

# Programming Paradigms

Mark Cheeseman
*Mark.Cheeseman@csiro.au*

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

# Overview

## Types of parallelism

- task parallelism
- domain decomposition

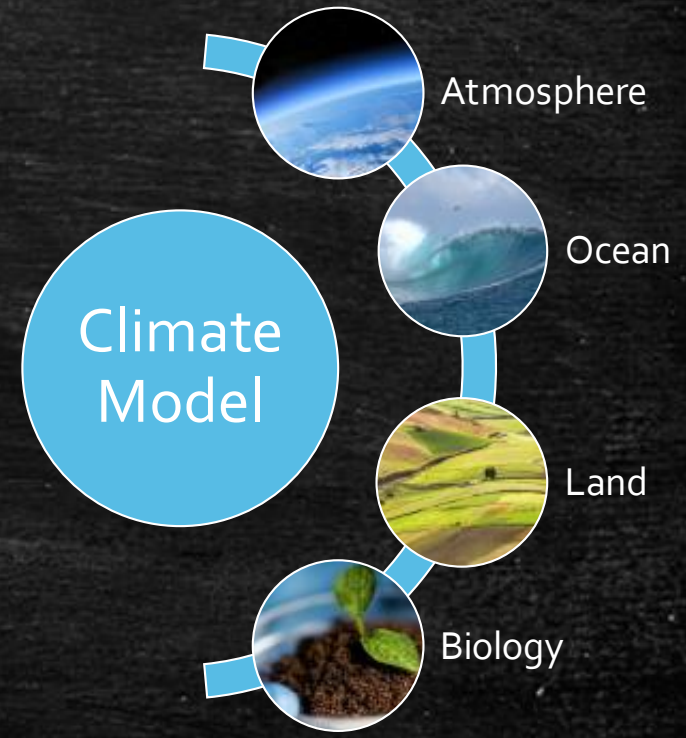
## Memory models

- shared vs distributed
- state
- *hybrid**

# Parallelism

# Task Parallelism

- Occurs where clearly distinguishable operations can be run independently.
  - Input/output of each process may depend on another process BUT there is no/little data dependence during the operation.
- Can be in a single application or a workflow
- Example would be a coupled climate model.
  - Independent binaries for ocean, atmosphere, ice, land and biological models are run with minimal interaction.



# I/O Server Design

Assume we have an application running concurrent tasks that each have data we wish to write to hard disk. We can

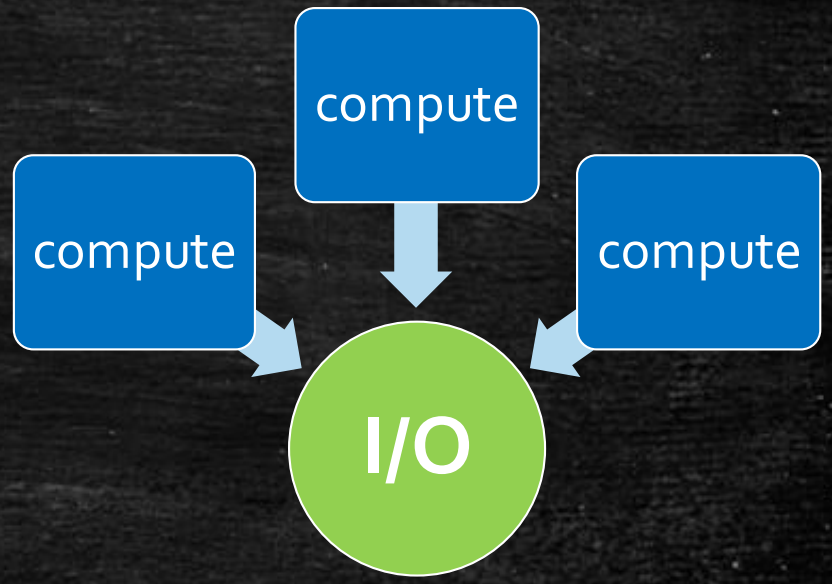
a) have each task write its own output file
   - messy, unmanageable as # of tasks grows

b) Assign another concurrent task to be responsible for the collection and writing of output data from all tasks ***(io server approach)***

c) Use a parallel-IO enabled infrastructure
   - Learn more on Day 3!

# I/O Server Design

- Allows a single output file to be written
- Implementation details are key!
  - Efficient transferring of data from compute to I/O
  - Will I/O cope with the workload?
  - How would we grow this design to deal with
    - More compute tasks?
    - Larger output data requests?
    - More frequent data requests?



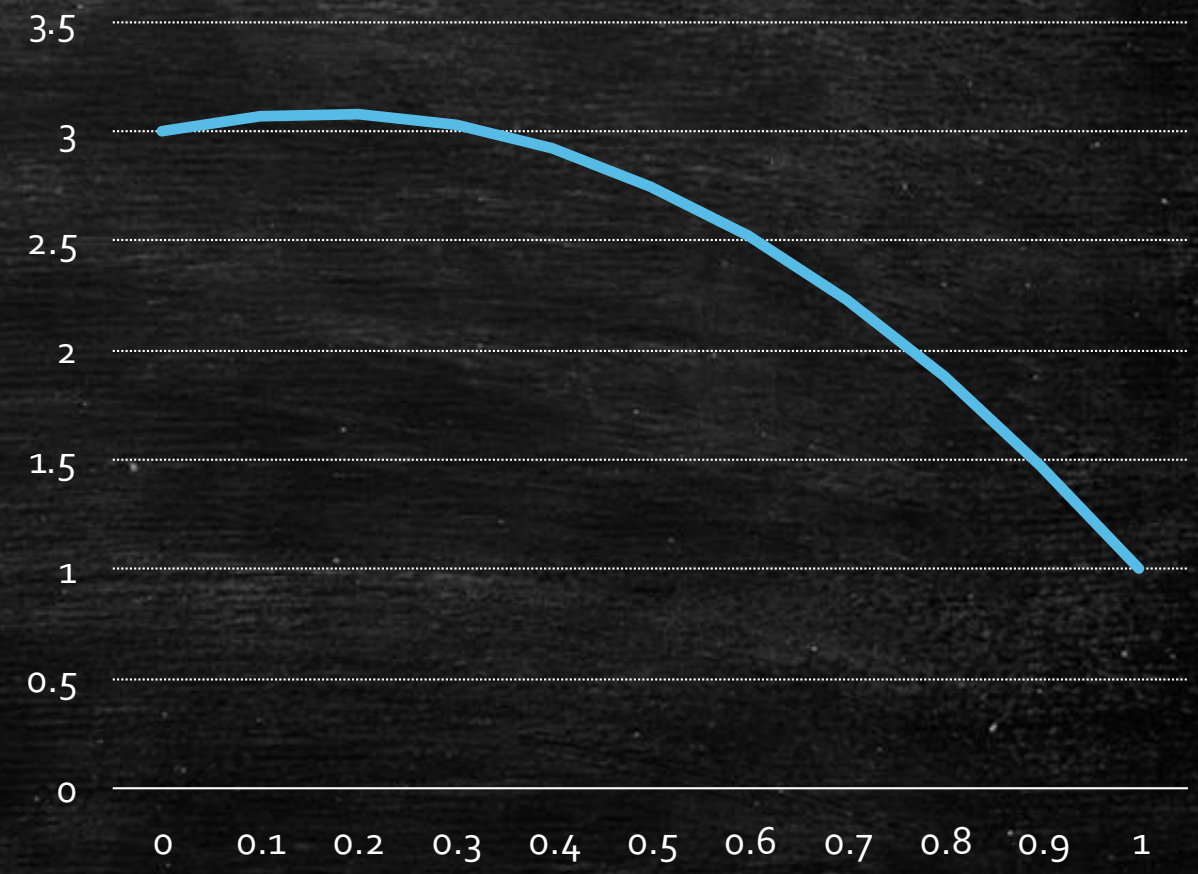
# Domain Decomposition

- Want to perform an operation on a large dataset.
- Identical instances of this operation can be performed independently
  - Data may need to be transferred between the instances
- Most commonly used form of parallelism in software
- Examples:
  - vector operations
  - Finite difference/element codes

# Numerical Integration

- Assume we have smooth continuous function over a finite interval [0, 1]
- Want to evaluate its integral over the interval.

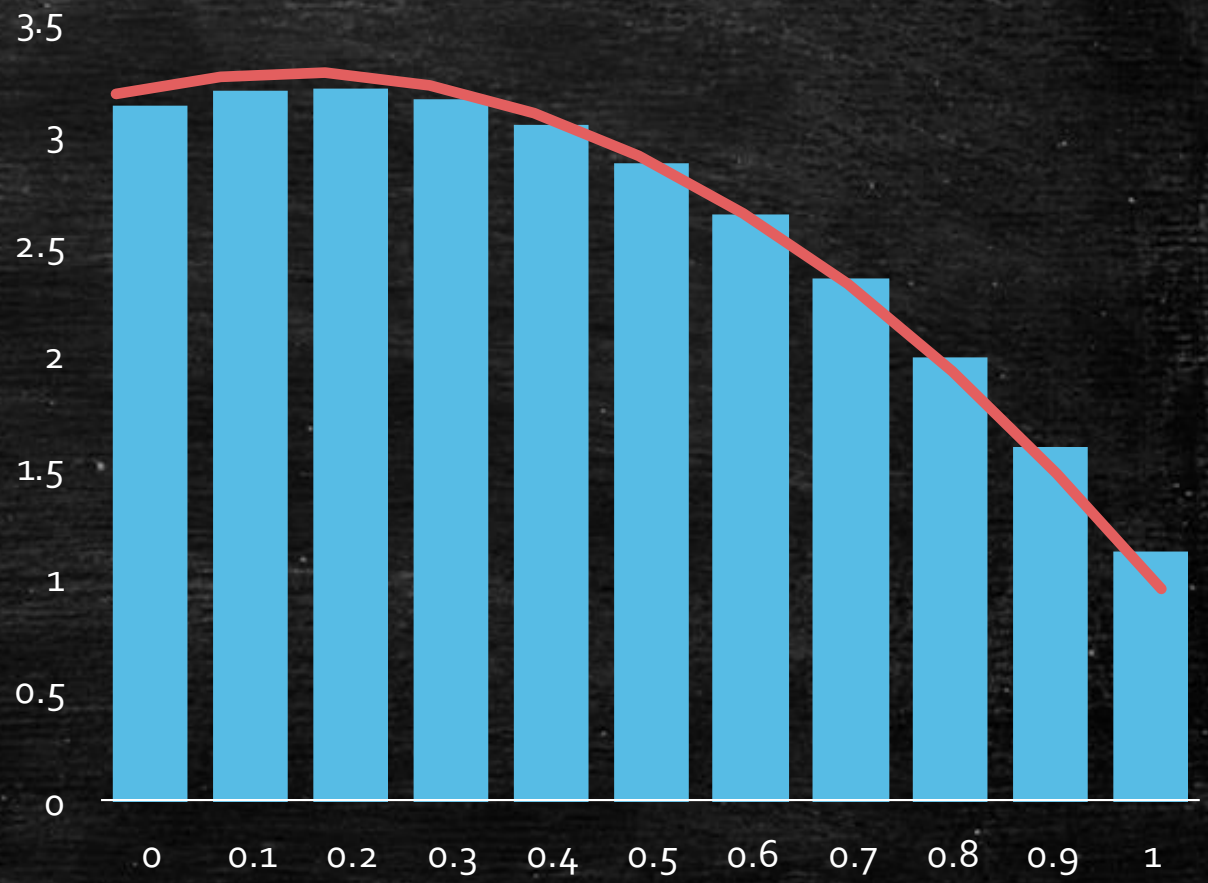
$$\int_0^1 f(x)\partial x$$



# Numerical Integration

- Integral can be approximated by the area under the curve.
- Can determine area by summing up a series of rectangles.

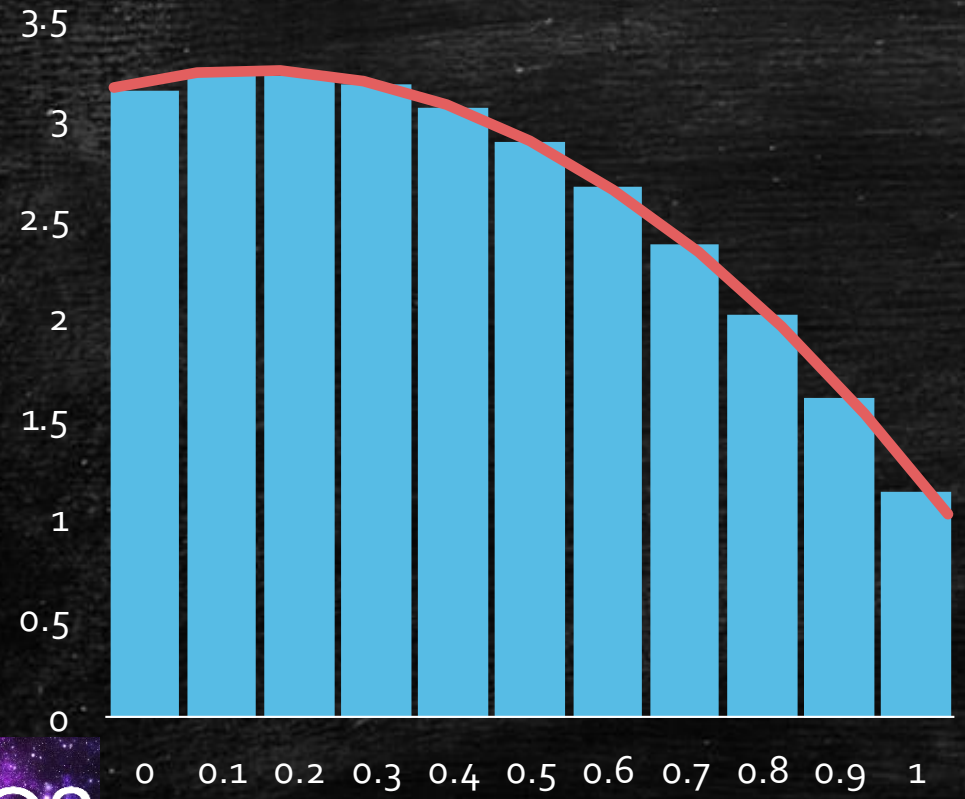
$$\int_0^1 f(x)\partial x \approx \sum_{i=1}^{N} a_i$$



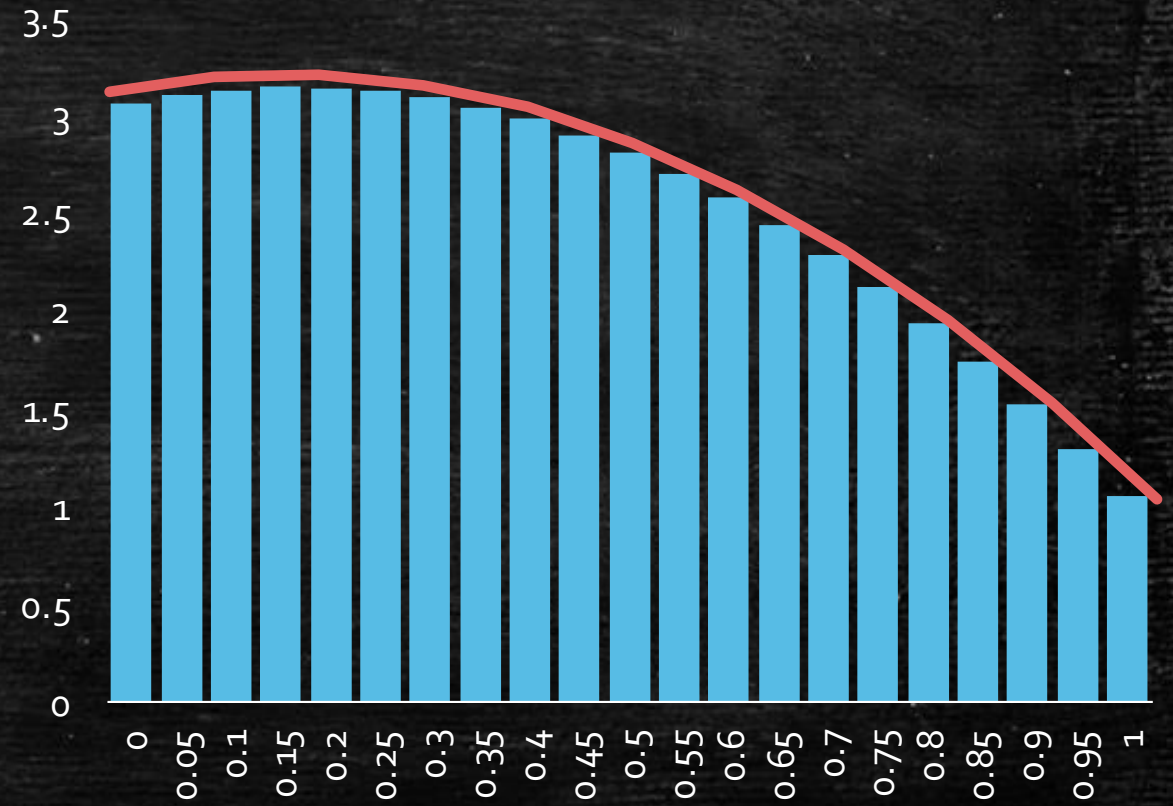

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

# Numerical Integration

- Solution accuracy increases by using more finer rectangles




- More rectangles leads to higher computational workload

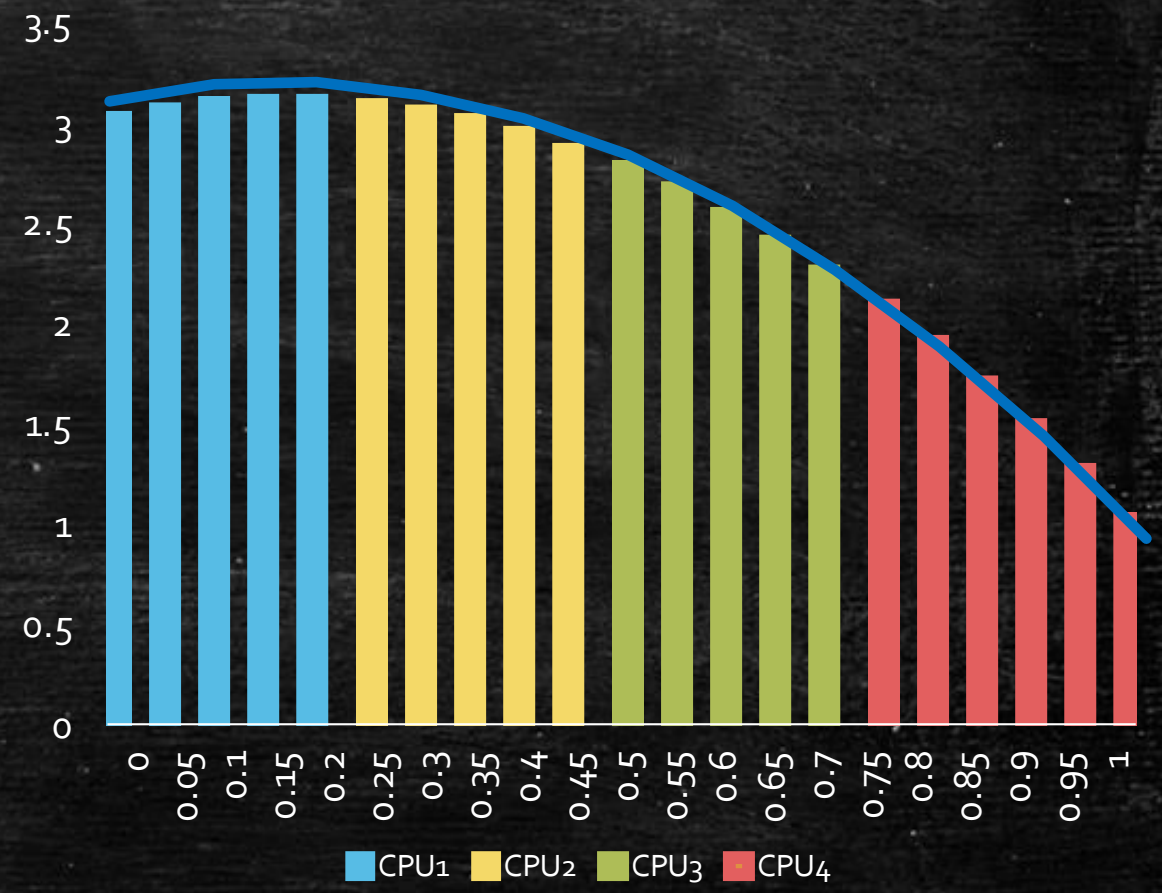



ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

# Numerical Integration

- Assign each process to compute area for a piece of the global domain
- Add up the individual areas to approximate the global integral

$$\int_0^1 f(x)\partial x \approx \mathrm{A1} + \mathrm{A2} + \mathrm{A3} + \mathrm{A4}$$




ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

# Numerical Integration

1. Define/read in f(x) and interval of interest $[x_1,x_2]$
2. Define sub-intervals for each concurrent task
3. Every task determines the sum of areas for all rectangles it owns
4. All sums are gathered onto a single (or all) tasks

# Numerical Integration

1. Define/read in f(x) and interval of interest $[x_1,x_2]$
2. Define sub-intervals for each concurrent task
3. Every task determines the sum of areas for all rectangles it owns
4. All sums are gathered onto a single (or all) tasks

Can be expensive and difficult

# Numerical Integration

1. Define/read in f(x) and interval of interest $[x_1, x_2]$
2. Define sub-intervals for each concurrent task
3. Every task determines the sum of areas for all rectangles it owns
4. All sums are gathered onto a single (or all) tasks

Load balancing:

- what if # of tasks doesn't evenly divide into interval?
- what if CPUs are not homogeneous?

# Why are task and domain parallelism important?

Many codes/workflows tend to naturally fit one. Factors affecting your choice include:

- Domain size
- Task independence / complexity
- Machine architecture (lots of cores tends towards data parallelism)
- Software used (programming language, libraries)
- Your objectives (ease of use, run time, memory usage, etc)

# Memory Models

# Memory Models

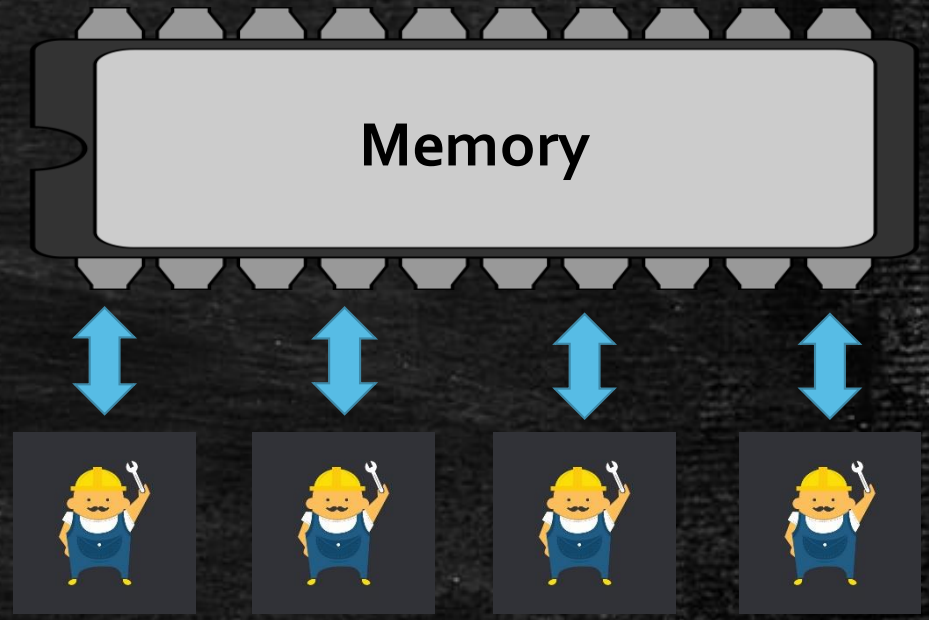
Shared

Distributed

*Hybrid*

# Shared memory model

- Description
  - multiple processes that have access to a single global memory space
  - *everyone sees everything*
- Implementations
  - OpenMP
  - Threaded programming (Java)
  - CUDA



# Shared memory model

## Cons

- state protection.
- problem size limited to size of memory available
- number of processes limited by platform
- performance issues may be harder to address
  - memory layout and/or hierarchies
  - architecture features

## Pros

- simple to program
  - Extra coding kept to a minimum
  - Native compiler/package support
- fast
  - Everything done on local platform (CPU, GPU, socket)
- generally available

# Memory and State

- All processes can see *and access* any variable in the shared memory

```
int tid, x = 1;
#pragma omp parallel {
    tid = omp_thread_num() + 1;
    x += tid;
    printf( "value of x is %f\n", x );
}
```

What is the printed value of x? (assuming 4 threads)

What would the value of x be outside of the parallel region?

# Memory and State

- Inside the parallel region:
  - Don't know the value of x at a particular time
  - Value of x depends on the thread scheduling (which is compiler and run specific)

- Outside the parallel region:
  - X = 11
  - Addition is an associative operation so the thread scheduling has no effect on the final outcome.
  - What could happen if we were adding a large series of very small and very big numbers?
    - *Remember that we are working with a finite number system!*

# Memory and State

- Need to explicitly monitor access to shared variables
  - Otherwise unintended consequences could occur!
- Be careful with non-associative operations
  - Order of operations would be important!

# Numerical Integration – Shared Memory

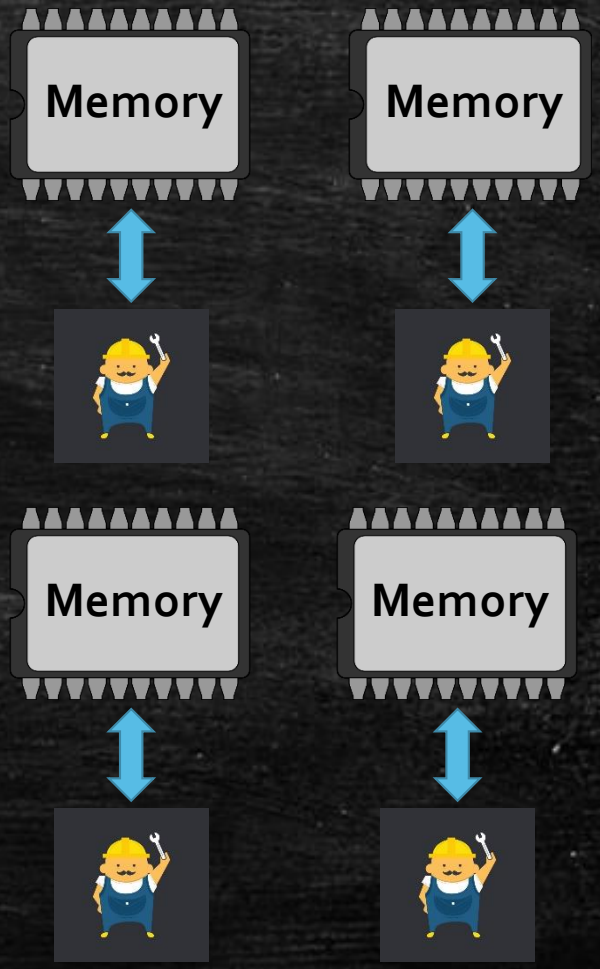
1. Define/read in f(x) and interval of interest $[x_1,x_2]$
2. Define sub-intervals for each concurrent task
   a. *Spawn 4 threads*
   b. *Each thread would determine its sub-interval using its thread ID and global interval limits*
3. Every task determines the sum of areas for all rectangles it owns
4. All sums are gathered onto a single (or all) tasks

# Numerical Integration – Shared Memory

1. Define/read in f(x) and interval of interest $[x_1,x_2]$
2. Define sub-intervals for each concurrent task
   a. *Spawn 4 threads*
   b. *Each thread would determine its sub-interval using its thread ID and global interval limits*
3. Every *thread* determines the sum of areas for all rectangles it owns
4. All sums are gathered onto a single (or all) tasks
   a. *Performed as a local addition. Order is irrelevant.*

# Distributed memory model

- Description
  - multiple processes located on physically separated platforms each with their own memory space. Global memory space is not physically contiguous.
  - *each process only sees <u>its</u> part of the problem*
- Implementation
  - Message Passing Interface (MPI)
  - Partitioned Global Address Space (PGAS) languages
    - Co-array Fortran, Charm++, Global Array Toolkit



# Memory and State

- Each individual compute node has its own copy of variables
- Be careful about local and expected global values

# Numerical Integration – Distributed Memory

1. Define/read in f(x) and interval of interest $[x_1,x_2]$
   a. *Start up the distributed environment with 4 active tasks*
   b. *f(X) and defined on every task – OR – defined on a root task and then distributed*
2. Define sub-intervals for each concurrent task
3. Every task determines the sum of areas for all rectangles it owns
4. All sums are gathered onto a single (or all) tasks

# Numerical Integration – Distributed Memory

1. Define/read in f(x) and interval of interest $[x_1,x_2]$
   a. *Start up the distributed environment with 4 active tasks*
   b. *f(X) and defined on every task – OR – defined on a root task and then distributed*
2. Define sub-intervals for each concurrent task
3. Every task determines the sum of areas for all rectangles it owns
4. All sums are gathered onto a single (or all) tasks
   a) *Need communication between ALL tasks*
   b) *Higher overhead than with the shared memory approach*

# Distributed memory model

## Cons

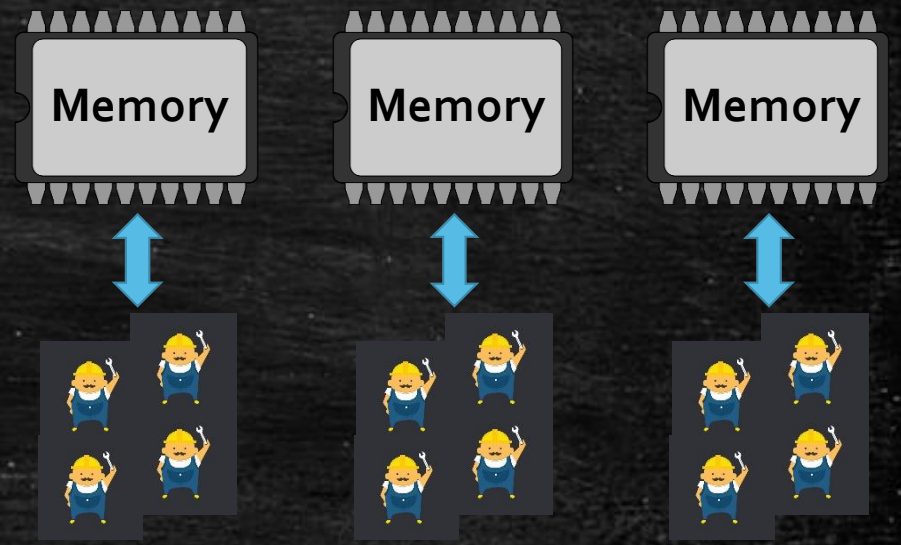
- more complex code
  - elaborate API to follow. Harder to visualize.
- slower passing of information between processes
  - Order of magnitude slower in shared memory case

## Pros

- State problem is easier
- acquire more resource when needed
  - more CPUs for smaller run times
  - more memory for larger domains
- Universal acceptance
  - MPI found on almost every platform
  - lots of libraries, tools and packages
- Other features
  - Parallel I/O
  - Multi-platform support (CPU + GPU)

# Hybrid - Multi-core

- As # of cores per CPU grows, combining task and data parallelism is required.
- Ex: adding OpenMP loop unrolling to MPI tasks
- Most commonly used to overcome bandwidth issues.
  - network saturation between CPUs
  - I/O



# Hybrid - GPU Computing

- GPUs and other accelerators (Xeon Phi) use a hybrid parallelism as well
  - Host and GPU device have separate memories
  - Both are capable of running multiple processes
- Modern GPGPUs are
  - Great for data parallelism (lots of threads)
  - Good for task parallelism as well (use of queues)
- Memory issues are paramount