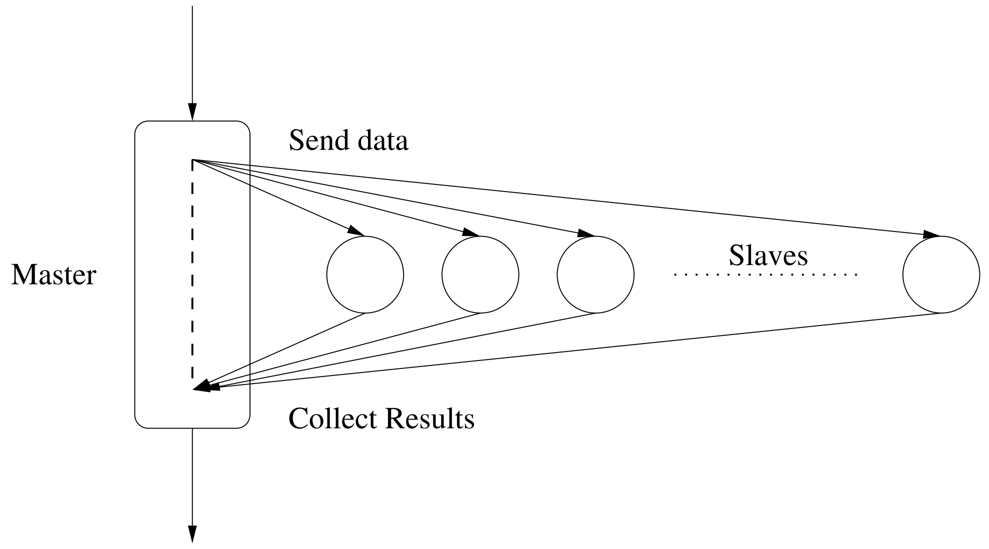
Outline: Embarrassingly (aka Naturally) Parallel Problems

- definition
- focus will be in two examples:
  - example #1: computation and visualization of Mandelbrot Set
    - definition of Mandelbrot set
    - sequential algorithm
    - static mapping parallelization
    - parallel cost analysis of static mapping parallelization
    - dynamic mapping parallelization (dynamic load balancing)
  - example #2: Monte Carlo methods (applied to numerical integration)
    - general definition of Monte Carlo methods
    - application to numerical integration
    - sequential algorithm for numerical integration
    - parallelization

Ref: Wilkinson and Allen Ch 3

Embarrassingly Parallel Problems (Definition)

- "ideal" computations from the parallelization view point
- they can be divided into completely independent parts for execution by separate processors (no data dependencies, completely disconnected computational graph)
  - paradigmatic example: Blocks of Independent Computations infrastructure
  - click here for science projects using such infrastructure
- distribution and collection of data are key issues (might be non-trivial and/or costly)
- frequently uses the master/slave approach



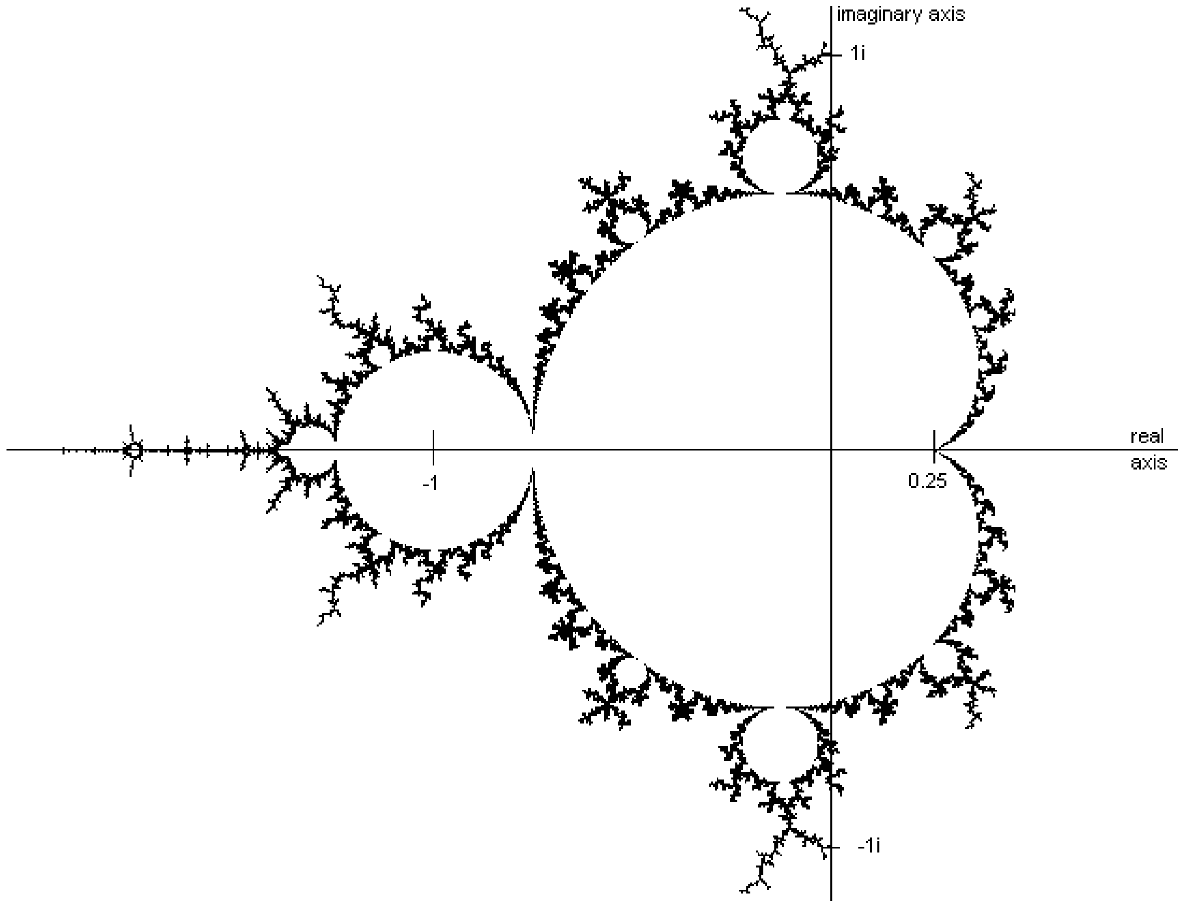
The Mandelbrot Set (Definition)

- a set of complex numbers (i.e., points in complex plane) that are “quasi-stable”
- a given complex number c (i.e., a position of a point in complex plane) is said to be quasi-stable if the series given given by the recurrence (with $z_0 = 0 + 0i$)

$$z_{k+1} = z_k^2 + c,$$

  remains bounded in absolute value no matter how large k becomes
- recall that absolute value of $z_{k+1} = a_{k+1} + b_{k+1}i$ is given by $|z_{k+1}| = \sqrt{a_{k+1}^2 + b_{k+1}^2}$
- it can be mathematically proven that the Mandelbrot set is enclosed by a circle centered at $(0, 0)$ of radius 2

The Mandelbrot Set (visually)



the Mandelbrot set is enclosed within the (boundary) black points scattered across the image

Computation of the Mandelbrot Set

- in practice, the [`real_min`, `real_max`] × [`imag_min`, `imag_max`] rectangle is split into a grid of `width` × `height` pixels (i.e, an image)
- typically, one uses the $[-2,2] \times [-2,2]$ rectangle (to visualize the full set), but not necessarily if, e.g., the focus is on a given region of the set
- if `0` ≤ `x` < `width` and `0` ≤ `y` < `height` denote the horizontal and vertical coordinates of a pixel, resp., then the corresponding point c in the complex plane is given by:

```
scale_width  = (real_max - real_min) / width;
scale_height = (imag_max - imag_min) / height;
c.real = real_min + ((float) x * scale_width);
c.imag = imag_min + ((float) y * scale_height);
```

- for each of pixel of the image, the recurrence

$$z_{k+1} = z_k^2 + c$$

  is executed until $|z_{k+1}| > 2$ (this indicates that the series will eventually divergence) or some arbitrary iteration limit is reached
- the output is the # of iterations required to fulfill such a condition

Computation for a Single Pixel

```
typedef struct complex{float real, imag;} complex;
const int MaxIter = 256;

int calc_pixel(complex c){
  int count = 0;
  complex z = {0.0, 0.0};
  float temp, abs_val_squared;
  do {
    temp = z.real * z.real - z.imag * z.imag + c.real
    z.imag = 2 * z.real * z.imag + c.imag;
    z.real = temp;
    abs_val_squared = z.real * z.real + z.imag * z.imag;
    count++;
  } while (abs_val_squared < 4.0  &&  count < MaxIter);
  return count;
}
```

- Note 1: if $z = a_z + b_z i$ and $c = a_c + b_c i$, then $z^2 + c = (a_z^2 - b_z^2 + a_c) + (2a_z b_z + b_c)i$
- Note 2: $|z_{k+1}| > 2$ if and only if $|z_{k+1}|^2 > 4$ (avoids square root computation)
- To-think: memory-bound or compute-bound computation?

Sequential Computation of the Full Mandelbrot Image

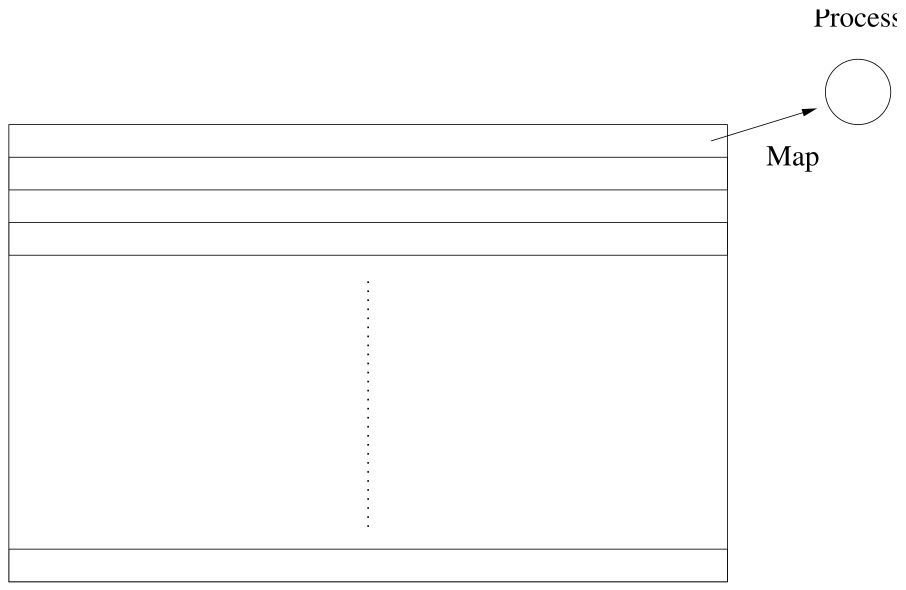
```
scale_width  = (real_max - real_min) / width;
scale_height = (imag_max - imag_min) / height;
for (x = 0; x < width; x++)
  for (y = 0; y < height; y++){
    c.real = real_min + ((float) x * scale_width);
    c.imag = imag_min + ((float) y * scale_height);
    color = calc_pixel(c);
    display(x, y, color);
}
```

From a parallelization view point:

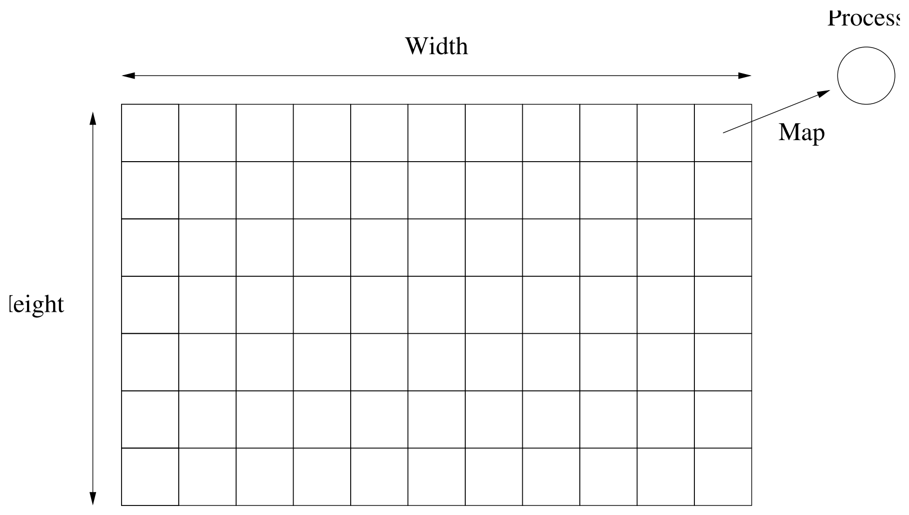
- `width` × `height` totally independent tasks (naturally parallel computation)
- computation of each pixel much less amenable to parallelization, though
- each task can be of different length (i.e., varying execution time)
- this property turns load balance among processors a challenge to be addressed

Static Mapping Parallelization

- static mapping if and only if correspondence among pixels and slave processes is known *a priori* (i.e., before the actual execution of the parallel algorithm)
- in order to have sufficient load per process, split image into regions, and a mapping among whole regions and processes is defined
- for data arranged in two dimensions (like images) one may either split across one dimension (e.g., by rows or by columns) or both dimensions (by blocks)



Row region (block of rows) for each process



Square block region for each process

One Possible Implementation of Static Mapping Parallelization

Master process:

```
row = 0;
block_num_rows=height/(nproc-1)
for (slave = 1;
      slave < nproc;
      slave++) {
 send(&row, 1, slave);
 row += block_num_rows;
}
for (npixel = 0;
      npixel < (width * height);
      npixel++) {
 recv({&x,&y,&color}, 3, any_proc);
 display(x, y, color);
}
```

Slave process:

```
const int master = 0;
recv(&firstrow, 1, master);
lastrow = firstrow + height/(nproc-1);
for (x = 0; x < width; x++) {
 for (y = firstrow; y < lastrow; y++) {
  c.real = real_min +
           ((float) x * scale_width);
  c.imag = imag_min +
           ((float) y * scale_height);
  color = calc_pixel(c);
  send({&x,&y,&color}, 3, master);
  }
}
```

- partition of image into blocks of consecutive rows (one block per slave)
- master process sends to each slave the identifier of the first row in the slave's block
- slave return results on a pixel-wise basis (one message exchange per pixel)
- master process visualizes points “on the fly” as soon as results are available

Discussion (potential improvements)

In the previous implementation ...

- do we actually need the initial data exchange among master and slaves?
- is it reasonable (from the parallel performance point of view) to send results to the master process on a pixel-wise basis?
- would it be more appropriate instead to send results into groups (e.g., one row at a time or even the full block of rows in a single message) to reduce the number of point-to-point messages (i.e., communication start-ups)?
- is it possible to leverage collective communication to communicate results (*hint*: what about gather?) instead of individual point-to-point messages?
- is there any guarantee that the workload will be perfectly balanced among processors?

Parallel Cost Analysis of Static Mapping Parallelization

Let p, m, n, I denote **`nproc, height, width, MaxIter`**, resp., and t_f the time/flop:

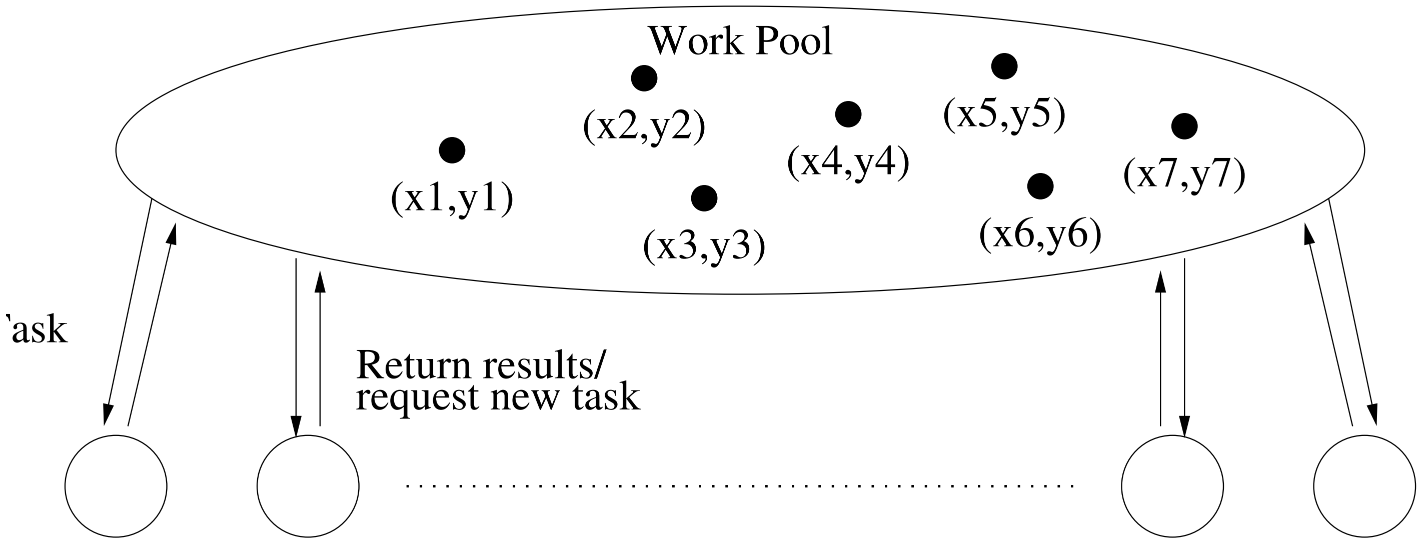
- sequential algorithm time:
  $t_{\text{seq}} \leq Imnt_f$
- parallel communication 1:
  $t_{\text{comm1}} = (p-1)(t_s + t_h + t_w)$
- parallel computation:
  $t_{\text{comp}} \leq \frac{Imn}{p-1} t_f$
- parallel communication 2:
  $t_{\text{comm2}} = t_s + t_h + \frac{m}{p-1} n t_w$
- parallel algorithm time:
  $t_{\text{par}} \leq t_{\text{comm1}} + t_{\text{comp}} + t_{\text{comm2}}$

Assumptions:

- initial data exchange among master and slaves present in the algorithm
- image split into blocks of rows, one block per slave
- results sent back into blocks of rows (full block of rows in each message)
- there is no communication contention in the node where the master process is executed

Dynamic Mapping Parallelization (aka Dynamic Load Balancing)

- the mapping among parallel tasks (i.e., pixels) and processes is unknown *a priori* but determined during the actual execution of the program (i.e.. dynamically)
- the goal is to dynamically load the balance among processors; to this end, the problem needs to be over-decomposed (i.e., more parallel tasks than processes)
- especially suited for applications with varying (and/or unknown) amount of work per task, and/or parallel computer with processors operating at different speed
- can be realized using a work-pool approach (aka processor farm); the slaves are supplied with work on demand as they become idle



Mandelbrot Set with Work-Pool Approach

Code leverages row-wise partition, with rows dynamically mapped to processes

Master

```
remaining = 0; row_to_compute = 0;
for (slave = 1; slave < nproc; slave++){
 send(&row_to_compute, 1,
      slave, compute_tag);
 remaining++; row_to_compute++;
}
do {
 recv({&slave,
       &row_result, rcolor},
       width+2, any_proc,
       result_tag);
 remaining--;
 if (row_to_compute<height) {
  send(&row_to_compute, 1,
       slave,
       compute_tag);
  remaining++; row_to_compute++;
 }
 else
  send(&row_to_compute, 1,
       slave,
       termination_tag);
 display_row(row_result, rcolor);
} while (remaining > 0);
```

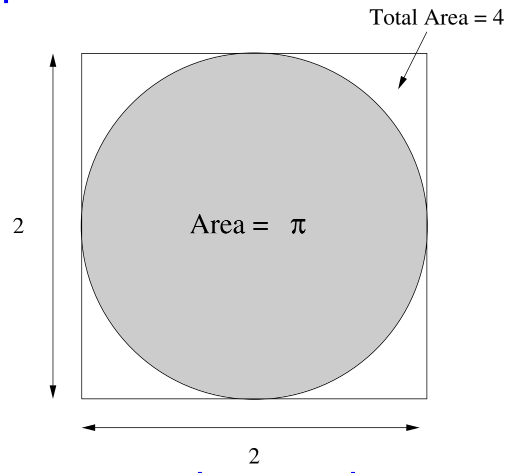
Slave (me is the slave process id)

```
recv(&y, 1, master, any_tag, &source_tag);
while (source_tag == compute_tag) {
 c.imag = imag_min + ...
 for (x = 0; x < width; x++) {
   c.real = real_min + ...
   rcolor[x] = calc_pixel(c);
 }
 send({&me, &y, rcolor}, width+2,
      master, result_tag);
 recv(&y, 1, master, &source_tag);
}
```

Monte Carlo Methods

- Monte Carlo Methods refer to a broad range of techniques that use randomly generated numbers to solve numerical and physical problems
- example: (inefficient) calculation of π
  - unit radius circle centered at origin within the $[-1,1] \times [-1,1]$ square
  - probability of random point in the square to be also within the circle given by:

$$\frac{\text{area of circle}}{\text{area of square}} = \frac{\pi(1)^2}{4} = \frac{\pi}{4}$$



  - throw N random points within the square and count how many within the circle
  - if N “large enough”, fraction within circle will approximate $\frac{\pi}{4}$
- another example: numerical integration (next slides)

Monte Carlo Numerical Integration

- the definite integral of a function $f(x)$ in the interval $[x_1, x_2]$ can we computed as:

$$\int_{x_1}^{x_2} f(x)\,dx = \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} f(x_i)(x_2 - x_1)$$

  where $x_1 \le x_i \le x_2$ is a randomly selected point within such interval

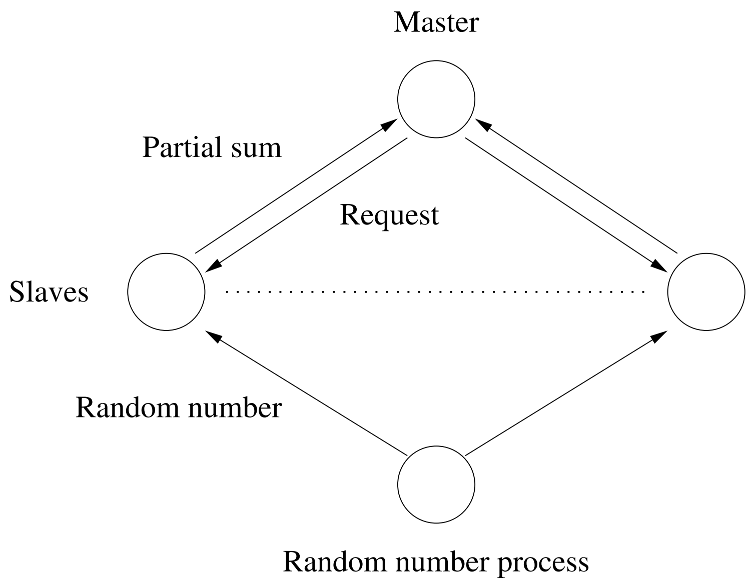
- in practice, we cannot compute infinite terms but a *"sufficiently large"* N (i.e., number of samples); the actual N depends on required accuracy and the function at hand

- example: computation of $\int_{x_1}^{x_2}(x^2 - 3x)\,dx$
  - `rand_v(x1, x2)` computes a pseudo-random number between `x1` and `x2`
  - code looks like:

```
sum = 0.0;
for (i = 0; i < N; i++) {
  xr = rand_v(x1, x2);
  sum += xr * xr - 3.0 * xr;
}
area = sum * (x2 - x1) / (float)N;
```

Parallelization of MonteCarlo Integration

- each iteration is independent of each other (thus naturally parallel)!
- hard challenge: generate random numbers such that the sequences of numbers are not statistically correlated among processes (local invocation of sequential random number generator on each process likely to lead to correlation!)
- one solution is to have a process devoted to issuing random numbers to the slaves



- in the next slide, we pursue this approach with the master process in charge of random number generation
- another approach is to use a parallel distributed version of a pseudo-random number generator (e.g., available at the SPRNG library); out of scope for this course

A Parallel Code for MonteCarlo Integration

Master:

```
n=... // # rand numbers in each chunk;
for (i = 1; i<N/n; i++) {
  for (j = 0; j < n; j++)
   xr[j] = rand_v(x1, x2);
  recv(&p_src, 1, any_proc, req_tag);
  send(xr, n, p_src, comp_tag);
}
for (i=1; i<nproc; i++) {
  recv(&p_src, 1, any_proc, req_tag);
  send(NULL, 0, p_src, stop_tag);
}
sum = 0.0;
reduce_add(&sum, 1, master);
area = sum * (x2 - x1) / (float)N;
```

Slave (me is the slave process id)

```
n = ... // # rand numbers in each chunk;
sum = 0.0;
send(&me, 1, master, req_tag);
recv(xr, n, master, any_tag, &tag);
while (tag == comp_tag) {
  for (i = 0; i < n; i++)
    sum += xr[i]*xr[i] - 3*xr[i];
  send(&me, 1, master, req_tag);
  recv(xr, n, master, any_tag, &tag);
}
reduce_add(&sum, 1, master);
```