The University of Texas at Arlington

Lecture 5

CSE 5343/4342
Embedded Systems II

Based heavily on slides by Dr. Roger Walker

Embedded II - Projects

- Teams of 2 students per team may be formed – teams have already been formed.
- Each team should select a class project involving multi-core/multi-threaded system(s) for embedded applications. Example topics include:
  - Write a new or convert an existing program to run on a multi-core/multi-threaded system around an embedded application.
  - Investigate, write and report on a particular area of a multi-core/multi-threaded system around an embedded application.
  - Select an application, design and build a system using multi-core/multi-threaded processing around an embedded application.

Embedded II – Projects (cont'd)

- **Some previous projects include:**
  - A device driver for Data Translation DT9816 with real-time Linux, Modifying;
  - DTConsole for uses with DT9800 series modules, Literature review of recent multi-treading/multi-core applications and, Conversion of Pavement Profiler and Automated Texture Measurement System from single core to multi-core.

Embedded II – Projects (cont'd)

- **Project Proposal:**
  - Projects proposed must obtain my approval before beginning. An initial proposal should be turned in to me by Thursday, March 4th.
  - Proposals should include:
    - a semi-detailed plan of what you propose to do
    - list the names and email addresses of each member of your group

Fundamental Concepts of Parallel Programming

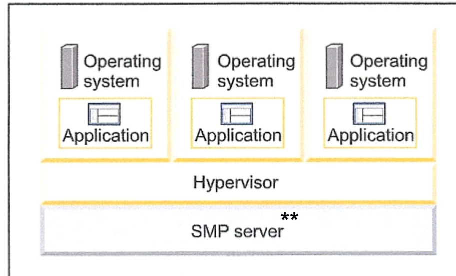
(Textbook - Chapter 3)

Lecture Topics

1. Additional Information on VMM Systems (IBM View)
2. Moving to Parallel Processing
3. Threads Role in Parallel Processing
4. Basic Concepts in Parallel Computing

VMM Information*

- VMM – two types are used for implementing hypervisors.
- Type 1 hypervisors run directly on the system hardware.



*http://publib.boulder.ibm.com/infocenter/eserver/v1r2/topic/eicay/eicay.pdf

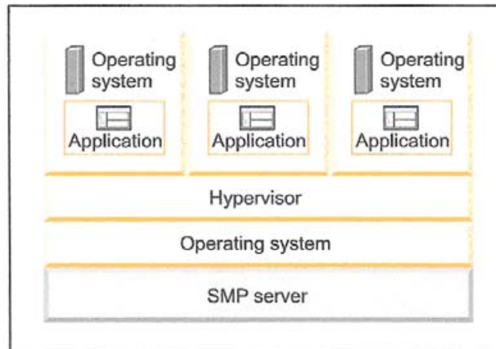
**Symmetric MultiProcessing (SMP) Server

Type 1 Hypervisors

- Hypervisors are currently classified in two types: [1]
- A **Type 1** (or **native** or **bare-metal**) hypervisor is software that runs directly on a given hardware platform (as an operating system *control program*). A **guest operating system** thus runs at the second level above the hardware. The classic type 1 hypervisor was CP/CMS, developed at IBM in the 1960s, ancestor of IBM's current z/VM. More recent examples are Xen, VMware's ESX Server, L4 microkernels, TRANGO, IBM's LPAR hypervisor (PR/SM), Microsoft's Hyper-V (currently in Beta), and Sun's Logical Domains Hypervisor (released in 2005). A variation of this is embedding the hypervisor in the firmware of the platform, as is done in the case of Hitachi's Virtage hypervisor. KVM, which turns a complete Linux kernel into a hypervisor, is also Type 1.

TYPE 2 Hypervisor

- Type 2 hypervisors run on a host operating system that provides virtualization services, such as I/O device support and memory management.



Type 2 Hypervisors

- A **Type 2** (or **hosted**) hypervisor is software that runs within an operating system environment. A "guest" operating system thus runs at the third level above the hardware. Examples include VMware Server (formerly known as GSX), VMware Workstation, VMware Fusion, the open source QEMU, Microsoft's Virtual PC and Microsoft Virtual Server products, InnoTek's VirtualBox, as well as SWsoft's Parallels Workstation and Parallels Desktop.

Hypervisors Supported by IBM systems

IBM supports at least one hypervisor for each IBM system. The following table lists IBM systems and the hypervisors that they support.

Table 1. IBM systems and supported hypervisors

Hypervisor	Hypervisor type	IBM system that runs hypervisor
Microsoft® Virtual Server	Type 2	System x™ and BladeCenter®
POWER5™	Type 1	System i™ and System p™
Processor Resource/System Manager	Type 1	System z™
VMware ESX Server	Type 1	System x and BladeCenter
VMware GSX Server	Type 2	System x and BladeCenter
VMware Server	Type 2	System x and BladeCenter

For other systems using Types 1 and 2 see::
http://en.wikipedia.org/wiki/Virtual_machine_monitor

Clock Speed Increase Has Flattened Out

- Recall the reasons for going to Parallel Processing
- Problems caused by higher speeds
  - Excessive power consumption
  - Heat dissipation
  - Current leakage
- Power consumption critical for mobile devices
- Mobile computing platforms increasingly important
  - Retail laptop sales now exceed desktop sales
  - Laptops may be 35% of PC market in 2007
  - **Embedded Market**
- New strategy: Limit CPU speed and sophistication; put multiple CPUs (“cores”) on a single chip

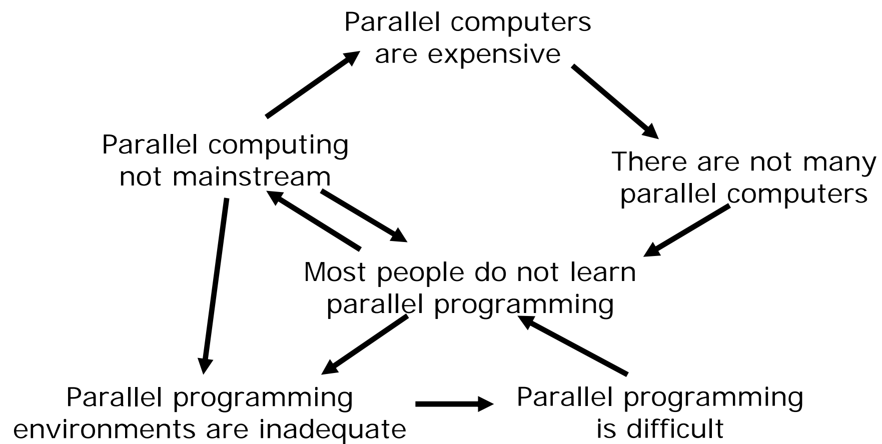
12

Early History of Parallel Computing

- Multiple-processor systems supporting parallel computing
- 1960s: Experimental systems
- 1980s: Microprocessor-based commercial systems

13

Old Dynamic of Parallel Computing

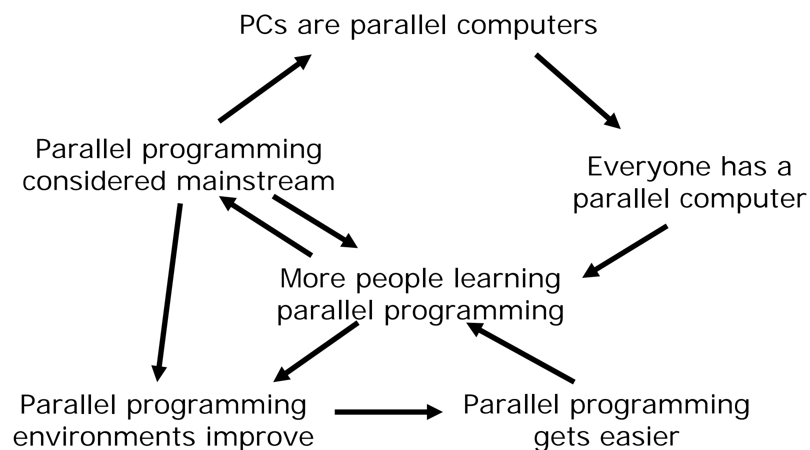


14

Late History of Parallel Computing (Intel View)

- 2004: Intel demos dual-core CPU
- 2006: Intel demos quad-core CPU
- Clovertown sub-architecture (Core) supposedly scalable to 32+ cores

New Dynamic of Parallel Computing



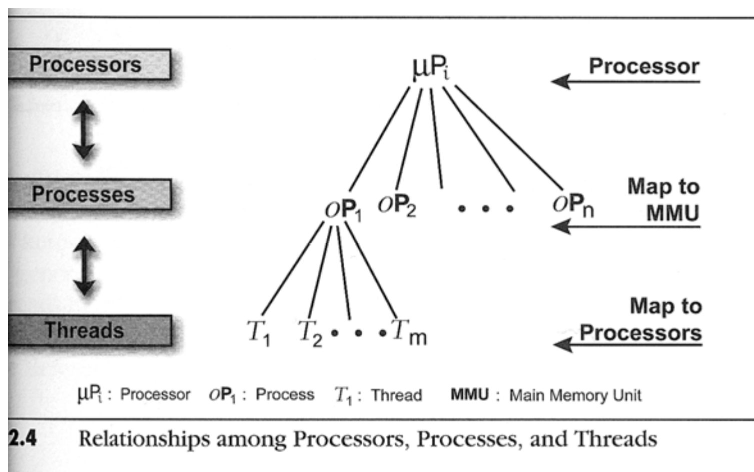
Thread Initiation

Recall:

- A process can have one or more threads, each of which operates independently.
- Threads within the same process share the same address space, certain resources but have their own stack.

17

Processors-Threads Relationship



2.4 Relationships among Processors, Processes, and Threads

18

Thread Concepts in Parallel Programming

- Parallel Programming Concepts center on the design, development and deployment of threads within an application.
- The program performs a sequence of actions. Sequential programs are relatively simple as only one thing happens at a time.

19

Methodology

- To move from linear model to parallel programming model designers must rethink the idea of process flow
- Study problem, sequential program, or code segment
- Look for opportunities for parallelism
- Try to keep all processors busy doing useful work

20

Ways of Exploiting Parallelism

- Data decomposition (Domain): different activities assigned to different threads (e.g., GUIs)
- Task (functional) decomposition: Multiple threads performing the same operation but on different datasets (e.g., signal processing, scientific computing)
- Pipelining (Data Flow): one thread's output is the input to the next thread

Task Decomposition

- Decomposing a program by the functions that it performs is called task decomposition.
- One of the simplest ways to achieve parallel execution.
- Running tasks in parallel this way usually requires slight modifications to the individual functions to avoid conflicts and to indicate that these tasks are no longer sequential.

Task Decomposition – Examples

Examples:

Weed and mow your garden

One gardener *to weed and one* gardener *to mow*

Grading

The job is to *grade a large stack of answer sheets* for a test. Break stack into sub-stacks or tasks

Paint-by-Numbers

Painting a single color is a single task
Number of tasks = number of colors
Two artists: one does even, other odd

23

Weed And Mow Your Lawn – Example

Consider a situation in which you want to weed and mow your lawn. You have two gardeners. You can assign the task to the gardeners based on the type of activity.

Example: Weeding and Mowing a Lawn

24

Weed And Mow Your Lawn – Example (cont'd)

- **Example:** If we were discussing gardening, task decomposition would suggest that gardeners be assigned tasks based on the nature of the activity: if two gardeners arrived at a client's home, one might mow the lawn while the other weeded. Mowing and weeding are separate functions broken out as such. To accomplish them, the gardeners would make sure to have some coordination between them, so that the 'weeder' is not sitting in the middle of a lawn that needs to be mowed.

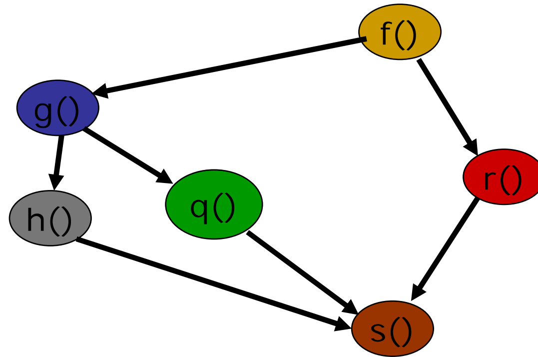
25

Task (Functional) Decomposition

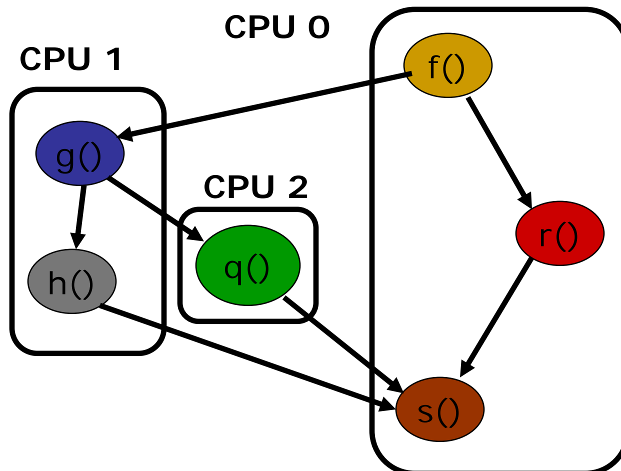
- First, divide tasks among processors
- Second, decide which data elements are going to be accessed (read and/or written) by which processors

Task (Functional) Decomposition

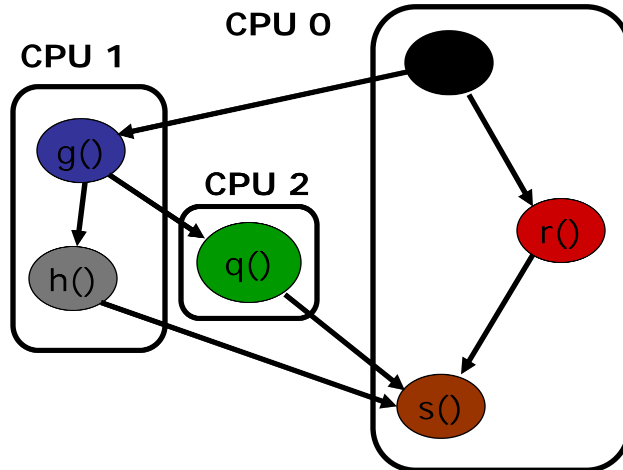
In a task decomposition we look for functions that can execute simultaneously. In this drawing the arrows represent the precedence constraints among the functions. For example, function “h” cannot execute until function “g” is completed, and function “s” cannot execute until functions “h”, “q”, and “r” are done.



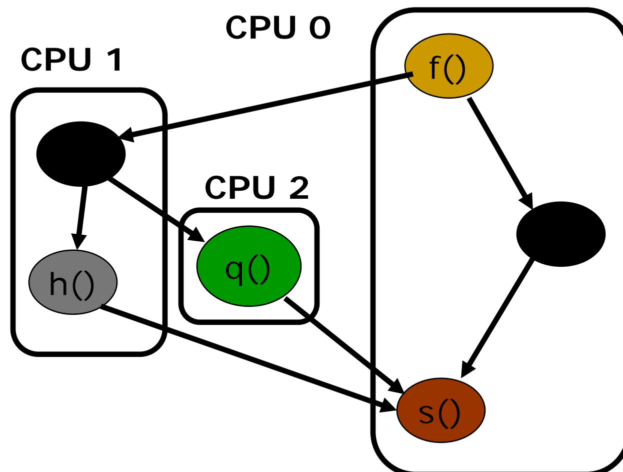
Task (Functional) Decomposition



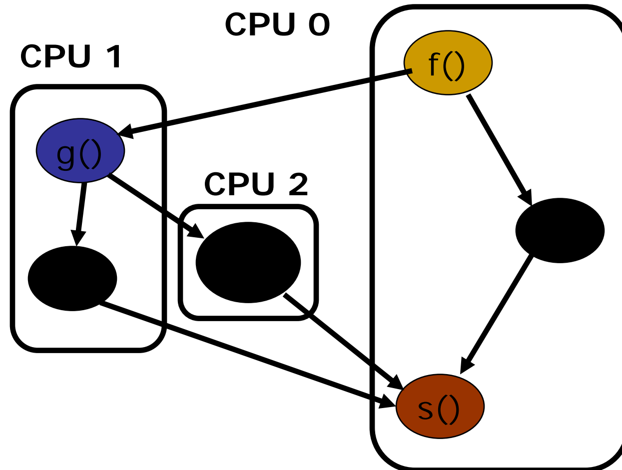
Task (Functional) Decomposition



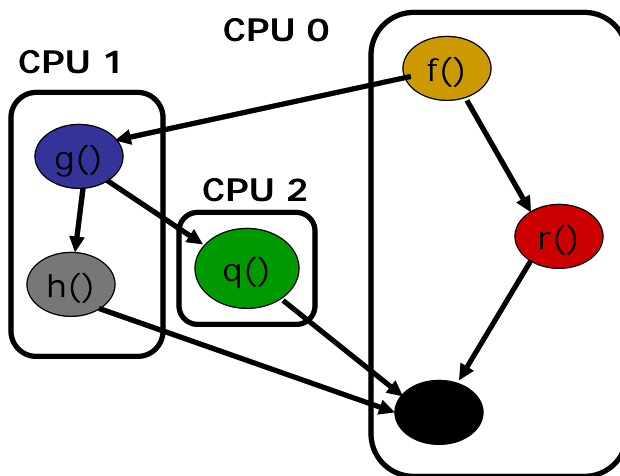
Task (Functional) Decomposition



Task (Functional) Decomposition



Task (Functional) Decomposition



Data Decomposition

- Dividing large data sets whose elements can be computed independently and associating the needed computation among threads is known as data decomposition.
- Key points to remember about data decomposition are:
  - The same independent operation is applied repeatedly to different data.
  - Computation-intensive tasks with a large degree of independence like computation-intensive loops in applications are good candidates for data decomposition.

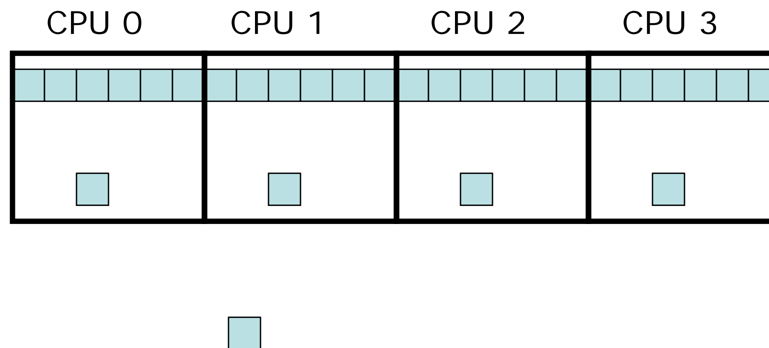
33

Data (Domain) Decomposition Example

Find the largest element of an array

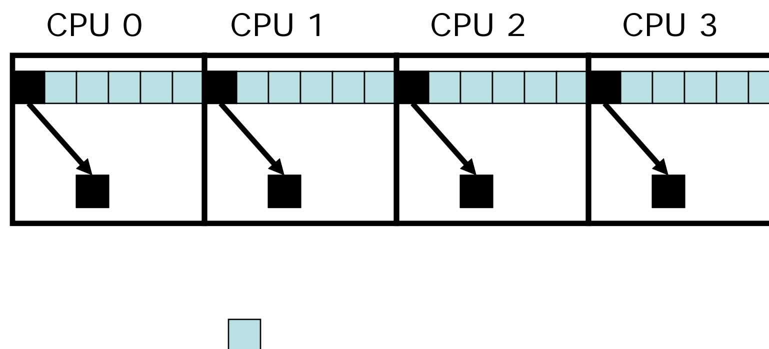
Data (Domain) Decomposition Example (cont'd)

Find the largest element of an array



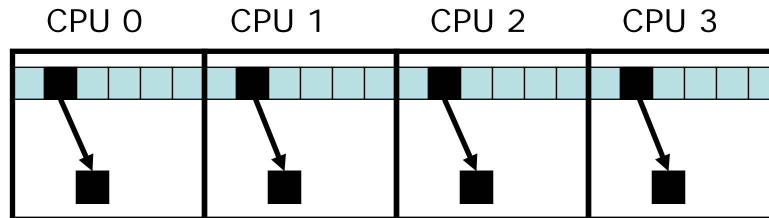
Data (Domain) Decomposition Example (cont'd)

Find the largest element of an array



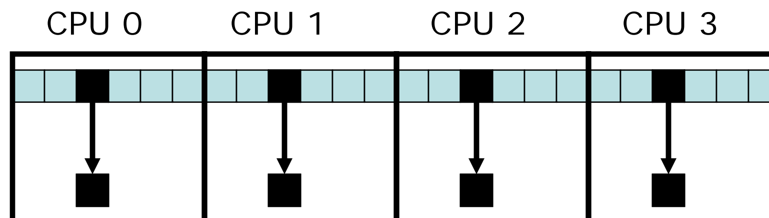
Data (Domain) Decomposition Example (cont'd)

Find the largest element of an array



Data (Domain) Decomposition Example (cont'd)

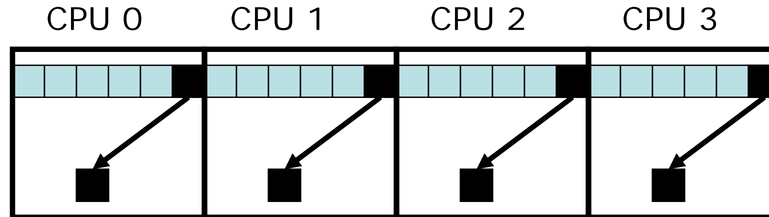
Find the largest element of an array



Data (Domain) Decomposition Example (cont'd)

Find the largest element of an array

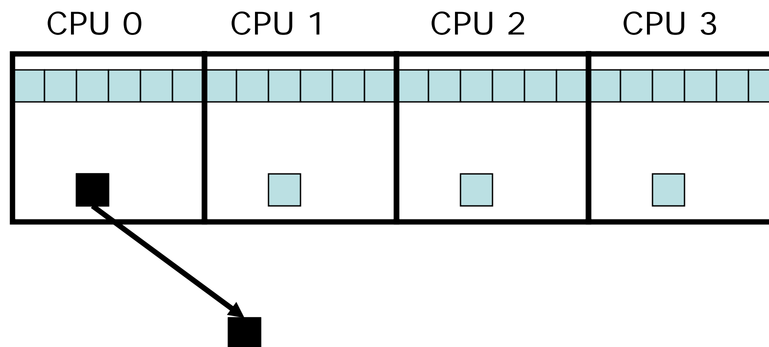
CPU 0 CPU 1 CPU 2 CPU 3



Data (Domain) Decomposition Example (cont'd)

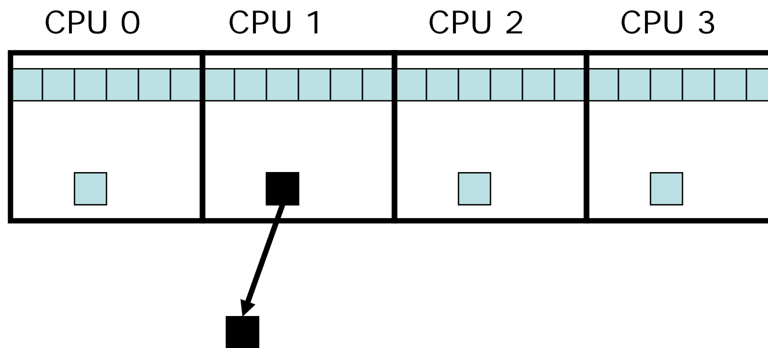
Find the largest element of an array

CPU 0 CPU 1 CPU 2 CPU 3



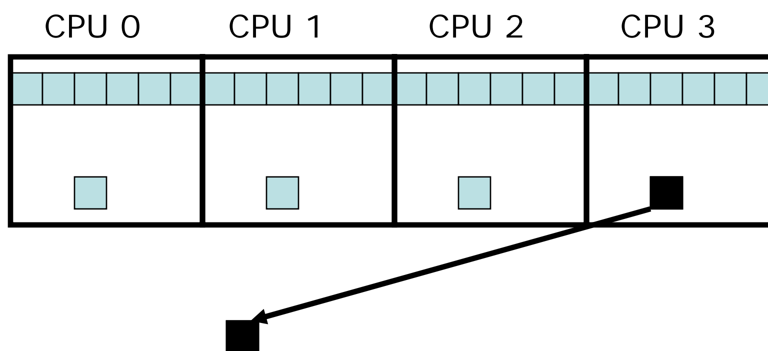
Data (Domain) Decomposition Example (cont'd)

Find the largest element of an array



Data (Domain) Decomposition Example (cont'd)

Find the largest element of an array



Data Decomposition

- As the number of processor cores increases, data decomposition allows the problem size to be increased. This allows for more work to be done in the same amount of time.
- To illustrate, consider the gardening example. Two more gardeners are added to the work crew. Rather than assigning all four gardeners to one yard, we can we can assign the two new gardeners to another yard, effectively increasing our total problem size. Assuming that the two new gardeners can perform the same amount of work as the original two, and that the two yard sizes are the same, we've doubled the amount of work done in the same amount of time.

43

Which Decomposition Method to Use?

Task decomposition and data decomposition can often be applied to the same problem; choose depending on the number of available resources and the size of the task.

Examples:

- **Data decomposition:** In the mural painting example, you can divide the wall into two halves and assign each of the artists one half and all the colors needed to complete the assigned area.
- **Task decomposition:** In the final exam grading problem, if the graders take a single key and grade only those exams that correspond to that key, it would be considered task decomposition. Alternatively, if there are different types of questions in the exam, such as multiple choice, true/false, and essay, the job of grading could be divided based on the tasks to specialists in each of those question types.

44

Data Flow Decomposition

- Many times, when decomposing a problem, the critical issue isn't what tasks should do the work, but how the data flows between the different tasks. In these cases, data flow decomposition breaks up a problem by how data flows between tasks.
- The producer/consumer problem is a well known example of how data flow impacts a programs ability to execute in parallel. The output of one task, the producer, becomes the input to another, the consumer. The two tasks are performed by different threads, and the second one, the consumer, cannot start until the producer finishes some portion of its work.

45

Delays in Data Flow

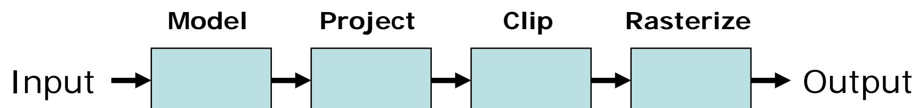
- One gardener prepares the tools, that is, he puts gas in the mower, cleans the shears, and other similar tasks for both gardeners to use. No gardening can occur until this step is mostly finished, at which point the true gardening work can begin. The delay caused by the first task creates a pause for the second task, after which both tasks can continue in parallel. In computer terms, this particular model occurs frequently.

46

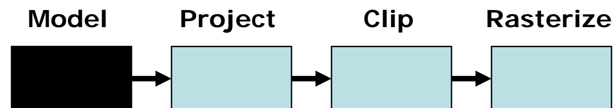
Data Flow Decomposition Example

- Special kind of task decomposition
- “Assembly line” parallelism
- Example: 3D rendering in computer graphics

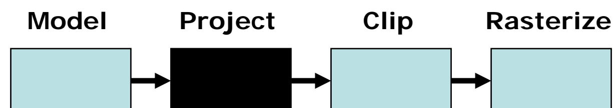
Data Flow Decomposition Example - Pipelining



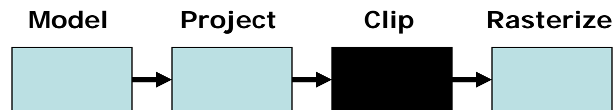
Data Flow Decomposition Example – Step 1



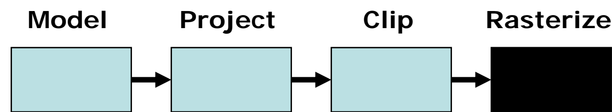
Data Flow Decomposition Example – Step 2



Data Flow Decomposition Example – Step 3

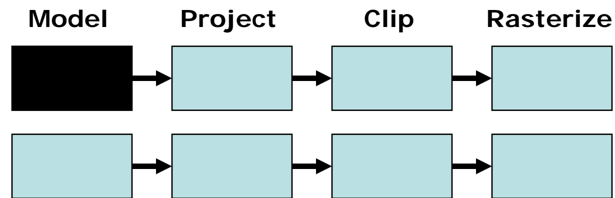


Data Flow Decomposition Example – Step 4

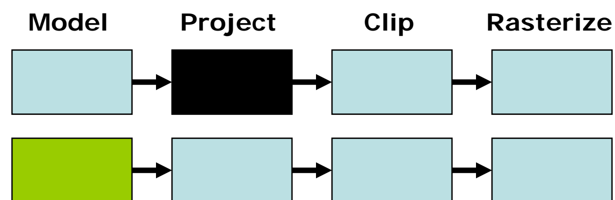


The pipeline processes 1 data set in 4 steps

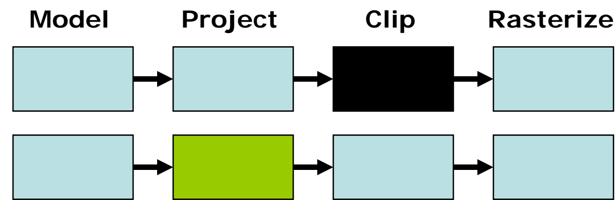
Data Flow Decomposition Example – Two Data Sets; Step 1



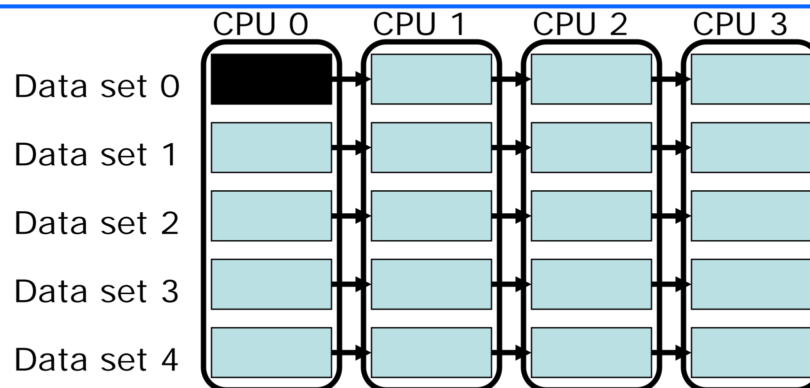
Data Flow Decomposition Example – Two Data Sets; Step 2



Data Flow Decomposition Example – Two Data Sets; Step 3

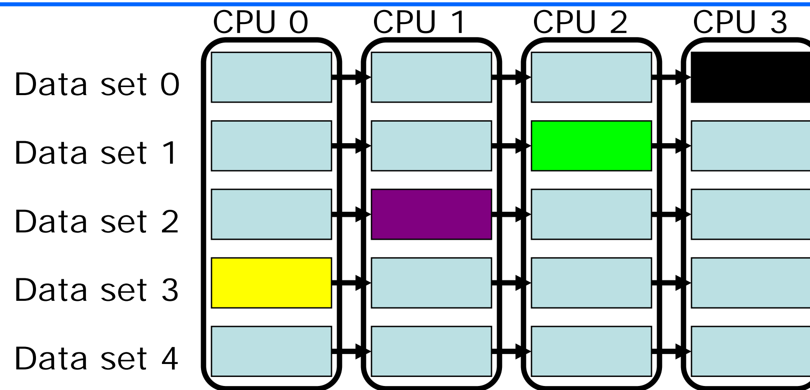


Data Flow Decomposition Example – Five Data Sets; Step 1



As the number of data sets increases, all four of the processors are busier a greater percentage of the time.

Data Flow Decomposition Example – Five Data Sets; Step 4



As the number of data sets increases, all four of the processors are busier a greater percentage of the time.

Characteristics of Producer/Consumer Problem

- The dependence created between consumer and producer can cause significant delays if this model is not implemented correctly. A performance-sensitive design seeks to understand the exact nature of the dependence and diminish the delay it imposes. It also aims to avoid situations in which consumer threads are idling while waiting for producer threads.
- The ideal case, the hand-off between producer and consumer is completely clean, as in the example of the file parser. The output is context-independent and the consumer has no need to know anything about the producer. Many times, however, the producer and consumer components do not enjoy this independence

58

Load Balancing

If the consumer is finishing up while the producer is completely done, one thread remains idle while other threads are busy working away. This issue violates an **important objective** of parallel processing, which is to **balance loads** so that all available threads are kept busy. Because of the logical relationship between these threads, it can be very difficult to keep threads equally occupied.

59

Challenges You'll Face

- Threads enable higher performance by by allowing two or more activities to occur simultaneously. However, **threads add a measure of complexity that requires thoughtful consideration to navigate correctly**. This is due to more than one activity is occurring in the program. Managing simultaneous activities and their possible interaction leads to four types of problems:

1. **Synchronization** is the process by which two or more threads coordinate their activities. For example, one thread waits for another to finish a task before continuing.
2. **Communication** refers to the bandwidth and latency issues associated with exchanging data between threads.
3. **Load balancing** refers to the distribution of work across multiple threads so that they all perform roughly the same amount of work.
4. **Scalability** is the challenge of making efficient use of a larger number of threads when software is running on more-capable systems. (E.g.,, if a program is written to make good use of four processor cores, will it scale properly when its run with eight processor cores?)

60

Parallel Programming Patterns

- For years object-oriented programmers have been using design patterns to logically design their applications.
- ‘Parallel programming is no different than object-oriented programming in that is also uses design patterns’

1. **Task-level parallelism - Task**
   - In this pattern, the problem is decomposed into a set of tasks that operate independently. It is often necessary remove dependencies between tasks or separate dependencies using replication.
2. **Divide and Conquer - Task/Data**
   - The problem is divided into a number of parallel sub-problems. Each sub-problem is solved independently.
3. **Geometric Decomposition - Data**
   - The geometric decomposition pattern is based on the parallelization of the data structures. - each thread is responsible for operating on data 'chunks'.
4. **Pipeline - Data Flow**
   - Identical to that of an assembly line. - break down the computation into a series of stages and have each thread work on a different stage simultaneously.
5. **Wavefront - Data Flow**
   - The wavefront pattern is useful when processing data elements along a diagonal in a two-dimensional grid

61

Task-level Parallelism Pattern

In many cases, the best way to achieve parallel execution is to focus directly on the tasks themselves. In this case, the task-level parallelism pattern makes the most sense. In this pattern, the problem is decomposed into a set of tasks that operate independently. It is often necessary remove dependencies between tasks or separate dependencies using replication. Problems that fit into this pattern include the so-called embarrassingly parallel problems, those where there are no dependencies between threads, and replicated data problems, those where the dependencies between threads may be removed from the individual threads.

62

Example: Wavefront Pattern

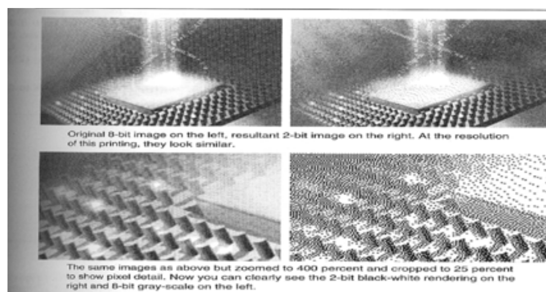
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

Wavefront Data Access Pattern

The Numbers illustrate the order in which the data elements are processed.

Example: Error Diffusion

- Image processing problem
- How to transform almost-continuous-tone digital images to lesser tone presentation devices (e.g., 8-bit grayscale to 1bit b/w)



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

Example: Error Diffusion Algorithm

The basic error diffusion algorithm does its work in a simple three step process:

1. Determine· the output value given the input value of the current pixel. This step often uses quantization, or in the binary case, thresholding. For an 8-bit grayscale image that is displayed on a one bit output device, all input values in the range [0, 127] are to be displayed as a 0 and all input values between [128, 255] are to be displayed as a 1 on the output device.

2. Once the output value is determined, the code computes the error between what should be displayed on the output device and what is actually displayed. As an example, assume that the current input pixel value is 168. Given that it is greater than our threshold value (128), we determine that the output value will be a 1. This value is stored in the output array. To compute the error, the program must normalize output first, so it is in the same scale as the input value. That is, for the purposes of computing the display error, the output pixel must be 0 if the output pixel is 0 or 255 if the output pixel is 1. In this case, the display error is the difference between the actual value that should have been displayed (168) and the output value (255), which is -87.

3. Finally, the error value is distributed on a fractional basis to the neighboring pixels in the region.

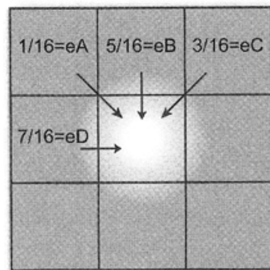
65

Example: Error Diffusion Initial Implementation

```
/*************************************
* 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;
      }
   }
}
```

66

Example: Error Diffusion Pixels Shown by Weights



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

67

Example: Error Diffusion Subdividing the Work Among Threads

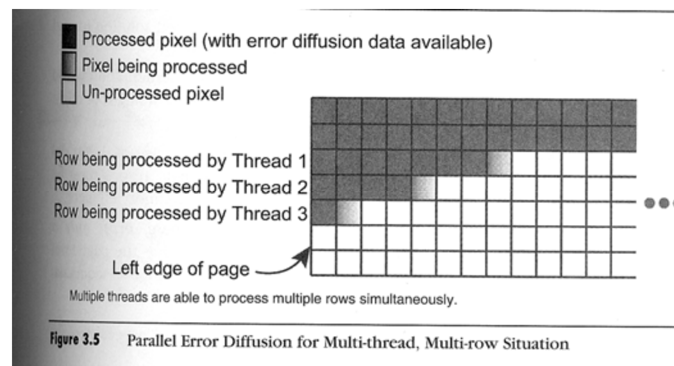


Figure 3.5 Parallel Error Diffusion for Multi-thread, Multi-row Situation

68

Example: Error Diffusion Multi-page Situation

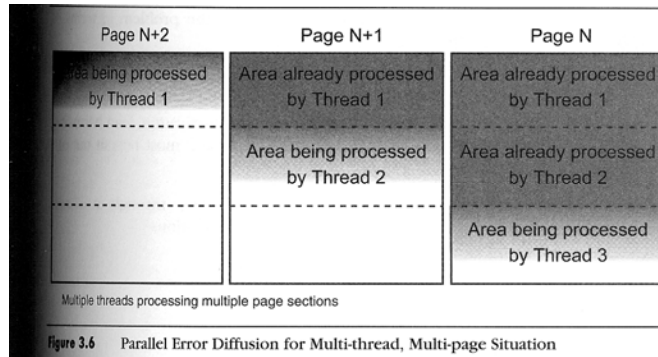


Figure 3.6 Parallel Error Diffusion for Multi-thread, Multi-page Situation

69

Chapter 3 Key Points

- Decompositions fall into one of three categories: task, data, and data flow:
  1. Task-level parallelism partitions the work between threads based on tasks
  2. Data decomposition breaks down tasks based on the data that the threads work on.
  3. Data flow decomposition breaks down the problem in terms of how data flows between the tasks.
- Most parallel programming problems fall into one of several well known patterns.
- The constraints of synchronization, communication, load balancing, and scalability must be dealt with to get the most benefit out of a parallel program.
- Many problems that appear to be serial may, through a simple transformation, be adapted to a parallel implementation.

70

References

- Richard H. Carver and Kuo-Chung Tai, *Modern Multithreading: Implementing, Testing, and Debugging Java and C++/Pthreads/ Win32 Programs*, Wiley-Interscience (2006).
- Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill (2004).
- http://publib.boulder.ibm.com/infocenter/eserver/v1r2/topic/eicay/eicay.pdf
- http://en.wikipedia.org/wiki/Virtual_machine_monitor
- Material from Intel course on Parallel Computing, 'Recognizing Potential Parallelism'
- Class Text – Chapter 3