initial quicksort step O(log (n/p) * n/p)
Sorting samples $O(p^2 log p^2)$
Merging subarrays O(log (n/p) * p)
Gather sampls O(log p)
Pivot broadcast O(log p)
Array exchange O(n/p)

Parallel Odd-Even Transportation Sort

Like bubble sort but with alternate phases odd and even.

In the parallel case, each processor gets N/P of the array.
And sorts it locally with this method.
The processors then sort between each other in an odd and even fashion reptitevely till the end.

Initial quicksort step O(log (n/p) * n/p)
Sorting smaller/larger values in each phase O(n/p)
Communication O(n/p)
Number of phases p



Comparison

- Hyperquicksort O(log (n) * n/p)
- PSRS O(log (n) * n/p) (but better load balancing than the previous)
- Odd-Even Sort O(log (n) * n/p + n) (perfect scalibility = C)

To Read: 23, 11, 12, 17, 18, ~19

Exams

2nd Exam 14-15

Group I

1) Write two possible outputs when executng the following code using two threads. The two solutions should differ in at least 4 lines.

```
#pragma omp parallel for schedule(dynamic,2)
for(i = 0; i < 6; i++) {
    printf("%i, %i\n", omp_get_thread_num(), i);
}
```

The threads get assigned tasks with chunks of size 2, ends it and then gets another chunk.
For example,
Option 1
1, 0
2, 2
2, 3
1, 1
2, 4
2, 5
Option 2
2, 0
1, 2
2, 1
2, 4
2, 5
1, 3

- static, defines which tasks belong to whom at compilation.
- dynamic, gives one task to each thread and assigns the next and next along the execution.
- dynamic chunk, is the same as dynamic but assigns chunks.
- guided, has chunks that vary on size along the execution.

2) In OpenMP, discuss the usage of the following directive. How does it work? When do you use it?

```
#pragma omp barrier
```

We use this directive in order to make all the threads wait for each other before continuing with the execution.

3) Parallelize the code below using OpenMP directives.

```
found = 0;
for(i=0; i<N; i++) {
    if(a[i] == value) {
        found+=1;
        printf("found so far: %i values\n", found);
    }
}
```

Using the for and atomic directive,

```
found = 0;
#pragma omp parallel for
for(i=0; i<N; i++) {
    if(a[i] == value) {
        #pragma omp atomic
        found+=1;
        printf("found so far: %i values\n", found);
    }
}
```

Group II

1) Consider a system with four MPI tasks running. Implement the function MPI_Scatter from node 0 using MPI_Send and MPI_Recv. Assume that an array V of integers of size N is being sent and received in array D.

MPI_Bcast sends the same piece of data to all processes.
MPI_Scatter sends chunks of an array to different processes.

```
void MPI_Scatter() {
    ...
    if(id == root) {
        MPI_Init_thread(argc, arv, MPI_THREAD_MULTIPLE,)
        #pragma omp parallel for
        for(i=0; i<p; i++) {
            MPI_Send(V)
        }
    }
    else {
        MPI_Init()
        MPI_Recv(D)
    }
}
```

2) Consider the following code. Suggest a more efficient alternative code.

```
for(i = 0; i < N, i++) {
    MPI_Recv(x, 10, MPI_DOUBLE, id-1, TAG, MPI_COMM_WORLD, &status);
    y = comp(x);
    MPI_Send(y, 10, MPI_DOUBLE, id+1, TAG, MPI_COMM_WORLD);
}
```

By using pipeline parallelism, it's possible to obtain a more efficient code. Instead of having each processor waiting for the previous to receive 10 doubles, compute them and send them to him, we could be receiving one double in one thread of each processor and computing and sending in another thread. The computation would be a 10 times loop but only over one double. Alternatively, this comutation could be divided into tasks.

3) What is the purpose of the keyword MPI_THREAD_SERIALIZED in the initialization of MPI?

MPI_THREAD_SERIALIZED means multiple threads are running and can act on MPI but only one at a time.
MPI_THREAD_FUNNELED mean multiple threads are running but only one of them is entitled to use MPI.
MPI_THREAD_MULTIPLE means everything is possible.
MPI_THREAD_SINGLE, means only one thread executes.

Group III

1) Consider a program that runs 9 times faster on a parallel system with 10 processors compared to the serial version. Assume the paralell program only executes in two modes, either sequentially or fully parallel.

a) Compute what fraction of the parallel execution time is spent in the sequential mode.

Using Amdahl's law,

$$Speedup <= \frac{1}{f + \frac{1-f}{p}}$$

$$9 <= \frac{1}{f + \frac{1-f}{10}}$$

f = 1/81 = 1%

b) Compute the maximum achievable speedup as the number of processors increases.

$$lim_{p->inf} \frac{1}{f + \frac{1-f}{p}} = \frac{1}{f} = 81$$

2) When using 4 processors, the measured speedup over the sequential execution time of a program is 2. Assuming that the inefficiency of the parallel program is dominated by a large serial fraction, indicate a reasonable estimate for the speedup on 8 CPUs. (hint: use the Karp-Flatt Metric)

$$e = \frac{1/Speedup - 1/p}{1 - 1/p}$$

$$e = \frac{1/2 - 1/4}{1 - 1/4} = 0.333...$$

For 8 CPUs,

$$0.333 = \frac{1/Speedup - 1/8}{1 - 1/8}$$

Speedup = 2.4

3) Consider a problem with a sequential algorithm that runs in Θ(n√n) and with a parallel implementation that runs in Θ(n√n/p * log p) with p processors and whose overhead (communication + redundant computation) per processor is given by Θ(n). If the required memory grows with n, compute the scalability function for this parallel algorithm. Discuss the result obtained.

Execution time of the parallel program in p processors.

$$T(n,p) = n\sqrt{n}/p * log\, p + n\sqrt{n} + n$$

Total overhead

$$T_0(n,p) = (p-1) * n\sqrt{n} + pn \sim pn$$

Since it’s only one processor, there is no communication. T(n,1) is the sequential algorithm.

$$T(n,1) = n\sqrt{n}$$

Isoefficiency relation is

$$T(n,1) >= C * T_0(n,p)$$

$$n\sqrt{n} >= C * pn$$

$$n >= (C * p)^2$$

$$f(p) = (C * p)^2$$

$$M(f(p))/p = (C * p)^2/p = C^2 * p = \Theta(p)$$

To mantain the level of efficiency when p increases, it’s necessary to increase the problem size, n.

IV

1) The Work Pool (aka Processor Farm) model is used to optimize the load balancing of a parallel application. Describe the situations where it should be employed and why. What is the main reason why it is not used for all applications?

The master can become a bottleneck as it only issues one task at a time. Ok if few slaves and/or intense tasks. Bad for finer grained tasks and/or many slaves.

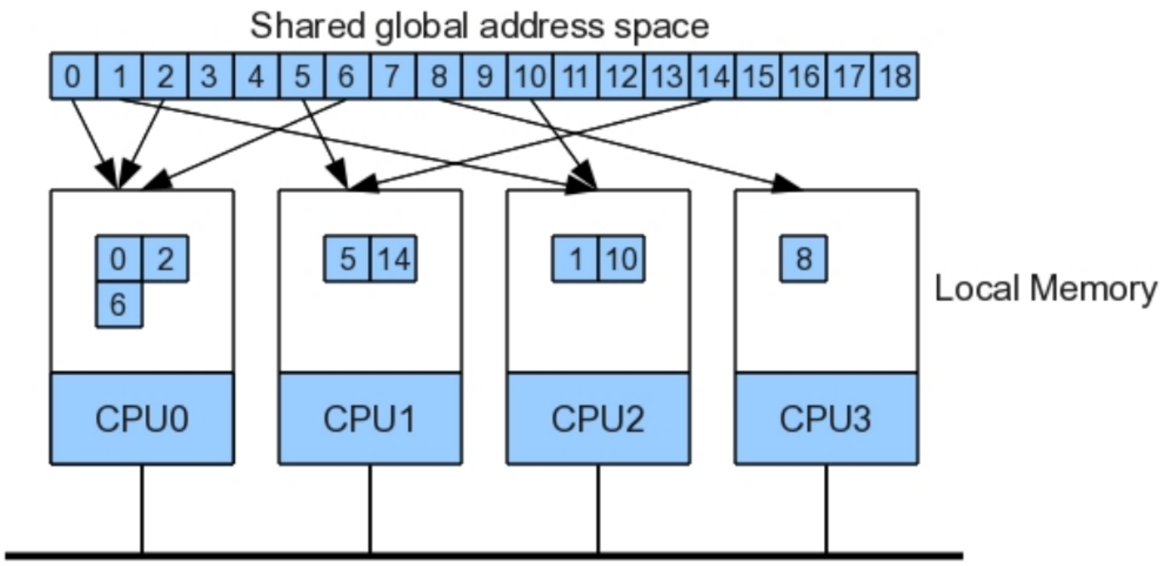
It should be used in situations of little communication and many or computationally intensive tasks. If the master has to gather tasks for the task queue as well as serve the slaves in can become very time consuming.

2) In a Parallel Branch and Bound Search implemented on a distributed system, discuss the relative advantages of having a single priority queue versus multiple priority queues (one at each node). What's the best solution?

Having a single priority queue entails less communication between the nodes, but a multiple priority queue allows for less time wasted on accesses and a finer grained depth search. The best is the multiple priority queue.

3) Explain why is cache-coherent NUMA (ccNUMA) considered as shared-memory programming. How is this achieved?

It can be implemented in a similar fashion to the virtual memory scheme. Logic address spaces are divided into pages.



1st Exam 14-15

Group I

1) Write the code to parallelize the outer loop of the following program using OpenMP commands. Assume that function random1000() returns a random number between 0 and 999.

```
for(i = 0; i < N; i++){
    M = random1000();
    for(j = 0; j < M; j++)
        a[i][j] *= 2.0 + a[i][j];
}
```

It could be something like,

```
#pragma omp for private(M, j)
for(i = 0; i < N; i++){
    M = random1000();
    for(j = 0; j < M; j++)
        a[i][j] *= 2.0 + a[i][j];
}
```

2) In the OpenMP directive omp parallel for what is the purpose of the, optional, chunk parameter? When should a larger or smaller value of chunk be used?

The chunk parameter allows tasks to be assigned a defined or real-time determined data size (chunk) when in lack of work on dynamic scheduling or pre-assgined on static scheduling, instead of the default next data unit to be computed.

The workload can be different according to what is to be computed.

If we have a lot of data, it’s easier to assign big chunks to reduce communication.
If we have a lot of process, it’s better to assign small chunk to reduce execution time.

3) Compare from a programmer’s point of view a UMA machine (uniform memory access) versus a DSM machine (distributed shared memory). Make sure you explain the mechanisms that support these systems.

UMA contains a shared memory and peripheral bus between processors.
DSM is implmented via virtual memory scheme accessing the memory of each CPU.

A distributed shared memory machine gathers the memory from all the CPUs, using a virtual memory scheme. Whereas UMA because it requires a cache coherence system, limits the memory. Thus, DSM is way more scalable.

Regarding access time, DSM is also faster since it doesn't have to access a bus like UMA does.

Group II

1) Consider that we want to parallelize the following code:

```
for(i = 0; i < N; i++)
    b[i] = DoComputation(a[i]);
```

Write an MPI program that implements this code using a workpool (aka master/slave) architecture. To simplify, assume that there are only 2 slave processes.

```
void main() {
    int a[SIZE], b[SIZE], recv;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    if(id == 0)
        // define a

    MPI_Scatter(&a, 1, MPI_INT, &recv, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if(id != 0)
        b = DoComputation(recv);

    MPI_Gather(&b, 1, MPI_INT, &recv, 1, MPI_INT, 0, MPI_COMM_WORLD);

}
```

2)

a) What is the purpose of the tag parameter in MPI's MPI_Send function?

It's a label associated with the message being sent, to be matched on the receiving end. It's purposes are to allow the receiver to ignore messages with the wrong tag/detect error. Or to organize different message types into different buffers to define different behaviors for each.

b) Is there a way to circumvent this "restriction"? Indicate a situation when this may be helpful.

Yes by using a WILDCARD, MPI_ANY_TAG. It may be helpful to receive several messages from the same source into the same buffer when the order is irrelevant. Or define different behaivor according to the incoming tag.

c) Discuss why you should not avoid using this tag parameter.

The parameter is useful for debugging and also to extend the code in the future.

3) In a good MPI implementation, what is the complexity of the MPI_AllGather function? Justify.

The MPI_AllGather function uses the hypercube network to send the messages to all the processors, whoose complexity is log(p).
To receive the complexity is m(p-1), where is the message size in each processor. Thus the final complexity is $O(log(p) + m(p-1))$.

Group III

1) Consider a problem with a sequential algorithm that runs in $O(log^2 n * n)$ and with a parallel implementation that runs in $O(log^2 n * n/p)$ with pprocessors and whose overhead (communication + redundant computation) per processor is given by $O(log\ n * n)$. If the required memory grows with n^2, compute the scalability function for this parallel algorithm. Discuss the result obtained.

Sequential Algorithm: $O(log^2 n * n)$
Parallel Algorithm
- Complexity $O(log^2 n * n/p)$
- Communication $O(log\ n * n)$

$$T(n,p) = log^2 n * n + \frac{log^2 n * n}{p} + log\ n * n$$

$$T_0(n,p) = (p-1) * log^2 n * n + p * log\ n * n$$

$$T(n,1) = 2 * log^2 n * n$$

$$T(n,1) >= C * T_0(n,p)$$

$$2 * log^2 n * n >= C * p * log\ n * n$$

$$2 * log\ n >= C * p$$

$$n >= 2^{C*p/2}$$

Since $M(n) = n^2$, $M(2^{C*p/2}) = 2^{C*p}$.

And the scalibility function is

$$\frac{2^{C*p}}{p}$$

The scalibility function shows how memory usage per processor must grow to mantan efficiency. It heavily relies on C, but because its exponential, it has bad scalibility.

2) Although the Amdahl’s Law defines a ceiling for the maximum speedup of a parallel program, the same speedup can be computed using the Gustafson-Barsis’ Law which presents no such limitation. Discuss the main underlying difference between these two expressions that explains this contradiction.

In Amdahl’s law

$$f = \frac{\sigma}{\sigma + \phi}$$

f is the fractions of operations in a computation that must be performed sequentially.

$$Speedup <= \frac{1}{f + \frac{1-f}{p}}$$

It’s focused only on decreasing the execution time, and not the size of the problem.

Whereas on Gustafson-Barsis’ Law

$$f = \frac{\sigma}{\sigma + \phi/p}$$

f is the fraction of sequential computation in the parallel program.

$$Speedup <= p + (1-p) * f$$

Predicts scaled speedup. In function of the problem size.
Although both ignore the communication time.

3)

a) What is the value of the Experimentally Determined Serial Fraction for a system with ideal speedup?

The effieciency = speedup/processors, for a system to have ideal speedup, it mesans having 100% efficiency, so speedup = processors.

The **Experimentally determined serial fraction**

$$e = \frac{\sigma(n) + k}{\sigma(n) + \phi(n)}$$

$$e = \frac{\frac{1}{Speedup} - 1/p}{1 - 1/p}$$

$$e = \frac{1/p - 1/p}{1 - 1/p} = 0$$

b) In general, why does the value of the Experimentally Determined Serial Fraction tend to increase?

It tends to increase with p, since it requires more communication overhead.

Group IV

1)

a) Why are Monte Carlo methods amenable to parallel programming?

On Monte Carlo method, inputs are gathered and also randomly generated, and then are deterministically computed. FInally they are aggregated.

This procedure is very easily parallelizable, since the inputs are not dependent.

b) What is the main difficulty when using Monte Carlo in parallel programming?

In the Monte Carlo method, the statistical output accuracy depends on a good random generation of inputs. This generation done parallely is the biggest difficulty.

It can be done in a centralized way where a master generates all random numbers and distributes. However if there are a lot of processors and the tasks aren't very intense, it can become a bottleneck.

In a decentralized way, the processes can use the same random number generator or create blocks of random samples of size p, or other methods. Anyways, it is hard to predict how many inputs are necessary and if they are being random enough, giving communication overhead.

2) Combinatorial search is usually performed through a tree. In a parallel implementation, typically each processor is assigned a subtree to search. Discuss what is the main issue that needs to be addressed, and how is this generally handled.

The main issue is balancing each processors workload. Some tasks may end earlier than others, and so more work has to be assigned to them. This can be generally solved by a master/slave approach, a ring approach orseveral masters in the tree.

3) What is the main reason why the Foster's design methodology is popular? Give a brief description of each step, indicating its main objective and how to achieve it.

Foster's Desgin Methodology delays machine dependent decisions to later stages. Concurrency is dealt with early

It includes four stages, partioning of data into small tasks, communication (identification of the communication patterns among the tasks), agglomeration of small tasks into larger tasks and mapping tasks to processors.

1st Exam 16-17

Group I

1) Consider the following parallel version of a function that computes the max on a vector of unsorted integer numbers:

```
int max(int a[], int N) {
    int i, m = a[0];
    #pragma omp parallel for
    for (i = 0; i < N; i++)
        #pragma omp critical
        if(a[i] > m)
            m = a[i];
}
```

a) The above implementation is very inefficient. Explain why.

For threads to assign a new maximum of their subarray they have to access a critical region which is time consuming.

b) Rewrite the function to make it as efficient as you can, having in mind its execution in a machine with a large number of cores.

```
int max(int a[], int N) {
    int i, m = a[0];
    #pragma omp parallel for reduction(max:m)
    for (i = 0; i < N; i++)
        if(a[i] > m)
            m = a[i];
}
```

2) Consider the following two code fragments:

```
#pragma omp parallel sections
{
    #pragma omp section
        f1();
    #pragma omp section
        f2();
    #pragma omp section
        f3();
}

#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        f1();
        #pragma omp task
        f2();
        #pragma omp task
        f3();
    }
}
```

Explain the differences between their execution.

On the first snippet all threads enter the pragma and then each of them executes one section. On the second, only one enters the omp single pragma and that thread calls others to execute tasks.

3) Write down a valid output produced by the code below, assuming that during its execution the value of the environment variable OMP_NUM_THREADS is 6

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

int main(int argc, char* argv[]) {
    int i, v[10];

    #pragma omp parallel for schedule(static,1)
    for(i = 0; i < 20; i++)
        v[omp_get_thread_num()] = i;
    #pragma omp parallel
        #pragma omp single
        for(i = 0; i < omp_get_num_threads(); i++)
            printf("%d\n", v[i]);
        return 0;
}
```

18
19
14
15
16
17

Group II

1) In an optimized implementation of the MPI function MPI_Bcast (broadcast), how many messages does the source process need to send? Explain. (assume P represents the number of processes and n the size of the array to send)

In the optimized version of MPI_Bcast, it is used a hypercube network, and so the complexity to send one message is log(p). Since the array to send has size n, the final complexity is n*log(p).

2) The Foster's design methodology consists of four steps, the second of which is "Communication". What are the objectives of this step and in what way does it help in achieving a more efficient implementation?

The communication is the stage where the communications between the primitve tasks are identified. It helps at defining which tasks should be aggregated into bigger tasks to reduce communications's overhead.

3) Consider the following piece of MPI code, where: variable id holds the identifier of the MPI task; P is the number of processes; N is the size of array arr.

```
for(i = BLOCK_LOW(id, P, N); i < BLOCK_HIGH(id, P, N); i++) {
    printf("Proc %d: %d, %c\", id, i, arr[i]); fflush(stdout);
}
```

Modify the code above such that:

a) the indexes over all the pieces of the array are printed in order, i.e., no i+ 1 before an i.

```
for(j=0; j<nprocs; i++) {
    MPI_Barrier(MPI_COMM_WORLD);
    if(j == id) {
        for(i = BLOCK_LOW(id, P, N); i < BLOCK_HIGH(id, P, N); i++) {
            printf("Proc %d: %d, %c\", id, i, arr[i]); fflush(stdout);
        }
    }
}
```

b) each process, in order, print one position at a time,i.e., process 0 prints index 0, then process 1 prints index 0, and so on until all processes have printed position 0, then process 0 prints index 1, and so on.

```
for(i = 0; i < N; i++) {
    printf("Proc %d: %d, %c\", id, i, arr[i]); fflush(stdout);
    MPI_Barrier(MPI_COMM_WORLD);
}
```

Group III

1) The Experimentally determined serial fraction metric is given by

$$e = \frac{\sigma(n) + k(n,p)}{\sigma(n) + \phi(n)}$$

Describe how we can use this metric to efficiently optimize a parallel program

This metric determines what part of the original program is not parallelizable.

Knowing the execution time of the parallel program is

$$T(n,p) = \sigma(n) + \phi(n)/p + k(n,p)$$

And the sequential is

$$T(n,1) = \sigma(n) + \phi(n)$$

We can derive the expression,

$$e = \frac{1/Speedup - 1/p}{1 - 1/p}$$

Contrarly to the other metrics like Amdahl's and Barsis', this one takes into account the size of the problem (number of processors in use) and also the communication overheads - basicaly all the parallel overheads. Allowing for the analysis of the source of the parallel inefficeincy.

2. Discuss the possibility of achieving an efficiency ε > 1 in a parallel system.

An ideal efficiency (=1) means the number of processors equals the speedup.
ε = Speedup/Processors
Speedup > number of processors is a superlinear speedup. And it's only achievable when there is nothing sequential and some operations like cache hits, memory pattern hits or flaws happen. It is originated from situations not taken into account.

$$S = \frac{1}{f + \frac{1-f}{p}}$$

$$lim_{p->inf} S = \frac{1}{f}$$

In metrics like Amdahl's and Barsis' this can easily happen since they both dont take into account communication, and the first doesn't take the number of processors into account.

3) Consider the problem of computing the sum of each row of a large n*n matrix. A given parallel MPI implementation, divides the matrix into smaller submatrices, in a checkerboard configuration, computes the sum of each row for such submatrices, and then share this local result such that the overall sum can be computed.

a) Derive the isoefficiency relation for this parallel implementation, assuming the number of processors to be p.

Each task is associated with a rectangular block of the matrix. Assuming it's a sqaure for simplicity. Each block has a side size of $\sqrt(n^2/p)$.

$$\sigma(n) = n^2$$

$$\phi(n)/p = n^2/p$$

Since it's a reduce operation, the complexity of summing a row is log(p). Having $n/\sqrt{p}$ rows per processor, the complexity of communication is

$$k = \frac{n * log\ p}{\sqrt{p}}$$

So,

$$T(n,p) = n^2 + n^2/p + \frac{n * log\ p}{\sqrt{p}}$$

$$T(n,1) = 2n^2 \sim n^2$$

$$T_0(n,p) = p * \frac{n * log\ p}{\sqrt{p}})$$

Isoefficiency relation,

$$n >= C * \sqrt{p} * log\ p$$

b) Is this implementation scalable? Justify using the scalability function.

Since it's a n*n matrix, we assume $M(n) = n^2$.

So the scalibility function is

$$(C * \sqrt{p} * log\ p)^2/p = C^2 * log^2 p$$

To mantain efficiency, the memory per processor must increase with the increase of $log^2 p$.

Group IV

1) (Has been answered before)

2) Branch and bound algorithms for optimization problems keep a list of touched nodes in the search tree, ordered in terms of the most promising nodes based on a lower-bound estimate. In a distributed implementation, each computational node keeps a local list. State what are the negative consequences of this approach and how they can be mitigated.

The major problem is the communication overhead. One approach would be to divide the tree into independent sub-trees that don't need to share the list. However, this would increase the wasted time searching redundant paths.

3) A Maximal Independent Set I of a graph G(V,E) is a set of vertices I ⊂ V such that no pair of vertices in I is connected via an edge in G and no other vertex in V can be added to I without violating this rule.

Luby's algorithm provides a good parallel solution to finding I:

1. Start with an empty set.
2. Assign a random number to each vertex.
3. Vertices whose random number are smaller than all of the numbers assigned to their adjacent

vertices are included in the MIS.
4. Vertices adjacent to the newly inserted vertices are removed.
5. While graph not empty, Goto 2.

Analyze its implementation under:

a) shared memory
b) distributed memory

In the parallel Luby's algorithm, each processor removes set of vertixes and its adjacents from the graph, saving the node with the smallest value for the MIS.

By using shared memory, the conflicts on the graph data are the big issue.
Shared memory has a uniform access time to memory. The communication is implicit, and so easier to program. But due to the conflicts of threads trying to access the same memory, it needs contention. And that, limits scalibility.
So, for small graphs, it could be ok and better than distributed memory since it wouldn't deal with big access times or big startup/termination overheads or communication overheads, but for bigger ones, the access time would just be too much.

Distributed memory is harder to deal with, messages are explicit and have to be dealt with by the programmer, communication overhead is a problem. However, it's highly scalable. For example, using a ring approach, the graph could be sent over the ring till it's empty.

1st Exam 17-18

Group I

1) In the #pragma omp parallel for directive, the schedule clause enables the specification of how the iterations of a loop should be scheduled, that is, allocated to threads. Describe how each of the following scheduling types work and their parameters (for the ones that have): static, dynamic, guided. For those scheduling types that have optional parameters, you should describe the behavior in the presence and absence of such parameters.

schedule (static | dynamic | guided [,chunk])
schedule (auto | runtime)

On static [,chunk], iterations are divided into blocks of size chunk and these blocks are assigned to the threads in a round-robin fashion.
In the absence of chunk, each thread executes approx N/P chunks.

On dynamic [,chunk], a chunk iterations are assigned to each thread (defaults to 1, if not specified).

When a thread finished it starts on a new chunk block. The last block may contin fewer iterations.

On guided [,chunk], same behavior as dynamic, but threads are assigned different block sizes proportional to the number of unassgined iterations divided by the number of threads, decreasing to chunk (minimum size, or 1 if not defined).

auto is a decision regarding scheduling delegated to the compiler and/or runtime system.

runtime, iteration scheduling is set at runtime through environment with OMP_SCHEDULE.

Larger chunks mais increase cache hit rate and reduce overhead.
Small chunks allow finer balancing of workload.

2) Consider the function int ones(int*m, int n) which receives a pointer m to the first element of a square matrix and its number of rows/columns n, and computes que number of elements with value 1. The matrix is organized in memory in row major order (i.e. by rows). Each element of the matrix can only assume the value 0 or 1. Provide the best parallel implementation for this function using OpenMP.

The best parallel implementation for matrix operations generally is checkerboard. Here the matrix gets divided into rectagles composed by subdivisions of rows.

Since the objective is to know the number of ones in the whole matrix, it can be implemented with a reduction function summing each rectangle block.

Since each rectangle block has $n/\sqrt{p}$ side size, the communication complexity is $n * log(p)/\sqrt{p}$.
The complexity of the sequential algorithm is n^2.
And the parallel complexity in each block is n^2/p.
Masking the overall complexity $O(n^2/p + n * log(p)/\sqrt{p})$.

This can be accomplished using #pragma omp parallel for reduction(+:ones).

3) What are the negative effects of false sharing, associated to shared-memory systems, and how can this problem be dealt with?

False sharing occurs when two or more processors reference the same shared data and one of them modifies its copy of data. In this case, all the others will gather stale copies of that altered data in their caches although they won't even use it.
This is the way the machines that are cache coherent ensure that the processor accessing a memory location receives the most up-to-date version of data, and happens because the machine has no way of knowing if the processor is using that data or not, the whole cache gets updated.

It **degrades performance**. This problem can be reduced by making use of private data as much as possible and using optimization features fromm the compiler.

4) What is the behavior of #pragma omp parallel if (expression)? When should it be used?

It parallelizes the execution only if the condition on the expresion is verified. It should be used in cases where parallelization is not needed, for example if the workload is too small or not very intense (low computational effort generally).

Group II

1) What is the difference, in terms of behavior and usage, between the MPI_Recv and MPI_Irecv methods? Under what circumstances should one use either of them?

MPI_Irecv is a non-blocking operation, it initates the communication and then it proceeds while the processor is doing other useful work. It is useful for communications that take up a long time and are not necessary for the execution of the rest of the program. They are very useful to avoid deadlocks.

Whereas MPI_recv is blocking, and only returns when the buffer has been filled with valid data.

2) Let A be a large array of N single digit integer values, i.e. varying between 0 and 9 inclusive. Write an MPI routine that efficiently computes the mode and prints the result. Assume that, initially, A is only available to the process with rank 0. For simplicity, you may also assume that A can be evenly distributed among the processes, i.e. N/P = C. Explicitly include in your code the data transfer routines.

An efficient way of doing so, would be to firstly sort the array by using an efficient parallel sorting method, for example Parallel Sort with Regular Sampling (PSRS) which has a good load balancing. And after it's done let rank 0, determine the mode, with a complexity of N.

Or maybe better, would be to partition the array in N/P, distribute it and each processor would send back an array with the count of each number in the sub-array from 1-9. And then the master would make a reduction of sum and find max.

Implementing this second option,

```
int count[10], count_sum[10];

MPI_Scatter(&A, 1, MPI_INT, &subarray, 1, MPI_INT, 0, MPI_COMM_WORLD);

if(id != 0) {
    for(i=0; i<subarray.size(); i++) {
        count[subarray[i]]++;
    }
}

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

if(id == 0)
    findMax(count_sum);
```

3) Consider the following MPI code

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int** a = (int**)malloc(N*sizeof(int*));
for (int i = 0; i < N; ++i) {
    a[i] = (int*)malloc(N*sizeof(int));
}

if (rank == 0) {
    read(a, N);
}

for (int i = 0; i < N; ++i) {
    MPI_Bcast(a[i], N, MPI_INT, 0, MPI_COMM_WORLD);
}
```

a) The code above is very inefficient. Explain why

Complexity of allocating N^2
Complexity of reading N^2
Complexity of sending is N^2 * log(p)
This is very bad.
As well as that by sending a, row by row, instead of the whole a, its produces overhead.

b) Rewrite the code above to make it as efficient as you can.

I propose a code where each process creates and reads part of the matrix, and then gathers all to all.

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int** a = (int**)malloc(N/p *sizeof(int*));
for (int i = 0; i < N/p; ++i) {
    a[i] = (int*)malloc(N/p *sizeof(int));
}

if (rank == 0) {
    read(a, N/p);
}

MPI_AllGather(a_total, 1, MPI_INT, a, 1, MPI_INT, MPI_COMM_WORLD);
```

Group III

1) The sequential implementation of an algorithm runs in $O(n^2 log\ n)$. The computation time of its parallel version is $O(log\ n * n^2/p)$. The parallel overhead is $O(n * log\ n * log\ p)$. The memory increases with n^2.

a) Compute the isoefficiency relation for this parallel implementation. Explain the meaning of the isoefficiency relation.

$$T(n,p) = n^2 log\ n + log\ n * n^2/p + n * log\ n * log\ p$$

$$T(n,1) = 2 * n^2 log\ n \sim n^2 log\ n$$

$$T_0(n,p) = p * n * log\ n * log\ p$$

$$T(n,1) >= C * T_0(n,p)$$

$$n >= C * p * log\ p$$

The isoefficiency relation means that to mantain the efficiency, when p increases, n must also increase.

b) Compute the scalability function and discuss whether this implementation is scalable of not.

$M(n) = n^2$
$M(C * p * log\ p) = C^2 * p^2 * log^2 p$
Scalibility is $C^2 * p * log^2 p$

Which means it has a bad scalibility. It shows how memory per processor must grow to mantain the efficiency.

2) Consider that for the sequential execution of a given program the relation between the time spent executing purely sequential code and the time spent executing completely parallelizable code is α. Assuming that all the code is either purely sequential or completely parallelizable, and according to Amdahl's Law, what is the maximum speedup achievable by parallelizing this program?

$$S = \frac{1}{f + \frac{1-f}{p}}$$

$$lim_{p->inf} S = 1/f$$

(??)

3) Benchmarking a parallel program on 1, 2,..., 8 processors produces the following speedup results:

p	2	3	4	5	6	7	8
ψ	1.87	2.61	3.23	3.73	4.14	4.46	4.71

What is the experimentally determined serial fraction, proposed by Karp and Flatt? Resorting to this metric, and explaining your reasoning, discuss what is the main factor that is hurting the parallelization.

The experimentally determined serial fraction is,

$$e = \frac{\sigma + k}{\sigma + \phi}$$

$$e = \frac{1/S - 1/p}{1 - 1/p}$$

$$S = \frac{\sigma + \phi}{\sigma + \phi/p + k}$$

For p = 2, e = 0.07, for p = 8, e = 0.69.
Because e is increasing, this means k is increasing with the number of processors.

Group IV

1) In the Backtrack Search algorithm a depth-first search on a tree is usually used. However, in a parallel implementation, in general, all tasks perform the expansion of the first levels of the tree, which leads to redundant operations. What is the reason for this apparent wasteful computation?

The computation of the first levels of the tree is timely inconsequential. And better than producing communication overhead with complexity log(p).

2) What is the limitation of Hyperquicksort that the Parallel Sorting by Regular Sampling algorithm tries to address? How do these algorithms compare in terms of scalability?

The limitation is the load balancing. Hyperquicksort only sorts the array, whereas regular sampling, samples pivot values. THeir scalibility is the same.

3) (Explained above already)

4) Consider the Heat Distribution problem studied in class. What is the purpose of using ghostpoints? Explain the compromise that is being targeted.

Ghost points are memory locations used to store redundant copies of data held by neightboring processes.

It simplifies parallel algorithms by allowing the same loop to update all cells.

If only one row is transmitted, communication time may be dominated by message latency. By sending two, we can advance simulation two steps before another computation.

Extra Questions Not Done Yet From Random Exams

1) Discuss the differences between #pragma omp critical and #pragma omp atomic.

Atomic applies a simple operation to the value in memory like +,-,*, guaranting the reading and writing is atomic.
The critical directive implements mutual exclusion in terms of execution of the code region. Whereas atomic implements mutual exclusion on the access of data.

LOL. I tried.

Last Minute Overview Before Repeating CPD next year

$$S = \frac{1}{f + \frac{1-f}{p}}$$

Shared	Distributed
Data sharing is much easier	Data needs to migrate using messages
Tasks are threads (simpler, less startup/terminate overhead, finer-grain parallelism)	Tasks are processes
Uniform data-access time	Variable

Shared	Distributed
Easier to program	Harder to program
Contention in memory limits scalability	Effective way to increase memory bandwidth
Communication is implicit (harder to debug)	It's explicit
More complex hardware	Simpler

$$S <= \frac{\sigma + \phi}{\sigma + \phi/p + k}$$

Amdahl's law

$$f = \frac{\sigma}{\sigma + \phi}$$

f is the fraction of operations that must be done sequentially. Doesn't care about the problem size.

Gustafson-Barsis' Law

$$f = \frac{\sigma}{\sigma + \phi/p}$$

$$S <= f * (p - 1) + p$$

f is the fraction of sequential computation on the parallel prograam. Scaled speedup.

Both ignore k.

Karp-Flatt Metric
Experimentally Determined Serial fraction

$$e = \frac{\sigma + k}{\sigma + \phi} = \frac{1/S - 1/p}{1 - 1/p}$$

Isoefficiency
Total execution time

$$T(n, p) = \sigma(n) + \phi(n)/p + k(n, p)$$

Total overhead increasing with p. Time the rest of the processes are idle while one executes the sequential part. Plus communication overhead.

$$T_0(n, p) = (p - 1) * \sigma(n) + p * k(n, p)$$

$$T(n,1) >= C * T_0(n,p)$$

$$C = \frac{\varepsilon(n,p)}{1-\varepsilon(n,p)}$$

$$\varepsilon(n,p) = S/p$$

Scalibility Function

$$\frac{M(f(p))}{p}$$

Dictates how memoryper processor must grow to mantain efficiency.

Hyperquicksort
Bad load balancing, scaliblity depends on C.
O(log n * n/p))

PSRP
Better load balancing, equal scalibility
O(log n * n/p))

Odd-Even Sort
Perfect scalibility
O(log n * n/p + n))