LabVIEW™ and Hyper-Threading

Introduction

Hyper-Threading is an advanced feature of high-end versions of the Intel Pentium 4 and later. A Hyper-Threaded computer has a single processor but acts as a computer with a multiprocessor. When you launch the Windows Task Manager on a Hyper-Threaded computer, and click the **Performance** tab, the Windows Task Manager displays the usage history for two CPUs.

LabVIEW and Hyper-Threading

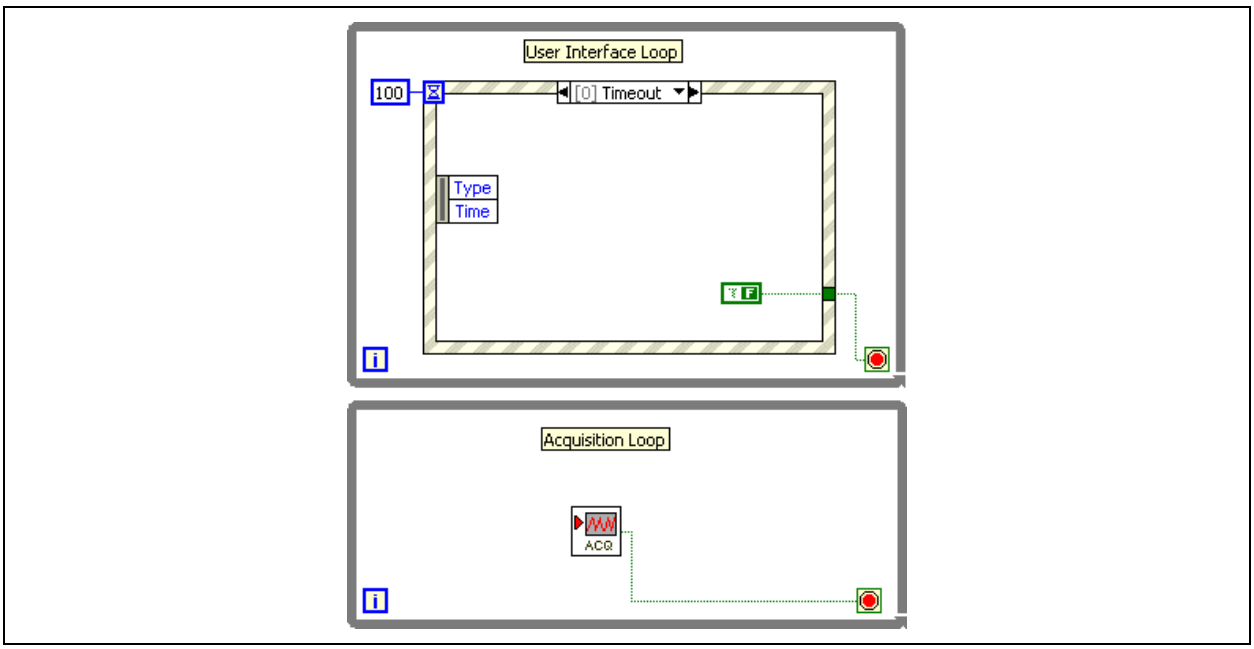
A Hyper-Threaded processor acts like multiple processors embedded on the same microchip. Some of the resources on the chip are duplicated, such as the register set. Other resources are shared, such as the execution units and the cache. Some resources, such as the buffers that store micro operations, are partitioned, with each logical processor receiving a portion.

Optimizing an application to take advantage of Hyper-Threading is similar to optimizing an application for a multiprocessor system, but there are some differences. For example, a Hyper-Threaded computer shares the execution units, and a dual-processor computer contains two complete sets of execution units. Therefore, any application that is limited by floating-point execution units performs better on the multiprocessor computer because you do not have to share the execution units. The same principle applies with cache contention. If two threads try to access the cache, the performance is better on a multiprocessor computer, where each processor has its own full-size cache.

LabVIEW Execution Overview

The LabVIEW execution system is already built for multiprocessing. In text-based programming languages, to make an application multithreaded, you have to create multiple threads and write code to communicate among those threads. LabVIEW, however, can recognize opportunities for multithreading in VIs, and the execution system handles multithreading communications for you.

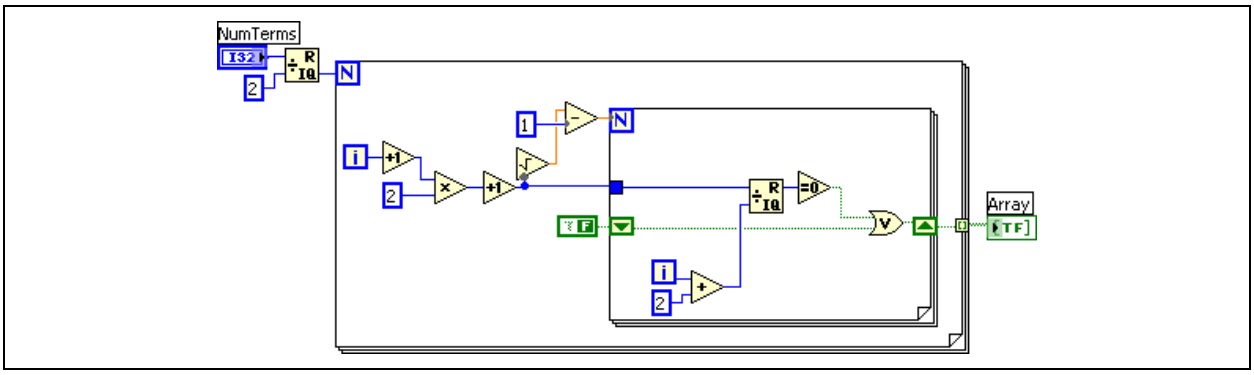
The following example takes advantage of the LabVIEW multithreaded execution system.



In this VI, LabVIEW recognizes that it can execute the two loops independently, and in a multiprocessing or Hyper-Threaded environment, often simultaneously.

Primes Parallelism Example

The following example calculates prime numbers greater than two.

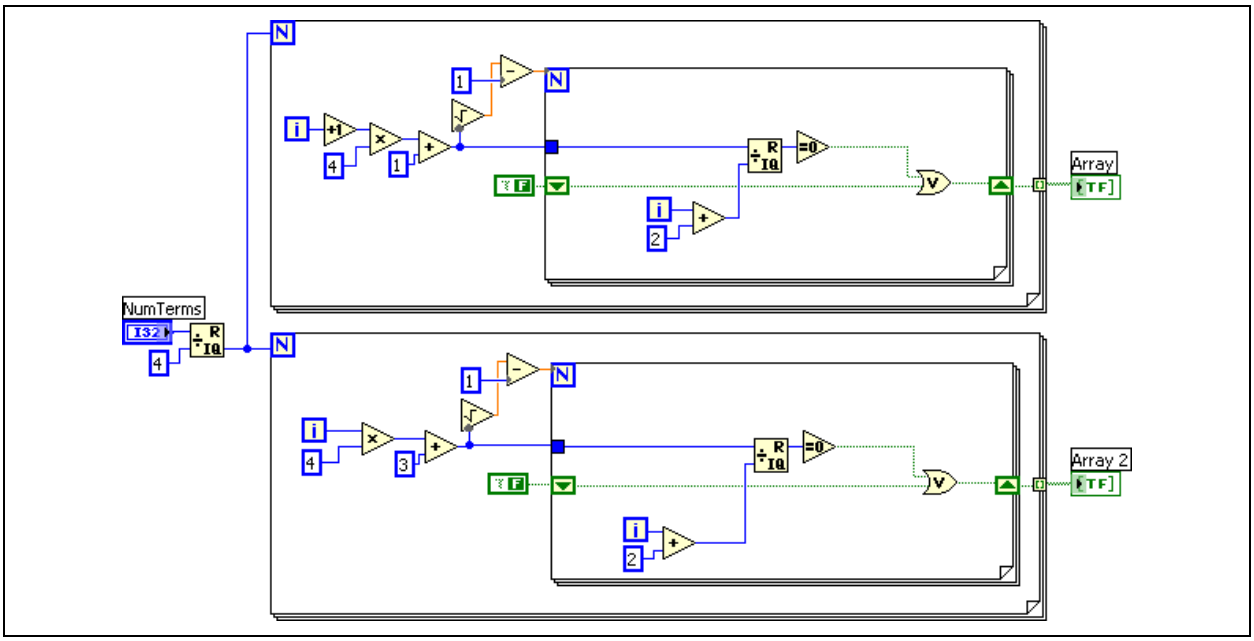


The block diagram evaluates all the odd numbers between three and **Num Terms** and determines if they are prime. The inner For Loop returns TRUE if any number divides the term with a zero remainder.

The inner For Loop is computationally intensive because it does not include any I/O or wait functions. The architecture of this VI prevents LabVIEW from taking advantage of any parallelism. There is a mandatory order for every operation in the loop. This order is enforced by dataflow, and there is no other execution order possible because every operation must wait for its inputs.

You can introduce parallelism into this VI. Parallelism requires that no single loop iteration depends on any other loop iteration. Once you meet this condition, you can distribute loop iterations between two loops. However, one LabVIEW constraint is that no iteration of a loop can begin before the previous iteration finishes. You can split the process into two loops after you determine that the constraint is not necessary.

In the following illustration, the primes parallelism example splits the process into two loops. The top loop evaluates half of the odd numbers, and the bottom loop evaluates the other half. On a multiprocessor computer, the two-loop version is more efficient because LabVIEW can simultaneously execute code from both loops. Notice that the output of this version of the VI has two arrays instead of one as in the previous example. You can write a subVI to combine these arrays and because the calculations consume most of the execution time, the additional VI at the end of the process becomes negligible.



Notice that these two example VIs do not include code for explicit thread management. The LabVIEW dataflow programming paradigm allows the LabVIEW execution system to run the two loops in different threads. In many text-based programming languages, you must explicitly create and handle threads.

LabVIEW Threading Model

LabVIEW 5.0 was the first multithreaded version of LabVIEW. LabVIEW 5.0 separated the user interface thread from the execution threads. Block diagrams executed in one, or possibly more than one, thread, and front panels updated in another thread. The operating system preemptively multitasked between threads by granting processor time to the execution thread, then to the user interface thread, and so on.

Before LabVIEW 7.0, LabVIEW allocated only one thread for a single priority and execution system by default, which meant that all activity in all VIs in the same priority and execution system had to wait together. LabVIEW 7.0 increased the default number of threads allocated for a single priority and execution system to four, which means that if one thread executes an instruction that causes a wait, other sections of the block diagram can continue to execute.

In addition to the preemptive multitasking of the operating system, LabVIEW employs a cooperative multitasking system. During compilation, LabVIEW analyzes VIs to find groups of nodes that can execute together in what are called *clumps*. Each priority and execution system combination has a run queue data structure that retains which clumps can run together. When the operating system activates a thread, the operating system retrieves and executes a clump from the run queue. When the operating system finishes executing, it stores additional clumps that meet the input conditions on the run queue, which allows the block diagram to execute in any of the available four execution threads. If the block diagram includes enough parallelism, it can simultaneously execute in all four threads.

LabVIEW does not permanently assign clumps of code to a particular thread. LabVIEW can execute a clump using a different thread the next time you run the VI.

Programming for Hyper-Threaded or Multiprocessor Systems

Optimizing the performance of an application for a Hyper-Threaded computer is nearly identical to doing so for a multiprocessor computer. However, differences exist because a Hyper-Threaded computer shares some resources between the two logical processors, such as the cache and execution units. If you think a shared resource on a Hyper-Threaded computer limits an application, test the application with an advanced sampling performance analyzer, such as the Intel VTune.

The following code example shows how to write a multithreaded program in a text-based programming language by rewriting the primes parallelism example from the previous section of this document in C++. This example demonstrates the kind of effort required to write thread-handling code and illustrates the special coding necessary to protect data that threads share.

Primes Example in C++

The following sample code was written and tested in Microsoft Visual C++ 6.0. The single-threaded primes parallelism example, following the same algorithm as in the previous LabVIEW example in this document, would look something like the following:

```
// Single-threaded version.
void __cdecl CalculatePrimes(int numTerms) {
       bool *resultArray = new bool[numTerms/2];
       for (int i=0; i<numTerms/2; ++i) {
              int primeToTest = i*2+3;// Start with 3, then add 2 each iteration.
              bool isPrime = true;
              for (int j=2; j<=sqrt(primeToTest); ++j) {
                     if (primeToTest % j == 0) {
                           isPrime = false;
                           break;
                           }
                     }
              resultArray[i] = isPrime;
              }
       ReportResultsCallback(numTerms, resultArray);
       delete [] resultArray;
       }
```

No parallelism exists in this single-threaded version of the code, which would consume 100 percent of one virtual processor without using the other processor. To use any of the bandwidth on the other processor, the application must initiate additional threads and distribute the work.

The following code is an example of a preliminary multithreaded version of the primes parallelism example:

```
struct ThreadInfo {
       int numTerms;
       bool done;
       bool *resultArray;
       };
static void __cdecl CalculatePrimesThread1(void*);
static void __cdecl CalculatePrimesThread2(void*);
void __cdecl CalculatePrimesMultiThreaded(int numTerms) {
       // Initialize the information to pass to the threads.
       ThreadInfo threadInfo1, threadInfo2;
       threadInfo1.done = threadInfo2.done = false;
       threadInfo1.numTerms = threadInfo2.numTerms = numTerms;
       threadInfo1.resultArray = threadInfo2.resultArray = NULL;
       // Start two threads
       _beginthread(CalculatePrimesThread1, NULL, &threadInfo1);
       _beginthread(CalculatePrimesThread2, NULL, &threadInfo2);
       // Wait for the threads to finish executing.
       while (!threadInfo1.done || !threadInfo2.done)
              Sleep(5);
       // Collate the results.
       bool *resultArray = new bool[numTerms/2];
       for (int i=0; i<numTerms/4; ++i) {
              resultArray[2*i] = threadInfo1.resultArray[i];
              resultArray[2*i+1] = threadInfo2.resultArray[i];
              }
       ReportResultsCallback(numTerms, resultArray);
       delete [] resultArray;
       }
static void __cdecl CalculatePrimesThread1(void *ptr) {
       ThreadInfo* tiPtr = (ThreadInfo*)ptr;
       tiPtr->resultArray = new bool[tiPtr->numTerms/4];
       for (int i=0; i<tiPtr->numTerms/4; ++i) {
              int primeToTest = (i+1)*4+1;
              bool isPrime = true;
              for (int j=2; j<=sqrt(primeToTest); ++j) {
                     if (primeToTest % j == 0) {
                           isPrime = false;
                           break;
                           }
                     }
              tiPtr->resultArray[i] = isPrime;
              }
       tiPtr->done=true;
       }
static void __cdecl CalculatePrimesThread2(void *ptr) {
       ThreadInfo* tiPtr = (ThreadInfo*)ptr;
       tiPtr->resultArray = new bool[tiPtr->numTerms/4];
       for (int i=0; i<tiPtr->numTerms/4; ++i) {
              int primeToTest = (i+1)*4+3;
              bool isPrime = true;
```

```
            for (int j=2; j<=sqrt(primeToTest); ++j) {
                  if (primeToTest % j == 0) {
                        isPrime = false;
                        break;
                        }
                  }
            tiPtr->resultArray[i] = isPrime;
            }
      tiPtr->done=true;
      }
```

In this example, the `CalculatePrimesMultiThreaded()` function creates two threads using the `_beginthread()` function. The first thread calls the `CalculatePrimesThread1()` function, which tests half of the odd numbers. The second thread calls the `CalculatePrimesThread2` function and tests the other half of the odd numbers.

The original thread, which is still running the `CalculatePrimesMultiThreaded()` function, has to wait for the two worker threads to finish by creating the data structure ThreadInfo for each thread and passing it into the `_beginthread()` function. When a thread finishes executing, it writes `true` into `ThreadInfo::done`. The original thread must continually poll `ThreadInfo::done` until the original thread reads a `true` value for each computation thread, at which time it is safe for the original thread to access the results of the calculation. The program then collates the values into a single array, so they are identical with the output of the single-threaded version of this example.

Note The previous step was not shown in the LabVIEW example, but is a trivial task to accomplish.

When you write multithreading code in a text-based programming language like C++, you must protect any data locations multiple threads can access. If you do not protect the data locations, the consequences are extremely unpredictable and often difficult to reproduce and locate. In any circumstance where the assignment of `tiPtr->done` to `true` is not atomic, you must use a mutex to protect both the assignment and the access.

You can improve the previous example, however. In particular, there is no reason to initiate two additional threads and leave one idle. Because this example contains code for a computer with two virtual processors, you can initiate one additional thread instead and use the original to perform half the computation. Also, you could pass both threads through the same function instead of writing what is basically the same function twice. You need to pass in an additional parameter to the function to indicate in which thread it can run.

Note You can make the same optimization to the LabVIEW example by creating a reentrant subVI that contains the For Loop. The calling VI would have two instances of the subVI instead of two For Loops. Refer to Application Note 114, *Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability* for more information about reentrant VIs.

The following code is a more efficient multithreaded application:

```
struct ThreadInfo2 {
      int threadNum;
      int numTerms;
      bool done;
      bool *resultArray;
      };
static void __cdecl CalculatePrimesThread(void*);
void __cdecl CalculatePrimesMultiThreadedSingleFunc(int numTerms) {
      // Initialize the information to pass to the threads.
      ThreadInfo2 threadInfo1, threadInfo2;
      threadInfo1.done = threadInfo2.done = false;
```

```
        threadInfo1.numTerms = threadInfo2.numTerms = numTerms;
        threadInfo1.resultArray = threadInfo2.resultArray = NULL;
        threadInfo1.threadNum = 1;
        threadInfo2.threadNum = 2;

        // Start a thread
        _beginthread(CalculatePrimesThread, NULL, &threadInfo1);

        // Use this thread for the other branch instead of spawning another thread.
        CalculatePrimesThread(&threadInfo2);

        // Maybe this thread finished first.  If so, wait for the other.
        while (!threadInfo1.done)
               Sleep(5);

        // Collate the results.
        bool *resultArray = new bool[numTerms/2];
        for (int i=0; i<numTerms/4; ++i) {
               resultArray[2*i] = threadInfo1.resultArray[i];
               resultArray[2*i+1] = threadInfo2.resultArray[i];
               }
        ReportResultsCallback(numTerms, resultArray);
        delete [] resultArray;
        }
static void __cdecl CalculatePrimesThread(void *ptr) {
        ThreadInfo2* tiPtr = (ThreadInfo2*)ptr;
        int offset = (tiPtr->threadNum==1) ? 1 : 3;
        tiPtr->resultArray = new bool[tiPtr->numTerms/4];
        for (int i=0; i<tiPtr->numTerms/4; ++i) {
               int primeToTest = (i+1)*4+offset;
               bool isPrime = true;
               for (int j=2; j<=sqrt(primeToTest); ++j) {
                      if (primeToTest % j == 0) {
                             isPrime = false;
                             break;
                             }
                      }
               tiPtr->resultArray[i] = isPrime;
               }
        tiPtr->done=true;
        }
```

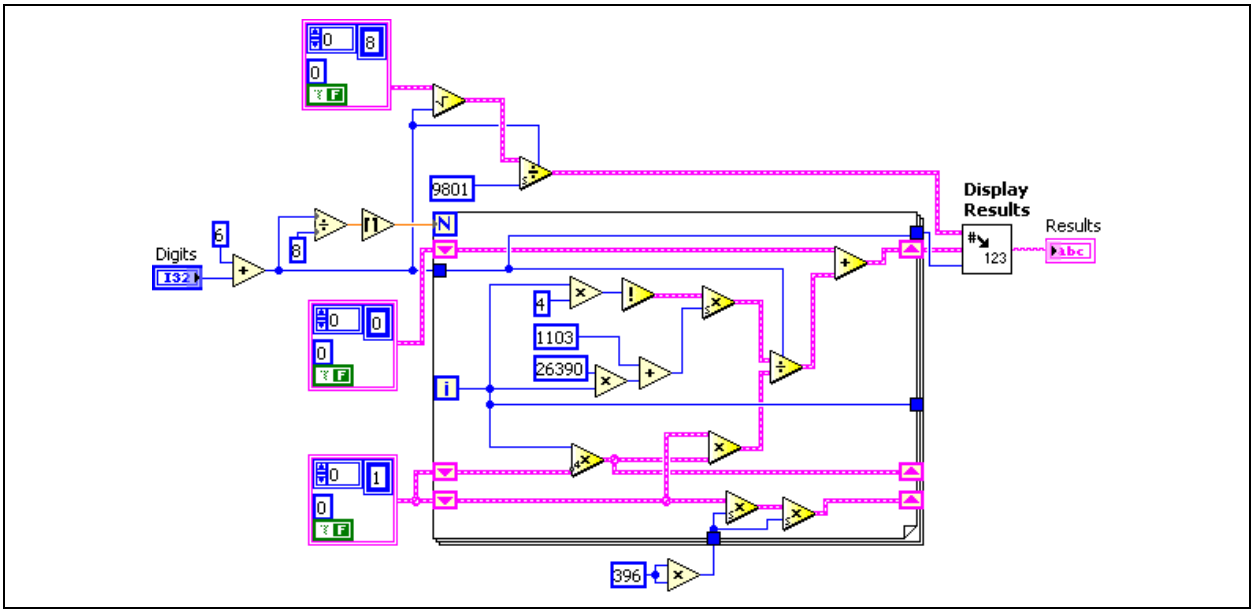
This version of the example code is more efficient because you initiate fewer threads and spend less time waiting for the worker threads to complete. Additionally, because the code performs the computation inside a single function, there is less duplicated code, which makes the application easier to maintain. However, a large portion of code handles thread management, which is inherently unsafe and difficult to test and debug.

A More Complicated Example in LabVIEW

The primes parallelism example is a simple example that demonstrates many of the concepts involved in multiprocessing. Most real-world applications are not so simple. Even if you needed to write a prime number generator, the algorithm the examples in this document use is not an efficient method.

The following section describes another computationally intensive algorithm that can benefit from the LabVIEW multithreaded execution system.

The block diagram in the following illustration calculates pi to any number of digits you specify.



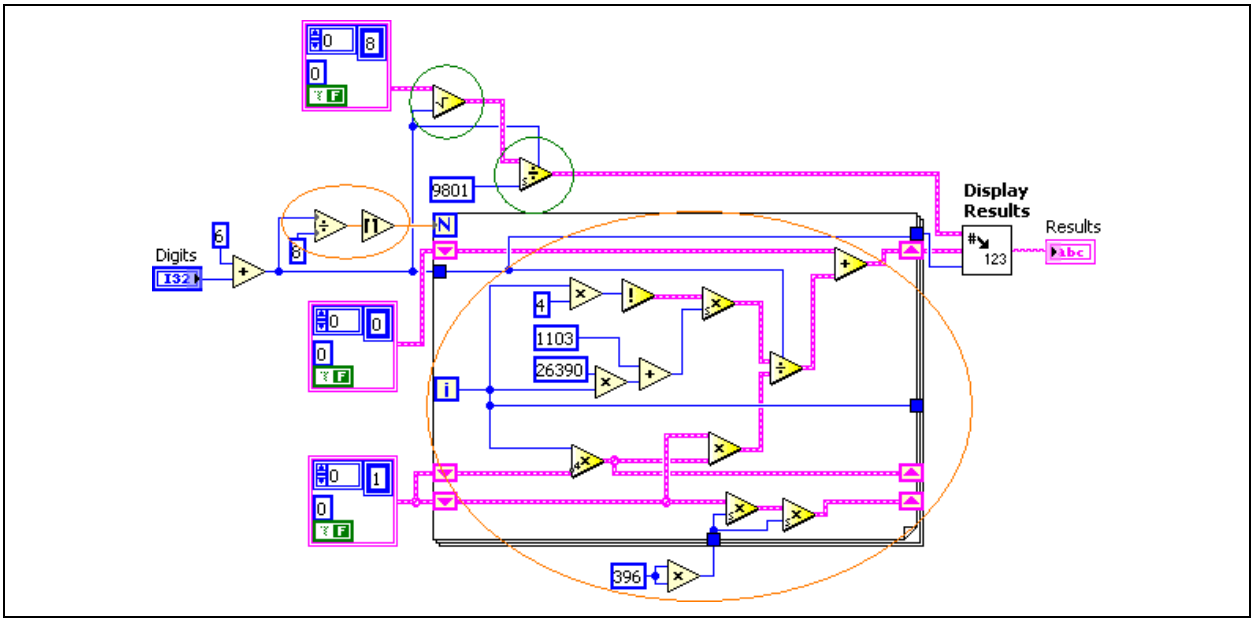
The pink wires are clusters that serve as arbitrary-precision numeric values. If you want to compute pi to 1,000 digits, you need more than an extended-precision, floating-point value. The operations that look like LabVIEW functions but operate on the pink clusters are VIs that perform computations on the arbitrary-precision numbers.

This VI also is computationally intensive. It computes pi based on the following formula:

$$\frac{1}{\pi} = \frac{\sqrt{8}}{9801} \sum_{i=0}^{\infty} \frac{(4i)![1103 + 26390i]}{(i!)^4 396^{4i}}$$

Given the numerical complexity of this equation, it would be difficult in most text-based programming languages to write a program that uses both logical processors on a Hyper-Threaded computer. LabVIEW, however, can analyze the previous equation and recognize it as an opportunity for multiprocessing.

When LabVIEW analyzes the following block diagram, it identifies some inherent parallelism. Notice that no dataflow dependency exists between the sections highlighted in red and those in green.



At first, it appears that little parallelism exists on this block diagram. It seems that only two operations can execute in parallel with the rest of the VI, and those two operations run only once. Actually, the square root is time-consuming, and this VI spends about half the total execution time running the square root operation in parallel with the For Loop. If you split the VI into two unrelated loops, LabVIEW can recognize the parallelism and handle the threads in parallel using both logical processors, which results in a large performance gain. This solution works on a Hyper-Threaded computer similarly to the way it works on a multiprocessor computer.

It can be difficult to analyze an application to identify parallelism. If you write an application in a text-based programming language such as C++, you must identify opportunities for parallelism and explicitly create and run threads to take advantage of Hyper-Threaded computers. The advantage to using LabVIEW is that in many instances, like this one, LabVIEW automatically identifies parallelism so the execution system can use both logical processors. In certain applications, it is beneficial to use an algorithm that creates opportunities for parallelism so you can take advantage of the multithreading capabilities of LabVIEW.

In addition to the operations highlighted in red and green on this block diagram, more parallelism exists in the For Loop. LabVIEW executes those operations in parallel, which saves time on a Hyper-Threaded computer. However, because there are dataflow dependencies later in the data stream, you gain more benefit when LabVIEW separates the highlighted sections.

The following table lists the execution time the VI needs to calculate pi to 1,500 digits without any performance optimization:

Hyper-Threading	27.9 s
No Hyper-Threading	25.3 s

Notice that the Hyper-Threaded computer executes slower. Because both logical processors on a Hyper-Threaded computer share a cache, it is easier to overwhelm cache lines and cause thrashing. LabVIEW stores information necessary for debugging in an area in memory that all execution threads need to access, both to write to and to read from. In some situations, two threads execute simultaneously on different logical processors, and one thread needs to read from the debug information at the same time that another thread is writing to the debug information. If two threads perform operations to bytes in the same cache line on a Hyper-Threaded computer, the loss in performance can be substantial.

You can improve the performance of the VI by disabling debugging. You can disable debugging for a performance-critical section of the VI and then enable it again if you need to debug that section of the VI.

The following table lists the execution time the VI needs to calculate pi to 1,500 digits after disabling debugging for the entire VI:

Hyper-Threading	21.6 s
No Hyper-Threading	21.5 s

Notice that the Hyper-Threaded computer has almost the same performance as the computer that is not Hyper-Threaded. The overhead of splitting the work does not seem to improve performance even though LabVIEW executes in parallel.

The following table displays the results of running the primes parallelism example in LabVIEW 7.1. The VI found all the prime numbers in the first 400,000 natural numbers on a 3.06 GHz Pentium 4 Windows XP computer.

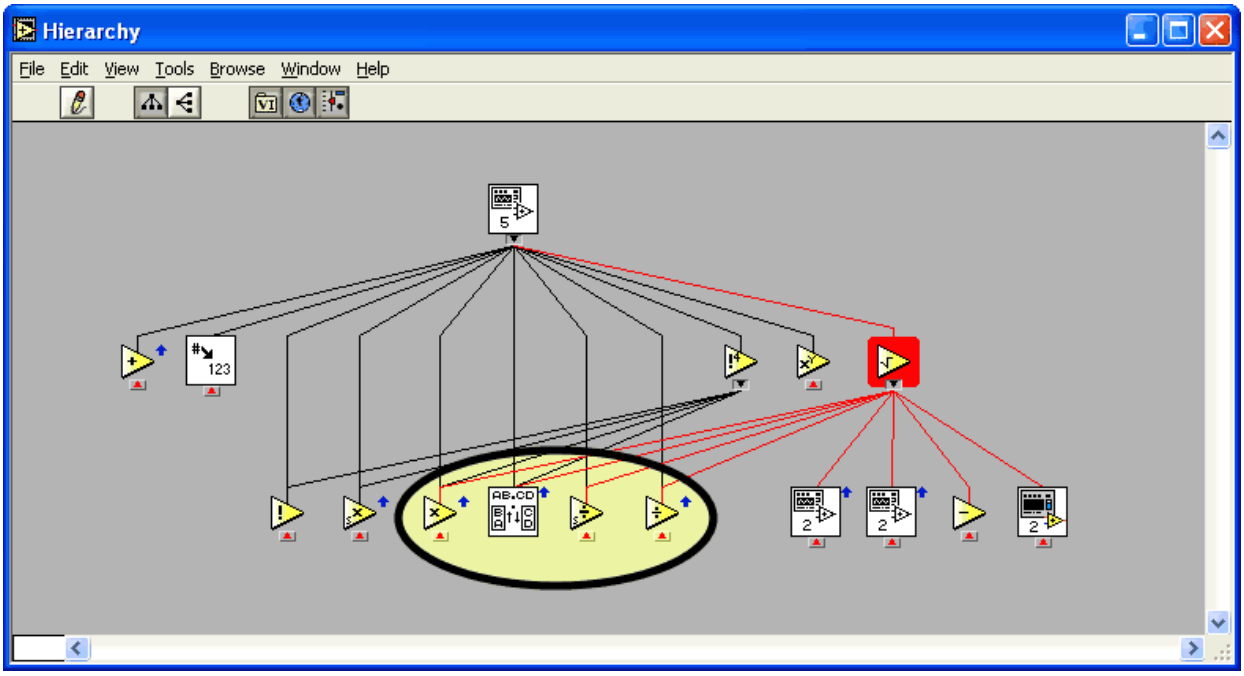
Computer Configuration	Time in Seconds
Hyper-Threading without Debugging	1.97
Hyper-Threading with Debugging	10.47

The difference with debugging enabled is significant, but this difference occurs only when two parallel sections execute in the same VI. If you wrote the application with more than just one VI or if the VI were complex enough to handle most of the debugging information for separate pieces in different cache lines, the difference with debugging enabled would not be as significant.

Another optimization that can improve performance on a Hyper-Threaded computer is to make some subVIs reentrant. Multiple threads can make simultaneous calls to the same subVI if the subVI is reentrant. It is important to understand reentrant execution in the LabVIEW execution system when you optimize a VI for a multiprocessor or Hyper-Threaded computer. Incorrectly marking a VI as reentrant could cause unnecessary delays, especially in a multiprocessor environment. Refer to Application Note 114, *Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability*, for more information about reentrant VIs.

To determine which subVIs can be reentrant, find VIs that both branches of parallel execution call. Select **Browse»Show VI Hierarchy** to display the hierarchy window for the main VI to find the VIs and functions both execution branches use.

In the previous example, the Square Root function and the For Loop operate in parallel.



The top-level VI calls the Square Root, Add, and Multiply functions and the Multiply Scalar and Divide Scalar VIs in a For Loop. These are two ideal sections to make reentrant because LabVIEW calls these sections from different threads. The following table lists the execution time the VI needs to calculate pi to 1,500 digits after marking the Add and Multiply functions and the Multiply Scalar and Divide Scalar VIs as reentrant:

Hyper-Threading	20.5 s
No Hyper-Threading	21.9 s

Marking the VIs reentrant made the VI execute in a slightly faster time on the Hyper-Threading computer. The execution time for the computer without Hyper-Threading was slightly slower after making the four VIs reentrant because LabVIEW creates additional dataspace when it calls reentrant VIs.

You can continue to optimize an application by using the VI Profiler and then specify some VIs to execute at subroutine priority. You also can mark more VIs as reentrant. The following table lists the execution time the VI needs to calculate pi to 1,500 digits after several rounds of optimization:

Hyper-Threading	18.0 s
No Hyper-Threading	20.4 s

These results display a 10 percent performance improvement on the Hyper-Threaded computer. Notice that you did not have to change any of the code of the VI to take advantage of the improvement. You only had to disable debugging, make some VIs reentrant, and execute several VIs at subroutine priority. In total, these changes make about a 35 percent performance improvement on the Hyper-Threaded computer.

LabVIEW 7.1 Improves Performance on Hyper-Threaded Computers

LabVIEW 7.1 is optimized to share the cache between the two logical processors on a Hyper-Threaded computer. When Hyper-Threading was introduced, multithreaded applications, including LabVIEW, had to adjust to take advantage of the new performance potential. In some cases, applications written before Hyper-Threading was available suffered in performance on the new processors, especially when two threads had to share a single cache. As a result, in many cases, LabVIEW 7.0 and earlier versions perform slower on a Hyper-Threaded computer. If you cannot upgrade to LabVIEW 7.1 but need better performance from an application on a Hyper-Threaded computer, disable Hyper-Threading or set the processor affinity of the process to a single logical CPU.

The following table displays the results in seconds of running the primes parallelism example. The VI found all the prime numbers in the first 400,000 natural numbers on a 3.06 GHz Pentium 4 Windows XP computer.

Version	Hyper-Threading	No Hyper-Threading
LabVIEW 7.0	6.77	3.39
LabVIEW 7.1	1.91	3.40

The difference in performance of LabVIEW 7.0 and LabVIEW 7.1 without Hyper-Threading is a result of sampling error.

Linux

The performance benchmarks in the previous sections were all tested on Windows computers, but Linux systems also can run on Hyper-Threaded computers. Because of the many variations of Linux, it is difficult to gather accurate benchmarking data for LabVIEW for each variation of Linux. However, tests demonstrate that LabVIEW 7.1 outperforms earlier versions of LabVIEW on multiprocessor Linux systems on Hyper-Threaded computers.

342428A-01 Mar04