# Load Balancing and Manager/Worker Parallelism

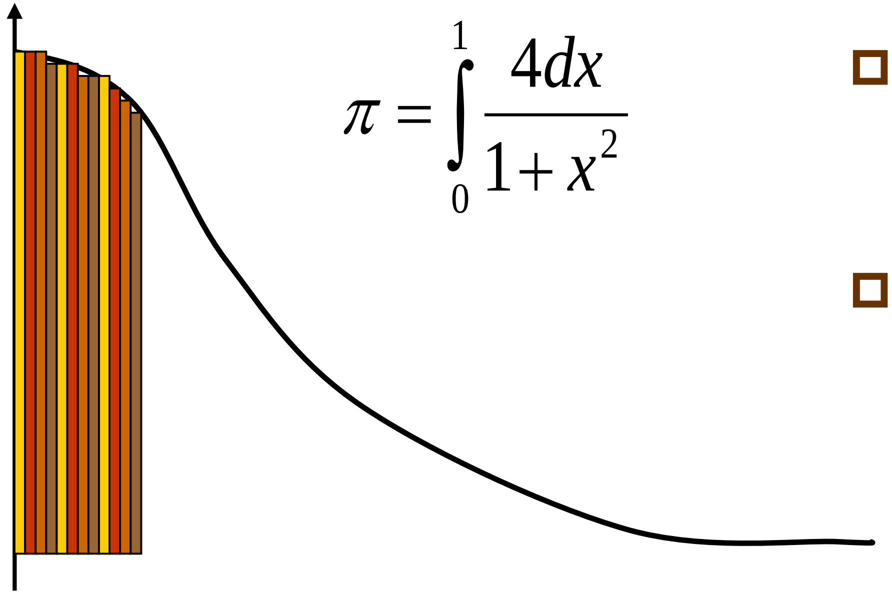
# Load Balancing

- The overall run time is determined by the slowest processor.

- To make most effective use of the parallel machine, we want to divide the ***computation***, the ***communication***, and the ***data*** evenly among the processors.

# Static Load Balancing: equal sized tasks

- In some applications you can determine ahead of time the work load and create a partition that shares work equally.



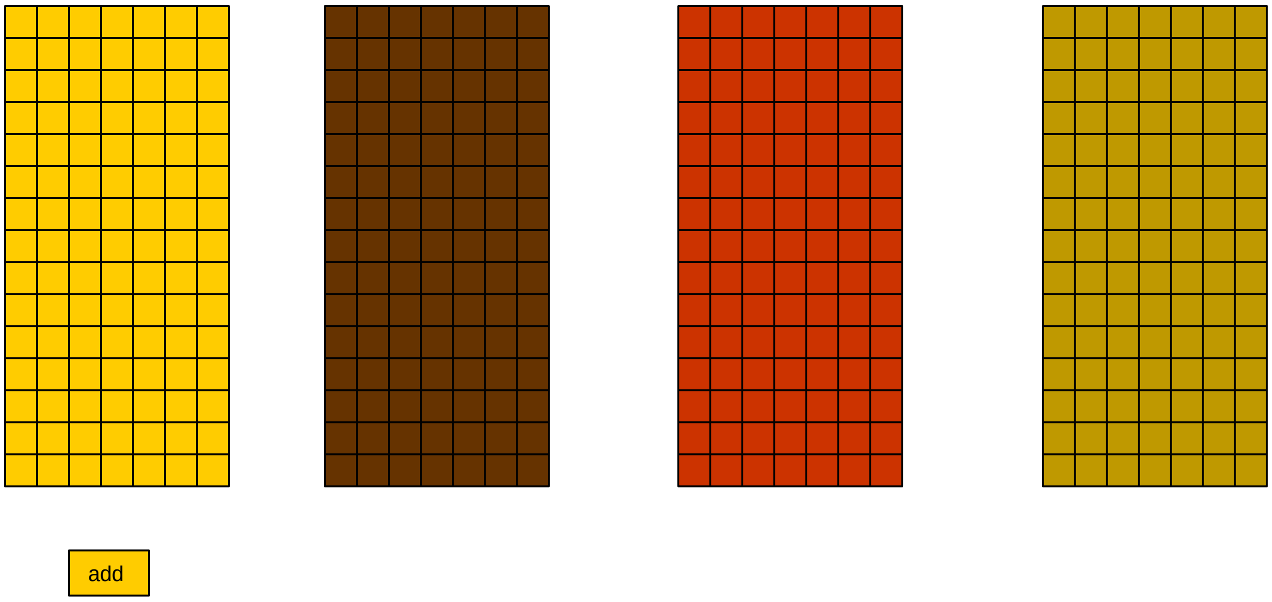
$$\pi = \int_0^1 \frac{4dx}{1+x^2}$$

- The time to compute each rectangle is identical.
- With lots of rectangles compared to the number of processes, the adding up partial sums takes a trivial amount of time.

Numerical Integration

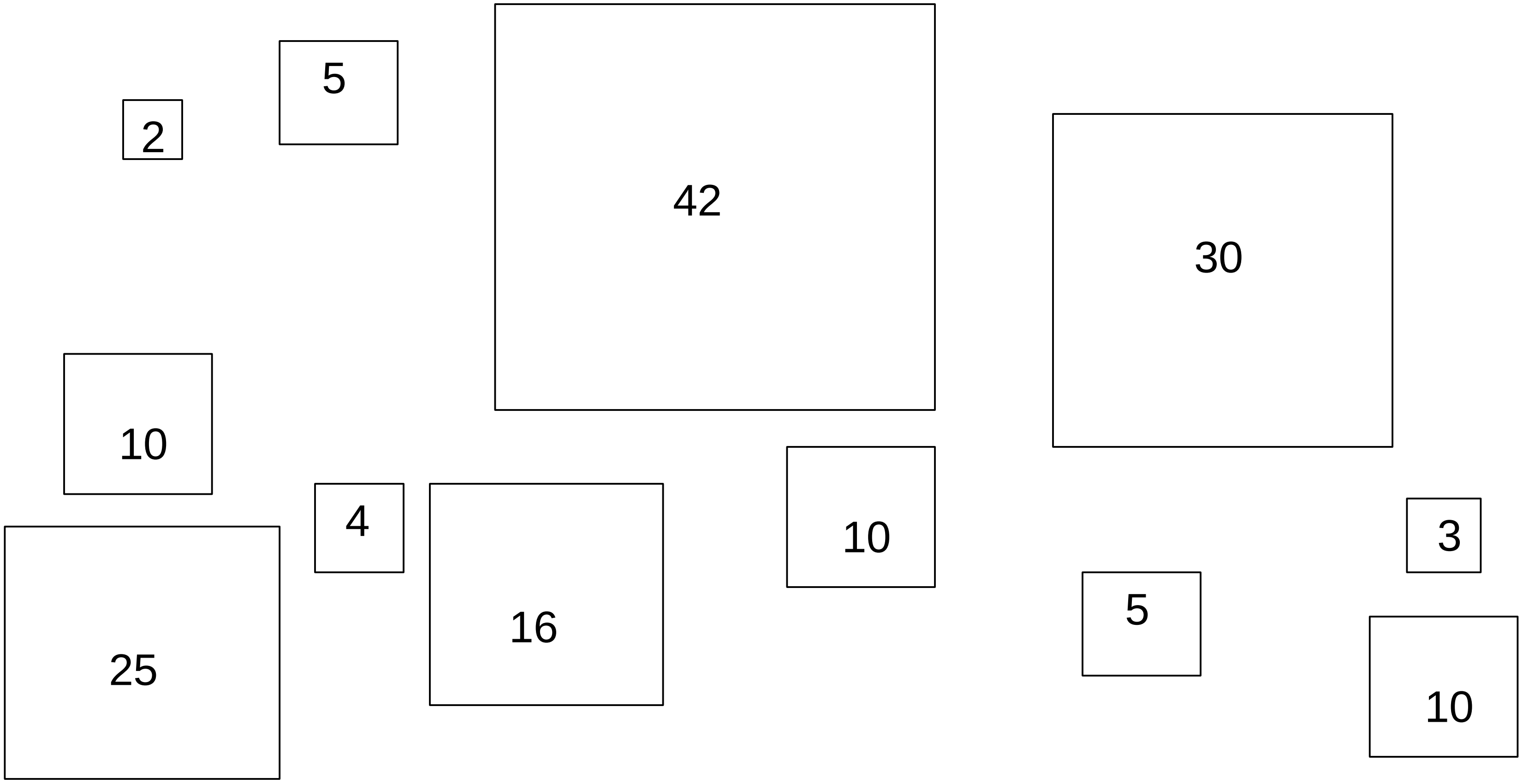
# Static Load Balancing: equal sized tasks

- Each process adds up an equal number (n/p) of rectangles.
- P0 adds up p partial sums.



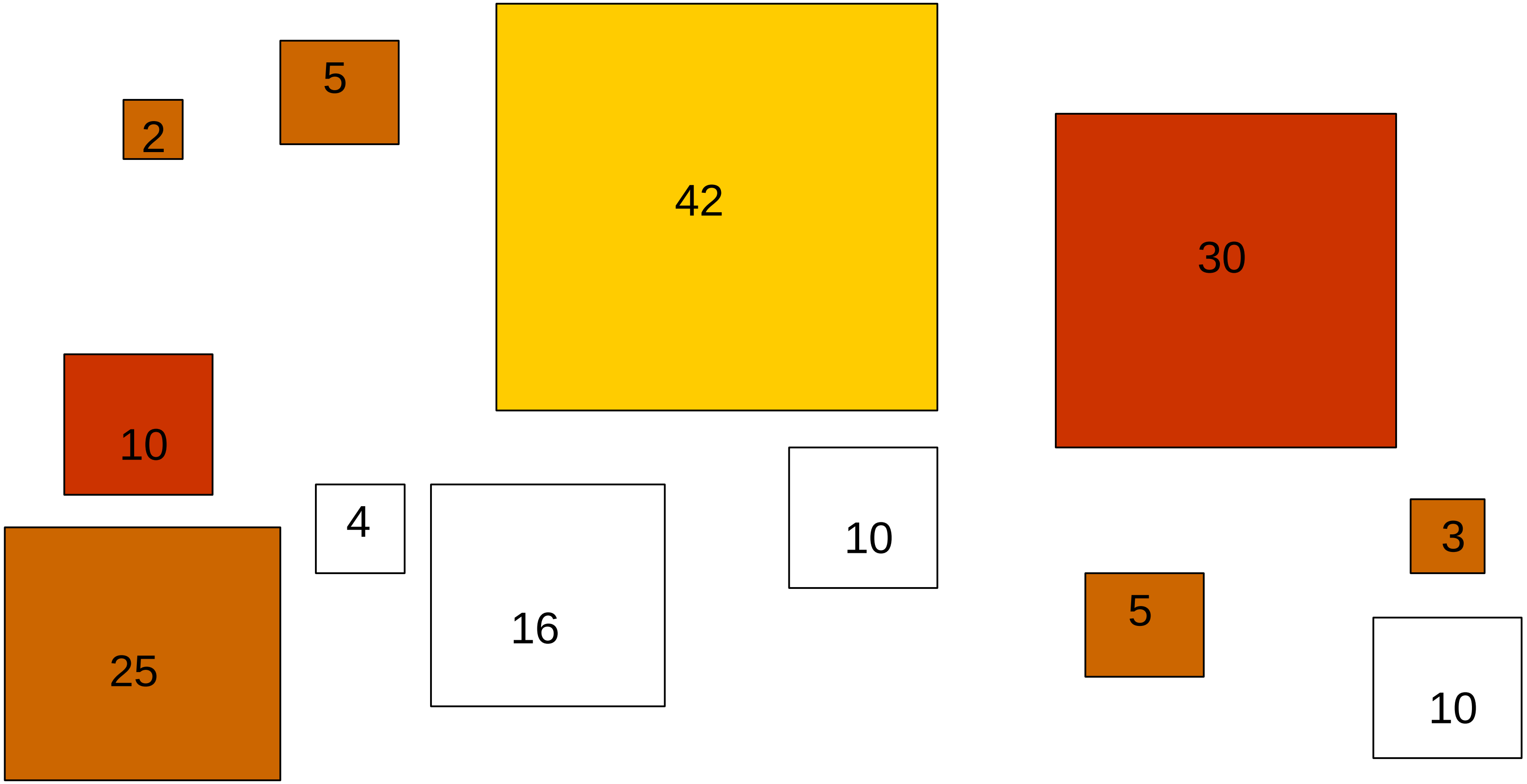
# Static Load Balancing: unequal sized tasks

- Task run time known (a priori).
- Partitioning is assigning tasks to processes.



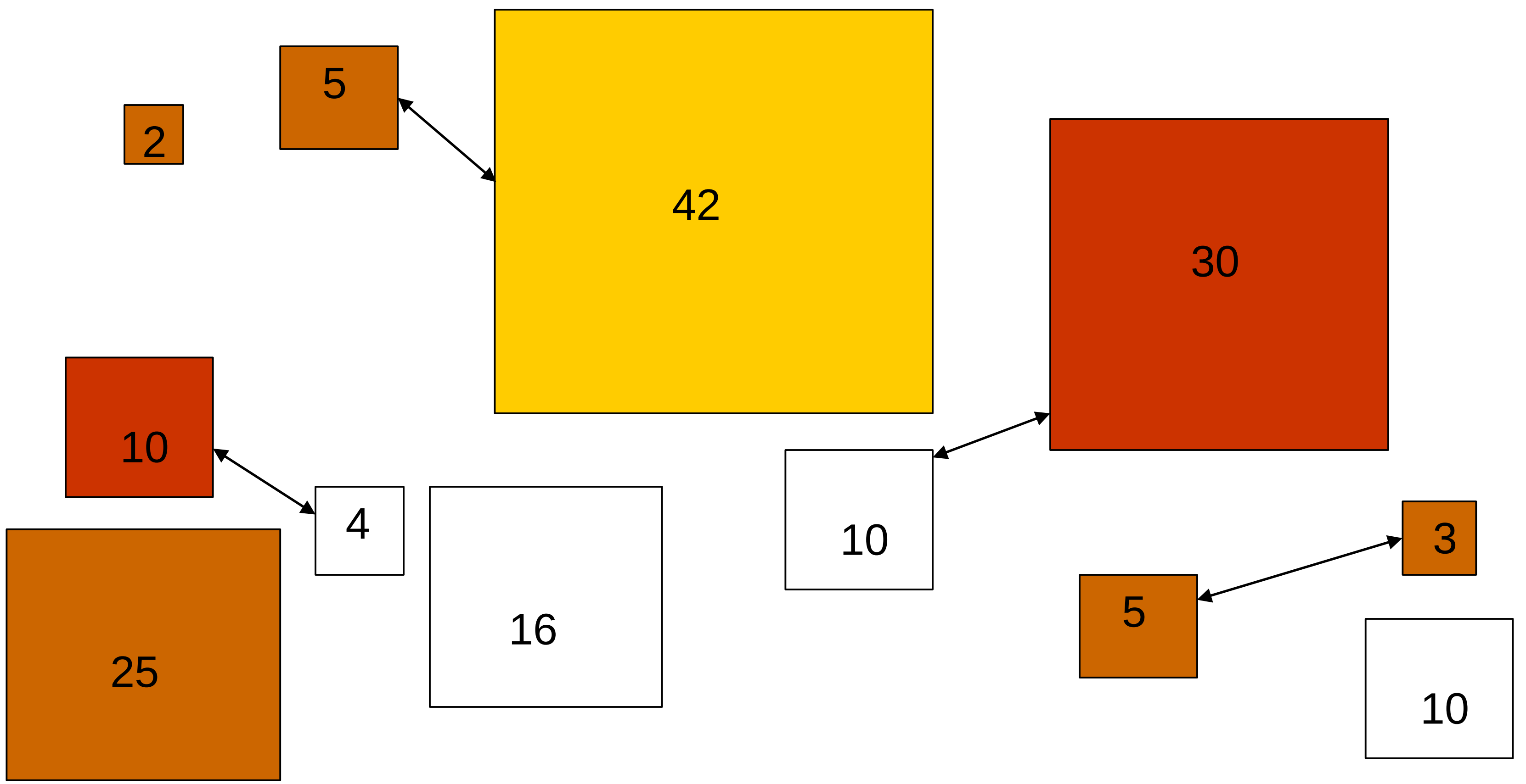
# Static Load Balancing: unequal sized tasks

- Task run time known (a priori).
- Partitioning is assigning tasks to processes.



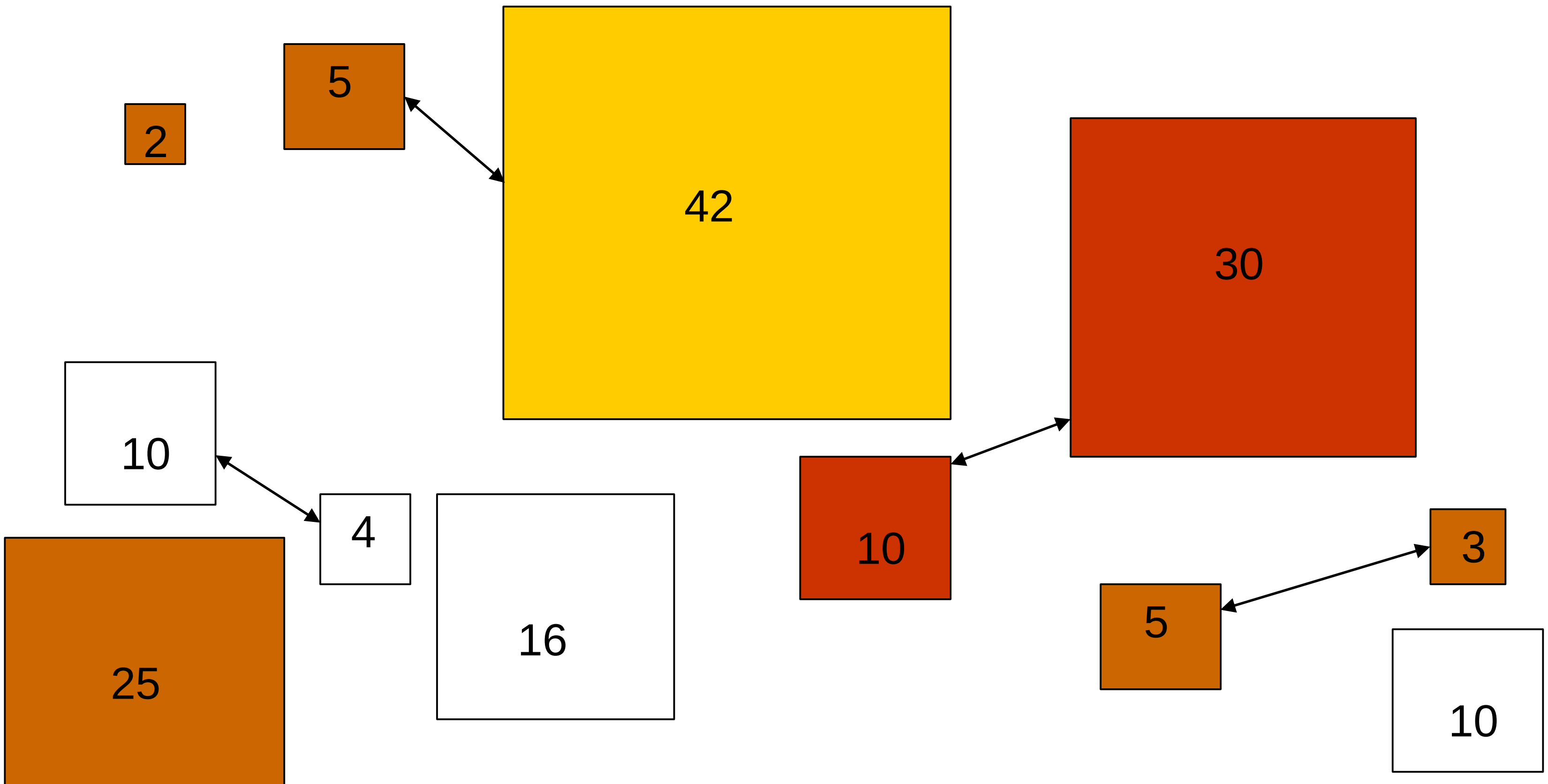
# Static Load Balancing: unequal sized tasks

- Caveat: Good partitioning takes communication into account as well



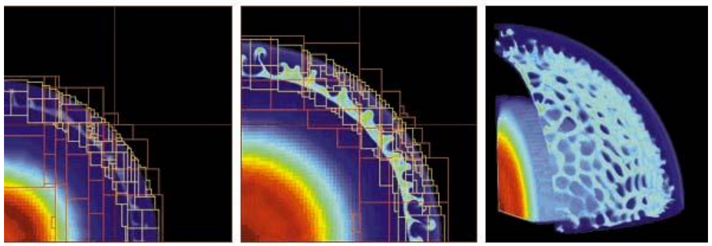
# Static Load Balancing: unequal sized tasks

- Caveat: Good partitioning takes communication into account as well.
- Fewer inter-process communications



# Dynamic Load Balancing

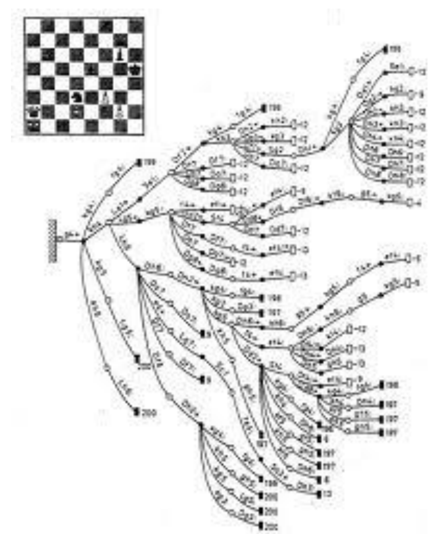
- In some applications you can ***not*** know ahead of time the work load and create a partition that shares work equally.



Lawrence Berkeley Lab:
Adaptive Mesh Refinement

```
if(found) break;
```

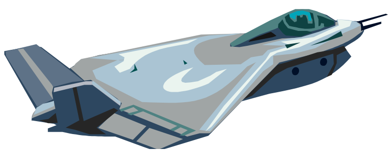
Searching

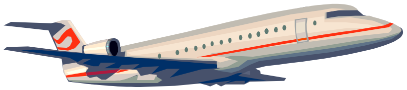


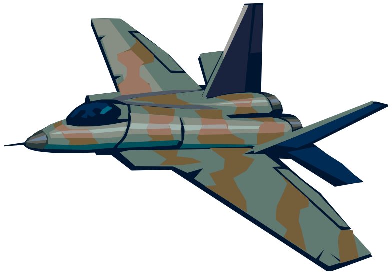
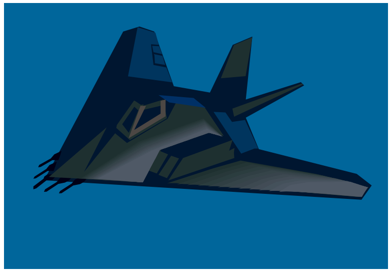
Game trees: Chess.com

# Suppose we want to find an aircraft design that maximizes fuel efficiency

- A design will be characterized by a parameter array design_parameters[].
- To compute the fuel efficiency, we run a computational fluid dynamics simulation of the design.
- Of all the designs, we want to find the one with the best fuel efficiency.





# Serial Code

```
#define NUM_PLANES 100
double* design_parameters[NUM_PLANES];

/* Get designs for each of the 100 planes */
for (d=0; d<NUM_PLANES; d++)
{
	GetDesignParameters(d, design_parameters[d]);
}

/* Compute efficiency of each design and keep track of best */
for (d=0; d<NUM_PLANES; d++)
{
	fuel_efficiency = CFD(design_parameters[d]);
	if (fuel_efficiency > best_efficiency)
	{
		best_design = d;
		best_efficiency = fuel_efficiency;
	}
}
```

# Parallelize the code with following assumptions.

- MPI style parallelism: Think of a few, say 4, processes each with their own memory.
- Assume almost all the work is in the CFD simulations.
- Each CFD simulation takes approximately the same amount of time. (equal sized tasks)

# Parallel Code 1

```
my_num_planes = NUM_PLANES/num_procs;
my_first_design = my_id * my_num_planes;

/* Get designs for each of my planes */
for (design=0; design<my_num_planes; design++)
{
        d= design + my_first_design;
        GetDesignParameters(d, design_parameters[design]);
}
```

# Parallel Code 1 (cont.)

```
/* Compute efficiency of each of my designs
   and keep track of best */

for (design=0; design<my_num_planes; design++)
{
      d= design + my_first_design;
      fuel_efficiency=
                   CFD(design_parameters[design]);
      if (fuel_efficiency > best_efficiency)
       {
            best_design = d;
            best_efficiency = fuel_efficiency;
      }
}
```

# Parallel Code 1 (cont.)

Tag = 0 for efficiency
Tag = 1 for design

```
/* Send my best efficiency to process 0 */
MPI_Send( &best_efficiency, 1, MPI_DOUBLE, 0, 0,
      MPI_COMM_WORLD);
MPI_Send( &best_design, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);

/* Process 0 gets the best designs from each process and
determines the overall best */
for (p=0; p<num_procs; p++)
{
  MPI_Recv( &efficiency, 1, MPI_DOUBLE,p, 0, MPI_COMM_WORLD,
      &status);
  MPI_Recv( &design, 1, MPI_INT,p, 1, MPI_COMM_WORLD,
      &status);
      if (efficiency > best_efficiency)
       {
            best_design = design;
            best_efficiency = efficiency;
      }
}
```

# Parallel Code 1 (cont.)

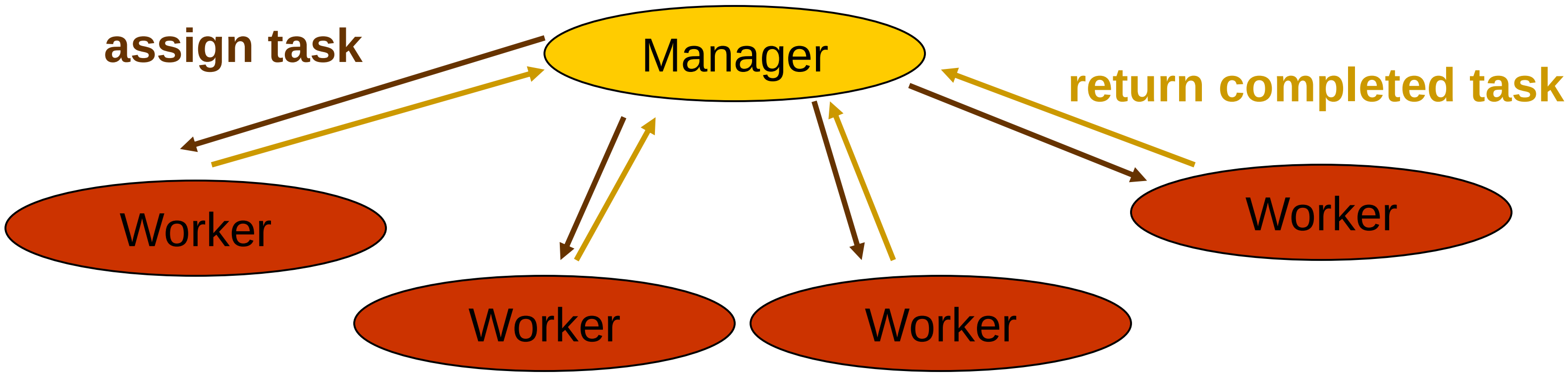
```
/* Send my best efficiency to process 0 */
MPI_Send( &best_efficiency, 1, MPI_DOUBLE, 0, 0,
      MPI_COMM_WORLD);
MPI_Send( &best_design, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);

/* Process 0 gets the bes
determines the overall be
for (p=0; p<num_procs; p+
{
  MPI_Recv( &efficiency, 1, MPI_DOUBLE,p, 0, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
  MPI_Recv( &design, 1, MPI_INT,p, 1, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
      if (efficiency > best_efficiency)
       {
            best_design = design;
            best_efficiency = efficiency;
      }
}
```

MPI_STATUS_INGORE
- May not need to access Status object
- Don’t have to declare object

# Parallelize the code with following ***new*** assumptions.

- MPI style parallelism: Think of a few, say 4, processes each with their own memory.
- Assume almost all the work is in the CFD simulations.
- ***The time to do a CFD simulation varies greatly from design to design, and is not predictable.***

# Manager/Worker paradigm for parallel computing




- Manager process assigns work
  - Keeps track of tasks to be done
  - Assigns tasks to workers
  - Receives information about completed tasks
- Workers processes do tasks
  - Receive assignments from manager
  - Sends information about completed tasks to manager

# Manager/Worker Parallel Code: A first start.

```
int main (int argc, char *argv[])
{

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
   MPI_Comm_size(MPI_COMM_WORLD, &p);

   if (my_id == 0)
   {
       manager(p);
   }
   else
   {
      worker():
   }
   MPI_Finalize();
   return 0;
}
```

# Manager/Worker Parallel Code: Code outline

```
Manager:

Get designs of all planes
Assign first p-1 designs

Repeat
{
   Recv efficiency
   compare to best
   if undone designs
        assign new design
   else
        fire worker
} until all workers fired
```

```
Worker:

Repeat
{
   Recv message from manager
   if assigned design
           do CFD
           send efficiency to manager
   else
           fired: exit loop
} forever
```

# Manager/Worker Parallel Code: Code issues

```
Manager:

Get designs of all planes
Assign first p-1 designs

Repeat
{
    Recv efficiency
    compare to best
    if undone designs
         assign new design
    else
         fire worker
} until all workers fired
```

The manager can't know which process will finish up first.  Has to recv from any process.

# Manager/Worker Parallel Code: Code issues

```
Manager:

Get designs of all planes
Assign first p-1 designs

Repeat
{
    Recv efficiency
    compare to best
    if undone designs
         assign new design
    else
         fire worker
} until all workers fired
```

The manager can't know which process will finish up first. Has to recv from any process.

```
MPI_Recv( &efficiency, 1, MPI_DOUBLE,
          MPI_ANY_SOURCE, 0,
          MPI_COMM_WORLD, &status);
src = status.MPI_SOURCE
```

# Manager/Worker Parallel Code: Code issues

Worker receives a message but may not know how long it is. It could contain many design_parameters or few. We'll also use messages of length zero to indicate the worker is fired.

```
Worker:

Repeat
{
    Recv message from manager
    if assigned design
            do CFD
            send efficiency to manager
    else
            fired: exit loop
} forever
```

# MPI_Probe: check for incoming messages

```
MPI_Probe(
        int             src,
        int             tag,
        MPI_Comm        comm,
        MPI_Status*     status)
```

- This call blocks until a message matching the source and tag parameters is available to be received.
- Can use wildcards MPI_ANY_SOURCE or MPI_ANY_TAG
- status object can be used to get information about the source, tag, or length of message.
- Does not actually receive the message.

# Worker Code

Source process 0,
Tag = 2 for assignment

```
for(;;)
{
   MPI_Probe(0, 2, MPI_COMM_WORLD, &status);
   MPI_Get_count(&status, MPI_DOUBLE, &length);

   parameters = (double *) malloc(length)
   MPI_Recv(parameters, length, MPI_DOUBLE, 0, 2,
         MPI_COMM_WORLD, &status);

   if (length == 0)   break;  /* FIRED! */

   efficiency = CFD(parameters);

   free(parameters);

   MPI_Send ( &efficiency, 1, MPI_DOUBLE, 0, 0,
       MPI_COMM_WORLD);
}
```

# Manager Code

```
/* Get designs of all planes */
for (d=0; d<NUM_PLANES; d++)
{
      GetDesignParameters(d, design_parameters[d]);
}

/* Assign first p-1 designs */
d=0
for (proc = 1; proc < p; proc++)
{
   current_design[proc] = d
   MPI_Send(design_parameters[d],size(design_parameters[d]),
         MPI_DOUBLE, proc, 2, MPI_COMM_WORLD);
   d++;
}
```

# Manager Code (cont.)

```
fired = 0;
do {
   /* Recv efficiency from worker */
   MPI_Recv( &efficiency, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0,
             MPI_COMM_WORLD, &status);
   proc = status.MPI_SOURCE
   if (efficiency > best_efficiency){
               best_design = current_design[proc];
               best_efficiency = efficiency;
   }
   /*Assign more work if available */
   if (d < NUM_PLANES){
    current_design(proc) = d
    MPI_Send(design_parameters[d],size(design_parameters[d]),
        MPI_DOUBLE, proc, 2, MPI_COMM_WORLD);
    d++;
   } else {
    MPI_Send(NULL, 0, MPI_DOUBLE, proc, 2, MPI_COMM_WORLD);
    fired++;
   }
} while(fired < (p-1))
```

# Key Concepts

- Manager/Worker paradigm is an effective way to think about parallel computations where it is difficult to preallocate work to processes and guarantee balanced workloads.

- MPI_Probe function is useful when you want a process to receive a message, but the receiving process can’t know ahead of time all details about the message.