Embarrassingly Parallel Applications

- Some history:
  - parallel processing was often viewed with suspicion
  - OK for "toy problems" but cost-effective for real work?
- Hurdles to overcome:
  - new algorithms
  - new languages (or so we thought...)
  - new machines (hypercubes, Sequent, dataflow, ...)
- The term "embarrassingly parallel" was coined to refer to problems that were "too easy" to solve
  - Geoffrey Fox (physicist at Caltech/Syracuse) [W&A]
  - Cleve Moler (mathematician, Intel/Mathworks)

Characteristics

- What makes an application “embarrassingly parallel”?
- Easy decomposition into subproblems
  - not necessarily recursive (self-similar) but that’s often the case
- Independent subproblems
  - no information passed between processes working on the subproblems
- Test: can you solve the subproblems in any order?
  - necessary but not sufficient...

Gordon Bell Prize

- Gordon Bell (DEC / CMU / NSF) established a prize in 1986
  - offered two $1K awards per year for 10 years
  - from the announcement of the 1988 competition:
    - Application: The problems run for this test must be complete applications, no computational kernels allowed. They must contain all input, data transfers from host to parallel processors, and all output.
    - The problems chosen should be the kind of job that a working scientist or engineer would submit as a batch job to a large supercomputer. In addition, I am arbitrarily disqualifying all problems that Cleve Moler calls "embarrassingly parallel". These include signal processing with multiple independent data streams, the computation of the Mandelbrot set, etc.
- Prize (now $10K) still awarded each year at Supercomputing
Performance

Ideal case: perfect speedup

"Master" process does decomposition
Takes one piece of the work itself
Collects, saves results

Examples (W&A Ch 3)

- Image processing
  - A polygon can be represented by vectors x and y (coordinates of the vertices)
  - Transformations involve independent operations on vector elements

\[
\begin{align*}
T_P &= T_S/n \\
T_P &= T_S/n
\end{align*}
\]

Examples (cont’d)

- Monte Carlo integration
  - For complicated (especially multi-dimensional) functions it may be easy to compute \( f(x,y,...) \) but hard to integrate
  - MC methods generate millions of random points \((x,y,...)\)
    - The integral is \( (#\text{points below the curve}) / (#\text{points}) \)

\[
\begin{align*}
x^2 + y^2 &= r^2 \\
A &= \pi r^2 \\
\pi/4 &= \int_0^1 \sqrt{1-x^2} \, dx
\end{align*}
\]

Mandelbrot Set

- The first "real" project this term
- MPI program to generate image
Set Definition

- The Mandelbrot set is a set of points in the imaginary plane:
  \[ c = x + yi \]
- For any point \( c \) consider the recurrence:
  \[ z_0 = 0 + 0i \]
  \[ z_k = z_{k-1}^2 + c \]
- For some points the values increase but for others the magnitude hovers near the origin.

Set Definition (cont’d)

- If the magnitude ever goes above 2.0 the series will start to increase from that point on.
- MS = the set of points that always stay near the origin.
- Image: use color map M with \((0 \leq N \leq 255)\) colors.
- The color for point \( c \) is:
  - black if series starting at \( c \) is in the set.
  - color \( M[j] \) if series has \( j \) elements less than 2.0.

Full Set

- The full set has lots of points on the \( x \) axis.
- The set is a fractal:
  - infinitely complex boundary.
  - self-similar patterns.
  - all points in the set are connected.

MPI Application

- Calculation of the Mandelbrot set is clearly embarrassingly parallel:
  - an image with \( n \times m \) pixels evaluates \( n \times m \) recurrences.
  - each series can be evaluated independently.
  - no particular order is required (row by row, column by column, ...).
- Plan (basic version):
  - process 0 gets coordinates from command line.
  - process 0 allocates parts of the image to a set of other processes.
  - other processes send back filled-in parts of the image.
  - process 0 writes complete image to file.
Static Decomposition

- In a static decomposition the parallel pieces are determined when the program starts.

Example:
- Divide image into N equal-size regions.
- Send each process:
  - (x, y) of upper left pixel
  - dx and dy (width, height of region).

Inefficiency of Static Decomposition

- It is unlikely each region will take the same amount of time:
  - Regions with lots of black pixels will require more iterations.
  - Region 3 will be finished before region 0...

Dynamic Allocation (Work Pool)

- A more efficient method: make more pieces of work, and allocate them on demand.

Example:
- Make 16 regions.
- Allocate the first four regions to the four processes.
- When a process finishes, give it the next region in the "pool".
- In this figure, process 3 might compute region 4...

Dynamic Decomposition

- The work pool strategy on the previous slide still used a static decomposition:
  - The number of regions is fixed at the beginning.
  - Dynamic allocation of regions to processes leads to better efficiency.

- For many applications it is also possible to define the jobs in the pool dynamically:
  - As the application progresses it discovers more work and adds it to the pool.
  - The application terminates when the pool is empty.
Dynamic Decomposition (cont’d)

- A trick to speed up the Mandelbrot application:
  - pick four points that define a rectangle in the image
  - evaluate the recurrence at these four points
  - if all corners are black then color all points in the interior black
- Not as accurate as full method but it’s a reasonable approximation

```
Dynamic Decomposition (cont’d)

- A dynamic version of the Mandelbrot application:
  
  ```
  Q = [new region(x0,xn,y0,yn)]
  while not Q.empty
      r = Q.remove_first
      if black_corners(r)
          r.fill(black)
      else
          Q.insert(new region(x0,mid(x0,xn),y0,mid(y0,yn)))
          Q.insert(new region(mid(x0,xn),xn,y0,mid(y0,yn)))
          Q.insert(new region(x0,mid(x0,xn),mid(y0,yn),yn))
          Q.insert(new region(mid(x0,xn),xn,mid(y0,yn),yn))
  ```

Load Balancing

- Process 0 does not always have to coordinate all tasks
- Each process can have its own local queue
- New jobs added to the local queue
- When a process runs out of jobs it can ask others for work
- Jobs can “migrate” between processes
- Many dynamic load balancing strategies

```
Project 2

- Download project outline (Makefile, etc) from web site
- First task: sequential implementation
  - command line args: coordinates of region
    - locations of points at corners
    - size of area covered by each pixel
- Save output in file, view with R
  - read pixel values into array
  - use `image` command to draw picture using array elements as colors
    ```
    > A = scan("mandelbrot.rda")
    > dim(A) = c(200,200)
    > image(A)
    ```
Project 2 (cont’d)

- Simple parallel version:
  - static decomposition and allocation
  - process 0 creates subprocesses, prints results
- Develop message “protocol”
  - what information does process 0 send to others?
  - what information will be returned?
  - how can this information be sent with MPI functions?
- Develop communication pattern (hint: think “hello, world”)
- NOTE: compare output of this version with sequential version
  % diff mandelbrot.rda mandelbrot.seq.rda

Writeup

- All programming projects should be submitted with documentation
- Some general guidelines:
  - what did you do for this project?
  - what are the main sections of the program?
  - is there anything in particular you want me to know about your project?
  - what is the output? how did you test the program?
  - how does the program perform?
  - what would you fix or add if you were to work on it some more?
- Neatness counts, and a picture is worth.....

Extras

- Implement a dynamic allocation strategy
- Implement dynamic decomposition (e.g. “black corners trick”)
- Link with a graphics library, make an interactive program
  - select region to “zoom in”
  - new image starts out all gray
  - render pieces as they are computed
- Other ideas?

A Note About Grades

- A working program with no frills and nothing special to recommend it usually earns about 80 points (out of 100)
- To earn 100, implement one or more of the extras, or do an excellent job on the writeup (performance analysis, etc), or an especially clean and elegant implementation (style matters)
- A program with one or more major flaws, or a missing or inadequate writeup, results in 60 or less