Towards Developing a Repository of Logical Errors Observed in Parallel Code for Teaching Code Correctness

Trung Nguyen Ba
University of Massachusetts at Amherst
Amherst, MA, USA
tnguyenba@cs.umass.edu

Ritu Arora
Texas Advanced Computing Center
University of Texas at Austin
Austin, TX, USA
rauta@tacc.utexas.edu

***Abstract*—Debugging parallel programs can be a challenging task, especially for the beginners. While the debuggers like DDT and TotalView can be extremely useful in tracking down the program statements that are connected to the bugs, often the onus is on the programmers to reason about the logic of the program statements in order to fix the bugs in them. These debuggers may neither be able to precisely indicate the logical errors in the parallel programs nor they may provide information on fixing those errors. Therefore, there is a need for developing tools and educational content on teaching the pitfalls in parallel programming and writing correct code. Such content can be useful to guide the beginners in avoiding commonly observed logical errors and in verifying the correctness of their parallel programs. In this paper, we 1) enumerate some of the logical errors that we have seen in the parallel programs (OpenMP, MPI, and CUDA) that were written by the beginners working with us, and 2) discuss the ways to fix those errors. The errors are mainly related to the data distribution, exiting distributed for-loops, and workload-imbalance. The documentation on these logical errors can contribute in enhancing the productivity of the beginners, and can potentially help them in their debugging efforts. We have added the code samples containing logical errors and their solutions in a Github repository so that the others in the community can reproduce the errors on their systems and learn from them. The content presented in this paper may also be useful for those developing high-level tools for detecting and removing logical errors in parallel programs.**

***Index Terms*—logical errors, parallel programming, code correctness**

I. Introduction

The term "logical error", as used in this paper, refers to those errors in the code that result in unintended behavior or output. Unlike syntax errors, the logical errors do not prevent a program from compiling. A code with logical errors may be having the correct syntax and could be following the constraints of the programming language in which it is written. However, it may not be conforming to the software requirement specifications and may be showing unintended runtime behavior such as crashing when running at scale, or hanging, or producing incorrect results. Due to multiple factors, some of which are enumerated in this section, it can be difficult for the beginners in parallel programming to detect and fix logical errors in parallel programs.

Often, beginners find it challenging to convert existing serial programs into parallel programs (MPI/OpenMP/CUDA/Hybrid) as there are no fixed rules or guidelines for developing parallel programs or checking their correctness [1]. There are several institutions offering parallel programming courses and trainings [2] [3] [4] [5] [6]. We assessed the syllabi and content of some of these courses and training programs and found that, the majority of them do not cover the topics related to code correctness. Even though the debuggers like DDT [7] and TotalView [8] are covered in some courses, and these debuggers can help in isolating the code regions associated with the logical errors, there are no high-level tools or guidelines that teach the programmers on how to detect and fix the errors in the code.

The formal methods for checking the correctness of parallel programs are not widely known and the authors could not find much information on them through literature review. Interestingly, several software engineering courses cover formal methods for checking code correctness [9]. There are some tools and modeling languages, like Petri nets [10], that are covered in software engineering courses which can also be useful for checking the correctness of concurrent programs. These methods can be critical in reliably and theoretically ensuring the correctness of large-scale parallel applications, and in developing high-level tools that can automate the process of error detection [11]. However, such methods are not typically discussed in the parallel programming courses and training programs.

Due to the unavailability of the content that systematically teaches the topic of error detection in parallel programs, the beginners often find it hard to understand the logical errors that they encounter (such as, race conditions, deadlocks, uninitialized variables, and incorrect use of abstractions), and then fix them. This in turn impacts their productivity. Even advance-level parallel programmers may at times find it difficult to detect the logical errors in parallel programs. They may have manually optimized the code to take advantage of certain features in the underlying platform or have written a program mixing multiple parallel programming paradigms (MPI+X),

thereby, making the code complex, and hence, difficult to reason about its correctness [11].

It is also observed that there are parallel applications in which the errors manifest themselves when these applications are ported to new hardware platforms and are compiled using the latest compilers, supporting software stack (e.g., MPI libraries), and environment settings (e.g., memory limits). In fact, the combination of certain compiler flags, and input sizes can also lead to incorrect application behavior (viz., incorrect results and segmentation faults [12]), and it can be hard for beginners to detect and fix such problems. Often, extensive Google search and reading message boards on various websites and mailing lists is involved in the process of troubleshooting the errors. There is no single repository that contains the record of different types of commonly seen logical errors in parallel programs or libraries. Hence, comprehensive, searchable, open-access repositories or databases of logical errors are needed. There is also a need to establish best practices for parallel program verification, and to develop high-level tools and theoretical methods for testing the correctness of large-scale parallel applications.

While developing a rich, searchable, and publicly accessible repository of commonly seen logical errors is a part of our future work, in this paper, we explain some of the logical errors that we have found in the MPI, OpenMP and CUDA programs developed by the beginners working with us, and we also provide solutions to fix those errors. For reproducibility and broadening the impact of our work, we have created a Github repository and have added the complete programs showing the logical errors described in this paper [13]. While the code samples provided in the Github repository can help in reproducing the bugs described in this paper, a thorough study of the broader impact of this repository and any empirical studies associated with it are part of our future work. The content in the repository is available under the new BSD license and the community is welcome to reuse it as needed. We also welcome contributions to this repository. Those interested in contributing to this repository, can first make a copy of the repository, commit their changes to it, and then raise pull requests so that we can review their changes and merge them in the repository.

II. Logical Errors in MPI Programs

We have rougly classified the logical errors that we observed in the MPI programs written by the beginners working with us into different categories. As we continue to collect more samples of the logical errors, the existing category names will be refined and new categories will be added. In this section, we present some of these categories and also include the sample code related to these errors.

A. Incorrect initialization of variables

One of the serial programs that we assigned to the beginners working with us had a for-loop that was the hotspot for parallelization. The variables that were used inside the for-loop, were initialized to 1 just before the beginning of the for-loop. After parallelization, all the MPI processes executed the variable initialization statements before the distributed for-loop and increased the value of the variable inside the for-loop. This led to the wrong result. Table I shows a simplified version of this type of logical error. In this instance, the variable `sum` is initialized to 1 in the serial program. In the erroneous parallel program (snippet shown in column B of Table I), the initialization statement is kept unchanged. Assuming that this program is run with 2 MPI processes, this leads to the value of `sum` in both the MPI processes to be 1. Hence, after the for-loop, the value of `sum` in each MPI process would be increased to 51 and after the reduction process, the value of variable `sum` would be 102. As per the serial program, the final value of `sum` would be 101. The nature of this error is diagrammatically explained in Figure 1. As explained in the aforementioned sentences, the error in the result in this case is due to keeping the original initialization statement (i.e., `sum = 1`) unchanged during the parallelization step.

TABLE I: Incorrect initialization of variables.

For clarity and simplicity, we assume that the parallel programs are run with 2 MPI processes.

A: Serial Program	B: Parallel Program (with logical error)	C: Parallel Program (fixed)

`int sum = 1;`
`for(int i=0;i<100;i++)`
`{`
`sum +=1;`
`}`
	`int range = 100/size;`
	`int start = rank*range;`
	`int end = start + range;`
	`int sum =1;`
	`for(int i=start;i < end;i++)`
	`{`
	`sum +=1;`
	`}`
	`int recv = 0;`
	`MPI_Allreduce(&sum,&recv,1,`
	`MPI_INT,MPI_SUM,`
	`MPI_COMM_WORLD);`
	`sum = recv;`
		`int range = 100/size;`
		`int start = rank*range;`
		`int end = start + range;`
		`int sum = (rank==0)? 1: 0;`
		`for(int i=start;i < end;i++)`
		`{`
		`sum +=1;`
		`}`
		`int recv = 0;`
		`MPI_Allreduce(&sum,&recv,1,`
		`MPI_INT,MPI_SUM,`
		`MPI_COMM_WORLD);`
		`sum = recv;`

It is not hard to fix this error. One way to fix it is to set the value of the reduced variable to a default value depending upon the type of the reduce operation (e.g., 0 for sum, 1 for product). In the case of the example shown in Table I, this error can be fixed if the value of `sum` is set to 0 by all MPI processes but one. On the process with rank 0, we can initialize the value of `sum` to 1.

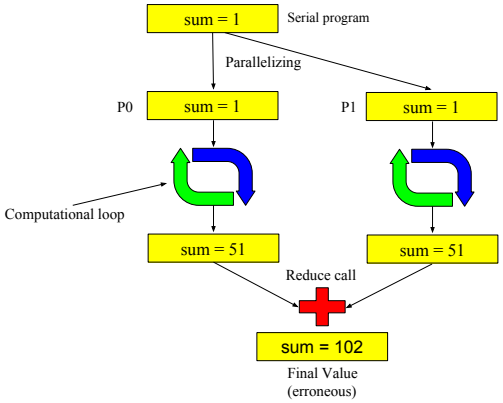

Fig. 1: Incorrect initialization of a variable used in a reduce call.

B. Missing code for handling unknown number of MPI processes

Large-scale MPI programs are typically written such that they can be run with a varying number of MPI processes (a run-time decision) without giving errors. This requires handling the various scenarios of workload distribution amongst the available MPI processes, such as, the scenario in which the number of MPI processes available is greater than the amount of workload, or when the amount of workload is not evenly divisible by the number of MPI processes participating in the computation. The beginners working with us missed adding the logic for handling all such cases, and hence, during testing, some of the sample programs failed to produce the expected results. The code snippet in column B of Table II shows a simplified version of this error, and a fix for it is shown in the column C of Table II.

In the serial code snippet in the column A of Table II, the elements of array `arr` are assigned the values of the loop iteration counter. A snippet of the erroneous parallel version of this code is shown in the column B of Table II. This code produces correct results only when it is run with a certain number of MPI processes - the number should be a factor of 100, which is the number of iteration of the for-loop. If for example, this program is run with 3 MPI processes, the value of the range for each MPI process would be 33 due to integer division. This leads to the total number of elements that are assigned and gathered to be 99 instead of 100.

The code snippet of the corrected parallel version is shown in column C of Table II. The extra (last) iteration is assigned to the MPI process with the highest rank in the group. Notice that because not every MPI process performs the same number of iterations, the number of elements gathered from each processes would be different. Hence, instead of using `MPI_Gather`, we need to use the `MPI_Gatherv` call. In order to use `MPI_Gatherv`, we need to calculate the number of elements that should be gathered from each MPI process (`recvcounts`) and the displacement of these elements (`displs`) from the beginning of the array in which the results are being gathered. The version of the program shown in column C of Table II is not the most efficient one with respect to load-balancing across all the MPI processes but it gives correct results and we use it for clarity and simplicity.

TABLE II: Missing code for handling unknown number of MPI processes.

For clarity and simplicity, we assume that the parallel programs are run with 3 MPI processes.

A: Serial Program

```
int arr[100];

for(int i=0;i<100;i++)
{
  arr[i]=i;
}
```

B: Parallel Program (with logical error)

```
int arr [100];
int range = 100/size;

int start = rank*range;
int end = start + range;

int recvbuf[range];

for (int i=0;i < range;i++)
{
  recvbuf[i]=i;
}

MPI_Gather(recvbuf,range,
       MPI_INT,arr, range,
       MPI_INT, 0,
       MPI_COMM_WORLD);

sum = recv;
```

C: Parallel Program (fixed)

```
int arr [100];
int range = 100/size;

int start =rank*range;
int end = start + range;
int recvcounts [size];
int displs [size];

if (rank == size - 1) {
   end += 100 % size;
   range += 100 % size;
}

for (int i=0;i < size;i++)
{
   recvcounts[i] = range;
   displs[i]= (i==0)? 0:
          recvcounts[i-1];
}

int recvbuf[range];

for (int i=0;i < range;i++)
{
  recvbuf[i] = i;
}

MPI_Gatherv(recvbuf,range,
       MPI_INT,arr,displs,
       recvcounts, MPI_INT,
       0,MPI_COMM_WORLD);
```

C. Incorrect distribution of non-contiguous data

It is not unusual for a serial program to access or update data in a non-contiguous manner. For example, in an array of 100 elements, the element at index number 20 of the array may be updated before the element at index number 10. Parallelizing such programs was challenging for the beginners working with us. Column A of Table III shows a code snippet in which the elements of the array `arr` are updated in a non-contiguous manner. The array elements are updated such that the even-index elements are assigned values before the odd-index elements. The erroneous parallel version of the code is shown in column B of Table III. In this code, the array elements are updated in the correct order by the different MPI processes, but they are not collected correctly using the `MPI_Gather` call. The first argument to the `MPI_Gather` call specifies the address of the buffer from which the data should be copied (in the code snippet in column B of Table III it is `arr[start]`) and the second argument specifies the number of *continuous* elements that should be copied from the buffer (in the code snippet in column B of Table III this argument is `range`). Figure 2 describes the error diagrammatically. This error leads to collecting wrong values in the result array named `rev` and, subsequently leads to incorrect output from the program.

The snippet of the correct parallel code for this example is shown in column C of Table III. There is a need for a mechanism to keep track of the updated array elements for each MPI process. In the code snippet in column C of Table III, a temporary array named `sendbuff` is created for storing updated elements of the array, and the variable `counter` is used to track the total number of elements that are updated. With this approach, the data gathered using the `MPI_Gather` call is now correct. However, in order to achieve the desired arrangement of elements in the output array, a rearrangement of elements is necessary. For this example, since the correct computation pattern of the elements is known beforehand, a simple for-loop can be used to achieve such a rearrangement. A more robust solution for this problem involves using the derived data types in MPI.

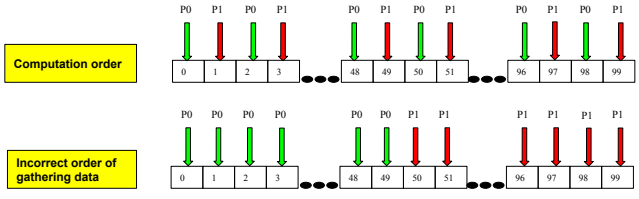

Fig. 2: Incorrect order of gathering non-contiguous data.

TABLE III: Incorrect collection of non-contiguous data.

For clarity and simplicity, we assume that the parallel programs are run with 2 MPI processes.

A: Serial Program	B: Parallel Program (with logical error)	C: Parallel Program (fixed)

`int arr [100];`
`for(int i=0;i<100;i++)`
`{`
`if (i < 50)`
`{`
`arr[i*2]=2;`
`}`
`else`
`{`
`arr[(i-50)*2+1]=3;`
`}`
`}`
	`int arr [100];`
	`int rev [100];`
	`int range = 100/size;`
	`int start = rank*range;`
	`int end = start + range;`
	`for(int i=start;i<end;i++)`
	`{`
	`if (i < 50)`
	`{`
	`arr[i*2]=2 ;`
	`}`
	`else`
	`{`
	`arr[(i-50)*2+1]=3;`
	`}`
	`}`
	`MPI_Gather(&arr[start],range`
	`,MPI_INT,rev,range,`
	`MPI_INT,0,`
	`MPI_COMM_WORLD);`
	`arr = rev;`
		`int arr [100];`
		`int rev [100];`
		`int range = 100/size;`
		`int start = rank*range;`
		`int end = start + range;`
		`int sendbuff[range];`
		`int counter = 0;`
		`for (int i=start;i<end;i++)`
		`{`
		`if (i < 50)`
		`{`
		`arr[i*2] =2 ;`
		`sendbuff[counter++]=arr[i*2];`
		`}`
		`else {`
		`arr[(i-50)*2+1] =3;`
		`sendbuff[counter++]=arr[i*2];`
		`}`
		`}`
		`MPI_Gather(sendbuff,range,`
		`MPI_INT,rev,range,`
		`MPI_INT,0,`
		`MPI_COMM_WORLD);`
		`for (int i=0;i<100;i++)`
		`{`
		`if (i < 50)`
		`{`
		`arr[i*2]=rev[i] ;`
		`}`
		`else`
		`{`
		`arr[(i-50)*2+1]=rev[i];`
		`}`
		`}`

D. Incorrect choice of collectives for handling data distribution

Uneven distribution of data happens when all the MPI processes in a group do not receive the same amount of data due to the logic of the program or the problem to be solved. This causes some MPI processes to execute more or fewer steps as compared to the other MPI processes. Column A of Table IV shows the code snippet of the serial version of a program, and column B shows the parallelized version having uneven distribution of data. In this example, two elements of an array are initialized in each iteration of the for-loop, except the last iteration, in which only one element is initialized. For simplicity and clarity, we assume that the parallel version is run with 5 MPI processes. Due to the code in the for-loop, the MPI process that executes the last iteration would have one less array element updated compared to other processes.

The beginners working with us used the `MPI_Gather` call in this case. This resulted in array `rev` having correct values for elements 0 to 99. However, due to the way the program is written, the `MPI_Gather` call may gather 100 elements in total in some cases instead of 99. Consequently, this type of error can result in memory leak, segmentation fault, or wrong output in the computations downstream. To fix this error, the programmers need to have a method to determine the exact number of array elements to be updated by each MPI process. For the example in Table IV, the computational workload (or the number of array elements to be updated) for each MPI process can be computed and stored in the `sendcount` variable. Because a different number of data elements are collected from different MPI processes, the correct MPI call to use instead of `MPI_Gather` here is `MPI_Gatherv`. Just like in the example shown in section B, `recvcounts` and `displs` are computed as well for passing as parameters to `MPI_Gatherv`.

TABLE IV: Incorrect choice of collectives for handling data distribution.

For clarity and simplicity, we assume that the parallel programs are run with 5 MPI processes.

A: Serial Program	B: Parallel Program (with logical error)	C: Parallel Program (fixed)

`int arr[99];`
`for(int i=0;i<50;i++)`
`{`
`arr[i*2]=1;`
`if (i < 49)`
`{`
`arr[i*2+1]=2;`
`}`
`}`
	`int arr [99];`
	`int rev[99];`
	`int range = 50/size;`
	`int start = rank*range;`
	`int end = start+range;`
	`for(int i=start;i<end;i++)`
	`{`
	`arr[i*2]=1;`
	`if (i < 49)`
	`{`
	`arr[i*2+1] =2;`
	`}`
	`}`
	`MPI_Gather(&arr[rank*range`
	`*2],range*2,MPI_INT,`
	`rev,range*2,MPI_INT`
	`,0,MPI_COMM_WORLD);`
		`int arr [99];`
		`int rev[99];`
		`int range = 50/size;`
		`int start = rank*range;`
		`int end = start + range;`
		`int sendcount=(rank==size-1)`
		`? 1: 2;`
		`int recvcounts[size];`
		`int displs[size];`
		`for (int i=0;i<size;i++)`
		`{`
		`recvcounts[i]=(i==size-1)?`
		`1: 2;`
		`displs[i]=(i==0)? 0:`
		`recvcounts[i-1];`
		`}`
		`for (int i = start; i < end;`
		`i++)`
		`{`
		`arr[i*2]=1;`
		`if (i < 49) {`
		`arr[i*2+1] =2;`
		`}`
		`}`
		`MPI_Gatherv(&arr[rank*range`
		`*2],sendcount,MPI_INT`
		`,rev,displs,`
		`recvcounts,MPI_INT,0,`
		`MPI_COMM_WORLD);`

E. Incorrect handling of break statements in for-loops

It is common to have break statements in the for-loops in serial programs. When programs with for-loops are parallelized using MPI, as a first step, the loops are checked for dependencies. When there are no dependencies in a loop, the loop iterations can be distributed across all the MPI processes in the group. The distribution of the iterations are done as evenly as possible to avoid any load imbalance, but at times, especially when the iterations are not evenly divisible across all the MPI processes, some processes in the group could be doing more work than the others (also covered in subsection D). One question that arises is: *upon finding the for-loop exit condition in an iteration (i.e., upon finding a break statement), how can an MPI process alert the other MPI processes in the group about the exit statement before itself making a graceful exit from the loop?* It is important that all the MPI processes exit from the loop when an exit statement is found by one MPI process so that the result from the parallel implementation matches the result from the serial version of the program. One way to gracefully exit from a for-loop when a break statement is encountered is by using a broadcast (`MPI_Bcast`) or a reduce (`MPI_Reduce`/`MPI_Allreduce`) call in the loop. The MPI process that encounters the break statement can broadcast a variable to all the MPI processes to inform about the break statement that it has encountered - this is a collective call and should be visible to all the MPI processes in the group. All MPI processes can have a variable, for example `flag`, that is initialized to 0. When an MPI process finds the break statement it can increment this `flag` variable by 1 as an indicator that the break statement has been found. The beginners working with us invoked the `MPI_Allreduce` call with `MPI_SUM` operation to update all the MPI processes about the updated value of the `flag` and then used the updated value of the `flag` to exit from the for-loop (see the `result` variable in column B of Table V). However, if the for-loop iterations are not evenly distributed across all the MPI

TABLE V: Incorrect handling of break statements in for-loops.

For clarity and simplicity, we assume that the parallel programs are run with 3 MPI processes.

A: Serial Program	B: Parallel Program (with logical error)	C: Parallel Program (fixed)

`for (int i=0;i<20;i++)`
`{`
`if (i == 19)`
`{`
`break;`
`}`
`}`
	`int range = (20/size);`
	`int start = rank*range;`
	`int end = start + range ;`
	`if (rank == size - 1)`
	`{`
	`end = end + (20 % size);`
	`}`
	`for (int i=start;i<end;i++)`
	`{`
	`int flag = 0;`
	`if (i ==19)`
	`{`
	`flag =1;`
	`}`
	`int result = 0;`
	`MPI_Allreduce(&flag,&result`
	`,1,MPI_INT,MPI_MAX,`
	`MPI_COMM_WORLD);`
	`if (result == 1)`
	`{`
	`break;`
	`}`
	`}`
		`int range = (20/size);`
		`int start = rank*range;`
		`int end = start + range ;`
		`if (rank == size - 1)`
		`{`
		`end = end + (20 % size);`
		`}`
		`for (int i=start;i<end;i++)`
		`{`
		`int flag=0;`
		`if (i==(end-1)||(i==19)){`
		`flag =1;`
		`}`
		`int result = 0;`
		`if (flag == 1)`
		`{`
		`MPI_Allreduce(&flag,&`
		`result,1,MPI_INT,`
		`MPI_SUM,`
		`MPI_COMM_WORLD);`
		`if(result == size)`
		`{`
		`break;`
		`}`
		`}`
		`}`

processes, then it is possible that by the time an MPI process reaches the break statement, updates the value of the `flag` variable, and reaches the `MPI_Allreduce` call, the other MPI processes may have already completed their work and exited from the for-loop. Figure 3 illustrates this case with a diagram. Hence, a situation arises when one MPI process indefinitely continues to wait (or block) on the other MPI processes to complete the `MPI_Allreduce` call, and the program hangs. The code snippet depicting this scenario is shown in column B of Table V.

One way to avoid the situation that causes the code to hang, is to force all the MPI processes to wait at the `MPI_Bcast` or `MPI_Allreduce` call, and the code snippet for achieving this is shown in column C of Table V. Assume that we have shown the case of dividing 20 iterations of a for-loop across 3 MPI processes in this code snippet. The MPI process with rank 2 gets two extra iterations to work on. As soon as all the MPI processes have reached the last iteration of their share of for-loop iterations, or if the value of the loop variable `i` is 19 (the original `break` condition in the serial program shown in column A of Table V), then a `flag` variable is set to 1. All the MPI processes that set `flag` equal to 1 are then guaranteed to reach the `MPI_Allreduce` statement that is inside the if-condition (`if (flag == 1)`), and wait for each other to complete their work or send/receive a signal to break from the for-loop through the `MPI_Allreduce` call. Note that instead of `MPI_MAX` operation, we are using `MPI_SUM` operation in column C of Table V, and are breaking from the for-loop when the value of `result` is equal to the number of MPI processes participating in the group.

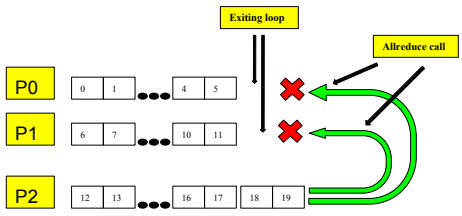

Fig. 3: Incorrect handling of break statements in for-loops.

TABLE VI: Incorrect selection of data sharing/storage attribute.

A: Serial Program	B: Parallel Program (with logical error)	C: Parallel Program (fixed)

`int arr[5];`
`int sum=0;`
`for(int i=0;i<100;i++)`
`{`
`for(int j=0;j<5;j++)`
`{`
`arr[j] = i+j;`
`}`
`sum+=(arr[0]+arr[1]+`
`arr[2]+arr[3]+`
`arr[4])%2;`
`}`
	`int arr[5];`
	`int sum = 0;`
	`#pragma parallel for shared(`
	`arr,sum) reduction(+:`
	`sum)`
	`for ( int i = 0;i < 100;i++)`
	`{`
	`for (int j =0;j < 5;j++)`
	`{`
	`arr[j] = i +j ;`
	`}`
	`sum+=(arr[0]+arr[1]+arr[2]+`
	`arr[3]+ arr[4])%2;`
	`}`
		`int arr[5];`
		`int sum = 0;`
		`#pragma parallel for shared(`
		`sum) private(arr)`
		`reduction(+:sum)`
		`for(int i=0;i < 100;i++)`
		`{`
		`for (int j=0;j < 5;j++)`
		`{`
		`arr[j]= i+j;`
		`}`
		`sum+=(arr[0]+arr[1]+arr`
		`[2]+arr[3]+arr[4])`
		`%2;`
		`}`

III. Logical Errors in OpenMP Programs

We have classified the logical errors that we observed in the OpenMP programs written by the beginners working with us into different categories. In this section, we present some of these categories and also include the sample code related to these errors.

A. Incorrect selection of data sharing/storage attributes

The beginners working with us made errors in correctly using the data sharing/storage attributes in OpenMP, such as, `shared`, and `private` and as mentioned in [14], this seems to be a common pitfall while writing OpenMP programs. At times, it may not be easy even for experienced programmers to select the correct data sharing/storage attributes. Table VI shows a serial code snippet and an OpenMP version of it in which a variable that should be added to the `private` clause is erroneously added to the `shared` clause. In the serial program (code snippet shown in column A of Table VI), each iteration of the outer-loop updates the array `arr` and then uses it to compute the value of the variable `sum`. Often, in OpenMP code, arrays are listed under `shared` clause and the same is done in the erroneous parallel program (code snippet in column B). However, in this case, including `arr` in the `shared` clause results in nondeterministic output.

The error occurs because during each iteration of the parallelized loop (outer-loop), the whole array `arr` is updated, which leads to data-race condition between OpenMP threads. In this case, since all the elements of the array named `arr` are updated in each iteration and each change is independent, array `arr` needs to be added to `private` clause so that each thread can safely update `arr` value without interfering with other threads.

B. Incorrect use of `critical` and `atomic` sections

Even though it would be ideal if the code inside a parallel section does not have any loop or data dependencies, sometimes, executing certain lines of code in a serial manner may be necessary. However, deciding on how many lines of code to run serially can be tricky. The beginners working with us either ran too many statements in a parallel region serially or too few. Serially running too many lines of code in a parallel section slows down the program and not running enough number of lines in serial in some cases can lead to wrong output. The literature review shows that this is a common OpenMP error [14]. Table VII shows an example of this type of error

TABLE VII: Incorrect use of critical or atomic section.

A: Serial Program	B: Parallel Program (with logical error)	C: Parallel Program (fixed)

`int arr[10];`
`for(int i=0;i<10;i++)`
`{`
`arr[i] = i;`
`}`
`for(int i=0;i<100;i++)`
`{`
`int sum = rand();`
`for(int j=0;j<10;j++)`
`{`
`arr[j]+=sum;`
`}`
`}`
	`int arr[10];`
	`for (int i=0;i < 10;i++)`
	`{`
	`arr[i] = i;`
	`}`
	`#pragma omp parallel for`
	`shared(arr)`
	`for ( int i=0;i < 100;i++)`
	`{`
	`int sum = rand();`
	`for (int j=0;j < 10;j++)`
	`{`
	`arr[j]+=sum;`
	`}`
	`}`
		`int arr[10];`
		`for (int i=0;i < 10;i++)`
		`{`
		`arr[i] = i;`
		`}`
		`#pragma omp parallel for`
		`shared(arr)`
		`for ( int i=0;i < 100;i++)`
		`{`
		`int sum=rand();`
		`#pragma omp critical`
		`{`
		`for (int j=0;j < 10;j++)`
		`{`
		`arr[j] +=sum;`
		`}`
		`}`
		`}`

in column B. The output of this OpenMP program would be undefined due to running lesser than the required number of lines serially.

In the serial nested for-loops shown in the code snippet in column A of Table VII, the inner for-loop is used to update the value of the elements in array `arr`. Notice that the entire array is updated in each iteration of the outer-loop - the loop that is parallelized. The erroneous OpenMP version of this code is shown in column B of Table VII. Here, the array `arr` is specified in the `shared` clause. Unlike the example shown in section A, specifying the array `arr` in the `shared` clause for this example is correct. This is because each thread only partially updates the array in their share of iterations, and all the threads should have access to the entire array. The error in this example is linked to not running certain lines of code in serial, and hence, multiple threads are able to update the same elements of `arr` at the same time, thereby, leading to a data-race condition.

The correct version of the OpenMP code for this example is shown in column C of Table VII. As can be noticed from this code snippet, the lines of code for updating the array `arr` are wrapped in an OpenMP `critical` block. This `critical` block allows only one OpenMP thread at a time to update the wrapped code. By doing so, it resolves the data-race error caused by accessing the same array elements simultaneously from multiple threads.

C. Incorrect specification of storage attributes for the variables used in both serial and parallel regions

The code in column B of Table VIII shows a simple case of incorrect handling of data across serial and parallel regions, causing the program to give wrong answer. In all the three programs - shown in columns A, B, and C - the initial value of the index `i` of the for-loop is random between 0 and 1. In order to execute correctly, the parallel program needs to have the value of `i` that is initialized outside the parallel region to also be available inside the parallel region. However, in the erroneous version of the parallel code snippet shown in column B of Table VIII, `i` is added to the `private` clause. Hence, the value of `i` after entering the loop is nondeterministic and can cause different kinds of errors including wrong output, slow termination, etc. The error in this example can be easily fixed by switching `i` from `private` clause to `firstprivate` clause.

TABLE VIII: Incorrect specification of storage attributes for the variables used in serial and then parallel region.

A: Serial Program	B: Parallel Program (with logical error)	C: Parallel Program (fixed)

`int sum=1;`
`int i=rand()%2;`
`for (i;i < 10;i ++)`
`{`
`sum +=1;`
`}`
	`int sum=1;`
	`int i=rand()%2;`
	`#pragma omp parallel for`
	`private(i)`
	`reduction(+:sum)`
	`for (i=i;i < 10;i ++)`
	`{`
	`sum +=1;`
	`}`
		`int sum=1;`
		`int i=rand()%2;`
		`#pragma omp parallel for`
		`firstprivate(i)`
		`reduction(+:sum)`
		`for (i=i;i < 10;i ++)`
		`{`
		`sum +=1;`
		`}`

TABLE IX: Incorrect specification of storage attributes for the variables used in parallel and then serial region.

A: Serial Program	B: Parallel Program (with logical error)	C: Parallel Program (fixed)

`int sum=0;`
`int i=rand()%2;`
`for (i;i < 10;i++)`
`{`
`sum +=1;`
`}`
`int equal=(sum == i);`
	`int sum=0;`
	`int i=rand()%2;`
	`#pragma omp parallel for`
	`firstprivate(i)`
	`reduction(+:sum)`
	`for ( i=i;i < 10;i ++)`
	`{`
	`sum +=1;`
	`}`
	`int equal=(sum == i);`
		`int sum=0;`
		`int i=rand()%2;`
		`#pragma omp parallel for`
		`lastprivate(i)`
		`reduction(+:sum)`
		`for ( i=i;i < 10;i ++)`
		`{`
		`sum +=1;`
		`}`
		`int equal=(sum == i);`

Table IX shows another case of updating a variable in both serial and parallel regions of the code. In this example, the loop index `i` is used after the for-loop. Due to this setup, adding `i` to `firstprivate` or `private` would not be correct since the value of `i` after the loop is nondeterministic. Therefore, in order to resolve this, `i` needs to be added to `lastprivate` clause.

Note that computational overheads are involved in using `firstprivate` or `lastprivate`. Therefore, developers need to understand and use them only when needed.

IV. Logical Errors in CUDA Programs

Finding logical errors in CUDA programs can be difficult for beginners especially when they are still adjusting to the concepts of CUDA kernels, explicit memory management, memory hierarchies, and data transfer between the host and the GPU. In this section, we describe couple of logical errors that we noticed in the CUDA programs that were written by the beginners working with us.

A. Incorrect initializing of the arrays inside the kernel

While converting nested for-loops (see column A of Table X) into a CUDA kernel, it may be required to allocate arrays on the GPU's global memory using a function like `cudaMalloc` and initialize them. The snippet of the CUDA kernel shown in column B of Table X was written by a beginner. In this code snippet, the array named `dNpairs` is initialized to 0 in the kernel. The way this code is written, all the threads in the grid will execute the lines of code in which the array `dNpairs` is initialized to 0. These threads may not be working exactly simultaneously. Hence, by the time a thread initializes the array elements to 0 and finishes updating them further, another thread could still be at the initialization step and thus resetting the values of the array elements to 0. This scenario is likely to produce incorrect results. One way to fix

TABLE X: Incorrect initilization of array inside kernel.

A: Serial Program	B: Parallel Program (with logical error)	C: Parallel Program (fixed)
`for(i=0;i<N;i++) { for(j=(i+1);j<N;j++) { // other code if (flag) { //other computations npairs[mbin+1]++; } //other calculation } }`	`void __global__ kernel0(..., int device_N, int64_t * dNpairs,...){ int64_t i = blockIdx.x * blockDim.x + threadIdx.x; for (j=0; j<N*N; j++) { dNpairs[j] = 0; } if((i< N)) { for(j=(i+1);j<N;j++) { // other code if (flag) { //other computation dNpairs[mbin+1]++; } //other calculation } } }`	`void __global__ kernel0(..., int device_N, int64_t * dNpairs,...){ int64_t i = blockIdx.x * blockDim.x + threadIdx.x; if(i==0) { for (j=0; j<N*N; j++) { dNpairs[j] = 0; } } __syncthreads() if((i< N)) { for(j=(i+1);j<N;j++) { // other code if (flag) { //other computation dNpairs[mbin+1]++; } //other calculation } } }`

this code is to have only one thread initialize the entire array - the fixed code is shown in column C of Table X. Another way to fix this error is to move the initialization step to the function from where the kernel is invoked. As yet another way, each thread could be made to update only selected elements of the given arrays.

B. *Incorrect use of* `__syncthreads`

In the sample code, shown in column A of Table XI, an array of `N` elements is being updated in a nested for-loop. To split the process of updating this array across multiple GPU threads by flattening the outer for-loop, a global array was constructed. The number of elements in this global array was equal to (`N`**number of threads*). Each thread updated its set of `N` elements. After the threads had finished updating the elements in their share of array, an element-wise reduction of the array across the grid was needed to get the final result. The value of the elements in the resulting array would be equal to the sum total of the values of the corresponding elements in the copies of the array that were local to each thread in the grid. The beginners working with us called `__syncthreads()` in a CUDA kernel just before performing the element-wise reduction of the array. They hoped that calling `__syncthreads()` would help in synchronizing the threads across the grid. However, calling `__syncthreads()` in the kernel only ensures that all warps in a thread block have synchronized - in other words, it ensures that all the threads in a block have reached a barrier. Since calling `__syncthreads()` does not force inter-block synchronization of the threads, the code produced incorrect results for the element-wise reduction of the array. Figure 4 shows a diagram for this error.

To reliably synchronize all the threads in a grid before doing the element-wise reduction of an array, one can exit from the kernel in which the elements of the array are being computed, and immediately call another kernel in which the element-wise reduction can be done. The code snippet in column C of Table XI shows this method and will guarantee the synchronization of the threads after exiting `kernel0`.

TABLE XI: Incorrect use of `__syncthreads`.

A: Serial Program	B: Parallel Program (with logical error)	C: Parallel Program (fixed)
`for(i=0;i<N;i++) { for(j=(i+1);j<N;j++) { // other code if (flag) { //other computation npairs[mbin+1]++; } //other calculation } }`	`void __global__ kernel0(..., int device_N, int64_t * dNpairs,...) { int64_t i =blockIdx.x* blockDim.x+ threadIdx.x; if((i< N)) { for(j=(i+1);j<N;j++) { // other code if (flag) { //other computation dNpairs[mbin+1]++; } //other calculation } } __syncthreads(); // do element-wise //reduction of dNpairs } int main(){ //other code kernel0<<<dimGrid,dimBlock >>>(..); //pther code }`	`void __global__ kernel0(..., int device_N, int64_t * dNpairs,...) { int64_t i =blockIdx.x* blockDim.x+ threadIdx.x; if((i< N)) { for(j=(i+1);j<N;j++) { // other code if (flag) { //other computation dNpairs[mbin+1]++; } //other calculation } } } /*second kernel*/ void __global__ kernel1(...) { int64_t i = blockIdx.x * blockDim.x + threadIdx.x; if(i==0) { // do element-wise //reduction of dNpairs } } int main(){ //other code kernel0<<<dimGrid,dimBlock >>>(..); kernel1<<<dimGrid,dimBlock >>>(..); //other code }`

V. Related Work

To the best of our knowledge, there is no publicly accessible repository containing sample parallel programs showing some of the common logical errors. However, there are multiple research projects that have explored the topic of HPC code correctness. These studies can be characterized into three areas:

A. *Studies on bugs in parallel programming*

There are various studies about bugs in parallel and concurrent programs due to the need for reliable HPC software. Many aspects of the bugs have been explored, such as, bug characteristics [15], commonly seen patterns of bugs [14], impact of the bugs [16] [17], and the errors in parallel programs that get exposed when run at scale [18]. Unfortunately, not much work has been done in creating an organized and characterized database for bugs in parallel and concurrent programs. The work presented in this paper is our preliminary step towards building such a database.

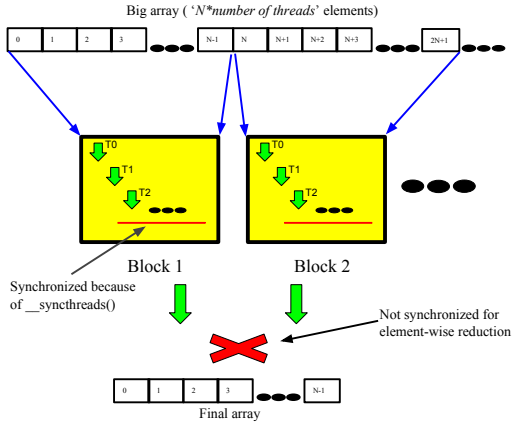

Fig. 4: Incorrect use of `__syncthreads`.

B. Debugging and verification of parallel programs

Debugging parallel programs can be difficult. There is a large body of literature in this area that can be useful for understanding the challenges in the process of verification and validation of parallel programs. For debugging MPI programs, many methods have been proposed, such as, checking runtime information [19] [20] [21] [22], verification of correctness based on MPI scheduling [23], or based on the combination of symbolic execution and model checking [24]. Chen *et al.* have built FlowChecker [25] to detect the bugs inside MPI libraries instead of bugs in user programs. However, these studies focus mostly on communication errors (e.g., deadlocks) or API/resource misuse (e.g., mismatch in the send and receive data types). Our focus is on developing content that can educate the developers on the topic of logical errors in parallel programs, and provide better understanding about these errors.

There is a large body of work in the area of debugging OpenMP programs too, and some tools have also been developed to debug OpenMP programs, such as those mentioned in [26] and [8]. Techniques such as online-offline model [27] have also been suggested. In addition, some studies have focused on special bugs such as data-race conditions [28] [29]. However, the use of these methods or tools requires additional resources (e.g., computation-time and memory) and these methods do not provide explanations for the occurrences of the bugs. Our work complements these studies as it provides complete code of the actual test cases showing errors, and presents a discussion on those errors so that it can benefit the community in developing new techniques and models for checking the logical correctness of OpenMP programs.

Similarly, many research efforts focusing on debugging and verifying CUDA programs have been undertaken. For general debugging, the most notable tool is CUDA-GDB [30] developed by Nvidia. Techniques to dynamically detect data-race condition and share memory bank conflict errors have been developed by Boyer *et al.* [31]. In addition to this, multiple verification tools and techniques are also proposed such as GPUVerify, a verification tool based on operational semantics developed by Betts *et al.* [32]. Leung *et al.* have worked on verifying GPU kernels via test amplification [33]. However, there is still a lack of resources and materials for developers, especially beginners to help them understand the incorrect CUDA code. Our work aims to improve this issue by providing examples of common errors in CUDA programs, their explanations, and how to avoid them.

C. Parallel programming frameworks and automation

Another related area comprises of high-level frameworks and tools for developing parallel programs. Suzaku [34], a pattern-based framework for MPI programming has been developed by Wilkinson and Ferner. A semi-automatic parallel code transformation tool named IPT [35], and semi-automatic checkpointing tool named ITALC [36] have been developed by Arora *et al.* Algorithms for automatic OpenMP "for-loop" parallelization have also been proposed by different groups [37] [38]. The work presented in this paper is complementary to these efforts as it can be used as a guideline to improve these high-level frameworks and tools such that they can avoid generating parallel code with the aforementioned logical errors.

VI. Future Work

We plan to incorporate the knowledge on the patterns of logical errors in our high-level parallelization tool named IPT [35] - so that at the time of generating parallel code using IPT, we can ensure its logical correctness too.

We have set-up online quizzes on the topic of parallel program correctness [13]. These quizzes are timed and present code snippets with logical errors. They are designed to help us in collecting data on the difficulty level of the logical errors - that is, how difficult is it to spot the errors. We have started using these quizzes in our parallel programming training and are hopeful of collecting enough data-points for doing meaningful analyses and evaluations on the productivity of the parallel programmers over a period of a year.

The software engineering community has already developed tools and techniques for the verification of serial programs. We plan to explore these tools and techniques, and find the relevant ones that can be adapted for supporting the verification of parallel programs.

VII. Conclusions

In this paper, we have presented an overview of some of the logical errors seen in MPI, OpenMP, and CUDA programs written by the beginners. We believe that the awareness about such errors can help the beginners in detecting and fixing similar errors in their real-world applications. We have provided the code samples discussed in this paper in a Github repository so that the others in the community can take advantage of our work in their education and training efforts. In doing so, we also ensure that the work presented in this paper is reproducible.

VIII. Acknowledgement

The work presented in this paper was made possible through the National Science Foundation (NSF) award number 1642396. We are very grateful to NSF for the same.

References

[1] R. Arora, P. Bangalore, and M. Mernik, "Raising the level of abstraction for developing message passing applications," in *The Journal of Supercomputing*, Springer, US, 2012.

[2] W. Bohm, "CS 475: Parallel programming," 2017. https://www.cs.colostate.edu/~cs475/CurrentSemester/. Accessed on August-17-2018.

[3] Cornell University-Center of Advanced Computing, "Parallel computing on stampede," 2013. https://www.cac.cornell.edu/education/training/StampedeOct2013.aspx. Accessed on August-17-2018.

[4] Nvidia, "Intro to parallel programming," 2018. https://eu.udacity.com/course/intro-to-parallel-programming--cs344. Accessed on August-17-2018.

[5] B. Barney, "Introduction to parallel computing," 2018. https://computing.llnl.gov/tutorials/parallel_comp/. Accessed on August-17-2018.

[6] TACC-Training, "Parallel programming foundations," 2018. https://www.tacc.utexas.edu/education/institutes/parallel-programming-foundations. Accessed on August-17-2018.

[7] ARM, "ARM-DDT." https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt. Accessed on August-17-2018.

[8] B. Barney, "TotalView." https://computing.llnl.gov/tutorials/totalview/. Accessed on August-17-2018.

[9] M. Priestley, "The logic of correctness in software engineering," in *A Science of Operations*, pp. 253–276, Springer, London, 2011.

[10] J. L. Peterson, "Petri nets," *ACM Comput. Surv.*, vol. 9, pp. 223–252, Sept. 1977.

[11] G. Gopalakrishnan, P. D. Hovland, C. Iancu, S. Krishnamoorthy, I. Laguna, R. A. Lethin, K. Sen, S. F. Siegel, and A. Solar-Lezama, "Report of the HPC correctness summit, jan 25-26, 2017, washington, DC," *CoRR*, vol. abs/1705.07478, 2017.

[12] "Segmentation fault when using large array with openmp," 2009. https://software.intel.com/en-us/forums/intel-fortran-compiler-for-linux-and-mac-os-x/topic/269671. Accessed on August-17-2018.

[13] R. Arora and T. N. Ba, "Bug patterns." https://github.com/ritua2/IPT/tree/master/bug-patterns. Accessed on Sep-25-2018.

[14] M. Süß and C. Leopold, "Common mistakes in openmp and how to avoid them," *OpenMP Shared Memory Parallel Programming*, 2005. IWOMP 2005.

[15] M. Zhang, Y. Wu, K. Chen, and W. Zheng, "What is wrong with the transmission? a comprehensive study on message passing related bugs," in *2015 44th International Conference on Parallel Processing*, pp. 410–419, Sept 2015.

[16] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," *SIGOPS Oper. Syst. Rev.*, vol. 35, pp. 73–88, Oct. 2001.

[17] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pp. 221–230, June 2010.

[18] H. Li, Z. Chen, R. Gupta, and M. Xie, "Non-intrusively avoiding scaling problems in and out of mpi collectives," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 415–424, May 2018.

[19] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov, "Automated, scalable debugging of mpi programs with intel message checker," in *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, SE-HPCS '05, (New York, NY, USA), pp. 78–82, ACM, 2005.

[20] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller, "A graph based approach for mpi deadlock detection," in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, (New York, NY, USA), pp. 296–305, ACM, 2009.

[21] Lawrence Livermore National Laboratory, "Stat: the stack trace analysis tool." https://github.com/LLNL/STAT. Accessed on August-17-2018.

[22] B. Krammer, K. Bidmon, M. MÃijller, and M. Resch, "Marmot: An mpi analysis and checking tool," in *Parallel Computing* (G. Joubert, W. Nagel, F. Peters, and W. Walter, eds.), vol. 13 of *Advances in Parallel Computing*, pp. 493 – 500, North-Holland, 2004.

[23] A. Vo and G. Gopalakrishnan, "Scalable verification of mpi programs," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pp. 1–4, April 2010.

[24] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke, "Combining symbolic execution with model checking to verify parallel numerical programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, pp. 10:1–10:34, May 2008.

[25] Z. Chen, Q. Gao, W. Zhang, and F. Qin, "Flowchecker: Detecting bugs in mpi libraries via message flow checking," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, Nov 2010.

[26] Intel, "Intel Inspector." https://software.intel.com/en-us/intel-inspector-xe. Accessed on August-17-2018.

[27] J. Li, D. Hei, and L. Yan, "Correctness analysis based on testing and checking for openmp programs," in *2009 Fourth ChinaGrid Annual Conference*, pp. 210–215, Aug 2009.

[28] Y. Kim, S. Song, and Y. Jun, "Adat: An adaptable dynamic analysis tool for race detection in openmp programs," in *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, pp. 304–310, May 2011.

[29] S. Atzeni, G. Gopalakrishnan, Z. Rakamarić, I. Laguna, G. L. Lee, and D. H. Ahn, "Sword: A bounded memory-overhead detector of OpenMP data races in production runs," in *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2018. To appear.

[30] Nvidia, "CUDA-GDB." https://developer.nvidia.com/cuda-gdb. Accessed on August-17-2018.

[31] M. Boyer, K. Skadron, and W. Weimer, "Automated Dynamic Analysis of CUDA Programs," in *Third Workshop on Software Tools for Multi-Core Systems*, 2008.

[32] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: A verifier for gpu kernels," *SIGPLAN Not.*, vol. 47, pp. 113–132, Oct. 2012.

[33] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner, "Verifying gpu kernels by test amplification," in *PLDI*, 2012.

[34] B. Wilkinson and C. Ferner, "The suzaku pattern programming framework," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 978–986, May 2016.

[35] R. Arora, J. Olaya, and M. Gupta, "A tool for interactive parallelization," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, XSEDE '14, (New York, NY, USA), pp. 51:1–51:8, ACM, 2014.

[36] R. Arora and T. N. Ba, "Italc: Interactive tool for application-level checkpointing," in *Proceedings of the Fourth International Workshop on HPC User Support Tools*, HUST'17, (New York, NY, USA), pp. 2:1–2:11, ACM, 2017.

[37] A. G. Bhat, M. N. Babu, and A. M. R, "Towards automatic parallelization of "for" loops," in *2015 IEEE International Advance Computing Conference (IACC)*, pp. 136–142, June 2015.

[38] M. Mathews and J. P. Abraham, "Automatic code parallelization with openmp task constructs," in *2016 International Conference on Information Science (ICIS)*, pp. 233–238, Aug 2016.