Dr.Dobb's

Fundamental Concepts of Parallel Programming

Moving from a linear to a parallel programming model

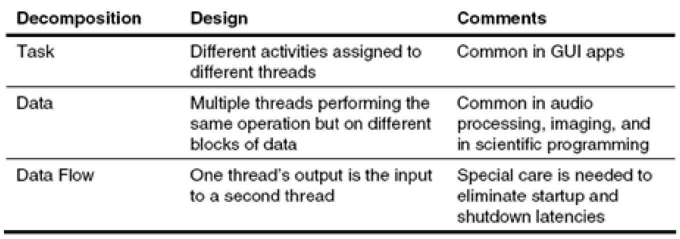
By Shameem Akhter and Jason Roberts , Dr. Dobb's Journal
Dec 31, 2008
URL:http://www.ddj.com/hpc-high-performance-computing/212002418

Shameem Akhter, a platform architect at Intel, and Jason Roberts, a senior software engineer at Intel, are the authors of Multi-Core Programming: Increasing Performance through Software Multithreading on which this article is based. Copyright (c) 2008 Intel Corporation. All rights reserved.

Developers who are unacquainted with parallel programming generally feel comfortable with traditional programming models, such as object-oriented programming. In this case, a program begins at a defined point, such as the **main()** function, and works through a series of tasks in succession. If the program relies on user interaction, the main processing instrument is a loop in which user events are handled. From each allowed event -- a button click, for example -- the program performs an established sequence of actions that ultimately ends with a wait for the next user action.

When designing such programs, developers enjoy a relatively simple programming world because only one thing is happening at any given moment. If program tasks must be scheduled in a specific way, it's because the developer imposes a certain order on the activities. At any point in the process, one step generally flows into the next, leading up to a predictable conclusion, based on predetermined parameters.

To move from this linear model to a parallel programming model, designers must rethink the idea of process flow. Rather than being constrained by a sequential execution sequence, programmers should identify those activities that can be executed in parallel. To do so, they must see their programs as a set of tasks with dependencies between them. Breaking programs down into these individual tasks and identifying dependencies is known as decomposition. A problem may be decomposed in several ways: by task, by data, or by data flow. Table 1 summarizes these forms of decomposition. As you shall see, these different forms of decomposition mirror different types of programming activities.



Decomposition	Design	Comments
Task	Different activities assigned to different threads	Common in GUI apps
Data	Multiple threads performing the same operation but on different blocks of data	Common in audio processing, imaging, and in scientific programming
Data Flow	One thread's output is the input to a second thread	Special care is needed to eliminate startup and shutdown latencies

Table 1: Major Forms of Decomposition

Task Decomposition

Decomposing a program by the functions that it performs is called task decomposition. It is one of the simplest ways to achieve parallel execution. Using this approach, individual tasks are cataloged. If two of them can run concurrently, they are scheduled to do so by the developer. 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.

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.

In programming terms, a good example of task decomposition is word processing software, such as Microsoft Word. When users open a very long document, they can begin entering text right away. While users enter text, document pagination occurs in the background, as you can readily see by the quickly increasing page count that appears in the status bar. Text entry and pagination are two separate tasks that its programmers broke out by function to run in parallel. Had programmers not designed it this way, users would be obliged to wait for the entire document to be paginated before being able to enter any text. Many of you probably recall that this wait was common on early PC word processors.

Data Decomposition

Data decomposition, also known as "data-level parallelism," breaks down tasks by the data they work on rather than by the nature of the task. Programs that are broken down via data decomposition generally have many threads performing the same work, just on different data items. For example, consider recalculating the values in a large spreadsheet. Rather than have one thread perform all the calculations, data decomposition would suggest having two threads, each performing half the calculations, or *n* threads performing 1/nth the work.

If the gardeners used the principle of data decomposition to divide their work, they would both mow half the property and then both weed half the flower beds. As in computing, determining which form of decomposition is more effective depends a lot on the constraints of the system. For example, if the area to mow is so small that it does not need two mowers, that task would be better done by just one gardener -- that is, task decomposition is the best choice -- and data decomposition could be applied to other task sequences, such as when the mowing is done and both gardeners begin weeding in parallel.

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.

Data Flow Decomposition

Many times, when decomposing a problem, the critical issue isnt 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. Here, 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.

Using the gardening example, 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.

In common programming tasks, the producer/consumer problem occurs in several typical scenarios. For example, programs that must rely on the reading of a file fit this scenario: the results of the file I/O become the input to the next step, which might be threaded. However, that step cannot begin until the reading is either complete or has progressed sufficiently for other processing to kick off. Another common programming example is parsing: an input file must be parsed, or analyzed semantically, before the back-end activities, such as code generation in a compiler, can begin. The producer/consumer problem has several interesting dimensions:

- 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.
- In the ideal scenario, 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 such a clean division of labor, and scheduling their interaction requires careful planning.
- 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.

In the next section, we look at the pipeline pattern that allows developers to solve the producer/consumer problem in a scalable fashion.

Implications of Different Decompositions

Different decompositions provide different benefits. If the goal, for example, is ease of programming and tasks can be neatly partitioned by functionality, then task decomposition is more often than not the winner. Data decomposition adds some additional code-level complexity to tasks, so it is reserved for cases where the data is easily divided and performance is important.

The most common reason for threading an application is performance. And in this case, the choice of decompositions is more difficult. In many instances, the choice is dictated by the problem domain: some tasks are much better suited to one type of decomposition. But some tasks have no clear bias. Consider for example, processing images in a video stream. In formats with no dependency between frames, you'll have a choice of decompositions. Should they choose task decomposition, in which one thread does decoding, another color balancing, and so on, or data decomposition, in which each thread does all the work on one frame and then moves on to the next? To return to the analogy of the gardeners, the decision would take this form: If two gardeners need to mow two lawns and weed two flower beds, how should they proceed? Should one gardener only mow -- that is, they choose task based decomposition -- or should both gardeners mow together then weed together?

In some cases, the answer emerges quickly; for instance, when a resource constraint exists, such as only one mower. In others where each gardener has a mower, the answer comes only through careful analysis of the constituent activities. In the case of the gardeners, task decomposition looks better because the start-up time for mowing is saved if only one mower is in use. Ultimately, you determine the right answer for your applications use of parallel programming by careful planning and testing. The empirical timing and evaluation plays a more significant role in the design choices you make in parallel programming than it does in standard single-threaded programming.

Challenges You'll Face

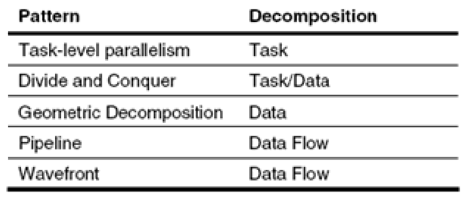
The use of threads enables you to improve performance significantly by allowing two or more activities to occur simultaneously. However, developers cannot fail to recognize that threads add a measure of complexity that requires thoughtful consideration to navigate correctly. This complexity arises from the inherent fact that more than one activity is occurring in the program. Managing simultaneous activities and their possible interaction leads you to confronting four types of problems:

- 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.
- Communication refers to the bandwidth and latency issues associated with exchanging data between threads.
- Load balancing refers to the distribution of work across multiple threads so that they all perform roughly the same amount of work.
- Scalability is the challenge of making efficient use of a larger number of threads when software is run on more-capable systems. For example, if a program is written to make good use of four processor cores, will it scale properly when run on a system with eight processor cores?

Each of these issues must be handled carefully to maximize application performance.

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 -- parallel programming problems generally fall into one of several well-known patterns. A few of the more common parallel programming patterns and their relationship to the aforementioned decompositions are in Table 2.



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

Table 2: Common Parallel Programming Patterns

In this section, we provide a brief overview of each pattern and the types of problems that each pattern may be applied to.

- **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.
- **Divide-and-Conquer Pattern**. In the divide-and-conquer pattern, the problem is divided into a number of parallel sub-problems. Each sub-problem is solved independently. Once each subproblem is solved, the results are aggregated into the final solution. Since each sub-problem can be independently solved, these sub-problems may be executed in a parallel fashion.

  The-divide-and conquer approach is widely used on sequential algorithms such as merge sort. These algorithms are very easy to parallelize. This pattern typically does a good job of load balancing and exhibits good locality; which is important for effective cache usage.

- **Geometric Decomposition Pattern**. The geometric decomposition pattern is based on the parallelization of the data structures used in the problem being solved. In geometric decomposition, each thread is responsible for operating on data "chunks". This pattern may be applied to problems such as heat flow and wave propagation.
- **Pipeline Pattern**. The idea behind the pipeline pattern is identical to that of an assembly line. The way to find concurrency here is to break down the computation into a series of stages and have each thread work on a different stage simultaneously.
- **Wavefront Pattern**. The wavefront pattern is useful when processing data elements along a diagonal in a

two-dimensional grid. This is shown in Figure 1.

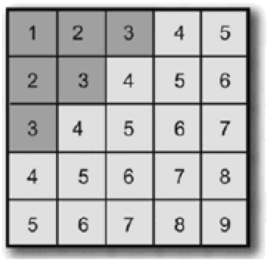


Figure 1: Wavefront Data Access Pattern

The numbers in Figure 1 illustrate the order in which the data elements are processed. For example, elements in the diagonal that contains the number **3** are dependent on data elements **1** and **2** being processed previously. The shaded data elements in Figure 1 indicate data that has already been processed. In this pattern, it is critical to minimize the idle time spent by each thread. Load balancing is the key to success with this pattern.

For a more extensive and thorough look at parallel programming design patterns, see Patterns for Parallel Programming, by Timothy Mattson et al.

A Motivating Problem: Error Diffusion

To see how you might apply the aforementioned methods to a practical computing problem, consider the error diffusion algorithm that is used in many computer graphics and image processing programs. Originally proposed by Floyd and Steinberg (Floyd 1975), error diffusion is a technique for displaying continuous-tone digital images on devices that have limited color (tone) range. Printing an 8-bit grayscale image to a black-and-white printer is problematic. The printer, being a bi-level device, cannot print the 8-bit image natively. It must simulate multiple shades of gray by using an approximation technique. An example of an image before and after the error diffusion process is shown in Figure 2. The original image, composed of 8-bit grayscale pixels, is shown on the left, and the result of the image that has been processed using the error diffusion algorithm is shown on the right. The output image is composed of pixels of only two colors: black and white.

[Click image to view at full size]

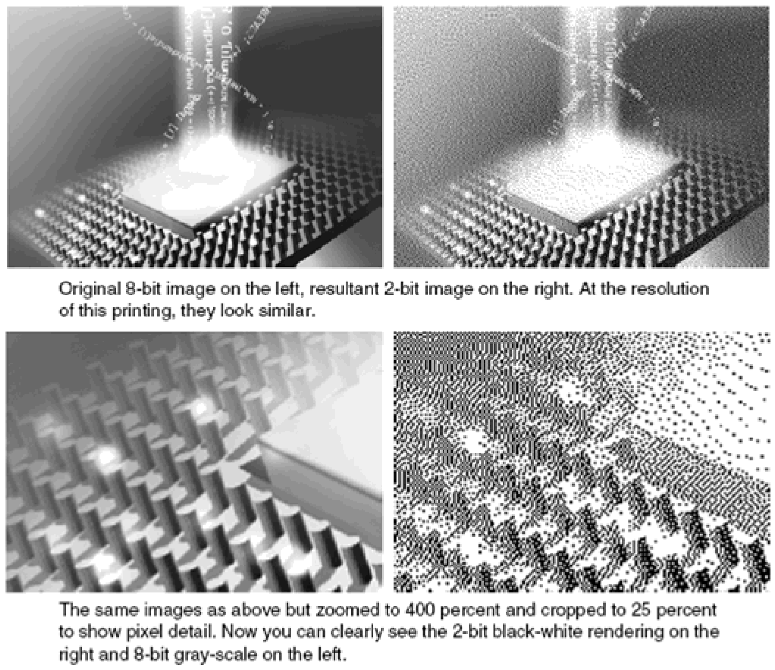

Original 8-bit image on the left, resultant 2-bit image on the right. At the resolution of this printing, they look similar.
The same images as above but zoomed to 400 percent and cropped to 25 percent to show pixel detail. Now you can clearly see the 2-bit black-white rendering on the right and 8-bit gray-scale on the left.

Figure 2: Error Diffusion Algorithm Output

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 1-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, as in Figure 3.

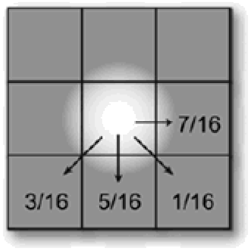


Figure 3: Distributing Error Values to Neighboring Pixels

This example uses the Floyd-Steinberg error weights to propagate errors to neighboring pixels. 7/16ths of the error is computed and added to the pixel to the right of the current pixel that is being processed. 5/16ths of the error is added to the pixel in the next row, directly below the current pixel. The remaining errors propagate in a similar fashion. While you can use other error weighting schemes, all error diffusion algorithms follow this general method.

The three-step process is applied to all pixels in the image. Listing 1 shows a simple C implementation of the error diffusion algorithm, using Floyd-Steinberg error weights.

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

Listing 1: C-language Implementation of the Error Diffusion Algorithm

Analysis of the Error Diffusion Algorithm

At first glance, one might think that the error diffusion algorithm is an inherently serial process. The conventional approach distributes errors to neighboring pixels as they are computed. As a result, the previous pixel's error must be known in order to compute the value of the next pixel. This interdependency implies that the code can only process one pixel at a time. It's not that difficult, however, to approach this problem in a way that is more suitable to a multi-threaded approach.

An Alternate Approach: Parallel Error Diffusion

To transform the conventional error diffusion algorithm into an approach that is more conducive to a parallel solution, consider the different decomposition that were covered previously in this article. Which would be appropriate in this case? As a hint, consider Figure 4, which revisits the error distribution illustrated in Figure 3, from a slightly different perspective.

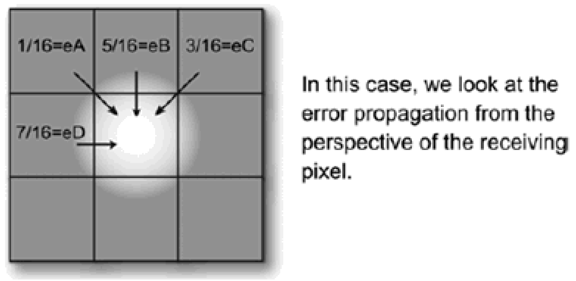

1/16=eA
5/16=eB
3/16=eC
7/16=eD
In this case, we look at the error propagation from the perspective of the receiving pixel.

Figure 4: Error-Diffusion Error Computation from the Receiving Pixel's Perspective

Given that a pixel may not be processed until its spatial predecessors have been processed, the problem appears to lend itself to an approach where we have a producer -- or in this case, multiple producers -- producing data (error values) which a consumer (the current pixel) will use to compute the proper output pixel. The flow of error data to the current pixel is critical. Therefore, the problem seems to break down into a data-flow decomposition.

Now that we identified the approach, the next step is to determine the best pattern that can be applied to this particular problem. Each independent thread of execution should process an equal amount of work (load balancing). How should the work be partitioned? One way, based on the algorithm presented in the previous section, would be to have a thread that processed the even pixels in a given row, and another thread that processed the odd pixels in the same row. This approach is ineffective however; each thread will be blocked waiting for the other to complete, and the performance could be worse than in the sequential case.

To effectively subdivide the work among threads, we need a way to reduce (or ideally eliminate) the dependency between pixels. Figure 4 illustrates an important point that's not obvious in Figure 3 -- that in order for a pixel to be able to be processed, it must have three error values (labeled **eA**, **eB**, and **eC1** in Figure 3) from the previous row, and one error value from the pixel immediately to the left on the current row. Thus, once these pixels are processed, the current pixel may complete its processing. This ordering suggests an implementation where each thread processes a row of data. Once a row has completed processing of the first few pixels, the thread responsible for the next row may begin its processing. Figure 5 shows this sequence.

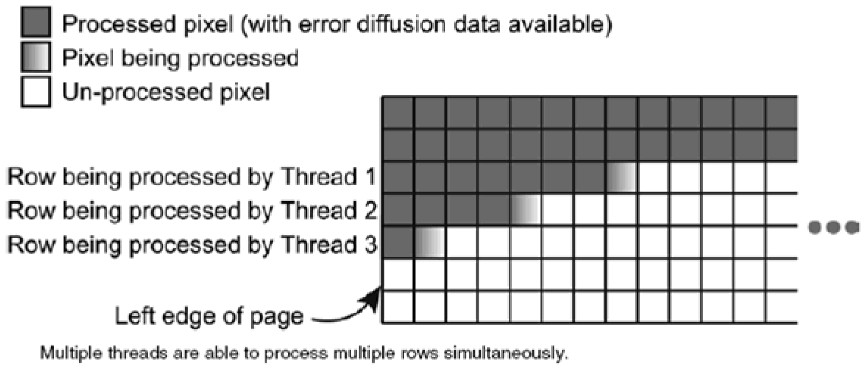

Processed pixel (with error diffusion data available)
Pixel being processed
Un-processed pixel
Row being processed by Thread 1
Row being processed by Thread 2
Row being processed by Thread 3
Left edge of page
Multiple threads are able to process multiple rows simultaneously.

Figure 5: Parallel Error Diffusion for Multi-thread, Multi-row Situation

Notice that a small latency occurs at the start of each row. This latency is due to the fact that the previous row's error data must be calculated before the current row can be processed. These types of latency are generally unavoidable in producer-consumer implementations; however, you can minimize the impact of the latency as illustrated here. The trick is to derive the proper workload partitioning so that each thread of execution works as efficiently as possible. In this case, you incur a two-pixel latency before processing of the next thread can begin. An 8.5x11-inch page, assuming 1,200 dots per inch (dpi), would have 10,200 pixels per row. The two-pixel latency is insignificant here.

The sequence in Figure 5 illustrates the data flow common to the wavefront pattern.

Other Alternatives

In the previous section, we proposed a method of error diffusion where each thread processed a row of data at a time. However, one might consider subdividing the work at a higher level of granularity. Instinctively, when partitioning work between threads, one tends to look for independent tasks. The simplest way of parallelizing this problem would be to process each page separately. Generally speaking, each page would be an independent data set, and thus, it would not have any interdependencies. So why did we propose a row-based solution instead of processing individual pages? The three key reasons are:

- An image may span multiple pages. This implementation would impose a restriction of one image per page, which might or might not be suitable for the given application.
- Increased memory usage. An 8.5x11-inch page at 1,200 dpi consumes 131 megabytes of RAM. Intermediate results must be saved; therefore, this approach would be less memory efficient.
- An application might, in a common use-case, print only a single page at a time. Subdividing the problem at the page level would offer no performance improvement from the sequential case.

A hybrid approach would be to subdivide the pages and process regions of a page in a thread, as in Figure 6.

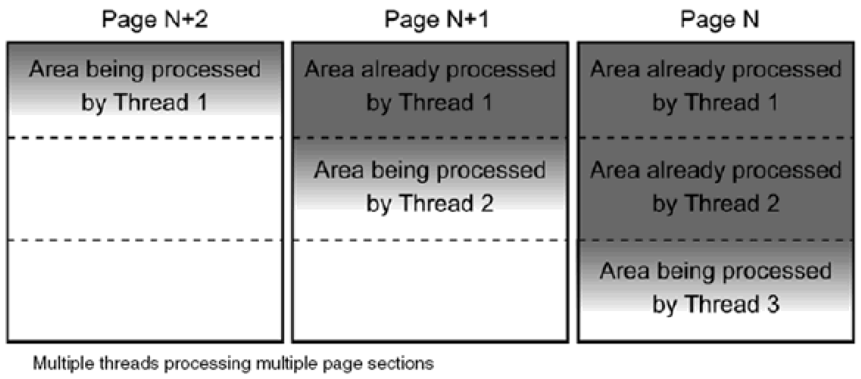

Page N+2
Page N+1
Page N
Area being processed by Thread 1
Area already processed by Thread 1
Area being processed by Thread 2
Area already processed by Thread 1
Area already processed by Thread 2
Area being processed by Thread 3
Multiple threads processing multiple page sections

Figure 6: Parallel Error Diffusion for Multi-thread, Multi-page Situation

Note that each thread must work on sections from different page. This increases the startup latency involved before the threads can begin work. In Figure 6, Thread 2 incurs a 1/3 page startup latency before it can begin to process data, while Thread 3 incurs a 2/3 page startup latency. While somewhat improved, the hybrid approach suffers from similar limitations as the page-based partitioning scheme described above. To avoid these limitations, you should focus on the row-based error diffusion implementation illustrated in Figure 5.

Key Points

This article explored different types of computer architectures and how they enable parallel software development. The key points to keep in mind when developing solutions for parallel computing architectures are:

- Decompositions fall into one of three categories: task, data, and data flow.
- Task-level parallelism partitions the work between threads based on tasks.
- Data decomposition breaks down tasks based on the data that the threads work on.
- 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.