Pilgrim

Scalable and (near) Lossless MPI Tracing

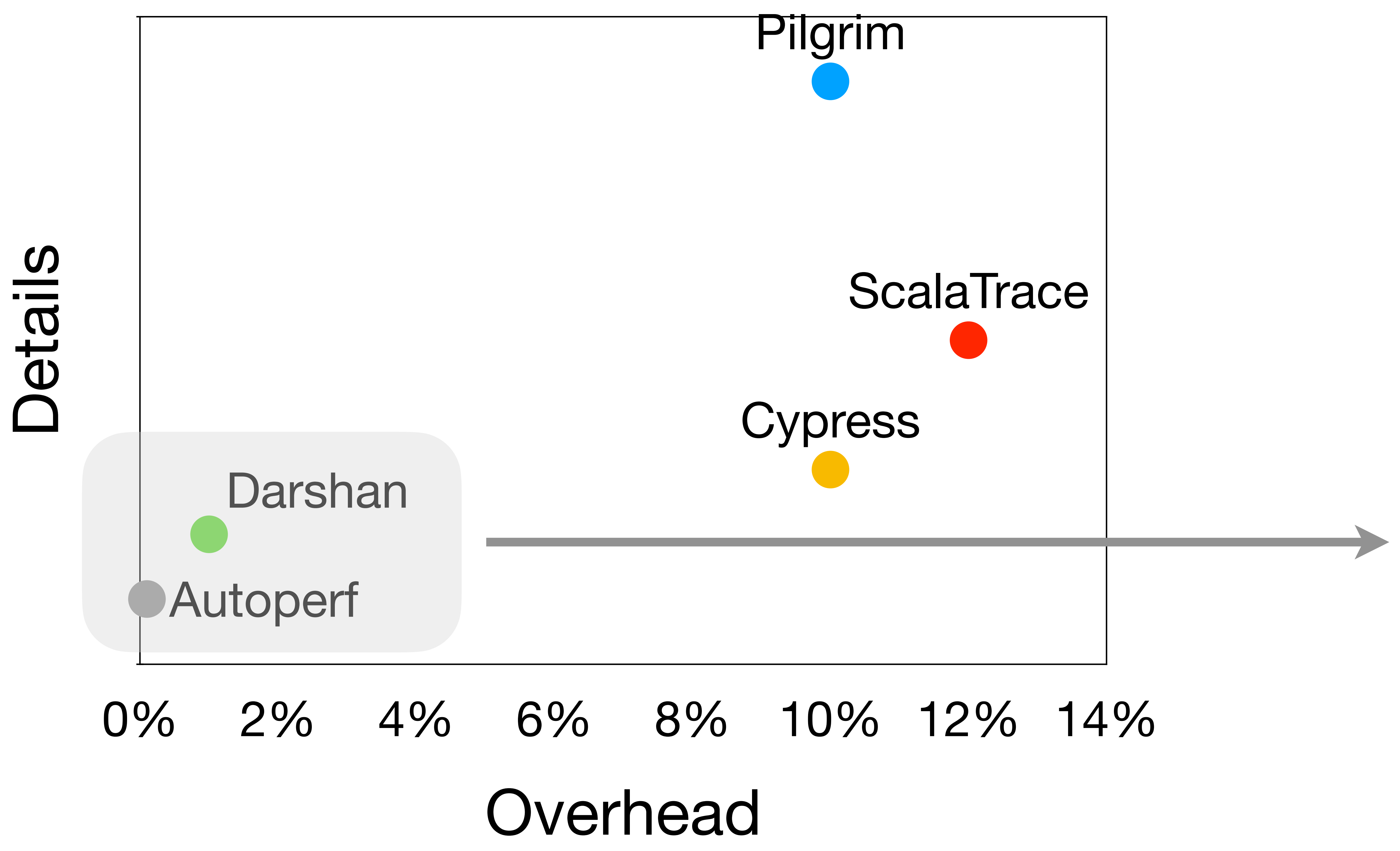
Chen Wang[1], Pavan Balaji[2], Marc Snir[1]
1. University of Illinois Urbana-Champaign
2. Meta, Inc; Argonne National Laboratory

2021/11/17

Why do we collect MPI information

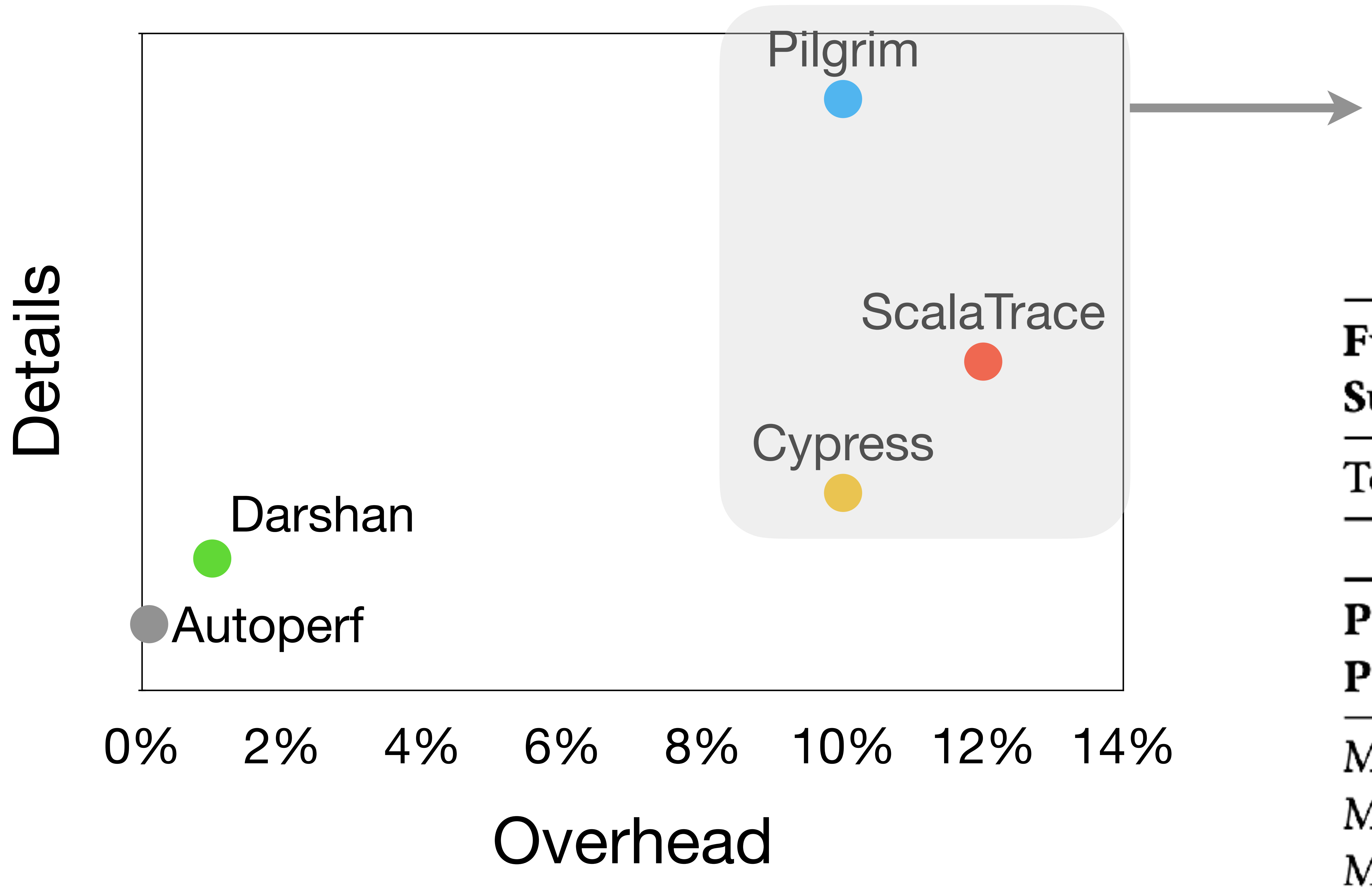
- MPI is a prominent programming model used for scientific computing.
  - Different applications use MPI differently.
  - Important to understand MPI usage for different applications.
- For MPI and application users:
  - How frequent calls are?
  - Am I providing the right hints to MPI for my usage?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

Tradeoff between details and overhead



- Profiling tools store summarized (lossy) information about MPI calls.
- They have very low overhead.

Tradeoff between details and overhead



- Tracing tools keep detailed information but incur higher overhead
- Existing tools are either incomplete or have unaccepted overhead (time or space)

Functions Supported	Cypress	ScalaTrace	Pilgrim
Total: 446	56	125	446

Popular Parameters	Cypress	ScalaTrace	Pilgrim
MPI_Status	✓	✓	✓
MPI_Request	×	✓	✓
MPI_Comm	intra	intra and inter	intra and inter
MPI_Datatype	only the size	✓	✓
src/dst/tag	✓	✓	✓
memory pointer	×	×	✓

Pilgrim

- Lossless for MPI functionality
  - **Pilgrim stores every parameter of every MPI call**.
- Lossy for non-MPI data
  1. The entry/exit timestamps are approximated to save space.
     - Useful for understanding skew between processes, depending on how much approximation
  2. Actual communicated and I/O data is not saved.
  3. Virtual addresses of memory buffers are summarized using symbolic representations

How can we use lossless MPI traces?

- In-depth analysis is made possible
  - Understanding patterns of communication when multiple processes are involved.
  - 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 functionalities, ...
- Generating automatically MPI mini apps from full applications (including from closed source or export controlled applications, e.g., from the NNSA labs).

Challenges

- Scalability: the longer an application run or the more nodes it runs one, the more function calls it will make
  - Need to store huge volume of information for large scale runs with acceptable overhead (space and time)
- Usefulness: the stored information should be meaningful for post-processing.
  - What information do we need to store for each MPI object, e.g., `MPI_Comm` and `MPI_Request`
  - Memory pointers?
- Correctness and completeness:
  - Over 400 MPI functions
  - Many corner cases, e.g., non-blocking communication creation.

Design and Implementation

How to store lossless MPI information for large scale runs?

- The longer an application run or the more nodes it runs one, the more function calls it will make
- Primarily relies on “**recurring pattern recognition**”
  - Most applications have recurring patterns of communication
    - Intra-process and inter-process.
  - We use a context-free-grammar (CFG) and a well-known algorithm called “Sequitur algorithm” for this.
  - The key is to detect as many patterns as possible.

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.
  - Sequitur is optimized by adding to the notation **repetition counts.** Reduces space complexity for regular loops from $O(logN)$ to $O(1)$.

“a a b a b c d c d”

Sequitur Algorithm

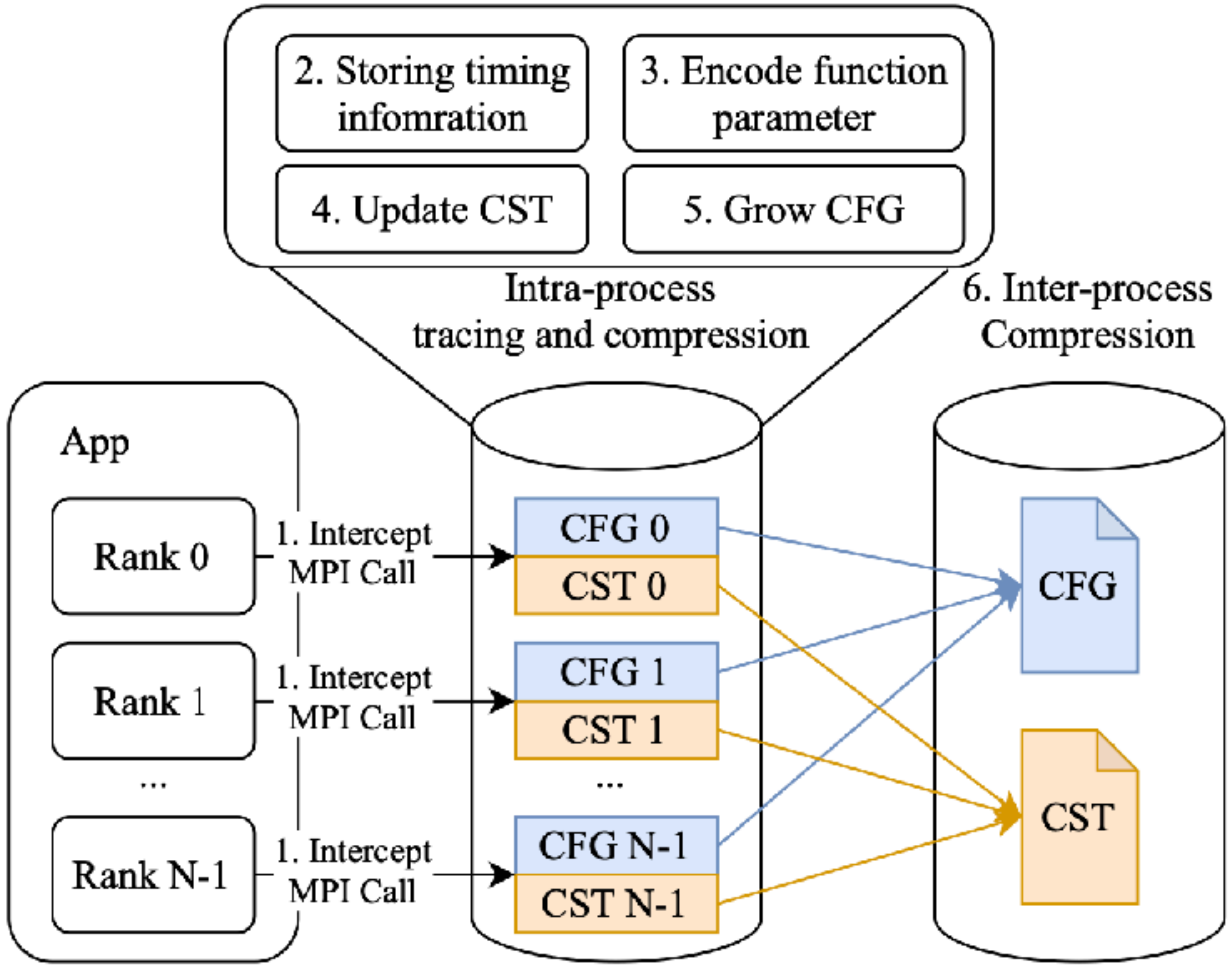
$S \rightarrow a\ A^2\ B^2$

$A \rightarrow a\ b$

$B \rightarrow c\ d$

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*** (existing or newly created)
5. Use Sequitur algorithm to grow the CFG
6. Perform inter-process compression at the finalize point



Intercepting MPI calls

- Wrappers for intercepting the calls are generated automatically based on MPI document (Latex files).

```
prologue();
PMPI_*();      // calls the original function
epilogue();
```

- `prologue()` stores *call entry time* and *input parameters*.
- `epilogue()` stores *call exit time* and *output parameters* such as `MPI_Status`.

Call Signature

- Call signature: function name and function parameter values
- Each terminal symbol in the grammar represents a **unique call signature**.

`MPI_Barrier(comm1);`

“a a b a b c d c d”

Sequitur

S → a A² B²
A → a b
B → c d

`MPI_Comm_size(comm1, 2);`

`MPI_Comm_size(comm2, 3);`

Call Signature	Terminal
`MPI_Barrier(comm1)`	a
`MPI_Comm_size(comm1, 2)`	b
`MPI_Comm_size(comm2, 3)`	c
…	

A **call signature table (CST)** is used to maintain the mapping between the call signature and the terminal symbols

Encoding function parameters

Generated automatically

1. Basic data types, e.g., int, double, etc.
    - Directly store the values
2. MPI objects, e.g., `MPI_Request, MPI_Comm, etc.`
3. Pointers to memory buffers

Encoding function parameters

MPI Objects

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

```
MPI_Isend(…, request)

…

MPI_Wait*(…, request)
```

```
MPI_Comm_split(…, &newcomm)

…

// On rank A
MPI_Send(…, newcomm)

// On rank B
MPI_Recv(…, newcomm)
```

Encoding function parameters

MPI Objects

- Can not directly use the MPI handle as it may be reused.
- Symbolic representation for every MPI Object.
  - e.g., `MPI_Datatype`, `MPI_Request`, `MPI_Comm`, etc.
  - One symbolic Id pool per MPI objet.
    - All MPI objects get a **locally unique ID**.

```
MPI_Isend(…, request)

…

MPI_Wait*(…, request)
```

Encoding function parameters

MPI_Comm

- Unlike other MPI objects, Id of `MPI_Comm` is **globally unique** to simplify the matching process.
- Basic idea: Choose a leader to decide a unique ID and broadcast to others.
  - Intra-communicators:
    - `MPI_Comm_split()`, `MPI_Comm_create()`, etc.
  - Inter-communicators:
    - `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 (ask me for details over a beer)
    - `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.
  - *(Symbolic ID, Buffer size, Offset, CPU or GPU, Device if on GPU)*
  - Intercept memory operations, e.g., malloc, calloc, free, etc.
    - Using stack variables is legal, but evil. Don't use them. :-)

```
MPI_Send(&data, …)
…

MPI_Send(&data, …);
```

```
MPI_Send(&(data[0]), …)
…

MPI_Send(&(data[1]), …);
```

Encoding function parameters

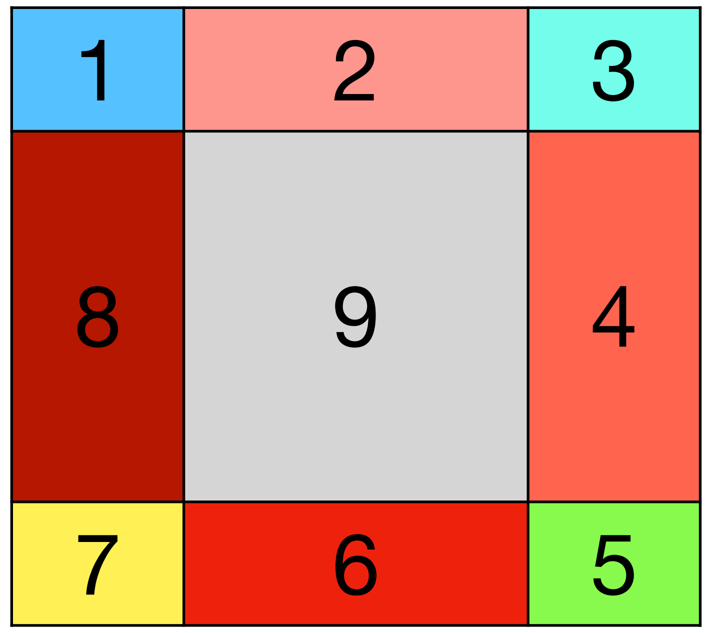
Optimizations

- Rank-related encoding
  - e.g., `MPI_Send(dst = my_rank + 1)`, `MPI_Comm_split(color?)`
  - Parameters that are rank related. We can detect linear patterns of the form $a*my\_rank + b$.
  - Critical for inter-process compression.
- Non-deterministic loops
  - Nondeterministic loops will generate different sequence of call signatures per iteration.
  - One symbolic id pool per call signature

```
for {
    MPI_Irecv(from = my_rank + 1, &req1);
    MPI_Irecv(from = my_rank + 2, &req2);
    MPI_Isend(to = my_rank + 3, &req3);
    while(!(all requests finished)) {
        MPI_Waitany([req1, req2, req3]);
        handle received message;
    }
}
```

Inter-process compression

- Inter-process compression is important to achieve the scalability.
  - Detect recurring communication patterns across ranks.
    - e.g., 2D 5-points periodical stencil will generate up to 9 unique grammars
  - Parallel pairwise merge for process-local CSTs and CFGs
    - Bottom-up approach, $O(logP)$ time complexity.



Example

Rank 0:

```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
for(int i = 0; i < 10; i++)
    MPI_Send(buf, MPI_INT, 1, 999, comm)
```

Rank 1:

```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
for(int i = 0; i < 10; i++)
    MPI_Recv(buf, MPI_INT, 0, 999, comm)
```

Example

Rank 0:

```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
for(int i = 0; i < 10; i++)
    MPI_Send(buf, MPI_INT, 1, 999, comm)
```

Rank 1:

```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
for(int i = 0; i < 10; i++)
    MPI_Recv(buf, MPI_INT, 0, 999, comm)
```

CST of Rank 0:

Call Signature	Terminal
`MPI_Comm_size(comm, 2)`	1
`MPI_Comm_rank(comm, 0)`	2
`MPI_Send(buf, MPI_INT, 1, 999, comm)`	3

CST of Rank 1:

Call Signature	Terminal
`MPI_Comm_size(comm, 2)`	1
`MPI_Comm_rank(comm, 1)`	2
`MPI_Recv(buf, MPI_INT, 0, 999, comm)`	3

Example

Rank 0:

```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
for(int i = 0; i < 10; i++)
    MPI_Send(buf, MPI_INT, 1, 999, comm)
```

Rank 1:

```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
for(int i = 0; i < 10; i++)
    MPI_Recv(buf, MPI_INT, 0, 999, comm)
```

CST of Rank 0:

Call Signature	Terminal
`MPI_Comm_size(comm, 2)`	1
`MPI_Comm_rank(comm, 0)`	2
`MPI_Send(buf, MPI_INT, 1, 999, comm)`	3

CST of Rank 1:

Call Signature	Terminal
`MPI_Comm_size(comm, 2)`	1
`MPI_Comm_rank(comm, 1)`	2
`MPI_Recv(buf, MPI_INT, 0, 999, comm)`	3

CFG of Rank 0: $S \rightarrow 1\ 2\ 3^{10}$

CFG of Rank 1: $S \rightarrow 1\ 2\ 3^{10}$

Example

Inter-process CST compression

Call Signature	Terminal
`MPI_Comm_size(comm, 2)`	1
`MPI_Comm_rank(comm, myrank)`	2
`MPI_Send(buf, MPI_INT, 1, 999, comm)`	3
`MPI_Recv(buf, MPI_INT, 0, 999, comm)`	4

Example

Inter-process CST compression

Call Signature	Terminal
`MPI_Comm_size(comm, 2)`	1
`MPI_Comm_rank(comm, myrank)`	2
`MPI_Send(buf, MPI_INT, 1, 999, comm)`	3
`MPI_Recv(buf, MPI_INT, 0, 999, comm)`	4

CFG of Rank 0:

$$S \rightarrow 1\ 2\ 3^{10}$$

CFG of Rank 1:

$$S \rightarrow 1\ 2\ 4^{10}$$

Inter-process CFG compression

$$S \rightarrow S_1\ S_2$$
$$S_1 \rightarrow A\ 3^{10}$$
$$S_2 \rightarrow A\ 4^{10}$$
$$A \rightarrow 1\ 2$$

Evaluation

- What is the trace size for large scale runs?
- How do trace size and overhead scale with the number of iterations and the number of processes
- How does Pilgrim compare with other systems?

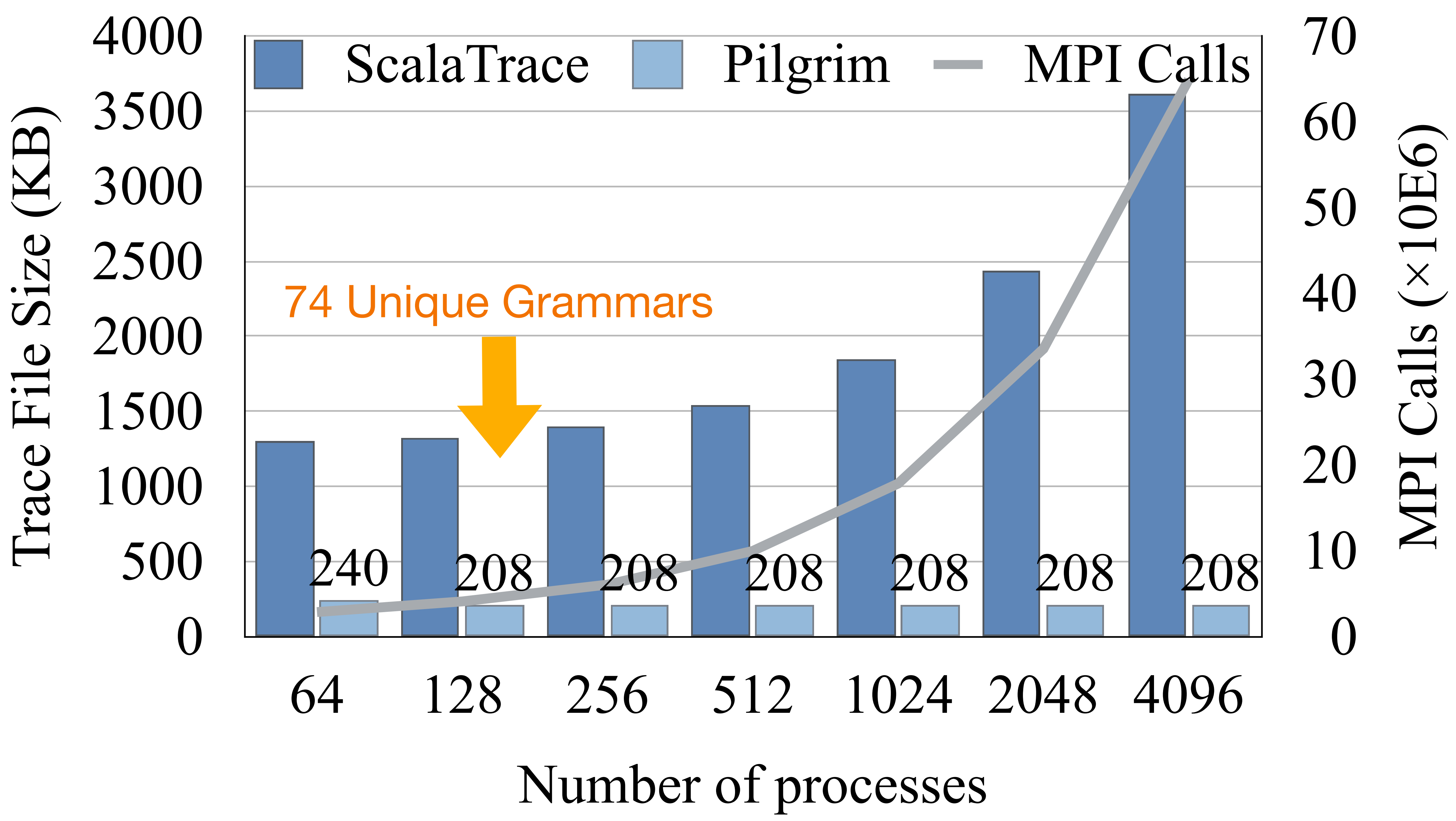
Type	**Code**	**Platform**
Benchmark	2D and 3D Stencils OSU Mrico-Benchmarks	Catalyst at LLNL: Intel Xeon E5-2695, 24 cores; 128GB DDR4, IB QDR
Mini App	NAS Parallel Benchmark	
Real App	FLASH and MILC	Theta at ANL: Intel KNL 7230, 64 cores; 192GB DDR4; Aries Dragonfly

Evaluation

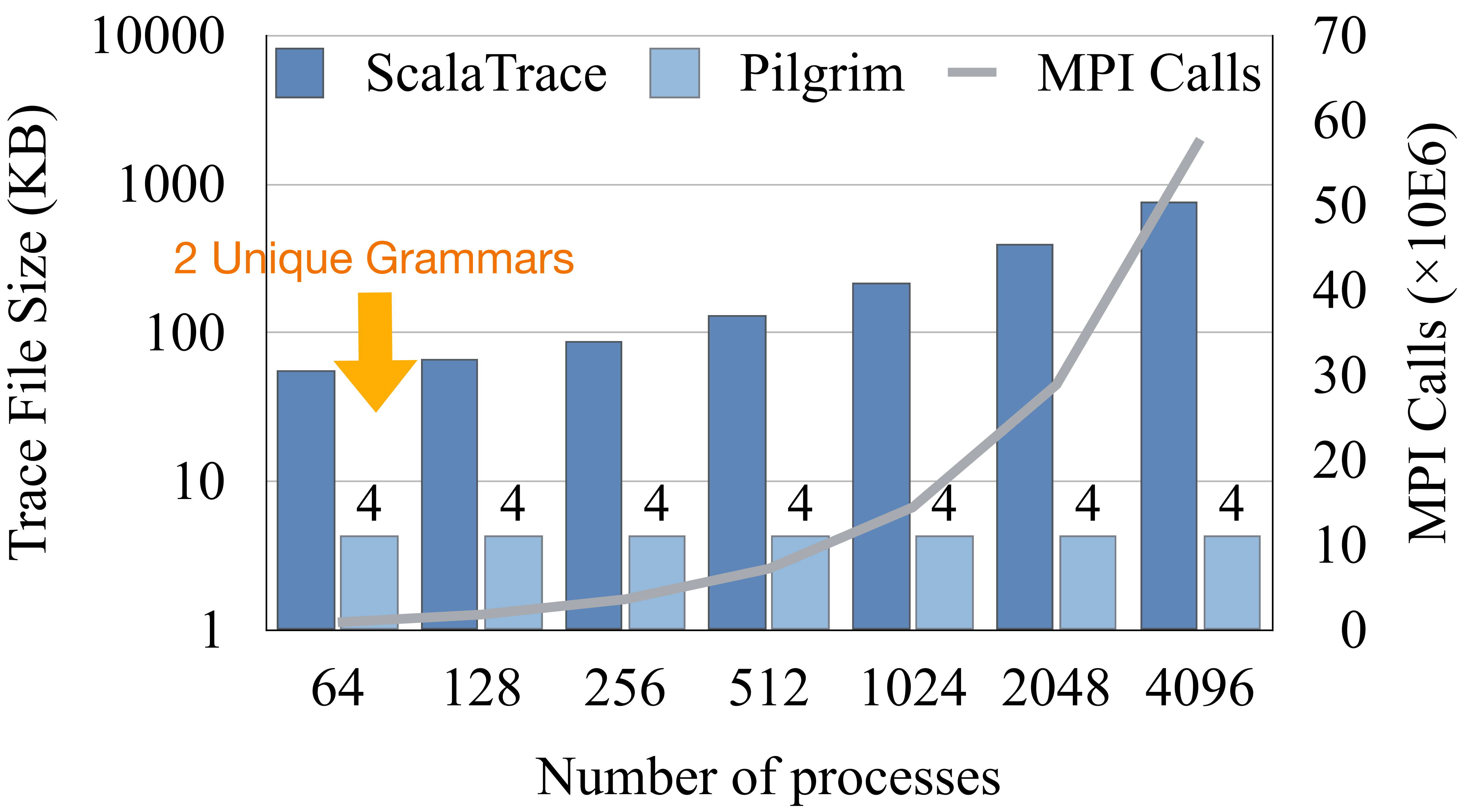
How does trace size scale with the number of processes?

- Only unique communication patterns (unique grammars) matter.
  - Trace size will stay constant if no new patterns are introduced.

FLASH - Sedov (500 iterations)



FLASH - StirTurb (500 iterations)

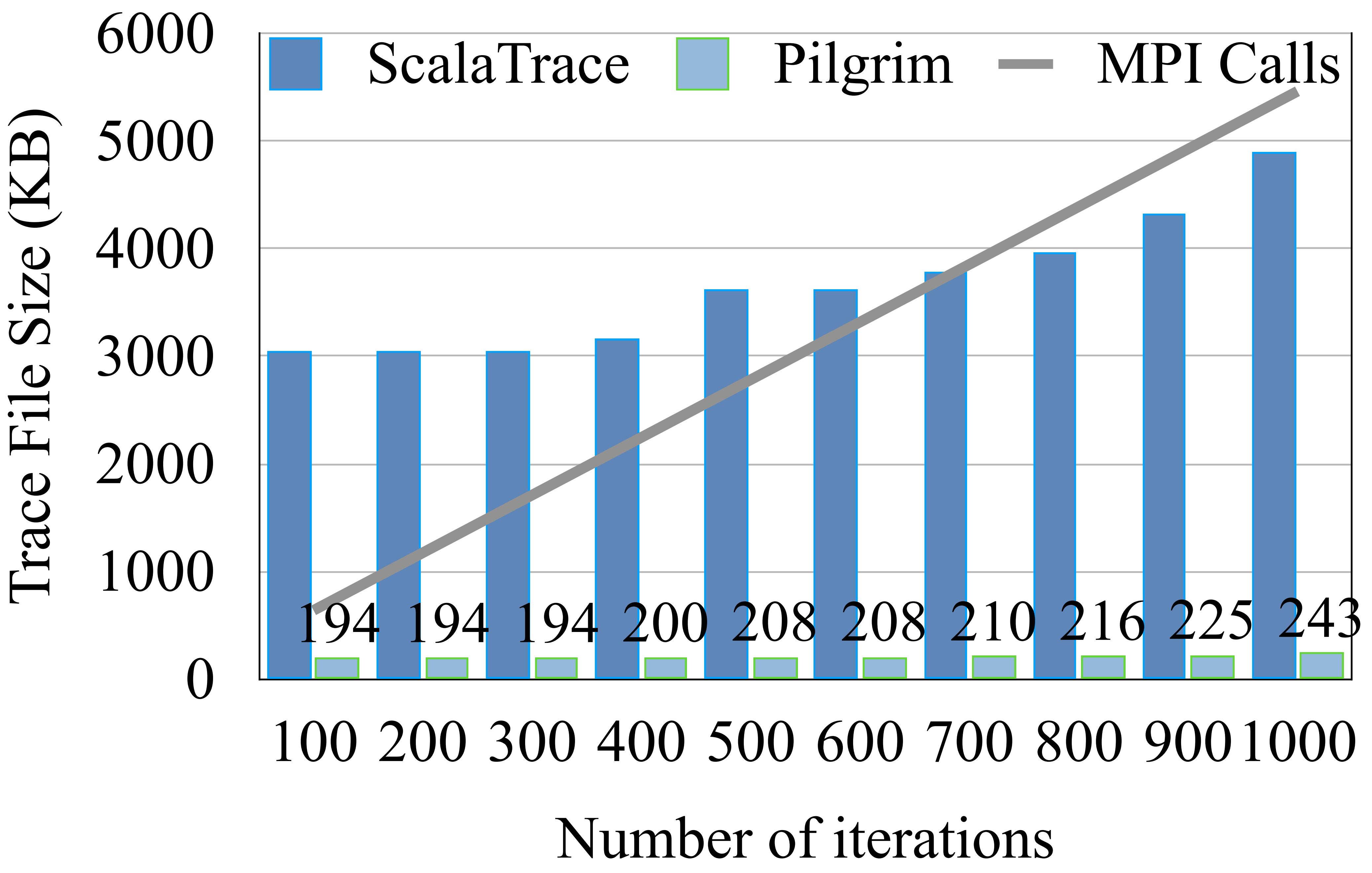


Evaluation

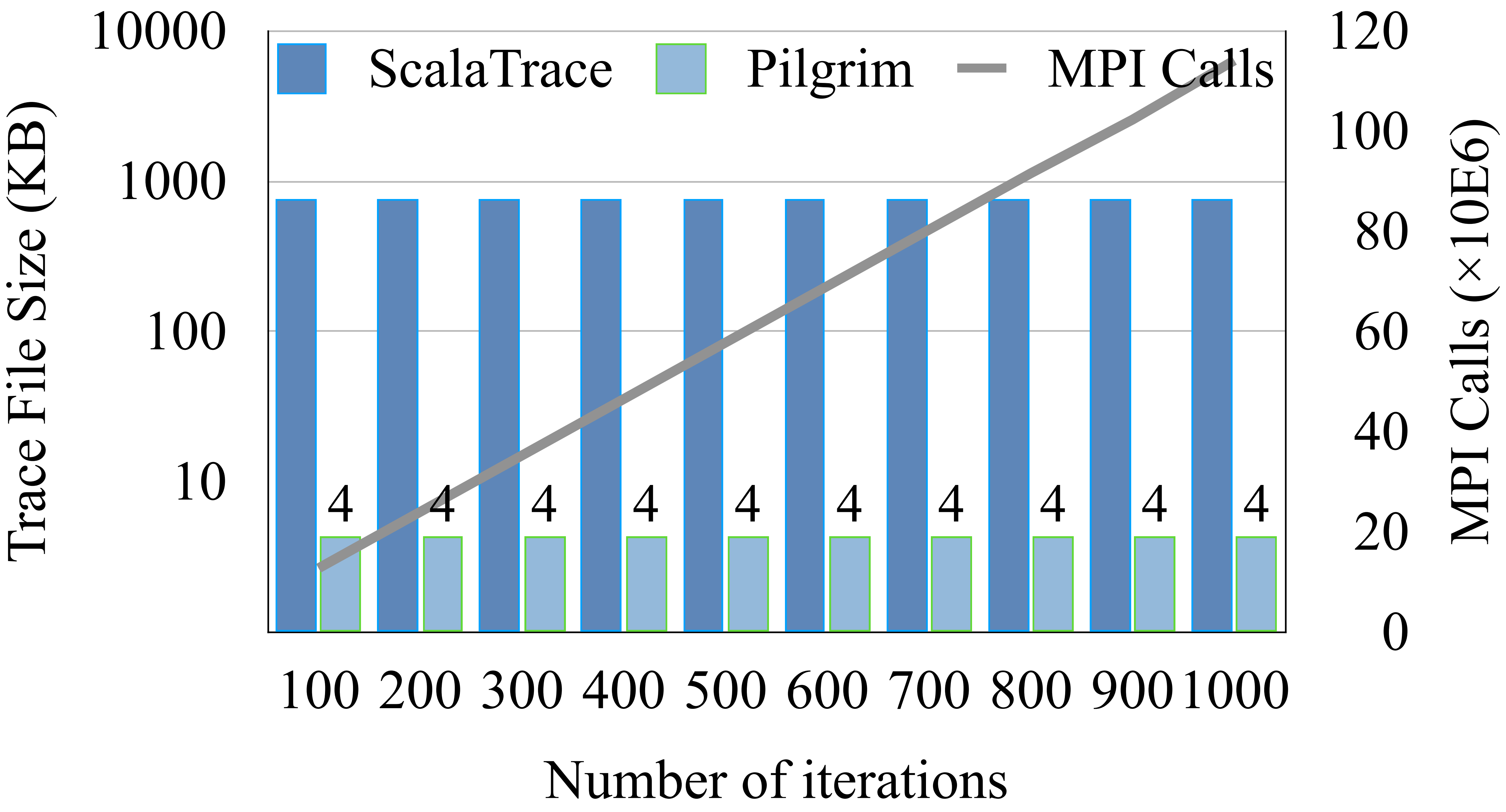
How does trace size scale with the number of iterations?

- Only unique communication patterns matter.
  - Trace size will stay constant if no new patterns are introduced.
  - Adaptive mesh refinement (AMR) will introduce new patterns.

FLASH - Sedov (4096 processors)



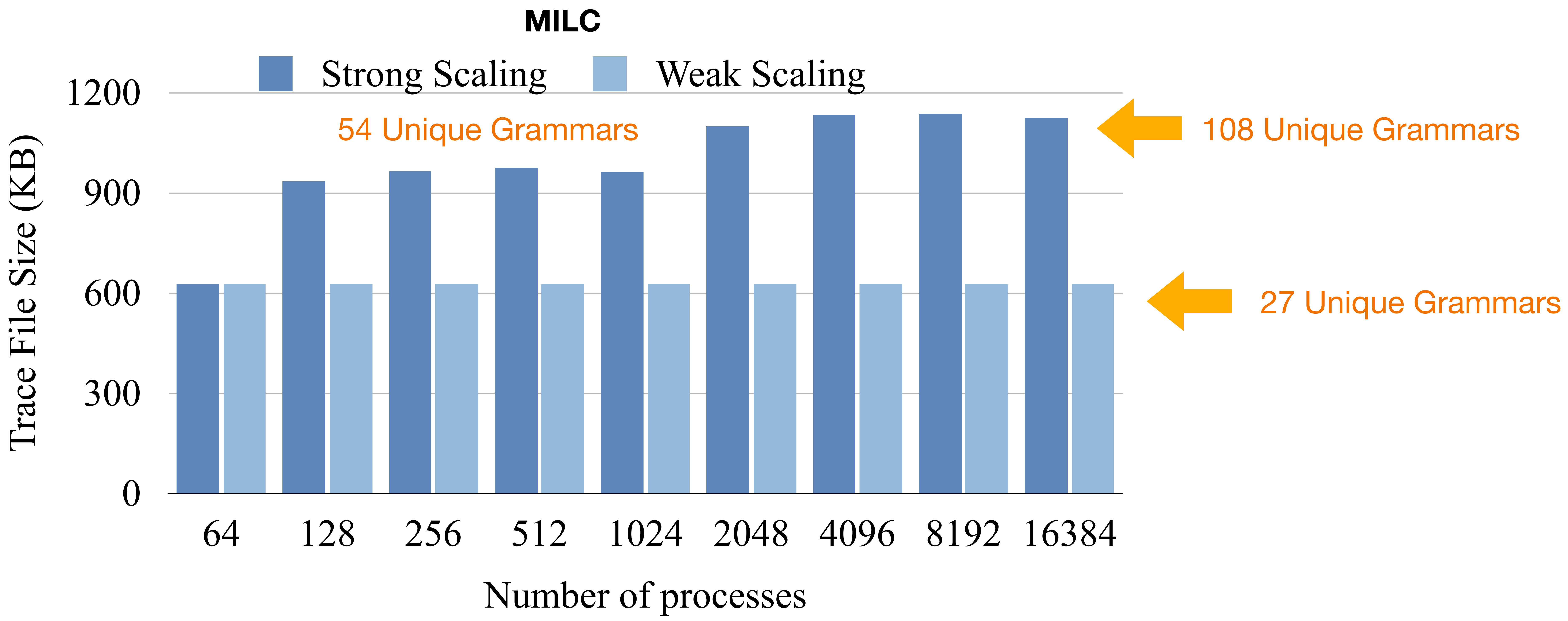
FLASH - StirTrub (4096 processors)



Evaluation

Large scale experiments

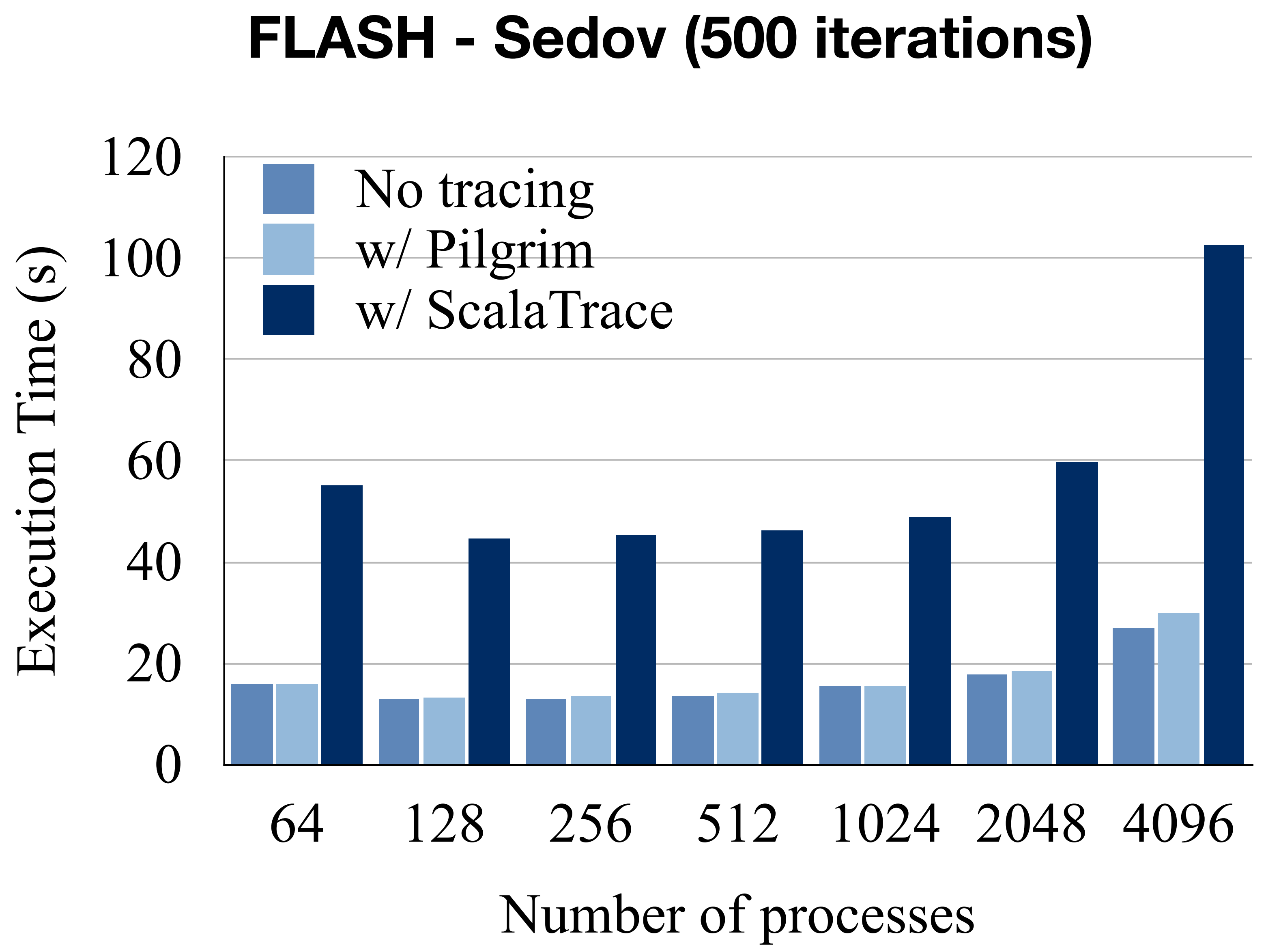
- Only unique communication patterns matter.
  - Trace size will stay constant if no new patterns are introduced.

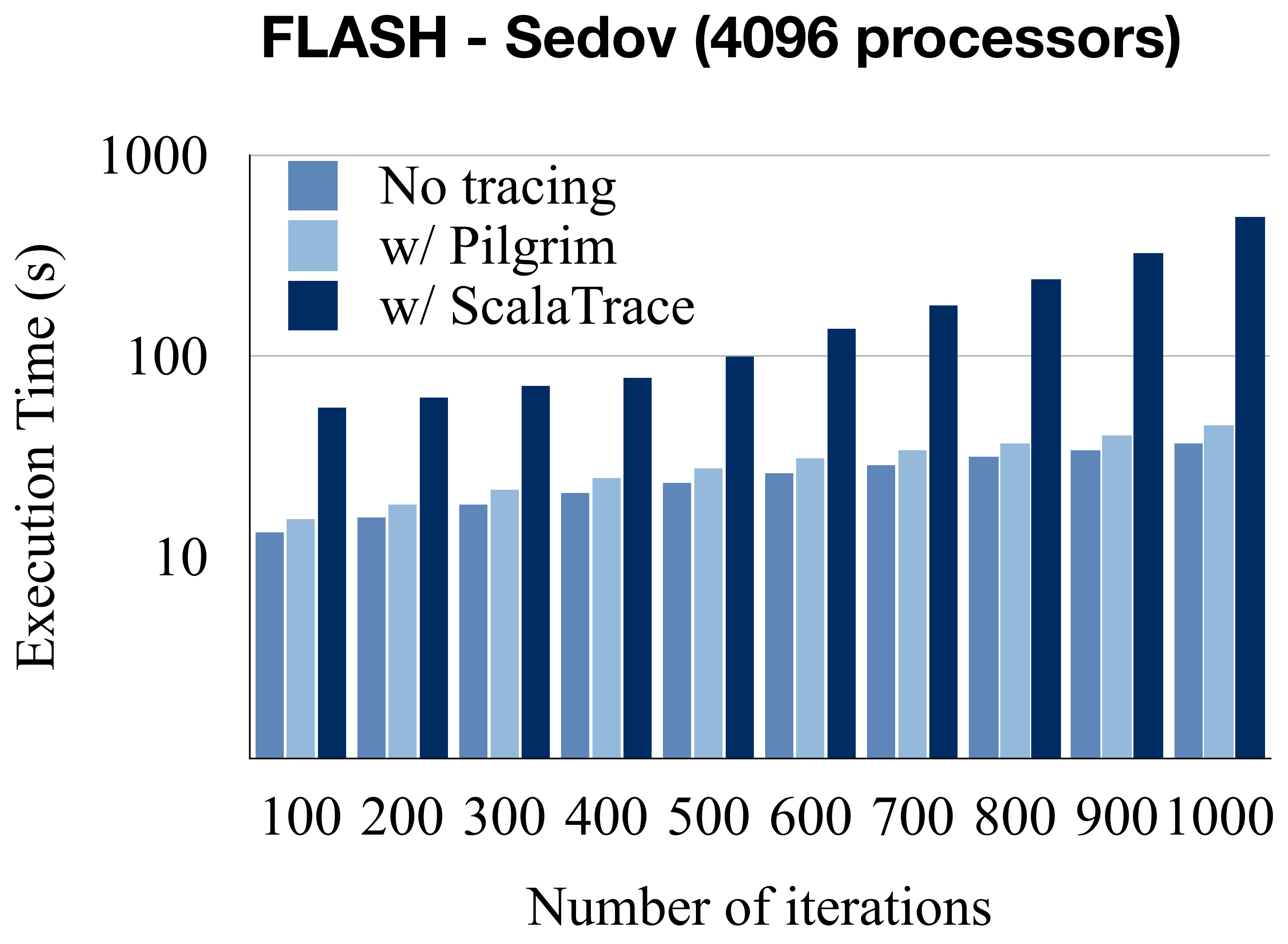


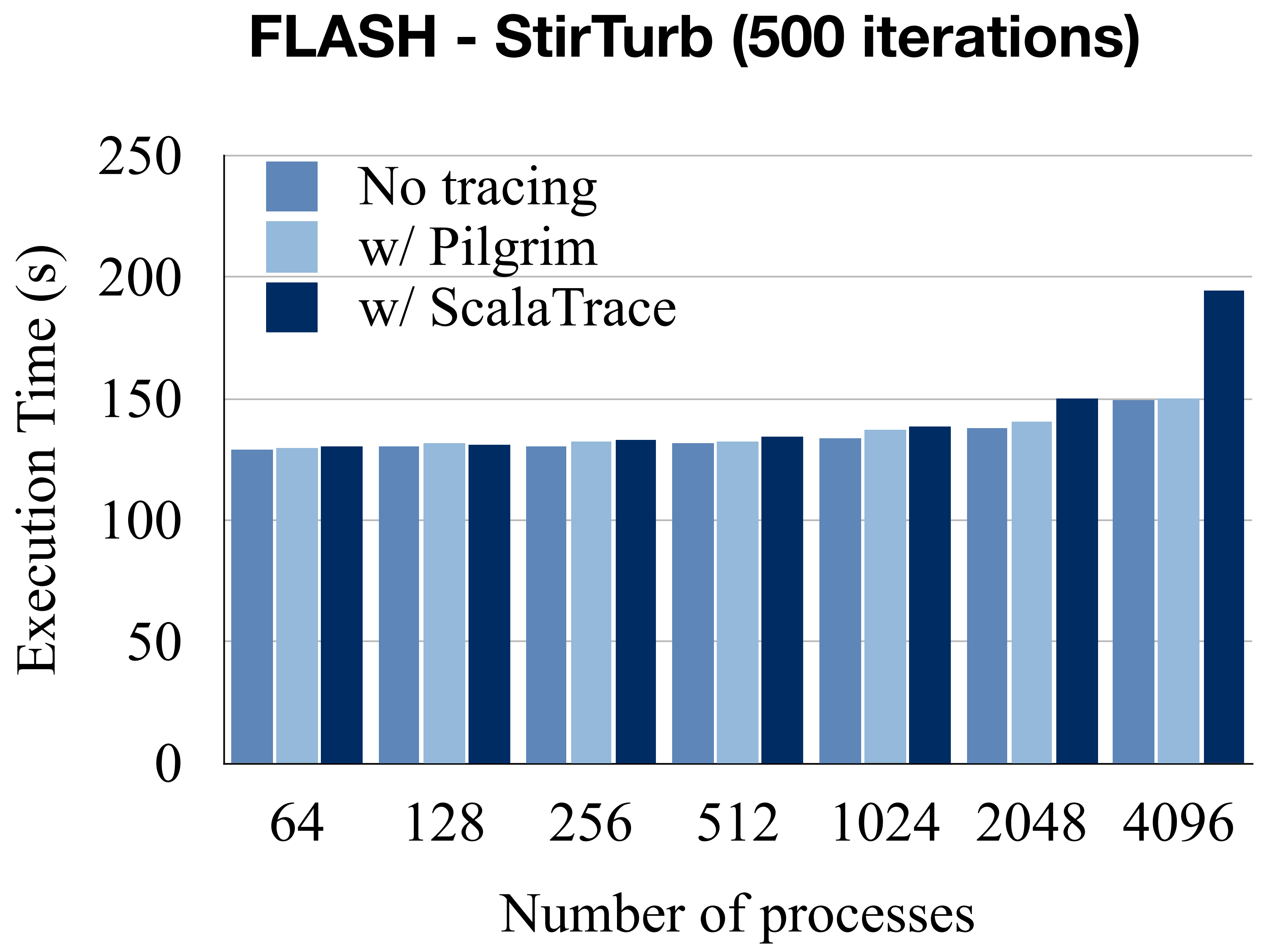
Evaluation

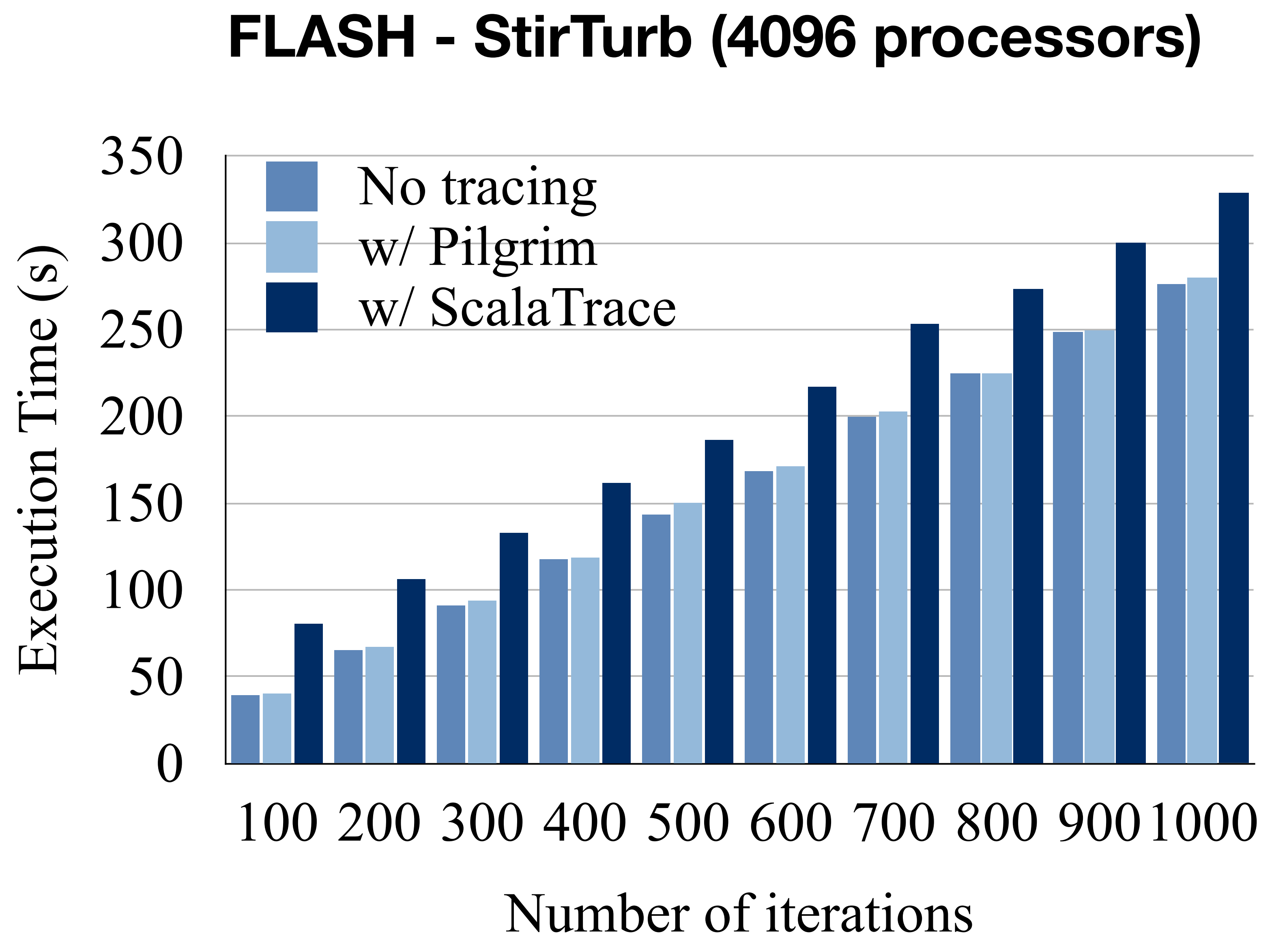
Overhead

- Maximum overhead incurred:
  - 21% for Sedov
  - 4% for StirTurb
- Components:
  - Intra-process compression
    - MPI Interception
    - Build CFG — ~ 60%
  - Inter-process compression
    - CFG — ~ 30%
    - CST — ~ 1%









Conclusion and Future Work

- **Conclusion:**
  - Pilgrim is a scalable and (near) lossless tracing tool
    - We keep more information with less space
  - For regular communication patterns, Pilgrim can store the lossless MPI information in constant space regardless the number of iterations and the number of processes.
    - e.g., 600KB for MILC with 16K processes
- **Future Work:**
  - Further optimize code to reduce the overhead
  - Better compression for “slowly evolving irregular codes” (AMR)
  - Better time encoding to avoid drift
  - Detect non-linear communication patterns
  - Mini-app auto-generator (mostly done)
- Pilgrim is publicly available at https://github.com/pmodels/pilgrim

Thanks!
Questions?

Contact: Chen Wang (chenw5@illinois.edu)