Pilgrim

A Lossless MPI Tracing Tool

Chen Wang[1], Pavan Balaji[2] and Marc Snir[1]
1. University of Illinois at Urbana-Champaign
2. Argonne National Laboratory

2020/01/21

Motivation

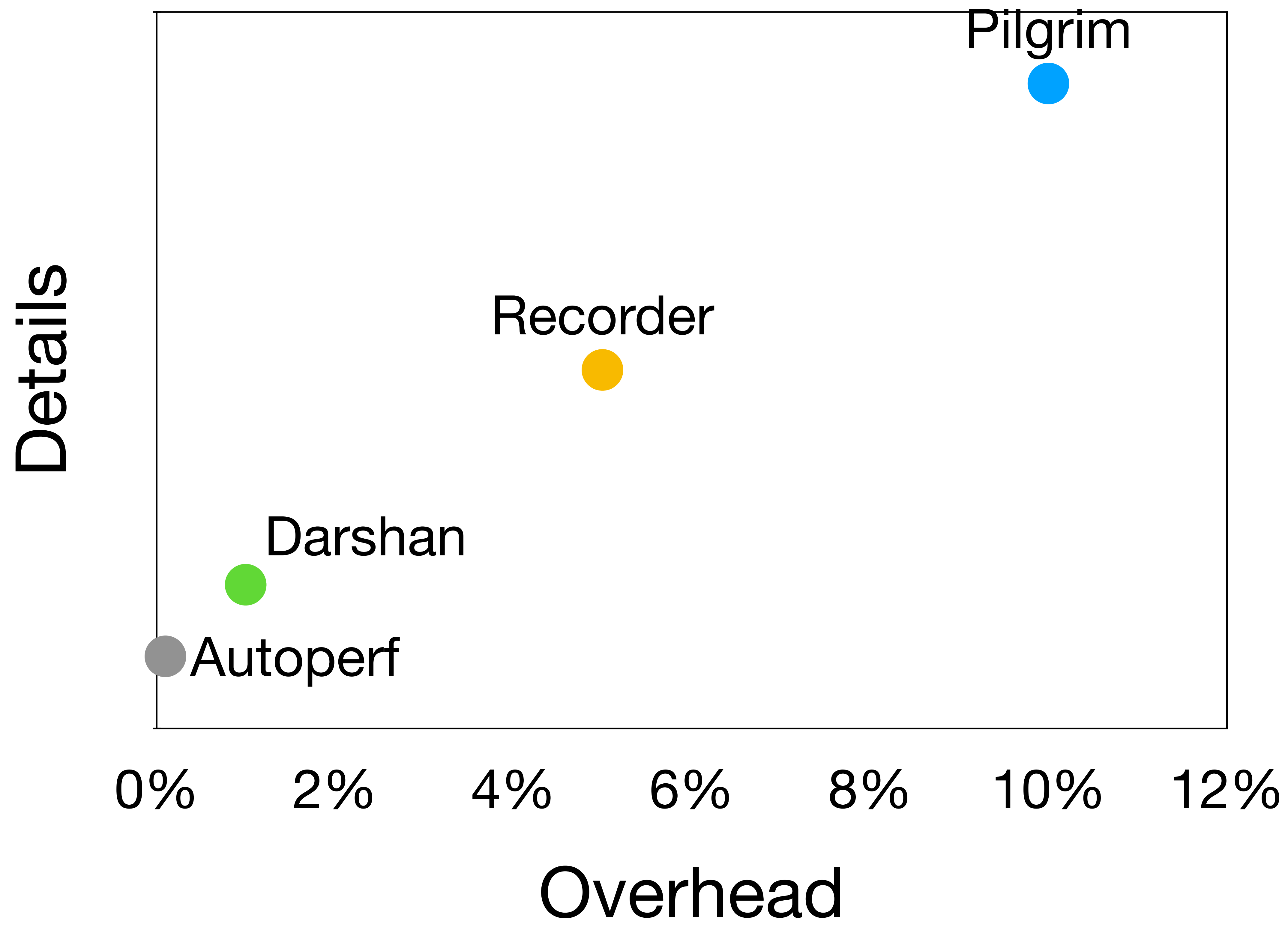
Why do we need MPI traces?

- MPI is a prominent programming model used for scientific computing.
    - Different applications use MPI differently.
    - Important to understand MPI usage for different applications
- For MPI and application users:
    - How is MPI used in my code? How frequent calls are? How are they ordered?
    - Am I providing the right hints to MPI for my usage?
    - Am I using MPI correctly?
- For MPI developers:
    - What MPI functions are used an in what way?
        - Message sizes, communicator sizes, buffer reuse, CPU/GPU buffers, are send/recv sizes the same or different, are collective operation datatype on all processes the same or different?

Motivation

Tradeoffs in lossy and lossless tracing

- Profiling tools, e.g., autoperf and Darshan store summarized (lossy) information about MPI calls.
- Many tracing tools are either incomplete or have unacceptable overhead at large scale (time or space).
  - Trade-off between details and overhead
- Our work (Pilgrim) is lossless with respect to the MPI calls
  - It adds acceptable overhead compared with other approaches.
  - Many encoding and compression techniques employed to achieve this goal



Motivation

Lossless MPI traces

- Compared to summarized information, e.g., function counters, accumulated execution time, etc., we keep **every parameter of every MPI call**.
  - How? Primarily relies on “recurring pattern recognition”
  - Most (but not all) applications have recurring patterns of communication -- we detect these patterns and store patterns to avoid repeatedly storing the same information
    - We use a context-free-grammar and a well-known algorithm called "Sequitur algorithm" for this — $O(N)$ parsing time, $O(N)$ — $O(log(N))$ storage (N is number of executed MPI calls)
    - The key is to detect as many patterns as possible

Motivation

Lossy non-MPI metadata

- Pilgrim is lossless for MPI functionality, but lossy for non-MPI metadata

1. The starting time and duration of function call are approximated to save space.
    - Useful for understanding skew between processes, depending on how much approximation
2. Actual communicated data is not saved.
3. Virtual addresses of memory buffers are summarized using symbolic representations

Motivation

How can we use lossless MPI traces?

- In-depth analysis is made possible
  - Understanding patterns of communication when multiple processes are involved (e.g., communication patterns of a stencil computation).
  - Understanding skew between processes during collective or P2P operations.
  - Understanding cases where applications use MPI sub-optimally and provide recommendations as to what they can do to improve.
    - E.g., MPI info hints, new/different MPI functionality, ...
- Generating automatically MPI mini apps from full applications (including from closed source or export controlled applications, e.g., from the NNSA labs).

Design and Implementation

Context Free Grammar and Sequitur algorithm

- A Context Free Grammar (CFG) contains a set of production rules in form of $A \rightarrow \alpha$
  - A is a nonterminal symbol, and α is a string of terminals and/or nonterminals.
  - For any nonterminal, there will be only one rule. i.e., the CFG can only generate one string.
  - There is particular starting nonterminal symbol S. By repeated rule applications from S, we can get the original uncompressed string.

S → a A A B B
A → a b
B → c d

Repeated Rule Application →

"a a b a b c d c d"

Context Free Grammar and Sequitur algorithm

- We use a well known algorithm called “Sequitur” algorithm to build a CFG that encodes a string on-the-fly.
  - Sequitur algorithm is an incremental algorithm that can append one terminal symbol at time.
  - Sequitur algorithm has O(N) time complexity.

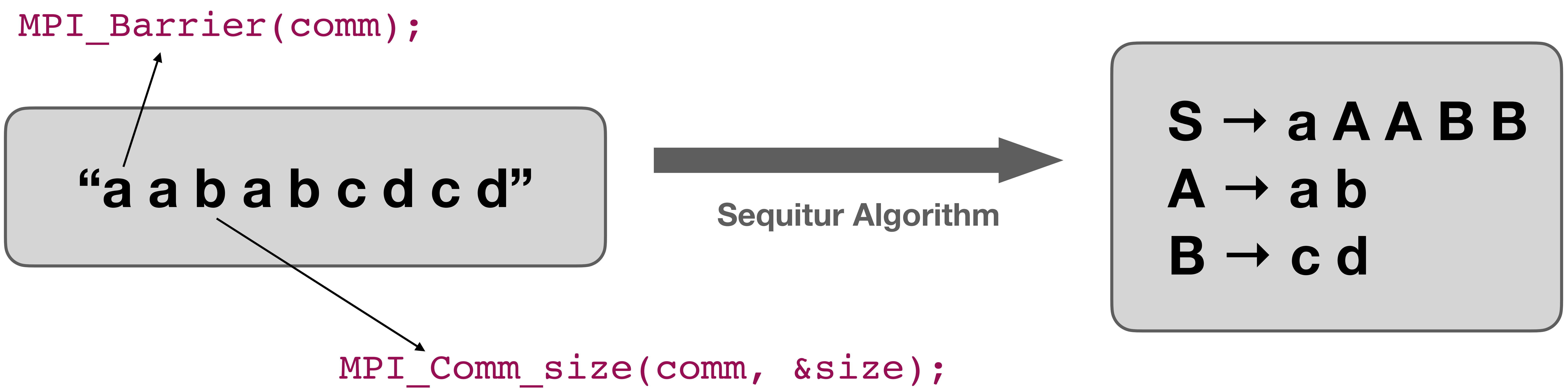
“a a b a b c d c d”

Sequitur Algorithm →

S → a A A B B
A → a b
B → c d

CFG for a string of MPI calls

- Each terminal symbol in the grammar represents a **unique call signature**.
- Call signature: function name and function parameter values
- A program execution produces a **string of terminal symbols**.



Call Signature ↔ Terminal Symbol

Call Signature (key-value) Table (CST)

- Call signature as key, terminal symbol as value.
- Identical calls have same terminal symbol.
- Some function parameters (e.g., pointers) are replaced by a symbolic representation.
- Entry/Exit times are not included in the call signature. They are kept separately.

Call Signature	Terminal Symbol
`MPI_Comm_size(comm1, 2)`	1
`MPI_Comm_rank(comm1, 0)`	2
`MPI_Comm_send(buf, 1, MPI_INT, 2, 100, comm1)`	3
`MPI_Barrier(comm1)`	4
`MPI_Barrier(comm2)`	5

Workflow of Pilgrim

1. Intercept every MPI call
2. Store ***entry/exit time***
3. Encode parameters and compose the ***call signature***
4. Map the ***call signature*** to a ***terminal symbol***
5. Use Sequitur algorithm to grow the CFG
6. Inter-process compression at the finalize point

Workflow of Pilgrim

1. **Intercept every MPI call**
2. Store ***entry/exit time***
3. Encode parameters and compose the ***call signature***
4. Map the ***call signature*** to a ***terminal symbol***
5. Use Sequitur algorithm to grow the CFG
6. Inter-process compression at the finalize point

Intercepting MPI calls

- Wrappers for intercepting the calls are generated automatically based on MPI document (Latex files).
- Trace generation MACROS executed before and after every call.
  - Generate a trace record for each executed MPI call that is is passed to the compression code
    - Number of parameters
    - Parameter types
    - In/out/inout

Workflow of Pilgrim

1. Intercept every MPI call
2. **Store *entry/exit time***
3. Encode parameters and compose the ***call signature***
4. Map the ***call signature*** to a ***terminal symbol***
5. Use Sequitur algorithm to grow the CFG
6. Inter-process compression at the finalize point

Function entry/exit time

- We keep **interval** and **duration** instead.
  - Smaller values and easier to compress
  - Can be used to compute entry/exit time.
- **Interval** is the elapsed time between the current call and the previous call who has the same call signature.
  - Ideally, there should only be a few unique intervals **per call signature**, e.g., one per loop.
- **Duration** is the time spent on the call.
  - Same function calls should have similar durations.
  - Variances exist due to network conditions, resource utilizations, irregularity in code execution, etc.

Function entry/exit time

- Both interval and duration are approximated using exponential bins.
  - ***Interval (or duration) = b^x***
  - ***x*** values are binned in equally spaced segments
  - Relative error is bounded
  - The ***base b*** can be specified by users on a per-function basis.
    - E.g., time-consuming calls may have a larger base

Workflow of Pilgrim

1. Intercept every MPI call
2. Store ***entry/exit time***
3. **Encode parameters and compose the *call signature***
4. Map the ***call signature*** to a ***terminal symbol***
5. Use Sequitur algorithm to grow the CFG
6. Inter-process compression at the finalize point

Encoding function parameters

Generated automatically

1. Basic data types
2. MPI objects, e.g., `MPI_Request, MPI_Comm, etc.`
3. Pointers to memory buffers

Encoding function parameters

Basic data types

- We directly store the value of string and numerical parameters.
  - For **in** or **out** parameters, we store their values upon the return of the function call.
  - For **in/out** parameters, we keep the values both before and after the function call.

Encoding function parameters

MPI Objects

- Keep useful information to allow post-processing
  - Match `Isend/Wait` within one rank.
  - Match communicators across processes.

```
MPI_Isend(…, MPI_Request *request)

…

MPI_Wait*(MPI_Request *request)
```

```
MPI_Comm_split(…, MPI_Comm* newcomm)

…

// On rank A
MPI_Send(…, MPI_Comm comm)

// On rank B
MPI_Recv(…, MPI_Comm comm)
```

Encoding function parameters

MPI Objects

- Symbolic representation for every MPI Object.
    - e.g., `MPI_Datatype, MPI_Request, MPI_Comm`, etc.
- All MPI objects have a **locally unique ID**.
    - Can not directly use the MPI handle as it may be reused.
- Free on destroy function, e.g., `MPI_Type_free()`

```
MPI_Isend(…, MPI_Request *request)

…

MPI_Wait*(MPI_Request *request)
```

Encoding function parameters

MPI_Comm

- Most MPI calls require a `MPI_Comm` parameter. Same communicator should have have the same ID.
  - Make the comparison between processes meaningful.
  - Enables better compression ratio across processes.
    - e.g, `MPI_Barrier(comm)`
- The MPI_Comm object is locally unique, but not necessarily globally unique (two processes can have two different handles to the same underlying communicator)
  - We need to do extra work to convert the locally unique handle to a globally unique handle.

Encoding function parameters

MPI_Comm

- Basic idea: `MPI_Comm_dup()`
    - Choose a leader to decide a unique ID and broadcast to others.
- Inter-communicators are tricky and are handled slightly differently (ask me for details over a coffee!):
    - `MPI_Intercomm_create()`, `MPI_Comm_spawn()`
    - `MPI_Comm_accept()`, `MPI_Comm_connect()`
- Non-blocking communicator creation is messy because the communicator handle is not immediately created (again, ask for details over a coffee):
    - `MPI_Comm_idup()`

Encoding function parameters

Memory addresses (void*)

- Memory address itself does not provide much information
- We also use symbolic representation for all memory pointer parameters.
  - Same symbol means same memory starting address.
  - Intercept memory operations, e.g., malloc, calloc, free, etc.
    - Using stack variables is legal, but evil. Don't use them. :-)
  - Future work: Reserve some bits in the symbolic ID to store CPU/GPU and device information -- current implementation skips this

```
MPI_Send(&data, …)
…

MPI_Send(&data, …);
```

```
MPI_Send(&(data[0]), …)
…

MPI_Send(&(data[1]), …);
```

Review: Workflow of Pilgrim

1. Intercept every MPI call
2. Store the ***interval*** and ***duration***
3. Encode parameters and compose the ***call signature***
4. Map the ***call signature*** to a ***terminal symbol***
5. Use Sequitur algorithm to grow the CFG
6. Inter-process compression at the finalize point

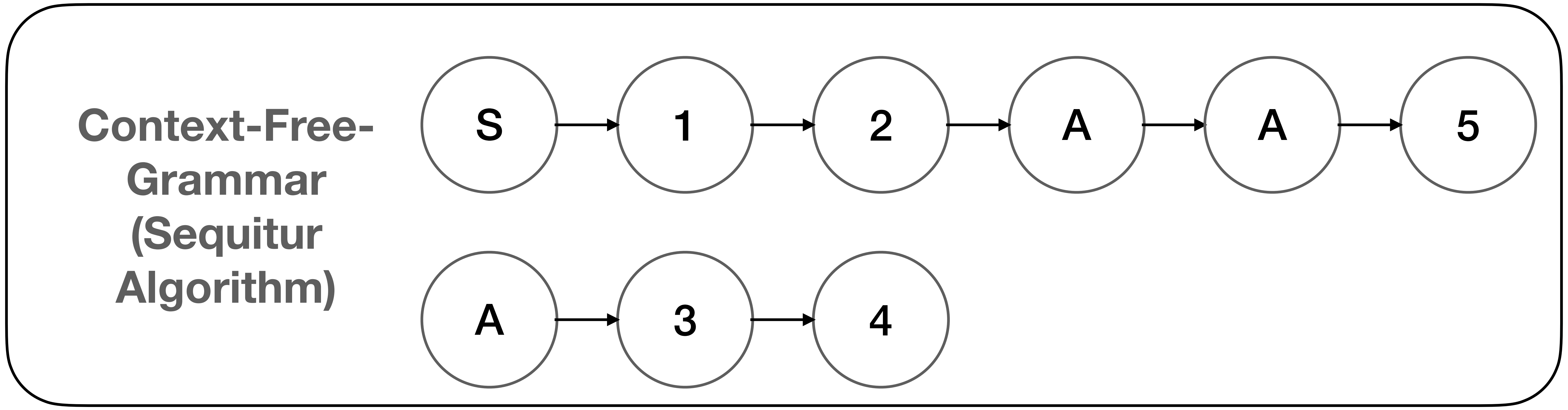
Review: Workflow of Pilgrim

Example

```
MPI_Comm_size(comm1, &size)
MPI_Comm_rank(comm1, &rank)
MPI_Send(buf, 1, MPI_INT, dest, tag, comm1)
MPI_Barrier(comm1)
MPI_Send(buf, 1, MPI_INT, dest, tag, comm2)
MPI_Barrier(comm1)
MPI_Barrier(comm2)
```

Call Signature	Terminal Symbol
`MPI_Comm_size(comm1, 2)`	1
`MPI_Comm_rank(comm1, 0)`	2
`MPI_Comm_send(buf, 1, MPI_INT, 2, 100, comm1)`	3
`MPI_Barrier(comm1)`	4
`MPI_Barrier(comm2)`	5

1 → 2 → 3 → 4 → 3 → 4 → 5



Inter-process Compression

Call Signatures Table (CST)

1. Merge the CST (bottom-up) from every process.
2. Duplicated entries are eliminated during the merging process.
3. In the end, one rank holds the merged CST
    1. Re-assign terminal ID. (Each unique call signature has a globally unique terminal symbol)
    2. Broadcast the updated CST.
4. Update the grammar as the terminal symbols may have been changed.

Call Signature	Terminal
`MPI_Barrier(comm1)`	1
`MPI_Barrier(comm2)`	2
`MPI_Barrier(comm3)`	3

Call Signature	Terminal
`MPI_Barrier(comm1)`	1
`MPI_Barrier(comm2)`	2

Rank 0

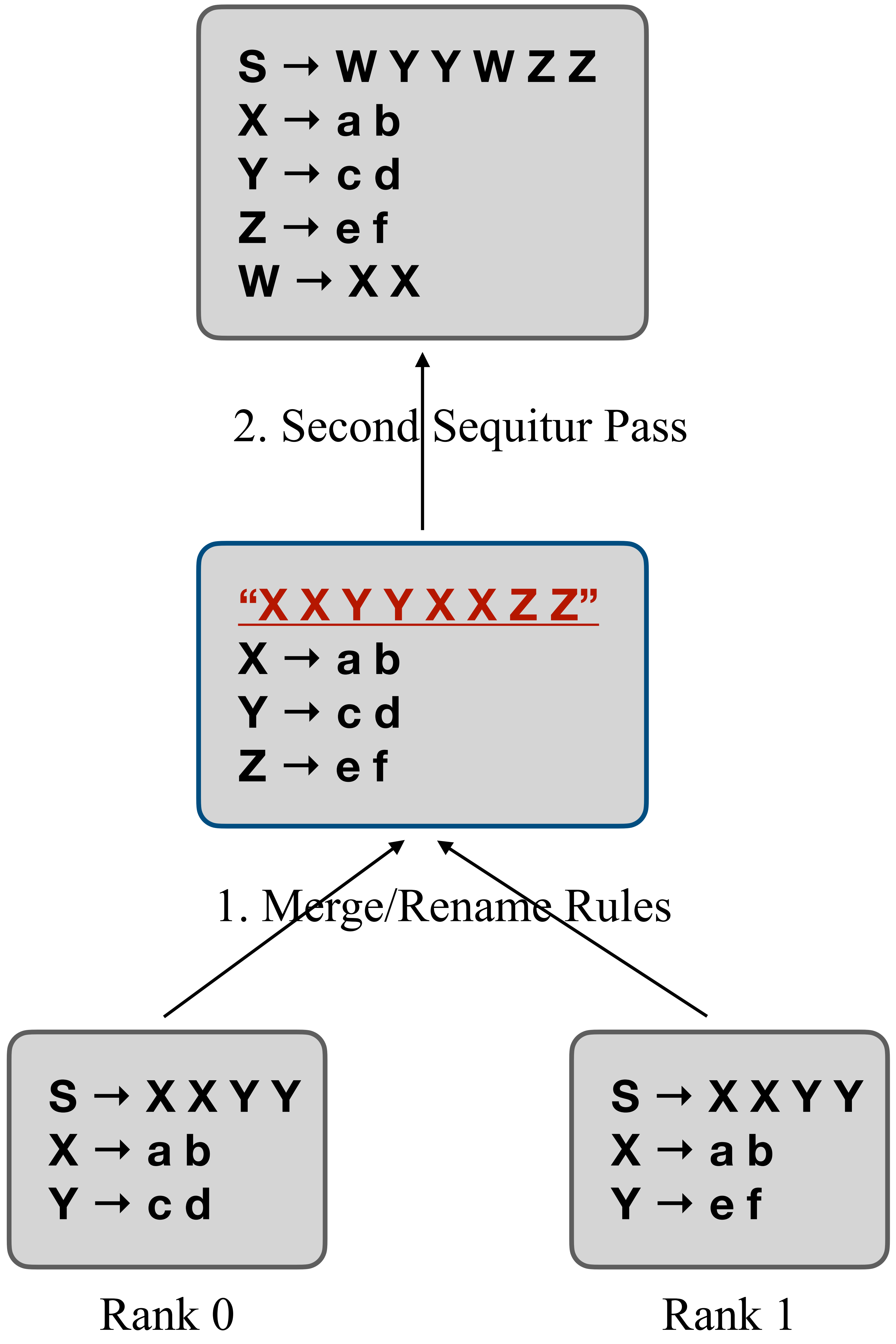
Call Signature	Terminal
`MPI_Barrier(comm1)`	1
`MPI_Barrier(comm3)`	2

Rank 1

Inter-process Compression

CFG

1. Merge grammars (bottom-up) from every process.
    - Duplicated rules are eliminated during the merging process.
    - Rules may need renaming.
2. Run another Sequitur pass to build (and output) the final grammar.



Evaluation

- 6 scientific applications with 1024 process runs.
    - Trace file size
    - Overhead
- How does the trace file size scale with the scale for runs?
    - Problem Size
    - Number of iterations
    - Number of processes

Evaluation

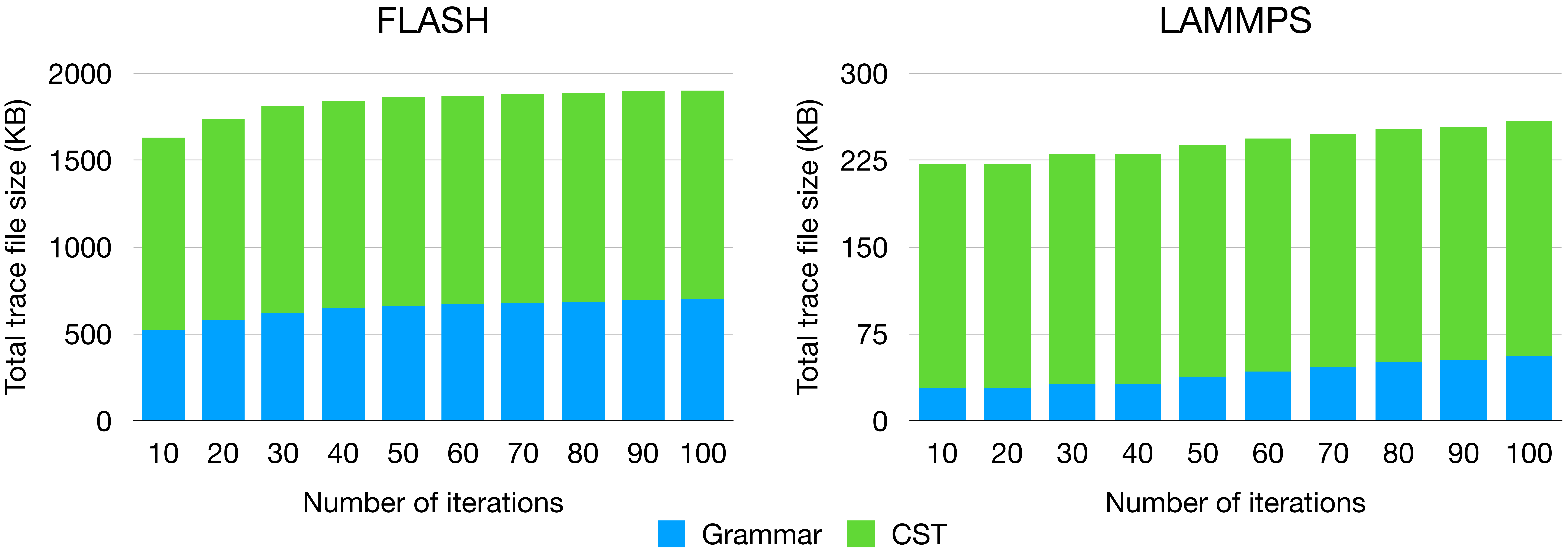
1024 Procs Run (32 nodes, 32 processes per node)

App	Execution Time(s)	Trace File Size	Overhead
QMCPack	204.41	11.6MB	4.61%
Chombo	31.32	39.2MB	13.4%
FLASH	216.37	390KB	10.7%
MILC	103.88	6.7MB	14.3%
LAMMPS	114.47	1.04MB	12.5%
GAMESS	13.01	817KB	11.7%

Evaluation

Trace File Size vs. Number of Iterations

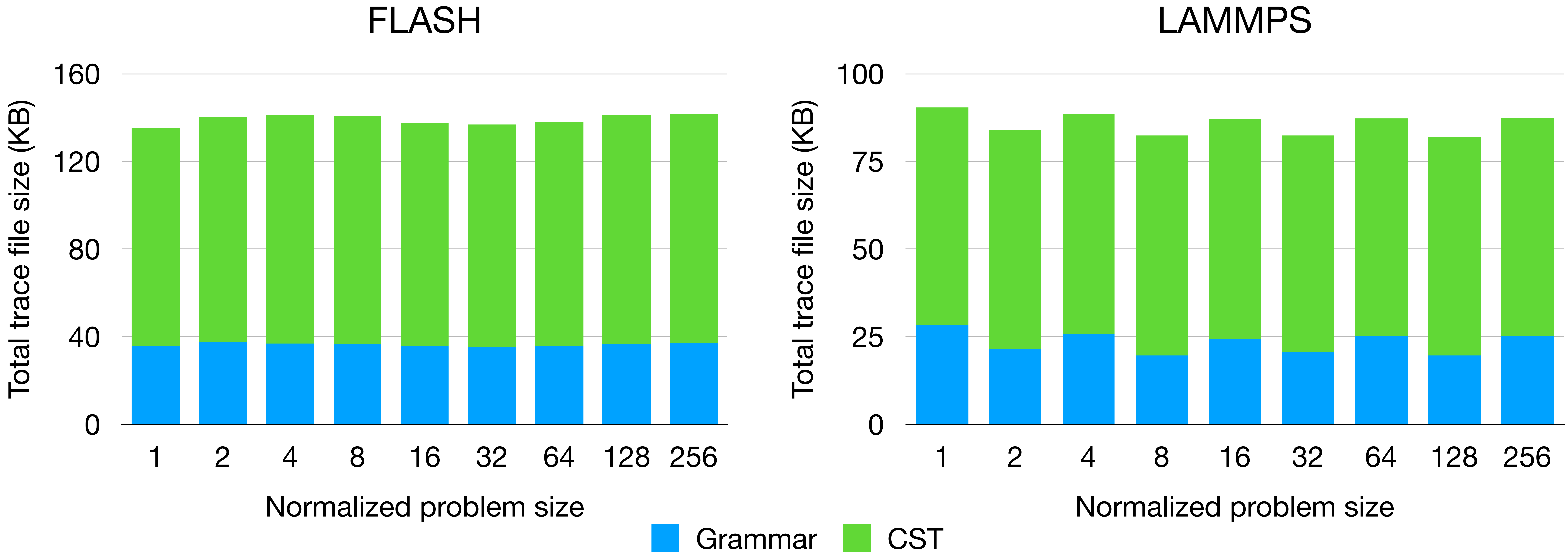
- 32 nodes and 32 processes per node for 1024 MPI ranks in total.
- FLASH: Sedov 2D, mesh size: 512x512; LAMMPS: LJ 2D, mesh size: 1024x1024.



Evaluation

Trace File Size vs. Problem Size

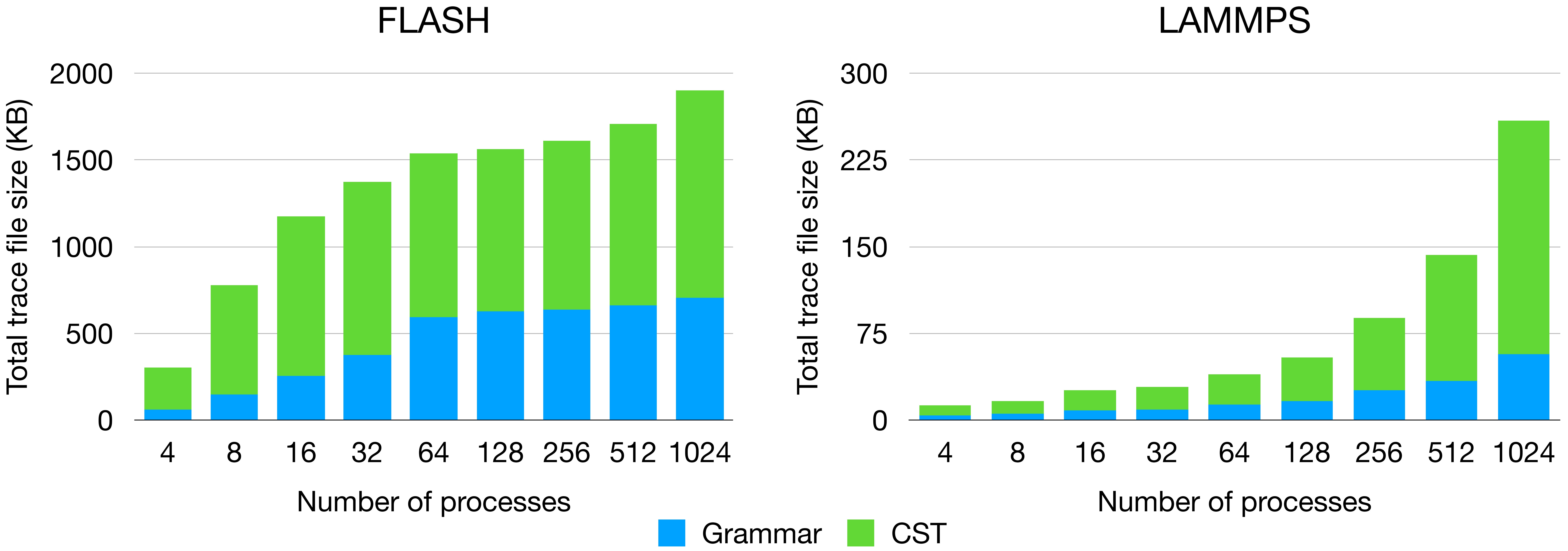
- 32 nodes and 32 processes per node for 1024 MPI ranks in total. Both run 100 iterations.
- Increase the problem size by doubling one dimension at a time.



Evaluation

Trace File Size vs. Number of Processes

- FLASH: Sedov 2D, weak scaling, mesh size 16x16/process.
- LAMMPS: LJ 2D, strong scaling, mesh size: 1024x1024.



Details Not Covered and Future Work

- Decoder
- Test on large-scale and longer runs.
- The context-free-grammar can be optimized especially for loops
  - O(1) space instead of O(logN)
- Extend symbolic representation for memory buffers to include memory locations.
- Further optimize code to reduce the overhead.
- Generate mini-apps from traces.
- Better compression for “slowly evolving irregular codes” (AMR)
- Better time encoding to avoid drift
- Encoding communication graph
  - Done for 1D, need to generalize 2D,3D and for irregular meshes

Thanks!
Questions?

Backup Slides

Encoding Function Parameters

Symbolic representation for every MPI object

- One hash table for each MPI Object Type: **MPI Object → id**
- One doubly linked list for each MPI Object Type to keep track of **free ids**.
- General MACROS:
    - `MPI_OBJ_ID(Type, obj)`
        - e.g., `MPI_OBJ_ID(MPI_Request, req)`
        - Query the object id, create one if not exists.
    - `MPI_OBJ_FREE(Type, obj)`
        - Free the resource in hash table; Insert the associated into the free id list.
        - Need to be called at object release point, e.g., `MPI_Type_free()`

Encoding Function Parameters

Symbolic representation for every MPI object

`MPI_Comm`: same communicator should have the same id even across ranks.

- Inter-communicator: e.g., `MPI_Comm_accept`/`MPI_Comm_connect`
    1. Server creates the id.
    2. Send to the client.
    3. Broadcast within the local communicator.
- `MPI_Intercomm_create`(), `MPI_Comm_spawn`(), etc.

Encoding Function Parameters

Symbolic representation for every MPI object

`MPI_Comm_idup(comm, &newcomm, &request)`

- At the time of this call, `newcomm` may not be ready.
- Need to remember the request and check later on `MPI_Wait`*() or `MPI_Test`*().

Encoding Function Parameters

Symbolic representation for every MPI object

`MPI_Request` and `MPI_Status`

- Remember the source (could be ANY_SOURCE) and the tag (could be ANY_TAG) of a MPI_Request object.
- For **MPI_Status**, save `status->MPI_TAG` or `status->MPI_SOURCE` only if `source == ANY_SOURCE` or `tag == ANY_TAG`.