# User

I want to use rdrand64\_step intel intrinsic for generating random numbers in rust. I have following rust code but results in compilation error:

use std::sync::{Arc, Mutex};
use std::thread;
use std::io;
use rdrand::RdRand;

fn generate\_random\_numbers(num\_threads: usize, num\_rand\_nums: usize) -> Vec<u64> {
let result = Arc::new(Mutex::new(Vec::new()));
let handles: Vec<*> = (0..num\_threads)
.map(|*| {
let result = Arc::clone(&result);
thread::spawn(move || {
let mut local\_result = Vec::new();
let rng = RdRand::new().unwrap();
for \_ in 0..num\_rand\_nums / num\_threads {
let rand\_num: u64 = match rng.try\_next\_u64() {
Some(num) => num,
None => continue,
};
// Ensure the random number is 15 digits long and within the specified range
let rand\_num = 100\_000\_000\_000\_000 + (rand\_num % 900\_000\_000\_000\_000);
local\_result.push(rand\_num);
}
result.lock().unwrap().extend(local\_result);
})
})
.collect();

for handle in handles {

handle.join().unwrap();

}

Arc::try\_unwrap(result).unwrap().into\_inner().unwrap()

}

fn main() {
println!("Enter the number of threads: ");
let mut num\_threads = String::new();
io::stdin().read\_line(&mut num\_threads).expect("Failed to read line");
let num\_threads: usize = num\_threads.trim().parse().unwrap();

println!("Enter the number of random numbers to generate: ");

let mut num\_rand\_nums = String::new();

io::stdin().read\_line(&mut num\_rand\_nums).expect("Failed to read line");

let num\_rand\_nums: usize = num\_rand\_nums.trim().parse().unwrap();

let random\_numbers = generate\_random\_numbers(num\_threads, num\_rand\_nums);

println!("{:?}", random\_numbers);

}

Error:

Compiling rust\_rand\_codeium v0.1.0 (D:\Downloads\rust\_dll\_codeium)
error[E0308]: mismatched types
--> src/rust\_rand\_codeium.rs:16:25
|
15 | let rand\_num: u64 = match rng.try\_next\_u64() {
| ------------------ this expression has type Result<u64, ErrorCode>
16 | Some(num) => num,
| ^^^^^^^^^ expected Result<u64, ErrorCode>, found Option<\_>
|
= note: expected enum Result<u64, ErrorCode>
found enum Option<\_>

error[E0308]: mismatched types
--> src/rust\_rand\_codeium.rs:17:25
|
15 | let rand\_num: u64 = match rng.try\_next\_u64() {
| ------------------ this expression has type Result<u64, ErrorCode>
16 | Some(num) => num,
17 | None => continue,
| ^^^^ expected Result<u64, ErrorCode>, found Option<\_>
|
= note: expected enum Result<u64, ErrorCode>
found enum Option<\_>

For more information about this error, try rustc --explain E0308.
error: could not compile rust\_rand\_codeium (bin "rust\_test\_codeium") due to 2 previous errors

# Bing

The error is due to the mismatched types in your match statement. The try\_next\_u64() function from the rdrand crate returns a Result<u64, ErrorCode>, not an Option<u64>.

You can modify your code as follows:

let rand\_num: u64 = match rng.try\_next\_u64() {

Ok(num) => num,

Err(\_) => continue,

};

In this case, Ok(num) corresponds to a successful generation of a random number, and Err(\_) corresponds to an error during the generation. The continue statement will skip the current iteration of the loop if an error occurs. This should resolve your compilation error. Please try this and let me know if you have any other questions!

# User

Can you give me full code?

# Bing

Sure, here is the corrected code:

use std::sync::{Arc, Mutex};

use std::thread;

use std::io;

use rdrand::RdRand;

fn generate\_random\_numbers(num\_threads: usize, num\_rand\_nums: usize) -> Vec<u64> {

let result = Arc::new(Mutex::new(Vec::new()));

let handles: Vec<\_> = (0..num\_threads)

.map(|\_| {

let result = Arc::clone(&result);

thread::spawn(move || {

let mut local\_result = Vec::new();

let rng = RdRand::new().unwrap();

for \_ in 0..num\_rand\_nums / num\_threads {

let rand\_num: u64 = match rng.try\_next\_u64() {

Ok(num) => num,

Err(\_) => continue,

};

// Ensure the random number is 15 digits long and within the specified range

let rand\_num = 100\_000\_000\_000\_000 + (rand\_num % 900\_000\_000\_000\_000);

local\_result.push(rand\_num);

}

result.lock().unwrap().extend(local\_result);

})

})

.collect();

for handle in handles {

handle.join().unwrap();

}

Arc::try\_unwrap(result).unwrap().into\_inner().unwrap()

}

fn main() {

println!("Enter the number of threads: ");

let mut num\_threads = String::new();

io::stdin().read\_line(&mut num\_threads).expect("Failed to read line");

let num\_threads: usize = num\_threads.trim().parse().unwrap();

println!("Enter the number of random numbers to generate: ");

let mut num\_rand\_nums = String::new();

io::stdin().read\_line(&mut num\_rand\_nums).expect("Failed to read line");

let num\_rand\_nums: usize = num\_rand\_nums.trim().parse().unwrap();

let random\_numbers = generate\_random\_numbers(num\_threads, num\_rand\_nums);

println!("{:?}", random\_numbers);

}

This code should compile and run without errors. It generates the specified number of random numbers using the specified number of threads. Each random number is 15 digits long and within the range of 100\_000\_000\_000\_000 to 999\_999\_999\_999\_999. If there's an error generating a random number, it skips that iteration and continues with the next one. Let me know if you have any other questions!

# User

have got another rust code that compiles into dll library as below. That one does not use rdrand crate but uses rayon for parallel processing. Can you tell me which of these codes (compare between code below and code given above by you) are faster and more efficient?

#![feature(core\_intrinsics)] // Enable core intrinsics for \_rdrand64\_step

use std::alloc::{alloc, dealloc, Layout};
use core::arch::x86\_64::\_rdrand64\_step;
use rayon::prelude::\*; // Import the rayon crate for parallel processing

extern crate rand; // Import the rand crate
use rand::Rng;

#[no\_mangle]
pub extern "C" fn rdrand64\_step(rand: &mut u64) -> i32 {
unsafe {
if \_rdrand64\_step(rand) == 1 {
// Check if the number has exactly 15 digits
if \*rand >= 100\_000\_000\_000\_000 && \*rand <= 999\_999\_999\_999\_999 {
return 1; // Success
}
}
// Fallback to software RNG if hardware RNG fails or does not produce a 15-digit number
\*rand = rand::thread\_rng().gen\_range(100\_000\_000\_000\_000..=999\_999\_999\_999\_999);
return 0; // Indicate fallback
}
}

#[no\_mangle]
pub extern "C" fn generate\_random\_numbers(num\_threads: i32, num\_numbers: i32) {
let numbers\_per\_thread = num\_numbers / num\_threads;
(0..num\_threads).into\_par\_iter().for\_each(|thread\_num| {
for i in 0..numbers\_per\_thread {
let mut rand: u64 = 0;
rdrand64\_step(&mut rand);
unsafe {
\*NUMBERS.offset((thread\_num \* numbers\_per\_thread + i) as isize) = rand;
}
}
});
}

static mut NUMBERS: \*mut u64 = std::ptr::null\_mut();

#[no\_mangle]
pub extern "C" fn get\_numbers() -> \*mut u64 {
unsafe { NUMBERS }
}

#[no\_mangle]
pub extern "C" fn allocate\_memory(num\_numbers: i32) {
unsafe {
let layout = Layout::from\_size\_align(num\_numbers as usize \* std::mem::size\_of::<u64>(), std::mem::align\_of::<u64>()).unwrap();
NUMBERS = alloc(layout) as \*mut u64;
}
}

#[no\_mangle]
pub extern "C" fn free\_memory() {
unsafe {
let layout = Layout::from\_size\_align(std::mem::size\_of\_val(&\*NUMBERS), std::mem::align\_of\_val(&\*NUMBERS)).unwrap();
dealloc(NUMBERS as \*mut u8, layout);
}
}

# Bing

The efficiency of the two codes can depend on several factors. Here are some points to consider:

1. **Hardware vs Software RNG**: The first code uses the rdrand crate which provides a software interface to the hardware RNG, while the second code uses the \_rdrand64\_step intrinsic directly. If the hardware RNG is available and functioning correctly, using it directly could be faster. However, if the hardware RNG fails or does not produce a 15-digit number, the second code falls back to a software RNG, which could be slower.
2. **Parallel Processing**: Both codes use parallel processing, but they do it differently. The first code uses the standard library's threading model, while the second code uses the rayon crate. rayon is designed to be more efficient for data-parallel computations, as it uses a work-stealing algorithm to dynamically balance load across threads. This could make the second code faster if the workload is unevenly distributed.
3. **Memory Management**: The second code uses unsafe Rust to manually manage memory, which could be faster but also more error-prone. The first code uses Rust's standard library data structures, which are safer but could have some overhead.

In conclusion, the second code could potentially be faster due to direct use of hardware RNG and the rayon crate for parallel processing, but it also uses unsafe Rust which could lead to bugs if not used carefully. The first code might be slower, but it is safer as it avoids unsafe Rust and manual memory management.

However, the actual performance can vary depending on the specific hardware and workload. To get a definitive answer, you could benchmark the two codes under the same conditions. Please let me know if you have any other questions!