

# Developing Parallel Programs

Ranjan Sen  
Computer Science Dept., Calcutta University, India

## Abstract

Parallel programming is an extension of sequential programming; today, it is becoming the mainstream paradigm in day-to-day information processing. Its aim is to build the fastest programs on parallel computers. The methodologies for developing a parallel program can be put into integrated frameworks. Development focuses on algorithm, languages, and how the program is deployed on the parallel computer.

## 1. Introduction

Parallel programming utilizes concurrency to achieve high-performance computing. Historically confined to supercomputing parlance, parallel programming today is becoming the mainstream paradigm in regular day-to-day information processing. This is energized by the widespread availability of multi-core multiprocessors and cost-effective server clusters. The software industry in general is integrating rich desktop and server software-development tools with new-generation parallel-processing tools.

Examples include use of Microsoft Visual Studio and the .NET extension for parallel computing, Microsoft Windows HPC Server, decentralized distributed service-oriented programming, grid computing, and so on. Many of these are rich in ideas that are based on decades of research; side-effect-free functional programming, giving protection against race; data-flow paradigm for non-von Neumann architecture; and many more.

Parallel programs are built by combining sequential programs. The goal is to allow independent sequential programs to run in parallel and produce partial results that then are merged into the final solution via different combination patterns. We want to get correct, bug-free parallel programs that can deliver performance and possibly other benefits, such as reliability, availability, and

fault tolerance that is integrated with an existing software ecosystem.

Parallel programming is fast becoming an essential developer skill. Multifarious variations in parallel-processing technology, from clients to server clusters, provide diverse developer toolsets and runtime environments. Knowing the basic concepts helps in a better comprehension of the complexity, and it is never more crucial to the developer than now.

## 2. Correctness and Performance

Developers must continue creating correct and efficient applications. Both correctness and performance confirm that a program produces the result that it is supposed to deliver within an expected time frame. In establishing this, the conventional model that is used for sequential computers is von Neumann's "stored-program" model. In the "storedprogram" model, there is a single thread of execution; instructions are executed by one processor at a time.

In parallel computers, there is more than one processor, each of which executes an execution thread simultaneously. Parallel-computer models that are used for correctness and performance analysis are simple extensions of stored-program models. The two models that are used are the shared-memory model and the distributed-memory model (Figure 1). In the first model, a common memory is shared by all processors; in the latter model, it is not.

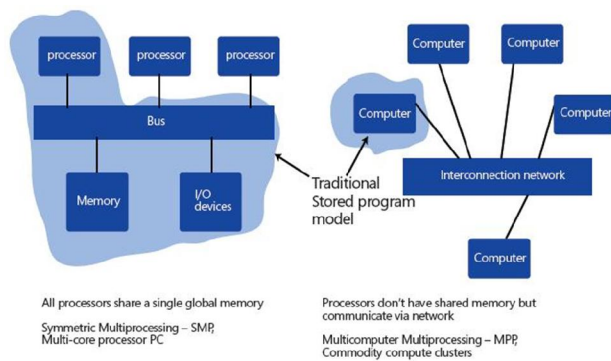


Figure 1: Parallel computers -- shared-memory and distributed-memory models

The goal of achieving a high-performance application is achieved by having several sequential programs run simultaneously, overlapped in time, with the common goal of solving the same problem. This leads to two important concepts: decomposition and pattern.

### 3. Decomposition and Pattern

*Decomposition* is the art of splitting (or decomposing) a problem into independent parts to be solved concurrently. Each of these parts might obtain (partial) results that can be combined to obtain the final result; we need a combining scheme ( *or pattern* ) for these parts. We can establish correctness and analyze for performance for each of the parts, as well as the pattern that is used, to argue about correctness and performance of the overall parallel computation.

As an example, consider the problem of finding maximum of 16 data. We can divide the data into four parts of 4 data and find the maximum for each of these parts concurrently on four processors. Then, we can find the maximum of the maximums. The sequential parts that are used are the method of finding maximum of 4 data. The pattern that is used is finding four intermediate possible maximums in parallel, and then finding the actual maximum. Figure 2 illustrates this scheme.

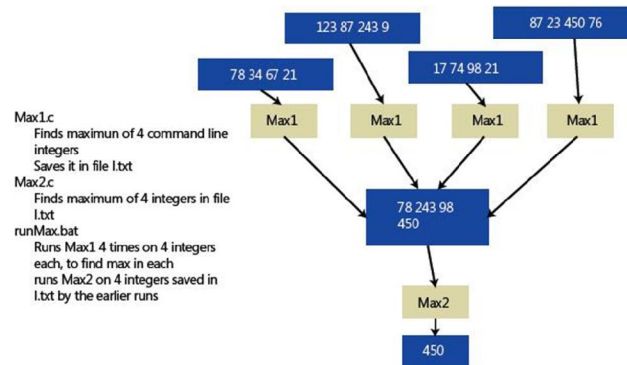


Figure 2: Scheme showing problem of finding maximum of 16 data

The idea of using decomposition and pattern is not new (see a standard text, such as [Quinn, 2004] in References). One can think of decomposition as finding one or more pieces of sequential algorithm (sequential program) that can be run concurrently on more than one processor. Such a sequential piece is often referred to as *computational grain* or simply the *grain* of a parallel computation. Similarly, a pattern corresponds to a high-level algorithm of coordination or a *composition* scheme. Several patterns are known to be useful (see [Mattson, 2005] in References).

### 4. Analysis of Parallel Programs

Parallel-computer models can be used to analyze parallel algorithms or the corresponding programs for correctness and performance. A parallel algorithm is correct if both the sequential program and the pattern used are correct. We can follow methods that are similar to those used for sequential programs/algorithms to establish correctness. We can use the same approach for *debugging/diagnosing* a faulty parallel program in this way.

In determining correctness, we examine the memory states of data that the program is supposed to transform. In parallel programs, *dependencies are linear within the sequential pieces of programs* that run in parallel. However, the *pattern may have nonlinear dependencies*. For example, the pattern that is used in the preceding algorithm to find the maximum of 16 integers is a correct

scheme, because the programs Max1, Max2, and the composition scheme that is given by runMax all are correct. Figure 2 shows a graph of the dependencies of the sequentially executing programs that are given by Max1 and Max2, as expressed in runMax.bat. More often, the nodes of these graphs can represent either data or tasks (computation), or both. In the former case, it is called a *data-flow graph*; in the latter, it is called a *task graph*.

Similarly, an algorithmic approach can give us an estimate of performance. For example, we can reason that in the first stage (of parallel computation), four execution instances of Max1 on four processors can take place in time T (to find maximum of 4 integers) concurrently. In the second stage, we may have one execution instance of Max2 on one processor. Then, the overall time of the algorithm that is used is 2T with four processes (a *process* is an execution instance of a program). This estimate gives a good point of reference as to what to expect.

## 5. Speedup: A Measure of Performance

The ratio of time that is taken by a sequential program to time that is taken by a parallel program is called speedup. In general, you can find different parallel algorithms to solve a problem. It is important to know which achieves the best performance.

Amdahl's law gives  $1/[S + (1-S)/n]$  as an estimate of maximum speedup, where S is the fraction of inherently sequential code in an application, and n is the number of processors. By way of illustration, in the preceding maximum-finding program, the fraction S is given by the program Max1.c. In the example of finding maximum of the 16 integer, the fraction S is 0.2 (four instances of Max1 and one instance of Max2 run sequentially would be 100 percent) and, by Amdahl's law, speedup can be at most 2.5 with four processors.

The notion of scaled speedup is given by Gustafson-Barsis's law. According to it, scaled speedup is bounded by  $n + (1-n)*s$ , where n is the number of processors, and s is the ratio of the time that is spent in the serial part of the program versus the total execution time. In our preceding example,  $s = 1/(\log_4 16) = 0.5$ . So that, for n = 4, this is

2.5; for n = 16 (s = 0.3), it is 11.5; for n = 64 (s = 0.25), it is 49; and so on.

## 6. Parallel-Computing Platforms

In the early days of parallel processing, architectures were expensive and specialized. Recently, multi-core processors have become the de facto processor technology. 1 The multi-core phenomenon caused a large-scale impact on game software in early 2000, when Sony used multiple processors for its PlayStation PS2. 2 At the same time, high performance server-cluster programs are superseding the supercomputers in performance. 3

There is also the trend of special hardware, such as gate arrays (for example, FPGA); Graphics Processor Units (GPU) or cell processors are bringing out new ways to assemble parallel architecture. Today, diverse scenarios of distributed systems are using parallel processing for improved resource utilizations, throughput, reliability, and availability.

In the large-scale parallel-computing platform technology, operating systems are updated for multi-core processors, and new and extension in optimizing compilers and development systems are being crafted out. In the distributed-systems arena, we are seeing rapid integration of mainstream enterprise-grade technology, as well as a growth in loosely coupled systems. Some of the related software and switching technology are mentioned later.

Myrinet is an ANSI 4 standard that is used widely in computer clusters. 5 Features include an interface card that uses firmware to process protocols and off-loads host processors, OS bypass for low-latency communication, and so on. Ten-gigabit Ethernet is an IEEE standard and is the fastest version of the Ethernet standard. This is 10 times as fast as Gigabit Ethernet, which is the technology for transmitting Ethernet frames at the rate of one gigabit per second. Network switched fabrics, such as InfiniBand, 6 are commonly used in parallel-computer architectures

## 7. Computer Clusters

Clusters of computers and workstations are a very popular hardware/software commodity as a cost-effective parallel-processing platform (see [Sterling, 2002] in References). However, administering and managing such clusters can be quite complex. Clusters of Windows Server (here, called Windows HPC Server) address these problems in addition to the high performance platform goals. Windows HPC Server provides necessary cluster services and tools, including Microsoft MPI, job scheduler, and cluster-management service to make powerful cluster solutions in diverse scenarios. The high-performance ranking is in the top 10 of the top-500 list. 7 New-generation network services are added with MSMPI for support of very high-speed communication between physical computes in a cluster. The job scheduler can run jobs that are defined in service-oriented architecture (SOA), in addition to traditional job definitions, as a composition of tasks that execute programs around the cluster nodes. Also, it accepts jobs via the HPCBP service interface — thus allowing interoperability from any platform that adheres to the grid-interface protocol.

A Dryad 8 is an infrastructure for using the resources in a cluster or data center that allows a programmer to express a program in terms of sequential programs and connecting them via one-way channels. Dryad can express common computing frameworks, such as map-reduce 9 or the relational algebra; it handles job creation and management, resource management, job monitoring, visualization, fault tolerance, re-execution, scheduling, and accounting. (See Figure 3.)

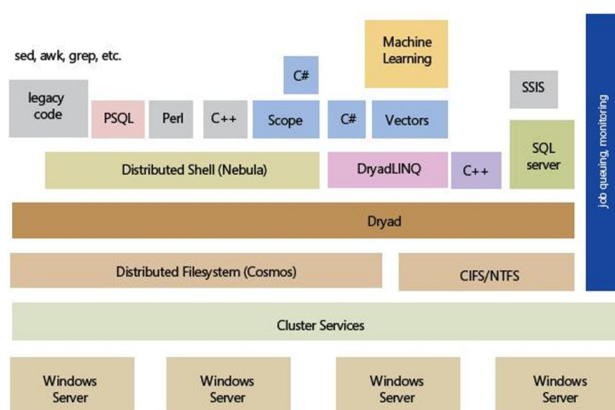


Figure 3: Dryad architecture

SSIS SQL Server 2005 Integration Service has been built on top of Dryad. It executes many instances of Microsoft SQL Server, each on a Dryad vertex, and uses fault tolerance and scheduling services. This is being used currently as part of the AdCenter 10 log-processing pipelines. The goal of DryadLINQ, 11 a related project, is to make distributed computing on a large computer cluster simple enough for ordinary programmers. DryadLINQ translates LINQ programs into distributed Dryad computations and distributes them to different nodes of a cluster.

The features include declarative programming; automatic parallelization (both multi-core on a workstation and cluster-wide); integration with Visual Studio (Intellisense, code refactoring, integrated debugging, build, source-code management); automatic serialization; job graph optimizations, via both static term rewriting and dynamic query-plan optimizations; and conciseness.

## 8. Decentralized Software Services (DSS)

The DSS runtime is built on top of Concurrency and Coordination Runtime (CCR), 12 which is a highly concurrent, message-oriented programming model. CCR has powerful orchestration primitives, enabling coordination of messages without the use of manual threading, locks, semaphores, and so on. CCR addresses the need of service oriented applications by providing a programming model that facilitates managing asynchronous operations, dealing with concurrency, exploiting parallel hardware, and handling partial failure.

Run-time files for CCR and DSS are available on the Microsoft .NET Framework and .NET Compact Framework. The DSS protocol is being distributed via the Microsoft Open Specification Promise. 13 The availability of the protocol will make communication between a variety of hardware and software easier.

Binary serialization gives faster throughput. VPL development tools support regular as well as mobile development. Also, there is a DSS Service-generation tool: Visual Simulation Tool.

between task  $i$  and task  $j$  indicates a dependency of task  $j$  on task  $i$ . Independent tasks can run in parallel. Consequently, if the tasks are executed in parallel, task  $j$  will have to wait for task  $i$  to send data. This is called *data dependency*, and the graph is a data-flow graph. However, if the channels represent completion signals, this depicts *control dependency*; in that case, the graph is a control-flow graph (also called a task graph).

## 9. Developing Parallel Programs

Consider a four-step process for parallel-program design: partition, communication, agglomeration, and mapping (see [Foster, 1995] and [Dongarra, 2003] in References). *Partitioning* is the process of dividing the computation and the data into pieces or primitive tasks. Increasing the number of primitive tasks reduces the inherently sequential fraction in the parallel program that is designed. This helps in raising the parallelism that is possible, according to the theoretical bounds that are given by Amdahl's law and Gustafson-Barsis's law. Communication considers the plan for interprocess communication necessary for the parallel program.

The diagram illustrates the execution of a task on a multi-processor system. On the left, a cloud represents the 'Program' environment, containing 'Local Memory' and 'Data'. A 'Task' is shown as a sequence of nodes (0, 1, 2, 3, 4) connected by arrows. The first arrow is labeled 'channel'. A curved arrow labeled 'dependency' indicates a feedback loop from node 3 back to node 0.

Figure 4 gives a conceptual view of the task/channel model. Tasks are represented as circular nodes and channels are represented by directed lines. A direct line

Parallel programming aims to build the fastest programs on parallel computers. These programs must be correct as well as amenable to modern software-engineering practices for efficient life-cycle management. The main factors to achieve this are the following:

- 22



### 3. Programming environment and tools

### 4. Target parallel-computing platform

There is considerable literature on designing parallel algorithms (see [Akl, 1989], [Leighton, 1992], and [Miller, 2005] in References). Essentially, the basic approach is finding sequential pieces that can run in parallel and combining efficiently the results that they obtain.

Tools for developing parallel programs are based on four different approaches. The first is to extend a compiler. The second is to extend a sequential programming language and allow core parallel-programming schemes to be captured from known environments. The third is to add a parallel-programming layer; this is a layer on a sequential core that controls creation and synchronization of processes and partitioning data. The fourth is to create a new parallel-programming language, such as Fortran 90, High Performance Fortran, 17 or C. 18 We will discuss the two more popular approaches: OpenMP, which is an extension of C++, and Message-Passing Interface (MPI).<sup>19</sup>

## 11. OpenMP

OpenMP is based on the shared-memory model. The standard view of parallelism in a shared-memory program is fork/join parallelism. When the program begins execution, only a single thread (master thread) is active. The master thread executes the sequential portions of the algorithm. At points where parallel operations are necessary, the master thread forks (creates or awakens) additional threads. Then, the master thread and these new threads work concurrently through the parallel section. At the end of the parallel code, the created threads die or are suspended, and the flow of control returns to the single master thread.

A sequential program is a special case of a shared-memory parallel program—one that has no fork/join. The shared-memory model supports incremental parallelization, which makes it possible to transform a sequential program into a parallel program one block of code at a time. This is a quick way to develop a parallel version of an existing program. However, the underlying algorithm might not be

the best parallel algorithm. OpenMP makes it easy to indicate when the iterations of a **for** loop can be executed in parallel. See the second commented-out line in the following code snippet:

```
#pragma omp parallel private( t, x,y,local_count)
{
    local_count = 0;
    Random^ rand = gcnew Random();
    t = omp_get_num_threads();
    #pragma omp parallel for
    for (int i = tid; i < samples; i += t) {
        x = rand->Next(0,10000)*.0001;
        y = rand->Next(0,10000)*.0001;
        if (x*x+y*y <= 1.0) local_count++;
    }
    #pragma omp critical
    count += local_count;
}
```

The **#pragma omp parallel for** directives in OpenMP are denotations to the C++ compiler to process the portion in the curly brackets for parallel execution. Also, note how it is possible to define parameters that are private to each thread (to reduce contention for shared memory), and the use of a critical segment using **pragma s**.

In the preceding example, private variables are declared via a clause of the parallel **pragma** declaration. This allows avoiding contention when all threads access these variables (in the parentheses). Note that we have used a critical segment to allow the threads to add their results back to the value to the shared variable **count**.

## 12. Message-Passing Interface (MPI)

MPI is a standard programming library that is available from FORTRAN, C, or C++. It enables creation of a distributed-memory programming environment that can be established across different physical computers. There are

different flavors of MPI: Microsoft MPI (which is based on MPI-2 20 ), HP MPI, Intel MPI, Open MPI, LAM/MPI, MPICH, FT-MPI, and others.

SPMD for a distributed-memory parallel-computer model is the underlying approach of the programming. The same program is run on all participating computers (processors, cores, and nodes). An MPI runtime makes services available through application programming interfaces (APIs) for necessary support of parallel computation. Processors are identified by rank in a communication world, and it is possible to have one-to-one as well as collective communication between them.

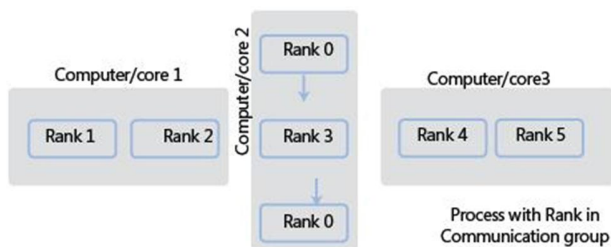


Figure 5: Three physical computers hosting multiple processes with distinct ranks

In Figure 5, three physical computers are shown to host multiple processes that have distinct ranks.

The entire collection forms a communication world, so that any processes that are in it can access each other via message-based communication.

A simple MPI program is shown in the following code snippet:

```

#include "mpi.h"
#include <stdio.h>

int main(int argc, char* argv[]) {
    int numtasks, rank, rc;

    /** initialize MPI environment **/
    rc = MPI_Init(&argc, &argv);

```

```

    if (rc != MPI_SUCCESS) {
        printf("Error starting MPI program. Terminating.\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    /** get the number of processes and their ranks **/
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Number of tasks= %d My rank= %d and\n",
        numtasks, rank, getenv("COMPUTERNAME"));

    MPI_Finalize();
}

```

MPI functions and constants are defined in the `mpi.h` file and the data types; operations and constants are similar to the standard C and FORTRAN equivalents. For complete list of MPI functions, see [Gropp, 1999] in References.

### 13. New-Generation Tools

Java and .NET programming languages have programming extensions to support parallel programming in managed runtimes (see [Lea, 1999] in References). Parallel FX Library (PFX) runs on .NET Framework 3.5 and the to be released new .NET Framework 4.0. 21 The .NET Framework provides a runtime that is called the CLR and which runs the code in a managed environment, with automatic garbage collection, just-in-time execution, added code-access security, and so on. In this way, parallel processing is integrated with the hosts of modern .NET-based technologies. 22

The underlying technique in PFX is to use anonymous functions—building expressions by using them and then executing such expressions in parallel. It is convenient to represent anonymous functions as anonymous delegates 23 or as lambda expressions. 24 Also, it is possible to create expression trees by using nesting of expressions; and, with the help of lambda expressions, we can use functions in such expressions.

Imperative task parallelism is achieved via the task parallel library: System.Threading.dll. Task Parallel Library (TPL) is built on a scheduler that uses cooperative scheduling and work stealing to achieve fast, efficient scheduling and maximum processor utilization.

TPL provides the **System.Threading.Parallel**, **System.Threading.Tasks.Task**, and **System.Threading.Tasks.Future<T>** types, respectively. The first type is used for parallelizing loops and regions. The static methods that are available with the **Parallel** type are **For**, **ForEach**, and **Invoke**. For example:

#### Task Parallel:

```
for (int i=0; i < n; i++) results[i] = compute(i);
```

```
Parallel.For(0, n, I => results[i] = compute(i));
```

#### Data Parallel:

```
(IEnumerable<T> objects;
```

Use of foreach and ForEach keywords):

```
foreach(testClass t in data) compute (t);
```

```
Parallel.ForEach(data, delegate(testClass t)
{compute(c);});
```

Note that the **For** and **ForEach** methods take a lambda expression for definition of the function to apply in parallel. The **Invoke** static method can be used to run statements in a block of statement in parallel. The **Task** class can be used to create and operate on a task; it is similar to what **ThreadPool** provides. A delegate is queued for execution. The **Task** is simpler to use and offers more functionality. Methods for wait, status check of tasks are present. Illustrations are given in C#, but TPL is available also in Visual Basic 2008 and F#, which is a functional programming language (see [MacLennan, 1990] in References).


The **Future<T>** class derives from **Task**. This has a value associated with it that is the result of the asynchronous execution of the **System.Func<T>** type instance that is provided as parameter. The value can be accessed from the **Future** instance and can be used to wait until it is available. **Future** provides a mechanism to define a data-driven or dataflow computing architecture. 25

High-level constructs — such as thread-safe collections, more sophisticated locking primitives, data structures for work exchange, types to control how variables are productive, and the repertoire of powerful synchronization primitives — include **CountEvent**, **LazyInit<T>**, **ManualResetEventSlim**, **SemaphoreSlim**, **SpinLock**, **SpinWait**, **WriteOnce<T>**, and the **Collections.BlockingCollection<T>**, **Collections.ConcurrentQueue<T>**, and **Collections.ConcurrentStack<T>**.

Parallel LINQ (PLINQ) is a component of PFX. The data parallel nature ensures that programs can scale efficiently as data increases. PLINQ offers an incremental way of taking advantage of parallelism for existing solutions to existing problems. To use PLINQ, you will have to wrap the data source in an **IParallelEnumerable<T>** with a call to the **System.Linq.ParallelEnumerable.AsParallel** extension method (**IParallelEnumerable** is an extension of **IEnumerable<T>**).

```

IEnumerable<T>data = ...;
var q = data.Where(x -> p(x)).OrderBy(x-> f(x));
foreach (var e in q) a(e);
    
```



The **var q** defines the query, and **foreach** actually executes it over the data source **q**. This declarative query helps the PLINQ to delay determination of optimal resource uses, such as the number of processors to run the query until it is actually executed in the **foreach** with action **a**. It will arrange for parts of the query to run on the available processors through the hidden use of multiple threads.





## 14. MPI.NET

MPI.NET 26 is an efficient interface for using the native MPI library from C#. It simplifies interface and extends MPI by taking advantage of features of C# and the managed-unmanaged interoperability mechanism. Several innovative measures have been taken to reduce abstraction penalties in performance. For example, generic versions of point-to-point **Send** allow the use of any user-defined types for transmission. In general, this is extended to all types of communication operations. For more information, see [MPI.NET, 2008] in References

## 15. Programming Environment and Tools

In multi-core systems, operating systems are revamped to include various paradigms to ensure better resource utilization. In clusters, many of these supports are integrated development and deployment services and tools, including service-oriented job scheduling. 27

Tools provide debugging support at both source levels, such as in Visual Studio. Visual Studio also provides a parallel debugger extension. 28 Trace logs can help diagnose these problems. 29 Portland Group has a debugger for Windows cluster. 30 Other providers include TotalView. 31

The most common process is to profile the behavior via tracing tools, followed by analysis and tuning. MPI was developed with tracing in mind. MPE, which is trace library, is available with MPI distribution 32 ; also, it is shipped with Windows HPC Cluster. 33 The trace may be viewed by using viewing tools, such as Jumpshot. 34 Other tools include Intel Trace Analyzer and Collector (Vampir), 35 MPICL + ParaGraph, 36 and Epilog and KOJAK. 37

## 16. Conclusion

Parallel programming is an extension of sequential programming. A parallel algorithm is given by algorithms of the constituting sequential program and a pattern to

combine them. The programming model for analyzing sequential programming is extended to the shared-memory model and the distributed-memory model. Various processor and cluster architectures that support parallel computing are variations of these two models.

Correctness of parallel programs can be established via correctness of the sequential programs and the pattern of combination of these pieces. Performance of parallel programs depends on algorithm, implementation details, and target-computer architecture. Parallel computers range from multi-core processors to clusters, computational grids, and cloud computers.

All of the methodologies for developing parallel programs can be put into an integrated framework. Development focuses on algorithm, languages, and how the program is deployed on the parallel computer.

## References

- [1] Akl, Selim G. *The Design and Analysis of Parallel Algorithms* . Englewood Cliffs, NJ: Prentice Hall, 1989.
- [2] Dongarra, Jack J., et al. *Sourcebook of Parallel Computing* . San Francisco: Morgan Kaufman Publishers, 2003.
- [3] Foster, Ian. *Designing and Building Parallel Programs : Concepts and Tools for Parallel Software Engineering* . Reading, MA: Addison-Wesley, 1995.
- [4] Gropp, William, et al. *Using MPI: Portable Parallel Programming with the Message-Passing Interface* . Cambridge, MA: MIT Press, 1999.
- [5] Lea, Doug. *Concurrent Programming in Java: Design Principles and Patterns* . Second edition. Reading, MA: Addison-Wesley, 1999.
- [6] Leighton, Frank Thomson. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes* . San Mateo, CA: M. Kaufmann Publishers, 1992.
- [7] MacLennan, Bruce J. *Functional Programming: Practice and Theory* . Reading, MA: Addison-Wesley, 1990.
- [8] Mattson, Timothy G., et al. *Patterns for Parallel Programming* . Boston: Addison-Wesley, 2005.
- [9] Miller, Russ, et al. *Algorithms Sequential and Parallel: A Unified Approach* . Second edition. Hingham, MA: Charles River Media, 2005.
- [10] Gregor, Douglas, and Andrew Lumsdaine. "Design and Implementation of a High-Performance MPI for C# and the Common Language Infrastructure." MPI.NET Publications. Proceedings of 13th ACM SIGPLAN Symposium on Principles

and Practice of Parallel Programming, Salt Lake City, February 2008.

[11] Quinn, Michael J. *Parallel Programming in C with MPI and OpenMP*. Dubuque, IA: McGraw-Hill, 2004.

[12] Sterling, Thomas L. *Beowulf Cluster Computing with Windows*. Cambridge, MA: MIT Press, 2002

**Ranjan Sen** earned his Ph.D. in Computer Science at Calcutta University in 1978, and has served on the faculty of several universities since 1979, including Indian Institute of Technology, Rutgers University, and Hampton University. Ranjan specializes in graph theoretic modeling for algorithm to architecture mapping and has published extensively on the subject.