Cornell Theory Center

Virtual Workshop Module

Parallel Program Design

Daniel Sverdlik

Video Introduction

View with modem or broadband connection
Read the text transcript

Table of Contents

---

1. Goals, Decisions

1.1 Goals (ideal)

Speedup is defined as the serial execution time divided by the parallel execution time, for a given problem size and number of processors. Perfect speedup (generally not attainable) would equal the number of processors. Scalability looks at how speedup is preserved as the problem size and number of processors increases.

Ideal (read: unrealistic) goals for writing a program with maximum speedup and scalability:

- Each process has a unique bit of work to do, and does not have to redo any other work in order to get its bit done.
- Each process stores the data needed to accomplish that work, and does not require anyone else's data.
- A given piece of data exists only on one process, and each bit of computation only needs to be done once, by one process.
- Communication between processes is minimized.
- Load is balanced; each process should be finished at the same time.

Usually it is much more complicated than this!

Keep in mind that:

- There may be several parallel solutions to your problem.
- The best parallel solution may not flow directly from the best serial solution.

1.2 Major Decisions

Data or Functional Parallelism?

- Partition by task (functional parallelism)
  - Each process performs a different "function" or executes a different code section
  - First identify functions, then look at the data requirements
  - Commonly programmed with message-passing libraries

- Partition by data (data parallelism)
  - Each process does the same work on a unique piece of data
  - "Owner computes"
    First divide the data. Each process then becomes responsible for whatever work is needed to process that data.
  - Data placement is an essential part of a data-parallel algorithm
  - Data parallelism is probably more scalable than functional parallelism
  - Can be programmed at a high-level with High Performance Fortran (HPF), or at a lower-level with message-passing libraries.

- These can be used in combination. A program can be partitioned by function; each function can then be partitioned by data. In addition, there are some cases in which the distinction between the two categories blurs.

- Additional material on data and functional parallelism is in the module Fundamentals of Distributed Memory Computing, specifically section 9.

SPMD or Master Worker?

- Single Program Multiple Data (SPMD)
  - All processes run the same program, operating on different data. This model is particularly appropriate for problems with a regular, predictable communication pattern. These tend to be scalable if all processes read/write to files and if global communication is avoided.

- Master Worker
  - A single program (called the Master) coordinates the work done on all the processes. These are called Workers. The Master may or may not contribute to computation. This model has limited scalability due to the communication bottleneck caused by all Workers needing to communicate with a single Master.
  - The Master and Workers may all run the same program, or different programs. If they are running the same program, conditional (if) statements cause different tasks to run different code segments.

- The background material on SPMD and Master Worker is in the module Fundamentals of Distributed Memory Computing, specifically section 9.

- Additional material on data and functional parallelism is in the module Fundamentals of Distributed Memory Computing, specifically section 9.

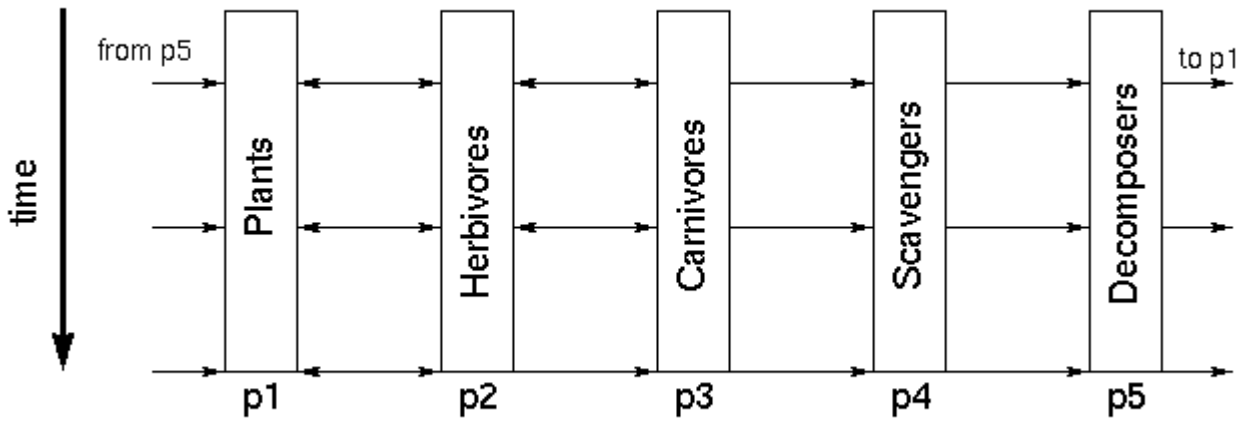
Table of Contents 1 2 3 4 5 Less Detail

---

2. Examples of Functional Parallelism

This section contains two examples of types of problems that could be solved using functional parallelism.

2.1 Ecosystem Modeling

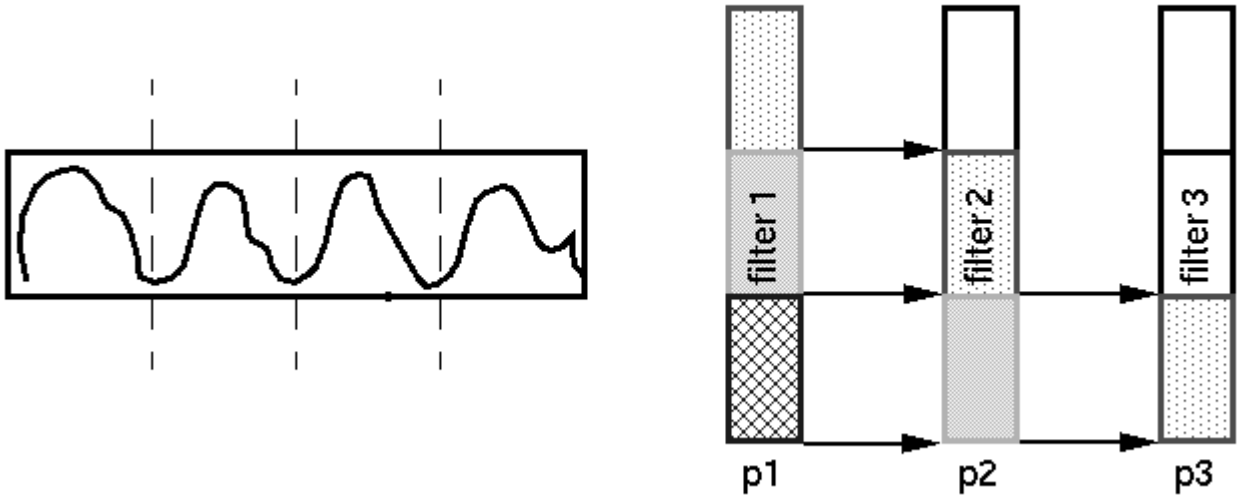


The diagram shows five processes, each running a different program. In this case, each program calculates the population of a given group, where each group's growth depends on that of its neighbors. As time progresses, each process calculates its current state, then shares information with the neighbor populations, so they can all go on to calculate the state at the next time step.

The load balancing for this program is static (pre-scheduled) -- each process' load is determined and inflexible at the start of the application. It is also likely to be unequal, with the different programs requiring different amounts of computation before sharing state.

The communication pattern is a ring. This will influence how the different programs are mapped to physical processors. Those programs that need to communicate should ideally be only one communication "hop" from each other.

2.2 Audio Signal Processing (pipeline)



To process the audio signal, the data set is passed through three distinct computational filters. Each filter is a separate process. The first chunk of data must pass through the first filter before progressing to the second. When it does, the second chunk of data passes through the first filter. By the time the third chunk of data is in the first filter, all three processes are busy.

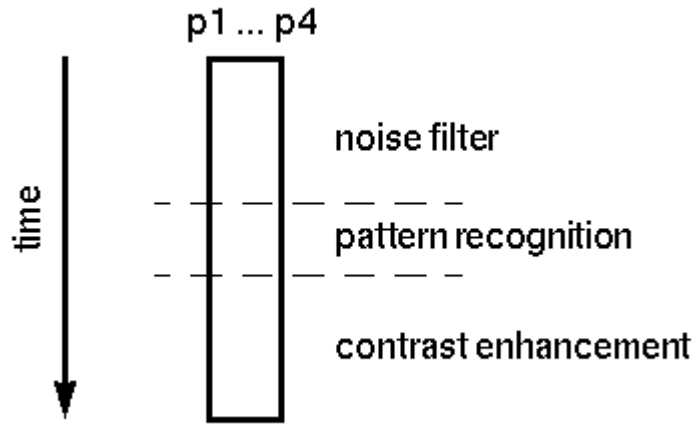
Again, load balancing is static and will be unequal if different filters require different amounts of computation. The communication pattern is a 1-dimensional mesh.

Table of Contents 1 2 3 4 5 Less Detail

3. Examples of Data Parallelism

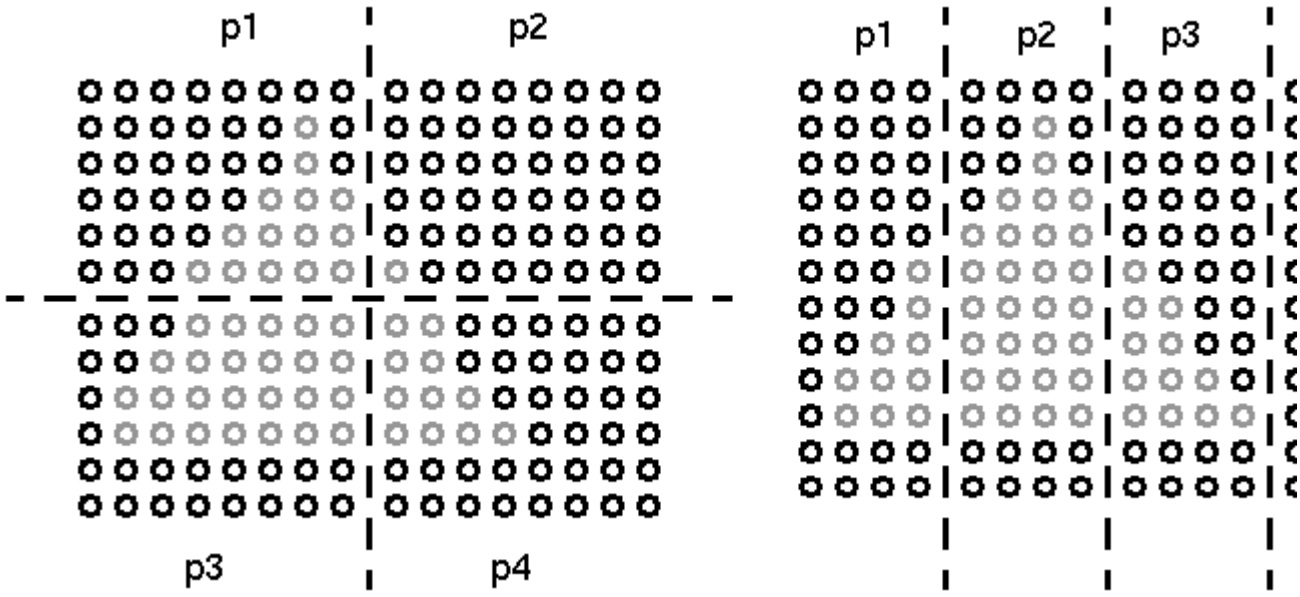
This section contains three examples of types of problems that could be solved using data parallelism.

3.1 Image Processing



The diagram above shows a SPMD solution. All four processes are running the same three-step program, processing different data. The dashed horizontal lines represent barriers or synchronization points. Each process must complete that step before all processes can proceed to the next step.

The sketches below show an image that will be processed. The green points represent locations where there is an image, the black points where there is not. The layout on the left shows a 2-D block decomposition, on the right a 1-D block decomposition.



Data decomposition:

- Domain characteristics
  - Large multidimensional matrix encoding location and color
  - Homogeneous members -- all points require the same amount of calculation
  - The value calculated for each point will depend on neighboring values. For example, a green point can be identified as "noise", rather than as part of the pattern, by recognizing that it is surrounded by black points.

- Goals for domain decomposition
  - Balance computational load
  - Minimize and regularize communication between processes

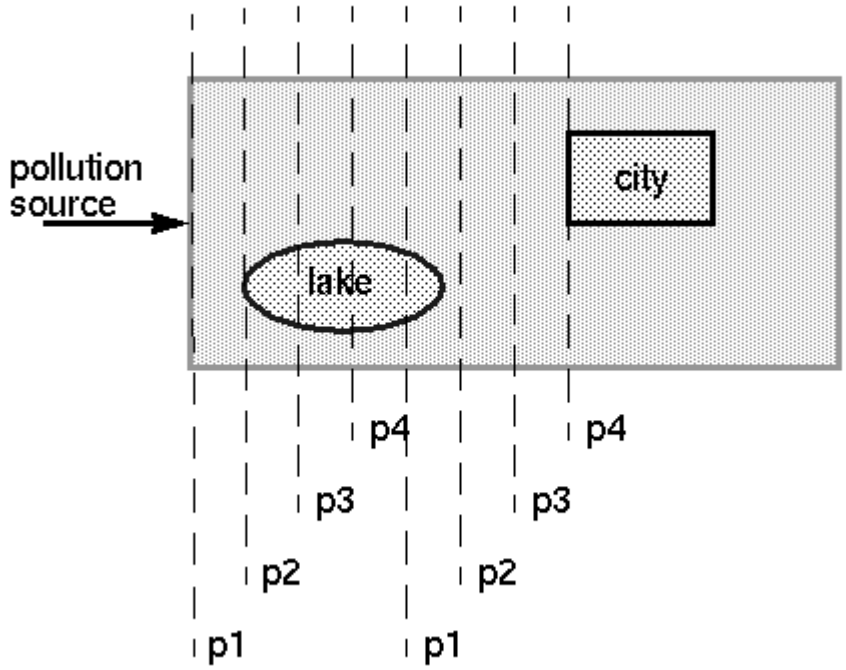
- Technique
  - Block decomposition

    Assign each process a contiguous set of points. Since each point requires the same amount of work, points should be divided evenly between processes. Load balancing is static since the division of work is determined at compile time.
  - The communication pattern must be considered when deciding upon a decomposition geometry. Higher dimension decompositions are generally preferable, since they minimize surface to volume ratio.

    Processes will need to exchange their outer rows and columns with neighboring processes. In the 2-D decomposition shown above, each process has 13 edge points which will need to be communicated, for a total of 52 points. In the 1-D decomposition, p1 and p4 each have 12 edge points, and the middle processes have 24, for a total of 72.

- What if domain members are not homogeneous?
  - If the green points require more computation than the black, neither decomposition is ideal. The 1-D decomposition is slightly more imbalanced than the 2-D. More sophisticated methods are needed to decompose the image.

3.2 Effect of Pollution on Forested Areas



This program calculates the effect of pollution on tree growth and mortality for a geographic area.

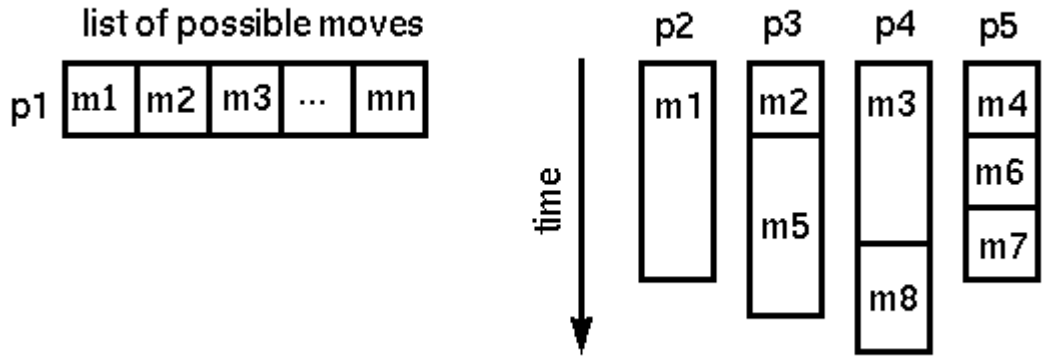
- SPMD
  - Each process will run the same program.

- Domain
  - Irregular geometry, dynamic
    The domain is irregular because it is broken up by the lake and the city (no trees). It is dynamic because the computation shifts with time. Initially, only trees near the pollution source may be affected, so may require more computation. By the end of the simulation, these trees may have died (now requiring no computation) and trees far from the source may be showing long-term effects.
  - Non-homogeneous members
    Different tree species may be affected in different ways, and thus require different amounts of computation.
  - No interaction between members (trees)

- Technique
  - The diagram shows cyclic decomposition, where each process is dealt slices of data in round-robin fashion. The areas of no work (lake and city) are now divided between multiple processes. As computation requirements shift with time, the load should remain balanced. This model works well for this program because there is no communication

pattern based on location of members.

3.3 Chess



In this example, a computer is playing chess. In choosing its next move, it will analyze responses to all possible moves for one or more rounds into the future.

- Domain
  - Non-homogeneous members
    Different moves require different amounts of computation. One may allow checkmate on the next response, so can be eliminated immediately. Others may require projection further into the future.
  - No interaction between members
    When analyzing one move, no information is needed about other moves.

- Technique
  - Self-scheduling (or "pool of tasks")
    The figure shows the Master process on the left, ready to distribute data on possible moves to the Workers. The four Workers on the right are all doing the same analysis on different data (different moves). When a Worker is finished analyzing a move, the Master assigns it another move to analyze.
  - Dynamic load balancing
    The specific work assigned to processes is determined at run time.

- The communication pattern is one to all
  - This program will not scale if there is one process managing the pool of tasks. The Master will become a bottleneck as the number of Workers

increases.

Table of Contents 1 2 3 4 5 Less Detail

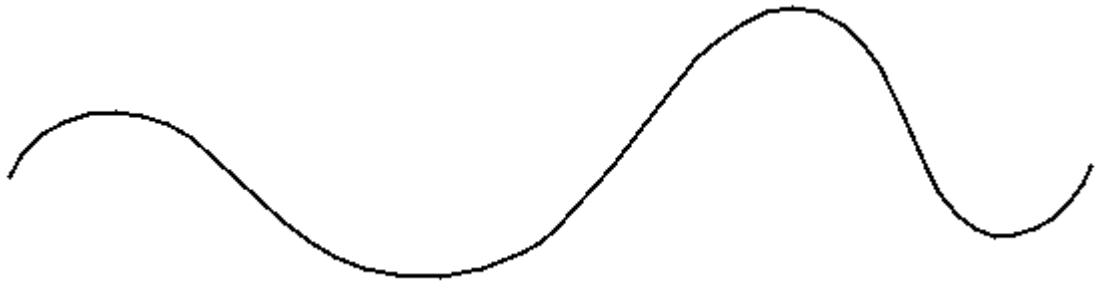
---

4. Walk Through

In this section we will walk through the parallel program development of a simple problem.

4.1 Problem Description

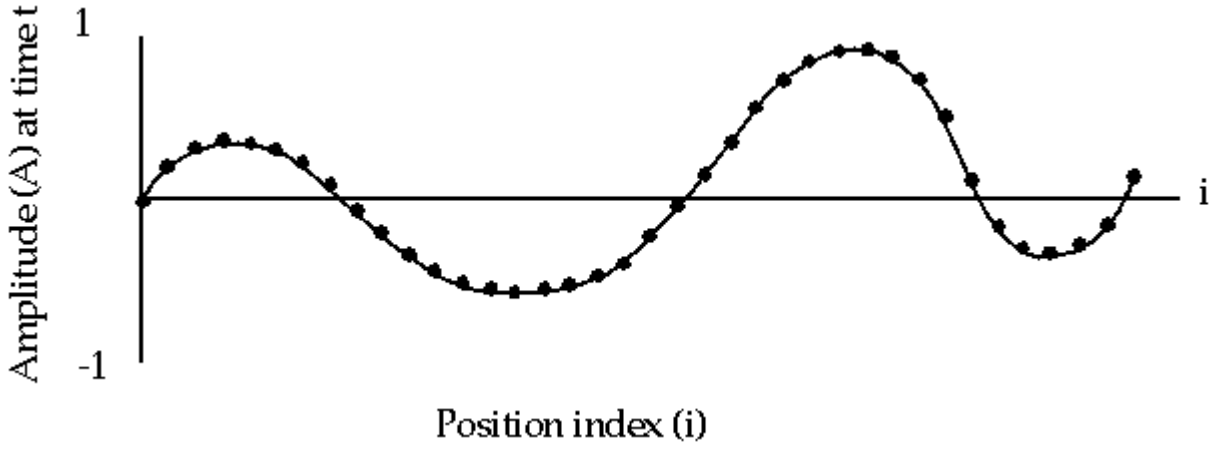
This problem was developed from a description found in *Fox et al. (1988) Solving Problems on Concurrent Processors, vol. 1. Prentice Hall.*



Calculate the amplitude along a uniform, vibrating string after a specified amount of time has elapsed.

Numerical Solution

First, impose a framework on the problem.



The framework includes *amplitude* on the y axis, *i* as the position index along the x axis, and node points imposed along the string. The amplitude will be updated at discrete time steps.

The equation to be solved is the one-dimensional wave equation:

A(i,t+1) = (2.0 * A(i,t)) - A(i,t-1)
+ (c * (A(i-1,t) - (2.0 * A(i,t)) + A(i+1,t)))

where c is a constant.

Note that amplitude will depend on previous timesteps (t, t-1) and neighboring points (i-1, i+1).

4.2 Decomposition

4.2.1 Decompose the work by *function* or *data*?

Function: divide the computation into chunks of disjoint or unassociated work

- All the work in this problem is accomplished in one loop. It would be contrived (and more work!) to split it up.

Data: Give each process a subset of a domain

- The domains for this problem are position along the string (i), and time (t)
- Which can be computed concurrently?
  - TEST FOR DATA INDEPENDENCE:
    If the calculation of the value of an element of an array in a loop is based upon the value of another element of the same array, calculated in a different iteration, then the iterations cannot be done concurrently.
    Examples:
    F(I)= F(I-1) is NOT independent
    F(I)= F(I)*2 is independent
    F(I)= G(I-1) is independent
  - Time fails this test: A(i,t+1) requires A(i,t) and A(i,t-1).
    This is called a data dependence. (It seems intuitive that a simulation can't be decomposed by time -- this is a more concrete test)
  - Position passes this test: A(i,t+1) does not require any other values at (t+1).

- What data is required?
  - Amplitude depends on values at neighboring points for the previous timestep.
    A(i,t+1) requires A(i-1,t) and A(i+1,t)
  - This will result in communication overhead and idle time spent waiting for processes with unequal work loads to "catch up".

4.2.2 How to decompose by position?

Data replication

- Some program parameters are replicated, but not the amplitude array
  In general, if large amounts of data are replicated, this will limit the size of the subset of the domain that can be stored on one process, and thus the largest problem size that can be solved. This is not a problem for the wave code.

Load balancing

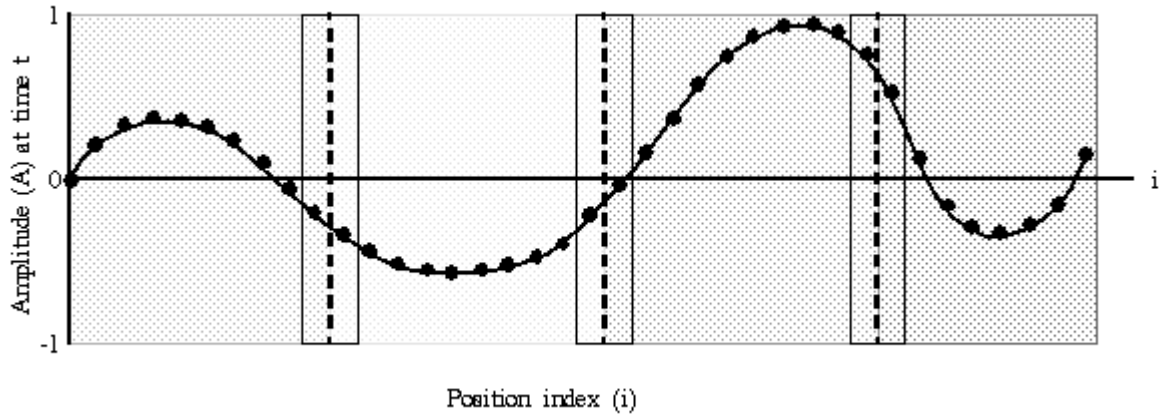
- All points require equal work, so the points should be divided equally amongst the processes.
- Cyclic
  A cyclic decomposition would have each point given to the available nodes in turn until all are distributed, like dealing cards to the players. In this distribution, the work load is identical for each set of points +/- one point.
- Block
  A block decomposition would have the work split into the number of nodes, leaving contiguous data points on the same node. In this distribution, the work load identical for each block of points +/- one point.

Communication

- Cyclic
  Neighboring points are all assigned to different processes, therefore each point would require communication.
- Block
  Only the point at each end of the block requires communication, so the larger the block size, the smaller percentage of communication overhead.

Block decomposition by position

Each color represents a different process. The boxes at the edges of each color indicate that the endpoints will require communication at each time step with the neighboring process.



4.3 Code Structure

1. Read in starting values
2. Establish communication channels
3. Divide data among processes

4. Exchange endpoints
5. Calculate amplitude for new time step

   Repeat last two steps for given number of time steps

6. Output results

4.4 SPMD Solution

This program is:

- SPMD because all processes run the same program
- DATA PARALLEL because the work is partitioned by data
- SCALABLE because none of the work is constrained by one process and no global communication is required

Reading/writing data

- All tasks read in the number of points along the string and the number of time steps
- Reading in the array of initial amplitudes can be done by
  - All tasks read in all data, throw out all but their chunk
  - or- Supply initial data in separate files, each task reads in its own file
  - or- Read in appropriate lines from a direct-access file. This method is illustrated in the complete wave code linked to on the following page.

4.4.1 Pseudo Code

```
      program wave_spmd

C     Learn number of tasks and taskid
      call initialize task
      call get task identification and information

C     Identify left and right neighbors

C     Get program parameters
      read tpoints, nsteps

C     Divide data amongst processes
      read values

C     Update values for each point along string
      do t = 1, nsteps
C        Send to left, receive from right
            call send left endpoint to left neighbor
            call receive left endpoint from right neighbor
C        Send to right, receive from left
            call send right endpoint to right neighbor
            call receive right endpoint from left neighbor

C        Update points along line
         do i = 1, npoints
           newval(i) = (2.0 * values(i)) - oldval(i)
     &     + (sqtau * (values(i-1) - (2.0 * values(i)) + values(i+1)))
         end do

      end do

C     Write results out to file
      write values
```

```
      call terminate parallel environment
```

Click here for a more fully-developed pseudo code using MPI calls.

Click here for the complete program.

4.5 SPMD with Master Worker Embedded

This program is:

- SPMD because all processes run the same program
- DATA PARALLEL because the work is partitioned by data
- MASTER WORKER because flow control assigns certain work to a "Master" or "Worker"

Reading/writing data

- Master reads in number of points along the string and number of time steps, broadcasts to all Workers
- Master reads in the array of initial positions, then sends a chunk to each Worker
- After final time step, each Worker sends its chunk of the array back to the Master
- Master writes out final results

4.5.1 Pseudo Code

All	`      program wave_mw`
	
	`C     Learn number of tasks and taskid`
	`      call` **`initialize task`**
	`      call` **`get task identification and information`**

Master	`C     Get program parameters`
	`      if (taskid .eq. MASTER) then`
	`         read tpoints, nsteps`
	
	`C        Master broadcasts total points, time steps`
	`         call` **`send two numbers to all Workers`**

Workers	`      else`
	`C        Workers receive total points, time steps`
	`         call` **`all Workers receive two numbers`**

	`      end if`

Master	`      if (taskid .eq. MASTER) then`
	`         do i = 1, tpoints`
	`            read(10) values(i)`
	`         end do`
	
	`C        Master sends chunks to Workers`
	`         do i = 1, nproc-1`
	`C              Send first point and number of points handled t`
	`              call **send two numbers**`
	
	`C              Send chunk of array to Worker`
	`              call **send chunk of array**`
	`         end do`

Workers	`      else`
	`C        Receive first point and number of points`
	`         call **receive two numbers**`
	
	`C        Receive chunk of array`
	`         call **receive chunk of array**`
	
	`      end if`

All	`C     Update values along the wave for nstep time steps`
	`      do t = 1, nsteps`
	
	`C        Send to left, receive from right`
	`            call **send left endpoint to left neighbor**`
	`            call **receive left endpoint from right neighbor**`
	`C        Send to right, receive from left`
	`            call **send right endpoint to right neighbor**`
	`            call **receive right endpoint from left neighbor**`
	
	`C        Update points along line`
	`            do i = 1, npoints`
	`               ***newval(i) = (2.0 * values(i)) - oldval(i) +***`
	`     ***&      (sqtau * (values(i-1) - (2.0 * values(i)) + values***`
	`            end do`
	
	`      end do`

Master	`C     Master collects results from Workers and prints`
	`      if (taskid .eq. MASTER) then`
	`         do i = 1, nproc - 1`
	`C           Receive first point and number of points`
	`            call **receive two numbers**`
	
	`C           Receive results`
	`            call **receive chunk of results**`

	C Write out results write results(i)

Workers	else
	C Send first point and number of points handled to Mast
	call **send two numbers**
	
	C Send results to Master
	call **send results**
	end if

All	call **terminate parallel environment** end

Click here for a more fully-developed pseudo code using MPI calls.

Click here for the complete Fortran program.
Click here for the complete C program.

Table of Contents 1 2 3 4 5 Less Detail

---

5. Repositories

Online locations of parallel programming solutions to a variety of computational problems.

- High Performance Fortran Applications from CRPC and NPAC

Table of Contents 1 2 3 4 5 Less Detail

References

Ian Foster
"Designing and Building Parallel Programs"
1995 Addison-Wesley Publishing Company, Inc
Online version of this book

Geoffrey C. Fox
"Solving Problems on Concurrent Processors"
1988 Prentice Hall

Quiz Take a multiple-choice quiz on this material, and submit it for grading.

Evaluation Please complete this short evaluation form. Thank you!

Last modified on 09/23/2002 13:13:07