

Fault Tolerance and Parallel Computing

Loic Niragire

School of Technology, Northcentral University

TIM 8121: Distributed Algorithms and Parallel Computing

Dr. Carol Cusano

December 26th, 2022

This paper accompanies a C++ multicore program that ensures fault tolerance using the OpenMP API on a dual-core processor. The program seeks to invert a color image stored in RAM as a character array. The inverse is obtained by computing one's complement for each bit in the input array. To achieve fault tolerance, the input array is distributed across available cores where their corresponding inverses are computed. Then, the results from each core are compared to assert consistency. Finally, the process is repeated until all cores generate identical results. The source code is structured into three files. The header file, *Inverter.h*, contains all declarations implemented in *Inverter.cpp*, while the main function is in the *ImageInverter.cpp* file.

By definition, a fault-tolerant application must be able to recover from detected faults or errors so that failure from one component does not render the entire system futile. One technique for detecting errors, *dual modular redundancy* (DMR), applies two processing elements to the same input data. When the output from processing elements diverges, an error is detected (H. Mushtaq, 2011). Therefore, detecting failures by redundant computation. Other versions of this technique exist where more than two processing elements are used to detect software or hardware faults. The program presented in this paper implements an error handling function to recover from transient faults without locating the source of the error. This is a simplification done for illustration purposes whereby the source of the error has been narrowed to the function computing one's complement. Otherwise, the application must undergo a series of fault diagnosis steps to locate failed components. Fault tolerance can also be achieved by periodically saving computational states into stable storage. Then, when a failure occurs, the program would recover by restarting the operations from the latest saved

computational state. However, this strategy suffers from I/O transfer overhead and does not scale well (Fu & Ding, 2010).

Programming Environment

The program was developed using visual studio 2022 with support for *openmp:llvm* to enable the compiler to target LLVM's OpenMP runtime library on x86 architecture. The OpenMP Task construct specifies a unit of work that can be done by any thread and is an alternative to the *parallel for* directive (Hagger, 2022). The primary advantage of using Tasks to achieve computational redundancy is that they allow computational steps to be scheduled in arbitrary order.

Fault Tolerance Strategy

The program implements a redundant computation fault tolerance strategy using OpenMP. Each thread executes one's complement on the same input array, and the result is accepted when the output arrays are identical. To simulate this functionality, a voting function has been implemented to return a *true* or *false* response value indicating whether input arrays are identical. Thus, returning a *false* means that the input arrays are not identical, which implies that an error has been detected. Meanwhile, returning a *true* signifies that the inputs are identical; therefore, no failure has been detected.

Distributing input data across threads is the first step to achieving redundant computation. Thus, rather than partitioning the input array so that each thread processes T/n items – where T and n denote the total number of items and the number of nodes, respectively, the desired setting is for each thread to process the entire input data. The scenario where each thread only processes a slice of the input data is illustrated in Figure 1. Note that OpenMP handles the partitioning of the input data across threads. Meanwhile, Figure 2 shows an

equivalent processing but with computational redundancy where threads 1 and 2 each process the entire input data rather than a slice of the input array, as in Figure 1. This is accomplished by declaring an OpenMP parallel region to generate a team of threads where each thread executes the entire code block.

Figure 1 Parallel processing without redundancy

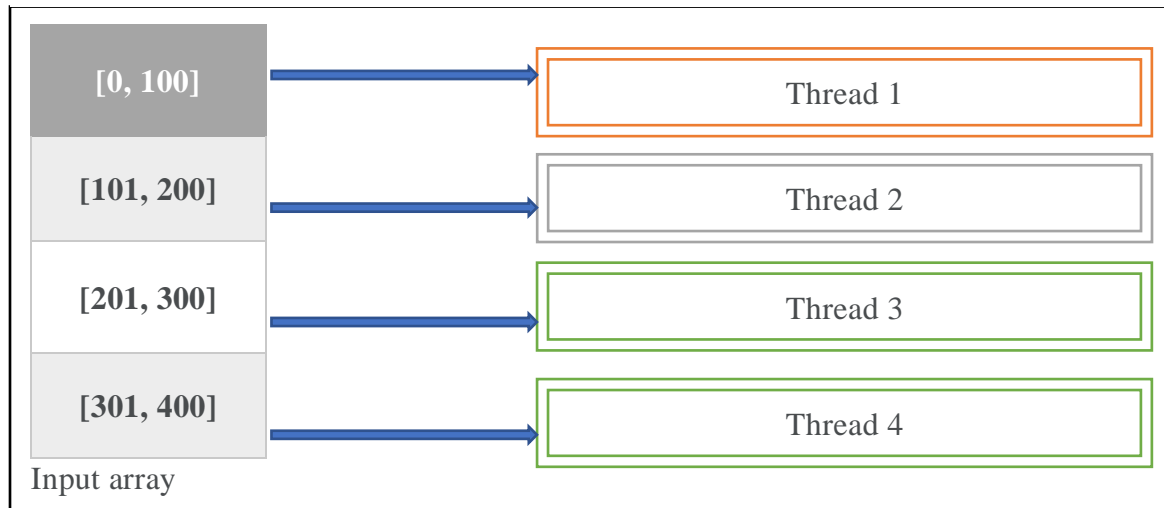


Figure 2 Parallel processing with computational redundancy on two threads

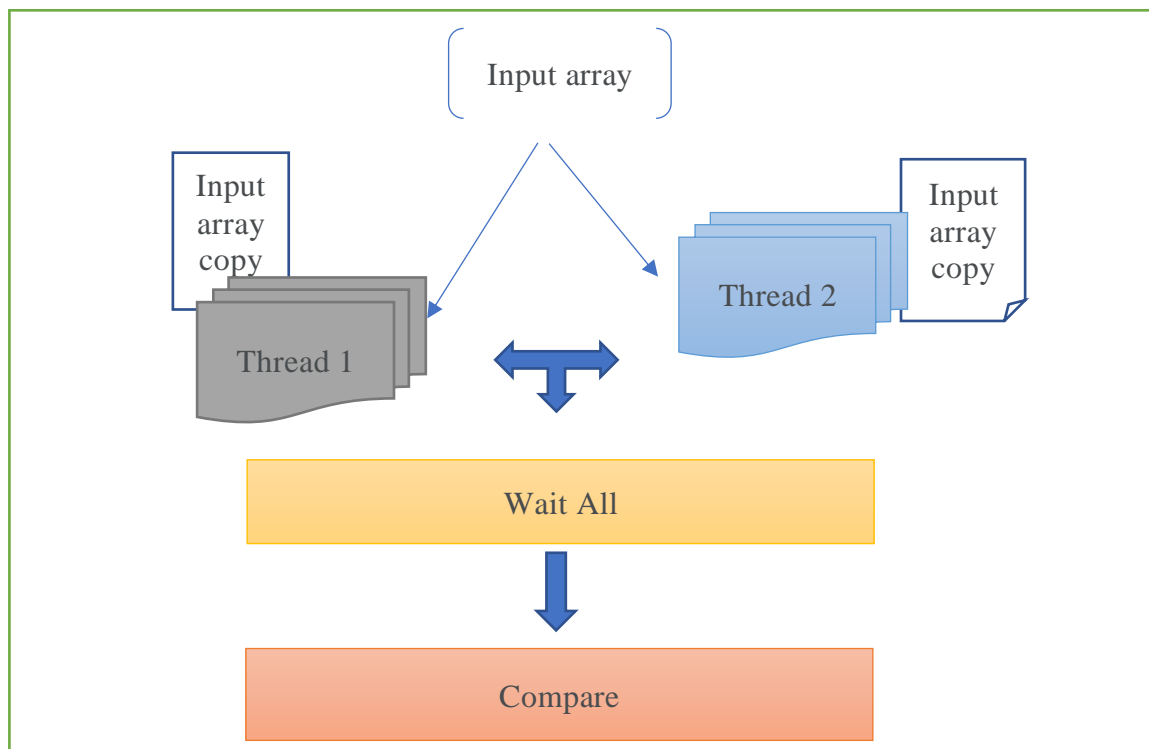


Figure 3 shows the implementation of the complement function used to invert an image passed in as an unsigned character array. Line 1 and 2 initializes a simple lock to synchronize access to shared resources to prevent data race issues. The number of threads the function utilizes is passed in as a parameter and gets set on line 5. Within the parallel region defined on line 6, each thread creates a task to invert a portion of the input array, and the tasks are synchronized before comparing the outputs from each thread. The synchronization on line 13 instructs the runtime to wait until all threads have completed the execution of the *for* loop on line 8. Therefore, lines 14 to 18 do not execute until all threads have executed lines 9 through 12.

Figure 3 Complement function implementation

```
void Complement(unsigned char image[], unsigned char output[], int
size, int threads) {

(1)    omp_lock_t lck;
(2)    omp_init_lock(&lck);
(3)    int ct = 0;
(4)    int upperLimit = threads * size - 1;

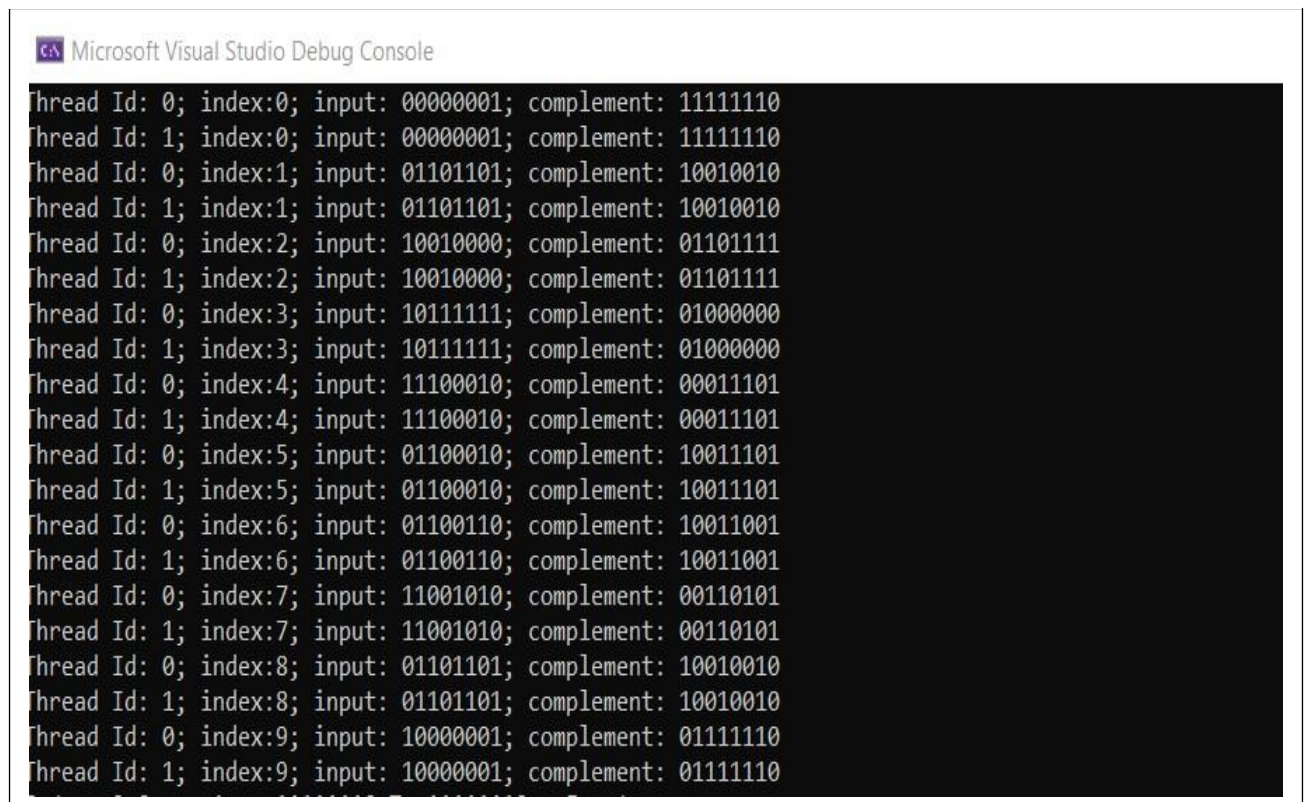
(5)    omp_set_num_threads(threads);
(6)    #pragma omp parallel firstprivate(image) shared(output, ct)
{
(7)        #pragma omp task
{
(8)            for (int i = 0; i < size; ++i)
{
(9)                omp_set_lock(&lck);
(10)               output[ct] = ~image[i];
(11)               ++ct;
(12)               omp_unset_lock(&lck);
}

(13)           #pragma omp taskwait
(14)           for (int x = 0; x < upperLimit; x += 2)
{
(15)               omp_set_lock(&lck);
(16)               if (output[x] != output[x + 1]) {
(17)                   Complement(image, output, size, threads);
}
(18)               omp_unset_lock(&lck);
}
}
}

(19)    omp_destroy_lock(&lck);
}
```

The *firstprivate* property defined on line 6 instructs the runtime to assign each thread a private image copy. Moreover, the image is expected to be initialized before the parallel construct (Microsoft, 2022). Meanwhile, the output array and its index counter are shared among threads. For instance, when executing the complement function using n number of threads, at each iteration of the *for* loop on line 8, the complement operation on line 10 is executed n times, as shown in Figure 4, where two threads process an input array of 10 elements. Therefore, comparing the complement results from each thread is sufficient to iterate through the output array while comparing n consecutive entries. Figure 3, line 16 is a case where $n = 2$; therefore, the comparison made is for index x and $x + 1$. When an error is detected, a recursive call is made to repeat the computation at line 17.

Figure 4 Complement function on 2 threads



```

Microsoft Visual Studio Debug Console

Thread Id: 0; index:0; input: 00000001; complement: 11111110
Thread Id: 1; index:0; input: 00000001; complement: 11111110
Thread Id: 0; index:1; input: 01101101; complement: 10010010
Thread Id: 1; index:1; input: 01101101; complement: 10010010
Thread Id: 0; index:2; input: 10010000; complement: 01101111
Thread Id: 1; index:2; input: 10010000; complement: 01101111
Thread Id: 0; index:3; input: 10111111; complement: 01000000
Thread Id: 1; index:3; input: 10111111; complement: 01000000
Thread Id: 0; index:4; input: 11100010; complement: 00011101
Thread Id: 1; index:4; input: 11100010; complement: 00011101
Thread Id: 0; index:5; input: 01100010; complement: 10011101
Thread Id: 1; index:5; input: 01100010; complement: 10011101
Thread Id: 0; index:6; input: 01100110; complement: 10011001
Thread Id: 1; index:6; input: 01100110; complement: 10011001
Thread Id: 0; index:7; input: 11001010; complement: 00110101
Thread Id: 1; index:7; input: 11001010; complement: 00110101
Thread Id: 0; index:8; input: 01101101; complement: 10010010
Thread Id: 1; index:8; input: 01101101; complement: 10010010
Thread Id: 0; index:9; input: 10000001; complement: 01111110
Thread Id: 1; index:9; input: 10000001; complement: 01111110

```

Execution steps

The execution steps ran to invert the image in this program are highlighted in Table 1 below. Note that the compilation steps require the *libomp* library, which must be downloaded separately. Additionally, *openmp:llvm* command line option must be specified to use the Task feature. Finally, the program generates two output files, namely *in.csv* and *out.csv*, containing the input image and the output data, respectively.

Table 1 Execution steps to achieve image inverse

Step	Description	Outcome
1	Input generation	An array representing a random image
2	Thread team creation	A set of threads to invert the image
3	Tasks creation	A task is created for each thread
4	Tasks await	Tasks completion
5	Results evaluation	Fault detection
6	Output	Output files are saved locally for analysis

While computational redundancy achieves fault tolerance, it increases space complexity and takes a fair amount of work to get it right. For instance, processing the entire input data on each node may not be feasible, depending on the problem size. Additionally, detecting a node failure is not trivial depending on the nature and number of failures. For example, the failure detection mechanism used in the program above assumes that a failure can be detected by comparing the output from each node. Meanwhile, nodes are subject to network latencies and hardware dependencies that could affect the timely delivery of calculated results. Therefore, a robust fault tolerance strategy must account for hardware and software failures.

References

- Fu, H., & Ding, Y. (2010). Using Redundant Threads for Fault Tolerance of OpenMP Programs. International Conference on Information Science and Applications (pp. 1-8). IEEE.
- H. Mushtaq, Z. A.-A. (2011). Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. 2011 IEEE 6th International Design and Test Workshop (IDT) (pp. 12-17). IEEE.
- Hagger, B. (2022, December 23). C++ Team Blog. Retrieved from OpenMP Task Support for C++ in Visual Studio: <https://devblogs.microsoft.com/cppblog/openmp-task-support-for-c-in-visual-studio/>
- Microsoft. (2022, December 28). OpenMP Clauses. Retrieved from learn.microsoft.com: <https://learn.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-clauses?view=msvc-170>