# User

Can you define c++ code to mimic dask like parallel programming using workers and threads to generate random numbers using rdrand\_step64 intel intrinsic function and give me compilation under msys using mingw64 tool chain (gcc / g++). I wish you to use boost library for the coding. Please note that I want to compile the code as shared dll.

Input shall be: (1) number of random numbers to be generated, (2) No of workers, (3) number of threads

Output shall be array of random numbers.

# Bing

Sure, here's a basic example of how you might structure your code using the Boost library to create a multi-threaded program that generates random numbers. Please note that this is a simplified example and does not include error checking or optimization.

#include <boost/thread.hpp>

#include <boost/bind.hpp>

#include <iostream>

#include <vector>

#include <immintrin.h>

class Worker {

public:

Worker(int numNumbers) : numNumbers(numNumbers) {}

void operator()() {

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

unsigned long long randomNumber;

\_rdrand64\_step(&randomNumber);

numbers.push\_back(randomNumber);

}

}

std::vector<unsigned long long> getNumbers() {

return numbers;

}

private:

int numNumbers;

std::vector<unsigned long long> numbers;

};

int main(int argc, char\* argv[]) {

int numNumbers = std::stoi(argv[1]);

int numWorkers = std::stoi(argv[2]);

int numThreads = std::stoi(argv[3]);

boost::thread\_group threads;

std::vector<Worker> workers;

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

workers.push\_back(Worker(numNumbers));

threads.create\_thread(boost::ref(workers.back()));

}

threads.join\_all();

std::vector<unsigned long long> allNumbers;

for(auto& worker : workers) {

std::vector<unsigned long long> numbers = worker.getNumbers();

allNumbers.insert(allNumbers.end(), numbers.begin(), numbers.end());

}

for(auto num : allNumbers) {

std::cout << num << std::endl;

}

return 0;

}

To compile this code into a shared library using the MinGW-w64 toolchain, you can use the following command:

g++ -shared -o mylib.dll mylib.cpp -lboost\_thread-mt -lboost\_system-mt

Please note that you need to replace mylib.cpp with the name of your source file, and mylib.dll with the desired name of your output DLL file. Also, make sure that the Boost library is correctly installed and the paths are set correctly in your environment.

This code uses the \_rdrand64\_step function, which is an Intel intrinsic function for generating random numbers. It creates a number of worker threads, each of which generates a specified number of random numbers. The numbers are then collected and printed out.

Please note that this is a very basic example and does not include any error checking or exception handling. In a real-world application, you would want to add appropriate error checking and handle any exceptions that might occur. Also, the \_rdrand64\_step function requires a processor that supports the RDRAND instruction, which is available on certain Intel processors. If you're running this code on a processor that does not support the RDRAND instruction, it will not work.

# User

You have missed out switch -mrdrnd in compilation command

# Bing

You're absolutely right. I apologize for the oversight. The -mrdrnd switch is necessary to enable the use of the RDRAND instruction set. Here's the corrected command:

g++ -shared -o mylib.dll mylib.cpp -lboost\_thread-mt -lboost\_system-mt -mrdrnd

This command will compile the code into a shared library with the RDRAND instruction set enabled. Thank you for pointing out the mistake. I appreciate your help!

# User

Do I need to use switch lpthread also or not required in compilation?

# Bing

The -lpthread switch is used to link the POSIX threads library. However, when using the Boost.Thread library on a MinGW-w64 toolchain, you typically don't need to explicitly link the pthreads library because Boost.Thread takes care of threading support for you.

So, in your case, you should not need to include -lpthread in your compilation command. Your command would remain as:

g++ -shared -o mylib.dll mylib.cpp -lboost\_thread-mt -lboost\_system-mt -mrdrnd

Please note that this might vary depending on your specific setup and configuration. If you encounter any issues, please let me know! I'm here to help.

# User

Following compilation error:

In file included from D:/Programs/Msys2/mingw64/include/boost/smart\_ptr/detail/requires\_cxx11.hpp:9,
from D:/Programs/Msys2/mingw64/include/boost/smart\_ptr/shared\_ptr.hpp:17,
from D:/Programs/Msys2/mingw64/include/boost/shared\_ptr.hpp:17,
from D:/Programs/Msys2/mingw64/include/boost/date\_time/time\_clock.hpp:17,
from D:/Programs/Msys2/mingw64/include/boost/thread/thread\_time.hpp:9,
from D:/Programs/Msys2/mingw64/include/boost/thread/win32/thread\_data.hpp:10,
from D:/Programs/Msys2/mingw64/include/boost/thread/thread\_only.hpp:15,
from D:/Programs/Msys2/mingw64/include/boost/thread/thread.hpp:12,
from D:/Programs/Msys2/mingw64/include/boost/thread.hpp:13,
from boost\_rdrand.cpp:1:
D:/Programs/Msys2/mingw64/include/boost/bind.hpp:36:1: note: '#pragma message: The practice of declaring the Bind placeholders (\_1, \_2, ...) in the global namespace is deprecated. Please use <boost/bind/bind.hpp> + using namespace boost::placeholders, or define BOOST\_BIND\_GLOBAL\_PLACEHOLDERS to retain the current behavior.'
36 | BOOST\_PRAGMA\_MESSAGE(
| ^~~~~~~~~~~~~~~~~~~~

# Bing

The message you're seeing is a deprecation warning, not an error. It's informing you that declaring the Boost.Bind placeholders (\_1, \_2, etc.) in the global namespace is deprecated and suggesting two alternatives:

1. Use <boost/bind/bind.hpp> and using namespace boost::placeholders, or
2. Define BOOST\_BIND\_GLOBAL\_PLACEHOLDERS to retain the current behavior.

This warning shouldn't prevent your code from compiling or running, but it's a good idea to address it to ensure your code remains compatible with future versions of Boost.

If you want to suppress this warning, you can define BOOST\_BIND\_GLOBAL\_PLACEHOLDERS before including any Boost headers:

#define BOOST\_BIND\_GLOBAL\_PLACEHOLDERS

#include <boost/thread.hpp>

// rest of your includes and code...

Or, if you want to follow the recommended practice, you can include <boost/bind/bind.hpp> and use boost::placeholders:

#include <boost/bind/bind.hpp>

#include <boost/thread.hpp>

// rest of your includes...

// in your code, use boost::placeholders::\_1, boost::placeholders::\_2, etc.

Please try these suggestions and let me know if you have any other questions! I'm here to help.

# User

Will compilation command need additional switch if I include:
#include <boost/bind/bind.hpp>

# Bing

No, including #include <boost/bind/bind.hpp> in your code does not require any additional switches in the compilation command. The compilation command remains the same:

g++ -shared -o mylib.dll mylib.cpp -lboost\_thread-mt -lboost\_system-mt -mrdrnd

This command should work as long as the Boost library is correctly installed and the paths are set correctly in your environment. If you encounter any issues, please let me know! I'm here to help.

# User

The compilation was successful. Give me another cpp code that calls the function from this dll with all input parameters and give me compilation command which will result into exe file.

# Bing

Sure, here's an example of how you might write a C++ program that loads the DLL and calls the function. This example assumes that the function in the DLL is named generateRandomNumbers and takes three parameters: the number of random numbers to generate, the number of workers, and the number of threads.

#include <windows.h>

#include <iostream>

#include <vector>

typedef std::vector<unsigned long long> (\*GenerateRandomNumbersFunc)(int, int, int);

int main(int argc, char\* argv[]) {

HINSTANCE hGetProcIDDLL = LoadLibrary("mylib.dll");

if (!hGetProcIDDLL) {

std::cout << "Could not load the dynamic library." << std::endl;

return EXIT\_FAILURE;

}

GenerateRandomNumbersFunc generateRandomNumbers = (GenerateRandomNumbersFunc) GetProcAddress(hGetProcIDDLL, "generateRandomNumbers");

if (!generateRandomNumbers) {

std::cout << "Could not locate the function." << std::endl;

return EXIT\_FAILURE;

}

int numNumbers = std::stoi(argv[1]);

int numWorkers = std::stoi(argv[2]);

int numThreads = std::stoi(argv[3]);

std::vector<unsigned long long> numbers = generateRandomNumbers(numNumbers, numWorkers, numThreads);

for (auto num : numbers) {

std::cout << num << std::endl;

}

FreeLibrary(hGetProcIDDLL);

return EXIT\_SUCCESS;

}

You can compile this code into an executable using the following command:

g++ -o myprogram myprogram.cpp -lboost\_thread-mt -lboost\_system-mt -mrdrnd

This command will compile the code into an executable named myprogram. You can run this program with the number of random numbers to generate, the number of workers, and the number of threads as command-line arguments.

Please note that this

# User

Regenerate.

# Bing

I apologize for the abrupt end in my previous message. Here's the complete response:

Sure, here's an example of how you might write a C++ program that loads the DLL and calls the function. This example assumes that the function in the DLL is named generateRandomNumbers and takes three parameters: the number of random numbers to generate, the number of workers, and the number of threads.

#include <windows.h>

#include <iostream>

#include <vector>

typedef std::vector<unsigned long long> (\*GenerateRandomNumbersFunc)(int, int, int);

int main(int argc, char\* argv[]) {

HINSTANCE hGetProcIDDLL = LoadLibrary("mylib.dll");

if (!hGetProcIDDLL) {

std::cout << "Could not load the dynamic library." << std::endl;

return EXIT\_FAILURE;

}

GenerateRandomNumbersFunc generateRandomNumbers = (GenerateRandomNumbersFunc) GetProcAddress(hGetProcIDDLL, "generateRandomNumbers");

if (!generateRandomNumbers) {

std::cout << "Could not locate the function." << std::endl;

return EXIT\_FAILURE;

}

int numNumbers = std::stoi(argv[1]);

int numWorkers = std::stoi(argv[2]);

int numThreads = std::stoi(argv[3]);

std::vector<unsigned long long> numbers = generateRandomNumbers(numNumbers, numWorkers, numThreads);

for (auto num : numbers) {

std::cout << num << std::endl;

}

FreeLibrary(hGetProcIDDLL);

return EXIT\_SUCCESS;

}

You can compile this code into an executable using the following command:

g++ -o myprogram myprogram.cpp -lboost\_thread-mt -lboost\_system-mt -mrdrnd

This command will compile the code into an executable named myprogram. You can run this program with the number of random numbers to generate, the number of workers, and the number of threads as command-line arguments.

Please note that this is a basic example and does not include any error checking or exception handling. In a real-world application, you would want to add appropriate error checking and handle any exceptions that might occur. Also, the generateRandomNumbers function must be exported from the DLL in order for this code to work. If you encounter any issues, please let me know! I'm here to help.

# User

Compilation was successfule but execution resulted in function not found error. So I ran gdb. See the output:

(gdb) run testcpp
Starting program: testcpp
No executable specified, use `target exec'.
(gdb) file testcpp
Reading symbols from testcpp...
(gdb) run
Starting program: D:\Programs\Msys2\home\juhi123\Downloads\testcpp.exe testcpp
[New Thread 6076.0x24e8]
[New Thread 6076.0x3d30]
[New Thread 6076.0x30d4]
Could not locate the function.
[Thread 6076.0x24e8 exited with code 1]
[Thread 6076.0x3d30 exited with code 1]
[Thread 6076.0x30d4 exited with code 1]
[Inferior 1 (process 6076) exited with code 01]

# Bing

The error message "Could not locate the function" suggests that the function generateRandomNumbers was not found in the DLL. This could be due to a few reasons:

1. **Function Export**: The function might not have been exported correctly from the DLL. In C++, you need to declare functions that you want to export with \_\_declspec(dllexport). For example:
2. The extern "C" part is necessary to prevent the compiler from mangling the name of the function. It's also important to ensure that the function is defined in the global scope (not inside a class or namespace), as GetProcAddress can only find globally scoped functions.
3. **Function Signature**: The function signature of the function in the DLL must match exactly with the function signature in the program that's loading the DLL. This includes the return type and the types of all parameters.
4. **Calling Convention**: The calling convention (\_\_cdecl, \_\_stdcall, etc.) of the function in the DLL must match the calling convention in the program that's loading the DLL. If you're not explicitly specifying a calling convention, then it defaults to \_\_cdecl.
5. **Linker Optimization**: Sometimes, the linker might optimize away a function if it doesn't see it being used. To prevent this, you can use a .def file to tell the linker which functions to export, or you can use the /EXPORT linker option.

Please check these points in your DLL code and the code that's loading the DLL. If everything seems correct and you're still getting the error, there might be an issue with the way the DLL is being loaded or with the DLL itself. In that case, you might want to use a tool like Dependency Walker to inspect the DLL and see what functions it exports.

# User

Where should I include:

extern "C" \_\_declspec(dllexport) std::vector<unsigned long long> generateRandomNumbers(int, int, int);

in dll source?
#define BOOST\_BIND\_GLOBAL\_PLACEHOLDERS
//#include <boost/bind/bind.hpp>
#include <boost/thread.hpp>
#include <boost/bind.hpp>
#include <iostream>
#include <vector>
#include <immintrin.h>

class Worker {
public:
Worker(int numNumbers) : numNumbers(numNumbers) {}

void operator()() {

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

unsigned long long randomNumber;

\_rdrand64\_step(&randomNumber);

numbers.push\_back(randomNumber);

}

}

std::vector<unsigned long long> getNumbers() {

return numbers;

}

private:
int numNumbers;
std::vector<unsigned long long> numbers;
};

int main(int argc, char\* argv[]) {
int numNumbers = std::stoi(argv[1]);
int numWorkers = std::stoi(argv[2]);
int numThreads = std::stoi(argv[3]);

boost::thread\_group threads;

std::vector<Worker> workers;

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

workers.push\_back(Worker(numNumbers));

threads.create\_thread(boost::ref(workers.back()));

}

threads.join\_all();

std::vector<unsigned long long> allNumbers;

for(auto& worker : workers) {

std::vector<unsigned long long> numbers = worker.getNumbers();

allNumbers.insert(allNumbers.end(), numbers.begin(), numbers.end());

}

for(auto num : allNumbers) {

std::cout << num << std::endl;

}

return 0;

}

# Bing

The function generateRandomNumbers is not present in your current code. You need to define this function and make sure it's exported from the DLL. Here's how you can modify your code:

#define BOOST\_BIND\_GLOBAL\_PLACEHOLDERS

#include <boost/thread.hpp>

#include <boost/bind.hpp>

#include <iostream>

#include <vector>

#include <immintrin.h>

class Worker {

public:

Worker(int numNumbers) : numNumbers(numNumbers) {}

void operator()() {

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

unsigned long long randomNumber;

\_rdrand64\_step(&randomNumber);

numbers.push\_back(randomNumber);

}

}

std::vector<unsigned long long> getNumbers() {

return numbers;

}

private:

int numNumbers;

std::vector<unsigned long long> numbers;

};

extern "C" \_\_declspec(dllexport) std::vector<unsigned long long> generateRandomNumbers(int numNumbers, int numWorkers, int numThreads) {

boost::thread\_group threads;

std::vector<Worker> workers;

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

workers.push\_back(Worker(numNumbers));

threads.create\_thread(boost::ref(workers.back()));

}

threads.join\_all();

std::vector<unsigned long long> allNumbers;

for(auto& worker : workers) {

std::vector<unsigned long long> numbers = worker.getNumbers();

allNumbers.insert(allNumbers.end(), numbers.begin(), numbers.end());

}

return allNumbers;

}

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul\_reason\_for\_call, LPVOID lpReserved) {

switch (ul\_reason\_for\_call) {

case DLL\_PROCESS\_ATTACH:

case DLL\_THREAD\_ATTACH:

case DLL\_THREAD\_DETACH:

case DLL\_PROCESS\_DETACH:

break;

}

return TRUE;

}

In this code, I've added the generateRandomNumbers function and marked it for export with \_\_declspec(dllexport). I've also added a DllMain function, which is the entry point for the DLL. This function is called when the DLL is loaded and unloaded.

Please replace the main function with the generateRandomNumbers function in your DLL source code. The main function is not needed in a DLL, as DLLs do not have a main entry point like executables do.

After making these changes, you should be able to compile your DLL and have the generateRandomNumbers function be visible for other programs to use. If you encounter any issues, please let me know! I'm here to help.

# User

Remove main function

# Bing

Sure, here's your code with the main function removed:

#define BOOST\_BIND\_GLOBAL\_PLACEHOLDERS

#include <boost/thread.hpp>

#include <boost/bind.hpp>

#include <iostream>

#include <vector>

#include <immintrin.h>

class Worker {

public:

Worker(int numNumbers) : numNumbers(numNumbers) {}

void operator()() {

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

unsigned long long randomNumber;

\_rdrand64\_step(&randomNumber);

numbers.push\_back(randomNumber);

}

}

std::vector<unsigned long long> getNumbers() {

return numbers;

}

private:

int numNumbers;

std::vector<unsigned long long> numbers;

};

extern "C" \_\_declspec(dllexport) std::vector<unsigned long long> generateRandomNumbers(int numNumbers, int numWorkers, int numThreads) {

boost::thread\_group threads;

std::vector<Worker> workers;

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

workers.push\_back(Worker(numNumbers));

threads.create\_thread(boost::ref(workers.back()));

}

threads.join\_all();

std::vector<unsigned long long> allNumbers;

for(auto& worker : workers) {

std::vector<unsigned long long> numbers = worker.getNumbers();

allNumbers.insert(allNumbers.end(), numbers.begin(), numbers.end());

}

return allNumbers;

}

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul\_reason\_for\_call, LPVOID lpReserved) {

switch (ul\_reason\_for\_call) {

case DLL\_PROCESS\_ATTACH:

case DLL\_THREAD\_ATTACH:

case DLL\_THREAD\_DETACH:

case DLL\_PROCESS\_DETACH:

break;

}

return TRUE;

}

This code now only contains the generateRandomNumbers function and the DllMain function. The generateRandomNumbers function is marked for export with \_\_declspec(dllexport), so it should be visible to other programs that load this DLL. If you encounter any issues, please let me know! I'm here to help.

# User

Error:

boost\_rdrand.cpp:49:1: error: 'BOOL' does not name a type
49 | BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul\_reason\_for\_call, LPVOID lpReserved) {
| ^~~~

# Bing

I apologize for the confusion. The error is because the types BOOL, HMODULE, DWORD, and LPVOID are defined in the Windows-specific header file windows.h. You need to include this header file at the beginning of your source code. Here's the corrected code:

#define BOOST\_BIND\_GLOBAL\_PLACEHOLDERS

#include <windows.h>

#include <boost/thread.hpp>

#include <boost/bind.hpp>

#include <iostream>

#include <vector>

#include <immintrin.h>

class Worker {

public:

Worker(int numNumbers) : numNumbers(numNumbers) {}

void operator()() {

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

unsigned long long randomNumber;

\_rdrand64\_step(&randomNumber);

numbers.push\_back(randomNumber);

}

}

std::vector<unsigned long long> getNumbers() {

return numbers;

}

private:

int numNumbers;

std::vector<unsigned long long> numbers;

};

extern "C" \_\_declspec(dllexport) std::vector<unsigned long long> generateRandomNumbers(int numNumbers, int numWorkers, int numThreads) {

boost::thread\_group threads;

std::vector<Worker> workers;

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

workers.push\_back(Worker(numNumbers));

threads.create\_thread(boost::ref(workers.back()));

}

threads.join\_all();

std::vector<unsigned long long> allNumbers;

for(auto& worker : workers) {

std::vector<unsigned long long> numbers = worker.getNumbers();

allNumbers.insert(allNumbers.end(), numbers.begin(), numbers.end());

}

return allNumbers;

}

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul\_reason\_for\_call, LPVOID lpReserved) {

switch (ul\_reason\_for\_call) {

case DLL\_PROCESS\_ATTACH:

case DLL\_THREAD\_ATTACH:

case DLL\_THREAD\_DETACH:

case DLL\_PROCESS\_DETACH:

break;

}

return TRUE;

}

This should resolve the compilation error. If you encounter any further issues, please let me know! I'm here to help.

# User

Compiled okay. My testcpp code is below which also compiles okay bit there is runtime error:

$ ./testcpp
terminate called after throwing an instance of 'std::logic\_error'
what(): basic\_string: construction from null is not valid

Code is as under:

#include <windows.h>
#include <iostream>
#include <vector>

typedef std::vector<unsigned long long> (\*GenerateRandomNumbersFunc)(int, int, int);

int main(int argc, char\* argv[]) {
HINSTANCE hGetProcIDDLL = LoadLibrary("boost\_rdrand.dll");

if (!hGetProcIDDLL) {

std::cout << "Could not load the dynamic library." << std::endl;

return EXIT\_FAILURE;

}

GenerateRandomNumbersFunc generateRandomNumbers = (GenerateRandomNumbersFunc) GetProcAddress(hGetProcIDDLL, "generateRandomNumbers");

if (!generateRandomNumbers) {

std::cout << "Could not locate the function." << std::endl;

return EXIT\_FAILURE;

}

int numNumbers = std::stoi(argv[1]);

int numWorkers = std::stoi(argv[2]);

int numThreads = std::stoi(argv[3]);

std::vector<unsigned long long> numbers = generateRandomNumbers(numNumbers, numWorkers, numThreads);

for (auto num : numbers) {

std::cout << num << std::endl;

}

FreeLibrary(hGetProcIDDLL);

return EXIT\_SUCCESS;

}

# Bing

The error message "basic\_string: construction from null is not valid" suggests that a std::string is being constructed with a null pointer. In your case, this could be happening when the LoadLibrary function fails to load the DLL and returns a null pointer, and then you're trying to print the error message.

To fix this, you should check if LoadLibrary returns a null pointer and print an appropriate error message. Here's how you can modify your code:

#include <windows.h>

#include <iostream>

#include <vector>

typedef std::vector<unsigned long long> (\*GenerateRandomNumbersFunc)(int, int, int);

int main(int argc, char\* argv[]) {

HINSTANCE hGetProcIDDLL = LoadLibrary("boost\_rdrand.dll");

if (!hGetProcIDDLL) {

std::cerr << "Could not load the dynamic library." << std::endl;

return EXIT\_FAILURE;

}

GenerateRandomNumbersFunc generateRandomNumbers = (GenerateRandomNumbersFunc) GetProcAddress(hGetProcIDDLL, "generateRandomNumbers");

if (!generateRandomNumbers) {

std::cerr << "Could not locate the function." << std::endl;

return EXIT\_FAILURE;

}

int numNumbers = std::stoi(argv[1]);

int numWorkers = std::stoi(argv[2]);

int numThreads = std::stoi(argv[3]);

std::vector<unsigned long long> numbers = generateRandomNumbers(numNumbers, numWorkers, numThreads);

for (auto num : numbers) {

std::cout << num << std::endl;

}

FreeLibrary(hGetProcIDDLL);

return EXIT\_SUCCESS;

}

In this code, I've replaced std::cout with std::cerr for printing error messages. This is a good practice because std::cerr is intended for error messages and it's unbuffered, so the messages are printed immediately.

If you're still encountering issues, it might be helpful to print out the error code when LoadLibrary fails. You can do this using the GetLastError function:

if (!hGetProcIDDLL) {

DWORD errorCode = GetLastError();

std::cerr << "Could not load the dynamic library. Error code: " << errorCode << std::endl;

return EXIT\_FAILURE;

}

This will give you a more specific error message that can help in diagnosing the problem. If you encounter any further issues, please let me know! I'm here to help.

# User

But in testcode I do not see number of random numbers to be generated, workers and threads. Where have you defined these variables?

# Bing

In the test code, the number of random numbers to be generated, the number of workers, and the number of threads are passed as command-line arguments when you run the program. They are retrieved in the main function with these lines:

int numNumbers = std::stoi(argv[1]);

int numWorkers = std::stoi(argv[2]);

int numThreads = std::stoi(argv[3]);

Here, argv is an array of C-style strings that represent the command-line arguments, and argc is the number of command-line arguments. The std::stoi function is used to convert the strings to integers.

So, when you run the program, you should provide these three values as command-line arguments. For example:

./testcpp 1000 10 4

In this example, the program will generate 1000 random numbers using 10 workers and 4 threads. Please replace these numbers with the actual values you want to use. If you encounter any further issues, please let me know! I'm here to help.

# User

Okay actually error was that I did not supply arguments and that resulted in runtime error:

So I did not have to make any change in my source code as under.

#include <windows.h>
#include <iostream>
#include <vector>

typedef std::vector<unsigned long long> (\*GenerateRandomNumbersFunc)(int, int, int);

int main(int argc, char\* argv[]) {
HINSTANCE hGetProcIDDLL = LoadLibrary("boost\_rdrand.dll");

if (!hGetProcIDDLL) {

std::cout << "Could not load the dynamic library." << std::endl;

return EXIT\_FAILURE;

}

GenerateRandomNumbersFunc generateRandomNumbers = (GenerateRandomNumbersFunc) GetProcAddress(hGetProcIDDLL, "generateRandomNumbers");

if (!generateRandomNumbers) {

std::cout << "Could not locate the function." << std::endl;

return EXIT\_FAILURE;

}

int numNumbers = std::stoi(argv[1]);

int numWorkers = std::stoi(argv[2]);

int numThreads = std::stoi(argv[3]);

std::vector<unsigned long long> numbers = generateRandomNumbers(numNumbers, numWorkers, numThreads);

for (auto num : numbers) {

std::cout << num << std::endl;

}

FreeLibrary(hGetProcIDDLL);

return EXIT\_SUCCESS;

}

Now Lets modify dll code to genearate random numbers with conditions as they should be >=100000000000000 and <=999999999999999

# Bing

Sure, to generate random numbers within a specific range, you can use a simple scaling technique. Here's how you can modify the Worker class to generate random numbers between 100000000000000 and 999999999999999:

class Worker {

public:

Worker(int numNumbers) : numNumbers(numNumbers) {}

void operator()() {

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

unsigned long long randomNumber;

\_rdrand64\_step(&randomNumber);

randomNumber = randomNumber % 900000000000000 + 100000000000000;

numbers.push\_back(randomNumber);

}

}

std::vector<unsigned long long> getNumbers() {

return numbers;

}

private:

int numNumbers;

std::vector<unsigned long long> numbers;

};

In this code, I've added a line to scale the random number to the desired range. The expression randomNumber % 900000000000000 + 100000000000000 takes the remainder of dividing randomNumber by 900000000000000 (which gives a number between 0 and 899999999999999), then adds 100000000000000 to shift the range to between 100000000000000 and 999999999999999.

Please replace the Worker class in your DLL source code with this modified version. If you encounter any issues, please let me know! I'm here to help.

# User

After the above change there is segmentation fault. I think worker - thread parallel management is faulty. Can you check / modify. I think memory locations are accessed illegally.

# Bing

I apologize for the inconvenience. The segmentation fault might be due to concurrent access to the numbers vector from multiple threads. In C++, the standard library containers like std::vector are not thread-safe, which means you can't safely modify them from multiple threads at the same time without additional synchronization.

To fix this, you can use a boost::mutex to synchronize access to the numbers vector. Here's how you can modify the Worker class:

#include <boost/thread/mutex.hpp>

class Worker {

public:

Worker(int numNumbers) : numNumbers(numNumbers) {}

void operator()() {

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

unsigned long long randomNumber;

\_rdrand64\_step(&randomNumber);

randomNumber = randomNumber % 900000000000000 + 100000000000000;

boost::lock\_guard<boost::mutex> guard(mutex);

numbers.push\_back(randomNumber);

}

}

std::vector<unsigned long long> getNumbers() {

boost::lock\_guard<boost::mutex> guard(mutex);

return numbers;

}

private:

int numNumbers;

std::vector<unsigned long long> numbers;

boost::mutex mutex;

};

In this code, I've added a boost::mutex member to the Worker class. Before modifying the numbers vector, we lock the mutex with a boost::lock\_guard. This ensures that only one thread can modify the numbers vector at a time, preventing data races and other concurrency issues.

Please replace the Worker class in your DLL source code with this modified version. If you encounter any further issues, please let me know! I'm here to help.

# User

Errors dll compilation after mutex:

In file included from D:/Programs/Msys2/mingw64/include/c++/13.2.0/x86\_64-w64-mingw32/bits/c++allocator.h:33,
from D:/Programs/Msys2/mingw64/include/c++/13.2.0/bits/allocator.h:46,
from D:/Programs/Msys2/mingw64/include/c++/13.2.0/string:43,
from D:/Programs/Msys2/mingw64/include/boost/date\_time/c\_time.hpp:18,
from D:/Programs/Msys2/mingw64/include/boost/date\_time/time\_clock.hpp:16,
from D:/Programs/Msys2/mingw64/include/boost/thread/thread\_time.hpp:9,
from D:/Programs/Msys2/mingw64/include/boost/thread/win32/thread\_data.hpp:10,
from D:/Programs/Msys2/mingw64/include/boost/thread/thread\_only.hpp:15,
from D:/Programs/Msys2/mingw64/include/boost/thread/thread.hpp:12,
from D:/Programs/Msys2/mingw64/include/boost/thread.hpp:13,
from boost\_rdrand.cpp:3:
D:/Programs/Msys2/mingw64/include/c++/13.2.0/bits/new\_allocator.h: In instantiation of 'void std::\_\_new\_allocator<\_Tp>::construct(\_Up\*, \_Args&& ...) [with \_Up = Worker; \_Args = {Worker}; \_Tp = Worker
':
D:/Programs/Msys2/mingw64/include/c++/13.2.0/bits/alloc\_traits.h:537:17: required from 'static void std::allocator\_traits<std::allocator<\_CharT> >::construct(allocator\_type&, \_Up\*, \_Args&& ...) [with \_Up = Worker; \_Args = {Worker}; \_Tp = Worker; allocator\_type = std::allocator<Worker>]'
D:/Programs/Msys2/mingw64/include/c++/13.2.0/bits/vector.tcc:117:30: required from 'std::vector<\_Tp, \_Alloc>::reference std::vector<\_Tp, \_Alloc>::emplace\_back(\_Args&& ...) [with \_Args = {Worker}; \_Tp = Worker; \_Alloc = std::allocator<Worker>; reference = Worker&]'
D:/Programs/Msys2/mingw64/include/c++/13.2.0/bits/stl\_vector.h:1296:21: required from 'void std::vector<\_Tp, \_Alloc>::push\_back(value\_type&&) [with \_Tp = Worker; \_Alloc = std::allocator<Worker>; value\_type = Worker]'
boost\_rdrand.cpp:40:26: required from here
D:/Programs/Msys2/mingw64/include/c++/13.2.0/bits/new\_allocator.h:187:11: error: use of deleted function 'Worker::Worker(Worker&&)'
187 | { ::new((void *)\_\_p) \_Up(std::forward<\_Args>(\_\_args)...); }
| ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
boost\_rdrand.cpp:10:7: note: 'Worker::Worker(Worker&&)' is implicitly deleted because the default definition would be ill-formed:
10 | class Worker {
| ^~~~~~
boost\_rdrand.cpp:10:7: error: use of deleted function 'boost::mutex::mutex(const boost::mutex&)'
In file included from D:/Programs/Msys2/mingw64/include/boost/thread/detail/move.hpp:24,
from D:/Programs/Msys2/mingw64/include/boost/thread/detail/thread.hpp:16,
from D:/Programs/Msys2/mingw64/include/boost/thread/thread\_only.hpp:22:
D:/Programs/Msys2/mingw64/include/boost/thread/win32/mutex.hpp:29:9: note: declared here
29 | BOOST\_THREAD\_NO\_COPYABLE(mutex)
| ^~~~~~~~~~~~~~~~~~~~~~~~
In file included from D:/Programs/Msys2/mingw64/include/c++/13.2.0/memory:69,
from D:/Programs/Msys2/mingw64/include/boost/smart\_ptr/detail/sp\_counted\_impl.hpp:35,
from D:/Programs/Msys2/mingw64/include/boost/smart\_ptr/detail/shared\_count.hpp:27,
from D:/Programs/Msys2/mingw64/include/boost/smart\_ptr/shared\_ptr.hpp:18,
from D:/Programs/Msys2/mingw64/include/boost/shared\_ptr.hpp:17,
from D:/Programs/Msys2/mingw64/include/boost/date\_time/time\_clock.hpp:17:
D:/Programs/Msys2/mingw64/include/c++/13.2.0/bits/stl\_uninitialized.h: In instantiation of 'constexpr bool std::\_\_check\_constructible() [with \_ValueType = Worker; \_Tp = Worker&&]':
D:/Programs/Msys2/mingw64/include/c++/13.2.0/bits/stl\_uninitialized.h:182:4: required from '\_ForwardIterator std::uninitialized\_copy(\_InputIterator, \_InputIterator, \_ForwardIterator) [with \_InputIterator = move\_iterator<Worker*>; \_ForwardIterator = Worker\*]'
D:/Programs/Msys2/mingw64

# Bing

I apologize for the confusion. The error is due to the fact that boost::mutex is non-copyable and non-movable, which means it cannot be copied or moved when the Worker objects are being added to the workers vector.

To resolve this issue, you can dynamically allocate the Worker objects on the heap and store pointers to them in the vector. Here's how you can modify your code:

#define BOOST\_BIND\_GLOBAL\_PLACEHOLDERS

#include <windows.h>

#include <boost/thread.hpp>

#include <boost/bind.hpp>

#include <iostream>

#include <vector>

#include <immintrin.h>

class Worker {

public:

Worker(int numNumbers) : numNumbers(numNumbers) {}

void operator()() {

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

unsigned long long randomNumber;

\_rdrand64\_step(&randomNumber);

randomNumber = randomNumber % 900000000000000 + 100000000000000;

boost::lock\_guard<boost::mutex> guard(mutex);

numbers.push\_back(randomNumber);

}

}

std::vector<unsigned long long> getNumbers() {

boost::lock\_guard<boost::mutex> guard(mutex);

return numbers;

}

private:

int numNumbers;

std::vector<unsigned long long> numbers;

boost::mutex mutex;

};

extern "C" \_\_declspec(dllexport) std::vector<unsigned long long> generateRandomNumbers(int numNumbers, int numWorkers, int numThreads) {

boost::thread\_group threads;

std::vector<Worker\*> workers;

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

Worker\* worker = new Worker(numNumbers);

workers.push\_back(worker);

threads.create\_thread(boost::ref(\*worker));

}

threads.join\_all();

std::vector<unsigned long long> allNumbers;

for(auto& worker : workers) {

std::vector<unsigned long long> numbers = worker->getNumbers();

allNumbers.insert(allNumbers.end(), numbers.begin(), numbers.end());

delete worker;

}

return allNumbers;

}

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul\_reason\_for\_call, LPVOID lpReserved) {

switch (ul\_reason\_for\_call) {

case DLL\_PROCESS\_ATTACH:

case DLL\_THREAD\_ATTACH:

case DLL\_THREAD\_DETACH:

case DLL\_PROCESS\_DETACH:

break;

}

return TRUE;

}

In this code, I've changed the workers vector to hold Worker\* instead of Worker. I've also added new Worker(numNumbers) to dynamically allocate the Worker objects on the heap, and delete worker to deallocate them after they're no longer needed.

This should resolve the compilation error. If you encounter any further issues, please let me know! I'm here to help.

# User

Yes, it was resolved and exe of test cpp also produced output in a desired way. Have you ensured that dll returns output as an array of numbers?

# Bing

Yes, the DLL function generateRandomNumbers is designed to return a std::vector<unsigned long long>, which is essentially an array of numbers. When you call this function from your test program, it generates a specified number of random numbers using the specified number of workers and threads, and then returns these numbers as a std::vector<unsigned long long>.

In your test program, you're correctly capturing this returned vector and printing out the numbers:

std::vector<unsigned long long> numbers = generateRandomNumbers(numNumbers, numWorkers, numThreads);

for (auto num : numbers) {

std::cout << num << std::endl;

}

So, you should see the generated random numbers printed out on the console when you run your test program. If you encounter any issues or have any further questions, please let me know! I'm here to help.

# User

Now onto the next step: linking the dll with xlwings. See I have already a py scipt file named xlwings\_trial which uses another dll for calling function like this:

import xlwings as xw
import rdrand
from randomgen import RDRAND
import sys
import numpy as np
import dask.dataframe as dd
from dask.distributed import Client
from scipy.optimize import newton
import pandas as pd
from datetime import datetime, timedelta
import calendar
from cffi import FFI
import numpy as np
import os
import csv
import time

print(sys.executable)
ffi = FFI()
#NUM\_NUMBERS = 100000
#NUM\_THREADS = 16

# Define the functions in the DLL

ffi.cdef("""
int rdrand64\_step(unsigned long long *rand);
void generate\_random\_numbers(int num\_threads, int num\_numbers);
unsigned long long* get\_numbers();
void free\_numbers(unsigned long long \*numbers);
""")

# Load the DLL

C = ffi.dlopen('D:\OneDrive - 0yt2k\Compiled dlls & executables\rdrand\_multithreaded\_new\_ucrt\_gcc.dll')

@xw.func
def generate\_and\_get\_data(NUM\_THREADS, NUM\_NUMBERS):
NUM\_THREADS = int(NUM\_THREADS)
NUM\_NUMBERS = int(NUM\_NUMBERS)
C.generate\_random\_numbers(NUM\_THREADS, NUM\_NUMBERS)
numbers\_ptr = C.get\_numbers()
numbers = [[int(numbers\_ptr[i])] for i in range(NUM\_NUMBERS)]
C.free\_numbers(numbers\_ptr)
return numbers

I want to add another UDF in that xlwings file calling random numbers from this boost dll. As you know, xlwings output is list of list. So can I include that boost dll link to this already existing code? Also can you give me similar code for defining UDF in xlwings like above - It should be named intel\_rdrand\_boost.

# Bing

Sure, you can add another user-defined function (UDF) in your xlwings file to call the generateRandomNumbers function from the boost DLL. Here's how you can do it:

# Define the functions in the boost DLL

ffi.cdef("""

std::vector<unsigned long long> generateRandomNumbers(int, int, int);

""")

# Load the boost DLL

C\_boost = ffi.dlopen('D:\\path\\to\\your\\boost\_rdrand.dll') # replace with the actual path to your DLL

@xw.func

def intel\_rdrand\_boost(NUM\_THREADS, NUM\_NUMBERS):

NUM\_THREADS = int(NUM\_THREADS)

NUM\_NUMBERS = int(NUM\_NUMBERS)

numbers\_ptr = C\_boost.generateRandomNumbers(NUM\_THREADS, NUM\_NUMBERS, NUM\_THREADS)

numbers = [[int(numbers\_ptr[i])] for i in range(NUM\_NUMBERS)]

return numbers

In this code, I've added a new UDF named intel\_rdrand\_boost that calls the generateRandomNumbers function from the boost DLL. This function takes the number of threads and the number of numbers as parameters, generates the random numbers, and returns them as a list of lists.

Please note that you need to replace 'D:\\path\\to\\your\\boost\_rdrand.dll' with the actual path to your DLL. Also, please ensure that the function signature in the ffi.cdef call matches the actual function signature in the DLL.

If you encounter any issues or have any further questions, please let me know! I'm here to help.

# User

I find following issues in the code:

First issue: See the first ffi definition:

ffi.cdef("""
int rdrand64\_step(unsigned long long *rand);
void generate\_random\_numbers(int num\_threads, int num\_numbers);
unsigned long long* get\_numbers();
void free\_numbers(unsigned long long \*numbers);
""")

In the second one, you have not included all the parameters like the first one.

Second issue:

See your function definition for second one:

def intel\_rdrand\_boost(NUM\_THREADS, NUM\_NUMBERS):
NUM\_THREADS = int(NUM\_THREADS)
NUM\_NUMBERS = int(NUM\_NUMBERS)
numbers\_ptr = C\_boost.generateRandomNumbers(NUM\_THREADS, NUM\_NUMBERS, NUM\_THREADS)
numbers = [[int(numbers\_ptr[i])] for i in range(NUM\_NUMBERS)]
return numbers

You need to supply 3 input arguments in order as number of random numbers, number of workers and finally number of threads. But you have supplied 2 arguments and that too not in proper order. My testcpp.exe testing this dll was success with ./testcpp 1000 10 4 (where 1000 numbers, 10 workers and 4 threads) So xlwings also should follow same order. Isn't it?

# Bing

I apologize for the confusion. You're correct. The generateRandomNumbers function in the DLL requires three parameters: the number of random numbers to generate, the number of workers, and the number of threads. Also, the order of parameters should be the same as in the DLL function.

Here's how you can modify the ffi definition and the intel\_rdrand\_boost function:

# Define the functions in the boost DLL

ffi.cdef("""

std::vector<unsigned long long> generateRandomNumbers(int numNumbers, int numWorkers, int numThreads);

""")

# Load the boost DLL

C\_boost = ffi.dlopen('D:\\path\\to\\your\\boost\_rdrand.dll') # replace with the actual path to your DLL

@xw.func

def intel\_rdrand\_boost(NUM\_NUMBERS, NUM\_WORKERS, NUM\_THREADS):

NUM\_NUMBERS = int(NUM\_NUMBERS)

NUM\_WORKERS = int(NUM\_WORKERS)

NUM\_THREADS = int(NUM\_THREADS)

numbers\_ptr = C\_boost.generateRandomNumbers(NUM\_NUMBERS, NUM\_WORKERS, NUM\_THREADS)

numbers = [[int(numbers\_ptr[i])] for i in range(NUM\_NUMBERS)]

return numbers

In this code, I've added the numNumbers parameter to the generateRandomNumbers function in the ffi definition, and I've added the NUM\_WORKERS parameter to the intel\_rdrand\_boost function. Now, when you call the intel\_rdrand\_boost function, you should provide the number of random numbers, the number of workers, and the number of threads as parameters, in that order.

Please replace the ffi definition and the intel\_rdrand\_boost function in your Python script with these modified versions. If you encounter any further issues, please let me know! I'm here to help.

# User

What about my first issue:

First issue: See the first ffi definition:

ffi.cdef("""
int rdrand64\_step(unsigned long long *rand);
void generate\_random\_numbers(int num\_threads, int num\_numbers);
unsigned long long* get\_numbers();
void free\_numbers(unsigned long long \*numbers);
""")

In the second one, you have not included all the parameters like the first one:

ffi.cdef("""
std::vector<unsigned long long> generateRandomNumbers(int numNumbers, int numWorkers, int numThreads);
""")

# Bing

I apologize for the confusion. The ffi.cdef function is used to provide the C definitions that are needed to call the functions in the DLL. Each ffi.cdef call should contain the definitions for the functions in one DLL.

In your case, you have two DLLs: one that contains the rdrand64\_step, generate\_random\_numbers, get\_numbers, and free\_numbers functions, and another one that contains the generateRandomNumbers function.

So, you should have two separate ffi.cdef calls, one for each DLL:

# Define the functions in the first DLL

ffi.cdef("""

int rdrand64\_step(unsigned long long \*rand);

void generate\_random\_numbers(int num\_threads, int num\_numbers);

unsigned long long\* get\_numbers();

void free\_numbers(unsigned long long \*numbers);

""")

# Load the first DLL

C = ffi.dlopen('D:\\path\\to\\your\\first.dll') # replace with the actual path to your first DLL

# Define the functions in the boost DLL

ffi.cdef("""

std::vector<unsigned long long> generateRandomNumbers(int numNumbers, int numWorkers, int numThreads);

""")

# Load the boost DLL

C\_boost = ffi.dlopen('D:\\path\\to\\your\\boost\_rdrand.dll') # replace with the actual path to your boost DLL

In this code, I've added a separate ffi.cdef call for the boost DLL that contains the generateRandomNumbers function. This function takes three parameters: the number of random numbers to generate, the number of workers, and the number of threads.

Please replace the ffi.cdef calls and the DLL paths in your Python script with these modified versions. If you encounter any further issues, please let me know! I'm here to help.

# User

But first ffi.def has many parameters:

ffi.cdef("""
int rdrand64\_step(unsigned long long *rand);
void generate\_random\_numbers(int num\_threads, int num\_numbers);
unsigned long long* get\_numbers();
void free\_numbers(unsigned long long \*numbers);
""") - there are 4 parameters in the brackets while second one has only one:

# Define the functions in the boost DLL

ffi.cdef("""
std::vector<unsigned long long> generateRandomNumbers(int numNumbers, int numWorkers, int numThreads);
""")

# Bing

I apologize for the confusion. The ffi.cdef function is used to provide the C definitions that are needed to call the functions in the DLL. Each ffi.cdef call should contain the definitions for the functions in one DLL.

In your case, you have two DLLs: one that contains the rdrand64\_step, generate\_random\_numbers, get\_numbers, and free\_numbers functions, and another one that contains the generateRandomNumbers function.

So, you should have two separate ffi.cdef calls, one for each DLL:

# Define the functions in the first DLL

ffi.cdef("""

int rdrand64\_step(unsigned long long \*rand);

void generate\_random\_numbers(int num\_threads, int num\_numbers);

unsigned long long\* get\_numbers();

void free\_numbers(unsigned long long \*numbers);

""")

# Load the first DLL

C = ffi.dlopen('D:\\path\\to\\your\\first.dll') # replace with the actual path to your first DLL

# Define the functions in the boost DLL

ffi.cdef("""

std::vector<unsigned long long> generateRandomNumbers(int numNumbers, int numWorkers, int numThreads);

""")

# Load the boost DLL

C\_boost = ffi.dlopen('D:\\path\\to\\your\\boost\_rdrand.dll') # replace with the actual path to your boost DLL

In this code, I've added a separate ffi.cdef call for the boost DLL that contains the generateRandomNumbers function. This function takes three parameters: the number of random numbers to generate, the number of workers, and the number of threads.

Please replace the ffi.cdef calls and the DLL paths in your Python script with these modified versions. If you encounter any further issues, please let me know! I'm here to help.

# User

Since you have defined:

# Define the functions in the boost DLL

ffi.cdef("""
std::vector<unsigned long long> generateRandomNumbers(int numNumbers, int numWorkers, int numThreads);
""")

Change your xlwings code of UDF for second UDF.

# Bing

I apologize for the confusion earlier. The cffi library, which is used in your Python script to interface with the DLL, doesn't support C++ directly and therefore doesn't understand std::vector. It only supports C.

However, you can still use the DLL by creating a C-compatible interface. This would involve modifying the DLL code to add two new functions: one to call generateRandomNumbers and store the result in a global variable, and another to retrieve the numbers from the global variable.

Here's an example of how you can modify the DLL code:

std::vector<unsigned long long> g\_numbers;

extern "C" \_\_declspec(dllexport) void generateRandomNumbersC(int numNumbers, int numWorkers, int numThreads) {

g\_numbers = generateRandomNumbers(numNumbers, numWorkers, numThreads);

}

extern "C" \_\_declspec(dllexport) unsigned long long\* getNumbersC() {

return g\_numbers.data();

}

extern "C" \_\_declspec(dllexport) int getNumbersSizeC() {

return g\_numbers.size();

}

Then, you can modify the Python script to call these new functions:

# Define the functions in the boost DLL

ffi.cdef("""

void generateRandomNumbersC(int numNumbers, int numWorkers, int numThreads);

unsigned long long\* getNumbersC();

int getNumbersSizeC();

""")

# Load the boost DLL

C\_boost = ffi.dlopen('D:\\path\\to\\your\\boost\_rdrand.dll') # replace with the actual path to your DLL

@xw.func

def intel\_rdrand\_boost(NUM\_NUMBERS, NUM\_WORKERS, NUM\_THREADS):

NUM\_NUMBERS = int(NUM\_NUMBERS)

NUM\_WORKERS = int(NUM\_WORKERS)

NUM\_THREADS = int(NUM\_THREADS)

C\_boost.generateRandomNumbersC(NUM\_NUMBERS, NUM\_WORKERS, NUM\_THREADS)

numbers\_ptr = C\_boost.getNumbersC()

numbers\_size = C\_boost.getNumbersSizeC()

numbers = [[int(numbers\_ptr[i])] for i in range(numbers\_size)]

return numbers

In this code, I've added the generateRandomNumbersC, getNumbersC, and getNumbersSizeC functions to the ffi definition, and I've modified the intel\_rdrand\_boost function to call these new functions.

Please replace the ffi definition and the intel\_rdrand\_boost function in your Python script with these modified versions. If you encounter any further issues, please let me know! I'm here to help.

# User

My current extern "C" in dll is like this:

extern "C" \_\_declspec(dllexport) std::vector<unsigned long long> generateRandomNumbers(int numNumbers, int numWorkers, int numThreads) {
boost::thread\_group threads;
std::vector<Worker\*> workers;

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

Worker\* worker = new Worker(numNumbers);

workers.push\_back(worker);

threads.create\_thread(boost::ref(\*worker));

}

threads.join\_all();

std::vector<unsigned long long> allNumbers;

for(auto& worker : workers) {

std::vector<unsigned long long> numbers = worker->getNumbers();

allNumbers.insert(allNumbers.end(), numbers.begin(), numbers.end());

delete worker;

}

return allNumbers;

}

Can you change?

# Bing

I apologize for the confusion earlier. The cffi library, which is used in your Python script to interface with the DLL, doesn't support C++ directly and therefore doesn't understand std::vector. It only supports C.

However, you can still use the DLL by creating a C-compatible interface. This would involve modifying the DLL code to add two new functions: one to call generateRandomNumbers and store the result in a global variable, and another to retrieve the numbers from the global variable.

Here's an example of how you can modify the DLL code:

std::vector<unsigned long long> g\_numbers;

extern "C" \_\_declspec(dllexport) void generateRandomNumbersC(int numNumbers, int numWorkers, int numThreads) {

g\_numbers = generateRandomNumbers(numNumbers, numWorkers, numThreads);

}

extern "C" \_\_declspec(dllexport) unsigned long long\* getNumbersC() {

return g\_numbers.data();

}

extern "C" \_\_declspec(dllexport) int getNumbersSizeC() {

return g\_numbers.size();

}

Then, you can modify the Python script to call these new functions:

# Define the functions in the boost DLL

ffi.cdef("""

void generateRandomNumbersC(int numNumbers, int numWorkers, int numThreads);

unsigned long long\* getNumbersC();

int getNumbersSizeC();

""")

# Load the boost DLL

C\_boost = ffi.dlopen('D:\\path\\to\\your\\boost\_rdrand.dll') # replace with the actual path to your DLL

@xw.func

def intel\_rdrand\_boost(NUM\_NUMBERS, NUM\_WORKERS, NUM\_THREADS):

NUM\_NUMBERS = int(NUM\_NUMBERS)

NUM\_WORKERS = int(NUM\_WORKERS)

NUM\_THREADS = int(NUM\_THREADS)

C\_boost.generateRandomNumbersC(NUM\_NUMBERS, NUM\_WORKERS, NUM\_THREADS)

numbers\_ptr = C\_boost.getNumbersC()

numbers\_size = C\_boost.getNumbersSizeC()

numbers = [[int(numbers\_ptr[i])] for i in range(numbers\_size)]

return numbers

In this code, I've added the generateRandomNumbersC, getNumbersC, and getNumbersSizeC functions to the ffi definition, and I've modified the intel\_rdrand\_boost function to call these new functions.

Please replace the ffi definition and the intel\_rdrand\_boost function in your Python script with these modified versions. If you encounter any further issues, please let me know! I'm here to help.

# User

But you have not responded properly yo my last prompt. Do I need to change my current structure:

extern "C" \_\_declspec(dllexport) std::vector<unsigned long long> generateRandomNumbers(int numNumbers, int numWorkers, int numThreads) {
boost::thread\_group threads;
std::vector<Worker\*> workers;

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

Worker\* worker = new Worker(numNumbers);

workers.push\_back(worker);

threads.create\_thread(boost::ref(\*worker));

}

threads.join\_all();

std::vector<unsigned long long> allNumbers;

for(auto& worker : workers) {

std::vector<unsigned long long> numbers = worker->getNumbers();

allNumbers.insert(allNumbers.end(), numbers.begin(), numbers.end());

delete worker;

}

return allNumbers;

}

# Bing

I apologize for the confusion. Yes, you need to modify your DLL code to make it compatible with the Python script. The cffi library in Python doesn't support C++ directly, so it doesn't understand std::vector. However, it can work with C-style arrays.

Here's how you can modify your DLL code:

std::vector<unsigned long long> g\_numbers;

extern "C" \_\_declspec(dllexport) void generateRandomNumbersC(int numNumbers, int numWorkers, int numThreads) {

boost::thread\_group threads;

std::vector<Worker\*> workers;

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

Worker\* worker = new Worker(numNumbers);

workers.push\_back(worker);

threads.create\_thread(boost::ref(\*worker));

}

threads.join\_all();

for(auto& worker : workers) {

std::vector<unsigned long long> numbers = worker->getNumbers();

g\_numbers.insert(g\_numbers.end(), numbers.begin(), numbers.end());

delete worker;

}

}

extern "C" \_\_declspec(dllexport) unsigned long long\* getNumbersC() {

return g\_numbers.data();

}

extern "C" \_\_declspec(dllexport) int getNumbersSizeC() {

return g\_numbers.size();

}

In this code, I've added a global variable g\_numbers to store the generated random numbers. The generateRandomNumbersC function now populates g\_numbers instead of returning a value. I've also added two new functions getNumbersC and getNumbersSizeC to retrieve the generated numbers and their count, respectively.

Please replace the generateRandomNumbers function in your DLL source code with these modified versions. If you encounter any further issues, please let me know! I'm here to help.

# User

Now my dll code has changed like this:

#define BOOST\_BIND\_GLOBAL\_PLACEHOLDERS
#include <windows.h>
#include <boost/thread.hpp>
#include <boost/bind.hpp>
#include <iostream>
#include <vector>
#include <immintrin.h>

class Worker {
public:
Worker(int numNumbers) : numNumbers(numNumbers) {}

void operator()() {

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

unsigned long long randomNumber;

\_rdrand64\_step(&randomNumber);

randomNumber = randomNumber % 900000000000000 + 100000000000000;

boost::lock\_guard<boost::mutex> guard(mutex);

numbers.push\_back(randomNumber);

}

}

std::vector<unsigned long long> getNumbers() {

boost::lock\_guard<boost::mutex> guard(mutex);

return numbers;

}

private:
int numNumbers;
std::vector<unsigned long long> numbers;
boost::mutex mutex;
};

std::vector<unsigned long long> g\_numbers;

extern "C" \_\_declspec(dllexport) void generateRandomNumbersC(int numNumbers, int numWorkers, int numThreads) {
boost::thread\_group threads;
std::vector<Worker\*> workers;

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

Worker\* worker = new Worker(numNumbers);

workers.push\_back(worker);

threads.create\_thread(boost::ref(\*worker));

}

threads.join\_all();

for(auto& worker : workers) {

std::vector<unsigned long long> numbers = worker->getNumbers();

g\_numbers.insert(g\_numbers.end(), numbers.begin(), numbers.end());

delete worker;

}

}

extern "C" \_\_declspec(dllexport) unsigned long long\* getNumbersC() {
return g\_numbers.data();
}

extern "C" \_\_declspec(dllexport) int getNumbersSizeC() {
return g\_numbers.size();
}

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul\_reason\_for\_call, LPVOID lpReserved) {
switch (ul\_reason\_for\_call) {
case DLL\_PROCESS\_ATTACH:
case DLL\_THREAD\_ATTACH:
case DLL\_THREAD\_DETACH:
case DLL\_PROCESS\_DETACH:
break;
}
return TRUE;
}

But my testcpp (see earlier) now says:

$ ./testcpp 55 5 3
Could not locate the function.

# Bing

I apologize for the confusion. The issue is that your test program is still trying to call the generateRandomNumbers function, which no longer exists in the DLL. Instead, you should call the new generateRandomNumbersC, getNumbersC, and getNumbersSizeC functions.

Here's how you can modify your test program:

#include <windows.h>

#include <iostream>

#include <vector>

typedef void (\*GenerateRandomNumbersFuncC)(int, int, int);

typedef unsigned long long\* (\*GetNumbersFuncC)();

typedef int (\*GetNumbersSizeFuncC)();

int main(int argc, char\* argv[]) {

HINSTANCE hGetProcIDDLL = LoadLibrary("boost\_rdrand.dll");

if (!hGetProcIDDLL) {

std::cout << "Could not load the dynamic library." << std::endl;

return EXIT\_FAILURE;

}

GenerateRandomNumbersFuncC generateRandomNumbersC = (GenerateRandomNumbersFuncC) GetProcAddress(hGetProcIDDLL, "generateRandomNumbersC");

GetNumbersFuncC getNumbersC = (GetNumbersFuncC) GetProcAddress(hGetProcIDDLL, "getNumbersC");

GetNumbersSizeFuncC getNumbersSizeC = (GetNumbersSizeFuncC) GetProcAddress(hGetProcIDDLL, "getNumbersSizeC");

if (!generateRandomNumbersC || !getNumbersC || !getNumbersSizeC) {

std::cout << "Could not locate the function." << std::endl;

return EXIT\_FAILURE;

}

int numNumbers = std::stoi(argv[1]);

int numWorkers = std::stoi(argv[2]);

int numThreads = std::stoi(argv[3]);

generateRandomNumbersC(numNumbers, numWorkers, numThreads);

unsigned long long\* numbers = getNumbersC();

int numbersSize = getNumbersSizeC();

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

std::cout << numbers[i] << std::endl;

}

FreeLibrary(hGetProcIDDLL);

return EXIT\_SUCCESS;

}

In this code, I've added new function pointers for generateRandomNumbersC, getNumbersC, and getNumbersSizeC, and I've modified the main function to call these new functions.

Please replace the code in your test program with this modified version. If you encounter any further issues, please let me know! I'm here to help.

# User

testcpp ran okay, But xlwings still cannot load library (boost dll):

pythoncom error: Python error invoking COM method.

Traceback (most recent call last):
File "D:\Programs\Python\Lib\site-packages\win32com\server\policy.py", line 305, in *Invoke*
return self.*invoke*(dispid, lcid, wFlags, args)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "D:\Programs\Python\Lib\site-packages\win32com\server\policy.py", line 310, in *invoke*
return S\_OK, -1, self.*invokeex*(dispid, lcid, wFlags, args, None, None)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "D:\Programs\Python\Lib\site-packages\win32com\server\policy.py", line 639, in *invokeex*
return func(\*args)
^^^^^^^^^^^
File "D:\Programs\Python\Lib\site-packages\xlwings\com\_server.py", line 231, in Call
return ToVariant(getattr(obj, method)(\*pargs, \*\*kwargs))
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "D:\Programs\Python\Lib\site-packages\xlwings\udfs.py", line 703, in import\_udfs
module = get\_udf\_module(module\_name, xl\_workbook)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "D:\Programs\Python\Lib\site-packages\xlwings\udfs.py", line 381, in get\_udf\_module
module = import\_module(module\_name)
^^^^^^^^^^^^^^^^^^^^^^^^^^
File "D:\Programs\Python\Lib\importlib\_*init*\_.py", line 90, in import\_module
return \_bootstrap.\_gcd\_import(name[level:], package, level)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "<frozen importlib.\_bootstrap>", line 1387, in \_gcd\_import
File "<frozen importlib.\_bootstrap>", line 1360, in \_find\_and\_load
File "<frozen importlib.\_bootstrap>", line 1331, in \_find\_and\_load\_unlocked
File "<frozen importlib.\_bootstrap>", line 935, in \_load\_unlocked
File "<frozen importlib.\_bootstrap\_external>", line 995, in exec\_module
File "<frozen importlib.\_bootstrap>", line 488, in \_call\_with\_frames\_removed
File "D:\OneDrive - 0yt2k\excel examples including add-in xlwings and connected python - vba - dlls, etc\xlwings\_trial.py", line 41, in <module>
C\_boost = ffi.dlopen('D:\OneDrive - 0yt2k\Compiled dlls & executables\boost\_rdrand.dll')
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "D:\Programs\Python\Lib\site-packages\cffi\api.py", line 150, in dlopen
lib, function\_cache = \_make\_ffi\_library(self, name, flags)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "D:\Programs\Python\Lib\site-packages\cffi\api.py", line 832, in \_make\_ffi\_library
backendlib = \_load\_backend\_lib(backend, libname, flags)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "D:\Programs\Python\Lib\site-packages\cffi\api.py", line 828, in \_load\_backend\_lib
return backend.load\_library(path, flags)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
OSError: cannot load library 'D:\OneDrive - 0yt2k\Compiled dlls & executables\boost\_rdrand.dll': error 0x7e

# Bing

I'm sorry to hear that you're having trouble loading the DLL with xlwings. The error message OSError: cannot load library typically indicates that Python is unable to find or load the specified DLL.

Here are a few things you could check:

1. **File Path**: Make sure that the file path to the DLL is correct. The path should be absolute and should include the full filename of the DLL. In your case, the path is 'D:\\OneDrive - 0yt2k\\Compiled dlls & executables\\boost\_rdrand.dll'. Please double-check that this path is correct and that the DLL file is located at this path.
2. **DLL Dependencies**: The DLL might depend on other DLLs that are not found. You can use a tool like [Dependency Walker](http://www.dependencywalker.com/) to check the dependencies of the DLL. If there are any missing dependencies, you'll need to make sure that they are available on your system and that they can be found by the DLL.
3. **Architecture Compatibility**: Ensure that the architecture of the DLL (32-bit or 64-bit) matches the architecture of your Python interpreter. You can check the architecture of your Python interpreter by running import platform; print(platform.architecture()) in Python.
4. **C++ Redistributable**: If the DLL was built with Visual Studio, it might require the Visual C++ Redistributable Packages to be installed on your system. You can download these packages from the [Microsoft website](https://support.microsoft.com/en-us/help/2977003/the-latest-supported-visual-c-downloads).

If none of the above solutions work, please let me know. I'm here to help!