Multithreading Performance

Ruediger R. Asche
Microsoft Developer Network Technology Group

January 31, 1996

Abstract

This article discusses strategies for rewriting single-threaded applications to be multithreaded applications. It analyzes the performance of multithreaded computations over compatible single-threaded ones in terms of throughput and response.

Introduction

Most of the material you can find about multithreading deals with synchronization concepts, such as how to serialize threads that share common data. This focus on synchronization makes sense because synchronization is an indispensable part of multithreaded programming. This article takes a step back and focuses on an aspect of multithreading that is hardly documented: Deciding how a computation can meaningfully be split into multiple threads. The accompanying sample application, THRDPERF, implements a test suite that compares serial and concurrent implementations of the same computations to each other in terms of throughput and performance.

The first section of this article establishes some vocabulary on multithreaded applications, discusses the scope of the test suite, and describes how the sample application suite was designed. The second section discusses the results of the tests and contains recommendations for multithreaded application design. The related article "Interacting with Microsoft Excel: A Case Study in OLE Automation" discusses one interesting side issue of the sample application suite, namely, how the data that was obtained using the test set was fed into Microsoft Excel using OLE Automation.

If you are an experienced programmer of multithreaded applications, you can probably safely skip the introductory section and jump right to the "Results" section below.

Multithreading Vocabulary

So your application has been around for a long time—it works great, is reliable, and the whole bit—but it's terribly sluggish, and you have a bazillion ideas of how you could make good use of multithreading. Wait a second before going to work; there are a number of traps that can lead you into believing that a particular multithreaded design is good when in reality it is not.

Before you jump to conclusions about what I'm getting you into here, let's first clarify what we are *not* discussing in this article:

- We are not concerned with the different libraries that provide access to multithreading under the Windows application programming interface (API). The sample application suite, Threadlib.exe, was written using the multithreading API in a Microsoft Foundation Class Library (MFC) application, but I am not concerned at all with whether the Microsoft C run-time (CRT) libraries, the MFC libraries, or the barebones Windows API is used to create and maintain the threads.

As a matter of fact, each of these libraries will, at the end of the day, call to the **CreateThread** function to create a worker thread, and the multithreading itself will always be performed by the operating system. Which one of the encapsulation mechanisms you use does not make any difference for the purpose of this article. There may be a performance penalty involved when using one or the other wrapper library, but here we are mostly concerned with the essentials of multithreading, not the wrapping.

- The discussions in this article refer to multithreaded applications that run on single-processor machines. Multiprocessor computers are in a completely different league, and almost none of the discussion in this article applies to multiprocessor machines. I haven't had a chance yet to execute the set on a scalable symmetric multiprocessing (SMP) machine. If you have access to one, I'd love to see your results.
- Within this article, I rather generically refer to "computations." A *computation* is defined as a subtask of your application that may be executed in parts or as a whole before or after another computation or concurrently with other computations. As an example, let us consider an application that requests user data and saves the data onto disk. We could argue that entering the data constitutes one computation, and saving the data another one. Depending on the application design, it is possible either to interleave the saving of the data with the input of new data or to wait until the user has entered all data before saving all the data onto disk. The former case can typically be implemented using some form of multithreading; we refer to that way of organizing the computations as *concurrent* or *interleaved*. The latter case is typically implemented in a single-threaded application and is in this article referred to as *serial execution*.
- The design of concurrent applications is a very complex process that is normally performed by people who make a ton of money, because it takes years of study to figure out how exactly a given task can benefit from concurrent execution. This article does not intend to teach you how to design multithreaded applications. Instead, I point out some of the problem areas of multithreaded application design to you, and I use real-life performance measurements to argue my case. After reading this article, you should be able to look at a given design and be able to determine if that particular design will enhance the overall performance of the application or not.
- Part of the process of multithreaded application design is to determine where data access conflicts between multiple threads can potentially lead to data corruption and how to avoid such conflicts using thread synchronization. This task will not be discussed in this article at all. For the discussion in this article, we will assume that the computations that can be interleaved do not share any data and, therefore, do not require any serialization. This stipulation may seem a little bit restrictive, but please keep in mind that there can be no "generic" discussion of synchronized multithreaded applications because each serialization forces a unique waiting-and-waking pattern onto the serialized threads, which directly affects the performance.
- Most input/output (I/O) operations come in two flavors: asynchronous or synchronous. It turns out that in many cases a multithreaded design using synchronous I/O can be approximated using asynchronous single-threaded I/O. This article does not discuss asynchronous single-threaded I/O as an alternative to multithreading, but I encourage you to consider both designs.

  Note that the way the Windows I/O system is designed provides a few mechanisms that make asynchronous I/O preferable over synchronous I/O (for example, I/O completion ports). I plan on taking up the issue of synchronous versus asynchronous I/O in a later article.

- As the article "Multiple Threads in the User Interface" points out, multiple threads and graphical user interfaces (GUIs) do not work together very well. In the course of this article I imply that the work that can be performed by a background thread does not utilize the Windows GUI at all; the type of threads I deal with are merely "worker threads" that perform background computations that do not require direct interaction with the user.
- There are *infinite* computations as opposed to *finite* computations. An example of an infinite computation would be a "listening" thread in a server application, which does not serve any purpose but to wait for a client to connect to the server. After a client has connected, that thread sends a notification to the main thread and returns to the listening state until the next client connects. Naturally, such a computation cannot possibly reside in the same thread as an application's user interface (UI) unless an asynchronous I/O operation is employed. (Note that this particular problem can, and should, be solved using asynchronous I/O and

completion ports rather than multiple threads; I use the example for illustration purposes only). In this article, I will deal only with finite computations, that is, subtasks of an application that are completed after a finite amount of time.

CPU-bound Versus I/O-bound Computations

One of the most important factors that determines whether a given computation is a good candidate for a separate thread is whether the computation is CPU-bound or I/O-bound. A *CPU-bound computation* is a computation that spends most of its time keeping the CPU busy. Typical examples of CPU-bound computations are the following:

- Complex mathematical computations, such as matrix manipulations, operations on graphs, or off-screen graphics calculations
- Operations on memory-resident file images, such as scanning a memory image of a text file for a given string

In contrast, *I/O-bound computations* are computations that spend most of their time waiting for an I/O request to finish. In most modern operating systems, incoming device I/O will be serviced asynchronously, either by a dedicated I/O processor or by an efficient interrupt handler, and the I/O request from an application will suspend the calling thread until the I/O is completed. Threads that spend most of their time waiting for I/O requests normally do not compete with other threads for the CPU; thus, I/O-bound computations may not deteriorate the performance of other threads as much as CPU-bound threads. (I will explain this statement later.)

Note that this juxtaposition is pretty radical. Most computations are not completely I/O-bound or CPU-bound but instead have both I/O- and CPU-bound components. The same set of computations may run better using serialized computations in one scenario and using concurrent computations in another scenario, depending on the relative CPU- and I/O-bound distributions.

Goals for Multithreaded Designs

Before you think about multithreading your application, you should ask yourself what the goal of such a transition would be. Multithreading has several potential benefits:

- Enhanced performance
- Increased throughput
- Greater user responsiveness

Let us address each of those benefits in turn.

Performance

For the time being, let us define performance simply as the total elapsed time of a given computation or set of computations. Performance comparison, by definition, is an issue only when it comes to finite computations.

Believe it or not, the scenarios in which multithreading enhances the performance of an application are rather limited. The reasons for that are not obvious, but perfectly reasonable:

- Unless the application executes on a multiprocessor machine (in which subcomputations can truly execute in parallel), CPU-bound computations cannot possibly execute faster in multiple threads than in a single thread. This is because the single CPU must still execute all computations, be it in little pieces (in the multithreaded case) or in big chunks (as is the case when the computations are executed one after another in the same thread). As a rule, a given set of computations, when executed in multiple threads, will typically finish later than the same set of computations executed sequentially because there is an overhead incurred in creating

the threads and switching the CPU between threads.
- Normally, there has to be some point at which the results of computations must be synchronized with each other anyway, regardless of which computation finishes first. For example, if multiple threads are employed to read several files into memory concurrently, it is very likely that regardless of what order the files are processed in, at some point the application must wait until all data is read into memory before proceeding. We will look at this idea next in the "Throughput" section.

For the purpose of this article, we will measure performance in terms of *elapsed time*, that is, the total time it takes for all computations to finish.

Throughput

Throughput (or *response*) refers to the average turnaround time for each computation. As an example to demonstrate throughput, let us assume a supermarket scenario (always a great visualization tool for operating systems): Let each computation be the servicing of one customer at a checkout counter. One could either open up a separate checkout counter for each customer, or try to gate all customers through the same counter. In order to make our analogy work, we would require that in the multiple checkout case, only one cashier (poor individual!) serves all customers regardless of whether they line up on one or more checkout counters. Such a super cashier would jump from counter to counter at hyperspeed, ringing up only one item from one customer at a time, then moving on to the next customer. This super cashier simulates the CPU being chopped up between multiple computations.

As we saw in the "Performance" section earlier, the overall time it takes to serve all customers does not decrease when several checkout stands are open because it is always the one cashier that does all the work, regardless of whether customers are served from one or several checkout stands. However, chances are that the customers will on the average prefer the super cashier over only one checkout stand. This is because normally the carts of customers will look very different; some customers have a lot of items in their carts, and some very few. If you ever wanted to buy a box of granola bars and a quart of milk and got stuck behind someone shopping for a 24-person household, you'll know what I'm talking about.

In any case, if you would be served by Mr. Clark Kent in hyperspeed instead of waiting in line, you probably wouldn't care if it takes a little longer or not to get your shopping done because two items are rung up pretty fast anyway, and the shopping cart for the 24-person household is processed at a different counter, so you're out of there quickly regardless.

Thus, throughput is a measurement of how many computations can be performed in a given time. Each computation measures its own progress by how long it takes to complete the computation in relation to how long the computation is supposed to be in the first place. In other words, if you go into a supermarket and hope to be out of there in two minutes, but it takes you two hours to get your two items rung up because you got stuck behind Betty Crocker shopping for her 1997 product line, you'd say your progress was pretty lousy.

For the purpose of this article, we define the response time for a computation to be the time it takes for the computation to finish divided by the time it was expected to take. Thus, a computation that would take 10 milliseconds (ms) and finished after 20 ms would have a response turnaround of 2, but if the same computation would be finished after 200 ms (possibly because another long computation completed first), the response turnaround would be 20. Intuitively, the shorter the response turnaround, the better.

As we will see later on, throughput can be a relevant factor in introducing multithreading into an application even if the overall performance decreases; however, in order for throughput to be relevant as a measurement, certain conditions have to be met:

1. Each computation must be independent of others, in that the result of any computation can be processed or used as soon as the computations are done. If you are a member of a college football team, each of whom

tries to buy his travel goodies at the same supermarket, it won't make a difference to you if your items get rung up first or last—or even how fast your two items get rung up compared to how long you waited—because in the end your bus won't leave before everybody has been served anyway, and your waiting time would simply be shifted from waiting in line to waiting for everybody else if super cashier served you. *This is an important point that is frequently neglected.* As I mentioned before, most applications sooner or later do synchronize their computations implicitly or explicitly. For example, if your application gathers data from separate files concurrently, you will probably want to display the results on the screen, or save them into another file. In the former case (displaying the results on the screen), you should be aware that most graphic systems perform some kind of internal batching or serialization that may very well display nothing until all output has been collected; in the latter case (saving results into another file), there is normally not a whole lot your application (or another one) can do until the entire protocol file has been written. Thus, all the benefits you could gain from processing the files may disappear if somebody or something—either the application, the operating system, or possibly even the user—serializes the results in some form.

2. The computations must have significantly different sizes. If everybody in the supermarket has only two items to ring up, the supercashier doesn't do any good; if he has to jump between three cash registers, each of which serves a customer who has exactly two items (or three, or four, or *n*) to ring up, then each customer has to wait *n* times as long to get his or her purchases finished, which is worse than lining up all customers in the same line. Think of multithreading as a shock absorber here: The short computations do not run a high risk of getting stuck behind long computations, but instead become distributed to threads in which they may finish sooner in the long run.
3. If the lengths of the computations can be determined beforehand, a serial process will be better than a multithreaded one, simply by ordering the computations in ascending order. In the supermarket example, this would correspond to lining up the customers by numbers of items (a variation of the Express Lane scheme), the idea being that customers with few items to ring up will appreciate it if their short jobs will not be delayed significantly, and those with many items don't care because they'll have to wait a long time anyway with all of their merchandise, and everybody before them has less than they do.

If you know approximately what the span of computation times is, but your application cannot sort the computations, you should take the time to perform a worst-case analysis. In such an analysis, you would assume that the computations would not be lined up in ascending time order, but on the contrary, in *descending* order. This scenario can be considered worst-case in terms of response because each computation would have the highest possible response turnaround as measured by the formula defined above.

Responsiveness

The final criterion for multithreading an application that I will discuss here is responsiveness (which is linguistically close enough to *response* to totally confuse you). Let us for the purpose of this article simply define an application as *responsive* if its design guarantees that the user can always interact with the application within a short time (short time here means short enough for the user not to have the impression that the application has hung).

For an application with a GUI, responsiveness can be accomplished rather easily as long as it is ensured that lengthy computation be delegated to background threads, but the architecture required to accomplish responsiveness may be a little tricky; as I mentioned earlier, somebody will probably wait for a computation to return sooner or later, so to perform a lengthy computation in the background may require changes to the UI (for example, a Cancel option may have to be added, and menu choices that depend on the result of the computation may have to be dimmed).

Reasons other than performance, throughput, and responsiveness may recommend multithreaded designs. For example, in certain architectures it is necessary to let computations interleave in a pseudo-random fashion (an example that comes to mind once more is the Bolzmann-machine type of neural networks, where the predicted behavior of the interconnected network only works if each node in the network performs its computation asynchronously). In this article, however, I will limit the discussion to the three factors mentioned above, that is, performance, throughput, and responsiveness.

Test Approach

I have heard a lot of discussions about abstraction mechanisms that encapsulate all the nasty aspects of multithreading into, say, a C++ object and therefore let an application get all the benefits of multithreading, but not the disadvantages.

I began this article by designing exactly such an abstraction. I defined a prototype for a C++ class, **ConcurrentExecution**, that would have member functions such as **DoConcurrent** and **DoSerial**, where the parameters to both member functions would be a generic array of objects and an array of callback functions that would be called on the respective objects concurrently or sequentially; the C++ class would encapsulate all of the gory details of maintaining the threads and the internal data structures.

It was pretty clear to me from the very beginning, however, that such an abstraction has extremely limited use at best because the bulk of the work in designing a multithreaded application goes into a task that cannot be automated—namely, the task of determining how to multithread. The first restriction on **ConcurrentExecution** was that the callback functions would not be allowed to share data implicitly or explicitly, or require any other form of synchronization that would immediately sacrifice all of the benefits of the abstraction and open up all of the traps and pitfalls of the wonderful world of synchronization, such as deadlocking, race conditions, or the need for fairly complex compound synchronization objects.

Likewise, it would not be permissible for computations that may potentially be executed concurrently to call into the UI because, as I mentioned earlier, the Windows API forces several implicit synchronizations onto threads that call into the UI. Note that there are many other API subsets and libraries that force implicit synchronizations upon threads that share them.

These restrictions left **ConcurrentExecution** with limited functionality, namely, an abstraction that manages pure worker threads (totally independent computations that are mostly limited to mathematical computations on disjoint memory areas).

Nevertheless, it turned out to be very useful to implement the **ConcurrentExecution** class and to put it to use through performance tests, because a lot of hidden details about multithreading surfaced when I implemented the class and designed and ran the test sets. Please be aware that although the **ConcurrentExecution** class can make multithreading easier to handle, the class implementation needs some work before being usable in a commercial product. In particular, I bail out of all error-condition handling in a very crude, intolerable way. I assume that for the purpose of the test suite (for which I exclusively use **ConcurrentExecution**), errors will simply not show up.

The ConcurrentExecution Class

Here is the prototype for the **ConcurrentExecution** class:

```
class ConcurrentExecution
{
 < private members omitted>
public:
   ConcurrentExecution(int iMaxNumberOfThreads);
   ~ConcurrentExecution();
   int DoForAllObjects(int iNoOfObjects,long *ObjectArray,
                       CONCURRENT_EXECUTION_ROUTINE pObjectProcessor,
                       CONCURRENT_FINISHING_ROUTINE pObjectTerminated);
```

```
    BOOL DoSerial(int iNoOfObjects, long *ObjectArray,
                  CONCURRENT_EXECUTION_ROUTINE pObjectProcessor,
                  CONCURRENT_FINISHING_ROUTINE pObjectTerminated);
};
```

The class is exported from the library Thrdlib.dll, which is one of the projects in the THRDPERF sample test suite. Let us first discuss the semantics of the member functions before discussing the internal architecture of the class:

```
ConcurrentExecution::ConcurrentExecution(int iMaxNumberOfThreads)
{
 m_iMaxArraySize = min(iMaxNumberOfThreads, MAXIMUM_WAIT_OBJECTS);
 m_hThreadArray = (HANDLE *)VirtualAlloc(NULL,m_iMaxArraySize*sizeof(HANDLE),
                            MEM_COMMIT,PAGE_READWRITE);
 m_hObjectArray = (DWORD *)VirtualAlloc(NULL,m_iMaxArraySize*sizeof(DWORD),
                            MEM_COMMIT,PAGE_READWRITE);
 // a real-life implementation must provide error handling here, of course...
};
```

You will notice that the **ConcurrentExecution** constructor takes a numeric argument. This argument specifies the "maximum degree of concurrency" that an instance of the class supports; in other words, if an instance of **ConcurrentExecution** is created with *n* as an argument, then not more than *n* computations will execute at any given time. According to our previous analogy, this argument means "do not open more than *n* checkout counters regardless of how many customers show up."

```
    int DoForAllObjects(int iNoOfObjects,long *ObjectArray,
                        CONCURRENT_EXECUTION_ROUTINE pObjectProcessor,
                        CONCURRENT_FINISHING_ROUTINE pObjectTerminated);
```

This is the only interesting member function that is implemented at this point. The main arguments to **DoForAllObjects** are an array of objects, a processor function, and a terminator function. There is no format whatsoever forced upon the objects; each time the processor is invoked, one of the objects is passed to it, and it is totally up to the processor to interpret the object. The first argument, **iNoOfObjects**, is simply to make known to **ConcurrentExecution** the number of elements there are in the array of objects. Note that calling **DoForAllObjects** with an object array of length 1 is fairly similar to simply calling **CreateThread** (except that **CreateThread** does not accept a terminator parameter).

The semantics of **DoForAllObjects** are as follows: The processor will be called for each of the objects. The order in which the objects are processed is not specified; all that is guaranteed is that each object will be passed to the processor at some point. The maximum degree of concurrency is determined by the parameter passed to the constructor of the **ConcurrentExecution** object.

The processor function cannot access shared data and cannot call into the UI or do anything else that requires explicit or implicit serialization. Currently, only one processor function exists to work on all objects; it would be easy, however, to replace the processor argument with an array of processors.

The prototype for the processor is as follows:

```
typedef DWORD (WINAPI *CONCURRENT_EXECUTION_ROUTINE)
        (LPVOID lpParameterBlock);
```

The terminator function is called immediately after the processor has finished working on one object. Unlike the processor, the terminator function is called serialized in the context of the calling function and can call everything and access all data that the caller can. It should be noted, however, that the terminator should be optimized as much as possible because lengthy computations in the terminator can affect the performance of **DoForAllObjects**. Note that, although the terminator is called as soon as the processor has finished each object, **DoForAllObjects** itself does not return before the last object has been terminated.

Why do we go through so much pain with the terminator? We could as well have each computation perform the terminator code at the very end of the processor function, right?

That is basically right; however, it is important to emphasize that the terminator is called in the context of the thread that called **DoForAllObjects**. That design makes it much easier to process the results of each computation as they come in, without having to worry about synchronization issues.

The prototype of the terminator function is as follows:

```
typedef DWORD (WINAPI *CONCURRENT_FINISHING_ROUTINE)
        (LPVOID lpParameterBlock,LPVOID lpResultCode);
```

The first parameter is the object that was processed, and the second argument is the result of the processor function on that object.

The sibling to **DoForAllObjects** is **DoSerial**, which has the same parameter list as **DoForAllObjects**, but the computations are processed in serial order, beginning with the first object in the list.

The Inner Works of ConcurrentExecution

Note

The discussion in this section is very technical and implies that you understand a lot about the threading API. If you are more interested in how the **ConcurrentExecution** class is used to gather the test data than in how **ConcurrentExecution::DoForAllObjects** is implemented, you can now scroll down to the "Using **ConcurrentExecution** to Sample Thread Performance" section.

Let's start with **DoSerial**, because that one is pretty much a no-brainer:

```
BOOL ConcurrentExecution::DoSerial(int iNoOfObjects,long *ObjectArray,
                        CONCURRENT_EXECUTION_ROUTINE pProcessor,
                        CONCURRENT_FINISHING_ROUTINE pTerminator)
{
    for (int iLoop=0;iLoop<iNoOfObjects;iLoop++)
    {
     pTerminator((LPVOID)ObjectArray[iLoop],(LPVOID)pProcessor((LPVOID)ObjectArray[iLoo
    };
 return TRUE;

};
```

The code simply loops through the array, calls the processor for each iteration, and then calls the terminator on the result of the processor and the object itself. Nice 'n clean, ain't it?

The interesting member function is **DoForAllObjects**. At first glance, there is nothing dramatic that **DoForAllObjects** should have to do—ask the operating system to create a thread for each computation, and make sure that the terminator function gets called properly. However, there are two issues that make **DoForAllObjects** trickier than it looks: First, the "maximum degree of concurrency factor" parameter that an instance of **ConcurrentExecution** is created with may require some additional bookkeeping when more computations than threads are available. Second, the terminator function for each computation is called in the context of the thread that calls **DoForAllObjects**, not the thread that the computation runs in; also, the terminator is called immediately after the processor has finished. It is a little tricky to address those concerns.

Let's go through the code to figure out what's going on. The code is adapted from the file Thrdlib.cpp, but abbreviated for clarity:

```
int ConcurrentExecution::DoForAllObjects(int iNoOfObjects,long *ObjectArray,
                       CONCURRENT_EXECUTION_ROUTINE pObjectProcessor,
                       CONCURRENT_FINISHING_ROUTINE
pObjectTerminated)
{
 int iLoop,iEndLoop;
 DWORD iThread;
 DWORD iArrayIndex;
 DWORD dwReturnCode;
 DWORD iCurrentArrayLength=0;
 BOOL bWeFreedSomething;
     char szBuf[70];

m_iCurrentNumberOfThreads=iNoOfObjects;

HANDLE *hPnt=(HANDLE *)VirtualAlloc(NULL,m_iCurrentNumberOfThreads*sizeof(HANDLE)
                               ,MEM_COMMIT,PAGE_READWRITE);
 for(iLoop=0;iLoop<m_iCurrentNumberOfThreads;iLoop++)
 hPnt[iLoop] = CreateThread(NULL,0,pObjectProcessor,(LPVOID)ObjectArray[iLoop],
              CREATE_SUSPENDED,(LPDWORD)&iThread);
```

First, we create individual threads for each of the objects. Because we use CREATE_SUSPENDED to create the threads, no thread will be started yet. An alternative would be to create each thread as it is needed. I decided not to use that alternative strategy because I found that a **CreateThread** call is much more expensive when called with several threads running in the same application; thus, the overhead to create the threads at this point is much more bearable than creating every thread "on the fly" as we go along.

```
for (iLoop = 0; iLoop < m_iCurrentNumberOfThreads; iLoop++)
   {
    HANDLE hNewThread;
    bWeFreedSomething=FALSE;
// If array is empty, allocate one slot and boogie.
     if (!iCurrentArrayLength)
     {
      iArrayIndex = 0;
     iCurrentArrayLength=1;
    }
    else
    {
// First, check if we can recycle any slot. We prefer to do this before we
// look for a new slot so that we can invoke the old thread's terminator right
// away...
      iArrayIndex=WaitForMultipleObjects(iCurrentArrayLength,
                                         m_hThreadArray,FALSE,0);
     if (iArrayIndex==WAIT_TIMEOUT)  // no slot free...
     {
       {
       if (iCurrentArrayLength >= m_iMaxArraySize)
       {
      iArrayIndex= WaitForMultipleObjects(iCurrentArrayLength,
                                         m_hThreadArray,FALSE,INFINITE);
      bWeFreedSomething=TRUE;
       }
      else // We could free up a slot somewhere, so go for it...
       {
        iCurrentArrayLength++;
        iArrayIndex=iCurrentArrayLength-1;
       }; // Else iArrayIndex points to a thread that has been nuked
       };
      }
      else bWeFreedSomething = TRUE;
    }; // At this point, iArrayIndex contains a valid index to store the
       // new thread in.
    hNewThread = hPnt[iLoop];
     ResumeThread(hNewThread);
     if (bWeFreedSomething)
       {
         GetExitCodeThread(m_hThreadArray[iArrayIndex],&dwReturnCode); //error
       CloseHandle(m_hThreadArray[iArrayIndex]);
```

```
        pObjectTerminated((void *)m_hObjectArray[iArrayIndex],(void *)dwReturnCode);
        };
      m_hThreadArray[iArrayIndex] = hNewThread;
      m_hObjectArray[iArrayIndex] = ObjectArray[iLoop];
   }; // End of for loop
```

The heart of **DoForAllObjects** is **hPnt**, an array of threads that is allocated when the **ConcurrentExecution** object is constructed. This array can accommodate as many threads as the maximum degree of concurrency specified in the constructor; thus, each element in the array is a "slot" into which one computation fits.

The algorithm to determine how to fill and free the slots is as follows: The array of objects is traversed from the beginning to the end, and for each object, we do the following: If no slot has been filled yet, we fill the first slot in the array with the current object and resume the thread that will process the current object. If there is at least one slot in use, we use the **WaitForMultipleObjects** function to determine whether any of the ongoing computations has finished yet; if yes, we call the terminator on that object and "recycle" the slot for the new object. Note that we might also first fill up every free slot until there are no slots left and then start filling up vacant slots. However, if we did that, the terminator functions for the vacated slots wouldn't be called until all slots have been filled, which violates our requirement that the terminator gets called as soon as the processor has finished one object.

Finally, it may be the case that no slot is free (that is, the number of currently active threads is equal to the maximum degree of concurrency that the **ConcurrentExecution** objects allows). In that case, **WaitForMultipleObjects** is called again to put **DoForAllObjects** to sleep until one slot is vacated; as soon as that happens, the terminator is called on the vacating object, and the thread that works on the current object is resumed.

Eventually, all computations will either have finished or will occupy slots in the array of threads. The following code will process all remaining threads:

```
iEndLoop = iCurrentArrayLength;
   for (iLoop=iEndLoop;iLoop>0;iLoop--)
   {
    iArrayIndex=WaitForMultipleObjects(iLoop, m_hThreadArray,FALSE,INFINITE);
    if (iArrayIndex==WAIT_FAILED)
    {
        GetLastError();
        _asm int 3;       // Do something intelligent here...
    };
     GetExitCodeThread(m_hThreadArray[iArrayIndex],&dwReturnCode);  // Error?
     if (!CloseHandle(m_hThreadArray[iArrayIndex]))
       MessageBox(GetFocus(),"Can't delete thread!","",MB_OK); // Make this
                                                               better...

     pObjectTerminated((void *)m_hObjectArray[iArrayIndex],
                                   (void *)dwReturnCode);
    if (iArrayIndex==iLoop-1) continue;   // We are fine here; no backfilling
                                          in need.
    m_hThreadArray[iArrayIndex]=m_hThreadArray[iLoop-1];
    m_hObjectArray[iArrayIndex]=m_hObjectArray[iLoop-1];
   };
```

Finally, clean up:

```
if (hPnt) VirtualFree(hPnt,m_iCurrentNumberOfThreads*sizeof(HANDLE),
                              MEM_DECOMMIT);

  return iCurrentArrayLength;

};
```

Using ConcurrentExecution to Sample Thread Performance

The scope of the performance test is as follows: The user of the test application Threadlibtest.exe can specify whether to test CPU-bound or I/O-bound computations, how many computations to perform, how long the computations are, how the computations are ordered (in order to test worst case versus random delays), and whether the computations are to be performed concurrently or serially.

In order to eliminate stray results, each test can be performed ten times, and the results of all ten tests are averaged to yield a more reliable result.

By choosing the menu option "Run entire test set," the user can request to run permutations of all test variables. The computation lengths used in the test vary between 10 and 3,500 ms base values (I will discuss these in a second), and the number of computations varies between 2 and 20. If Microsoft Excel has been installed on the machine that runs the tests, Threadlibtest.exe will dump the results in a Microsoft Excel sheet located at C:\Temp\Values.xls. The resulting values will in any case also be saved into a clear text file located at C:\Temp\Results.fil. Note that my hardcoding the protocol file locations is pure laziness; if you need to recreate the test results on your machine and need a different location, simply rebuild the project, changing the values of the TEXTFILELOC and SHEETFILELOC identifiers near the beginning of Threadlibtestview.cpp.

Keep in mind that running the entire test set will always order the computations in worst-case order (that is, in serial execution mode, the longest computation will be performed first, followed by the second longest, and so forth). This scenario penalizes the serial execution variation a little bit in that the response times for concurrent executions will not change under a scenario that is not worst-case, whereas the response times for serial execution would probably improve.

As I mentioned earlier, in a real-life scenario you should probably analyze whether the duration of the individual computations can be predicted.

The code that utilizes the **ConcurrentExecution** class to gather performance data is located in Threadlibtestview.cpp. The sample application itself (Threadlibtest.exe) is a straightforward single-document interface (SDI) MFC application. All the code that's relevant for the sampling resides in the view class implementation **CThreadLibTestView**, which derives from **CEasyOutputView**. There is not a whole lot of interesting code in that class; mostly it's statistical number crunching and user-interface processing. The "meat" in executing the tests is in **CThreadLibTestView::ExecuteTest**, which will perform one test run. Here's the abbreviated code for **CThreadLibTestView::ExecuteTest**:

```
void CThreadlibtestView::ExecuteTest()
{
 ConcurrentExecution *ce;
 bCPUBound=((m_iCompType&CT_IOBOUND)==0); // This is global...
 ce = new ConcurrentExecution(25);
 if (!QueryPerformanceCounter(&m_liOldVal)) return; // Get current time.
 if (!m_iCompType&CT_IOBOUND) timeBeginPeriod(1);
 if (m_iCompType&CT_CONCURRENT)
  m_iThreadsUsed=ce->DoForAllObjects(m_iNumberOfThreads,
                                     (long *)m_iNumbers,
                                     (CONCURRENT_EXECUTION_ROUTINE)pProcessor,
                                     (CONCURRENT_FINISHING_ROUTINE)pTerminator);
 else
  ce->DoSerial(m_iNumberOfThreads,
               (long *)m_iNumbers,
               (CONCURRENT_EXECUTION_ROUTINE)pProcessor,
               (CONCURRENT_FINISHING_ROUTINE)pTerminator);
 if (!m_iCompType&CT_IOBOUND) timeEndPeriod(1);
 delete(ce);
 < the rest of the code sorts the results into an array for Excel to process...>
}
```

The code first creates an object of the class **ConcurrentExecution**, then samples the current time (which will be used to compute the elapsed and response time for the computations), and, depending on whether a serial or concurrent executed was requested, calls the **DoSerial** or **DoForAllObjects** member of the **ConcurrentExecution** object, respectively. Note that I request a maximum concurrency degree of 25 for concurrent execution; if you want to run the test suite on more than 25 computations, you should raise that value to something that is greater than or equal to the maximum number of computations you run your test on.

Let us look at the processor and terminator functions to figure out what is measured exactly:

```
extern "C"
{
long WINAPI pProcessor(long iArg)
{
 PTHREADBLOCKSTRUCT ptArg=(PTHREADBLOCKSTRUCT)iArg;
 BOOL bResult=TRUE;
 int iDelay=(ptArg->iDelay);
 if (bCPUBound)
 {
  int iLoopCount;
  iLoopCount=(int)(((float)iDelay/1000.0)*ptArg->tbOutputTarget->m_iBiasFactor);
  QueryPerformanceCounter(&ptArg->liStart);
  for (int iCounter=0; iCounter<iLoopCount; iCounter++);
 }
 else
 {
   QueryPerformanceCounter(&ptArg->liStart);
   Sleep(ptArg->iDelay);
 };
 return bResult;
}

long WINAPI pTerminator(long iArg, long iReturnCode)
{
 PTHREADBLOCKSTRUCT ptArg=(PTHREADBLOCKSTRUCT)iArg;
 QueryPerformanceCounter(&ptArg->liFinish);
 ptArg->iEndOrder=iEndIndex++;
 return(0);
}

}
```

The processor simulates a computation of a length that has been placed into a computation-specific data structure, a THREADBLOCKSTRUCT. The THREADBLOCKSTRUCT holds data that is relevant for a computation, such as its delay, its beginning and end time (in terms of performance counter ticks), and a back pointer to the view that utilizes the structure.

I/O-bound computations are simulated by simply putting the computation to sleep for the time specified. A CPU-bound computation will enter a **for** loop with an empty body. Some comments are in order here to understand what the code does: The computation is CPU-bound and is supposed to execute for a specified number of milliseconds. In earlier versions of the test application, I had the **for** loop simply iterate as many times as the delay specified without worrying what the number meant. (Due to the coding involved, the number actually meant milliseconds for I/O-bound computations, but iterations for CPU-bound computations.) In order to be able to compare CPU- and I/O-bound computations to each other in terms of absolute times, however, I decided to rewrite the code so that the computation-specific delay meant milliseconds for both I/O-bound and CPU-bound computations.

I discovered that it is not easy to write code that simulates CPU-bound computations of a specific, predefined length. The reason for that is that such code cannot query the system time itself because the call involved will most likely yield the CPU sooner or later and, therefore, violate the requirement to be CPU-bound. Trying to use asynchronous multimedia timer events was also unsatisfying because of the way the timer services work under Windows. The thread that sets a multimedia timer is, in effect, suspended until the timer callback is called; thus, the CPU-bound computation suddenly becomes an I/O-bound operation.

Thus, I ended up using a slightly sleazy trick: The code in **CThreadLibTestView::OnCreate** runs 100 loops that count from 1 to 100,000 and samples the average time to go through that loop. The result is stored in the member variable **m_iBiasFactor**, a float that is used in the processor function to determine how milliseconds translate into iterations. Unfortunately, because of the highly dynamic nature of operating systems, it is difficult to determine reliably how many iterations through a given loop it takes to run a computation of a specific length. However, I found that the strategy employed works fairly reliably in determining computation times for CPU-bound operations.

Note

If you rebuild the test application, be careful with the optimization options. If you specify the "Minimize execution time" optimization, the compiler will detect a **for** loop with an empty body and eliminate the loop altogether.

The terminator is very simple: The current time is sampled and stored in the THREADBLOCKSTRUCT of the computation. After the test has been completed, the code computes the difference between the time that

ExecuteTest was executed and the terminator was called for each computation. The total elapsed time for all computations is then determined as the time it took the last of all finished computations to finish, and the response time is computed as the average of all the response times of the individual computations, where each response time, once more, is defined as the thread's elapsed time from the beginning of the test divided by the thread's delay factor. Note that the terminator runs serialized in the main thread's context, so the increment instruction on the shared **iEndIndex** variable is safe.

That's really all there is to the test; the rest is mostly setting up parameters for the test runs and performing some math on the results. The logic that stuffs the results into a Microsoft Excel sheet is discussed in the article "Interacting with Microsoft Excel: A Case Study in OLE Automation."

Results

If you wish to recreate the test results on your machine, you should do the following:

1. If you need to change test parameters, such as the maximum number of computations or the location of the protocol files, edit Threadlibtestview.cpp in the THRDPERF sample project and rebuild the application. (Note that you need long file name support on the build machine to build the application.)
2. Make sure that Thrdlib.dll is in a location from which Threadlibtest.exe can link to it.
3. If you would like to use Microsoft Excel to view the test results, make sure that Microsoft Excel is properly installed on the machine that runs the tests.
4. Execute Threadlibtest.exe and choose "Run entire test set" from the Run performance tests menu. One test run typically takes several hours to complete.
5. After the test is completed, examine the results using either the plain text protocol file in C:\Temp\Results.fil or the spreadsheet C:\Temp\Values.xls. Note that the Microsoft Excel automation logic does not automatically generate charts from the raw data for you; I have used a few macros to rearrange the results and generate the charts for you. I hate number crunching, but I really must give Microsoft Excel credit for providing such a good UI that even a spreadsheet-paranoid person like me can make a few columns of data into a useful chart within a few minutes.

The test results I present were gathered on a 486/33 MHz system with 16 MB of RAM. The computer has both Windows NT (version 3.51) and Windows 95 installed; thus, the test results for the respective tests on both operating systems are compatible in terms of underlying hardware.

So, let's go into the interpretation of the values. Here are the charts that summarize the computation results; the interpretations follow. The charts should be read as follows: The x-axis on each chart has six values (except for the elapsed-time charts for long computations, which have only five because in my test run, the counter overflowed at the very long computations). A value represents a number of computations; I ran each test with 2, 5, 8, 11, 14, and 17 computations. In the resulting Microsoft Excel sheet, you will find results for each number of computations for CPU-bound and I/O-bound threads, executed concurrently and sequentially, with delay biases of 10 ms, 30 ms, 90 ms, 270 ms, 810 ms, and 2430 ms, but in the charts I include only the 10 ms and 2430 ms results so that all the numbers are reduced to something easier to comprehend.

I need to explain the meaning of "delay bias." If a test is run with a delay bias of *n*, each computation has a multiple of *n* as its computation time. For example, if 5 computations with a delay bias of 10 are sampled, then one of the computations will execute for 50 ms, the second one for 40 ms, the third one for 30 ms, the fourth one for 20 ms, and the fifth one for 10 ms. Once more, when the computations are performed sequentially, worst-case order is assumed, so that the computation with the longest delay is executed first, and the other ones follow in descending order. Thus, in the "ideal" case (that is, an execution with no overhead), the total required time for all computations would be 50 ms + 40 ms + 30 ms + 20 ms + 10 ms = 150 ms for CPU-bound computations.

The values on the y-axis correspond to milliseconds for the elapsed charts and relative turnaround lengths (that is, milliseconds in fact executed divided by milliseconds scheduled) for response charts.

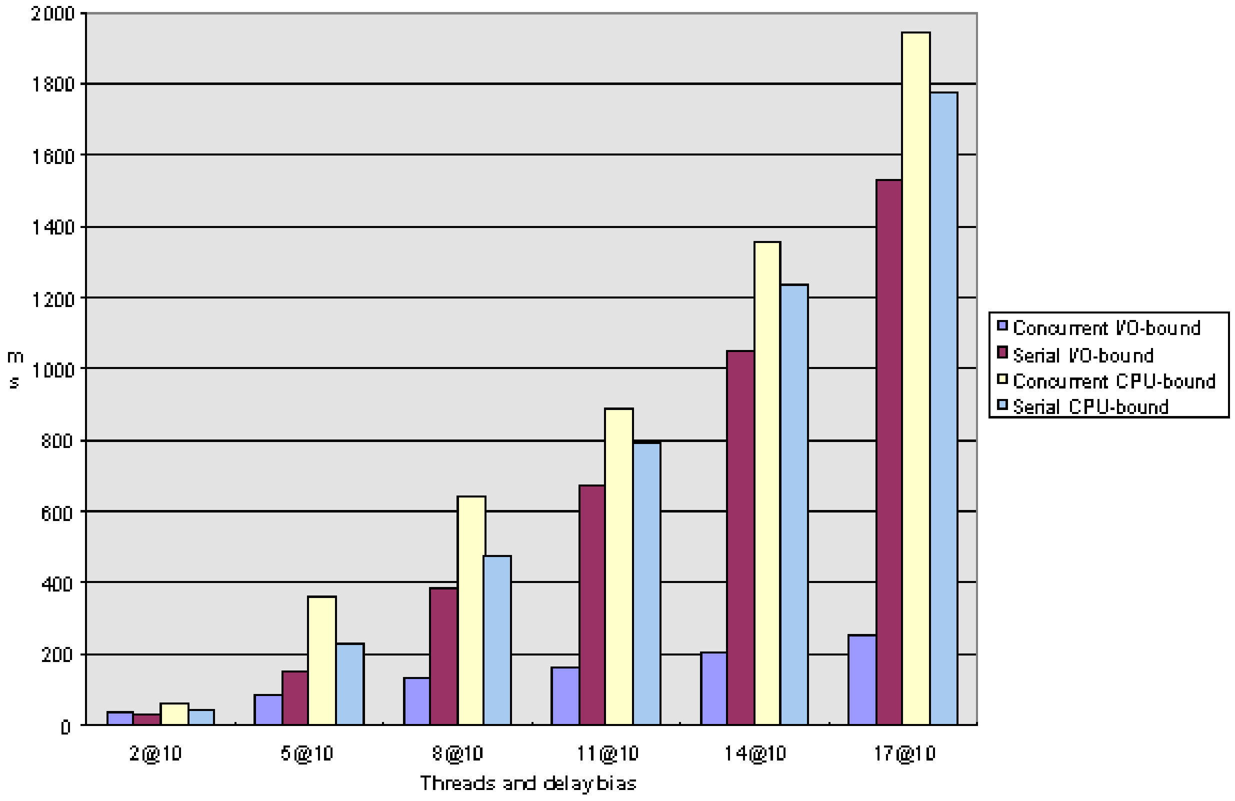

Figure 1. Elapsed-time comparison for short computations under Windows NT

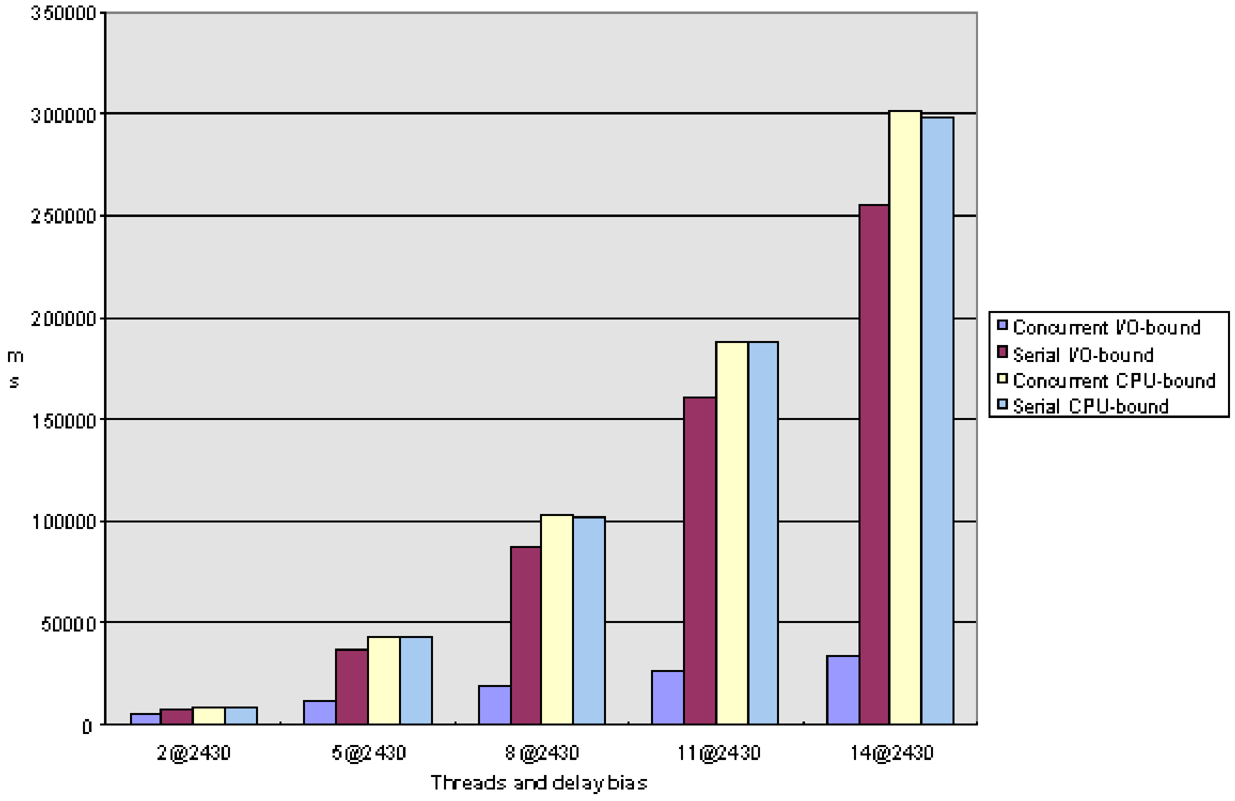

Figure 2. Elapsed-time comparison for long computations under Windows NT

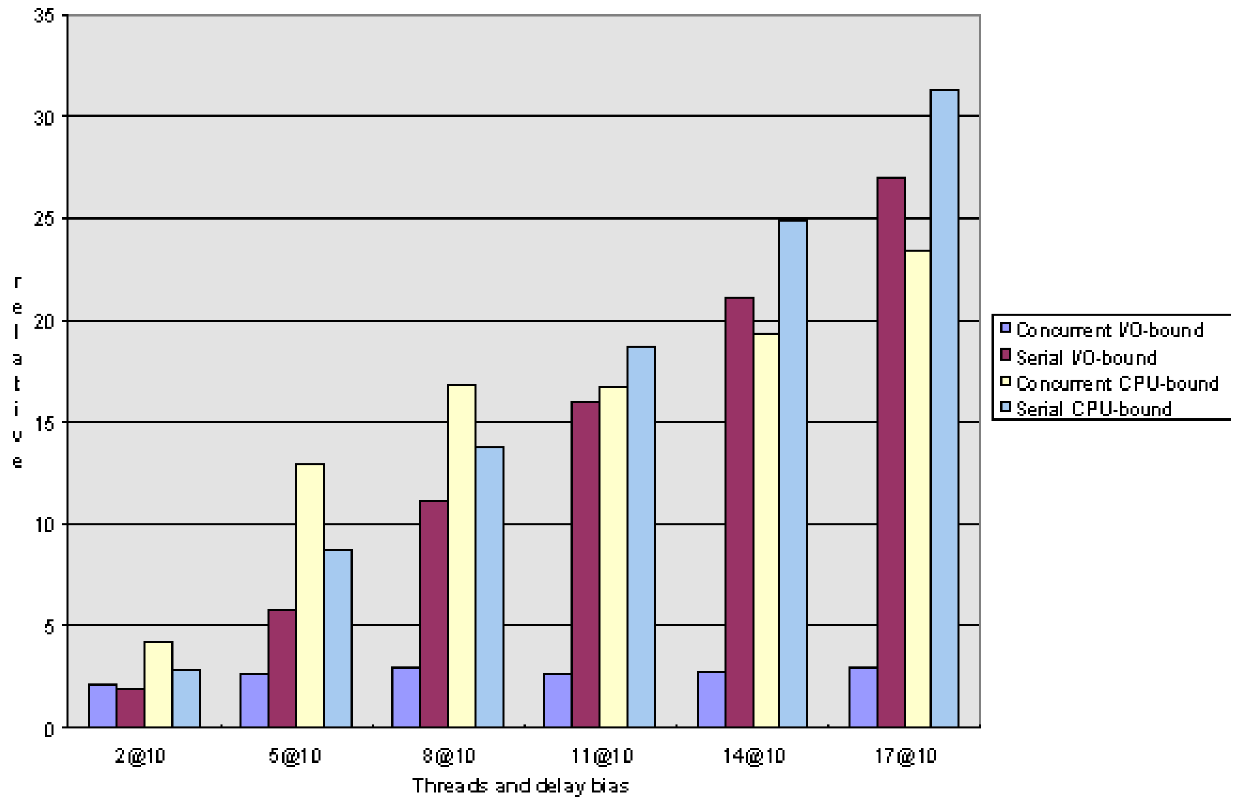

Figure 3. Response-time comparison for short computations under Windows NT

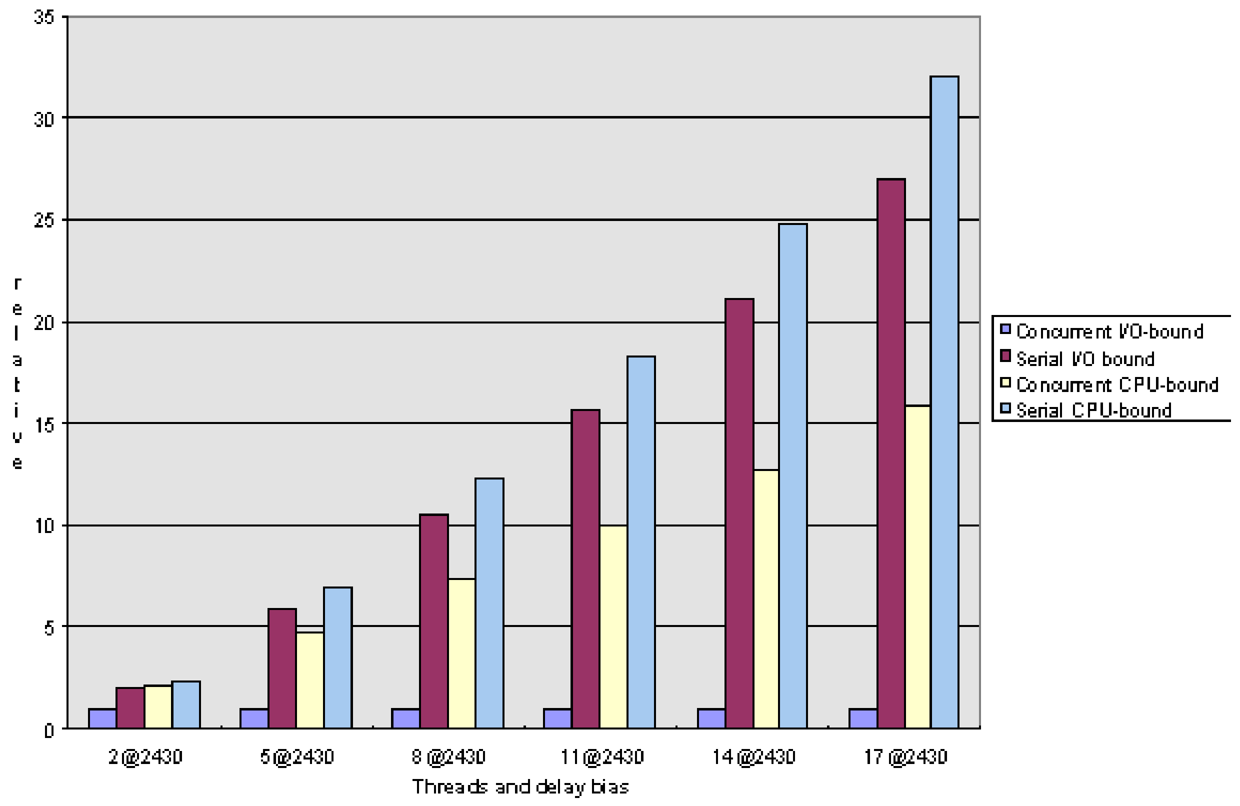

Figure 4. Response-time comparison for long computations under Windows NT

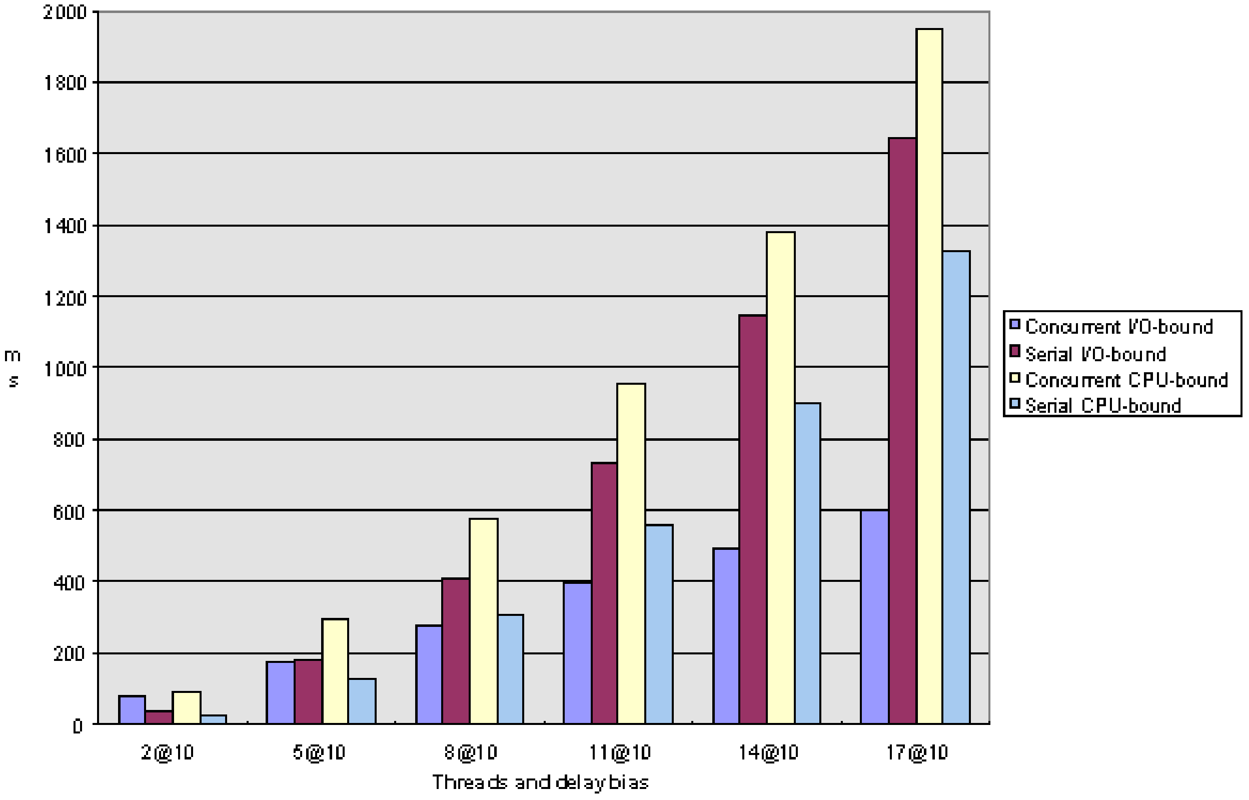

Figure 5. Elapsed-time comparison for short computations under Windows 95

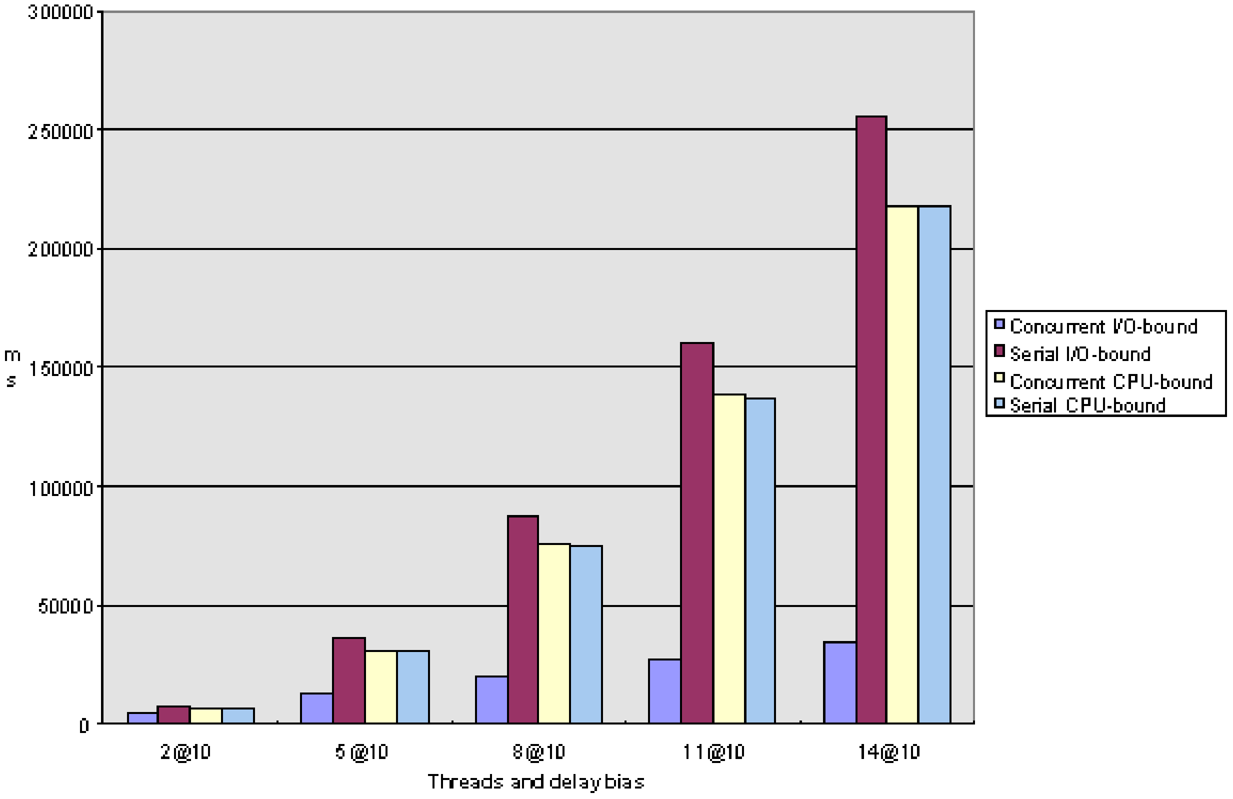

Figure 6. Elapsed-time comparison for long computations under Windows 95

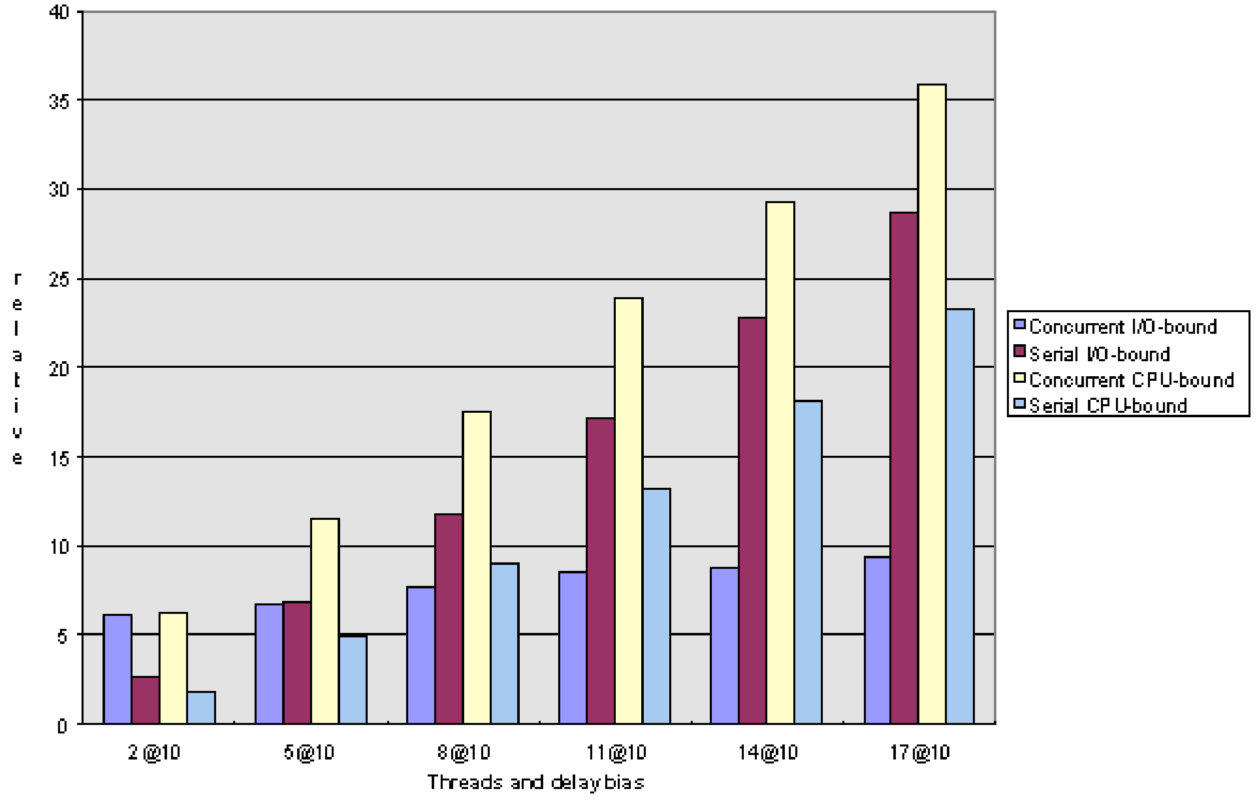

Figure 7. Response-time comparison for short computations under Windows 95

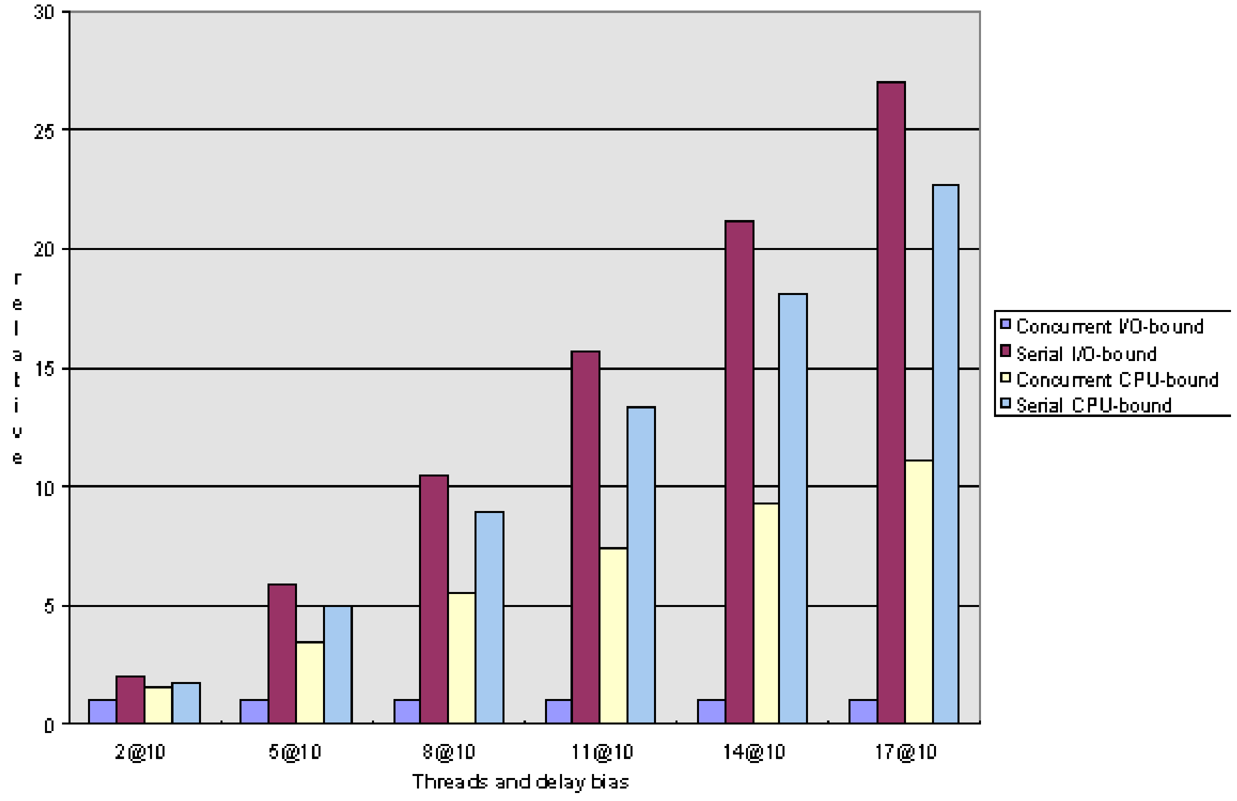

Figure 8. Response-time comparison for long computations under Windows 95

I/O-bound Tasks

In terms of both elapsed and turnaround time, I/O-bound threads perform dramatically better when performed concurrently rather than sequentially. As a function of computations, the elapsed time increases in a linear fashion for concurrent executions and exponentially for serial executions (see Figures 1 and 2 for Windows NT and Figures 5 and 6 for Windows 95).

Note that this observation is consistent with our earlier analysis that I/O-bound computations are good candidates for multithreading because the time that a thread is suspended while waiting for an I/O request to complete does not use up CPU-time and can, therefore, be meaningfully used by another thread.

The average response is about constant for concurrent computations and increases linearly for serial ones (see Figures 3, 4, 7, and 8, respectively).

Note that in any case, the scenarios in which few computations execute do not behave significantly differently if executed serially or concurrently, regardless of the test parameters.

CPU-bound Tasks

As we said earlier, CPU-bound tasks cannot possibly execute faster concurrently than serially when executed on a single-processor machine, but we can see that under Windows NT, the thread creation and switching overhead is quite good; for very short computations, the concurrent execution is only about 10 percent slower than the sequential one, and as the computation lengths increase, the two times approach within 99 percent of each other. In terms of response time, we can see that with long computations, the response gain for concurrent execution over serial execution can be as much as 50 percent, but for short computations, serial execution tends to actually do better than concurrent.

Comparisons Between Windows 95 and Windows NT

If we look at the charts for the long computations (that is, Figures 2, 4, 6, and 8), we can see that the behavior of Windows 95 and Windows NT is remarkably alike. Please don't be confused by the fact that Windows 95 seems to handle I/O-bound versus CPU-bound computations differently than Windows NT. I attribute this observation to the fact that the algorithm I employ to determine how many test iterations correspond to a millisecond (as described above) is rather inaccurate; I found that the same algorithm, executed several times under exactly the same circumstances, may yield results that differ by as much as 20%. Thus, comparing CPU-bound to I/O-bound operations is not really a fair thing to do.

The one area in which Windows 95 and Windows NT differ is when it comes to short computations. As we can see in Figures 1 and 5, Windows NT does much better for concurrent I/O-bound short computations. I attribute this observation to a more efficient thread-creation scheme. Note that for long computations, the difference between serial and concurrent I/O operations vanishes, so we are dealing with a fixed, relatively small overhead here.

In terms of response time for short computations (Figures 3 and 7), note that under Windows NT there is a break-even point at about 10 threads where more computations perform better when executed concurrently, whereas for Windows 95, serial computations do better throughout.

Please note that these comparisons are made based on the current versions of the respective operating systems (Windows NT version 3.51 and Windows 95), and as the operating systems evolve, the threading engines will very possibly be enhanced, so that differences in the respective behaviors of the two operating systems may disappear. It is interesting to note, however, that short computations do not generally seem to make good candidates for multithreading, and they specifically don't under Windows 95.

Recommendations

The results lead us to the following recommendations: The major factor that determines multithreading performance is I/O-bound computation versus CPU-bound computation, and the major criterion that determines whether to multithread at all is foreground user responsiveness.

Let us assume that in your application, there are several subcomputations that can potentially be executed in separate threads. In order to determine whether multithreading those computations makes sense, consider the following points.

If the user-interface-responsiveness analysis determines that something should be done in secondary threads, it makes sense to determine whether the tasks to be performed are CPU-bound or I/O-bound. I/O-bound computations are good candidates to be relocated into background threads. (Note, however, that asynchronous single-threaded I/O processing may be preferable to multithreaded synchronous I/O, depending on the problem.) CPU-bound threads that are very long may benefit from being executed in separate threads; however, unless the response of the threads is important, it probably makes sense to execute all CPU-bound subtasks in the same background thread instead of in separate threads. Remember that in any case, short computations will normally suffer from considerable overhead in thread creation when executed concurrently.

If the response is crucial for CPU-bound computations—that is, the results of the individual computations can be utilized as soon as they are obtained—you should try to determine whether the computations can be ordered in ascending order, in which case the overall performance will still be better when the computations are executed sequentially than when executed in parallel. Note that there are some computer architectures that are designed to process long computations (such as matrix operations) very efficiently; thus, by multithreading long computations on such a machine, you might actually forfeit some of the benefits of those architectures.

All of this analysis assumes that the application is run on a single-processor machine *and* the computations are independent. In fact, if the computations are dependent and serialization is required, the performance of serial execution will not be affected (because the serialization is implicit), whereas the concurrent version will always be affected adversely.

I also suggest that you base your multithreaded design on the degree of dependency. In most cases, it is intuitively clear which subcomputations are good candidates for multithreading, but if you have several choices for splitting your application into subcomputations that could be processed in separate threads, I'd recommend that you use the complexity of synchronization as a criterion. In other words, a split into multiple threads that requires very little and straightforward synchronization is preferable to one that requires heavy and complex thread synchronization.

As a final note, please remember that threads are a system resource that is not free; thus, there may be a greater penalty to multithreading than performance hits alone. As a rule of thumb, I would argue that you employ multithreading intelligently and conservatively. Use threads where they can benefit your application design, but avoid them wherever serial execution can achieve the same effect.

Summary

Running the attached performance test suite yields a few spectacular results that provide quite a few insights into the inner logic of concurrent application design. Please note that many of my assumptions are pretty radical; I chose to compare very long to very short computations, and I assumed the computations to be completely independent and either completely I/O-bound or completely CPU-bound. Most real-life problems lie somewhere in between in terms of computation lengths and boundedness. Please consider the material in the article as a starting point for you to have a closer look at your application to determine where to multithread.

A future article in this series will deal with performance enhancements for I/O-bound operations through asynchronous I/O.