Multi-core Programming Fundamental Concepts of Parallel Programmng

Based on slides from Intel Software College

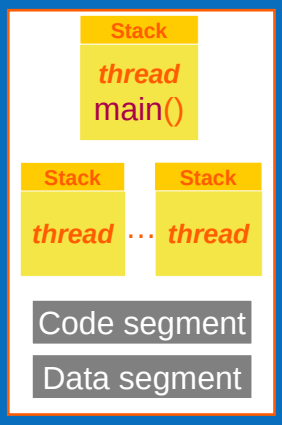
and

Multi-Core Programming –

increasing performance through software multi-threading

by Shameem Akhter and Jason Roberts,

Processes and Threads



Modern operating systems load programs as processes

- Resource holder
- Execution

A process starts executing at its entry point as a thread

Threads can create other threads within the process

- Each thread gets its own stack

All threads within a process share code & data segments

Agenda

Design concepts

- Threading for functionality or performance?
- Threading for throughput or turnaround?
- Decomposing the work

Why Use Threads

Benefits

- Increased performance
  - Easy method to take advantage of multi-core
- Better resource utilization
  - Reduce latency (even on single processor systems)
- Efficient data sharing
  - Sharing data through memory more efficient than message-passing

Risks

- Increases complexity of application
- Difficult to debug (data races, deadlocks, etc.)

Threading for Functionality

Assign threads to separate functions done by application

- Easiest method since overlap is unlikely

Example: *Building a house*

Bricklayer, carpenter, roofer, plumber,...

Threading for Performance

Increase the performance of computations

Thread in order to improve turnaround or throughput

Examples

- Automobile assembly line
  - Each worker does an assigned function
- Searching for pieces of Skylab
  - Divide up area to be searched
- US Postal Service
  - Post office branches, mail sorters, delivery

Threading for Throughput or Turnaround?

Turnaround

Complete single task in the smallest amount of time

Example: *Setting a dinner table*

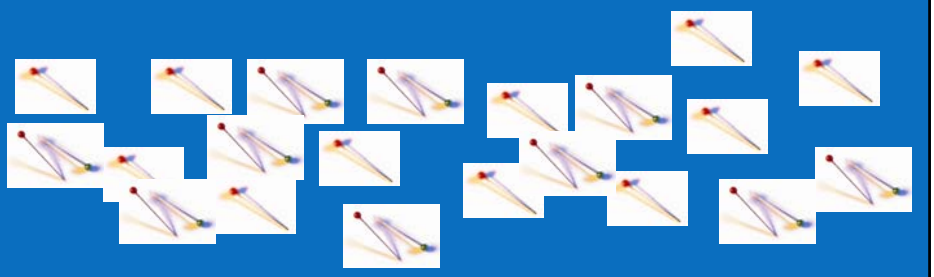
- One to put down plates
- One to fold and place napkins
- One to place utensils
  - Spoons, knives, forks
- One to place glasses

Threading for Throughput or Turnaround?

Throughput

Complete the most tasks in a fixed amount of time

Example: *Produce pins*
"One man draws out the wire, another straights it, a third cuts it, a fourth points it, a fifth grinds it at the top for receiving the head;..." (A. Smith 1776)



Latency

Guaranteed latency instead of other characteristics

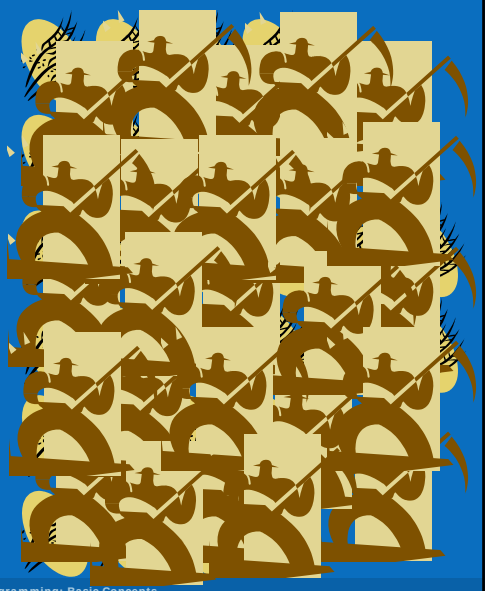
- Sample: ‘Real-Time’ OS

Granularity

Loosely defined as the ratio of computation to synchronization

Be sure there is enough work to merit parallel computation

Example: Two farmers divide a field. How many more farmers can be added?



Granularity and Load Balance

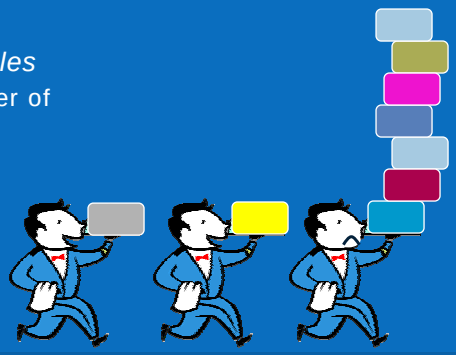
Load Balance

Most effective distribution is to have equal amounts of work per thread

- Threads that finish first sit idle
- Threads should finish close to same time

Example: *Busing banquet tables*

- Better to assign same number of tables to each bus person



Designing for Threads

- In sequential programming, work through series of tasks in sequence
- For User Interaction, normal method is loop handling user events
  - Ex: on button push perform procedure, then return to wait for next user action
- Relatively simple since only one thing is happening at a time
- To move to parallel programming need to change viewpoint
  - See program as set of tasks with dependencies between them
  - Programmer must *decompose* the program into these tasks

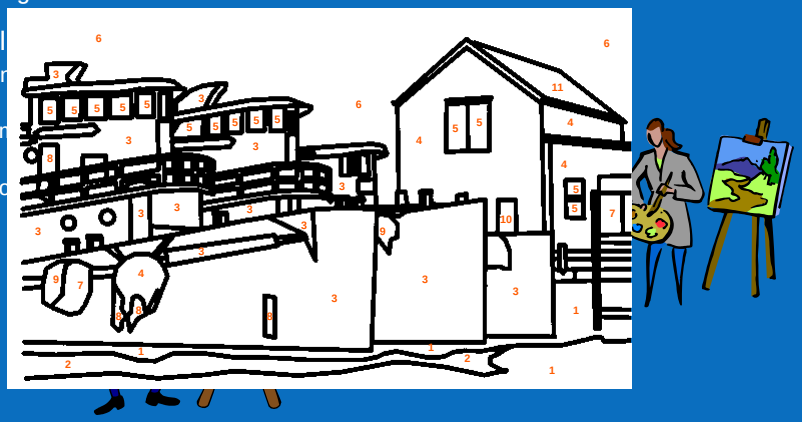
Major Forms of Decomposition

- Task
  - Different activities assigned to different threads
  - Common in GUI apps
- Data
  - Threads perform same operations on different blocks of data
  - Common in audio processing, imaging, scientific programming
- Data Flow
  - One thread's output is input for next
  - Special care needed to minimize startup and shutdown latencies
  - Examples: parsing then code generation in compilers

Task Decomposition

Divide computation based on natural set of independent tasks
- Assign data for each task as needed

Exampl
- Pair
- Nur
- Two



Task Decomposition

- Word Processing
  - One task permits the user to enter text
  - Other task paginates the file in the background
- On long file this prevents user having to wait until entire file is read in and paginated
- Try to decompose into independent tasks
- In *embarrassingly parallel* or *perfectly parallel* problems there are no dependencies
- In *replicated data* problems the dependencies can be removed by replicating the replicating some or all of the data

Data Decomposition

Large data sets whose elements can be computed independently
- Divide data and associated computation among threads

Example: Grading test papers
- Multiple graders with same key

What if different keys are needed?

Data Decomposition
Data Level Parallelism

- Recalculating the values in a large spreadsheet
  - Divide data and associated computation among threads
  - Give half to each of 2 threads or
  - 1/n to each of n threads

- As number of cores increase can increase problem size
  - Still complete in same time
  - Benefit from Gustafson's Law

Data Flow Decomposition

- Often critical issue is how data flows between tasks
  - Producer/consumer problems are good example
  - One thread's output is input for next
  - Consumer can't start until producer has produced some data
    - Startup latency
  - Consumer finishes producing but consumer has to finish consuming the data
    - Shutdown latency
  - Special care needed to minimize these startup and shutdown latencies
  - More generally, can be delays due to dependencies which must be minimized
  - Need to avoid consumer threads idling

Implications of Decompositions

- Task decomposition often easier if tasks easily partitioned
- Data decomposition usually adds some code complexity
- Aim is normally better performance
- Choice often determined by problem domain
- Example: processing images in video stream
  - If no dependency between frames can do both
  - Task: one decode, other color balances, etc
  - Data: Each handles frame
- How to choose
  - Try to do modelling of performance
  - Code and do empirical timing and evaluation

Challenges

1. Synchronization
   - Method by which 2 or more threads coordinate actions
2. Communication
   - Bandwidth and latency issues of exchanging data
3. Load Balancing
   - Equi-distribution of work across threads
4. Scalability
   - Challenge of making efficient use of larger number of threads

Parallel Programming Patterns

- Object Oriented Programming tries to use patterns to logically design applications
- Can do similarly for parallel programming

Pattern	Decomposition
Task level parallelism	Task
Divide and Conquer	Task/Data
Geometric Decomposition	Data
Pipeline	Data Flow
Wavefront	Data Flow

Parallel Patterns

- Divide and Conquer
  - Divide problem into sub-problems
  - Solve each independently
  - Aggregate the results to get final answer
  - Ex: find minimum of set, merge sort set
    - Very easy to parallelize, exhibits good locality – good for cache usage
- Geometric Decomposition
  - Each thread processes data 'chunks' from overall data structure
  - Ex: heat flow, wave propagation
- Pipeline
  - Like assembly line, break task into stages, have threads work on individual stages

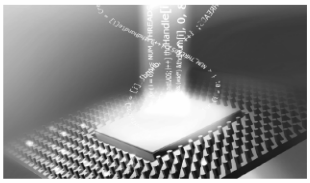
Parallel Patterns

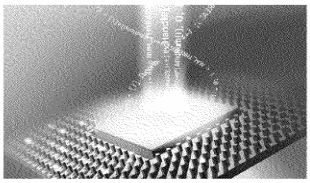
- Wavefront
  - Useful for processing data elements along diagonal in 2d grid
  - The numbers indicate the dependencies of cells
  - That is,
    - ones with 2 depends on ones with 1
    - ones with 3 on ones with 2 and 1 etc
  - Critical to minimize idle time of threads
    - Load balancing is key

Error Diffusion Algorithm Floyd and Steinberg 1975

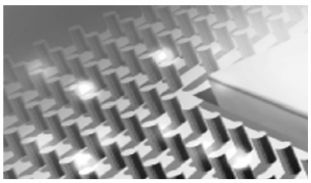
- Used in computer graphics and image processing
- Technique for displaying continuous tone images on devices with limited color range e.g. black-and-white printer
- Must simulate multiple shades of grey using approximation
- Example: 8 bit to 1 bit

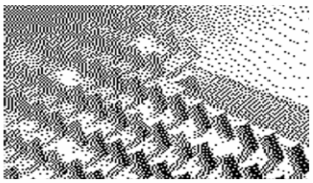
Error Diffusion - Example





Original 8-bit image on the left, resultant 2-bit image on the right. At the resolution of this printing, they look similar.





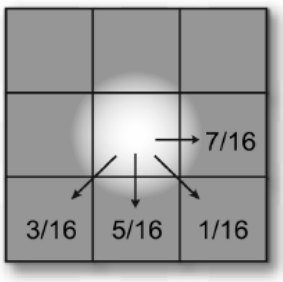
The same images as above but zoomed to 400 percent and cropped to 25 percent to show pixel detail. Now you can clearly see the 2-bit black-white rendering on the right and 8-bit gray-scale on the left.

Figure 3.2 Error Diffusion Algorithm Output

Error Diffusion – Simple 3 Step Process

1. Determine output value from input
   - quantization or thresholding (1 bit)
   - [0,127] -> 0 ; [128,255] -> 1
2. Compute error between input and normalized output (0 or 255)
   - if input is 170 error is -85
3. Distribute error on fractional basis to neighboring pixels as in figure

- Repeat for each pixel



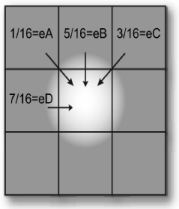
C Implementation of Error Diffusion

```
/*************************************
* Initial implementation of the error diffusion algorithm.
*************************************/
void error_diffusion(unsigned int width,
                     unsigned int height,
                     unsigned short **InputImage,
                     unsigned short **OutputImage)
{
   for (unsigned int i = 0; i < height; i++)
   {
      for (unsigned int j = 0; j < width; j++)
      {
         /* 1. Compute the value of the output pixel*/
         if (InputImage[i][j] < 128)
            OutputImage[i][j] = 0;
         else
            OutputImage[i][j] = 1;

         /* 2. Compute the error value */
         int err = InputImage[i][j] - 255*OutputImage[i][j];

         /* 3. Distribute the error */
         InputImage[i][j+1]   += err * 7/16;
         InputImage[i+1][j-1] += err * 3/16;
         InputImage[i+1][j]   += err * 5/16;
         InputImage[i+1][j+1] += err * 1/16;
      }
   }
}
```


Analysis of the Error Diffusion Process

- Previous pixel's error must be known to compute value of next pixel
  - Might seem that inherently serial
- Need to reformulate the problem

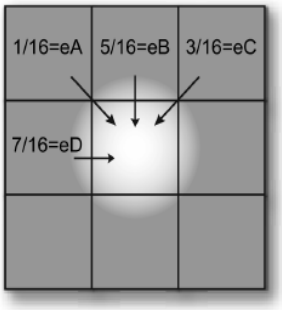
- Data Flow problem since need 4 values before can compute pixel value
- Need to determine dependencies and order of processing



In this case, we look at the error propagation from the perspective of the receiving pixel.

Figure 3.4 Error-Diffusion Error Computation from the Receiving Pixel's Perspective

A Parallel Approach

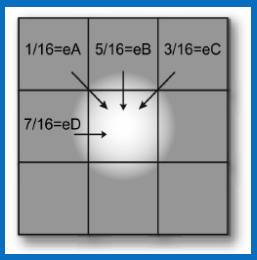


In this case, we look at the error propagation from the perspective of the receiving pixel.

Figure 3.4 Error-Diffusion Error Computation from the Receiving Pixel's Perspective

A Parallel Approach

- Want load balancing
- Try 1: one does even pixels in row, one odd
  - each would be blocked waiting on other
- Try 2
  - Need 3 values from previous row
  - one from left
  - So assign thread to process a row
  - A thread can start next row once needed pixels available
    - in this case 2 pixels, so have 2 pixels latency per row – not significant
- On 8.5"x11" page at 1200dpi have 10,200 pixels per row



Multi-thread Multi-Row Parallel Error Diffusion

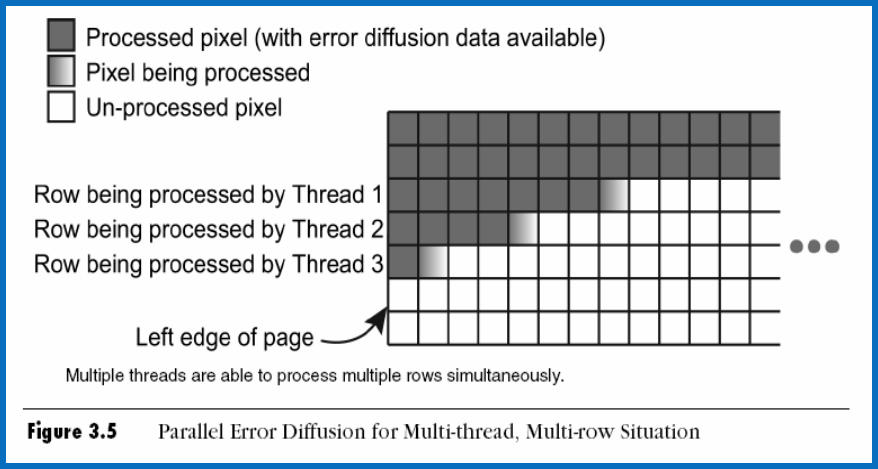
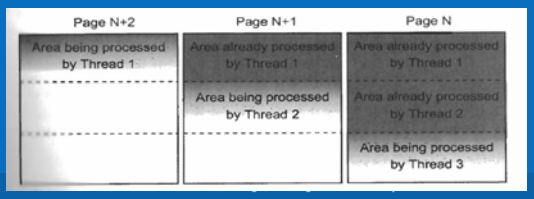


Figure 3.5 Parallel Error Diffusion for Multi-thread, Multi-row Situation

Other Alternatives

- Higher Level of Granularity
  - have each thread process a different page !!
  - Disadvantage:
    - Increased memory usage – page takes 131MB
    - image may spans pages; may be only one page to process
- Hybrid Approach
  - subdivide pages and process as below
  - latency now 1/3 page for thread 1, 2/3 page for thread 2



Key Points