

Writing Parallel Processing Compatible Engines Using OpenMP

Aniruddha Deshmukh
Cytel Inc.

Email: aniruddha.deshmukh@cytel.com

Introduction

Why Parallel Programming?

- Massive, repetitious computations
- Availability of multi-core / multi-CPU machines
- Exploit hardware capability to achieve high performance
- Useful in software implementing intensive computations

Examples

- Large simulations
- Problems in linear algebra
- Graph traversal
- Branch and bound methods
- Dynamic programming
- Combinatorial methods
- OLAP
- Business Intelligence etc.

What is OpenMP?

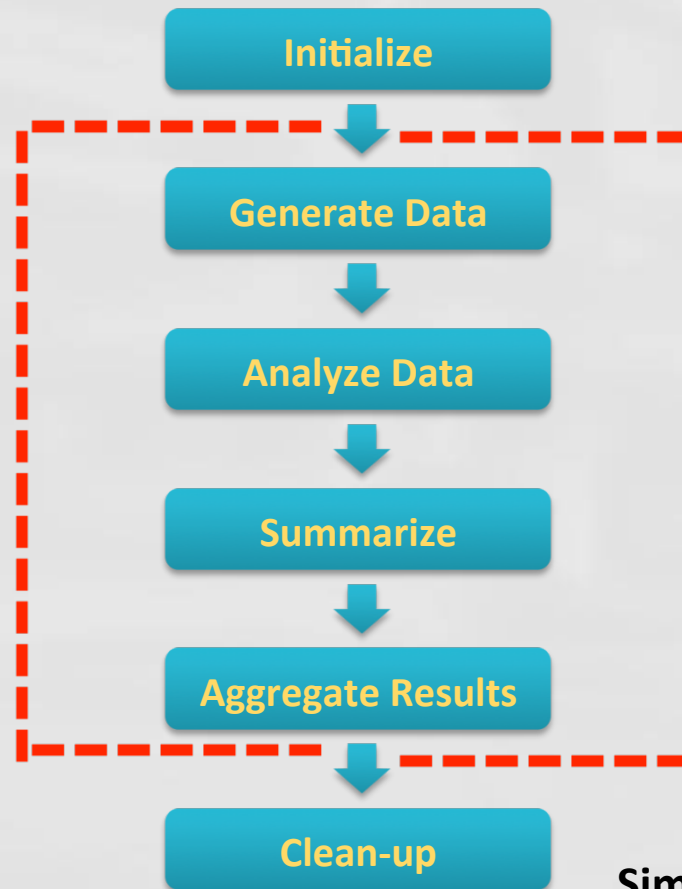
Open Multi Processing

- A standard for portable and scalable parallel programming
- Provides an API for parallel programming with shared memory multiprocessors
- Collection of compiler directives (pragmas), environment variables and library functions
- Works with C/C++ and FORTRAN
- Supports workload division, communication and synchronization between threads

An Example - A Large Scale Simulation

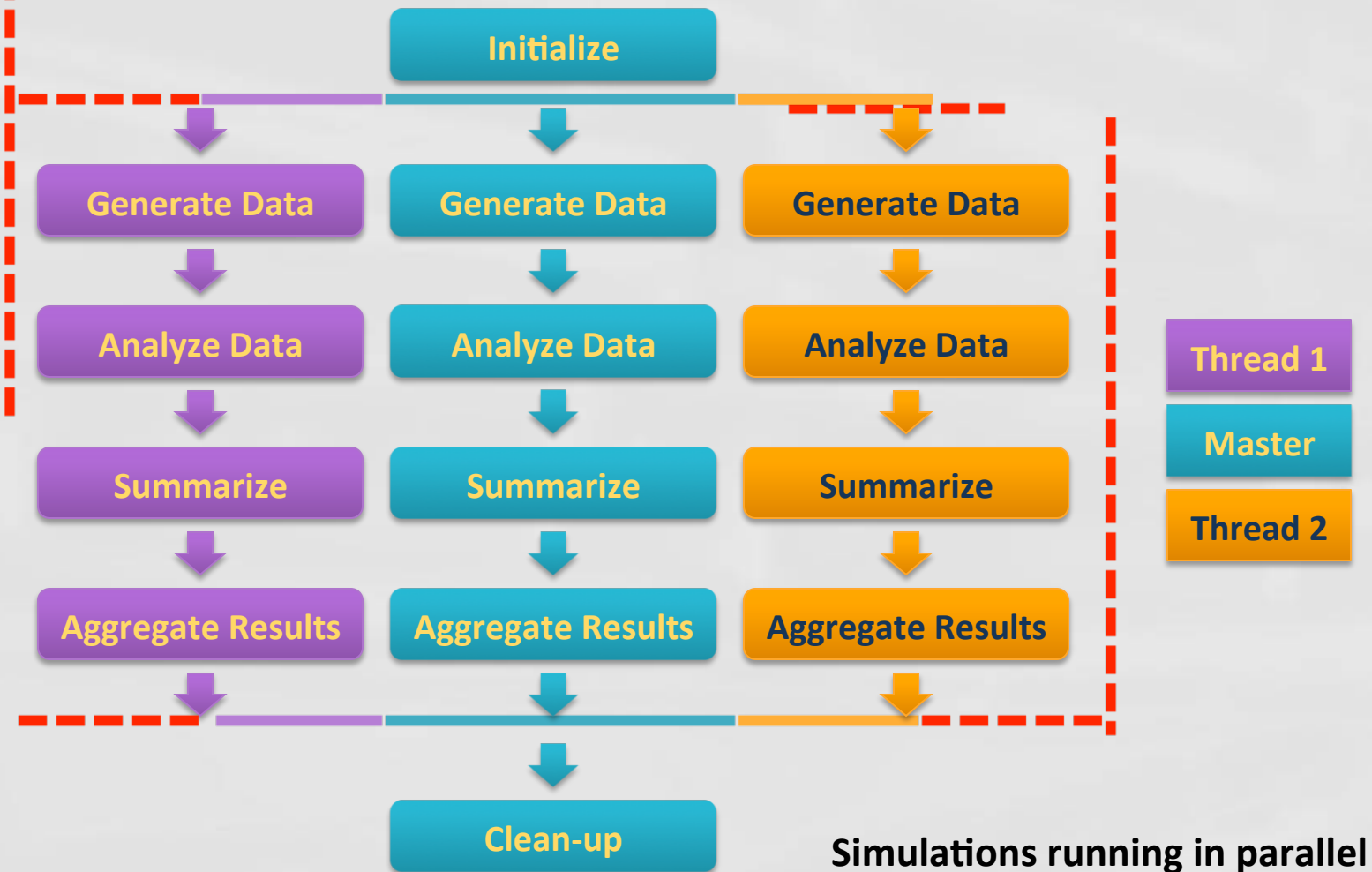
Clinical Trial Simulation

Simplified Steps



Simulations running sequentially

Parallelized Simulations



Simplified Sample Code

```
void PerformSimulations(long lSimCount, long lSampleSize, SIM_OUT* pSimOut, char* sDataFile)
{
    TRIAL_DATA* pTrialData[lMaxThreads];
    double* pdArrRndNum[lMaxThreads];

    // Allocate memory to pTrialData and pdArrRndNum arrays here.
    // Any other initialization

    #pragma omp parallel for ordered
    for(long lSimInd = 0; lSimInd < lSimCount; lSimInd++)
    {
        #pragma omp ordered
        {
            GenerateRandomNumbers(lSampleSize, pdArrRndNum[lCurThread], lSeed);
        }

        GenerateTrialData(pTrialData[lCurThread], lSampleSize, pdArrRndNum[lCurThread]);

        #pragma omp critical (SUMMARIZE)
        {
            AnalyzeAndSummarize(pTrialData[lCurThread], pSimOut);
        }
    }
}
```

Simplified Sample Code

```
void PerformSimulations(long lSimCount, long lSampleSize, SIM_OUT* pSimOut, char* sDataFile)
{
    TRIAL_DATA* pTrialData[lMaxThreads];
    double* pdArrRndNum[lMaxThreads];

    // Allocate memory to pTrialData and pdArrRndNum arrays here.
    // Any other initialization

    #pragma omp parallel for ordered
    for(long lSimInd = 0; lSimInd < lSimCount; lSimInd++)
    {
        #pragma omp ordered
        {
            GenerateRandomNumbers(lSampleSize, pdArrRndNum[lSimInd]);

            GenerateTrialData(pTrialData[lSimInd], pdArrRndNum[lSimInd]);
        }

        #pragma omp critical (SUMMARY)
        {
            AnalyzeAndSummarize(pTrialData[lSimInd], pSimOut);
        }
    }
}
```

- ◆ Declare and initialize variables
- ◆ Allocate memory
- ◆ Create one copy of trial data object and random number array per thread.

Simplified Sample Code

```
void PerformSimulations(long lSimCount, long lSampleSize, long lSeed)
{
    TRIAL_DATA* pTrialData[lMaxThreads];
    double* pdArrRndNum[lMaxThreads];

    // Allocate memory to pTrialData and pdArrRndNum
    // Any other initialization

    #pragma omp parallel for ordered
    for(long lSimInd = 0; lSimInd < lSimCount; lSimInd++)
    {
        #pragma omp ordered
        {
            GenerateRandomNumbers(lSampleSize, pdArrRndNum[lCurThread], lSeed);
        }

        GenerateTrialData(pTrialData[lCurThread], lSampleSize, pdArrRndNum[lCurThread]);

        #pragma omp critical (SUMMARIZE)
        {
            AnalyzeAndSummarize(pTrialData[lCurThread], pSimOut);
        }
    }
}
```

- ◆ Simulation loop
- ◆ Pragma omp parallel for creates multiple threads and distributes iterations among them.
- ◆ Iterations may not be executed in sequence.

Simplified Sample Code

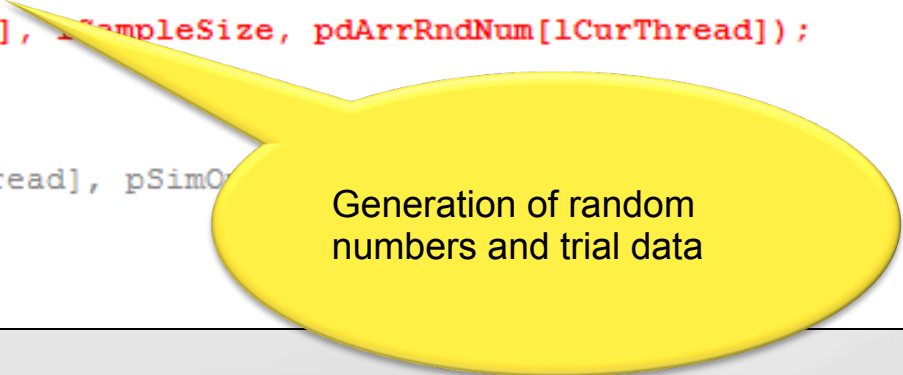
```
void PerformSimulations(long lSimCount, long lSampleSize, SIM_OUT* pSimOut, char* sDataFile)
{
    TRIAL_DATA* pTrialData[lMaxThreads];
    double* pdArrRndNum[lMaxThreads];

    // Allocate memory to pTrialData and pdArrRndNum arrays here.
    // Any other initialization

    #pragma omp parallel for ordered
    for(long lSimInd = 0; lSimInd < lSimCount; lSimInd++)
    {
        #pragma omp ordered
        {
            GenerateRandomNumbers(lSampleSize, pdArrRndNum[lCurThread], lSeed);
        }

        GenerateTrialData(pTrialData[lCurThread], lSampleSize, pdArrRndNum[lCurThread]);

        #pragma omp critical (SUMMARIZE)
        {
            AnalyzeAndSummarize(pTrialData[lCurThread], pSimOut);
        }
    }
}
```



Generation of random numbers and trial data

Simplified Sample Code

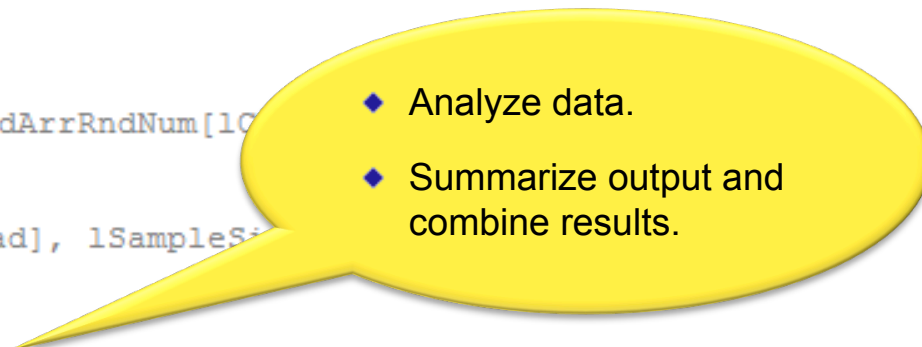
```
void PerformSimulations(long lSimCount, long lSampleSize, SIM_OUT* pSimOut, char* sDataFile)
{
    TRIAL_DATA* pTrialData[lMaxThreads];
    double* pdArrRndNum[lMaxThreads];

    // Allocate memory to pTrialData and pdArrRndNum arrays here.
    // Any other initialization

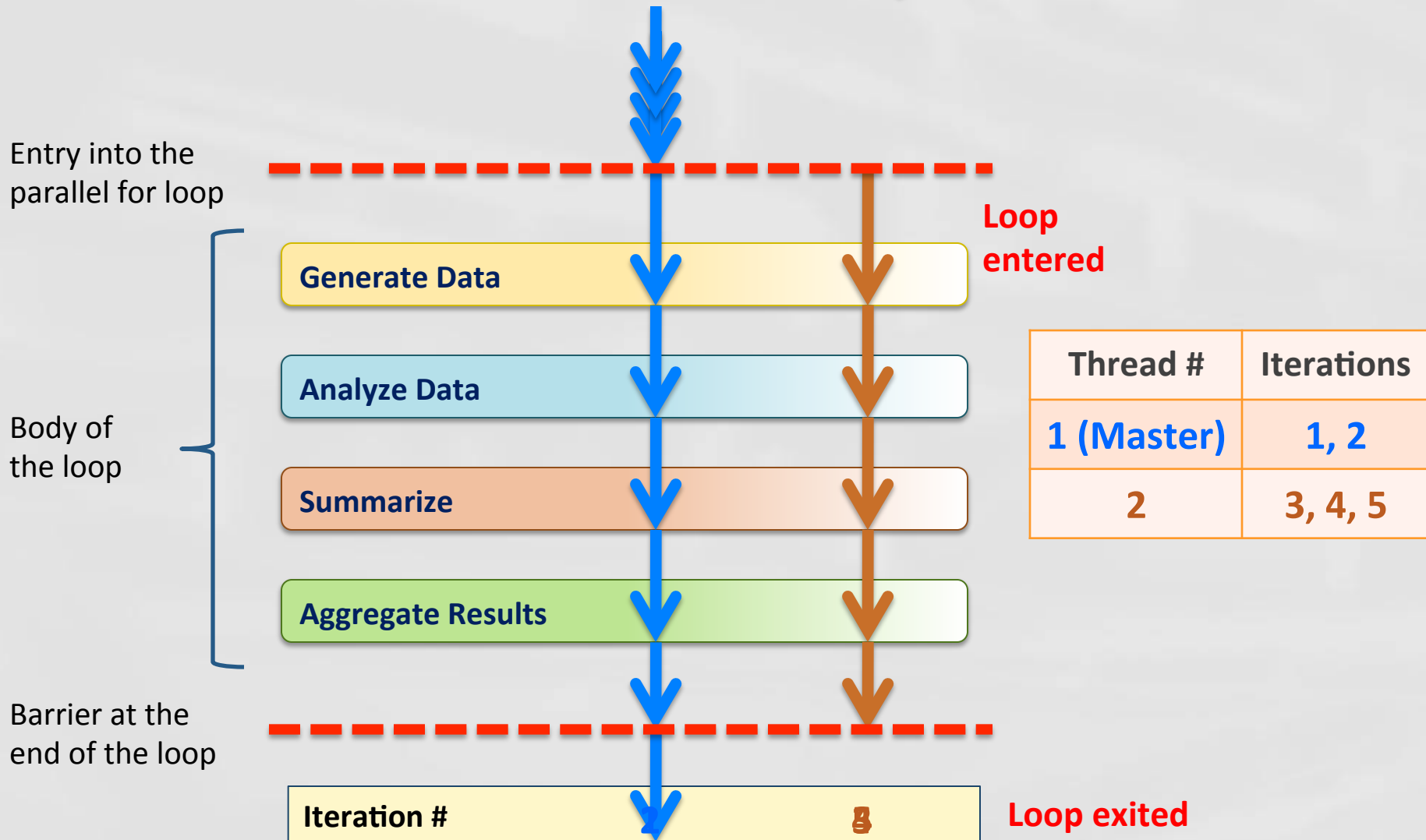
    #pragma omp parallel for ordered
    for(long lSimInd = 0; lSimInd < lSimCount; lSimInd++)
    {
        #pragma omp ordered
        {
            GenerateRandomNumbers(lSampleSize, pdArrRndNum[lCurThread]);

            GenerateTrialData(pTrialData[lCurThread], lSampleSize);

            #pragma omp critical (SUMMARIZE)
            {
                AnalyzeAndSummarize(pTrialData[lCurThread], pSimOut);
            }
        }
    }
}
```

- 
- ◆ Analyze data.
 - ◆ Summarize output and combine results.

Animation: 5 Iterations, 2 Threads



Pragma *omp parallel for*

- A work sharing directive
- Master thread creates 0 or more child threads. Loop iterations distributed among the threads.
- Implied barrier at the end of the loop, only master continues beyond.
- Clauses can be used for finer control – sharing variables among threads, maintaining order of execution, controlling distribution of iterations among threads etc.

Thread Synchronization

Example – Random Number Generation

- For reproducibility of results -
 - Random number sequence must not change from run to run.
 - Random numbers must be drawn from the same stream across runs.
- Pragma *omp ordered* ensures that attached code is executed sequentially by threads.
- A thread executing a later iteration, waits for threads executing earlier iterations to finish with the ordered block.

Thread Synchronization

Example – Summarizing Output Across Simulations

- Output from simulations running on different threads needs to be summarized into a shared object.
- Simulation sequence does not matter.
- `Pragma omp critical` ensures that attached code is executed by any single thread at a time.
- A thread waits at the critical block if another thread is currently executing it.

Results with OpenMP

OpenMP - Performance Improvement

Results from SiZ^{®†}

Test #	# of Simulations	Sample Size	Time Required [‡]		% Improvement with OpenMP
			OpenMP Disabled	OpenMP Enabled	
1	85000	2100	6 mins, 40 secs	2 mins, 6 secs	69%
2	100000	3800	14 mins, 48 secs	6 mins, 38 secs	55%

[†] SiZ[®] - a design and simulation package for fixed sample size studies

[‡] Tests executed on a laptop with 3 GB RAM and a quad-core processor with a speed of 2.4 GHz

OpenMP - Performance Improvement

Results from SiZ^{®†}

Test #	# of Simulations	Sample Size	Time Required [‡]		% Improvement with OpenMP
			OpenMP Disabled	OpenMP Enabled	
1	85000	2100	6 mins, 40 secs	2 mins, 6 secs	69%
2	100000	3800	14 mins, 48 secs	6 mins, 38 secs	55%

[†] SiZ[®] - a design and simulation package for fixed sample size studies

[‡] Tests executed on a laptop with 3 GB RAM and a quad-core processor with a speed of 2.4 GHz

OpenMP - Performance Improvement

Results from SiZ^{®†}

Test #	# of Simulations	Sample Size	Time Required [‡]		% Improvement with OpenMP
			OpenMP Disabled	OpenMP Enabled	
1	85000	2100	6 mins, 40 secs	2 mins, 6 secs	69%
2	100000	3800	14 mins, 48 secs	6 mins, 38 secs	55%

[†] SiZ[®] - a design and simulation package for fixed sample size studies

[‡] Tests executed on a laptop with 3 GB RAM and a quad-core processor with a speed of 2.4 GHz

Other Parallelization Technologies

- Win32 API

- Create, manage and synchronize threads at a much lower level
- Generally involves much more coding compared to OpenMP

- MPI (Message Passing Interface)

- Supports distributed and cluster computing
- Generally considered difficult to program – program's data structures need to be partitioned and typically the entire program needs to be parallelized

Concluding Remarks

- OpenMP is simple, flexible and powerful.
- Supported on many architectures including Windows and Unix.
- Works on platforms ranging from the desktop to the supercomputer.
- Read the specs carefully, design properly and test thoroughly.

References

- OpenMP Website: <http://www.openmp.org>

For the complete OpenMP specification

- Parallel Programming in OpenMP

Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan,

Jeff McDonald, Ramesh Menon

Morgan Kaufmann Publishers

- OpenMP and C++: Reap the Benefits of Multithreading without All the Work

Kang Su Gatlin, Pete Isensee

<http://msdn.microsoft.com/en-us/magazine/cc163717.aspx>

Thank you!
Questions?

Email: aniruddha.deshmukh@cytel.com