|  |
| --- |
| MS_logo_K |
| Taking Parallelism Mainstream |
| ***Parallel technologies in Microsoft Visual Studio 2010 can help simplify the transition to parallel computing for today’s software developers***  |
|  |
| **Published: 10/9/2008, Updated: 2/27/2009** |

|  |
| --- |
|  |



Taking Parallelism Mainstream

[Introduction: The Parallel Age 3](#_Toc223404267)

[Opportunities for Parallelism 4](#_Toc223404268)

[Challenges of Parallelism: The Hard Problems 5](#_Toc223404269)

[Expressing and Exploiting Fine-Grain Concurrency 5](#_Toc223404270)

[Coordinating Parallel Access to Shared State 6](#_Toc223404271)

[Testing and Debugging for Correctness and Performance 7](#_Toc223404272)

[Solutions: Parallel Technologies and Microsoft Visual Studio 2010 7](#_Toc223404273)

[Solution Overview 7](#_Toc223404274)

[Concurrency Runtime 8](#_Toc223404275)

[Libraries 9](#_Toc223404276)

[Parallel Pattern Library 9](#_Toc223404277)

[Asynchronous Agents Library 10](#_Toc223404278)

[Parallel Extensions 10](#_Toc223404279)

[Developer Tools 10](#_Toc223404280)

[Expressing Logical Parallelism with Programming Models 11](#_Toc223404281)

[Data Parallelism 11](#_Toc223404282)

[Parallel Loops 12](#_Toc223404283)

[Parallel LINQ 13](#_Toc223404284)

[Task Parallelism 13](#_Toc223404285)

[Parallel Invoke 14](#_Toc223404286)

[Tasks 14](#_Toc223404287)

[Dataflow Parallelism 15](#_Toc223404288)

[Futures 15](#_Toc223404289)

[Continuations 16](#_Toc223404290)

[Messaging Blocks and Asynchronous Agents 16](#_Toc223404291)

[Using Developer Tools for Parallel Programming 17](#_Toc223404292)

[Debugging 17](#_Toc223404293)

[Parallel Tasks 18](#_Toc223404294)

[Parallel Stacks 18](#_Toc223404295)

[Profiling 18](#_Toc223404296)

[CPU Utilization View 18](#_Toc223404297)

[Core Execution View 18](#_Toc223404298)

[Thread Blocking View 19](#_Toc223404299)

[Summary 19](#_Toc223404300)

[More Information 20](#_Toc223404301)

# Introduction: The Parallel Age

Personal computing has advanced considerably in the last 30 years. Exponential growth in processing power has transformed information management and personal productivity and has greatly expanded the capabilities of computers. The capacity to process more data, more quickly, and with rich graphical visualization has transformed business, science, and medicine. The capacity to render audio, video, and three-dimensional graphics in a highly networked environment has transformed entertainment, education, and communication. Until recently, applications like these gracefully grew faster and more responsive, scaling with the arrival of faster processors without requiring software developers to learn new programming paradigms.

However, the historical growth of raw sequential processing throughput has all but flattened as processor manufacturing approaches the physical limits of the materials science involved. Moore’s Law regarding the biennial doubling of transistor density continues, but the dividends of Moore’s Law can no longer be applied to continued dramatic increases in clock frequency. Instead, processors are now evolving under a new paradigm where multiple cores are placed on the same die to increase overall processor computational horsepower. Sequential applications that had previously benefited from faster processors do not see the same scaling as the number of processor cores grows.

*The software development industry is taking strides to make parallelism more accessible to all developers, and Microsoft is helping to lead the way.*

 To fully harness the scaling power of manycore systems, applications must be redesigned, decomposed into parts that can be computed in parallel, and distributed across available cores. But parallelizing code is not easy today given that the programming languages, frameworks, developer tools, and even the majority of developers, have grown up in a largely serial age. Today’s developers have generally been trained to write serial code—parallel code requires a different way of thinking about programming.

In response, the software development industry is taking strides to make parallelism more accessible to all developers, and Microsoft is helping to lead the way. In [*The Manycore Shift*](http://www.microsoft.com/downloads/details.aspx?FamilyId=633F9F08-AAD9-46C4-8CAE-B204472838E1&displaylang=en), Microsoft announced that it had established the Parallel Computing Initiative, which encompasses the vision, strategy, and innovative technologies for delivering natural and immersive personal computing experiences—harnessing the computing power of manycore architectures.

The Microsoft® Visual Studio® 2010 development system moves this initiative forward and provides a solid foundation for the parallel age. The significant improvements to its core platform and tools for both native and managed applications offer developers improved confidence, simplicity, and productivity in producing parallel applications. These innovative tools and technologies are the first wave of the long-term commitment Microsoft has made to help developers focus on solving business problems, scale their applications, and unlock next-generation user experiences in the face of the manycore shift.

In this white paper, we look first at the challenges associated with parallel computing and then focus on the Microsoft® solutions that address these challenges. We discuss how the new parallel programming abstractions introduced in Visual Studio 2010 make it possible to develop applications and libraries with abundant latent parallelism that scale well on today’s multi-core processors and continue to scale as core counts increase in the future.

# Opportunities for Parallelism

Parallel programming offers tremendous opportunities for scalability of both response time and capacity. With parallelism, a single instance of a large, complex problem can often be decomposed into smaller units and processed more quickly through the power of multiple cores working in parallel; alternately, many problem instances can be processed simultaneously on each of the multiple cores. As such, many “real-life” scenarios lend themselves well to parallelism, such as:

* *Business Intelligence (BI)*

BI reporting and analysis frequently use analytical procedures that iterate against large data models. Parallelizing these procedures—therefore distributing the work among multiple processors—increases the responsiveness of the algorithms, producing faster reports of higher quality. Since BI data often comes from a host of sources and business applications, using a parallel model that aggregates the data from the multiple sources lets users access the results more quickly. This, in turn, makes it possible for users to render data visually and to run additional “what-if” scenarios, ultimately leading to more relevant results and timelier decisions.

* *Multimedia*

Parallel computing can provide advantages to the next generation of multimedia processing systems. Current multimedia applications typically rely on sequential programming models to implement algorithms inherently containing a high degree of parallelism. Through the use of parallel computing techniques and multi-core processors, multimedia data can be processed by such algorithms in parallel, reducing overall processing time and enhancing user experiences.

* *Finance*

Parallel computing can lower risk in the financial sector by giving the user faster access to better information. For example, selecting representative stocks to mimic the performance of a larger financial index involves intensive optimization over a large amount of historical data. Parallelizing this process can thus provide dramatic performance gains. With parallel computing, it is possible to look at a variety of parameters as they change over time for a given set of financial instruments. Multiple instances of the same problem can be sent to the processing cores; as each core finishes its current simulation, it requests another.

Consider a foreign exchange currency trader who looks for arbitrage conditions (inefficiencies in the market) to make a profit. Such shifts are minute and quickly disappear as the market moves towards equilibrium, making very fast trades essential. Querying stock trade information using parallelism can enable close to real-time decision making, informed by large amounts of data and complicated analysis and computations.

# Challenges of Parallelism: The Hard Problems

*“We see a very significant shift in what architectures will look like in the future ... fundamentally the way we've begun to look at doing that is to move from instruction level concurrency to … multiple cores per die. But we're going to continue to go beyond there. And that just won't be in our server lines in the future; this will permeate every architecture that we build. All will have massively multicore implementations.”
Dr. Pat Gelsinger, Sr. VP, Intel Corporation and GM, Digital Enterprise Group, February 19, 2004, Intel Developer Forum, Spring 2004*

To make these scenarios possible, developers need to be able to productively build parallel applications that can be efficiently executed and can reliably share system resources. However, parallelism via the traditional multithreaded programming models available today is difficult to implement and error-prone for all but the most trivial applications.

To write effective parallel code, a developer must perform two key functions: identify opportunities for the expression of parallelism and map the execution of the code to the manycore hardware. Both functions are time consuming, difficult, and prone to errors, as there are many interdependent factors to keep track of, such as memory layout and load-balance scheduling. Furthermore, parallel applications can be challenging to test, debug, and analyze for functional accuracy, and they can frequently give rise to subtle bugs and performance problems that are unique to concurrent programs. The debugging and profiling tools that have evolved for building applications on a single-core desktop falter when facing such challenges in a manycore world.

Several challenges, or hard problems, must therefore be addressed before parallelism can be deployed more widely, including:

* How to express and exploit fine-grain concurrency
* How to coordinate parallel access to shared state
* How to test and debug for correctness and performance

## Expressing and Exploiting Fine-Grain Concurrency

The concurrency motivated by the manycore shift takes a single logical task and decomposes it into independent computations (subtasks), which can then be processed by the multiple processing cores. The basic function of the programming languages is then to describe tasks based on sequencing of subtasks. The opportunity to specify the concurrent execution of subtasks is a new programming paradigm; therefore, it requires a fundamental change in programming methodology. We call this “fine-grain concurrency” to highlight this deep change from other uses of concurrency such as managing asynchronous I/O.

Writing programs that express and exploit fine-grain concurrency is inherently more difficult than writing sequential programs because of the extra concepts the programmer must manage and the additional requirements parallelism places on the program. Chief among these concerns is the risk of unintended interactions between threads that share memory (“data races”) and the difficulties of proving that no such problems exist within a program.

To encourage code reuse and to maximize the benefits of parallelism, the programmer should ensure that the use of parallelism internal to the component is not part of the interface specification of the component. As new components are developed that are able to exploit concurrency, they can then replace the older components while preserving all other aspects of their behavior from the perspective of their use in client applications. Unstructured use of concurrency coupled with hiding parallelism in interfaces can exacerbate the problem of avoiding race conditions.

Additionally, since each component is free to present concurrency to the system that is potentially proportional to the size of the data (or other problem parameter), a hardware system might be pre­sented with more—possibly much more—concurrency than is needed to utilize the available resources. The developer (or application) must then be able to manage competing demands from the various components on the execution resources available, in addition to minimizing the overhead from any excess concurrency.

## Coordinating Parallel Access to Shared State

A second hard problem for parallel programmers involves managing the shared variables manipulated by the tasks within an application. Programmers need better abstractions than what is currently available to coordinate parallel access to application state; they also need better mechanisms to document and constrain the effects of functions with respect to application state. Incorporating the patterns of parallel computing into popular programming languages, such as C++, C#, and Microsoft® Visual Basic®, is not easy. Simply augmenting the languages with synchronization mechanisms like locks and event variables introduces new categories of errors not present in the base language (for example, deadlock and unintended dependence on the interleaving of accesses to shared state).

Any solution to the parallel computing problem must address three distinct issues:

* *The ordering of the subcomputations*. Many problems flow information through a graph following a topological order, a linear ordering of nodes in which each node comes before all nodes to which it has outbound edges. It is most natural to think of the concurrency in this problem as operators applied to all nodes subject to ordering constraints determined by the edges. Because there is no language support for this kind of pattern, developers today must resort to very low-level mechanisms.
* *The collection of unordered updates to shared state*. There are two key properties needed to support shared memory programming: atomicity, in which a transaction either executes to normal completion or has no effect on shared state, and isolation, in which the intermediate values in the shared state cannot be witnessed by nor modified by another transaction*.* Developers need to be able to provide these attributes; this lets them reason about the state of invariants based on the sequential effects of the code within a transaction, rather than on the various possible interleaving with other threads of control. Addressing this issue also provides support for a hard problem shared with sequential coding: recovering from exceptional conditions by restoring system invariants to allow continued processing.
* *The management of shared resources*. Frequently, there are shared pools of “buffers” that either represent real physical resources or virtual resources that are expensive to create; it is beneficial to have a bounded number of them shared as needed by subtasks within an application. Developers need to be able to manage these shared resources.

## Testing and Debugging for Correctness and Performance

A third hard problem is testing and debugging for correctness and performance. Concurrency puts additional demands on all stages of the application life cycle. Schedule-dependent results can potentially increase the space of possible executions exponentially, undermining the normal tools for coverage-based testing. Concurrency cripples the basic debugging technique of tracing backwards from failure and using repeated execution with incremental analysis of state to lead the developer to the fault. A further complication is the much larger control state of the system. With potentially hundreds of subtasks in progress, there is no clear definition of exactly “where” the program is or what its state should be at that point.

Parallel computing also introduces new dimensions to the problem of analysis and tuning of program performance, at the same time putting new emphasis on this problem. In addition to simple operation count, a parallel programmer must now worry about the amount of concurrency presented, the overhead associated with that concurrency, and the possibility of contention when concurrent activities must coordinate access to shared data. These are all new problems not faced by sequential programmers.

# Solutions: Parallel Technologies and Microsoft Visual Studio 2010

The transition to parallel programming mandated by the manycore shift presents both an opportunity and a burden to both developers and businesses. Without improved tools and technologies, developers may find they must deal more in the microcosm of creating parallel code than on the macrocosm of creating business value. The time that may have been spent in creating new user experiences may be siphoned off to solve concurrency issues. This reduces developer productivity and marginalizes their impact on a business’ bottom line.

In response, Microsoft delivers a solution with Visual Studio 2010, which draws on four goals:

* Offload the complexity of writing parallel code from developers to help them focus on solving business problems, thereby increasing their productivity.
* Simplify the process of writing robust, scalable, and responsive parallel applications.
* Take a comprehensive solution-stack approach, providing solutions which span from local to distributed computing and from task concurrency to data parallelism.
* Address the needs of both native and managed developers.

## Solution Overview

Microsoft Visual Studio 2010 confronts the hard problems of parallel computing with higher-level parallel constructs and abstractions that minimize the footprint on code and lower the conceptual barriers that complicate parallel development—helping developers express logical parallelism and map it to physical parallelism. Visual Studio 2010 also includes advanced developer tools that understand these constructs and provide debugger and profiler views that align with the way the parallelism is expressed in code. Figure 1 shows the parallel technologies included in Visual Studio 2010.

We begin with an overview of the solution components and then look at how they can be used.



Figure 1 The Microsoft solution

### Concurrency Runtime

Microsoft addresses the challenge of mapping the execution of code to the available multi-core hardware with the Concurrency Runtime, a standard runtime infrastructure that is well suited for the execution of fine-grained parallelism. The runtime has different concrete manifestations in managed and native scenarios, but its role is the same—to map logical concurrency to physical concurrency within a process. As work is queued for execution, the Concurrency Runtime balances the workload and assigns work across threads and processors.

To write parallel applications, a developer must both identify opportunities for parallelism and map the execution of the code to the hardware. These functions are challenging. With Windows® multithreading, developers must perform both of these functions independent of their counterparts—frequently in ad hoc, component-specific ways.

The Concurrency Runtime schedules tasks and manages resources, making it easier for developers to manage the physical underlying hardware platform. The Concurrency Runtime reduces the number of concepts exposed to developers so they can focus on building innovative and immersive user experiences, enabled by the processing power of manycore architectures.

With the Concurrency Runtime, the system handles the load balancing instead of leaving it to the developer. It allows the system to adjust available resources among competing requests dynamically, enabling external notions of priority and quality-of-service to be applied.

The Concurrency Runtime enables higher-level programming models such as the Parallel Pattern Library. Other programming models and libraries can directly target the Concurrency Runtime to take advantage of its extensive capabilities and to work with any other parallel frameworks used in the same application.

Third-party parallel technologies can use the Microsoft Concurrency Runtime as a common resource management and scheduling infrastructure on the Windows platform; third-party vendors can build on top of the Concurrency Runtime to provide their own interoperable solutions.

### Libraries

For constructing and executing parallel applications, Visual Studio 2010 includes new libraries for developing managed applications and for developing native applications with C++:

* Parallel Pattern Library
* Asynchronous Agents Library
* Parallel Extensions to the Microsoft® .NET Framework

#### Parallel Pattern Library

The Parallel Pattern Library (PPL) provides a parallel programming model abstraction for C++, making it easier for developers to build parallel components. The PPL exposes higher-level parallel constructs to native applications in the style of the Standard Template Library (STL) and takes advantage of features of the proposed C++0x language standard, such as lambda expressions. The PPL uses the Microsoft Concurrency Runtime to ensure the efficient and scalable use of system resources, and the PPL uses additional new support in the C Runtime Library (CRT) to ensure the proper propagation of exceptions. The library includes:

* *Concurrency Primitives and Algorithms.* A set of C++ primitives that provide high-level parallelism constructs for fine-grained parallel tasks. This includes parallel looping primitives, reduction primitives, and task primitives, which are an abstraction over threads.
* *Concurrency Containers.* A set of C++ containers that are building blocks for building parallel components including common synchronization primitives and data structures.

The PPL is primarily targeted at enterprise developers and independent software vendors (ISVs) who have existing investments in C++ applications and also at developers writing new C++ applications who want to introduce parallelism into these applications. One of the most common scenarios is the efficient parallelization of looping constructs, such as for and for\_each loops. Another is the parallelization of complex and computationally intensive algorithms by utilizing fine-grained tasks.

#### Asynchronous Agents Library

The Asynchronous Agents Library is a C++ template library that provides a high-level actor-based programming model and in-process message passing for fine-grained data flow and pipelining tasks. It includes:

* *Asynchronous Agents.* At design time, the actor based agent helps concurrent application development by encouraging developers to remove shared state and rely on message passing for communicating state changes. This has the side-effect of enabling better scalability by reducing the contention on shared state.
* *Messaging Blocks:* These enable the developer to coordinate state changes and realize additional concurrency within a process by building data flow pipelines of computations that are triggered by sending messages between the pipeline stages. Additional concurrency can be realized by using finer grained constructs from the Parallel Pattern Library within each stage of the pipeline.Concurrency Runtime.

#### Parallel Extensions

*“The need to develop code that runs effectively in multi-core environments is bringing a new set of challenges to the software developers. As processor core counts continue to increase over time, we need to work as a community to ensure the software development ecosystem evolves to exploit this new hardware reality.”
Earl Stahl, VP, Software Development, AMD*

Parallel Extensions to the .NET Framework is an addition to the core libraries of the Microsoft .NET Framework that exposes higher-level parallel constructs to managed applications. It includes:

* *Parallel LINQ.* A declarative model for data parallelism based on Language Integrated Query (LINQ).
* *Task Parallel Library (TPL).* An imperative model for both task and data parallelism based on explicit parallel patterns, such as tasks and futures, and parallel constructs, such as Parallel.For.
* *Coordination Data Structures (CDS).* A set of coordination and synchronization building blocks that simplify common communication and initialization patterns.

Parallel Extensions is targeted at developers using managed code (any .NET language, such as C#, Visual Basic, C++/CLI, and F#). Parallel Extensions makes it easy to automatically parallelize LINQ-to-Objects queries, empowering developers to achieve significant speedups with very few modifications to their existing and new code bases. The Task Parallel Library makes it similarly easy to introduce data and task parallelization into an application, whether through the parallelization of looping constructs or through the parallelization of more complicated algorithms utilizing fine-grained tasks

### Developer Tools

A key component of the Microsoft Parallel Computing Initiative is to simplify the construction of parallel applications by offering tools that let code developers manage the complexity, non-determinism, and asynchronicity inherent in parallel software. Microsoft provides diagnostic and optimization tools that operate at the same level of abstraction that developers use to write their code; this empowers them to understand the complex program state information associated with many concurrent activities, and it provides enhanced performance analysis functionality so developers can optimize software performance on multi-core hardware.

Microsoft has added new debugging toolwindows to Visual Studio 2010 that support task models in addition to traditional threading programming models. Visual Studio 2010 also includes profiling tools, which let developers analyze their parallel applications to measure the degree of parallelism within an application, discover contention for resources across the system, visualize thread distribution across cores, and drill into how much time is spent within the application executing program code versus waiting on synchronization objects, performing I/O, and more.

## Expressing Logical Parallelism with Programming Models

We now look at how developers can use the new programming models in Visual Studio 2010 to express logical parallelism, creating robust and efficient parallel applications.

Traditional multithreaded programming techniques require that developers not only define where concurrency is possible but also how that concurrency is to be achieved (such as through explicit thread management). With these techniques, concurrency is mandatory; parallelism will be used even if it is to the detriment of overall program performance.

Visual Studio 2010 introduces new, higher-level programming models for concisely expressing fine-grain concurrency, disentangling the parallel specification from the physical implementation. These models enable developers to invite (rather than demand) concurrent execution of their code. They rely on the Concurrency Runtime to efficiently schedule and execute their code on available cores.

*Microsoft Visual Studio 2010 introduces new, higher-level programming models that disentangle parallel specification from physical implementation.*

Visual Studio 2010 supports three major models for expressing logical parallelism:

* Data parallelism
* Task parallelism
* Dataflow parallelism

### Data Parallelism

We now consider the following problem:

Given an end-of-day market report for all stocks trading on the NASDAQ, find all stocks with market capitalization greater than USD $100B whose closing prices fell by more than 2.5 percent on volume 5 percent greater than average.

Here is what a sequential solution in C# might look like:

IEnumerable<StockQuote> Query(IEnumerable<StockQuote> stocks) {

 var results = new Queue<StockQuote>();

 foreach (var stock in stocks) {

 if (stock.MarketCap > 100000000000.0 &&

 stock.ChangePct < 0.025 &&

 stock.Volume > 1.05 \* stock.VolumeMavg3M) {

 results.Enqueue(stock);

 }

 }

 return results;

}

This problem is composed of subproblems that are independent of each other and can be executed concurrently. In this example, the subproblems are defined by *data decomposition* andthe unit of parallelism is an individual quote from the set of all stock quotes. The program here applies the same query function over each stock quote.

Visual Studio 2010 provides two techniques for expressing this type of data parallelism:

* Parallel loops
* Parallel LINQ

#### Parallel Loops

The parallel loops approach replaces for and foreach loops with explicit parallel loop constructs. For example:

IEnumerable<StockQuote> Query(IEnumerable<StockQuote> stocks) {

 var results = new **ConcurrentQueue**<StockQuote>();

 **Parallel.ForEach**(stocks, stock => {

 if (stock.MarketCap > 100000000000.0 &&

 stock.ChangePct < 0.025 &&

 stock.Volume > 1.05 \* stock.VolumeMavg3M) {

 results.Enqueue(stock);

 }

 });

 return results;

}

The principal difference between this and the sequential version above is that the foreach keyword has been replaced with a call to Parallel.ForEach and the body of the loop has become a lambda expression. An additional difference is the use of a thread-safe ConcurrentQueue<T> (found in the new System.Collections.Concurrent namespace of the .NET Framework 4.0) instead of a Queue<T>. Such constructs are provided for both managed (via Parallel Extensions to the .NET Framework) and native (via the Parallel Pattern Library) applications.

A native application, such as one performing matrix multiplication, can replace a for loop with a call to the parallel\_for template function. Here is what the sequential code looks like:

void MatrixMult(int size, double\*\* m1, double\*\* m2,double\*\* result){

 for (int i = 0; i < size; i++){

 for (int j = 0; j < size; j++){

 double temp = 0;

 for (int k = 0; k < size; k++) {

 temp += m1[i][k] \* m2[k][j];

 }

 result[i][j] = temp;

 }

 }

}

A possible parallel version looks like this:

void MatrixMult(int size, double\*\* m1, double\*\* m2,double\*\* result){

 parallel\_for (0,size,1,[&](int i){

 for (int j = 0; j < size; j++){

 double temp = 0;

 for (int k = 0; k < size; k++) {

 temp += m1[i][k] \* m2[k][j];

 }

 result[i][j] = temp;

 }

 });

}

This example also uses a lambda expression to automate the work of manually creating a function and capturing the variables used in the function.

#### Parallel LINQ

If the method is defined using LINQ in managed code, an even easier approach is provided; just add AsParallel to any enumerable collection:

IEnumerable<StockQuote> Query(IEnumerable<StockQuote> stocks) {

 return from stock in stocks**.AsParallel()**

 where stock.MarketCap > 100000000000.0 &&

 stock.ChangePct < 0.025 &&

 stock.Volume > 1.05 \* stock.VolumeMavg3M

 select stock;

}

The AsParallel extension method binds any enumerable collection to Parallel LINQ (PLINQ). PLINQ analyses the query, partitions the work to multiple processor cores to share the work, and yields a merged result. This provides performance and scalability improvements over many LINQ queries.

### Task Parallelism

Where the stock search problem implicitly created a number of tasks (where each task was involved in processing a distinct subset of all quotes), other problems are better expressed by explicit specification of tasks. One such problem is a common building block found in many multimedia algorithms for processing images, video, and sound: discrete Fourier transforms (DFT). An efficient algorithm for computing a DFT is a fast Fourier transform (FFT), and a common implementation of an FFT recursively decomposes a DFT into multiple smaller DFTs. Such an FFT is a specific example of a divide-and-conquer problem, others of which include visiting all of the nodes of a binary tree or sorting via the quicksort algorithm. The subproblems created by dividing the parent problem are often independent of each other and can be executed concurrently.

Microsoft Visual Studio 2010 provides two techniques for this kind of task parallelism:

* Parallel Invoke
* Tasks

#### Parallel Invoke

The Parallel Invoke construct lets the developer explicitly invite concurrent invocation of several actions. For example, in this naïve C++ implementation of quicksort, the algorithm partitions the array to be sorted and then recursively calls itself to sort each partition:

void quicksort(int \* a, int n) {
 if (n <= 1) return;
 int s = partition(a,n);
 quicksort(a,s);
 quicksort(a+s,n-s);
}

This can be parallelized with only a minor change:

void quicksort(int \* a, int n) {
 if (n <= 1) return;
 int s = partition(a,n);
 **parallel\_invoke**(

 [&]{quicksort(a,s);},
 [&]{quicksort(a+s,n-s);});
}

The recursive calls to quicksort are wrapped up in lambda expressions, and the expressions are passed to parallel\_invoke. As with the parallel looping constructs, this construct is available to both managed and native applications.

#### Tasks

All of the constructs demonstrated so far are higher-level constructs that help the developer create fine-grained tasks under the hood. In fact, these constructs employ another, slightly lower-level set of constructs called tasks. Provided by both Parallel Extensions to the .NET Framework and the Parallel Pattern Library, these task constructs make it possible for developers to create and manage individual tasks (as well as groups of tasks) while still capitalizing on the concurrency runtime for task scheduling and execution.

The quicksort example above can be modified to explicitly use tasks:

void quicksort(int \* a, int n) {
 if (n <= 1) return;
 int s = partition(a,n);
 **task\_group g;** **g.run**([&]{quicksort(a,s);});
 **g.run**([&]{quicksort(a+s,n-s);});
 **g.wait**();
}

This example is functionally equivalent to the version that uses parallel\_invoke. However, the explicit use of tasks permits more complex orchestration of actions than can be achieved with higher-level constructs like parallel invoke.

### Dataflow Parallelism

Some parallel programming problems can be greatly simplified by creating structure around the sources of parallelism and by specifying the computation structure separately from the processing of the individual elements. This structure can take the form of dependencies (specifying, for example, that a value or operation depends upon the availability of other values or the completion of other operations) or of message protocols (specifying that operations make progress in response to sending and receiving coordinated messages).

Consider a business intelligence (BI) scenario where customer data previously entered through kiosks is streamed into an application for processing. In order to limit bad entries entering the application, data mining techniques are used to provide a confidence level as to whether a particular customer record is likely valid or should be filtered out. Entries that make it through the screening are passed along to the next stage for additional processing. Such a pipeline is a prime example of dataflow parallelism. Stages in the pipeline may be executed concurrently on discrete data points (e.g. one customer record being validated while the next record is being loaded), and multiple data points may be processed in parallel by applicable stages (e.g. two customer records being validated at the same time).

Microsoft Visual Studio 2010 provides three ways to express dataflow parallelism:

* Futures
* Continuations
* Messaging blocks and asynchronous agents

#### Futures

Futures are a higher-level programming construct that helps developers define tasks that will produce values. Futures refer to an object that acts as a proxy for a result that is not yet known, usually because the computation of its value has not yet completed.

An example can be found in the pricing of stock options. The prices of the stocks are generated from a function (which may, for example, retrieve prices via a Web service) and encapsulated as futures. These futures are effectively tasks that, by contract, return a value. When the result value is needed, the caller retrieves the Result property of the future. If the operation producing the value has completed, the property will return the value immediately. If not, the caller waits until the value is available and may even assist in the computation of the result if applicable.

The operation encapsulated by a future can depend on the value of other futures. For example, the price of an option may depend on the price of one or more underlying instruments, each of which may depend on retrieving data from a network service or calling into a compute-intensive library. Futures can help model this scenario as a parallel dataflow. In the Task Parallel Library, a future is represented as a Task<TResult>.

The following code implements this example by generating a dictionary of stock prices that can be used for pricing options:

var symbols = new [] { "MSFT", "INTL", "AAPL", "GOOG" };

var prices = new Dictionary<string, Task<double>>(symbols.Length);

foreach(var symbol in symbols)

{

 prices.Add( symbol, Task.Factory.StartNew( () => GetPrice( symbol ) ) );

}

#### Continuations

Continuations are another higher-level construct that give developers the ability to define operations that will run when another operation is completed. Continuations can receive the result from their predecessor(s), and long chains of operations can be explicitly defined in code. For example:

var c = Task.Factory.StartNew(() => A())

.ContinueWith(a => B(a.Result))

.ContinueWith(b => C(b.Result));

Tasks (both those that do and don’t return results) can be linked together with continuations. Continuations can also be conditional, only executing upon success (or failure) of a predecessor. Continuations may also be predicated on multiple tasks, such that the resulting task will only be executed when any or all of the specified tasks have completed.

#### Messaging Blocks and Asynchronous Agents

The final form of dataflow parallelism is *asynchronous agents.* Asynchronous agents use messaging blocks to synchronize and exchange data. This structure supports a data-driven style between longer-lived persistent tasks, rather than the transient tasks in the examples above.

The following example uses messaging blocks and asynchronous agents to parse a file line by line. While a typical serial implementation would combine reading the file with parsing it, in this implementation the task of reading the file and parsing it are assigned to two distinct, concurrently executing agents. The first agent in the listing below, the parsing agent, waits for messages to arrive from the reading agent. The reading agent reads the file and sends each line as a message to the parsing agent, allowing for incremental, stream-oriented processing.

HRESULT LogChunkFileParser::ParseFile(){

    unbounded\_buffer<AgentMessage>\* MsgBuffer =

        new unbounded\_buffer<AgentMessage>();

    // the parsing agent

    AgentTask\* pParserAgent = new AgentTask([&]{

      AgentMessage msg;

      while((msg = receive(MsgBuffer))->type != EXIT)

      {

        Parse(msg->pCurrentLine);

        delete msg->pCurrentLine;

      }

    });

    //start the parsing agent

    pParserAgent->start();

    // the reading agent

    HRESULT hr = S\_OK;

    WCHAR\* wszLine;

    hr = reader->OpenFile(pFileName);

    if (SUCCEEDED(hr)){

       while(!reader->EndOfFile()){

          wszLine = new WCHAR[MAX\_LINE];

          // read the next line

          hr = reader->ReadLine(wszLine, MAX\_LINE);

          if (SUCCEEDED(hr)){

             //and parse it

             send(MsgBuffer, AgentMessage(wszLine));

          }

        }

        send(MsgBuffer, AgentMessage(EXIT));

        hr = agent::wait(&pParserAgent);

    }

    return hr;

};

The unbounded buffer template class allows the asynchronous agents to communicate with type-safe messages using a simple send/receive metaphor.

## Using Developer Tools for Parallel Programming

Visual Studio 2010 includes several enhanced tools to help developers as they transition to parallel programming. The goals are to:

* Let developers use the same nomenclature and abstractions in the tools as those used to write code.
* Let developers cope with the unique complexities of scale, non-determinism, and concurrency that exist in the parallel world.

### Debugging

All of the programming models described above use *tasks*, either directly or as the foundation for their implementation. A *task* represents a unit of user-defined work that is ultimately executed by a thread. Since tasks are so important to parallel programs, it does not make sense to force developers to make a mental jump back to threads when they are debugging. Therefore, two new toolwindows are added to Visual Studio 2010 that make tasks a first-class citizen in debugging:

* Parallel tasks
* Parallel stacks

#### Parallel Tasks

The Parallel Tasks toolwindows display the tasks created by the application and show whether they are running, they have run and are now waiting on a resource, or they have not run yet but are scheduled to do so. It also shows additional information such as the thread that is executing the task, the parent-child hierarchy, and the task’s call stack. This view enables the developer to understand the system load and current execution patterns.

#### Parallel Stacks

With the Parallel Stacks toolwindows, a developer can see at a glance all the call stacks for all tasks or threads in the application. This graphical view builds on familiar concepts of the call stack window and enhances it, not only by expanding the focus from one execution context to multiple, but also by visually indicating which methods are shared by which tasks or threads. The graph is interactive; the developer can use it to quickly change the focus area of debugging, and the rest of the development environment (such as the editor and other toolwindows) will reflect the change.

The new debugger toolwindows support both managed and native task models as well as traditional threading programming models.

### Profiling

With Visual Studio 2010, developers are able to analyze their parallel applications to measure the degree of parallelism within an application, discover contention for resources across the system, visualize thread distribution across cores, and drill into how much time is spent within the application executing program code versus waiting on synchronization objects, performing I/O, and more. In particular, Microsoft Visual Studio 2010 offers three new profiler views:

* CPU utilization view
* Core execution view
* Thread blocking view

#### CPU Utilization View

The CPU utilization view makes it possible for developers to determine the phases of an application that are CPU-bound, their relative durations, and the level of system core utilization. Developers can then zoom in on those phases and examine the forms of parallelism that exist and how they may be used. During the performance-tuning phase, this view can confirm whether the concurrency expected is indeed observed during execution.

Basic timing measurements can be made to estimate speed improvements achieved, compared to a previous or sequential execution. The concurrency view also gives the developer an idea of the interaction between the application in question and other processes in the system.

#### Core Execution View

The purpose of the core execution view is to show the mapping of threads to cores during a given profiling session. This information helps a developer to conceptualize the execution of their application over time and uncovers instances of thread migration. Thread migration occurs when a thread is moved from one core to another and is a resource-intense operation that can impair overall application performance. By examining thread state across the duration of the trace, the user can spot migration and tie it back to specific delays using the thread blocking view.

#### Thread Blocking View

Once a phase of interest has been identified using either the CPU utilization or core execution views, the developer can then further analyze the behavior of the application using the thread blocking view*.* This view provides a wealth of information about the behavior of each thread in the application.

First, it generates a bar graph with an execution breakdown of each thread, depicting the fraction of each thread’s lifetime that was spent executing code or blocking. Blocking delays are further broken down into various categories, such as I/O or synchronization.

 *“After decades of single core processors, the high volume processor industry has gone from single to dual to quad-core in just the last two years. Moore’s Law scaling should easily let us hit the 80-core mark in mainstream processors within the next ten years and quite possibly even less.”
—Justin Rattner, CTO, Intel, February 2007, Integrated Solid State Circuits Conference*

Second, the view provides a timeline visualization with time on the *x*-axis and threads and physical disk I/O as lanes on the *y*-axis. For disk I/O, the view shows reads and writes as they occur in time on disk and shows the files being accessed. For threads, the view shows when they are executing, when they are blocked, and the category of delay, using color to convey thread state. Measurement and marker features show the use of the new programming constructs made available by the managed Parallel Extensions to the .NET Framework and native Parallel Pattern Library. For example, an application’s use of a parallel for loop is displayed on the timeline to indicate the duration of the loop in the profile trace.

Third, call stack analysis lets the developer pause on any blocking segment in the timeline view to understand in detail what each thread was doing when it was blocked. For example, the view can show the call stack (including source code line numbers) that resulted in a blocking attempt to acquire the critical section. Reports can be generated that include detailed execution, blocking, and disk I/O statistics.

# Summary

Future improvements to software performance will largely hinge on software being able to take advantage of the proliferation of manycore processors through parallel computing. The software development industry is taking strides to make parallelism more accessible and feasible for all developers, and Microsoft is helping to lead the way. The comprehensive solution stack approach from Microsoft addresses manycore parallelism from operating system to applications. With Visual Studio 2010, Microsoft is delivering the first wave of powerful developer solutions- including a Concurrency Runtime and associated programming models, libraries, and tools that ease the transition to parallel programming. These technologies will enable the development of applications and libraries with abundant latent parallelism that will scale well on today’s multi-core processors and continue to scale as processor core counts multiply in the future.

# More Information

Callahan, D. (2008, October). *Paradigm Shift: Design Considerations For Parallel Programming*. Retrieved October 9, 2008, from MSDN Magazine: http://msdn.microsoft.com/magazine/cc872852

Duffy, J., & Essey, E. (2007, October). *Parallel LINQ: Running Queries on Multi-Core Processors*. Retrieved October 9, 2008, from MSDN Magazine: http://msdn.microsoft.com/magazine/cc163329

Microsoft Corporation. (n.d.). Retrieved October 9, 2008, from Parallel Programming with .NET: http://blogs.msdn.com/pfxteam/

Microsoft Corporation. (n.d.). Retrieved October 9, 2008, from Parallel Programming in Native Code: http://blogs.msdn.com/nativeconcurrency/

Microsoft Corporation. (2008, June 2). *Microsoft Parallel Extensions to .NET Framework 3.5, June 2008 Community Technology Preview*. Retrieved October 9, 2008, from Microsoft.com: http://www.microsoft.com/downloads/details.aspx?FamilyId=348F73FD-593D-4B3C-B055-694C50D2B0F3

Microsoft Corporation. (n.d.). *Parallel Computing Developer Center*. Retrieved October 9, 2008, from MSDN: http://msdn.microsoft.com/concurrency/

Microsoft Corporation. (2007, November). *The Manycore Shift: Microsoft Parallel Computing Initiative Ushers Computing into the Next Era.* Retrieved October 9, 2008, from Microsoft Corporation: http://go.microsoft.com/fwlink/?LinkId=124050

Toub, S., & Shafi, H. (2008, October). *Coding Tools: Improved Support For Parallelism In The Next Version Of Visual Studio*. Retrieved October 9, 2008, from MSDN Magazine: http://msdn.microsoft.com/en-us/magazine/cc817396.aspx

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2009 Microsoft Corporation. All rights reserved.

Microsoft, Visual Basic, Visual Studio, and Windows are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.