From Sequential to Parallel

Ranjan K. Sen, Microsoft Corporation
Pradip Peter Dey, National University

Abstract

There is an urgent need for traditional programmers who write programs meant for a single processor to be able to modify or write new codes for multiple processor machines. Unfortunately, writing parallel programs has been esoteric, considered too complex and had been confined to a few developers. This paper presents Microsoft .NET framework versions and enhancements that provide modern approaches that are relatively easy for traditional programmers to learn rapidly and use. These new technologies provide support for both shared memory as well as distributed memory parallel programming models such as a decentralized software service model. However, the presentation of these technologies needs to build on the general background of the stored programmed memory model used by the traditional programmer. It needs to add structures that enable them to incrementally move from sequential model to the parallel model of computation. The goal of the paper is to present a comprehensive structure that integrates the parallel programming model with the sequential model and introduces the technologies in this context. The paper argues that a second level programming course should be based on such an approach. Such a course will be useful for students as well as professional programmers who need this new skill in the light of more and more multi-core, many-core and cluster based commodity multi-computing.

1. Problem statement

Parallel programming has traditionally been a highly specialized area of programming. Interestingly a very limited aspect of parallel processing is often known to traditional sequential programmer. This is multi-threading and is used to improve response time by off loading slow computations such as input/outputs to a thread different from the main thread. However, in parallel programming one is interested to distributed data/code to run concurrently as well as ensure proper coordination to ensure the sum total of these computations produce the result correctly. This requires a more sophisticated programming model and development approach suited for the purpose.

Due to diverse hardware architectures, environments and approaches the meaning of parallel programs incorporated features that ranged from instruction level, data array to higher level concurrency. This has resulted in different models such as data parallel, task parallel, asynchronous, dataflow, agent based as well as distributed parallel programming. All of this has added to the complexity of parallel programming. The good news however is that considerable experience in parallel programming has taken place over the last more than 4 decades and a rich body of knowledge is available. The bad news is parallel programming looks very complex to most sequential programmers.

There has been recognition of the fact that it is urgently needed to offer training in parallel programming to students in general and pro developers as well. Most of them however, have hardly looked at a bridging approach where a sequential programmer is gradually moved to the world of

parallel programming. For example, the course proposed in [1] includes all the aspects of parallel programming such as algorithm design, architectural constraints, programming approaches such as SPMD, mapping and scheduling as well as parallel databases. Students use parallel programming languages and tools such as OpenMP [2, 3, 4] for C/C++ with MS Visual Studio 2005 and Intel C/C++ compiler. Debugging tools include VTune, and threading tools include Intel Thread Profiler and Intel Thread Checker. However, it ignores two important issues –

1) Limitations to "think parallel" without support from a suitable environment
2) The scores of existing developers with developing programming skills for conventional sequentially executing computers.

Our contention is that the approach should be that a sequential (one processor) computation model, as exemplified via conventional sequential programs, is not being replaced by parallel computation model but simply being extended. The approach we need is to add parallel programming as an extension to existing courses on sequential programming and architecture.

We believe that point 1) will be address by building a connection to the common stored program model used in sequential programming from parallel programming models. Point 2) will be addressed by programming extensions (implemented possibly in multiple ways).

This paper presents a proposal for developing a second level course in this approach. Such a course may be part of the engineering curricula while being available to professionals desiring to upgrade their skill sets. This course is intended to demonstrate that using appropriate tools and environments sequential programming ideas can be broadened in order to progressively reduce the cognitive gap between sequential and parallel program development.

As a starting point, in section 2 we identify basic concepts all programmers seem to be aware of. We highlight aspects of computing model, languages and system as well as idea of multi-threading. In section 3 we present the concepts in parallel computing and discuss how they relate to concepts identified in the earlier section. In the next section we further explain the parallel computing concepts emphasizing the role of coordination. Finally in the following section we present the outline of a course designed in this approach.

2. Concepts programmers know

The most widely used model for sequential computation is the von-Neumann stored programming model. This model consists of a processor, a program, data and memory. The program is a sequence of instructions that the processor executes when directed by a program counter/instruction pointer. There are various forms of memory such as registers, random access memory, and secondary memory etc. There are external devices that can communicate to the processors.

An algorithm is a high level representation of a program. It consists of steps of statements that is unambiguous and may be interpreted by a computer. An algorithm must terminate. An algorithm is

used to reason about a specific recipe to solve a problem using a computer. It is implemented as a program using programming language.

A programming language is one that can express an algorithm and may be compiled into a low level program that may be directly executed by the computer hardware. The computer hardware recognizes programs stored in the memory and executes it instructions one by one as pointed to by the program counter. One instruction is executed at a time by the single processor.

2.1 High level language

Programmers write programs using a high level language such as C#, C++ etc. these languages allow various programming structures to help developing programs, reusing existing programs or its parts, composing programs using parts, ensuring semantic guarantee (delivers what it supposed to mean all the time), … The notion of assignment, condition checking, read/write, mathematical operations, data and types of data are very natural to human understanding of computational domain. Composite data such as collections, containers are common. Modern programming environment provide support for using multiple programming languages, using run time just in time compilation that optimizes use of the available infrastructure and various other development tools. Note that the underlying programming model provided is still the stored program model.

2.2 Operating Systems

Programmers understand the role of the operating system in managing computing resource among multiple users. They often use operating system services programmatically. For example, in dynamically allocating memory it uses the memory allocation services, in acquiring a network port it uses socket service of underlying network system.

2.3 Data and Database

Programmers recognize the value of associating a type with a data, composing data in databases for efficient storage and retrievals. They are aware of the mechanisms used for opening database, connecting to it, querying for information etc.

2.4 Network and distributed processing

Network is how computers connect to each other and other devices of various types. They know the different protocols used in such communications. They are aware of ports that are the interface to a computer and machine network addresses.

2.5 Multi-threading

One important understanding in this context is the role of a processor which is the key element of a computer. Mechanism for offloading processors computational load via threads is quite well known. This is to avoid long waiting time for an interactive user. This requires dividing into several pieces of

computations and maintaining their state. This type of support is provided by most operating systems today.

3. Parallel processing concepts

Parallel processing may be viewed as several computations taking place at the same time or concurrently. The notion of "same time" needs a bit of a clarification. If the state of computations is maintained in a computer with only one processor so that any of them may be started, halted before completion, another computation that might have been halted restarted then we are doing parallel processing. In case of multiple processors it is possible to have two or more computations actually running simultaneously.

Overall the notion is to be able to declare computational tasks that may be scheduled to available processors. Each computational task runs on a single processor when it is scheduled to it. Thinking in this way, a parallel process is the agglomeration of tasks. Each task corresponds to a program or a segment of a program that runs on a single processor in a sequential manner. However, such a task may be stopped and saved unfinished to allow another task to use the processor if necessary. The stopped task may resume in the future on this or another processor.

The overall conceptual approach is given by:

Parallel program = Sequential Program + coordination (communication)

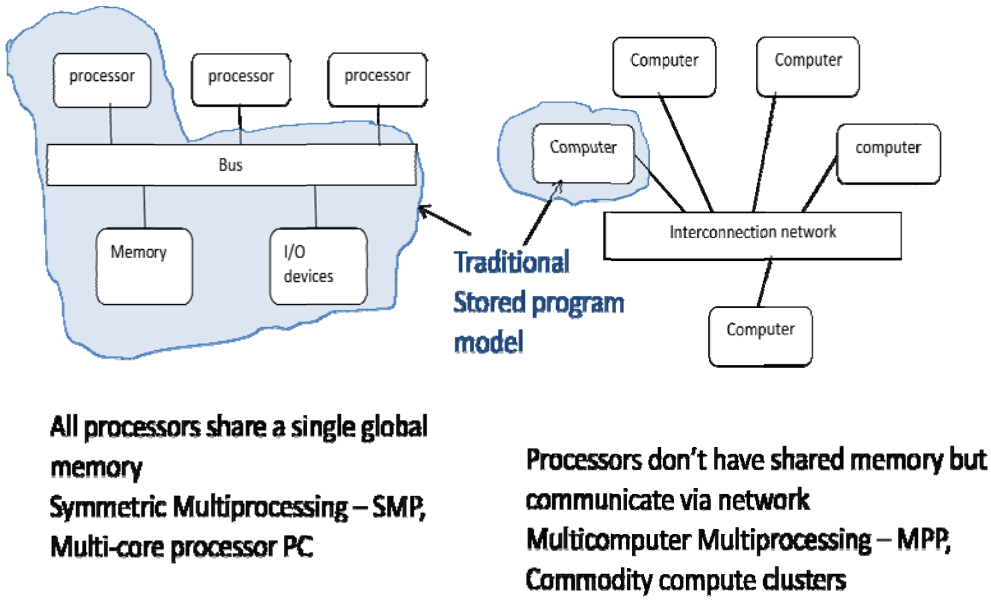
Programs process data. The basic goal is to produce an output from input via steps of computations; some of which can take place concurrently in parallel programs. We can analyze the overall process and determine the input-output dependencies amongst the sequentially executing instances. The input of one instance may be generated as output of another. Similarly more than one instance may need the same input. Some of the computational operations on such data are safe if performed by one instance and not so, if performed by several at the same time. This is the limitations associated with the underlying hardware commonly available. In parallel programming we get around this issue by a coordination discipline. One common type is programming mechanisms such as locks and semaphores.

3.1 Coordination

The new concept learners need to understand is that of coordination because they are already aware of concepts in sequential program. One can think of multiple programs running at the same time where each may potentially impact behavior of the other directly (via altering common data) or indirectly (by blocking a resource resulting in the other programming not able to proceed or in deadlock) unless the input-output relations is ascertained to be correct from both a spatial as well as temporal point of view. The first is enforced easily in conventional memories (Random Access Model) but the later requires ordering discipline generally implemented via synchronization primitives similar to those used in operating system design.

3.2 Use of programming model

Parallel computer programming models incorporated the idea of coordination via memory or message passing operations. Often these are too low level and lack the high level structure and pattern that is commonly shared across parallel applications. The two corresponding models are the shared memory and the distributed memory models of parallel programming.



The shared memory model is also known as the symmetric multiprocessing or SMP where cost of data access by different sequential program instances of a parallel computation are the same. This assumption is not necessarily always valid, say when intermediate memory caches are used to cut down on memory access overhead.

With more and more such instances the contention for memory impacts this cost limiting scalability of the parallel program. This prompted distribution of the memory in the distributed memory model. However, this requires multi- staging of memory using caches and external networking. Additionally, data access from memory often requires explicit network communication. This model is also called the Multi-computer Parallel Processing or MPP. Language extensions and libraries to support these models are well known – Open MP [2-5] for shared memory while MPI [5, 6] and PVM [7] for distributed memory models. MPI has become very popular as it is easily available to run on commodity clusters of computers. With the advent of multi-core and multi-computers, hardware accelerators such as FPGA, graphic processor matrix a complete overhaul of software development to move to parallel processing is warranted today.

3.3 Languages and libraries:

Most programming languages support "threads" which is the unit of computation scheduled by operating systems. Windows provide API for using thread objects; UNIX pthreads library offers similar functionality. Developing and maintaining parallel programs at the thread level is low level. Language

structures suitable for parallel processing based on appropriate programming models are known. Parallel version of popular languages such as FORTRAN, C, C++ were developed and used. Unfortunately, these tended to be proprietary and expensive. Approaches based special compiler directives as in OpenMP, on libraries such as MPI and PVM were more popular and are currently widely used. The libraries are available from more than one programming languages.

Among other popular libraries are the TBL from Intel, AMD. Among compilers is the Cilk++ compiler which is an extension of industry standard compiler, the native extensions and optimization of the base compiler is fully supported. The underlying model is the shared memory model in all of these except for MPI and PVM which are based on distributed memory model.

4. Extending concepts to include parallel processing

Although considerable research and development has taken place in the area of parallel processing during the last several decades the idea of a novel parallel processing language never succeeded. The approach that worked has been that taken up by Open MP and MPI/PVM. One concern is though that both avoids calling out a concept of "computation" as a distributable object as a first class entity. This has been to avoid intruding into common language standards and concepts. In particular the languages did not provide appropriate options to build such an object.

Going back to the equations used above and reprinted here for convenience, we draw attention to the role of coordination/communication in design and development of parallel programs.

Parallel program = Sequential Program + coordination (communication)

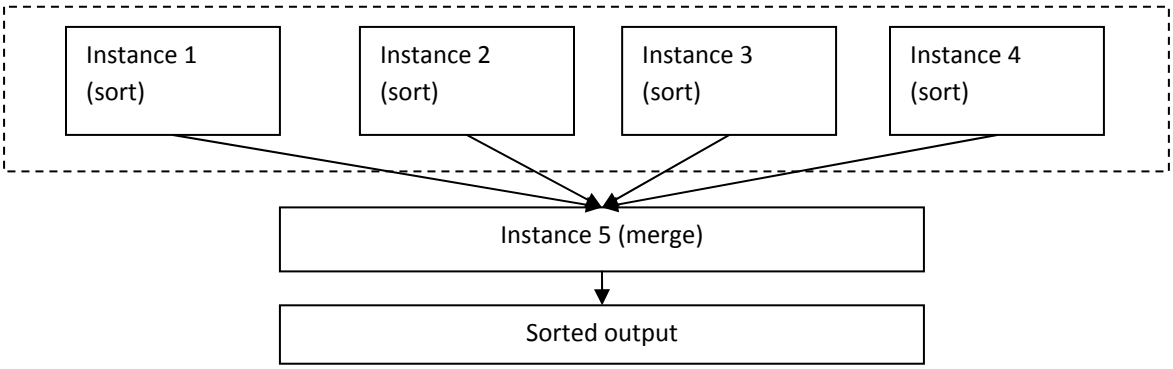
We need a first class representation of a sequential program as a computation in the language or its extension.

Looking at the topic of coordination observe that this depends upon the programming model to be used. For example in a shared memory model all coordination needs to happen by coordinating access to the shared memory. Alternately, if the model is a distributed memory model then such coordination is completely done via message passing often in an asynchronous manner.

This needs the new concept of coordination in shared memory and via message passing to be learnt. Considerable knowledge in both areas, albeit in different context, has been known to us. The goal is to identify them and present them in a unified and structured manner. This will be discussed in the next section.

At this point let us consider the simple example below of sorting a set of numbers:

Assuming parallel instances of several sequential sort programs we can visualize the enter operation as that of sort followed by merging of the sorted parts. This imposes input-output dependencies between the instance executions, shown with arrows.



Note that there is no need for direct coordination between the instances performing sort, while the merge instance may only execute after all sort operations are completed.

4. 1 Contrasting sequential and parallel algorithms

Sequential algorithms use three alternate models for algorithm analysis – RAM, RASP and the Turing machine. Parallel algorithms are essentially extension of a sequential algorithm model. The main ones for shared memory models are PRAM (Parallel RAM) and their variations such as EREW, CREW, and CRCW based on support for concurrent access to memory. Similar models are available for analysis of parallel algorithms targeted for distributed memory parallel computers. Excellent treaties on this is in [13]

In the above example, a PRAM model based algorithm is:

1. Do in parallel { Instance 1, Instance 2, Instance 3, Instance 4}
2. Merge sorted outputs from Instance 1, Instance 2, Instance 3, Instance 4 to create Instance 5
3. Print out the sorted output

Each instance in step 1 uses a sequential sorting algorithm that sorts one of four blocks of the total data to be sorted. The step 2 involves coordination of outputs produced by the four instances of step 1. It is evident that the characteristics of the algorithm depend on size of the blocks, speed of sequential computations, speed of coordination etc. For example, the time taken for step 1 is given by the maximum time spent by an instance of the four being executed. That of step 2 is given by the merging scheme etc.

Several coordination primitives and patterns are used in parallel programming. It is important to understand which should be used when. In the context of shared memory model most common primitives are those conventionally used for multi-threading and known in the OS context. This is quite known to traditional sequential developers as well as synchronization primitives.

Higher order synchronization and coordination primitives such as monitors, semaphores etc are built with lower ones such as locks. Principles and structures such as mutual exclusion, reader-writer paradigms are build from smaller primitives. Updating a variable in memory concurrently may lead to

unpredictable or non-deterministic value. This is due to race between concurrent operations that modify it. Locks are used to coordinate access to the variable by concurrent operations such that only one may deterministically modify it. This idea may be extended to larger data structures common in programming – such as stacks, queues, lists etc.

Patterns such as reduction … and generic schemes such as “map-reduce”, “data-parallel”, “task-parallel” are even higher level coordination schemes. These are actually the parallel algorithms we mentioned before. In the data parallel context sequential programs run on independent sets of data concurrently and there is no coordination necessary between them. Task parallel scheme defines independent computational blocks, known as tasks where having maximum fan-out in terms of maximum number of tasks to run concurrently lead to maximum speed up.

a) Task parallelism

   Tasks can execute concurrently when there are no dependencies among them. Examples include “ray-tracing” in image processing and Monte Carlo simulation programs. In some cases, it is possible to modify data and instructions to create independent tasks and engage task parallelism.

b) Divide and conquer

   This is similar to the parallel reduction scheme used for developing the maximum finding program. This general scheme is very popular in developing sequential as well as in parallel algorithms. In case of parallel algorithm the goal is to decompose to an extent that makes sense for the individual processors.

c) Geometric Decomposition

   In this approach the relationship of the data influences the pattern. For example, in computational fluid flow problem, operations on matrix element require only neighboring data. In such a case, the matrix geometry is used to formulate tasks. Also in matrix multiplication checker-board algorithms use geometric relationship.

d) Recursive data pattern

   Many sequential algorithms use recursive processing. In many cases these can be split into partitions. Processing recursively can occur in parallel across these partitions. Overall computation can be formulated as a sequence of parallel recursive operations followed by synchronization.

e) Pipelined

   A sequence of tasks can be executed in such a way that a processor is kept working all the time. For example, in case of executing a for-loop in a program, if there is no data dependency and

resource conflict, the first instruction (or task) of the n+1th loop can start executing immediately after the first instruction (or task) of the n-th loop has completed.

4.2 Task and data parallelism

When we partition the computation into tasks and execute them in parallel we get task parallelism. In this case data is not partitioned. Flynn's taxonomy gives four variations in task parallelism of which three are useful – Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD).

When we partition data but not partition the tasks we have data parallelism. Data parallelism is a convenient way to introduce parallelism when operational dependency does not involve the entire data. This approach is very popular and gives rise to a specialized model of the distributed memory parallel computer called Single Program Multiple Data (SPMD).

4.3 Role of Synchronization

A key concept in coordination is synchronization. This process enforces a synchronous mode of computation in which (the time of) a set of computations are hidden by time period. The intent is to ensure all computations are completed when that period or "clock" cycle is complete. This is a way to impose deterministic state properties. This is very close to the shared memory model. A popular variation in array processing is the systolic architecture.

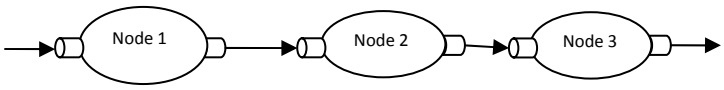
In a computation that does not use synchronization via clock; or in an asynchronous mode of computation, instructions may interact via generation of data or events. One common methodology known to conventional sequential programmers is to use events and call backs. Dataflow is a well known asynchronous parallel computing model. Other models include demand-flow or lazy-evaluation where a computation is performed only when its results are asked for or demanded.

Coordination algorithms may be implemented imperatively in code, in hardware network such as shuffle-exchange network, gate arrays, and distributed services and so on. Much of the coordination is embedded in the structure and topology of the network itself.

4.4 Network and arrays

Parallel coordination algorithms are implemented in arrays via simple messages or dataflow across them. In case of arrays of identical processors performing similar computations pipelined patterns are easily incorporated.

An algorithm for sorting on a linear array of three processors is given below. Note how the coordination scheme is somewhat embedded in the topology of the linear array. Note each node of the linear array sorts a block of data. Each node has an input and output port that establishes the linear relationship among them. We assume N data is being sorted, each node perform a sequential sort on N/3 using a simple insertion sort scheme.



Initially each node reads elements from its input and merges them into an internal sorted sequence of N/3 data.

1. Post N data at the input of node 1
2. Each node do in parallel {

   Read data from input and create sorted sequence locally;
   Pass data smaller than the smallest element in the sorted N/3 sequence to the output port

   }

3. Output sorted sequence

A systolic approach can clock the transmission of data across the network. A service oriented scheme based on CCR/DSS parallel technology supports this in an asynchronous parallel processing scheme.

4.6 Speedup and performance

A key point in parallel processing is performance. Speedup is given by the ratio of time taken by sequential program to the time taken by parallel program. Theoretical limits given by Amdahl's law or Boris-Gustafson's law are well known. It is important to understand concepts of speedup, and performance scalability.

This requires one to be able to understand the algorithmic issues governed by patterns, platform issues governed by the hardware and operating system that impact performance. One should be able to relate performance to code via tools such as profiler, which is rarely used in developing sequential programs.

Debugging parallel programs is difficult. This requires learning to reason about program behavior under multi-processing environment as well as knowing appropriate tools.

Aspects of availability, reliability and fault-tolerance are somewhat advanced topics. However, they should be related to the integrated approach we discuss here.

5. Foundation course

We present an high level outline of a course that help introduce parallel programming to students that have completed one or more traditional programming course. It is useful to professional programmers who have experience in programming single processor computers. The items below point to key topic areas that should be covered in such a course. It also gives relative hours to spend with a percent value for each topic.

- Executing code on a thread – understand the concept of computation in conventional programming environment.
- Developing algorithms – how a program has an underlying algorithm. How to look at an algorithm to reason about a program – correctness, performance and boundaries (max input size, type of input to process, any data ranges, accuracy of output etc.)
- Understanding parallel algorithms – sequential programs and coordination. Extending a sequential program into a parallel program. Identify sequential pieces, coordination process, any extension of boundary (improving performance – processing more data quicker, improving accuracy of data or range of data etc). Concepts of speed up and scale up. Ideal case of linear speed up, super-linear speed up etc. Compute to communication ratio limits speed up.
- Abstracting coordination – patterns. Consider other sequential programs and attempt to parallelize. Look at a good number of problems and identify patterns. At least 5 to 6 different types of patterns need to be discussed.
- Concurrency models – shared memory and distributed. Approach to developing parallel program. Type of model to choose and what type of programming languages etc.
- Scheduling and mapping – how does computational resource usage relate to parallel programming

We believe that the Microsoft .NET Framework and the latest Visual Studio 2010 (CTP available) includes rich set of parallel technology that can be used to develop such a course. This has been presented in Appendix I.

Appendix II illustrates features of the Microsoft .NET Framework parallel processing technology in providing a modern, rich approach to present the course in the above approach. It allows efficient delivery of the concept and approach of the course outlined. This is because we can quickly demonstrate how patterns such as data parallel, task parallel or service oriented design can be formulated in a modern programming environment around C# and Visual Studio. The availability of high quality development, debugging, profiling in the Visual Studio makes delivery of this course more efficient than other alternatives as understood by the authors. The goal is to provide the concepts and theories with complementary hands on experience. The balance can be only maintained with such a modern programming environment.

6. Conclusions

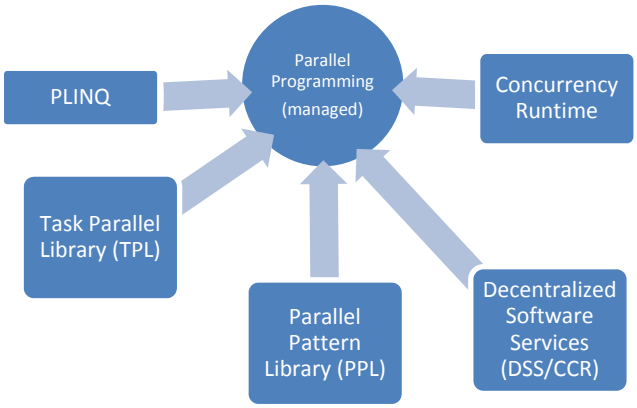
In summary the approach we propose is based on existing programming knowledge and practice of traditional sequential developers. The relationship between sequential and parallel programs/computations is a key underlying theme. The issue of parallel programming is viewed mainly as that of framing up the coordination patterns among sequential pieces. The area of sequential and parallel algorithm analysis and design is very well developed. In most Computer Science and Engineering curricula the first is usually covered. Correctness, performance and issues such as reliability, availability need to get due coverage. Tools for debugging and profiling can be introduced as part of the curriculum in this context.

References

1) *Think Parallel: Teaching Parallel Programming Today,* Ami Marowka, Shenkar College of Engineering and Design, Israel, 2008 http://dsonline.computer.org/portal/site/dsonline/index.jsp
2) Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., & Menon, R. (2000) *Parallel Programming in OpenMP*, Morgan Kaufmann, 2000.
3) *The OpenMP API specification for parallel programming*, http://openmp.org/wp/
4) Chapman, B. , Jost, G., & Van der Pas, R. (2007) *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press, 2007.
5) Quinn, M.J. (2004) *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, 2004.
6) Karniadakis, G., Kirby II, R. M. (2003*) Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation*, Cambridge University Press.
7) Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V. S. (1994) *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing*, MIT Press, 1994.
8) M. Paprzycki, *"Integrating Parallel and Distributed Computing in Computer Science Curricula," IEEE Distributed Systems Online*, vol. 7, no. 2, 2006, http://doi.ieeecomputersociety.org/10.1109/MDSO.2006.9.
9) I. Foster, Designing and Building Parallel Programs, Addison-Wesley, 1995, www-unix.mcs.anl.gov/dbpp/text/book.html.
10) *Cilk Arts:* http://www.cilk.com/multicore-products/cilk-solution-overview
11) *The Manycore Shift: Microsoft Parallel Computing Initiative Ushers Computing into the Next Era (2007),* http://www.microsoft.com/downloads/details.aspx?FamilyId=633F9F08-AAD9-46C4-8CAE-B204472838E1&displaylang=en
12) *Overview of Synchronization Primitives* http://msdn.microsoft.com/en-us/library/ms228964.aspx
13) T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays:Trees:Hypercubes* - Morgan Kaufmann, 1992.

Appendix I: Microsoft Parallel Processing Technology

Recent .NET Framework, 3.0 and later versions of .NET Frameworks support more than one high level parallel processing technologies (see figure). These provide the high level programming model missing in the legacy thread models.



The DSS/CCR runs in .NET Framework 3.0 (and 3.5) runtime. It provides an asynchronous message oriented programming model. This model helps develop a loosely coupled multi-task computing solution. It is available as a DSS/CCR development kit and also as part of the Microsoft Robotics Studio[1].

The remaining technology, available as libraries and core runtime, is part of the Microsoft .NET Framework 4.0. A Community Tech Release (CTP) for Visual Studio 2010 contains this.

PLINQ allows automatic parallelization of LINQ queries. The Thread Parallel Library (TPL) is available to be used for managed development in C#, and VB. The Parallel Patterns Library (PPL) as well as the asynchronous agent technology provides support for parallel development for native C++. The Concurrency Runtime provides the runtime to host the parallel objects. Additionally, new profiler tools are available for parallel programs.

The Windows HPC Server

The latest release of Windows HPC Server is based on .NET Framework 3.0 and provides a top notch modern and highly secured HPC run time environment. There is a Window HPC Pack that provides high quality administration, management, monitoring tools along with a top notch job scheduler. The Windows HPC Server data sheet, doc, case studies and licensing is available (see technology links table). The Windows HPC server is a supercomputer quality product as evidenced by the 10th rank in the international top500 list. It provides all traditional HPC application development support and has Microsoft MPI library based on MPICH2. This makes easy port of tradition MPI applications.

[1] CCR and the Asynchronous Programming Model (ref. http://msdn.microsoft.com/en-us/library/bb977678.aspx)

Additionally, it provides latest features such as SOA job definition and scheduling in addition to standard options of diverse schedule policies, integration with third-party scheduling services and the international grid computing standards such as HPCBP. It integrates Windows management and monitoring services leading to fast deployment support via Windows Management and Operations (MOM); system health reporting via Windows System Management Services (SMS) leading to high productivity computing in addition to high performance computing. Extensive world class support from Microsoft as well as third party for development, deployment and infrastructure are available.

Microsoft .NET Parallel Library

C# is a modern programming language that expresses events, objects, threads, lambda functions, anonymous delegates and callback methodologies. The new parallel library in .NET Framework 4.0 builds on these core language features. These libraries are usable from any .NET languages and directly supported on VB.NET and C++/CLI[2]. It is a runtime library instead of a compiler solution as Open MP

The primary goal of this library is simplify the development of parallel programs in .NET environment. It addresses the developer community who are accustomed to C# or other .NET programming languages, though in the traditional sequential programming point of viewpoint. The programming model therefore adheres to the shared memory model of parallel computing.

[2] Parallel Computing http://msdn.microsoft.com/en-us/concurrency/default.aspx

Appendix II: Microsoft Parallel Processing Technology

Microsoft .NET Parallel Extension library and the CCR/DSS asynchronous service oriented parallel processing technology are well suited to present the materials presented in the proposed course above. We present a few examples. We assume knowledge of C# 2.0.

As a developer steps into the realm of parallel programming easy but efficient schemes of parallel computing – such as data parallelism can quickly demonstrate the benefits. Data parallel feature is made available as a new extended type (Parallel) in the .NET 4.0. in the System.Threading namespace quite known to most .NET developers using multi-threading.

The Parallel type has For and ForEach methods to convert sequential iterations (for-loops, foreach-loops) into concurrently running sequential blocks of programs efficiently. The methods create suitable number of sequential blocks depending upon the number of processors in the underlying machine. This is done via the thread scheduling mechanisms in the .NET Framework along with Windows operating system. Having higher level types for defining parallelism hides the common developer pains associated with manually implementing this using low level Thread objects. However, a well disciplined approach to parallelization is not compromised.

If loop iterations are independent, i.e. no iteration need any value computed in another, then one can use suitable methods such as Parallel.For and pass the loop body and parameters as a delegate. A for loop of the type,

```
For (int i = 0; i < N; i++) results[i] = Compute(i);
```

may be converted to a version as follows, where each iteration may run concurrently.

```
using System.Threading;
...
Parallel.For(0, N, delegate(int i) {
    results[i] = Compute(i);
});
```

This is a call to the static For method of the Parallel type. The method takes the starting, ending values of parameter i for the delegate and the function is invoked for each value of i. Conceptually, this is very close to the loop structure and there is no need to add additional synchronization rules. This is so because there is no need for any coordination among the different calls.

The Parallel.ForEach method is used for looping over IEnumerable<T> objects. Since this method is part of .NET library, it can work with other .NET languages such as VB.NET or F#[3].

Parallel LINQ provides an easy to use parallelizing mechanism for LINQ queries[4]. This is very similar to the data parallel constructs. Again, the goal is to simplify adoption of parallel programming for conventional sequential programmers.

[3] Microsoft F# http://msdn.microsoft.com/en-us/fsharp/default.aspx

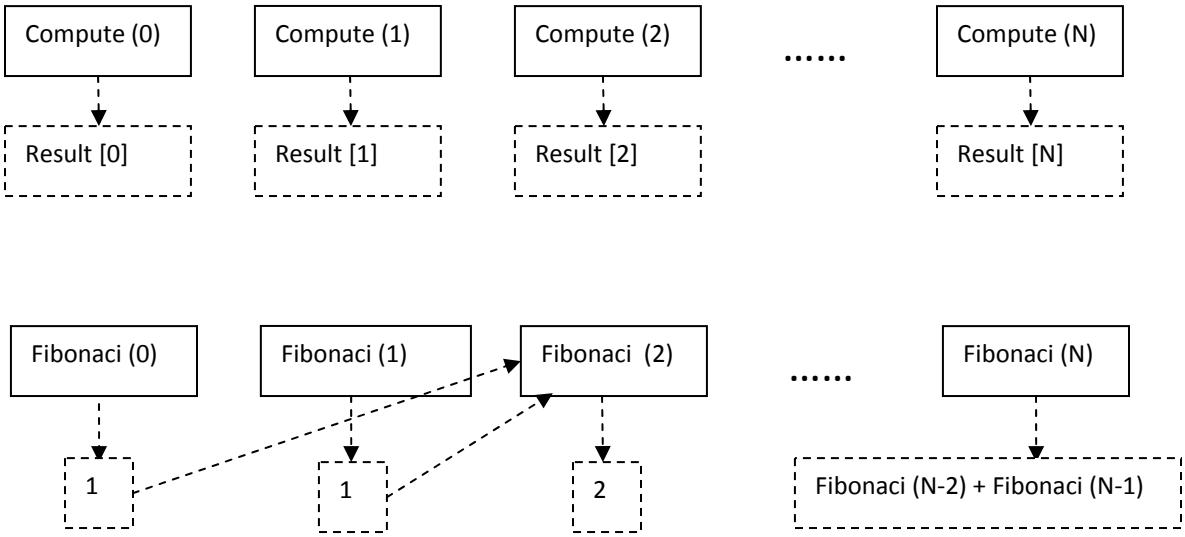
System.Threading.Tasks.Task type represents a task object. A task object is a unit of work in the parallel processing context. It is independently scheduled to run on a processor (general term for core, or processor). This is the basis of expressing task parallelism.

The Create method of Task allows one to create an instance of a Task type that is also associated with a code body that will be executed when this task is associated with a processor. Such association is made via Thread type which is how underlying operating system allocates process/threads to processors.

As a task instance often needs synchronization with other such task instances methods such as Wait and WaitAll are available for Task type. To implement a simple fork/join semantics one can use –

```
Task t1 = Task.Create(delegate { A(); });
Task t2 = Task.Create(delegate { B(); });
Task t3 = Task.Create(delegate { C(); });
Task.WaitAll(t1, t2, t3);
```

It is evident that the task type provides a direct access to the distribution of computational work in a system. It is lower level to the earlier loop parallelization methods.

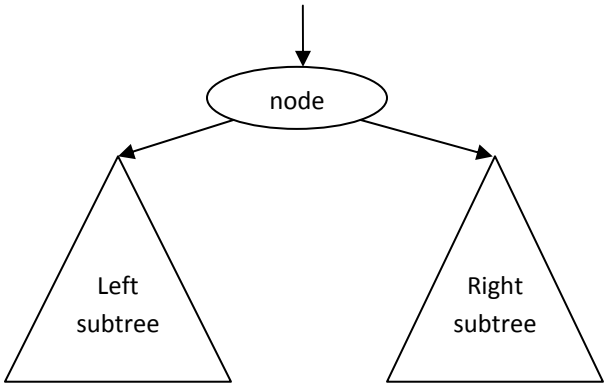


Consider the computation of the Fibonaci numbers where the N-th Fibonaci number is given by the sum of the previous two. The functions Fibonaci (i) depends upon the functions Fibonaci (i-1) and Fibonaci (i-2).

The dependency may be expressed as a general tree structure as follows, where “node” corresponds to the final computation and each tree, left and right corresponds to a recursive definition of the structure itself where the “node” is replaced by the corresponding two sub-computations. In case of the Fibonaci

[4] Running Queries on Multi-core processors http://msdn.microsoft.com/en-us/magazine/cc163329.aspx

calculation, node is Fibonaci(N) and the left and right subtrees corresponds to Fibonaci (N-2) and Fibonaci(N-1) respectively.



The structure may be defined and evaluated serially as follows:

```
class Tree<T>
{
    public T Data;
    public Tree<T> Left, Right;
    ...
}
```

In C#, iterating over this Tree sequentially might look something like the following:

```
static void WalkTree<T>(Tree<T> tree, Action<T> func)
{
    if (tree == null) return;
    WalkTree(tree.Left, func);
    WalkTree(tree.Right, func);
    func(tree.Data);
}
```

To parallelize this computation one may use Parallel.Invoke as follows:

```
static void WalkTree<T>(Tree<T> tree, Action<T> func)
{
    if (tree == null) return;
    Parallel.Invoke(
        () => WalkTree(tree.Left, func)  ,
        () => WalkTree(tree.Right, func) ,
        () => func(tree.Data));
}
```

Observe the Parallel.Invoke method takes several delegates and initiates tasks for each independently. The synchronization occurs via the return value mechanism of the called functions. Also, the parallelism is expressed in terms of dividing the entire computation in to tasks that run concurrently. Since,

WalkTree is called recursively; three tasks are generated each time it is called. This amounts to a large number of tasks being used. Tasks are mapped to physical processors via threads and too many tasks may cause excessive demand for threads and lead to resource contention. This is a key lesson in parallel processing and should be covered in details.

Finer control for task parallelism is implemented via the Task type. The concept of task is similar to a thread, however, at a higher abstraction level. It has its own local variables and environment. This is defined via constructors that can take parameters and a delegate for the method to execute. One can use lambda functions and anonymous delegates from C# 3.0. Wait methods are available to synchronize among tasks. Common semantics are available. For details see [].

Asynchronous task parallelism

Future<T> type derived from Task type provides a return value. This happens in an asynchronous manner. The entire mechanism of setting up callback and retrieval of result is hidden simplifying adoption and use.

Consider the example of counting all of the nodes in a large tree. Sequentially, this might be implemented as:

```
int CountNodes(Tree<int> node)
{
    if (node == null) return 0;
    return 1 + CountNodes(node.Left) + CountNodes(node.Right);
}
```

With Future<T>, this code can be parallelized as follows:

```
int CountNodes(Tree<int> node)
{
    if (node == null) return 0;
    var left = Future.Create(() => CountNodes(node.Left));
    int right = CountNodes(node.Right);
    return 1 + left.Value + right;
}
```

Comparing the asynchronous approach with the one with Parallel.Invoke or using Task directly one can see the benefit of the former quite easily. Note that unless the Value of a Future task type is retrieve the task is not started at all. This limits the number of tasks generated in the program and is more efficient in terms of implementation than the synchronous method given earlier.

Coordination structures

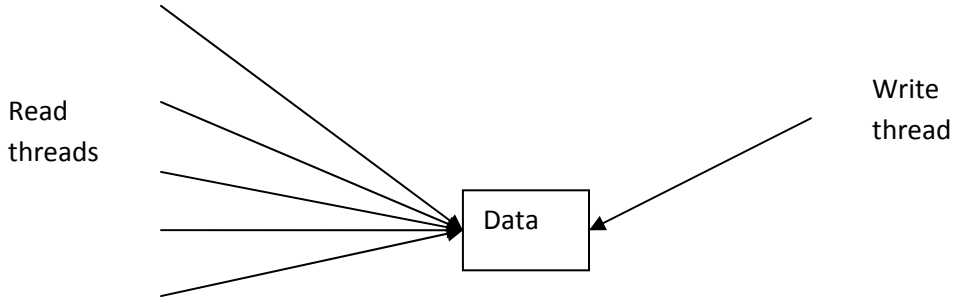
Coordination among multiple threads traditionally is done via control of interaction to avoid race condition. A race happens when the relative order of execution of threads result in wrong result. The goal in synchronization is to ensure this is avoided.

Methods used are locking, signaling and interlocked operations. Locking gives control of a resource to a thread at a time, or to a specified number of threads. The C# Lock statement is implemented using the Enter and Exit method of the Monitor class. It uses try…catch…finally to ensure the lock is released.

Although this form of coordination is essential it does not guarantee starvation, deadlock, fairness etc []. So, many powerful schemes are devised. For example, Monitor class provides TryEnter method that may block a thread for a specified interval and allowing detection of possible deadlock. The Wait method gives up control by a thread in critical section until resource is available, helping fairness and avoiding livelock. Mutex object may also be used for exclusive access to a resource. A thread calls the WaitOne method of a mutex to request ownership. The state of a mutex is signaled if no thread owns it.

Semaphores and reader-write locks are used for non-exclusive locking mechanism where a limited number of threads may be allowed to use a resource. Other mechanisms include Interlocked class, signaling via Join, use of WaitHandle and EventWaitHandles.

The coordination problem or pattern associated with the above synchronization primitives is well illustrated by the readers-writes problem []. Multiple threads read a resource concurrently, but require a thread to wait for an exclusive lock in order to write to the resource.



High level constructs such as thread-safe collections, more sophisticated locking primitives, data structures to facilitate work exchange, and types that control how variables are initialized are part of the .NET 4.0 parallel library. This considerably reduces the pain of developers in implementing correct and efficient coordination schemes.

The following table lists these higher level coordination structures. Counters are used to control thread execution. Threads perform decrement on counters and wait until it becomes zero. The decrement operation on such a counter is atomic and thread-safe (only one thread can get hold of such a counter and change its value at a time atomically). It also has an increment method (also atomic).

Table 1: High level thread-safe constructs

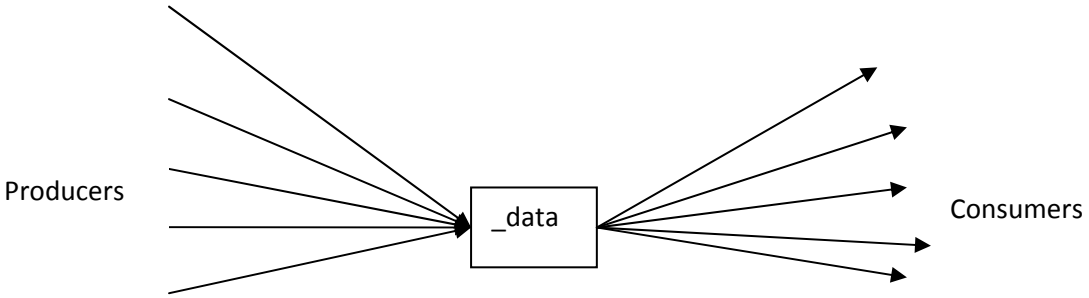
counters	CountdownEvent	ManualResetEventSlim	
locks	SemaphoreSlim	SpinLock	

Collections (namespace)	BlockingCollection<T>	ConcurrentQueue<T>	ConcurrentStack<T>
Lazy data structure	LazyInit<T>	WriteOnce<T>	

LazyInit<T> is a commonly-used tactic for delaying data initialization until the data is actually needed.

The high level coordination structure elevates the synchronization issue to higher level of abstraction that are more natural conceptually to sequential programmers while being in reality a good parallel implementation.

Consider the famous producer-consumer problem in which producers produces at a rate not necessary same as the rate of consumption by consumer. An intermediary collection structure may act as a buffer as the diagram show below.



The data structure for the buffer (_data) that holds strings may be expressed as follows.

```
private BlockingCollection<string> _data = new BlockingCollection<string>();
```

This could also be done by explicitly providing the underlying collection to be used:

```
private BlockingCollection<string> _data =
    new BlockingCollection<string>(new ConcurrentQueue<string>());
```

In the context of a producer-consumer problem described above, one or more producer threads can now add data to the queue:

```
private void Producer()
{
    while(true)
    {
        string s = ...;
        _data.Add(s);
    }
}
```

While one or more consumer threads are removing data from the queue, blocking as necessary until data is available:

```
private void Consumer()
{
    while(true)
    {
        string s = _data.Remove();
        UseString(s);
    }
}
```

Such a consumer loop can be made simpler by taking advantage of BlockingCollection<T>'s GetConsumingEnumerable method, which generates an IEnumerable<T> that removes the next element from the collection on each call to IEnumerable<T>.MoveNext. This allows for a BlockingCollection<T> to be consumed in standard constructs like a foreach loop:

```
private void Consumer()
{
    foreach(var s in _data.GetConsumingEnumerable())
    {
        UseString(s);
    }
}
```

It could even be used in a PLINQ query:

```
private void Consumer()
{
_data.GetConsumingEnumerable().AsParallel().Where(...).Select(...).ForAll(s
=>
    {
        UseString(s);
    });
}
```

BlockingCollection<T> acts as a wrapper around any concurrent collection that implements the System.Threading.Collections.IConcurrentCollection<T> interface, providing blocking and bounding capabilities on top of such a collection. Both the ConcurrentStack<T> and ConcurrentQueue<T> types implement IConcurrentCollection<T>, allowing them to be used with BlockingCollection<T>, but custom implementations of IConcurrentCollection<T> can also be used.

As is illustrated above, a high level buffer simplifies implementing the coordination scheme in comparison to conventional low level primitives. Also, this help in hiding much of the complexities of implementing such buffers and keeps the functionality close to what a sequential program developer can easily work with.

Handling Exceptions

In the parallel extension library all exceptions from different threads are aggregated into a AggregateException object. The original exceptions are accessible through the InnerException property of this object. This is a catchall approach and ensures that a developer can be made aware of all relevant exceptions and that all errors are properly bubbled up. Each exception has a valid stack trace.

Examples of scenarios where AggregateExceptions are thrown are from Parallel class – Parallel.For, Parallel.ForEach or Parallel.Invoke region; Task class – all exceptions thrown during the execution of all tasks will be at the tasks Wait method. The exceptions are also available from the Task's exception property. (is the latter Aggregate?). (need example); for Future<T> class – exceptions will be rethrown both from its Wait method and from its Value property; PLINQ – exceptions will occur when the query is executed – ForAll(), ToList(), ToDictionary(), ToLookup(), and MoveNext() on result of calling GetEnumerator() such as in a foreach loop.

Thread affinity needs to be honored for UI and COM based interoperation scenarios. This is an advanced issue and should be only presented in the context of interoperability.

Service oriented approach

Distributed memory model for parallel computing is based on message passing. This model has no shared memory and any resource sharing is via messages. The .NET library in the Microsoft Robotics Studio contains the Concurrency and Coordination Runtime (CCR) and the Decentralized Software Services (DSS) library to offer a loosely connected services oriented concurrency platform.

Although initially built with highly disperse concurrency requirement of a robotics computing environment the CCR offers basic concurrency, asynchronous message passing loosely connected computing model that inherently offers a general parallel programming framework. The parallel processing components are easily available.

We can look at software components as self-consistent services with known interfaces. A service can interact with another via service ports. This basic model is supported on one machine via CCR and across multiple machines via DSS. CCR provides mechanisms to define ports and services along with underlying asynchronous processing of messages that flow through the ports. As a message is posted to a message port of a service it is processed on a concurrently running thread.

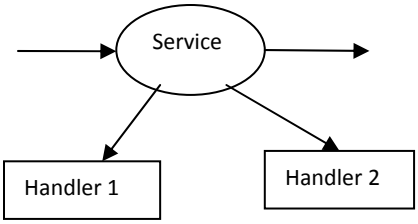
Parallelism is inherent in the model. Any number of items in a port is processed on different threads. The model does not need any explicit locking or synchronization as all dependencies are incorporated via the ports. The DSS extends the model to multiple machines where inter node message communication is done via HTTP based protocol called DSSP.

The model is similar to the well know dataflow model of parallel computing []. The Microsoft Robotics Studio also consists of a graphic lingual called Visual Programming Language that reflects the basic dataflow nature of the model.

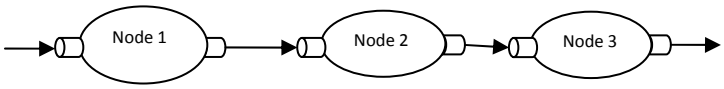
A program that defines a basic service class along with a Create method to create instances, a set of ports, and initialization of message handlers using a receiving task configured via a scheduler called Arbiter is given below.

```
/// PortSet that accepts items of int, string, double
public class CcrConsolePort : PortSet<int, string, double>
{
}

/// Simple example of a CCR component that uses a PortSet to abstract
public class CcrConsoleService
{
    CcrConsolePort _mainPort;
    DispatcherQueue _taskQueue;

    public static CcrConsolePort Create(DispatcherQueue taskQueue)
    {
        var console = new CcrConsoleService(taskQueue);
        console.Initialize();
        return console._mainPort;
    }

    private CcrConsoleService(DispatcherQueue taskQueue)
    {
        // create PortSet instance used by external callers to post items
        _mainPort = new CcrConsolePort();
        // cache dispatcher queue used to schedule tasks
        _taskQueue = taskQueue;
    }

    private void Initialize()
    {
        // Activate three persisted receivers (single item arbiters)
        // that will run concurrently to each other,
        // one for each item/message type
        Arbiter.Activate(_taskQueue,
            Arbiter.Receive<int>(true, _mainPort, IntWriteLineHandler),
            Arbiter.Receive<string>(true, _mainPort, StringWriteLineHandler),
            Arbiter.Receive<double>(true, _mainPort, DoubleWriteLineHandler)
        );
    }

    void IntWriteLineHandler(int item)
    {
        Console.WriteLine("Received integer:" + item);
    }
    void StringWriteLineHandler(string item)
    {
        Console.WriteLine("Received string:" + item);
    }
    void DoubleWriteLineHandler(double item)
    {
        Console.WriteLine("Received double:" + item);
    }
}
```

Note that concurrency is built in via the Arbiter which will activate a receiver task for any item posted on the port of the service. The task or handle of the message is scheduled via a DispatcherQueue and corresponding Dispatcher and optimizes the underlying cores of the hardware the program run on.



A program is given by set of service instances and ports instances connecting them. A service encapsulates a component. A sequence of service instance that processes output from the previous element may be shown as



Note that the model does not constrain the concurrent processing of the messages on the port. This is the typical pipeline pattern.

The CCR builds on three types of elements. These are CCR Ports and PortSets, coordination primitives called arbiters, and the Dispatcher and DispatcherQueue.

The Port class is a first in first out (FIFO) queue of items which is a Common Language Runtime type and a corresponding set of receivers that is guarded by arbiters. Arbiters execute user code, often a delegate to some method when some conditions are met. Arbiters can be nested to implement extended coordination logic. The Dispatcher and DispatcherQueue isolates scheduling and load balancing from the rest of the implementation. CCR avoids a single process wide execution resource as in CLR thread pool and allows the programmer to have multiple, completely isolated pools of OS threads that abstracts the notion of threading behind them. Multiple queues per dispatcher allow for a fair scheduling policy.

Adding an item to the port is an asynchronous operation. The interaction between components may take place by one posting an item in a port of another component in a non-blocking manner. This is done via Post() method.

```
Port<int>   portInt  = new Port<int>();

portInt.Post(10);
```

The second component can process such an item based on condition as implemented via arbiters. Such operation may either be passive or active. In the first case no tasks are scheduled when items are posted. Example is the Test() method on a port to detect presence of an item in it. Details are found in the user manual [].

The PortSet class allows for enumeration of all the port instances and item types it supports. The generic PortSet also provides convenient methods that automatically invoke the correct Port<> instance plus implicit conversion operations that can cast a PortSet<> instance to a Port<> instance automatically.

Aribiters implement a set of coordination logics. For example, it can create a task, a choice among two or more operations, a receiver, interleave, join of messages etc. It is possible to compose arbitrary coordination logic using them.

A simple example of using an arbiter type is given below:

```
// create a simple service listening on a port
ServicePort servicePort = SimpleService.Create(_taskQueue);

// create request
GetState get = new GetState();

// post request
servicePort.Post(get);

Arbiter.Activate(_taskQueue,
    get.ResponsePort.Choice(
        s => Console.WriteLine(s), // delegate for success
        ex => Console.WriteLine(ex) // delegate for failure
));
```

The get message is posted on the input port. The Activate method schedules a task on _taskQueue which listens to the ResponsePort associated with the get message (a message has a operation, response port and a body) via a Choice function and if the message type is that of s prints the value of s, or if it is of type ex, prints the value of ex.

An extensive set of types are available to develop parallel programs using this model. The Iterators is used for nesting asynchronous calls while being very close to the commonly understood iteration types. A sort program on a linear array may be designed with identical service instance wrapped in an iterator.

The CCR addresses failure handling with two approaches. One is the explicit or local failure handling using the **Choice** and **MultipleItemGather** arbiters. Combined with iterator support they provide a type safe, robust way to deal with failure since they force the programmer to deal with the success and failure case in two different methods and then also have a common continuation. The other one is an implicit or distributed failure handling, referred to as Causalities that allows for nesting and extends across multiple asynchronous operations. It shares in common with transactions the notion of a logical context or grouping of operations and extends it for a concurrent asynchronous environment. This mechanism can also be extended across machine boundaries.