Advanced MPI and Debugging

Parallel Programming

2024-12-16

Prof. Dr. Michael Kuhn

michael.kuhn@ovgu.de

Parallel Computing and I/O
Institute for Intelligent Cooperating Systems
Faculty of Computer Science
Otto von Guericke University Magdeburg
https://parcio.ovgu.de

Outline

- Which functionality is not used for high-speed networking?
    1. Remote direct memory access
    2. Zero copy
    3. Vectorization
    4. Kernel bypass

- Which technology improves at the fastest rate?
  1. Storage capacity
  2. Storage throughput
  3. Network throughput
  4. Memory throughput
  5. Computation

- When does Amdahl's Law apply?
  1. Fixed problem size
  2. Fixed runtime
  3. Serial portion is smaller than 10 %
  4. Multiple program, multiple data streams (MPMD)

- Which scaling behavior is preferable?
    1. Weak scaling
    2. Strong scaling
    3. Both are equally good

- What is strong scaling?
    1. Increase problem size with task count
    2. Increase task count with constant problem size
    3. Increase runtime with constant task count
    4. Decrease problem size with constant task count

Outline

Advanced MPI and Debugging

- Review
- Introduction
- One-Sided Communication
- Profiling Interface
- Debugging
- Summary

Motivation

- MPI supports basic and complex operations
  - Point-to-point and collective communication
  - Groups, communicators and topologies
  - Environment checks
  - Parallel I/O
- Advanced functionality
  - Dynamic process management
  - Non-blocking collectives
  - Profiling interface
  - One-sided communication

Motivation...

- One-sided communication enables more efficient interaction
  - Optimizations like RDMA, zero copy etc. can be utilized easily
  - One-sided communication is similar to shared memory programming
- Profiling interface gives insight into internals
  - Can be used for performance measurements, debugging etc.
  - Frameworks can hook into the profiling interface (for example, Score-P)

Motivation...

- Dedicated debugging support for parallel applications
  - Deadlocks or race conditions can be hard to find and correct
- Sophisticated optimizations can lead to hard-to-debug problems
  - Parallelization introduces deadlocks and race conditions
  - Traditional languages do not have means to detect problems
- New languages with native support for parallelism
  - Rust can detect data races at compile time due to its ownership concept

Outline

Overview

- One-sided communication provides remote memory access (RMA)
  - Can be handled efficiently by appropriate hardware
  - Both Ethernet and InfiniBand support native RDMA
- Point-to-point requires knowledge on both sides
  - For some applications or communication schemes, this might be difficult
  - Only the process doing the accesses might know what data to put where
- Theoretically offers better performance than other communication schemes
  - Other side can continue performing computation during communication

- Functions for basic operations
  - Write: MPI_Put and MPI_Rput
  - Read: MPI_Get and MPI_Rget
- More complex functionality is also available
  - Update: MPI_Accumulate and MPI_Raccumulate
  - Read and update: MPI_Get_accumulate, MPI_Rget_accumulate and MPI_Fetch_and_op
  - Atomic swap: MPI_Compare_and_swap
- Blocking or request-based variants
  - R stands for request-based and behaves like non-blocking
  - Request-based calls have to be finished with MPI_Wait etc.

Windows

- One-sided communication still does not allow access to whole address space
  - In contrast to shared memory, where everything is shared by default
- Memory regions have to be exposed via windows
  - Enables access to specified memory regions within a process
- Two main types of windows
  1. Allocated windows (includes backing memory)
     - Either local or shared memory
  2. Created windows (requires existing backing memory)
     - Either static or dynamic windows

- MPI_Win_create
    - Base: Memory address
    - Size: Memory size
    - Displacement unit: Element size
    - Info: Implementation hints
    - Communicator: Process mapping
    - Window: Exposed memory

```c
void window_create(void) {
    MPI_Win win;
    char str[100];
    snprintf(str, 100,
        "Hello from %d\n", rank);

    MPI_Win_create(str,
        sizeof(str), 1,
        MPI_INFO_NULL,
        MPI_COMM_WORLD,
        &win);
    MPI_Win_free(&win);
}
```

- MPI_Win_allocate
  - Size: Memory size
  - Displacement unit: Element size
  - Info: Implementation hints
  - Communicator: Process mapping
  - Base: New memory address
  - Window: Exposed memory

```c
void window_allocate(void) {
    MPI_Win win;
    char* str;

    MPI_Win_allocate(100, 1,
        MPI_INFO_NULL,
        MPI_COMM_WORLD,
        &str, &win);
    snprintf(str, 100,
        "Hello from %d\n", rank);
    MPI_Win_free(&win);
}
```

Memory Models

- MPI differentiates between public and private memory
  - Public: Exposed main memory, addressable by all processes
  - Private: Caches etc. that are only addressable by the local process
- There are two memory models based on public and private memory
  1. Separate: No assumptions about memory consistency, portable (non-coherent)
     - Changes to public require RMA calls to synchronize to private memory
  2. Unified: Updates to public memory are synchronized to private memory (coherent)
     - Public and private memory are always identical and require no synchronization
     - Without synchronization, data might still be inconsistent while in progress

- MPI_Win_get_attr
  - Window: Exposed memory
  - Key: Attribute to query
  - Value: Pointer to store value in
  - Flag: Whether attribute could be queried
- Create flavor
  - Find out how window was allocated
- Memory model
  - Get information about memory model

```c
void print_win(MPI_Win win) {
    int* val;
    int flag;

    MPI_Win_get_attr(win,
        MPI_WIN_CREATE_FLAVOR,
        &val, &flag);
    print_flavor(*val);

    MPI_Win_get_attr(win,
        MPI_WIN_MODEL,
        &val, &flag);
    print_model(*val);
}
```

Memory Models...

- `MPI_Win_get_attr`
  - Window: Exposed memory
  - Key: Attribute to query
  - Value: Pointer to store value in
  - Flag: Whether attribute could be queried
- Create flavor
  - Find out how window was allocated
- Memory model
  - Get information about memory model

```
flavor=create
model=unified
flavor=allocate
model=unified
```

Communication Models

- MPI clearly defines processes involved in RMA communication
  - Origin: Process that performs a call
  - Target: Process that is accessed by a call
- Might lead to unintuitive situations
  - Putting data into another process's memory
    - Source of the data is the origin
    - Destination for the data is the target
  - Getting data from another process's memory
    - Source of the data is the target
    - Destination for the data is the origin

Communication Models...

- MPI supports two modes for one-sided communication
  1. Active target communication
  2. Passive target communication
- Active target communication
  - Both origin and target are involved in the communication
  - Similar to message passing where both sides are involved
  - All arguments provided by one process, the other just participates in synchronization
- Passive target communication
  - Only origin process is involved in communication
  - Close to shared memory programming where other threads are not influenced

Epochs

- Communication calls must happen inside an access epoch
  - Epoch starts with a synchronization call on window
  - Followed by arbitrarily many communication calls
  - Epoch completes with another synchronization call
- Active target communication also has exposure epochs
  - Epoch starts with a synchronization call by target process
  - One-to-one matching of access and exposure epochs
- Passive target communication does not have synchronization on target
  - There also is no exposure epoch

Synchronization

- Two synchronization mechanisms for active target communication
  - MPI_Win_fence is a collective synchronization call
    - Starts access and exposure epochs
  - MPI_Win_start, MPI_Win_complete, MPI_Win_post and MPI_Win_wait are fine-grained
    - Only communicating processes synchronize
    - MPI_Win_start and MPI_Win_complete start and stop access epochs
    - MPI_Win_post and MPI_Win_wait start and stop exposure epochs
- One synchronization mechanism for passive target communication
  - MPI_Win_lock, MPI_Win_lock_all, MPI_Win_unlock and MPI_Win_unlock_all

Put

- Every process exposes a window
  - Other processes can write into it and read from it
  - Access is only possible via window
- Put local string into remote memory
  - str should be copied into window

```c
char str[100];
char buf[100];

MPI_Win win;

void window(void) {
    snprintf(str, 100,
        "Hello from %d\n", rank);
    MPI_Win_create(buf,
        sizeof(buf), 1,
        MPI_INFO_NULL,
        MPI_COMM_WORLD, &win);
}
```

Put...

- Passive target communication
  - Lock and unlock necessary
  - Put will be finished after unlock
- MPI_Win_lock
  - Type: Exclusive or shared
  - Rank: Target rank
  - Assert: Optimization hints
  - Window: Exposed memory
- MPI_Win_unlock
  - Rank: Target rank
  - Window: Exposed memory

```c
void put(void) {
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE,
        (rank + 1) % size,
        MPI_MODE_NOCHECK, win);
    MPI_Put(str, 100, MPI_CHAR,
        (rank + 1) % size, 0,
        100, MPI_CHAR, win);
    MPI_Win_unlock(
        (rank + 1) % size, win);

    MPI_Barrier(MPI_COMM_WORLD);
    printf("%d: %s", rank, buf);
}
```

Put...

- MPI_Put
  - Origin buffer: Data to put
  - Origin count: Number of elements
  - Origin datatype: Type of elements
  - Target rank: Where to put data
  - Target displacement: Offset in window
  - Target count: Number of elements
  - Target datatype: Type of elements
  - Window: Exposed memory

```c
void put(void) {
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE,
        (rank + 1) % size,
        MPI_MODE_NOCHECK, win);
    MPI_Put(str, 100, MPI_CHAR,
        (rank + 1) % size, 0,
        100, MPI_CHAR, win);
    MPI_Win_unlock(
        (rank + 1) % size, win);

    MPI_Barrier(MPI_COMM_WORLD);
    printf("%d: %s", rank, buf);
}
```

Put. . .

- Ring communication
  - Each process copies string into next process's memory
- Target is not involved
  - Origin locks remote window
  - Afterwards, data is put there

```
$ mpiexec -n 4 ./put
0: Hello from 3
1: Hello from 0
2: Hello from 1
3: Hello from 2
```

- What happens without `MPI_Barrier`?
  1. The same as with the barrier
  2. buf can be empty
  3. Processes crash
  4. Processes deadlock

```c
void put(void) {
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE,
        (rank + 1) % size,
        MPI_MODE_NOCHECK, win);
    MPI_Put(str, 100, MPI_CHAR,
        (rank + 1) % size, 0,
        100, MPI_CHAR, win);
    MPI_Win_unlock(
        (rank + 1) % size, win);

    MPI_Barrier(MPI_COMM_WORLD);
    printf("%d: %s", rank, buf);
}
```

- Schema is inverted with get
  - Every process exposes their string
  - Other processes can write into it and read from it
- Get remote string into local memory
  - Window should be copied into buf

```c
char str[100];
char buf[100];

MPI_Win win;

void window(void) {
    snprintf(str, 100,
        "Hello from %d\n", rank);
    MPI_Win_create(str,
        sizeof(str), 1,
        MPI_INFO_NULL,
        MPI_COMM_WORLD, &win);
}
```

Get…

- MPI_Get
  - Origin buffer: Where to get data
  - Origin count: Number of elements
  - Origin datatype: Type of elements
  - Target rank: From where to get data
  - Target displacement: Offset in window
  - Target count: Number of elements
  - Target datatype: Type of elements
  - Window: Exposed memory

```c
void put(void) {
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE,
        (size + rank - 1) % size,
        MPI_MODE_NOCHECK, win);
    MPI_Get(buf, 100, MPI_CHAR,
        (size + rank - 1) % size, 0,
        100, MPI_CHAR, win);
    MPI_Win_unlock(
        (size + rank - 1) % size,
        win);

    printf("%d: %s", rank, buf);
}
```

Get…

- Ring communication
  - Each process copies string from previous process's memory
- Target is not involved again
  - Origin locks remote window
  - Afterwards, get operation is performed

```
$ mpiexec -n 4 ./get
0: Hello from 3
1: Hello from 0
2: Hello from 1
3: Hello from 2
```

- Why is no `MPI_Barrier` used?
  1. It is a bug, barrier is required
  2. Implicit synchronization
  3. Window is small enough

```c
void put(void) {
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE,
        (size + rank - 1) % size,
        MPI_MODE_NOCHECK, win);
    MPI_Get(buf, 100, MPI_CHAR,
        (size + rank - 1) % size, 0,
        100, MPI_CHAR, win);
    MPI_Win_unlock(
        (size + rank - 1) % size,
        win);

    printf("%d: %s", rank, buf);
}
```

Accumulate

- MPI supports accumulate operations
  - Similar to reduce operations in collective communication
- Collect maximum rank across all processes
  - Works like MPI_Reduce with MPI_MAX

```c
int buf = 0;

MPI_Win win;

void window(void) {
    MPI_Win_create(&buf,
        sizeof(buf), 1,
        MPI_INFO_NULL,
        MPI_COMM_WORLD, &win);
}
```

Accumulate…

- MPI_Accumulate
  - Origin buffer: Data to accumulate
  - Origin count: Number of elements
  - Origin datatype: Type of elements
  - Target rank: Where to accumulate data
  - Target displacement: Offset in window
  - Target count: Number of elements
  - Target datatype: Type of elements
  - Op: Operation to perform
  - Window: Exposed memory

```c
void put(void) {
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE,
        0, 0, win);
    MPI_Accumulate(&rank, 1,
        MPI_INT, 0, 0, 1,
        MPI_INT, MPI_MAX, win);
    MPI_Win_unlock(0, win);

    MPI_Barrier(MPI_COMM_WORLD);

    printf("%d: %d\n", rank, buf);
}
```

Accumulate...

- Maximum is accumulated on rank 0
  - All other processes keep original value
- Accumulated value has to be distributed
  - For instance, using `MPI_Broadcast`

```
$ mpiexec -n 4 ./accumulate
0: 3
1: 0
2: 0
3: 0
```

Outline

Motivation

- Profiling interface allows debugging and performance analysis
  - Function calls can be intercepted and recorded
- Many different MPI implementations exist
  - Source code for a specific implementation may not be available
  - Some are proprietary and cannot be inspected
- Realized via a second set of function names
  - Functions are prefixed with `PMPI_` instead of `MPI_`

Motivation...

- Can also be used for other purposes
  - For instance, choose different functions from different implementations
- `MPI_Pcontrol` must be provided by implementations
  - Enable or disable profiling, flush buffers etc.
  - Default implementation does nothing
- Implementation
  - Weak symbols: Compiler takes care of symbols
  - Otherwise: Link in correct order (`-lmylib -lpmpi -lmpi`)

Overriding Functions

- Override functions with own definition
  - Compiler calls own definition
  - Weak symbols allow overriding
- Implementation available via PMPI_
  - Easy to cause infinite recursions

```
int MPI_Send(const void* buf,
             int count,
             MPI_Datatype datatype,
             int dest, int tag,
             MPI_Comm comm) {
    printf("MPI_Send: buf=%p,"
        " count=%d, datatype=%d,"
        " dest=%d, tag=%d,"
        " comm=%d\n", buf, count,
        datatype, dest, tag, comm);
    return PMPI_Send(buf, count,
        datatype, dest, tag, comm);
}
```

Overriding Functions

- Override functions with own definition
  - Compiler calls own definition
  - Weak symbols allow overriding
- Implementation available via `PMPI_`
  - Easy to cause infinite recursions

```c
int MPI_Recv(void* buf, int count,
             MPI_Datatype datatype,
             int source, int tag,
             MPI_Comm comm,
             MPI_Status* status) {
    printf("MPI_Recv: buf=%p,"
        " count=%d, datatype=%d,"
        " source=%d, tag=%d,"
        " comm=%d, status=%p\n",
        buf, count, datatype,
        source, tag, comm,
        (void*)status);
    return PMPI_Recv(buf, count,
        datatype, source, tag,
        comm, status);
}
```

Overriding Functions

- Override functions with own definition
  - Compiler calls own definition
  - Weak symbols allow overriding
- Implementation available via `PMPI_`
  - Easy to cause infinite recursions
- Easy to log all parameters
  - Frameworks like Score-P use this
  - Can be visualized with Vampir etc.

```
$ mpiexec -n 2 ./profiling
MPI_Send: [...], count=100, [...],
    ↪ dest=1, tag=0, [...]
MPI_Recv: [...], count=100, [...],
    ↪ source=1, tag=0, [...]
0: Hello from 1
MPI_Send: [...], count=100, [...],
    ↪ dest=0, tag=0, [...]
MPI_Recv: [...], count=100, [...],
    ↪ source=0, tag=0, [...]
1: Hello from 0
```

Outline

Error Classes

- Example: Race condition
  - Incrementing consists of three steps
    1. Loading the variable
    2. Modifying the variable
    3. Storing the variable
  - Have to be performed atomically

```c
static int counter = 0;

void* thread_func(void* data) {
    (void)data;

    for (int i = 0; i < 1000; i++) {
        counter++;
    }

    return NULL;
}
```

Error Classes

- Example: Race condition
  - Incrementing consists of three steps
    1. Loading the variable
    2. Modifying the variable
    3. Storing the variable
  - Have to be performed atomically

T0	T1	V
Load 0		0
Inc 1		0
Store 1		1
	Load 1	1
	Inc 2	1
	Store 2	2

Error Classes

- Example: Race condition
  - Incrementing consists of three steps
    1. Loading the variable
    2. Modifying the variable
    3. Storing the variable
  - Have to be performed atomically

T0	T1	V
Load 0		0
Inc 1		0
Store 1		1
	Load 1	1
	Inc 2	1
	Store 2	2

T0	T1	V
Load 0		0
Inc 1	Load 0	0
Store 1	Inc 1	1
	Store 1	1

Error Classes

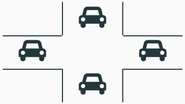
- Example: Race condition
  - Incrementing consists of three steps
    1. Loading the variable
    2. Modifying the variable
    3. Storing the variable
  - Have to be performed atomically
- Two new error classes
  1. Deadlocks
  2. Race conditions

T0	T1	V
Load 0		0
Inc 1		0
Store 1		1
	Load 1	1
	Inc 2	1
	Store 2	2

T0	T1	V
Load 0		0
Inc 1	Load 0	0
Store 1	Inc 1	1
	Store 1	1

Error Classes…

- Deadlocks cause parallel applications to stop progressing
  - Can have different causes, most often due to locking
  - May not be reproducible if there is time-dependent behavior
- Error condition can be difficult to find
  - Trying to lock an already acquired lock results in a deadlock
  - Erroneous communication patterns (everyone waits for the right neighbor)
- Error effect is typically easy to spot
  - Spinlocks or livelocks can look like computation, though



Error Classes...

- Race conditions can lead to differing results
  - Debugging often hides race conditions
- Error condition is often very hard to find
  - Can be observed at runtime or be found by static analysis
  - Modern programming languages like Rust can detect data races
- Error effect is sometimes not observable
  - Slight variations in the results are not obvious
  - The correct result cannot be determined for complex applications
  - Repeating a calculation can be too costly

- Access to counter is not synchronized
  - Race condition results in wrong value
- Output is non-deterministic
  - Depends on timing, scheduling etc.
  - Output is sometimes correct
- Compiler cannot help
  - Developer has to spot error manually

```c
static int counter = 0;

void* thread_func(void* data) {
    (void)data;

    for (int i = 0; i < 1000; i++) {
        counter++;
    }

    return NULL;
}
```

Race Conditions

- Access to counter is not synchronized
  - Race condition results in wrong value
- Output is non-deterministic
  - Depends on timing, scheduling etc.
  - Output is sometimes correct
- Compiler cannot help
  - Developer has to spot error manually

```
$ ./race
counter=10000
$ ./race
counter=9753
$ ./race
counter=10000
$ ./race
counter=10000
$ ./race
counter=9244
```

- Helgrind is part of Valgrind
    - Detects synchronization errors in C, C++ and Fortran
    - Supports POSIX threads
    - Also works with OpenMP but output can be confusing
- Supports three classes of errors
    1. Misuse of POSIX threads API
    2. Lock ordering problems
    3. Data races
- Helgrind analyzes memory access
    - Happens-before dependency graph

```
$ valgrind --tool=helgrind ./race
Helgrind, a thread error detector
[...]
Possible data race during read of
    ↪ size 4 at 0x404038 by thread #3
Locks held: none
    at 0x401157: [...]

This conflicts with a previous write
    ↪ of size 4 by thread #2
Locks held: none
    at 0x401160: [...]
    Address 0x404038 is 0 bytes
        ↪ inside data symbol
        ↪ "counter"
```

Race Conditions...

- Thread sanitizer can detect thread bugs
  - Data races
  - Races on mutexes, file descriptors, barriers etc.
  - Destroying locked mutexes
  - Signal-unsafe behavior
  - Potential deadlocks
  - ... and more
- Sanitizers are offered by the compiler
  - Can instrument code at compile time
  - Instruments memory access instructions

```
$ ./race-sanitize
==================
WARNING: ThreadSanitizer: data race
    Read of size 4 at 0x000000404068
        ↪ by thread T2:
    #0 [...]

    Previous write of size 4 at
        ↪ 0x000000404068 by thread T1:
    #0 [...]

    Location is global '<null>' at
        ↪ 0x000000000000
        ↪ (...+0x000000404068)
```

Deadlocks

- Mutex is locked but never unlocked
  - Application hangs immediately
  - No output is produced
- Reason can be hard to determine
  - Check stack traces with GDB
    - `thread apply all bt`
    - Unwieldy with many threads
  - Difficult to determine whether deadlocked or progressing

```c
static int counter = 0;
static pthread_mutex_t mutex =
    PTHREAD_MUTEX_INITIALIZER;

void* thread_func(void* data) {
    (void)data;

    for (int i = 0; i < 1000; i++) {
        pthread_mutex_lock(&mutex);
        counter++;
    }

    return NULL;
}
```

Deadlocks

- Mutex is locked but never unlocked
  - Application hangs immediately
  - No output is produced
- Reason can be hard to determine
  - Check stack traces with GDB
    - `thread apply all bt`
    - Unwieldy with many threads
  - Difficult to determine whether deadlocked or progressing
- Helgrind will show held locks

```
$ valgrind --tool=helgrind ./dead
Helgrind, a thread error detector
[...]
Thread #2: Exiting thread still
    ↪ holds 1 lock
   at 0x4877EA0: [...]
```

Deadlocks...

- Lock ordering might lead to deadlocks
  - Relevant if multiple locks are involved
  - Locking should occur in same order
- Example
  - Thread 0 locks `m[0]`
  - Thread 1 locks `m[1]`
  - Thread 0 tries to lock `m[1]`
  - Thread 1 tries to lock `m[0]`

```c
void* thread_func(void* data) {
    uint64_t id = (uint64_t)data;
    int j = id % 2;
    int k = (id + 1) % 2;

    for (int i = 0; i < 1000; i++) {
        pthread_mutex_lock(&m[j]);
        pthread_mutex_lock(&m[k]);
        counter++;
        pthread_mutex_unlock(&m[k]);
        pthread_mutex_unlock(&m[j]);
    }

    return NULL;
}
```

Deadlocks...

- Lock ordering might lead to deadlocks
  - Relevant if multiple locks are involved
  - Locking should occur in same order
- Example
  - Thread 0 locks `m[0]`
  - Thread 1 locks `m[1]`
  - Thread 0 tries to lock `m[1]`
  - Thread 1 tries to lock `m[0]`
- Helgrind can detect lock order problems

```
$ valgrind --tool=helgrind ./lock
Thread #3: lock order "0x4040A0
    ↪ before 0x4040C8" violated

Observed (incorrect) order is:
    ↪ acquisition of lock at
    ↪ 0x4040C8
 followed by a later acquisition of
     ↪ lock at 0x4040A0

Required order was established by
    ↪ acquisition of lock at
    ↪ 0x4040A0
 followed by a later acquisition of
     ↪ lock at 0x4040C8
```

Deadlocks…

- Lock ordering might lead to deadlocks
  - Relevant if multiple locks are involved
  - Locking should occur in same order
- Example
  - Thread 0 locks `m[0]`
  - Thread 1 locks `m[1]`
  - Thread 0 tries to lock `m[1]`
  - Thread 1 tries to lock `m[0]`
- Helgrind can detect lock order problems
- Thread sanitizer works as well

```
$ ./lock-sanitize
WARNING: ThreadSanitizer:
    ↪ lock-order-inversion
    ↪ (potential deadlock)
  Cycle in lock order graph: M9
      ↪ (0x0000004040c0) => M10
      ↪ (0x0000004040e8) => M9
  Mutex M10 acquired here while
      ↪ holding mutex M9 in thread
      ↪ T1:
    #0 [...]
  Mutex M9 acquired here while
      ↪ holding mutex M10 in
      ↪ thread T2:
    #0 [...]
```

MPI Debugging

- MPI problems are harder to debug
  - Application is distributed across several nodes
  - Application is split into many processes
- There are debuggers for parallel applications
  - Arm DDT (part of Arm Forge, formerly Allinea DDT)
  - TotalView
  - Eclipse Parallel Tools Platform (PTP)
- Another approach is static analysis
  - MPI-Checker can analyze MPI applications [Droste et al., 2015] [Alexander Droste, 2024]

- Non-blocking functions require waiting
  - Otherwise, it is not clear when buffer can be reused
  - `MPI_Wait` is missing
- Errors might be hard to observe
  - Works correctly most of the time
  - Behavior is timing-dependent and non-deterministic

```c
void mysend(char* str, char* buf) {
    MPI_Request req;

    MPI_Isend(str, 100000, MPI_CHAR,
        (rank + 1) % size,
        0, MPI_COMM_WORLD, &req);
    MPI_Recv(buf, 100000, MPI_CHAR,
        (size + rank - 1) % size,
        0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);

    printf("%d: %s", rank, buf);
}
```

MPI Debugging...

- Path-sensitive checks
  - Clang's static analyzer (LLVM 3.9)
  - Double non-blocking without wait
  - Missing wait for non-blocking operations
  - Waiting without non-blocking call
- Abstract syntax tree checks
  - Clang-Tidy (LLVM 4.0)
  - Type mismatches when communicating
  - Incorrect referencing of buffers

```
$ scan-build -enable-checker
    ↪ optin.mpi.MPI-Checker mpicc
    ↪ -std=c11 -Wall -Wextra
    ↪ -Wpedantic isend.c -o isend
isend.c:15:2: warning: Request
    ↪ 'req' has no matching wait.
    ↪ [optin.mpi.MPI-Checker]
    MPI_Recv(buf, 100000, MPI_CHAR,
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

Outline

Summary

- MPI has support for basic and complex operations
  - Point-to-point and collective communication involved multiple processes
  - One-sided communication only involves one process at best
- MPI's profiling interface allows instrumenting the implementation
  - Can be used for debugging and performance analysis
- Parallel debugging is more complicated than normal debugging
  - Race conditions and deadlocks can be timing-dependent and non-deterministic
  - MPI applications are distributed and therefore harder to handle

References

[Alexander Droste, 2024] Alexander Droste (2024). **MPI-Checker.** `https://github.com/0ax1/MPI-Checker`.

[Droste et al., 2015] Droste, A., Kuhn, M., and Ludwig, T. (2015). **MPI-checker: static analysis for MPI.** In Finkel, H., editor, *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015*, pages 3:1–3:10. ACM.

[Message Passing Interface Forum, 2015] Message Passing Interface Forum (2015). **MPI: A Message-Passing Interface Standard Version 3.1.** `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/mpi31-report.htm`.