Deadlock Detection Software for MPI

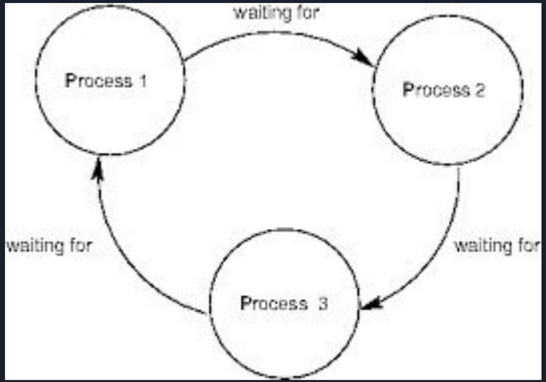
by Matthew Ong You Zhi, 31536492

Introduction

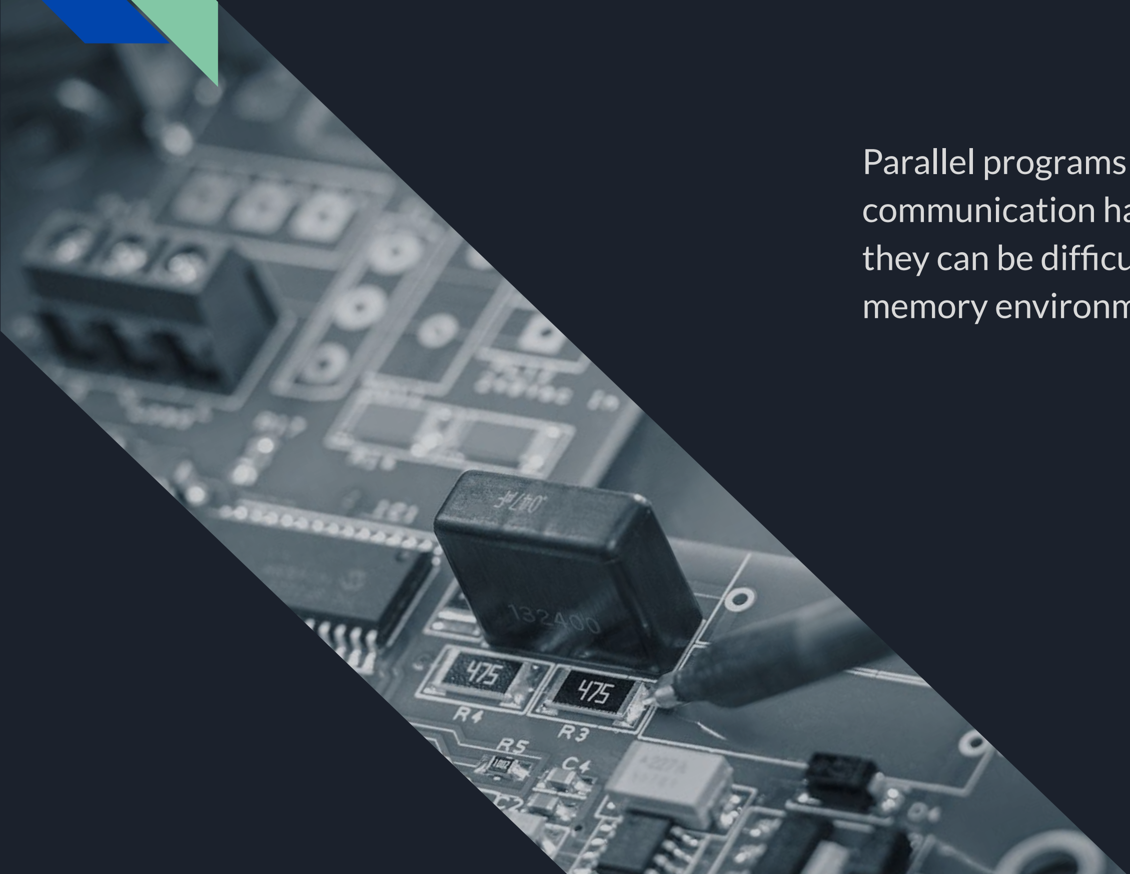
- Large number of computational programs use parallel computing(Quantum Computing, Bio Based Techniques)
- The implication of Ahmdal's Law is that increasing a task's parallel component is the only way to noticeably enhance speed.
- With distributed computing, complex computational applications can be executed more quickly and effectively thanks to a portable and effective solution.
- Interprocess communication is necessary when tasks are split across different processes
- The standard message passing library used is MPI

What is a deadlock?

Deadlock occurs when one process is trying to acquire a resource from another process and that process is in turn trying to acquire a resource from another and so on, thus creating a cycle



Problem Statement



Parallel programs using message passing for communication have the possibilities for deadlocks and they can be difficult to debug as they are on a distributed memory environment

MPI_Check

- Preprocessing is required, inserting its own code into the source code
- Original calls to MPI Functions are modified by replacing parameters in the arguments
- Indirect hand shaking code, which is placed before each call to the send or receive routine and checks for matching sends and receives, is used to achieve detection.
- Programmer may have to keep 2 copies of the code, Original and handshaking
- C++ Language is not supported

Umpire

- Similar to MPIDD, it detects deadlocks dynamically
- Utilizes the MPI Profiling layer
- Runs as a separate process using shared memory to communicate between processes
- Only works in a shared memory process environment

6 Components to MPIDD

01 MPI Wrappers

02 Manager

03 State

04 Detection algorithm

05 Command Interpreters

06 Deadlock handlers

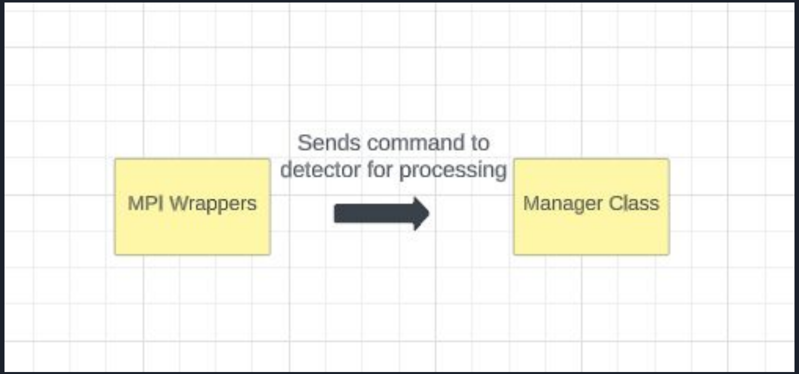
MPI Wrappers

- Allows programmers to redefine MPI routines using their profiling layer
- Sends an MPI Send to the Manager of the detector to process the command
- Example:

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Com
    // Your custom code before the actual MPI_Send call

    // Call the actual MPI_Send function
    int result = PMPI_Send(buf, count, datatype, dest, tag, comm);

    // Your custom code after the actual MPI_Send call
    // ...

    return result;
}
```

Manager

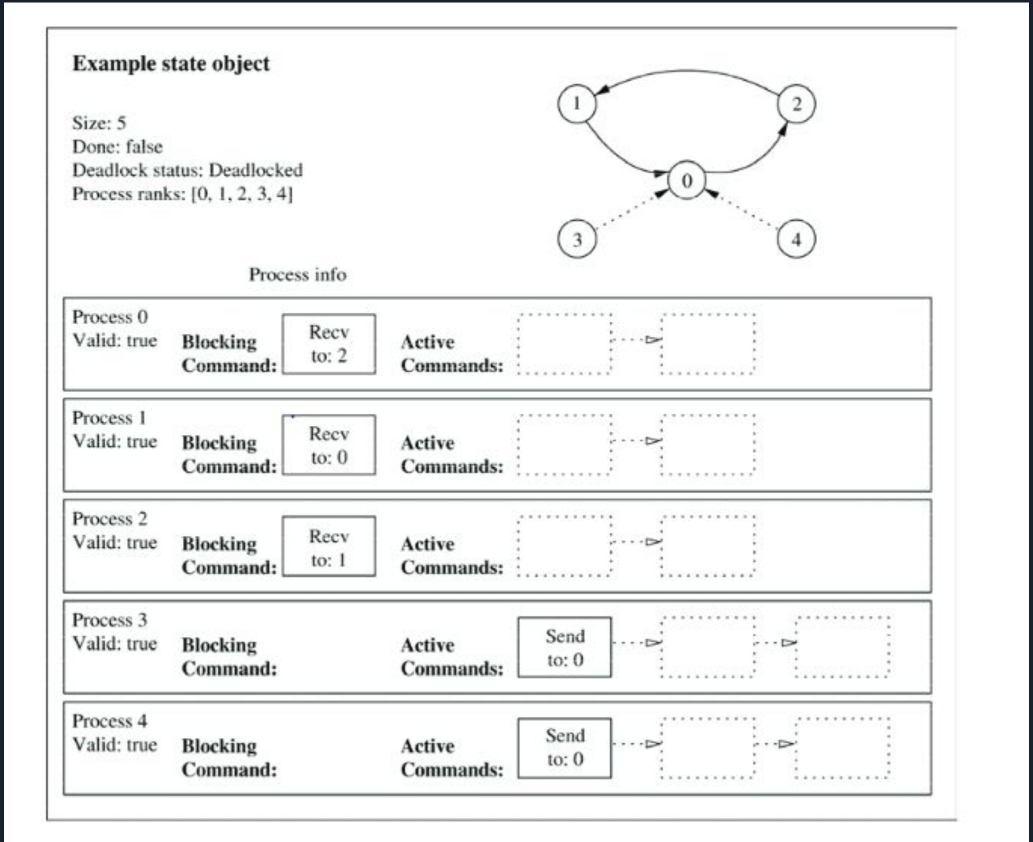
- run() method constantly listens for incoming commands from MPI Wrappers
- Holds the state as a member variable
- All processors must register with detector and de register
- Incoming command gets processed and set to command interpreter to update the state respectively



State

- Contains the deadlock status
- If clients have exited
- Array of process info objects that represents each client process
- Contains the Command class with the fields of :
    - From → Client that issued the command
    - To → Client who will receive the command
    - Type -> Type of MPI Command
    - Tag → Tag of command
    - Communicator → Communicator used by command

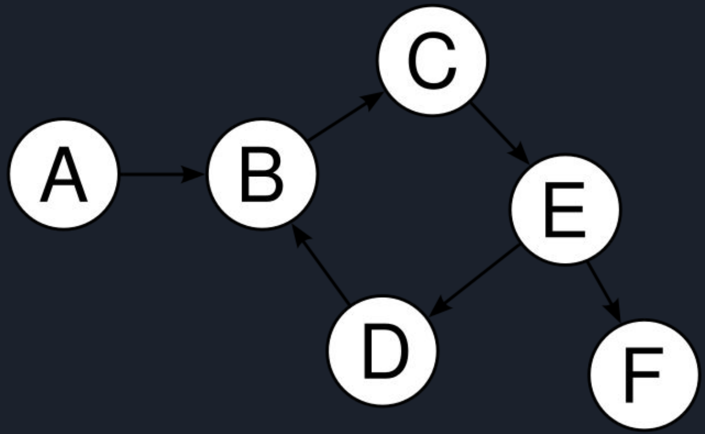
State



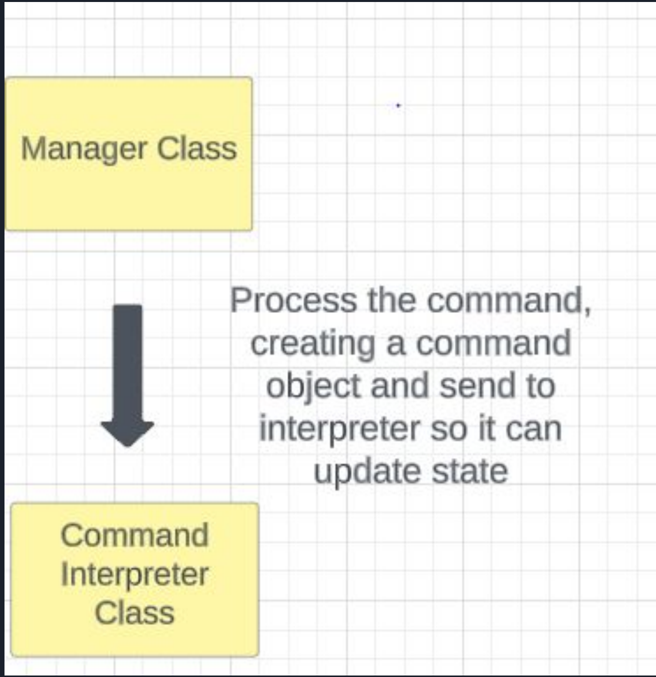
Example state object
Size: 5
Done: false
Deadlock status: Deadlocked
Process ranks: [0, 1, 2, 3, 4]
1
2
0
3
4
Process info
Process 0
Valid: true
Blocking Command:
Recv to: 2
Active Commands:
Process 1
Valid: true
Blocking Command:
Recv to: 0
Active Commands:
Process 2
Valid: true
Blocking Command:
Recv to: 1
Active Commands:
Process 3
Valid: true
Blocking Command:
Active Commands:
Send to: 0
Process 4
Valid: true
Blocking Command:
Active Commands:
Send to: 0

Detection Algorithm

- Recursive algorithm that runs Depth First Search starting from the source
- The algorithm will start from the source and search through dependencies and return true if a cycle is found, false if cycle is not found
- We only need to check blocking commands



Command Interpreters



- Handles updating the state and the potential case of calling the detection algorithm
- In manager, we call the respective interpreter based on the MPI Command routine, as each of them updates the state in a different way

Deadlock Handlers

- Decides what to do if a deadlock is found, will be signaled by the algorithm.
- User can specify one of 3 options:
  - Complain
  - Stop
  - Debug

Concurrency/Optimization

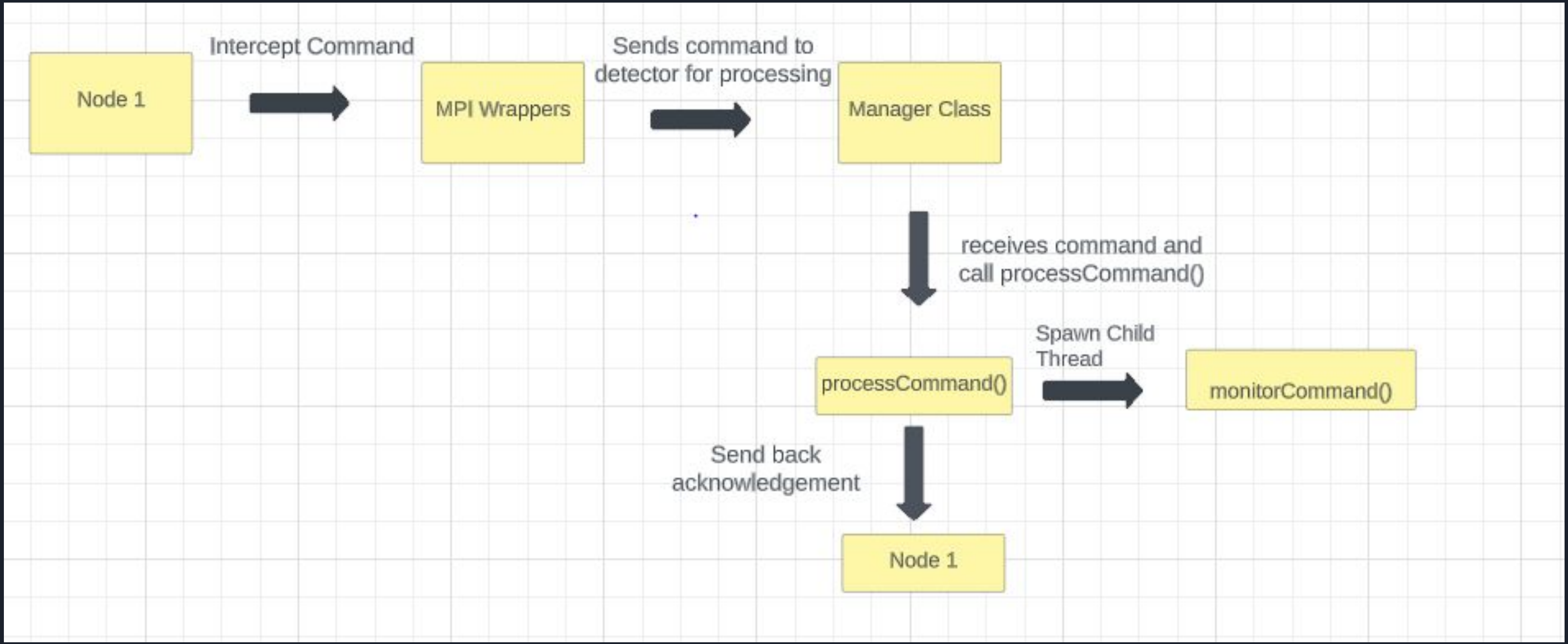
Problem:

- Inefficient if the algorithm is invoked every time an MPI Command is issued

Solution:

- In order to allow a command to "sit" for a predetermined amount of time or to quit early if the command unblocks before then, the manager establishes a child thread for each blocking command issued from the wrappers. This allows the thread to identify deadlocks that may arise in the time frame after a command given.
- If the thread times out before the command unblocks , only then we run the detection algorithm

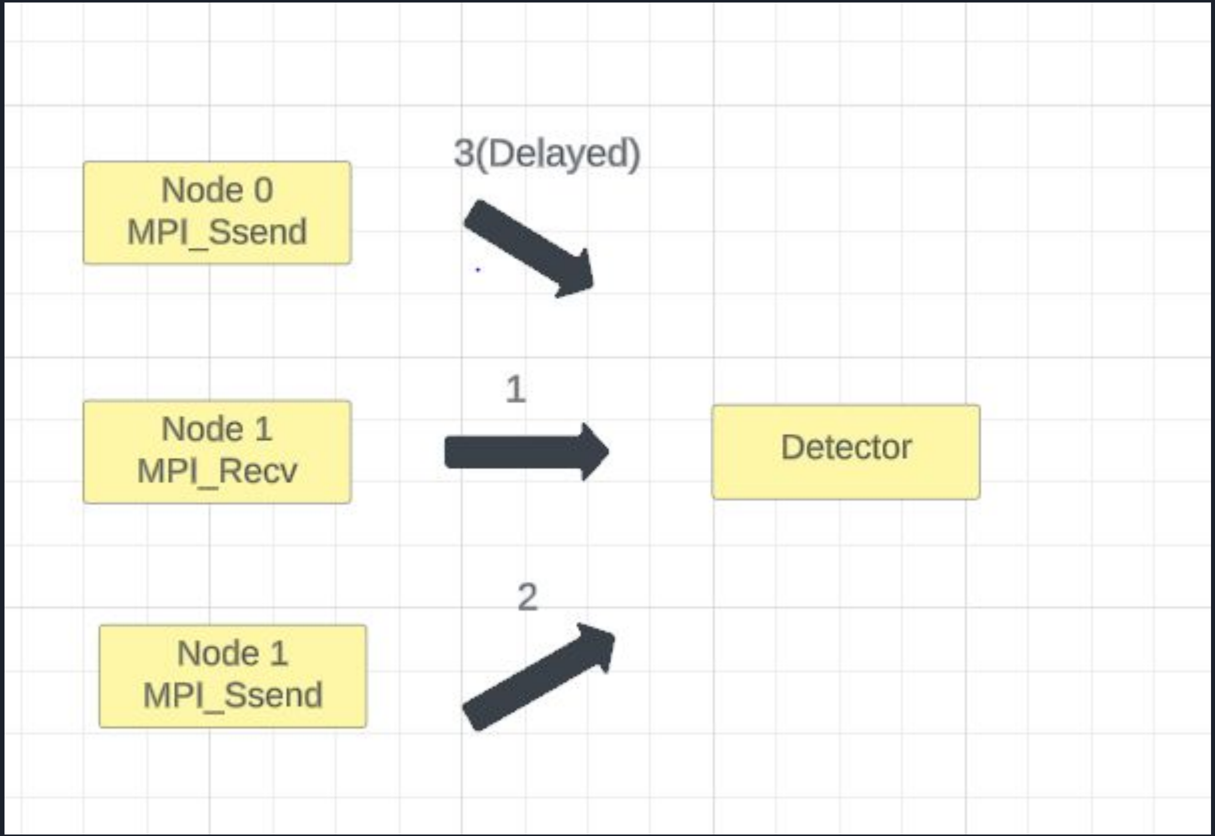
Optimization



Synchronization Problem

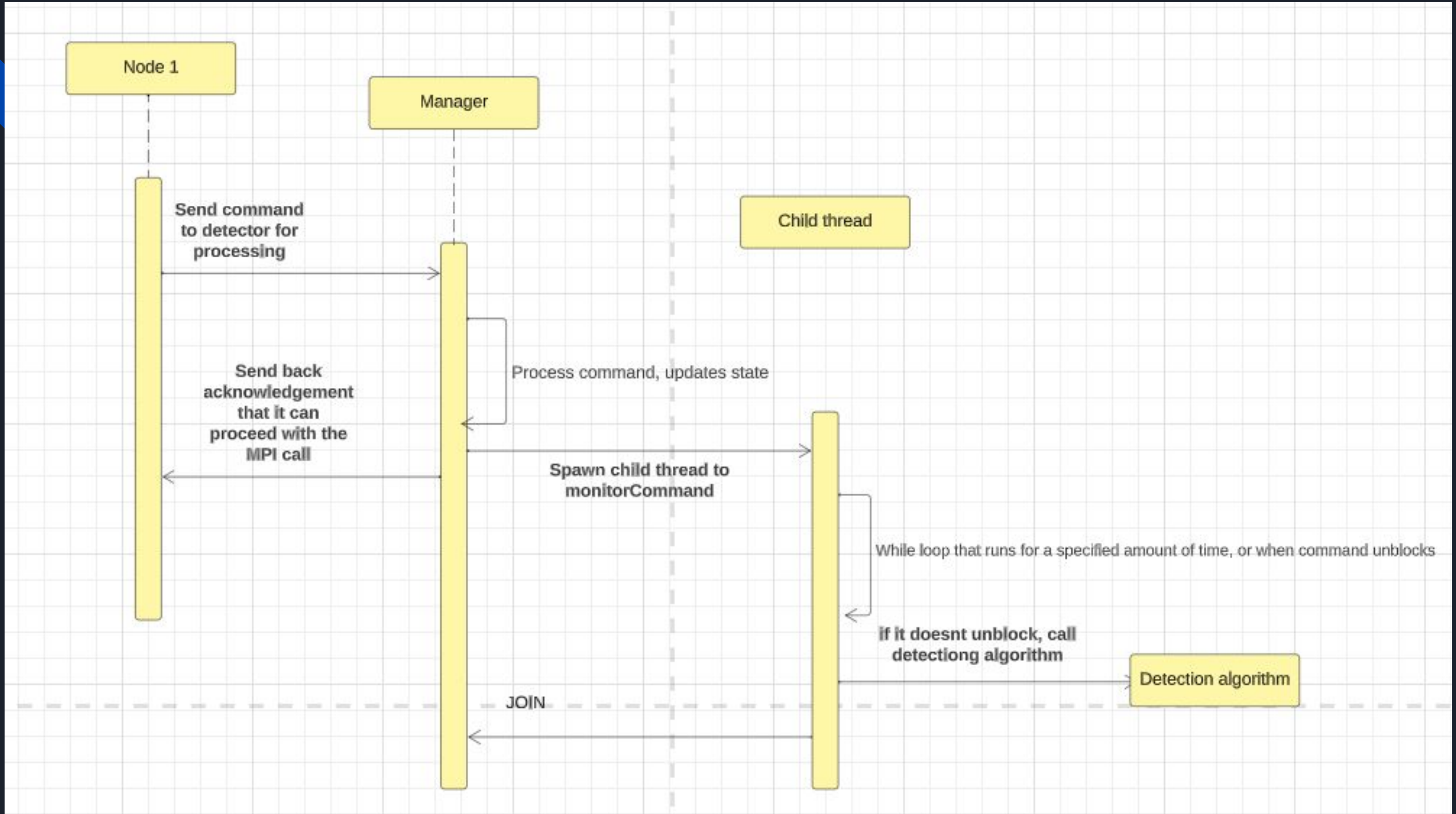
- False deadlocks might be detected from delayed arrival of messages to detector

```
if(currentNode == 0)
    MPI_Ssend (message, 100, MPI_CHAR, 1, tag,
      MPI_COMM_WORLD);
    MPI_Recv (message, 100, MPI_CHAR, 1, tag,
      MPI_COMM_WORLD, status);
else if(currentNode == 1)
    MPI_Recv (message, 100, MPI_CHAR, 0, tag,
      MPI_COMM_WORLD, status);
    MPI_Ssend (message, 100, MPI_CHAR, 0, tag,
      MPI_COMM_WORLD);
```


Node 0
MPI_Ssend
3(Delayed)
Node 1
MPI_Recv
1
Detector
Node 1
MPI_Ssend
2

Synchronization Problem Solution

- The original command must first wait for the detector's acknowledgement before continuing with the MPI function.
- Acknowledgement is only issued once the command is processed and the state is updated, ensuring the order stays consistent.


Node 1
Manager
Child thread
Send command to detector for processing
Process command, updates state
Send back acknowledgement that it can proceed with the MPI call
Spawn child thread to monitorCommand
While loop that runs for a specified amount of time, or when command unblocks
if it doesnt unblock, call detectiong algorithm
Detection algorithm
JOIN

Testing

First phase

- Test suite contains a variety of cases ranging from simple deadlock between a pair of processes to more complex ones involving a subset of processes or multiple cycles
- Tests are generated using script rules and desired complexity

Second phase

- Testing on publicly available application codes, such as:
    - NAS Parallel Benchmark
    - SKaMPI benchmark
- Modified code to introduce deadlocks

Analysis of Results

- Low overhead, when Complain or Stop option is implemented without the monitor.
- Detector introduces two types of overhead:

  Constant:

  - Initializing the detector and all the components
  - Processors registering and de-registering with the detector

  MPI Calls overhead:

  - Becomes insignificant when processing times are large at each node
- NAS Parallel Benchmark suite displays only 1% overhead when run with a 5 node cluster
- Matrix Multiplication code for 500x500 matrices produced 10.78% overhead in the same environment

Conclusion

- Most debugging is done manually using interactive techniques which are time consuming, error prone, and complicated, most common being print statements
- MPIDD is a good solution to allow users for dynamically detecting and resolving deadlocks
- Portable, low overhead are key advantages of MPIDD
- No modification to original source code necessary as it is ran on another process
- Not restricted to an SMP environment

References

1. W.Hque(2015). Concurrent Deadlock Detection in Parallel Programs. International Journal of Computers and Applications
2. G. Luecke, Y. Zou, J. Coyle, J. Hoekstra, & M. Kraeva,Deadlock detection in MPI programs, Concurrency and Com-putation: Practice and Experience,14(11), 2002, 911–932.
3. J.S. Vetter & B.R. de Supinski, Dynamic software testing ofMPI applications with Umpire,Proc. Supercomputing 2000,Dallas, TX, 2000.