FINM 32950: Intro to HPC in Finance

Lecture 3

Chanaka Liyanarachchi

April 4, 2025

Structured Parallel Programming

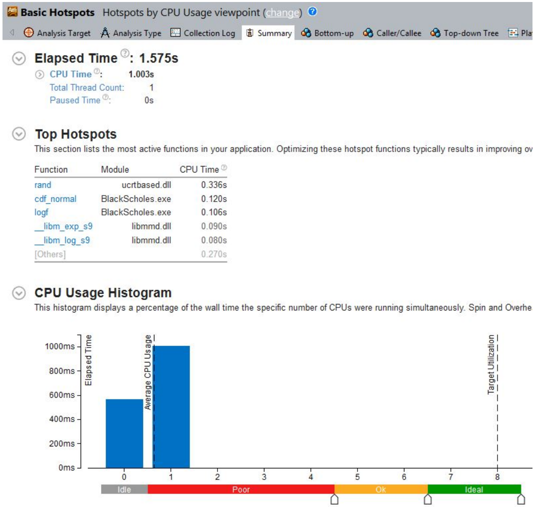
Road Map

We discussed some important concepts about multithreading:

- creating threads
- thread synchronization
- critical sections
- race conditions
- mutual exclusions
- locks
- deadlocks
- atomics

Problem

- Our goal: use threads to write parallel programs.
- Suppose we profile an application and see low CPU utilization:



Elapsed Time: 1.575s

CPU Time:	1.003s
Total Thread Count:	1
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving ov

Function	Module	CPU Time
rand	ucrtbased.dll	0.336s
cdf_normal	BlackScholes.exe	0.120s
logf	BlackScholes.exe	0.106s
__libm_exp_s9	libmmd.dll	0.090s
__libm_log_s9	libmmd.dll	0.080s
[Others]		0.270s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhe

- Which part/parts can be run in parallel?

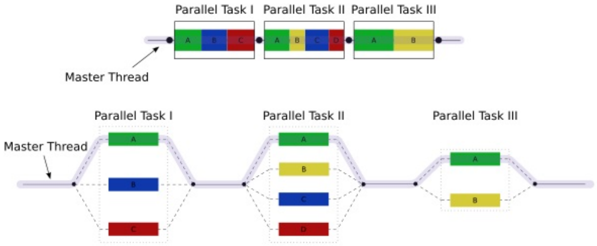
Pattern Based Approach

- We will learn how to do this in a structured way using a *pattern-based* approach.
- Patterns are widely used in software engineering.
- Capture best practices to solve common problems in a particular domain.
- *Parallel Patterns*: capture known solutions and best practices to solve known problems in parallel program design.
- We can use one or more parallel patterns to design a program.
- In this course, we will:
  - introduce some common parallel patterns
  - implement them (homework)
  - use them as high level design tools to write parallel programs

Parallel Patterns

Folk-join

- Very common programming pattern as shown below[1]

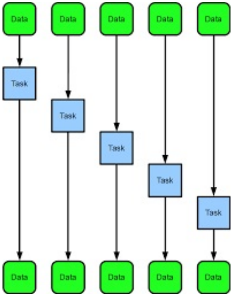


- Generally, this pattern involves:
  - serial work
  - parallel work
  - wait for all parallel work to finish
  - go to next task

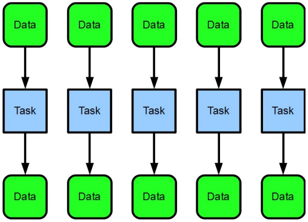
[1]figure source: wikipedia

Map

- Map pattern involves applying an operation to all elements in a sequence.
- This operation produces a new collection with the same shape as the input.
- Serial execution of this pattern takes the form of:



- Parallel execution of this pattern takes the form of:

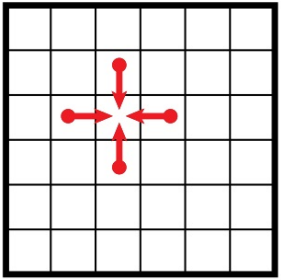


Map

- The operation is known as *elemental function*.
- The elemental function should not modify shared (global) data that other instances depend on.
- E.g.: Assignment A, Assignment B

Stencil

- Here the elemental function access an element in a sequence and its neighbors.



- E.g.: Moving average, finite difference.

Reduction

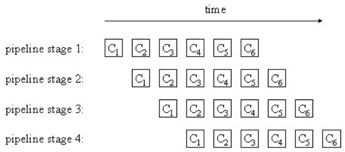
- A reduction combines elements in a collection into a single element.
- A simple example would be computing the sum of elements in a vector.
- A function used in reduction is known as the *combiner function*.
- E.g.: Monte Carlo.

Pipeline

- Pipeline is a very common example/concept in everyday life.
- E.g. Assembly line of a car factory.

- Performs a specific task on an item at each station.
- Move the item to the next station for the next task.

- Next item takes the place of an item when it leaves a station.
- All stations are busy.



- Widely used in Processor/FPGA designs.

Many Patterns

- We looked at some patterns we plan to use in this course.
- There are many more:
  - Scan
  - Scatter
  - Gather
  - Futures
  - Search
  - ...
- A detailed discussion of all common parallel patterns is beyond the scope of this course.
- Recommended reading: *Structured Parallel Programming*[2] is a great resource for further studies on the subject.

[2]See course references for details

Assignment 1 - Required

- Due: April 12 by midnight.
- Implement any 2 patterns below using threads (std::thread) to demonstrate how the pattern works.
    - Folk-join
    - Map
    - Stencil
    - Reduction
- Your program doesn't have to do anything *meaningful*. Great if it does.
- For example, an elemental function can just be a `cout` statement.

Using Threads: Difficulties

- We have seen how to use threads to do work in parallel.
- Using std::thread is not easy (low level construct):
  - How many threads to use is not clear
  - Load balancing difficult
  - Doesn't scale
- Topics we discussed are very important:
  - Need a good foundation to understand how threads are used to write parallel programs
  - Learn related concepts: critical sections, data races, locks etc.
- We are going to discuss a different way/approach to use threads next.

High Level Constructs for Parallel Programming

High Level Constructs for Concurrency

- We will look at some *high level programming constructs* for parallel programming.
- Express parallelism at high-level hiding low-level implementation details.
- Useful for the beginner and the experienced programmer, alike.
- Advantages:
  - Clarity
  - Ease of use (increase productivity)
  - Less error prone
  - Scalability

- Popular high-level constructs:
  1. OpenMP: very popular
  2. (Intel) Threading Building Blocks (TBB): many features
  3. (Microsoft) Parallel Patterns Library (PPL): features similar to TBB
  4. C++ Standard Library
- We don't have to use only one solution. We can mix-and-match.

OpenMP (**Open** specification for **M**ulti **P**rocessing)

OpenMP

- Very popular model.
- Provides a simple approach to develop parallel code, using:
    1. compiler directives
    2. functions
    3. environment variables
- We don't have to use all 3 of them (some features overlap).
- Implemented based on a specification[3] developed by industry participants, including, IBM, Intel, Oracle and several others[4].

[3] `www.openmp.org`

[4] Microsoft is no longer a member.

- Write parallel code in a platform (compiler, OS) neutral manner:
  - Works on many OSs:
    1. Windows
    2. (Various flavors of) Unix and Linux
  - Popular compiler support:
    1. Visual C++ supports OpenMP 2.0[5]
    2. `https://devblogs.microsoft.com/cppblog/improved-openmp-support-for-cpp-in-visual-studio/`
    3. Intel, gcc are more up-to-date.
  - Supports different programming languages:
    1. C++
    2. C
    3. Fortran

[5] `https://msdn.microsoft.com/en-us/library/0ca2w8dk.aspx`

OpenMP: Example 1

- Code below shows a simple *Hello World* program written using OpenMP directives:

```
#include <omp.h>

int main()
{
   #pragma omp parallel
   {
       cout << "Hello, world" << endl;
   }
}
```

- The OpenMP uses compiler directives to make regions of code execute in parallel:
  - We don't create threads explicitly
  - We instruct the compiler what to do
  - Compiler generates appropriate code
- OpenMP features are defined in `<omp.h>`.

Enabling OpenMP Support: Midway

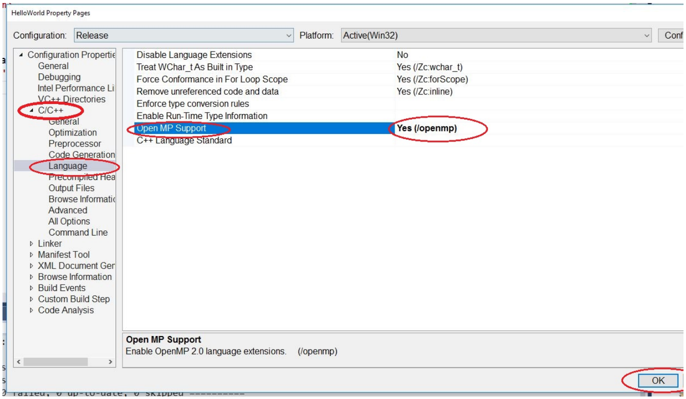
- Intel Compiler (Linux):
- use -qopenmp flag:

```
icc -qopenmp hello.cpp -o hello
```

- Note: You need to load intel module first (on midway).

Enabling OpenMP Support: Windows

- We also need to enable OpenMP support.
- Visual C++ (Windows):
  `Properties -> C/C++ -> Language -> OpenMP Support -> Yes`



HelloWorld Property Pages

Configuration: Release Platform: Active(Win32) Conf

Disable Language Extensions	No
Treat WChar_t As Built in Type	Yes (/Zc:wchar_t)
Force Conformance in For Loop Scope	Yes (/Zc:forScope)
Remove unreferenced code and data	Yes (/Zc:inline)
Enforce type conversion rules	
Enable Run-Time Type Information	
Open MP Support	**Yes (/openmp)**
C++ Language Standard	

Open MP Support
Enable OpenMP 2.0 language extensions. (/openmp)

OpenMP Directive Format

An OpenMP directive follows the format:

```
#pragma omp directive-name [clause]
```

- `#pragma` provides an mechanism to provide additional information to the compiler.
- `#pragma omp`: tells the compiler it is an OpenMP directive. This field is required.
- `directive-name`: a valid OpenMP directive must appear after the pragma. This field is required.
- `[clauses]`: tells the more about what has to be done. This field is optional.

OpenMP: Example 2

- For example, we can use an optional clause to specify the number of threads:

  ```
  #include <omp.h>

  int main()
  {
     #pragma omp parallel num_threads(2)
     {
         cout << "Hello, world" << endl;
     }
  }
  ```

OpenMP: Benefits

- OpenMP offers many features to write parallel code without using low level constructs.
- We still need to pay attention to:
  - Race conditions – when shared data members are accessible (read and write) by more than one thread
  - Dependencies – multiple threads in a block may run in any order
- References:
  - `https://www.openmp.org/`
  - Quick reference: `http://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf`
  - Text: *Using OpenMP*[6]

[6]See course references list for details

OpenMP Directives

Parallel:

- The `parallel` directive defines a parallel region:
  - At the beginning of the `parallel` directive block, worker threads are created
  - Each thread executes every instruction in the parallel block
  - At the end of the parallel block, all threads join (i.e. there is an implicit barrier).
- The order in which the threads run is not specified.
- To provide an explicit barrier:
  ```
  #pragma omp barrier
  ```

Parallel Loops:

- `parallel for`:
  - used to parallelize a for loop
    `#pragma omp parallel for`
- Example:

```
#pragma omp parallel for
for (int i=0; i<10; ++i)
{
   c[i] = a[i] + b[i];
}
```

OpenMP: Example 3

- Using OpenMP, multithreading Matrix multiplication is very easy. Thread-per-row can be implemented easily as follows:

```
void matrix_multiply_parallel(const matrix& m1,
        const matrix& m2,
        matrix& m3,
        int rows, int columns)
{
   #pragma omp parallel for
   for (int i = 0; i < rows; ++i)
   {
      for (int j = 0; j < columns; ++j)
      {
         m3[i][j] = 0;
         for (int k = 0; k < rows; ++k)
         {
            m3[i][j] += m1[i][k] * m2[k][j];
         }
      }
   }
}
```

- Exercise: Change the number of threads; observe execution times.

Sections

- Parallel for allows us to execute the same task on different data in parallel easily.
- We use sections to execute different tasks (different kinds of work) in parallel:
  - Specifies different code regions.
  - Each region is executed by one thread.
  - Each section should run independently of the other regions.

- Example:

```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      task1();

      #pragma omp section
      task2();
   }
}
```

- Here, task1 and task2 will be executed by two threads in parallel.

Thread Scheduling/Load Balancing

- OpenMP supports two thread scheduling policies:
  - static
  - dynamic
- Helpful in *load balancing*.
- Static scheduling (default):
  ```
  #pragma omp parallel for schedule(static)
  ```
  - Preallocates threads to loop indexes.
  - If one thread finishes first, the loop indexes won't be re-assigned.
- Dynamic scheduling:
  ```
  #pragma omp parallel for schedule(dynamic)
  ```
  - Allocates loop indexes to threads as threads become available to perform work.

- Loop below shows some uneven work.
- Let's use static scheduling first:

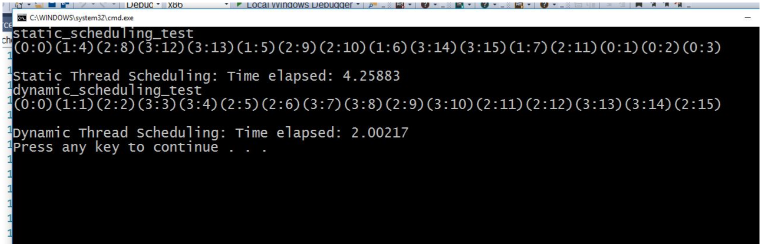
```
#pragma omp parallel for schedule(static) num_threads(4)
for (int i = 0; i<16; ++i)
{
   if (i<2)
   {
      Sleep(2000);  // simulate long work
   }
   else
   {
      Sleep(100);  // simulate short work
   }
   #pragma omp critical
   cout << "(" << omp_get_thread_num()
        << ":" << i << ")" << flush;
}
```

- A global lock (omp critical) is used to write to cout in an orderly fashion.

- And, dynamic scheduling next:

```
#pragma omp parallel for schedule(dynamic) num_threads(4)
for (int i = 0; i<16; ++i)
{
   if (i<2)
   {
      Sleep(2000);  // simulate long work
   }
   else
   {
      Sleep(100);  // simulate short work
   }
   #pragma omp critical
   cout << "(" << omp_get_thread_num()
         << ":" << i << ")" << flush;

}
```

- Static scheduling:
  - Each thread completes 4 tasks each
  - Number of work items equally shared
- Dynamic scheduling:
  - Allocates loop indexes to threads as threads become available to perform work
  - Some threads do more work items than others

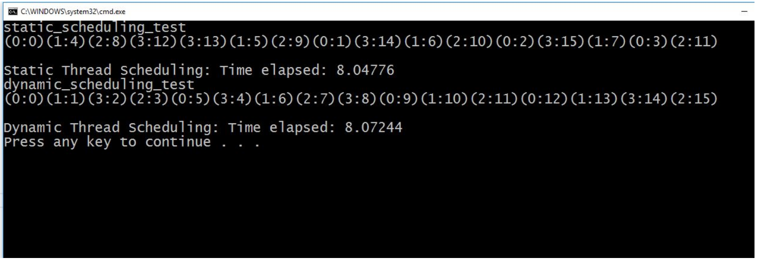


```
static_scheduling_test
(0:0)(1:4)(2:8)(3:12)(3:13)(1:5)(2:9)(2:10)(1:6)(3:14)(3:15)(1:7)(2:11)(0:1)(0:2)(0:3)

Static Thread Scheduling: Time elapsed: 4.25883
dynamic_scheduling_test
(0:0)(1:1)(2:2)(3:3)(3:4)(2:5)(2:6)(3:7)(3:8)(2:9)(3:10)(2:11)(2:12)(3:13)(3:14)(2:15)

Dynamic Thread Scheduling: Time elapsed: 2.00217
Press any key to continue . . .
```

- Dynamic scheduling works better for uneven work.

- If we have similar work, static scheduling works better:

```
#pragma omp parallel for schedule(static) num_threads(4)
for (int i = 0; i<16; ++i)
{
    Sleep(2000);  // simulate work

    #pragma omp critical(log)
    cout << "(" << omp_get_thread_num()
        << ":" << i << ")" << flush;

}
```



```
C:\WINDOWS\system32\cmd.exe
static_scheduling_test
(0:0)(1:4)(2:8)(3:12)(3:13)(1:5)(2:9)(0:1)(3:14)(1:6)(2:10)(0:2)(3:15)(1:7)(0:3)(2:11)

Static Thread Scheduling: Time elapsed: 8.04776
dynamic_scheduling_test
(0:0)(1:1)(3:2)(2:3)(0:5)(3:4)(1:6)(2:7)(3:8)(0:9)(1:10)(2:11)(0:12)(1:13)(3:14)(2:15)

Dynamic Thread Scheduling: Time elapsed: 8.07244
Press any key to continue . . .
```

- We're using a named lock (omp critical(log)) here to write to cout in an orderly fashion.

Synchronization Constructs

- `barrier`:
  - Provides a point of synchronization for threads in the group.
- `critical`:
  - A critical section without a name acts as a global lock
- `critical(lock_name)`:
  - A named critical section serializes execution with other critical sections with the same name
  - Critical sections with different names can run concurrently
- `master`:
  - This code block is executed by the master thread only
- atomic: Next slide

- An operation to be executed atomically (all or none, without interruptions).
- Operation supports additional clauses: read, write, update and capture.
- Counter example (see demo for details):

```
#pragma omp parallel for
for (int i = 0; i < 100000; ++i)
{
   #pragma omp atomic
   counter++;
}
```

Reductions

- Shown below is a simple example for reduction:

```
sum = 0;
for (int i = 0; i < 10; i++)
{
    sum += a[i];
}
```

- `sum` is a shared variable – doing this in parallel is not trivial.
- The OpenMP reduction clause is used to accumulate a value in parallel easily.

```
int sum_sq = 0;

#pragma omp parallel for reduction(+:sum_sq)
for (int i = 0; i < 5; ++i)
{
   sum_sq += i*i;
}
std::cout << sum_sq << std::endl;
```

- Reduction is supported for following operators:
  `+, -, *, &, |, ^, &&, ||`

Multithreading Examples

Matrix Multiplication: Serial to Parallel

- We matrix multiplication to illustrate parallel programming models.
- Serial:

```
void MatrixMultiply(const matrix& m1, const matrix& m2,
    matrix& m3,
    int rows, int columns)
{
    for (int i = 0; i < rows; ++i)
    {
        CalculateRow(/* args */);
    }
}
```

- Using std::thread:

```
void MatrixMultiply(const matrix& m1, const matrix& m2,
    matrix& m3,
    int rows, int columns)
{
    vector<thread> threads(rows);

    for (int i = 0; i < rows; ++i)
    {
        threads[i] = thread([&]()
        {
            CalculateRow(/* args */);
        });
    }

    for (thread& t : threads)
    {
        t.join();
    }
}
```

- Using OpenMP:

```
void MatrixMultiply(const matrix& m1, const matrix& m2,
    matrix& m3,
    int rows, int columns)
{
    #pragma omp parallel for
    for (int i = 0; i < rows; ++i)
    {
        CalculateRow(/* args */);
    }
}
```

- OpenMP solution has following advantages:
  - Clarity
  - Ease of use (increase productivity)
  - Less error prone
  - Scalability

Intel Parallel Programming Models

- Intel toolkit offers a wide range of support for high-performance and parallel computing.
- We've seen some features so far:
    - Tools: e.g. vtune for profiling, advisor for vectorization help
    - SVML
- Now, we will introduce two useful features in the Intel toolkit and illustrate how to use them in parallel Monte Carlo simulations:
    - Threading Building Blocks (TBB)
    - Math Kernel Library (MKL)

Threading Building Blocks (TBB)

TBB

- A library solution from Intel for high level *task based* parallel programming .
- Supports a wide range of features useful for parallel programming.
- Developer guide: `https://www.intel.com/content/www/us/en/docs/onetbb/developer-guide-api-reference/2021-6/onetbb-developer-guide.html`

Task Based Programming

- The task group class can be used to run tasks in parallel.

```
tbb::task_group g;

g.run( [&]() { work1(); } );
g.run( [&]() { work2(); } );
g.run( [&]() { work3(); } );

g.wait();
```

Detour: Function Objects (C++)

Function Objects

- Function objects are objects that behave like functions.
- We get this behavior when we overload `operator()` for the objects of a class.
- Function objects can have other member functions and member variables – they are functions with state.
- Function objects are used in various places in C++.
- We're going to use is them with threads/TBB.

Creating a Thread

- We can create a thread using a function object:

```
class ThreadTask
{
private:
   int threadID_;

public:
   ThreadTask(int id)
      : threadID_(id)
   {}

   void operator()() const
   {
      std::cout << "Hello, World, " << threadID_ << std::endl;
   }
};
```

- Usage:

```
std::thread t(ThreadTask(2));
t.join();
```

- We're passing an object to thread constructor.
- Thread executes the overloaded `operator()`.

Task Based Programming

- With TBB tasks:

```
const int NumTasks = 10;

tbb::task_group group;

for (int i = 0; i < NumTasks; ++i)
{
   group.run(ThreadTask(i));
}

group.wait();
```

Task Based Programming: Advantages

- Tasks allow us to design a program using logical (high level) tasks.
- As we saw earlier, the number of threads we use is important:
  - Create too many – poor performance due to overhead
  - Don't create enough – wasting cpu resources
  - Ideal number depends on the system
- When we use tasks, we don't need to figure out the exact number of threads.

Task scheduler:

- Creates the correct number of threads based on system resources
- Does load balancing
- Manages threads internally:
    - Doesn't create a thread every time we want to run task
    - Less thread creation overhead (i.e. creating a thread can be expensive compared to tasks)
- Schedules threads more efficiently:
    - Threads – OS uses fair scheduling; can be less efficient for some cases
    - Task scheduler claims to schedules threads more "intelligently" (using unfair scheduling). e.g. doesn't start a task until it can make progress

Monte Carlo

- We know how to write a serial version of Monte Carlo Option pricer to price European call options:

```
for (unsigned int i = 0; i < M; ++i)
{
   double z_i = generate_random_number();

   double ST_i = S0*exp((r - sigma*sigma / 2.0)*T
            + sigma*z_i*sqrt(T));

   sumPayoffs += (ST_i > K) ? ST_i - K: 0.0;
}

return exp(-r*T)*(sumPayoffs / M);
```

Generating Random Numbers

- We have many options to generate pseudo random numbers (discussed in Computing for Finance):
  - Implement an algorithm, e.g. Box Muller
  - Use generators in the C++ Standard Library
- Above methods generate random numbers sequentially.

Random Number Generation in MKL

- Intel MKL (Math Kernel Library)[7] provides efficient (pseudo) random number generators.
- Generates random numbers in serial/parallel mode.
- Generates a sequence of random numbers using one call.
- To generate random numbers using MKL:
  1. Create and initialize a stream
  2. Generate random numbers
  3. Delete the stream
- Related functions are defined in `<mkl_vsl.h>`

[7] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html#gs.aoqzyf`

RNG: Example 1

- Create a stream:
  ```
  VSLStreamStatePtr stream;
  ```
- Initializer:
  - Takes the stream, a uniform random number generator, and a seed as arguments.
  - The seed initializes the generator and allows us to generate the same, different or unique sequences.

  ```
  int seed = 777;
  vslNewStream( &stream, VSL_BRNG_MT19937, seed );
  ```

  We are using the Mersenne Twister pseudo-random generator above.
- Full list of supported generators:
  `https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-1/basic-generators.html`

- This example shows how to generate random numbers from the standard normal distribution:[8]
  - Here we use `ICDF` (Inverse Cumulative Distribution Function)
  - Generate 1000 random numbers from N(0, 1) and store them in an (`rands`) array

```
float mean = 0;
float stdev = 1.0f;

float* rands = new float[1000];
vsRngGaussian( VSL_RNG_METHOD_GAUSSIAN_ICDF,
                       stream,
                       1000, rands,
                       mean, stdev);
```

[8] https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-1/random-number-generators-naming-conventions.html#TBL10-1

- Use the random numbers:

  ```
  for (int i=0; i<10; ++i)
  {
      cout << rands[i] << endl;
  }
  ```

- Delete the stream:

  ```
  vslDeleteStream(&stream);
  ```

MKL: Distributions

- MKL supports several distributions: `https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-1/distribution-generators.html`

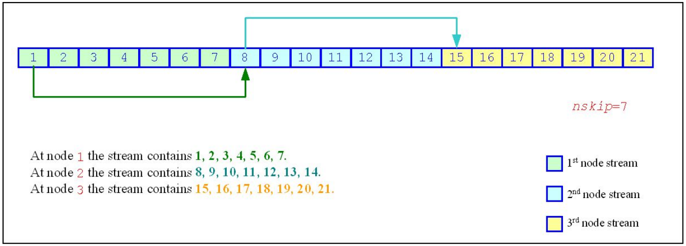
Parallel RNG

Parallel RNG

- Parallel simulations require us to use several streams.
    - E.g. each task/thread can use own parallel stream.
- Each parallel stream has to generate (pseudo) random numbers from the same underlying distribution.
    - I.e. each stream has to generate a non overlapping sub-sequence from the same underlying distribution.
- We can create such streams for parallel use, by:
    - Block splitting
    - Leap frogging

Block Splitting

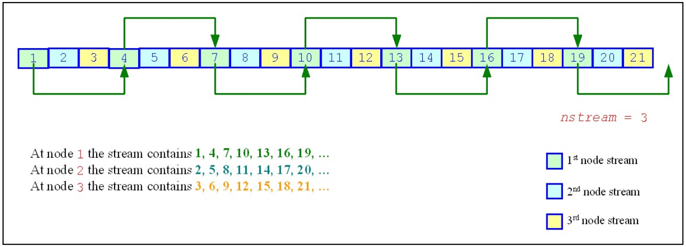
- Suppose the original sequence generates: x_1, x_2, x_3, x_n etc.
- One way to split an original sequence into non overlapping sub-sequences is:
  - 1^{st} stream generates: x_1, x_2, x_k
  - 2^{nd} stream generates: x_{k+1}, x_{k+2}, x_{2k}
  - 3^{rd} stream generates: $x_{2k+1}, x_{2k+2}, x_{3k}$ etc.
- Block splitting is to split a sequence into k non overlapping blocks, each generated by different processors.
- Example below shows how to split a sequence into 3 non overlapping streams using block-splitting.



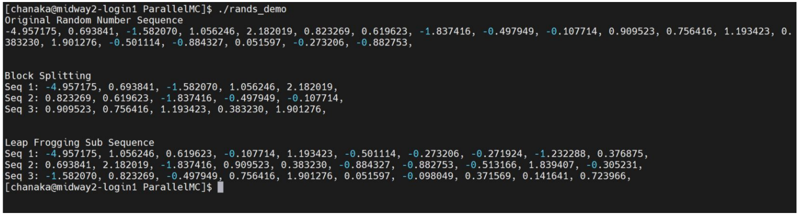
At node 1 the stream contains 1, 2, 3, 4, 5, 6, 7.
At node 2 the stream contains 8, 9, 10, 11, 12, 13, 14.
At node 3 the stream contains 15, 16, 17, 18, 19, 20, 21.

Leap Frogging

- Suppose the original sequence generates: x_1, x_2, x_3, x_n etc.
- Another way to split it into k non overlapping streams is to use how we deal a deck of cards to card players.
  - 1st stream generates: x_1, x_{k+1}, x_{2k+1}
  - 2nd stream generates: x_2, x_{k+2}, x_{2k+2}
  - 3rd stream generates: x_3, x_{k+3}, x_{2k+3} etc.
- This method is known as leap frogging.
- Example below shows how to split a sequence into 3 non overlapping streams using leap-frogging.


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
nstream = 3
At node 1 the stream contains 1, 4, 7, 10, 13, 16, 19, ...
At node 2 the stream contains 2, 5, 8, 11, 14, 17, 20, ...
At node 3 the stream contains 3, 6, 9, 12, 15, 18, 21, ...
1st node stream
2nd node stream
3rd node stream

- Actual example:



```
[chanaka@midway2-login1 ParallelMC]$ ./rands_demo
Original Random Number Sequence
-4.957175, 0.693841, -1.582070, 1.056246, 2.182019, 0.823269, 0.619623, -1.837416, -0.497949, -0.107714, 0.909523, 0.756416, 1.193423, 0.
383230, 1.901276, -0.501114, -0.884327, 0.051597, -0.273206, -0.882753,


Block Splitting
Seq 1: -4.957175, 0.693841, -1.582070, 1.056246, 2.182019,
Seq 2: 0.823269, 0.619623, -1.837416, -0.497949, -0.107714,
Seq 3: 0.909523, 0.756416, 1.193423, 0.383230, 1.901276,


Leap Frogging Sub Sequence
Seq 1: -4.957175, 1.056246, 0.619623, -0.107714, 1.193423, -0.501114, -0.273206, -0.271924, -1.232288, 0.376875,
Seq 2: 0.693841, 2.182019, -1.837416, 0.909523, 0.383230, -0.884327, -0.882753, -0.513166, 1.839407, -0.305231,
Seq 3: -1.582070, 0.823269, -0.497949, 0.756416, 1.901276, 0.051597, -0.098049, 0.371569, 0.141641, 0.723966,
[chanaka@midway2-login1 ParallelMC]$ █
```

- See demo for details.

Using MKL and TBB on Midway

We need load intel and mkl modules first:

```
module use /software/intel/oneapi_hpc_2022.1/modulefiles
module load intel/2022.0
module load mkl/latest
```

- To enable MKL:

  ```
  -qmkl[=<arg>]
     link to the Intel Intel MKL and bring
     in the associated headers
        parallel   - link using the threaded Intel
               MKL libraries. This is the default
               when -qmkl is specified
        sequential - link using the non-threaded
             Intel MKL libraries
  ```

  ```
   icc -qmkl rands_demo.cpp -o rands_demo
  ```

- Additionally, if you use TBB, to load TBB:

  ```
  module load tbb/latest
  ```

  To compile using tbb:

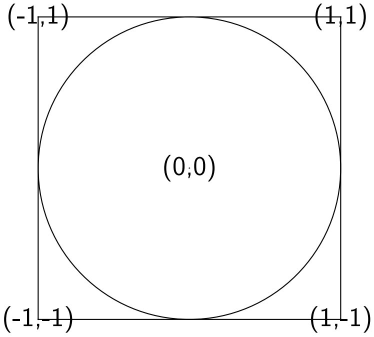
  ```
  icc -qtbb ....
  ```

Block Splitting vs Leap Frogging

- "Not all BRNGs support both these methods of generating independent sub-sequences. "
  – `https://cdrdv2-public.intel.com/671322/vslnotes-2.pdf`

Example: Monte Carlo π

- You wrote a program (Assignment D) to estimate π using Monte Carlo by:
  - The figure below shows a circle with radius $r = 1$ inscribed within a square.



  - Pick N random points (given by two random x, y coordinates) within the square.
  - Count the number of points inside the circle.

- Code below uses parallel random number streams to estimate the value of π using Monte Carlo:
  - Split the simulation into tasks
  - Each task uses a parallel stream to generate random numbers
  - A task is implemented as a function object
  - Use TBB task_group to manage parallel tasks

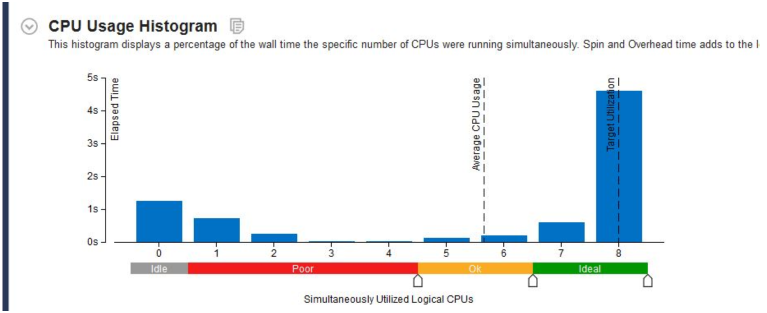
```
tbb::task_group group;
long samples_per_task = total_samples / num_tasks;
long points_inside_circle_per_task[num_tasks];

for (int i = 0; i < num_tasks; i++)
{
   group.run(PiTask(samples_per_task,
           stream[i],
           points_inside_circle_per_task[i]));
}

group.wait();
```

- See demo for details. To build:

```
icc -qmkl -qtbb mc_pi.cpp -o mc_pi
```

CPU Utilization

- Program uses more than one CPU core now:



CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the I