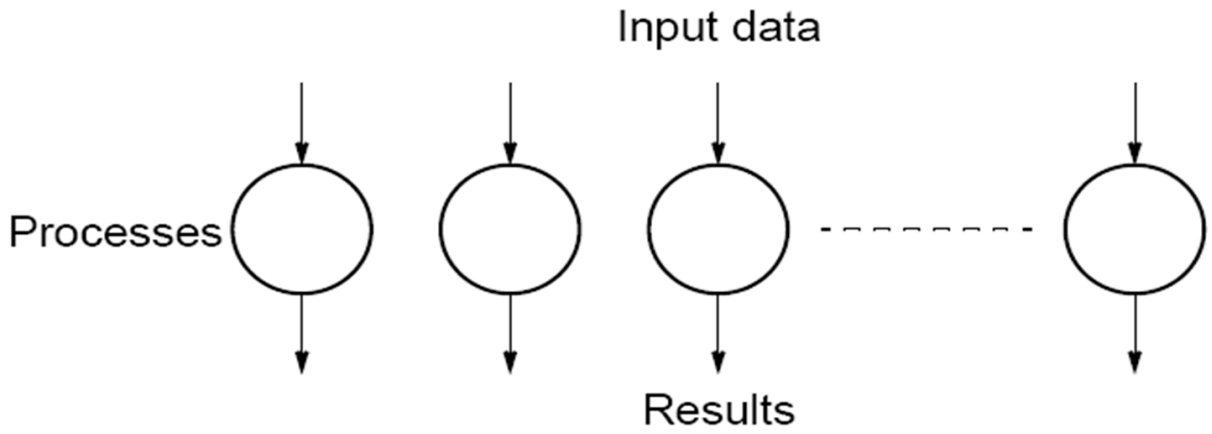
CPSC 4770/6770

Distributed and Cluster Computing

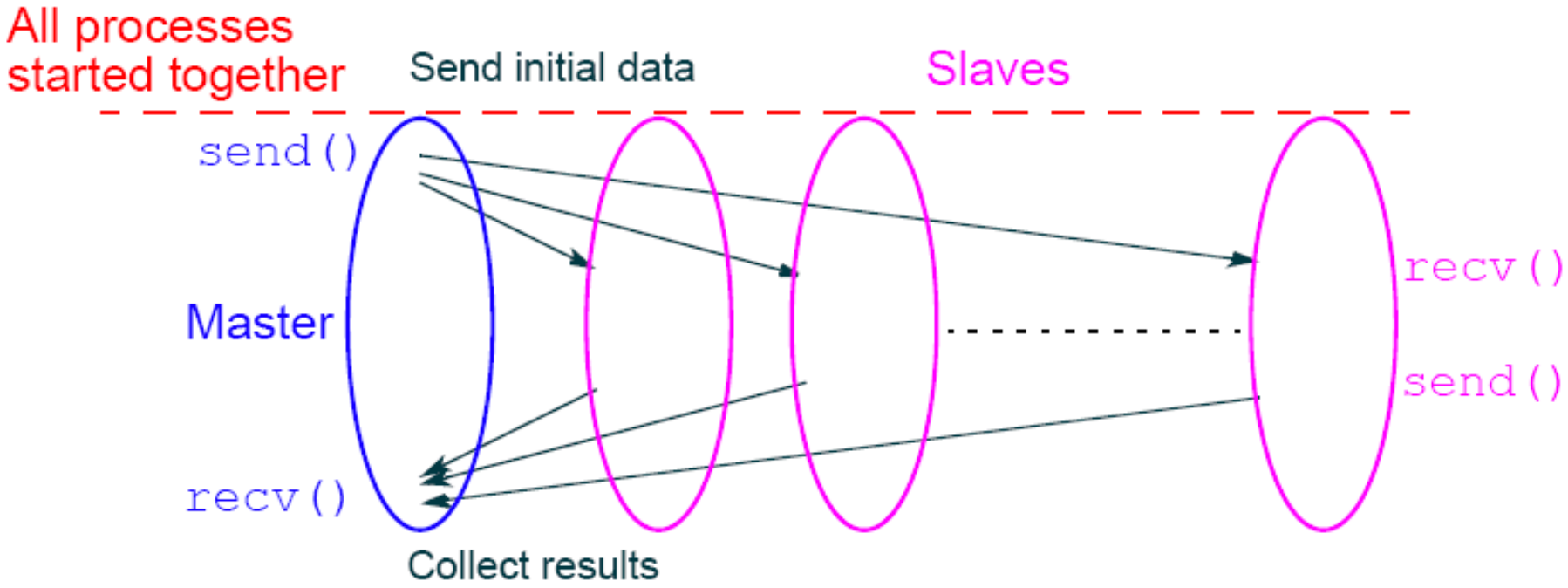
Lecture 5: Pleasant Parallelism

Pleasant Parallelism

- Embarrassingly parallel/naturally parallel/pleasantly parallel
- “A computation that can obviously be divided into a number of completely different parts, each of which can be executed by a separate process.”
- No communication or very little communication among the processes
- Each process can do its tasks without any interaction with the other processes



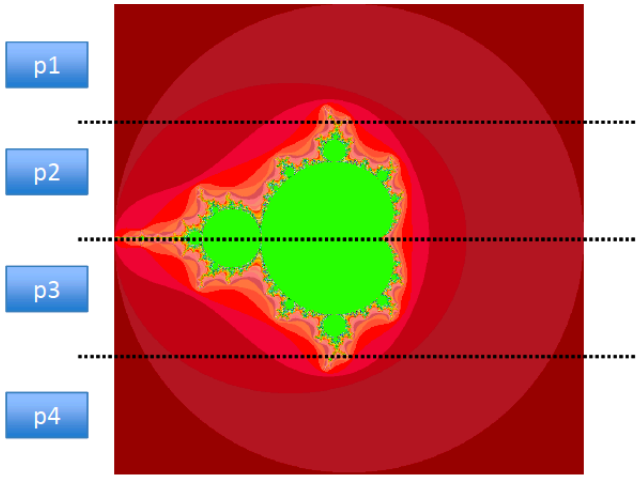
Pleasantly Parallel: Master-Worker



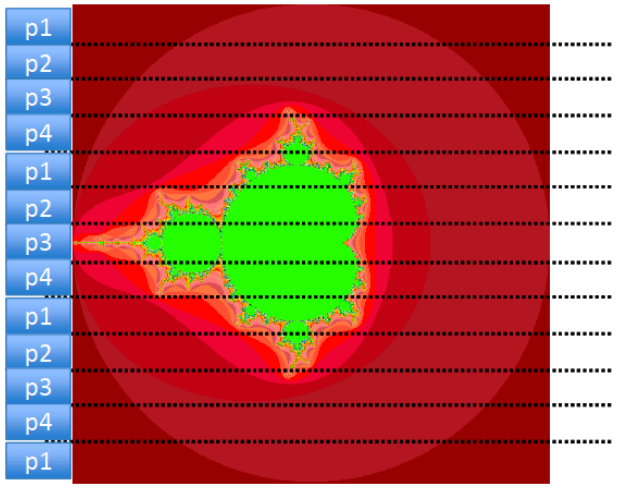
Usual MPI approach

Pleasantly Parallel: Work Assignment

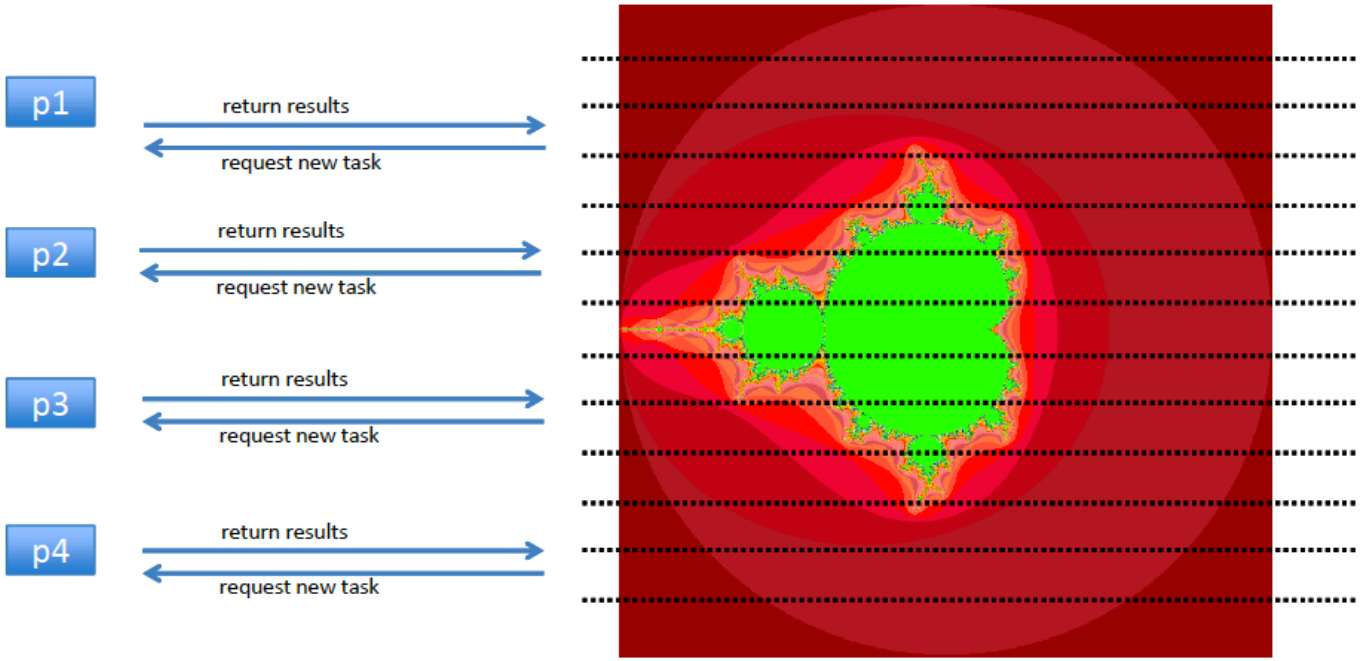
Static



Cyclic



Dynamic



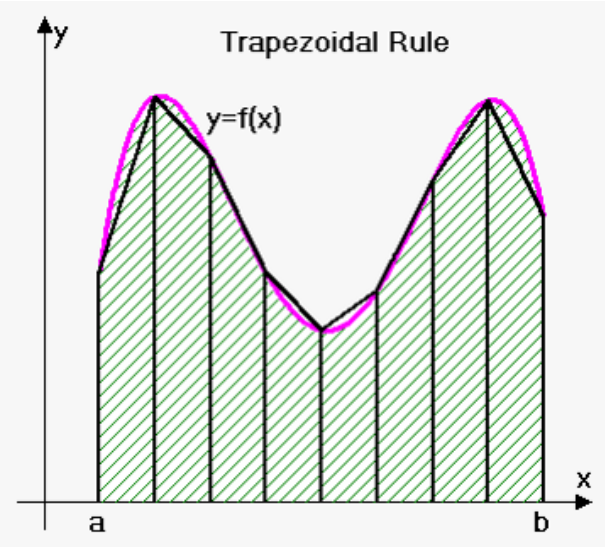
Example: Trapezoid Calculation

- Trapezoidal rule

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{2}\left(f(x_0) + 2f(x_1) + 2f(x_2) + 2f(x_3) + 2f(x_4) + \cdots + 2f(x_{n-1}) + f(x_n)\right),$$

where $\Delta x = \dfrac{b-a}{n}$ and $x_i = a + i\Delta x$.

$$\int_a^b f(x)\,dx \approx \sum_{k=1}^{N} \frac{f(x_{k-1}) + f(x_k)}{2}\Delta x_k\,.$$



- Which workload goes to which process?

```
if (rank == i) {
    do great things
}
```

- Start with small number of processes
- Calculate workload assignment manually for each count of processes
- Generalize assignment for process i based on sample calculations

```
# For example:
N = 8; a = 0; b = 2; h = (b - a)/N;

# With 4 processors (cores)
size = 4; rank = 1
local_N = N / size
local_a = a + rank * h * local_N
local_b = local_a + h * local_N
# What will local_a and local_b be?
```

Static Workload Assignment (static.c)

```c
/*
 * MPI implementation of the trapezoid approach to integral calculation following a static
 * workload distribution and standard send()/recv() calls.
 * We assume that the number of trapezoids is divisible by the number of MPI process.
 */

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

double Trap(double a, double b, int n);
double f(double x);

int main(int argc, char * argv[] ) {
  int rank;      /* rank of each MPI process */
  int size;      /* total number of MPI processes */
  double a, b;   /* default left and right endpoints of the interval */
  int n;         /* total number of trapezoids */
  double h;        /* height of the trapezoids */
  double local_a, local_b; /* left and right endpoints on each MPI process */
  int local_n;  /* number of trapezoids assigned to each individual MPI process */
  double result;       /* final integration result */
  double local_result; /* partial integration result at each process */
  int p;         /* counter */
  MPI_Status status;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  a = atof(argv[1]);
  b = atof(argv[2]);
  n = atoi(argv[3]);

  // calculate work interval for each process
  h = (b - a) / n;
  local_n = n / size;
  local_a = a + rank * local_n * h;
  local_b = local_a + local_n * h;
  local_result = Trap(local_a,local_b,local_n);

  // sending the results back to the master
  if (rank == 0){
    result = local_result;
    for (p = 1; p < size; p++){
      MPI_Recv(&local_result,1,MPI_DOUBLE,p,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
      result += local_result;
    }
  }
  else{
    MPI_Send(&local_result,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
  }

  // displaying output at the master node
  if (rank == 0){
    printf("Calculating the integral of f(x) from %lf to %lf\n", a, b);
    printf("The integral is %lf\n", result);
  }
  MPI_Finalize();
}

double Trap(double a, double b, int n) {
  double len, area;
  double x;
  int i;
  len = (b - a) / n;
  area = 0.5 * (f(a) + f(b));
  x = a + len;
  for (i=1; i<n; i++) {
    area = area + f(x);
    x = x + len;
  }
  area = area * len;
  return area;
}

double f(double x) {
  return ( x*x );
}
```

```
[jin6@node0329 06-pleasantly-parallel]$ mpirun -np 8 static 0 100 1000
Calculating the integral of f(x) from 0.000000 to 100.000000
The integral is 333333.500000
```

static_wtiming.c

- MPI_Wtime(): returns an elapsed time (in seconds) on the calling processor

```
36   // calculate work interval for each process
37   start = MPI_Wtime();
38   h = (b - a) / n;
39   local_n = n / size;
40   local_a = a + rank * local_n * h;
41   local_b = local_a + local_n * h;
42   local_result = Trap(local_a,local_b,local_n);
43   stop = MPI_Wtime();
44   tpar = stop - start;
45
46   //printf("process %d: (%lf, %lf)\n", rank, local_a, local_b);
47   printf("Process %d uses %lfs to calculate partial result %lf\n", rank, tpar, local_result);
48
49   // sending the results back to the master
50   start = MPI_Wtime();
51   if (rank == 0){
52     result = local_result;
53     for (p = 1; p < size; p++){
54       MPI_Recv(&local_result,1,MPI_DOUBLE,p,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
55       result += local_result;
56     }
57   }
58   else{
59     MPI_Send(&local_result,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
60   }
61   stop  = MPI_Wtime();
62   tcomm = stop - start;
```

```
[jin6@node0329 06-pleasantly-parallel]$ mpirun -np 8 static_wtiming 0 100 1000
Process 0 uses 0.000001s to calculate partial result 651.062500
Process 2 uses 0.000002s to calculate partial result 12369.812500
Process 3 uses 0.000002s to calculate partial result 24088.562500
Process 4 uses 0.000002s to calculate partial result 39713.562500
Process 5 uses 0.000001s to calculate partial result 59244.812500
Process 6 uses 0.000001s to calculate partial result 82682.312500
Process 7 uses 0.000002s to calculate partial result 110026.062500
Process 1 uses 0.000001s to calculate partial result 4557.312500
Calculating the integral of f(x) from 0.000000 to 100.000000
The integral is 333333.500000
Communication time: 0.00868s
```

Cyclic Workload Assignment (cyclic.c)

```
29   // calculate work interval for each process
30   start = MPI_Wtime();
31   h = (b-a)/n;     /* height is the same for all processes */
32   local_n = n/size;  /*  so is the number of trapezoids */
33
34   /* Each process' interval starts at: */
35   local_a = a + rank * h;
36   local_b = local_a + h;
37   local_result = 0;
38
39   for (i = 0; i < n/size; i++){
40     local_result = local_result + h * (f(local_a) +  f(local_b)) / 2;
41     //printf("Process %d (%d): (%lf, %lf)\n", rank, i, local_a, local_b);
42     local_a += h * size;
43     local_b = local_a + h;
44   }
45   stop = MPI_Wtime();
46   tpar = stop - start;
47
48   printf("Process %d uses %lfs to calculate partial result %lf\n", rank, tpar, local_result);
```

```
[jin6@node0329 06-pleasantly-parallel]$ mpirun -np 8 cyclic 0 100 1000
Process 1 uses 0.000003s to calculate partial result 41354.312473
Process 2 uses 0.000003s to calculate partial result 41478.812410
Process 3 uses 0.000003s to calculate partial result 41603.562379
Process 4 uses 0.000002s to calculate partial result 41728.562518
Process 5 uses 0.000003s to calculate partial result 41853.812428
Process 6 uses 0.000012s to calculate partial result 41979.312502
Process 7 uses 0.000003s to calculate partial result 42105.062441
Process 0 uses 0.000002s to calculate partial result 41230.062246
Calculating the integral of f(x) from 0.000000 to 100.000000
The integral is 333333.499397
Communication time: 0.00038s
```

Dynamic Workload Assignment (dynamic.c)

- The MPI_Status structure contains information including:
  - **The rank of the sender**. The rank of the sender is stored in the MPI_SOURCE element of the structure. That is, if we declare an MPI_Status stat variable, the rank can be accessed with stat.MPI_SOURCE
  - **The tag of the message**. The tag of the message can be accessed by the MPI_TAG element of the structure (similar to MPI_SOURCE)
  - **The length of the message**. The length of the message does not have a predefined element in the status structure. Instead, we have to find out the length of the message with MPI_Get_count

dynamic.c

- Initial workload assignment

```
33   /* initial job distribution is handled only by process 0 */
34   if (rank == 0){
35     a = atof(argv[1]);
36     b = atof(argv[2]);
37     n = atoi(argv[3]);
38     h = (b-a)/n;
39     count = 0;
40     /* send out the first round of work assignment, incrementing count as needed */
41     for (i = 1; i < size; i++){
42       param[0] = a + count * h;
43       param[1] = param[0] + h;
44       param[2] = h;
45       MPI_Send(param,3,MPI_DOUBLE,i,SEND,MPI_COMM_WORLD);
46       //printf("== Process %d: (%lf, %lf)\n", i, param[0], param[1]);
47       count = count + 1;
48     }
49   }
50   else {
51     MPI_Recv(param,3,MPI_DOUBLE,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
52   }
```

dynamic.c (Cont.)

- Worker process sends result back and receive new task

```
54    tpar = 0.0;
55    tcomm = 0.0;
56    partial_count = 0;
57    /* Each process that is not process 0 works on its portion, send the partial result back to 0,
58     * and wait for new workload unless the TAG of the message is 0
59     */
60    if (rank != 0){
61      do {
62        start = MPI_Wtime();
63        local_result = param[2] * (f(param[1]) +  f(param[0])) / 2;
64        partial_result += local_result;
65        stop = MPI_Wtime();
66        tpar += stop - start;
67        partial_count += 1;
68        start = MPI_Wtime();
69        MPI_Send(&local_result,1,MPI_DOUBLE,0,SEND,MPI_COMM_WORLD);
70        MPI_Recv(param,3,MPI_DOUBLE,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
71        stop = MPI_Wtime();
72        tcomm += stop - start;
73      } while(status.MPI_TAG != 0);
74      printf("Process %d uses %lfs to calculate partial result %lf of %d portions and %lfs for comm
    unication \n", rank, tpar, partial_result, partial_count, tcomm);
75    }
```

dynamic.c (Cont.)

- Master process receive results and assign new task to available worker

```
78   /* Process 0 receives results and sends out work while there is still work left to be sent
79    * (count < n) */
80   if (rank == 0) {
81     do {
82       MPI_Recv(&local_result,1,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
83       result = result + local_result;
84       param[0] = a + count * h;
85       param[1] = param[0] + h;
86       param[2] = h;
87       MPI_Send(param,3,MPI_DOUBLE,status.MPI_SOURCE,SEND,MPI_COMM_WORLD);
88       //printf("## Process %d: (%lf, %lf)\n", status.MPI_SOURCE, param[0], param[1]);
89       count = count + 1;
90     }
91     while (count < n);
92
93     /* Make sure that we receive everything */
94     for (i = 0; i < (size - 1); i++){
95       MPI_Recv(&local_result,1,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
96       result = result + local_result;
97     }
98   }
```

dynamic.c (Cont.)

- Print results when all the work has been done

```
100   /* All the work has been sent, */
101   if (rank == 0){
102     for (i = 1; i < size; i++){
103       MPI_Send(param,3,MPI_DOUBLE,i,STOP,MPI_COMM_WORLD);
104     }
105   }
106
107     /* Print the result */
108     if (rank == 0) {
109         printf("With n = %d trapezoids, our estimate\n",
110             n);
111         printf("of the integral from %f to %f = %f\n",
112             a, b, result);
113     }
```

```
[jin6@node0329 06-pleasantly-parallel]$ mpirun -np 8 dynamic 0 100 1000
Process 1 uses 0.000017s to calculate partial result 71146.951374 of 234 portions and 0.004391s for communication
Process 2 uses 0.000000s to calculate partial result 0.002500 of 1 portions and 0.002198s for communication
Process 3 uses 0.000007s to calculate partial result 48529.000494 of 72 portions and 0.003201s for communication
Process 4 uses 0.000024s to calculate partial result 70568.340897 of 227 portions and 0.004326s for communication
Process 5 uses 0.000000s to calculate partial result 0.020500 of 1 portions and 0.000023s for communication
Process 6 uses 0.000010s to calculate partial result 70763.003863 of 232 portions and 0.004360s for communication
Process 7 uses 0.000010s to calculate partial result 72326.179768 of 233 portions and 0.004351s for communication
With n = 1000 trapezoids, our estimate
of the integral from 0.000000 to 100.000000 = 333333.499397
```