CISC 372: Parallel Computing

Data Distribution and Nearest Neighbor Communication

Stephen F. Siegel

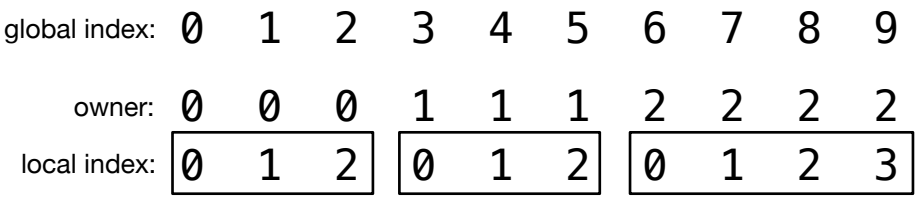
Department of Computer and Information Sciences
University of Delaware

Distributing arrays

The general problem

- given
  - an array a of length n
    - elements of a can be of any type
    - the important point is that the elements are ordered
    - indices (called global indices) run from 0 to $n-1$
  - the number of processes p
    - processes are numbered $0, 1, \ldots, p-1$
- determine a way to distribute the n elements among the p processes
  - for example, cyclic distribution
- each approach has advantages and disadvantages
  - e.g., cyclic distribution effective for embarrasingly parallel problems with clusters of longer-running tasks, like SAT
- different approaches are appropriate in different contexts
- in all cases: need easy way to convert between global and local views

Distribution example with $p = 3$, $n = 10$



- the sequential program has an array of length 10
- the parallel program has 3 processes
  - proc 0 has an array of length 3
    - 0: 0, 1: 1, 2: 2
  - proc 1 has an array of length 3
    - 0: 3, 1: 4, 2: 5
  - proc 2 has an array of length 4
    - 0: 6, 1: 7, 2: 8, 3: 9

Block Distribution Solutions

- each process owns a contiguous slice of the global array
  - example
    - rank 0 owns elements $0, 1, \ldots 4$
    - rank 1 owns $5, 6$
    - rank 2 owns nothing
    - rank 3 owns $7, 8, 9$
- the set of elements owned by process i can be specified by two numbers:
  - the number of elements owned by i
  - the first global index owned by i
- main advantage
  - many applications require frequent nearest neighbor communication
  - e.g.: to update $a[i]$, might need to read $a[i-1]$ and $a[i+1]$
  - increases probability that $a[i-1]$ and $a[i+1]$ will live on the same process as $a[i]$
  - communication will be minimized
  - cyclic distribution is very ineffective when nearest neighbor communication is required!

Additional Desirable Qualities of Block Distribution Solutions

- load balancing
- ideally, each process will own the same number of elements
- this is only possible if $p|n$
  - read "p divides n" (evenly)
  - means there exists an integer k such that $n = pk$
- additional requirement:
  - if $p|n$, all processes own n/p elements
  - otherwise, the number of elements owned by two different processes can differ by at most 1
    - some processes have $\lfloor n/p \rfloor$ elements ("small")
    - others have $\lceil n/p \rceil =$ elements ("big")

Note:

- $\lfloor x \rfloor =$ the greatest integer less than or equal to x (i.e., round down)
- $\lceil x \rceil =$ the least integer greater than or equal to x (i.e., round up)

Converting between local and global views in a block distribution

A block distribution of n elements over p processes must provide:

1. `FIRST`(r)
   - given a rank r, returns the first global index owned by proc r
2. `NUM_OWNED`(r)
   - given a rank r, returns the number of elements owned by r
3. `OWNER`(j)
   - given a global index j, returns the rank of the process owning j
4. `LOCAL_INDEX`(j)
   - given global index j, returns the local index of element j

- using these, you can easily convert between local and global view
- example: what is the global index corresponding to local index i on proc r?
  - answer: $i +$ `FIRST`(r)

The easy case: $p|n$

1. `FIRST`$(r) = r(n/p)$
2. `NUM_OWNED`$(r) = n/p$
3. `OWNER`$(j) = \lfloor j/(n/p) \rfloor$
4. `LOCAL_INDEX`$(j) = j\%(n/p)$

Example: $n = 12$, $p = 3$: each proc gets $n/p = 4$ elements:

global	0	1	2	3	4	5	6	7	8	9	10	11
owner	0	0	0	0	1	1	1	1	2	2	2	2
local	0	1	2	3	0	1	2	3	0	1	2	3

See `block1.c`, `block1_simp_mpi.c`.

The general case: the standard block distribution scheme

- $\texttt{FIRST}(r) = \lfloor rn/p \rfloor$
- $\texttt{NUM\_OWNED}(r) = \texttt{FIRST}(r+1) - \texttt{FIRST}(r)$
- $\texttt{OWNER}(j) = \lfloor (p(j+1) - 1)/n \rfloor$
- $\texttt{LOCAL\_INDEX}(j) = j - \texttt{FIRST}(\texttt{OWNER}(j))$

Intuition:

- If p divides n evenly, each proc gets n/p items.
- The first global index for rank i will be $i(n/p) = in/p$.
- Use same formula for first for the general case, but take floor.

Example: $n = 10$, $p = 3$:

- 0: first $= 0$
- 1: first $= \lfloor 10/3 \rfloor = 3$
- 2: first $= \lfloor 20/3 \rfloor = 6$
- 3: first $= \lfloor 30/3 \rfloor = 10 = n$
  - in general, $\texttt{FIRST}(p) = n$; there is no proc p but this is needed to compute $\texttt{NUM\_OWNED}(p-1)$

Work out these examples

1. $n = 14$, $p = 4$
2. $n = 14$, $p = 5$
3. $n = 2$, $p = 5$
4. $n = 3$, $p = 5$
5. $n = 18$, $p = 4$
6. $n = 18$, $p = 5$
7. $n = 1$, $p = 4$
8. $n = 10$, $p = 4$

$$\texttt{FIRST}(r) = \lfloor rn/p \rfloor$$
$$\texttt{NUM\_OWNED}(r) = \texttt{FIRST}(r+1) - \texttt{FIRST}(r)$$
$$\texttt{OWNER}(j) = \lfloor (p(j+1) - 1)/n \rfloor$$
$$\texttt{LOCAL\_INDEX}(j) = j - \texttt{FIRST}(\texttt{OWNER}(j))$$

Codes:

- `block1_mpi.c`
- `glob2loc.c`
- `loc2glob.c`

Example: block1.c

- sums the elements of an array of length N

```
#include <stdio.h>
#ifndef N
#define N 20
#endif
unsigned int a[N];
int main() {
  unsigned long sum = 0;
  for (int i=0; i<N; i++)
    a[i] = i*i;
  for (int i=0; i<N; i++)
    sum += a[i];
  printf("sum = %ld\n", sum);
}
```

Parallel version: block1_mpi.c

```
// Standard block distribution scheme: N items distributed over nprocs procs
#define FIRST(r) ((N)*(r)/nprocs)
#define NUM_OWNED(r) (FIRST((r)+1) - FIRST(r))
#define OWNER(j) ((nprocs*((j)+1)-1)/(N))
#define LOCAL_INDEX(j) ((j)-FIRST(OWNER(j)))

int main() {
  int nprocs, rank; // number of procs, rank of this proc
  int first;        // global index of first cell owned by this proc
  int n_local;      // number of cells owned by this proc

  MPI_Init(NULL,NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  first = FIRST(rank);
  n_local = NUM_OWNED(rank);
#ifdef DEBUG
  printf("Rank %d: first=%d, n_local=%d\n", rank, first, n_local);
#endif
```

Parallel version: block1_mpi.c, cont.

```
  unsigned int a[n_local]; // local block of global array a
  unsigned long sum = 0, global_sum;

  for (int i=0; i<n_local; i++) {
    const int j = first + i; // convert from local to global index
    a[i] = j * j;
  }
  for (int i=0; i<n_local; i++)
    sum += a[i];
  MPI_Reduce(&sum, &global_sum, 1, MPI_UNSIGNED_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
  MPI_Finalize();
  if (rank == 0) printf("sum = %ld\n", global_sum);
}
```

Nearest neighbor communication and Pascal's triangle

Usual representation:

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{array}
$$

Computer representation:

$$
\begin{array}{ccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 2 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 3 & 0 & 3 & 0 & 1 & 0 \\
1 & 0 & 4 & 0 & 6 & 0 & 4 & 0 & 1 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{array}
$$

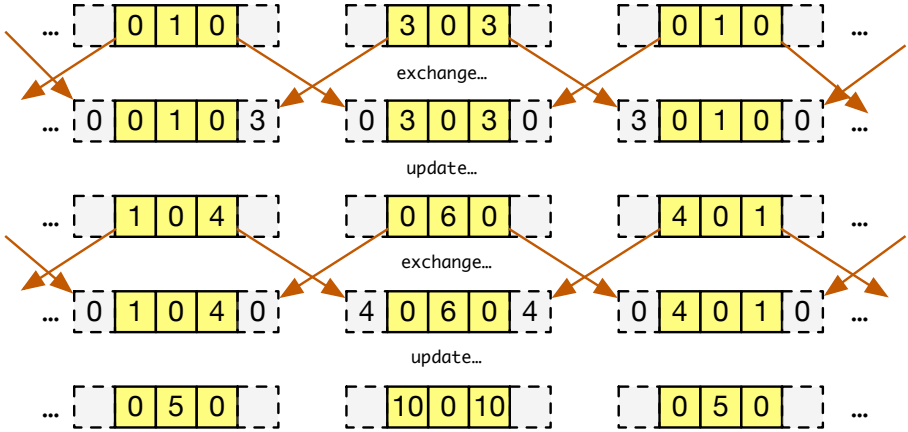
Pascal: sequential implementation

- see `pascal.c`
- two arrays are used
  - one always holds the current value
  - the other holds the previous value
- note use of pointer swapping

Pascal: parallel implementation with MPI

- block distribute arrays
- problem: how to update left and right endpoints of each block?
  - these depend on values on neighboring procs
  - not embarrassingly parallel — communication is required
- solution: **ghost cells**
  - each proc will have two extra cells
    - one on left to mirror value of left neighbor's right endpoint
    - one on right to mirror value of right neighbor's left endpoint
    - these are not owned by this proc—they duplicate information
  - at each iteration:
    - print
    - exchange ghost cells
    - perform the local update

Pascal: ghost cell exchange



- the length of the array on proc r is `NUM_OWNED`$(r) + 2$
- indexes are shifted up by 1
- see `pascal_mpi.c`

Pascal: printing

- all output is funneled through rank 0
  - proc 0 is the only proc that prints
  - exception: debugging (doesn't have to look perfect)
  - this is the only reliable way to get the output right
    - with tools currently at your disposable
    - MPI's I/O commands are the real "right" way
- all procs with positive rank:
  - send their (non-ghost) data to rank 0
- proc 0:
  - prints its own block (excluding ghosts)
  - loops $i = 0..\texttt{nprocs} - 1$
    - receives a block from proc i into the "scratch" buffer
    - prints that block
  - prints a newline and returns

1-dimensional Diffusion

- Problem
  - a metal rod of unit length is initially $100°$
  - a block of ice is placed at either end to keep ends at $0°$
  - heat diffuses out of rod
  - find the temperature at each point on the rod at each time
- flow of heat governed by the diffusion equation
  - a simple differential equation
  - $u = u(t, x)$ temperature function

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

  - α is a constant depending on the material
    - thermal diffusivity

Discretization

- divide rod into n discrete pieces of length Δx
- let `u[i]` be the temperature of the i^{th} piece
- loop over time
  - $\Delta t =$ duration of one discrete time step
- update formula comes from discrete approximations to the first and second derivatives
  - continuous

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

  - discrete

```
u_new[i] =  u[i] + k*(u[i+1] + u[i-1] - 2*u[i])
```

  - `k` $= \alpha \Delta t / \Delta x^2$
  - need $0 < k < 0.5$ for convergence

Implementations

- `diffuse1d.c`, `diffuse1d_mpi.c`
  - plain text output
  - all output funneled through process 0
  - similar to Pascal
- `diffusion1d.c`, `diffusion1d_mpi.c`
  - uses ANIM and MPIANIM libraries for graphical output

2-d Diffusion

- a metal unit square
  - initially $100°$
  - temperature on perimeter kept at $0°$
- $u = u(x, y, t)$ temperature function
- 2d diffusion equation

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

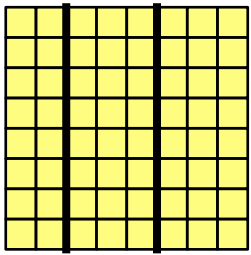
- discretization

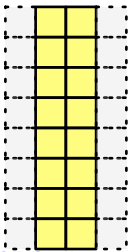
```
u_new[i][j] =  u[i][j]
  + k*(u[i+1][j] + u[i-1][j]
    + u[i][j+1] + u[i][j-1] - 4*u[i][j]);
```

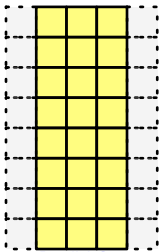
Parallelization of diffusion2d

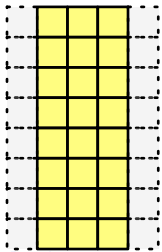
- how to distribute the 2d spatial domain?
- "striped" decompositions
  - apply the Standard Block Distribution Scheme to the columns
    - "column distribution"
    - each process gets a certain number of x values
    - a ghost cell column on the left and on the right
    - exchange ghost columns after each time step
  - apply the Standard Block Distribution Scheme to the rows
    - "row distribution"
    - ...

2d Diffusion: column distribution

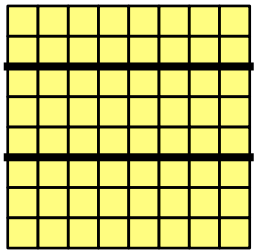


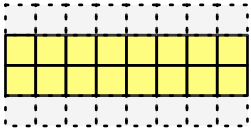


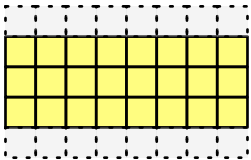


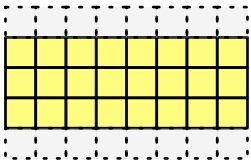


2d Diffusion: row distribution

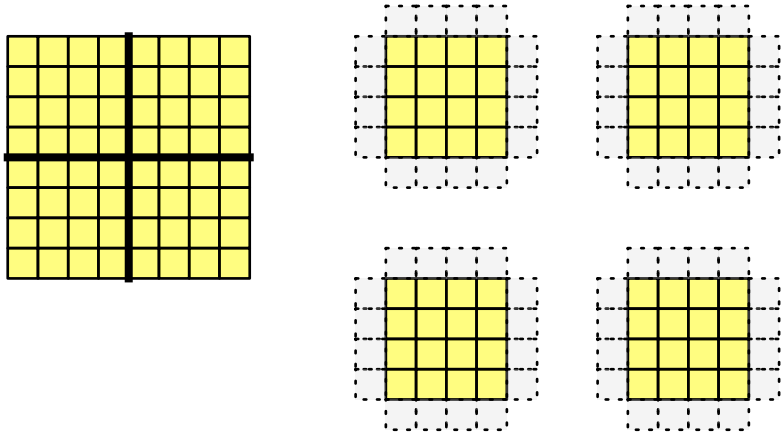








2d Diffusion: checkerboard decomposition



- 4 ghost regions for each process
- 4 exchanges: up, down, left, right

Analysis of Diffusion2d Decomposition

- assume an $n \times n$ grid, p processes
- measure the total "amount" of communication
  - roughly, the total number of ghost cells
- striped
  - on each process, there are $2n$ ghost cells
  - total number of ghost cells: $2np$
- checkerboard
  - assume p is a perfect square!
  - the p processes are arranged in a grid of dimension $\sqrt{p} \times \sqrt{p}$
  - each process has $4(n/\sqrt{p})$ ghost cells
  - total number of ghost cells: $4(np/\sqrt{p}) = 4n\sqrt{p}$
- conclusion: asymptotically
  - striped: $O(p)$
  - checkerboard: $O(\sqrt{p})$