

ComPar: Optimized Multi-Compiler for Automatic OpenMP S2S Parallelization

Idan Mosseri, Lee-or Alon, Re'em Harel and Gal Oren



Bar-Ilan University



הוועדה לאנרגיה אטומית
Israel Atomic Energy Commission



From Sequential to Parallel

Excellent performance in theory, time consuming in practice

One must have a **deep understanding of the code** and be very cautious **not to change the inner logic**

To fully exploit these systems, the code has to be adjusted

Automatic S2S Parallelization compilers

Growing usage in multi-core architectures

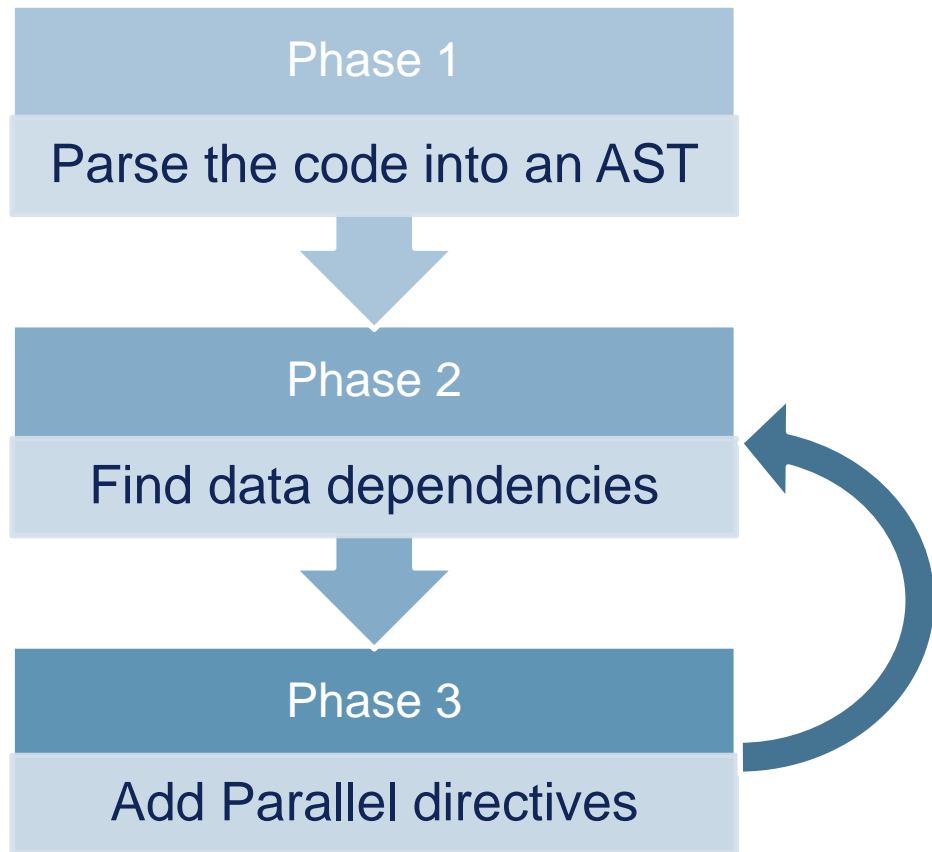
From wearable devices through personal computers to HPC

To ease this difficult process, automatic S2S compilers were introduced

From Sequential to Parallel

Automatic source-to-source parallelization compilers

How does it work?





From Sequential to Parallel

Automatic source-to-source
parallelization compilers

There is NO
best S2S automatic
parallelization
compiler

Each has its
advantages and
disadvantages

Currently, no
existing automatic
parallelization
compiler can fully
replace the
programmer's
insight



Compilers Comparison

There is no best
S2S automatic
parallelization
compiler

In [1], we concluded
that the most suitable
compilers for our task
would be AutoPar,
Par4All and Cetus

- Mostly because they are
free up-to-date S2S
compilers

Other S2S automatic
parallelization
compilers can be
easily added to
ComPar

- It just needs to
implement ComPar's API



AutoPar

Background

- Developed by Lawrence Livermore National Laboratory
- For C and C++ programs
- A module within the ROSE compiler
- Open-source

(main) Pros

- Inherently suitable for OOP
- Handles nested loops
- Verifies existing OMP directives in a given code
- Can be directed to add OMP directives regardless of errors
- Modifications are accompanied by explanation in its output

(main) Cons

- Requires programmer intervention to handle function side-effects etc
- Lacks the ability to tune the parallelization directives for each level in the nested loop
- May add incorrect OpenMP directives when given the "No-aliasing" option



Par4All

Background

- Developed by SILKAN, MINES ParisTech, and Institut Télécom
- For C and Fortran programs
- Open-source
- Its development was shut-down by 2015.

(main) Pros

- Automatically analyzes function side effects and pointer aliasing
- Suitable for GPUs
- Supports many data types
- Supports Fortran (hence more suitable for scientific legacy codes)

(main) Cons

- May change the code structure
- Unused functions will not be parallelized

Background

- Developed by ParaMount research group at Purdue University
- For C programs
- Open-source
- Contains a GUI and a client-server model

(main) Pros

- Handles nested loops
- Provides cross-platform interface
- Verifies existing OMP directives in a given code
- Modifications are accompanied by explanation in its output
- Loop size dependent parallelization

(main) Cons

- Adds its own pragmas which create excess code
- May create reduction clauses that are unknown for standard compilers
- Does not insert OMP directives to loops that contain function calls

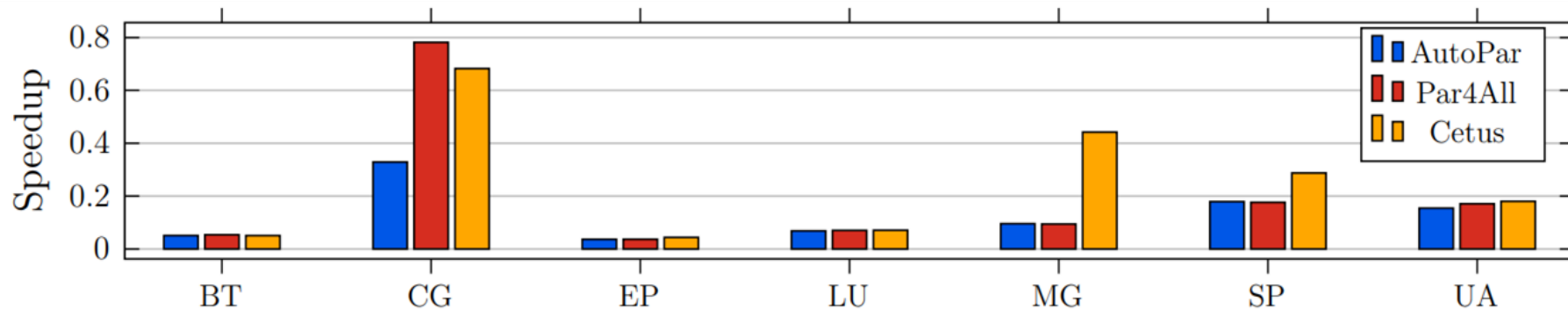


Compilers Comparison

Feature	AutoPar (ROSE)	Par4All (PIPS)	Cetus
Loop unrolling	No	Yes	Yes
Supported languages	C, C++	C, Fortran, CUDA	C
"No-aliasing" option	Yes	Yes	Yes
Check alias dependence	No	Yes	Yes
Reduction clauses	Yes	Yes	Yes
Array reduction/privatization	No	No	Yes
Nested loops	Yes	No	Yes
Function side effect	Annotation required	Yes	Yes
OOP compatible	Yes	No	No
Development status	Yes	No	Yes

NAS Parallel Benchmarks

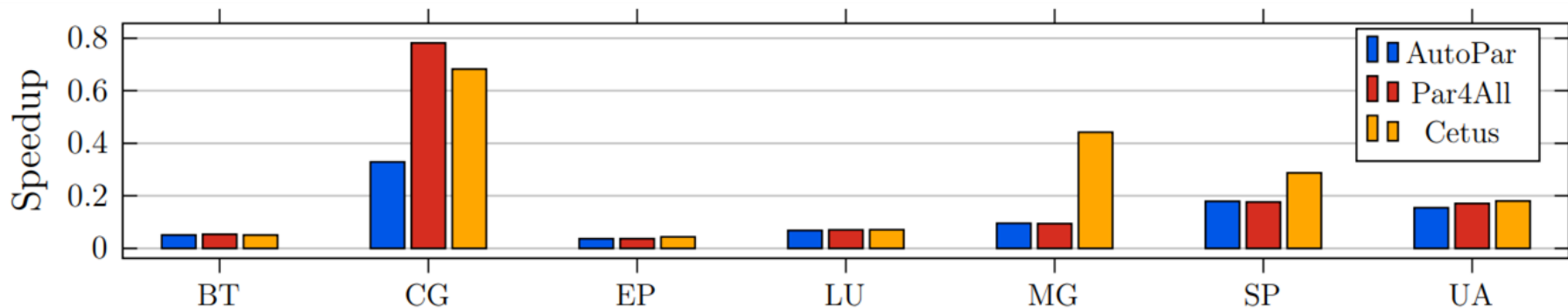
- Numerical Aerodynamics Simulations (NAS) Parallel Benchmarks
- Developed by NASA
- Evaluate the performance of HPC



NAS Parallel Benchmarks

- **There is a compiler for a suitable-for-parallelization individual segment**
- Using only one compiler at a time is not enough to fully exploit the hardware capabilities to the limit
- Carefully fuse the abilities of all compilers

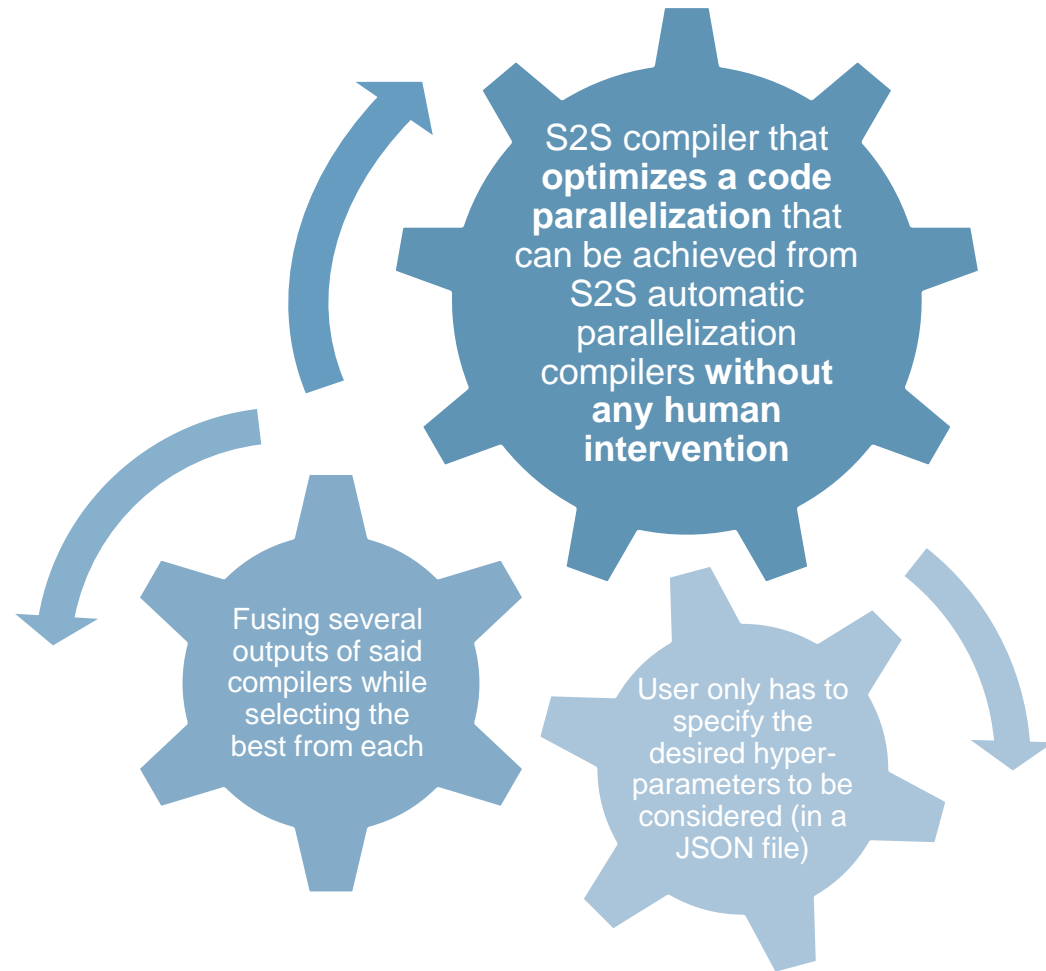
There is no best compiler for an entire program





ComPar

Enjoy the best of all worlds

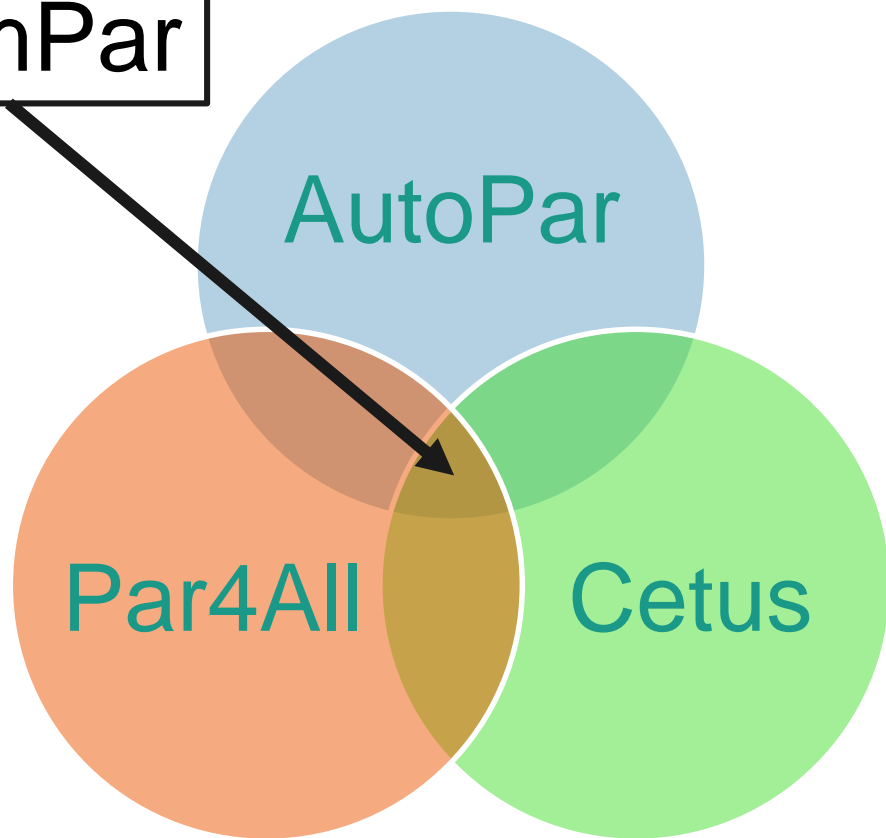




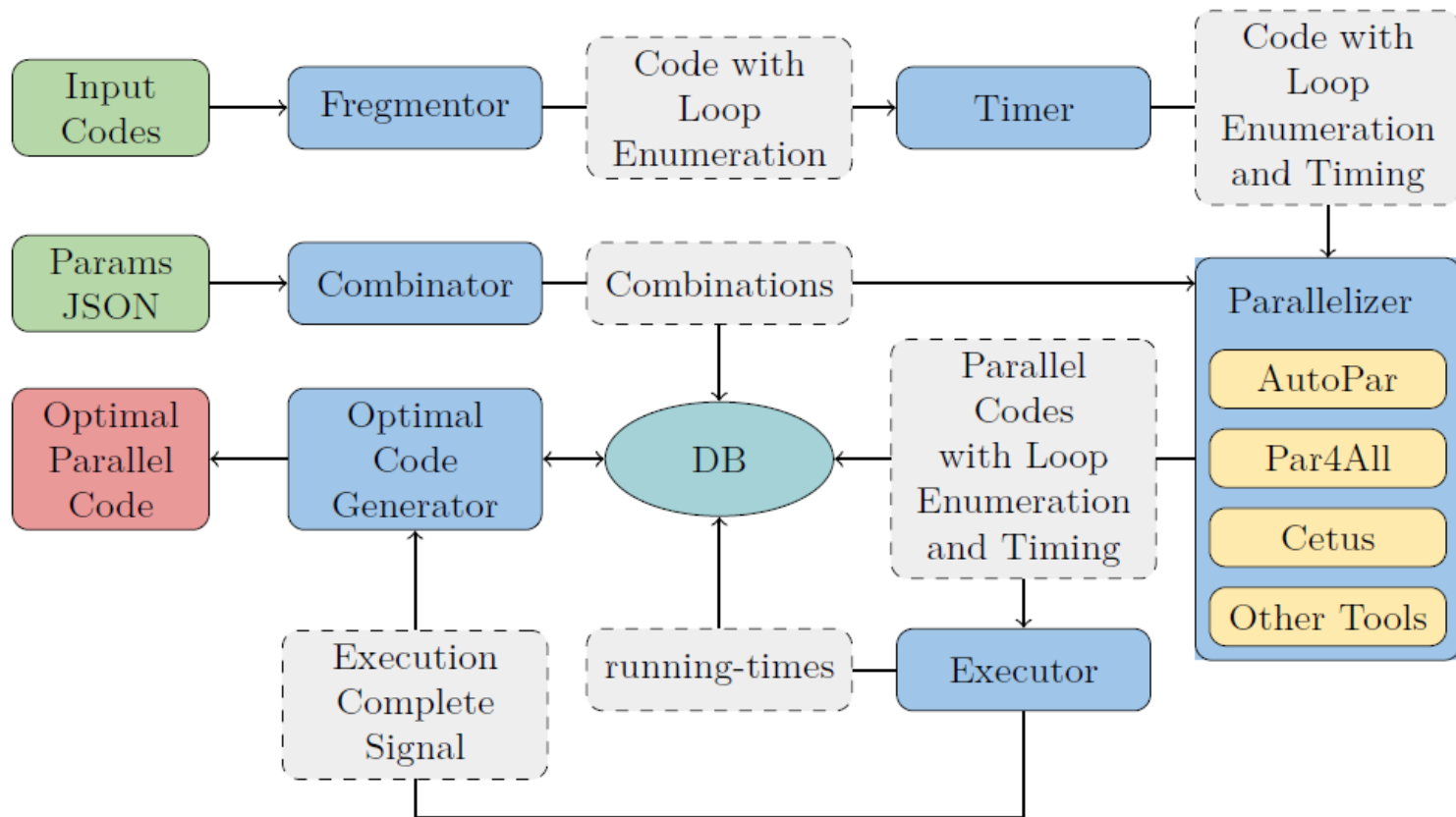
ComPar

In other words

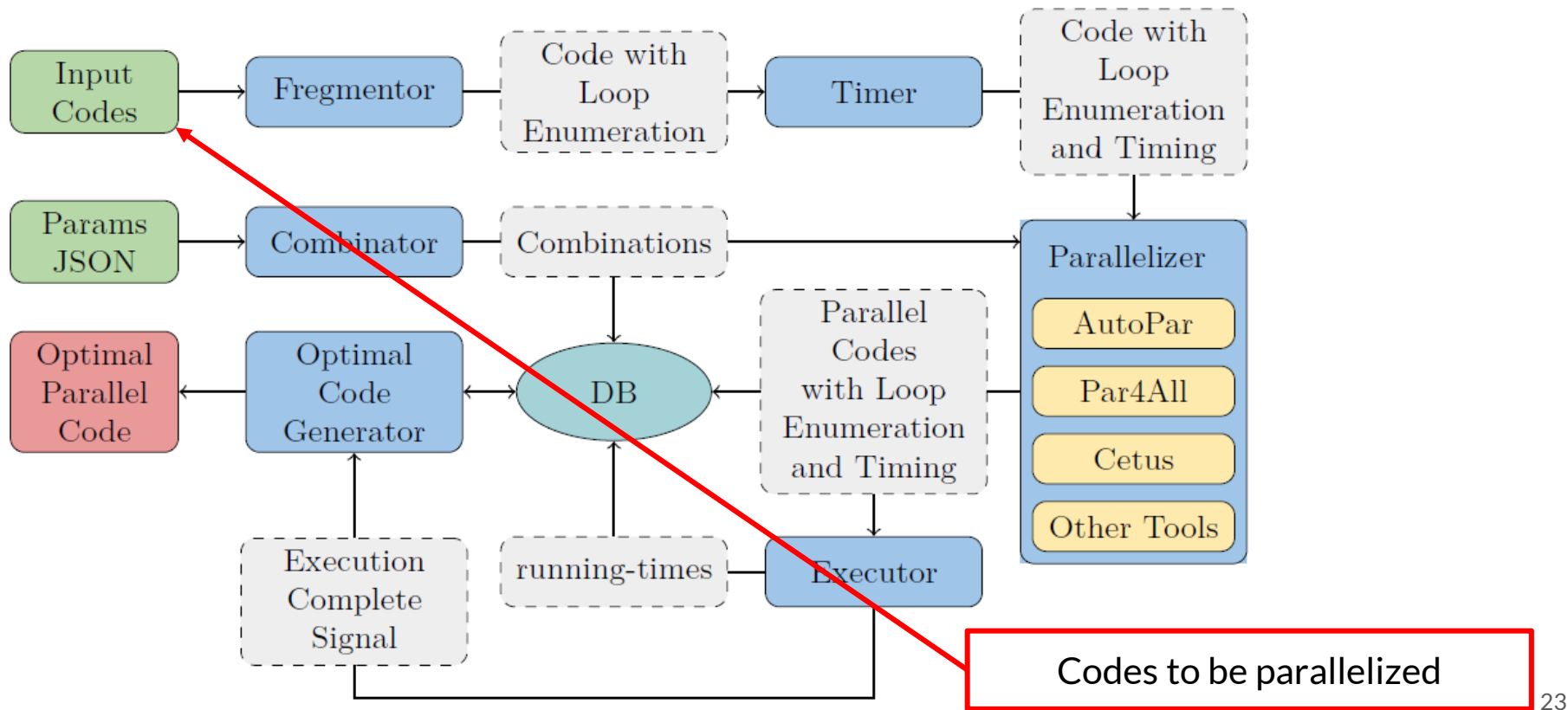
ComPar



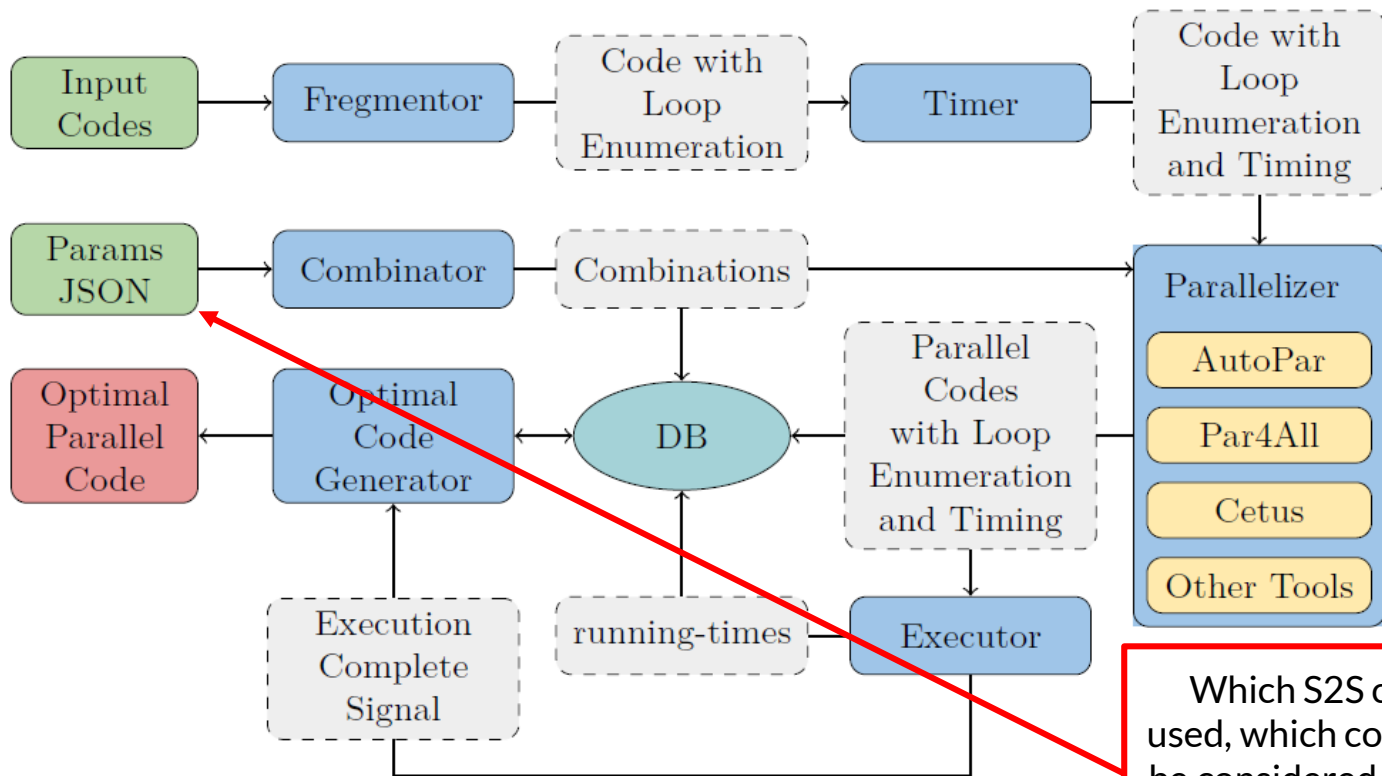
How does it Work?



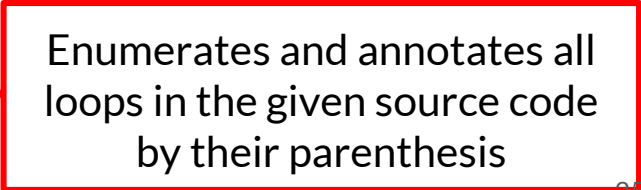
How does it Work?



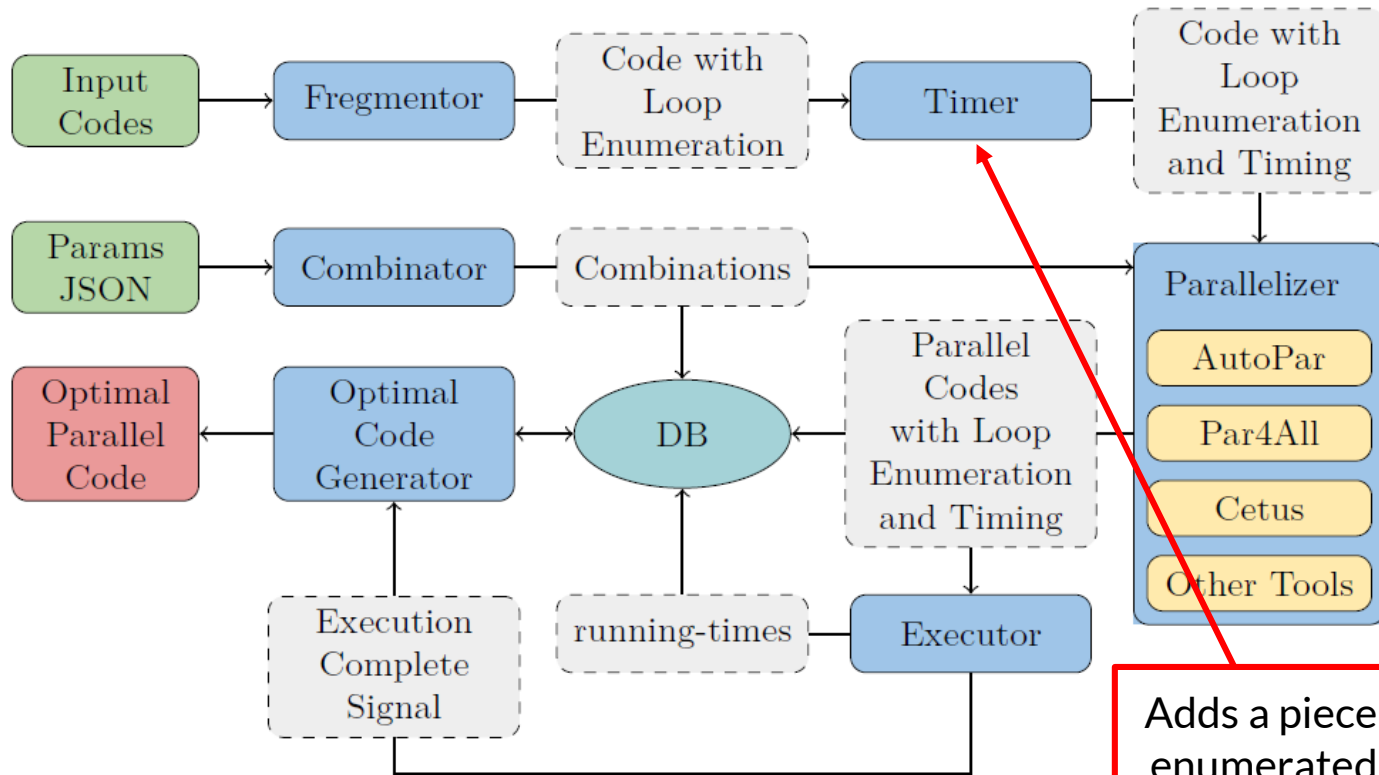
How does it Work?



Which S2S compilers should be used, which compilation flags should be considered for each compiler and which OpenMP directives and RTLs should ComPar consider

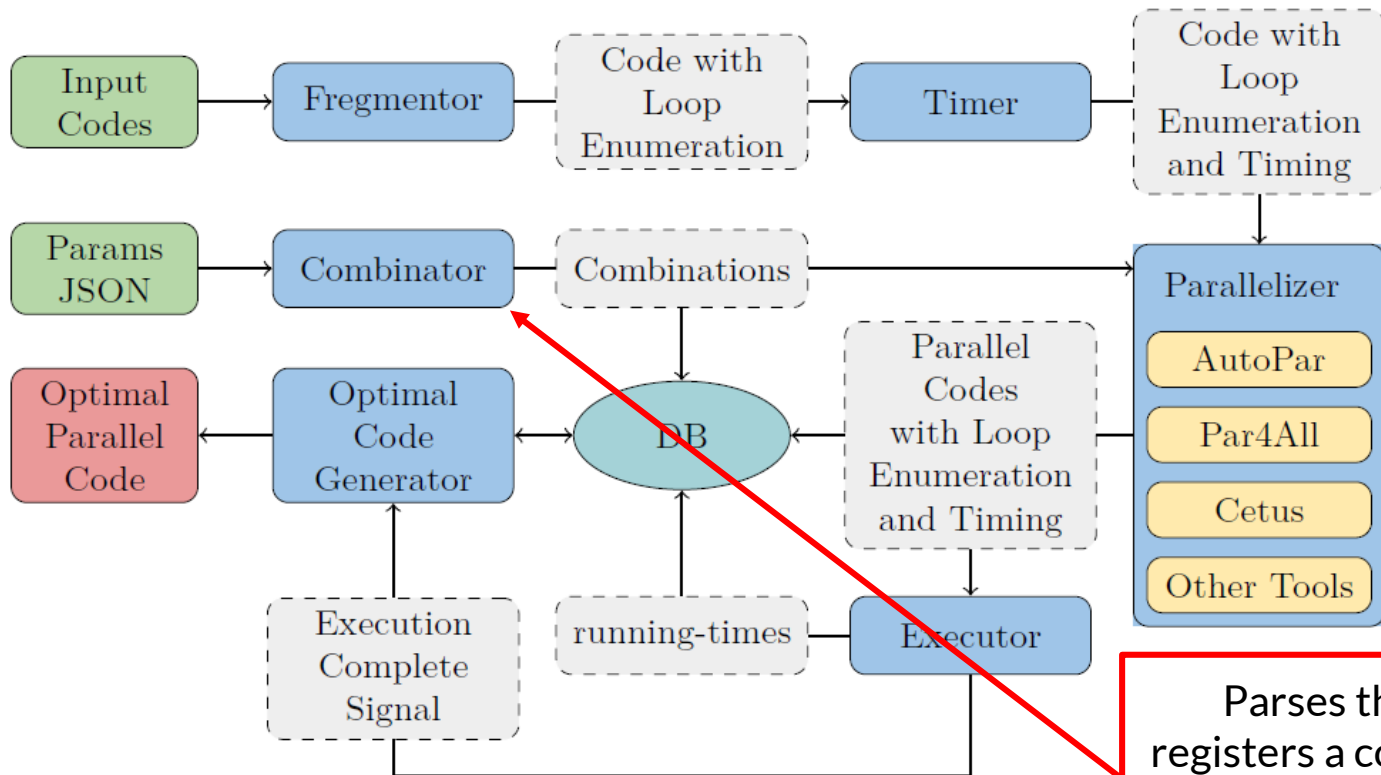


How does it Work?



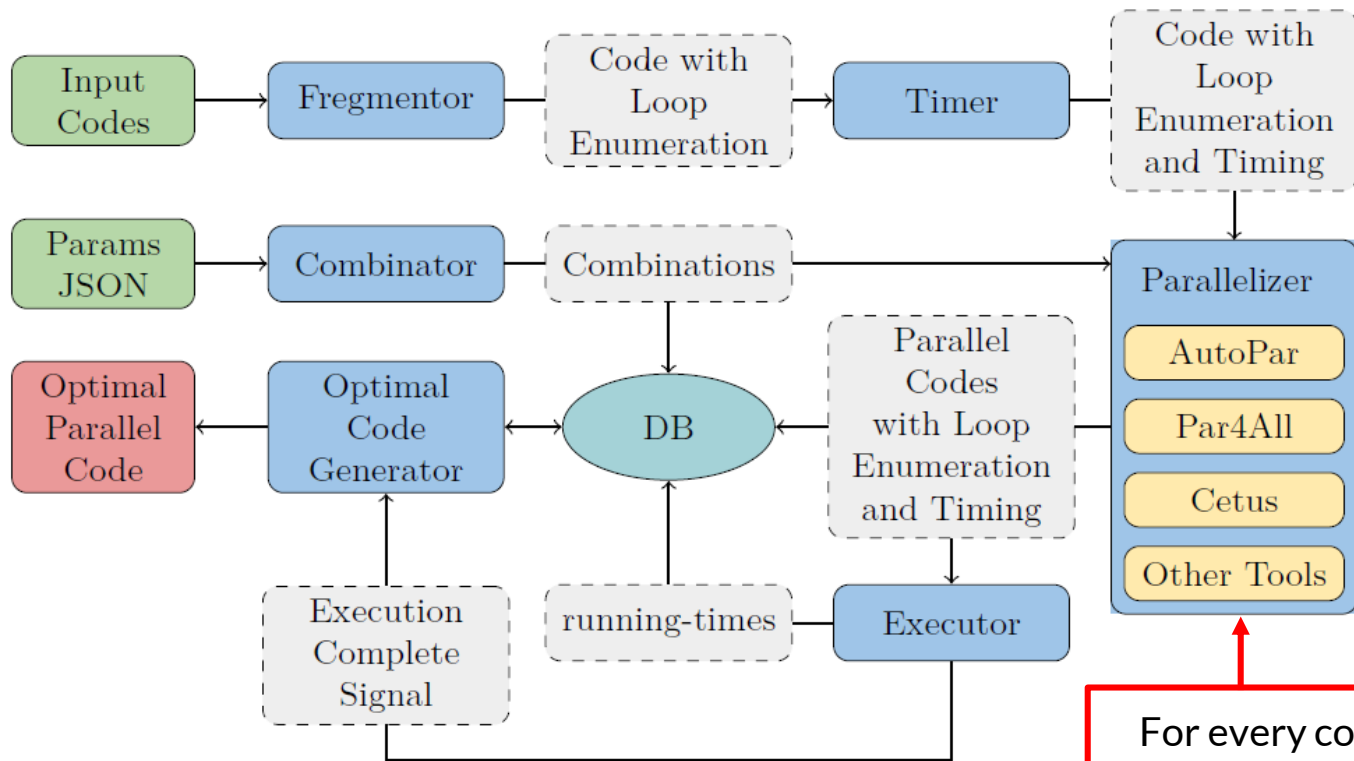
Adds a piece of code around each enumerated loop (will be used to measure its execution time)

How does it Work?



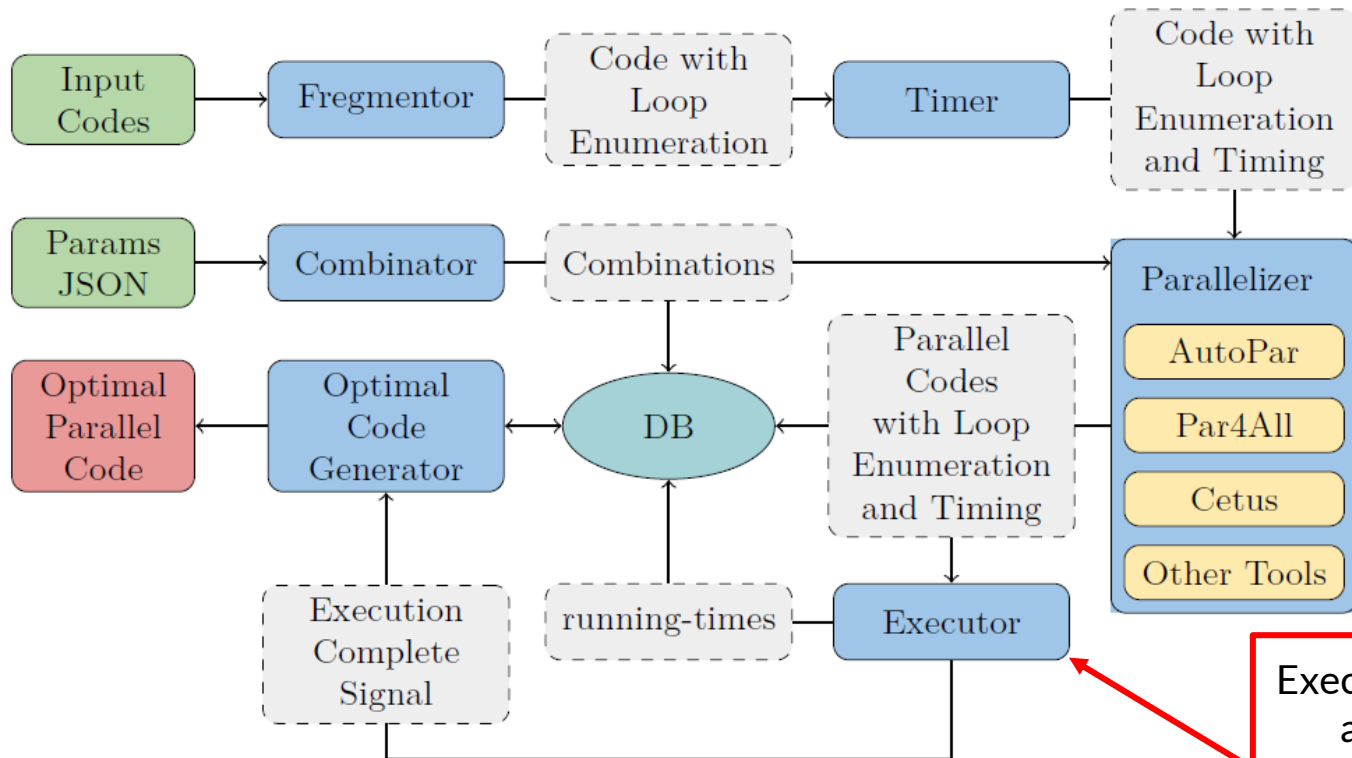
Parses the JSON files and registers a combination in the DB for each possible permutation of the parameters from the file

How does it Work?



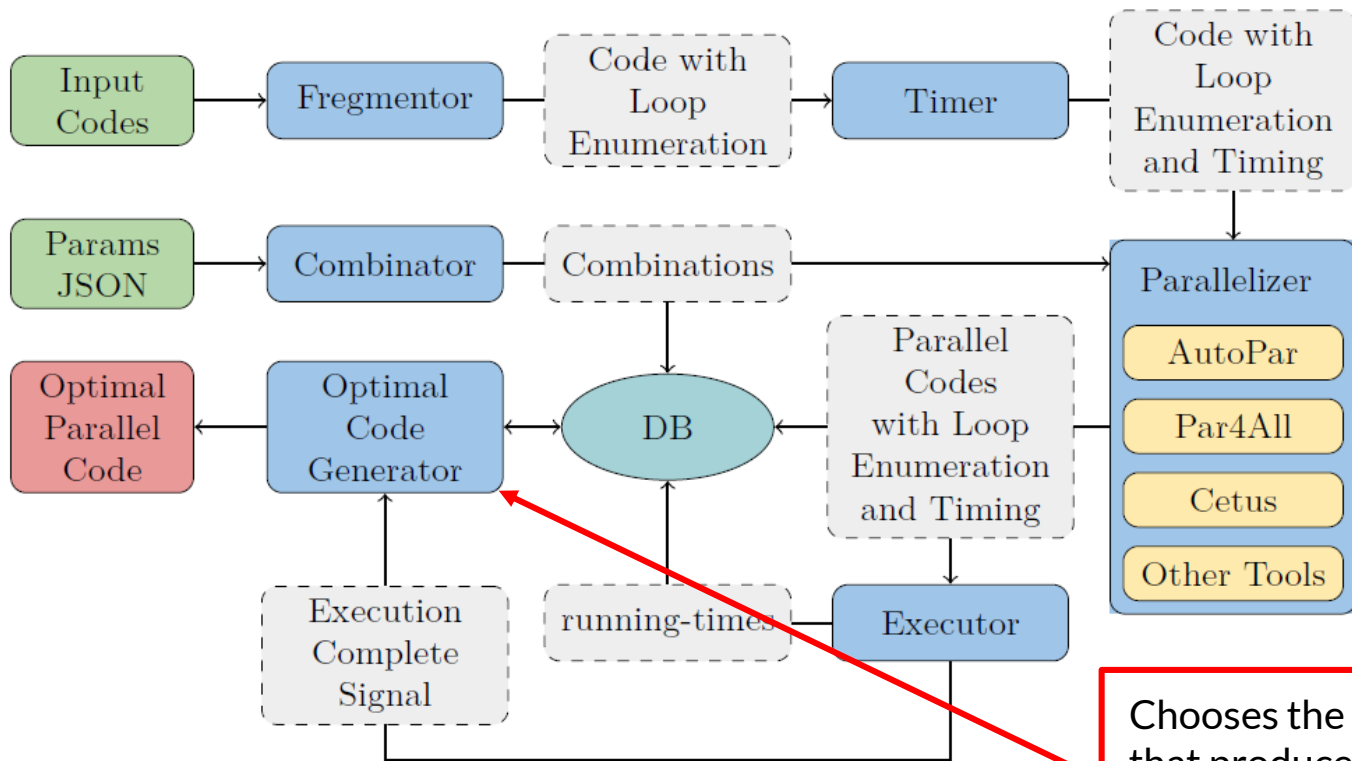
For every combination, parallelizes the code with the compiler and flags specified by the combination. Then, adds the directive clauses and RTLs

How does it Work?



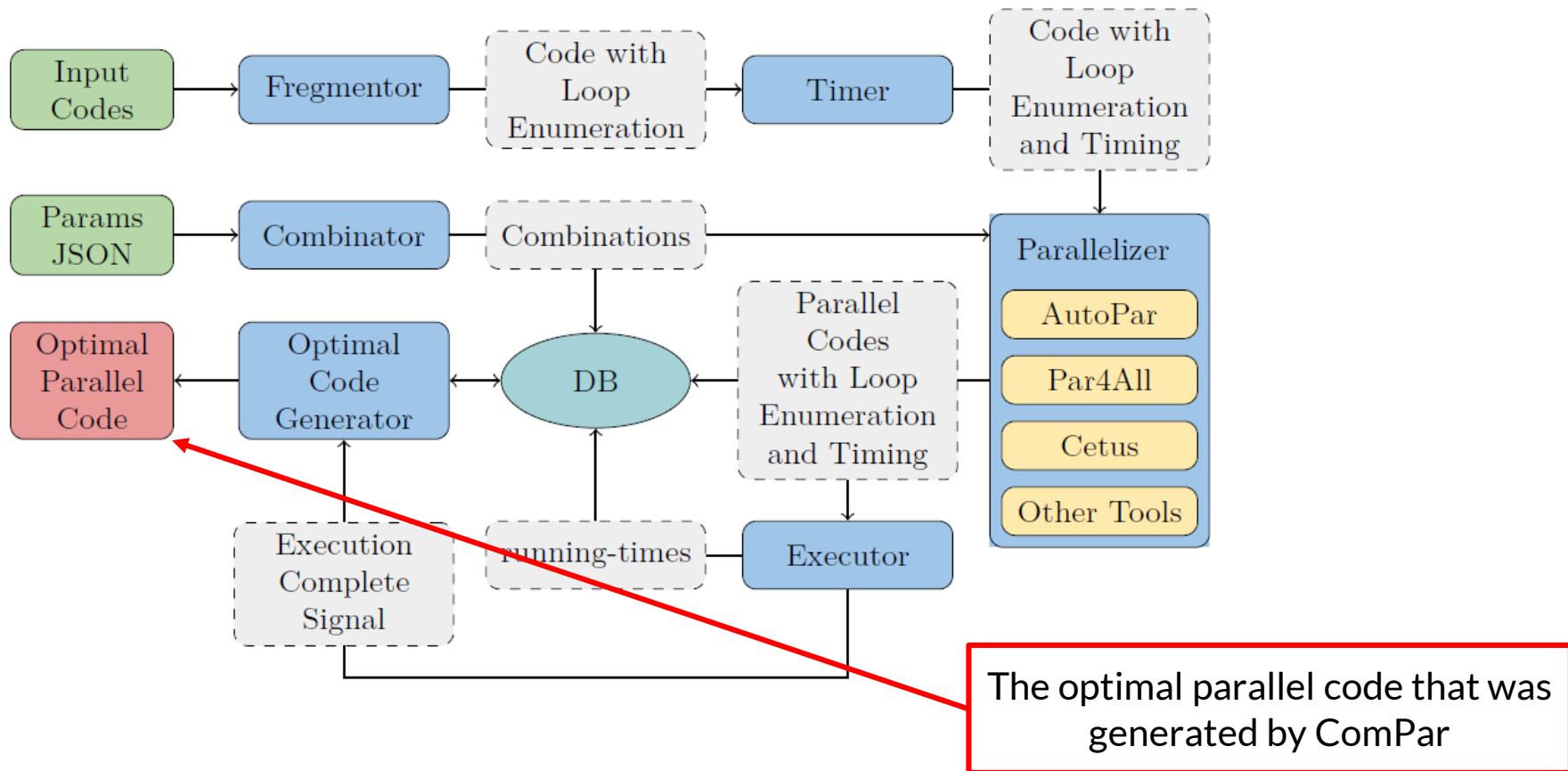
Executes each parallel code and logs code's total runtime and the runtimes of each of its loops in the DB

How does it Work?



Chooses the parallelization scheme that produced the shortest runtime across all combinations for every loop and fuses them

How does it Work?





ComPar

Correctness of the
generated program

- To validate the correctness of the generated code, ComPar uses **black-box testing**
 - Examines the application before and after the parallelization without peering into its internal structures or workings
- ComPar **rejects any combination that did not pass the tests**

Look at it Go

Listing 1 Daxpy Serial Code

```
1 void init(double *x, double *y, int N){
2     for (int i=0; i<N; ++i) {
3         x[i] = i*0.3; y[i] = i*0.2;}
4     }
5     int main() {
6         double *x, *y, alpha = 3.0;
7         int N = 1024*1024*256, R = 64, k,i,g;
8         x = (double *) malloc (N *
9             ↳ sizeof(double));
10        y = (double *) malloc (N *
11            ↳ sizeof(double));
12        init(x,y,N);
13        for(k=0; k<R; k++)
14            for (i=0; i<N; i++)
15                y[i] = alpha * x[i];}
```

Listing 2 ComPar's Parallel Optimized Daxpy

```
1 void init(double *x, double *y, int N) {
2     // START_LOOP_MARKER1 | COMB_ID: HASH#1 | COMPILER_NAME: autopar
3     #pragma omp parallel for firstprivate(N)
4     for (int i = 0; i <= N - 1; i += 1) {
5         x[i] = i * 0.3; y[i] = i * 0.2;} // END_LOOP_MARKER1
6     }
7     int main() {
8         double *x, *y, alpha = 3.0;
9         int N = 1024 * 1024 * 256, R = 64, k, i, g;
10        x = (double *)malloc(N * sizeof(double));
11        y = (double *)malloc(N * sizeof(double));
12        init(x, y, N);
13        // START_LOOP_MARKER2 | COMB_ID: HASH#2 | COMPILER_NAME: cetus
14        #pragma cetus firstprivate(y) private(i, k) lastprivate(y)
15        #pragma loop name main #0
16        #pragma cetus parallel
17        #pragma omp parallel for if ((10000 < ((1L + (3L * R)) + ((3L * N) *
18            ↳ R)))) private(i, k) firstprivate(y) lastprivate(y)
19        for (k = 0; k < R; k++) {
20            #pragma cetus private(i)
21            #pragma loop name main #0 #0
22            #pragma cetus parallel
23            #pragma omp parallel for if ((10000 < (1L + (3L * N))))
24            ↳ private(i)
25            for (i = 0; i < N; i++) {
26                y[i] = (alpha * x[i]);} // END_LOOP_MARKER2
27        }
```

Look at it Go

Listing 1 Daxpy Serial Code

```
1 void init(double *x, double *y, int N){
2     for (int i=0; i<N; ++i) {
3         x[i] = i*0.3; y[i] = i*0.2;}
4     }
5     int main() {
6         double *x, *y, alpha = 3.0;
7         int N = 1024*1024*256, R = 64, k,i,g;
8         x = (double *) malloc (N *
9             ↳ sizeof(double));
10        y = (double *) malloc (N *
11            ↳ sizeof(double));
12        init(x,y,N);
13        for(k=0; k<R; k++)
14            for (i=0; i<N; i++)
15                y[i] = alpha * x[i];}
```

Listing 2 ComPar's Parallel Optimized Daxpy

```
1 void init(double *x, double *y, int N) {
2     // START_LOOP_MARKER1 | COMB_ID: HASH#1 | COMPILER_NAME: autopar
3     #pragma omp parallel for firstprivate(N)
4     for (int i = 0; i <= N - 1; i += 1) {
5         x[i] = i * 0.3; y[i] = i * 0.2;} // END_LOOP_MARKER1
6     }
7     int main() {
8         double *x, *y, alpha = 3.0;
9         int N = 1024 * 1024 * 256, R = 64, k, i, g;
10        x = (double *)malloc(N * sizeof(double));
11        y = (double *)malloc(N * sizeof(double));
12        init(x, y, N);
13        // START_LOOP_MARKER2 | COMB_ID: HASH#2 | COMPILER_NAME: cetus
14        #pragma cetus firstprivate(y) private(i, k) lastprivate(y)
15        #pragma loop name main #0
16        #pragma cetus parallel
17        #pragma omp parallel for if ((10000 < ((1L + (3L * R)) + ((3L * N) *
18            ↳ R)))) private(i, k) firstprivate(y) lastprivate(y)
19        for (k = 0; k < R; k++) {
20            #pragma cetus private(i)
21            #pragma loop name main #0 #0
22            #pragma cetus parallel
23            #pragma omp parallel for if ((10000 < (1L + (3L * N))))
24            ↳ private(i)
25            for (i = 0; i < N; i++) {
26                y[i] = (alpha * x[i]);} // END_LOOP_MARKER2
27        }
```

Look at it Go

Listing 1 Daxpy Serial Code

```
1 void init(double *x, double *y, int N){
2     for (int i=0; i<N; ++i) {
3         x[i] = i*0.3; y[i] = i*0.2;}
4     }
5     int main() {
6         double *x, *y, alpha = 3.0;
7         int N = 1024*1024*256, R = 64, k,i,g;
8         x = (double *) malloc (N *
9             ↳ sizeof(double));
10        y = (double *) malloc (N *
11            ↳ sizeof(double));
12        init(x,y,N);
13        for(k=0; k<R; k++)
14            for (i=0; i<N; i++)
15                y[i] = alpha * x[i];}
```

Listing 2 ComPar's Parallel Optimized Daxpy

```
1 void init(double *x, double *y, int N) {
2     // START_LOOP_MARKER1 | COMB_ID: HASH#1 | COMPILER_NAME: autopar
3     #pragma omp parallel for firstprivate(N)
4     for (int i = 0; i <= N - 1; i += 1) {
5         x[i] = i * 0.3; y[i] = i * 0.2;} // END_LOOP_MARKER1
6     }
7     int main() {
8         double *x, *y, alpha = 3.0;
9         int N = 1024 * 1024 * 256, R = 64, k, i, g;
10        x = (double *)malloc(N * sizeof(double));
11        y = (double *)malloc(N * sizeof(double));
12        init(x, y, N);
13        // START_LOOP_MARKER2 | COMB_ID: HASH#2 | COMPILER_NAME: cetus
14        #pragma cetus firstprivate(y) private(i, k) lastprivate(y)
15        #pragma loop name main #0
16        #pragma cetus parallel
17        #pragma omp parallel for if ((10000 < ((1L + (3L * R)) + ((3L * N) *
18            ↳ R)))) private(i, k) firstprivate(y) lastprivate(y)
19        for (k = 0; k < R; k++) {
20            #pragma cetus private(i)
21            #pragma loop name main #0 #0
22            #pragma cetus parallel
23            #pragma omp parallel for if ((10000 < (1L + (3L * N))))
24            ↳ private(i)
25            for (i = 0; i < N; i++) {
26                y[i] = (alpha * x[i]);} // END_LOOP_MARKER2
27        }
```

Look at it Go

File	daxpy.c	
Comb'	<i>HASH#1</i>	<i>HASH#2</i>
Comp'	Autopar	Cetus
Comp' Flags	keep_going no_aliasing	parallelize-loops=2 privatize=2 alias=3
Runtime	0.11	1.99
Speedup	16.62	26.96
Total (sec)	2.21	

Listing 2 ComPar's Parallel Optimized Daxpy

```

1 void init(double *x, double *y, int N) {
2     // START_LOOP_MARKER1 | COMB_ID: HASH#1 | COMPILER_NAME: autopar
3     #pragma omp parallel for firstprivate(N)
4     for (int i = 0; i <= N - 1; i += 1) {
5         x[i] = i * 0.3; y[i] = i * 0.2;} // END_LOOP_MARKER1
6 }
7
8 int main() {
9     double *x, *y, alpha = 3.0;
10    int N = 1024 * 1024 * 256, R = 64, k, i, g;
11    x = (double *)malloc(N * sizeof(double));
12    y = (double *)malloc(N * sizeof(double));
13    init(x, y, N);
14    // START_LOOP_MARKER2 | COMB_ID: HASH#2 | COMPILER_NAME: cetus
15    #pragma cetus firstprivate(y) private(i, k) lastprivate(y)
16    #pragma loop name main #0
17    #pragma cetus parallel
18    #pragma omp parallel for if ((10000 < ((1L + (3L * R)) + ((3L * N) *
19        ↳ R)))) private(i, k) firstprivate(y) lastprivate(y)
20    for (k = 0; k < R; k++) {
21        #pragma cetus private(i)
22        #pragma loop name main #0 #0
23        #pragma cetus parallel
24        #pragma omp parallel for if ((10000 < (1L + (3L * N))))
25        ↳ private(i)
26        for (i = 0; i < N; i++) {
27            y[i] = (alpha * x[i]);} // END_LOOP_MARKER2
28    }

```



ComPar

At worst case, it will be as good as the most suitable compiler for the given code

ComPar support is limited to the compilers it uses

- ComPar can parallelize over accelerators, because AutoPar can
- The chosen compilers are limited to OMP v2.5, hence ComPar cannot benefit from the advantages of later versions



ComPar

Problem size

ComPar's runtime

Runtime depends on the runtime of the given source code

Choose problem size

It would be wise to choose a 'sweet-spot'

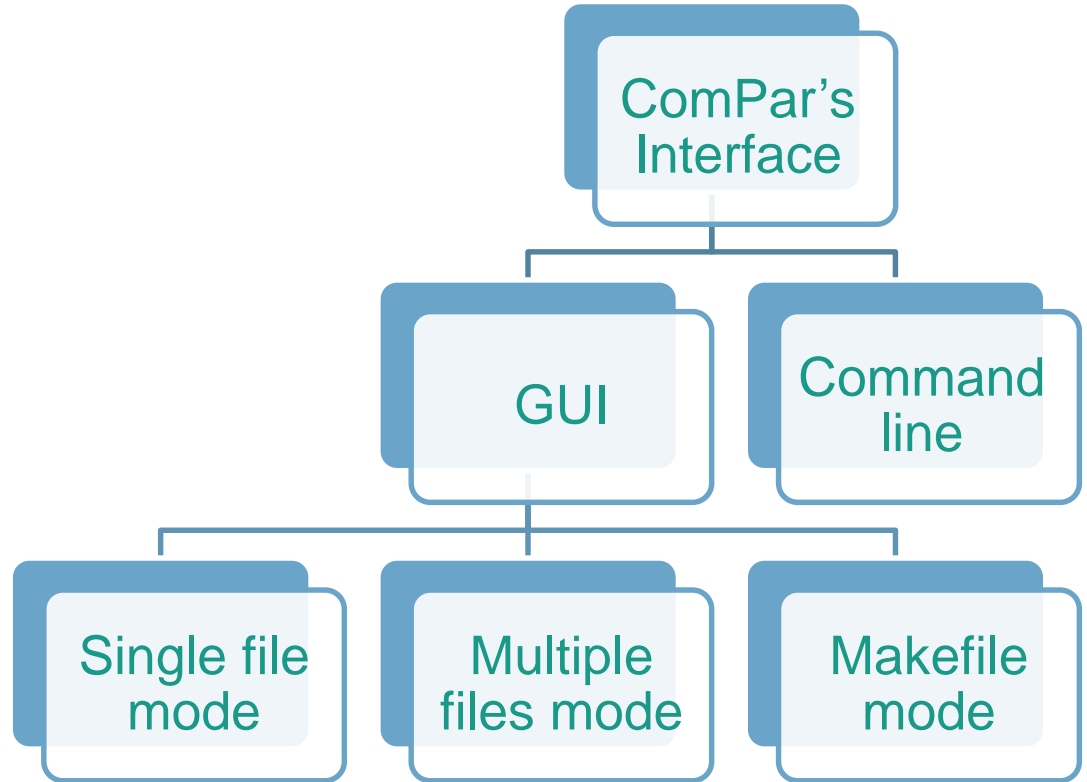
Run real problem size

Then run the real input using the parallel code generated



ComPar

Interface



Single File Mode

ComPar

Single File ModeMultiple Files ModeMakefile Mode

ComPar Parameters

Binary Compiler: GCC

Advanced options >>

Binary compiler flags:

Binary compiler version:

Slurm partition: mixedp

Save combinations folders: ☐

Multiply combinations: 1

Clear database data: ☒

Using ComPar output: ☐

Slurm parameters:

Maximum job count: 4

Execution time limit:

DHMS

Source File

Upload

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4 #include <math.h>
5
6 void initialize_x_and_y(double *x, double *y, int N){
7     for (int i=0; i<N; ++i) {x[i] = i*0.001; y[i] = i*0.03;}
8 }
9
10 int main() {
11
12     double *x, *y;
13     double alpha = 2.0;
14     int N = 100;
15     int R = 2;
16     int k,i;
17     double t_start, t_end, t_total;
18
19     x = (double *) malloc (N * sizeof(double));
20     y = (double *) malloc (N * sizeof(double));
```

Output

Download

```
1 Compar in progress ...
```

Progress

- Starting ComPar execution
- CombinationValidator: Checking the existence of test: 'test_output'.
- Running pytest assets/test_output.py::test_output command

Multiple Files Mode

ComPar

Single File ModeMultiple Files ModeMakefile Mode

ComPar Parameters

Input directory:

/home/leeora/auto_parallel

Output directory:

/home/leeora/auto_parallel

Project Name:

gemm

Main c file:

gemm.c

Binary Compiler:

GCC

Advanced options >>

Binary compiler flags:

-include
"/home/leeora/auto_parallel/
c-4.2.1-beta/utilities"
"/home/leeora/auto_parallel/
c-4.2.1-
beta/utilities/polybench.c"

Binary compiler version:

Slurm partition:

mixedp

Output

Compar in progress ...

Progress

- Starting ComPar execution
- CombinationValidator: Checking the existence of test: 'test_output'.
- Running pytest assets/test_output.py::test_output command

Makefile Mode

ComPar

Single File ModeMultiple Files Mode**Makefile Mode**

ComPar Parameters

Multiply combinations: 1

Clear database data: ☐

Using Compar output: ☐

Slurm parameters:

Maximum job count: 4

Execution time limit: D H M S
0 0 0 0

Main file parameters:

Log level:

Verbose

Compar mode:

Overwrite

Validation file path:

Terminate

Output

Compar in progress ...

Progress

- Starting ComPar execution
- CombinationValidator: Checking the existence of test: test_output.
- Running pytest assets/test_output.py:test_output command



Experiments

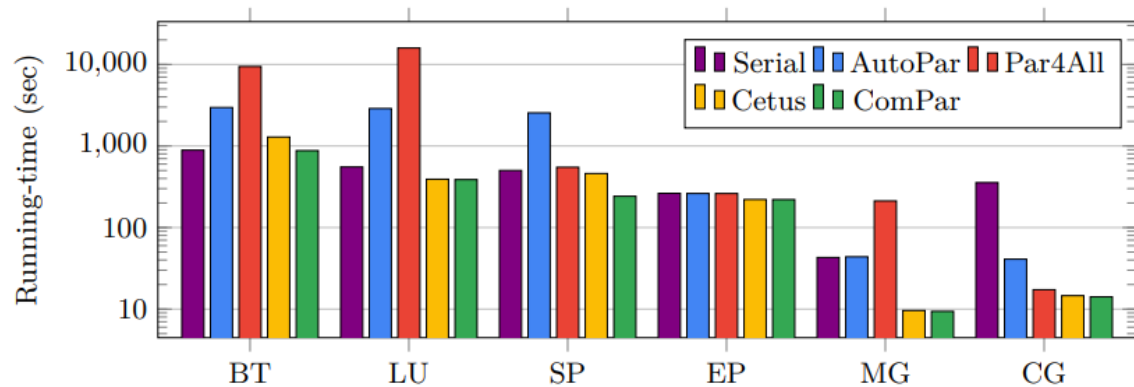
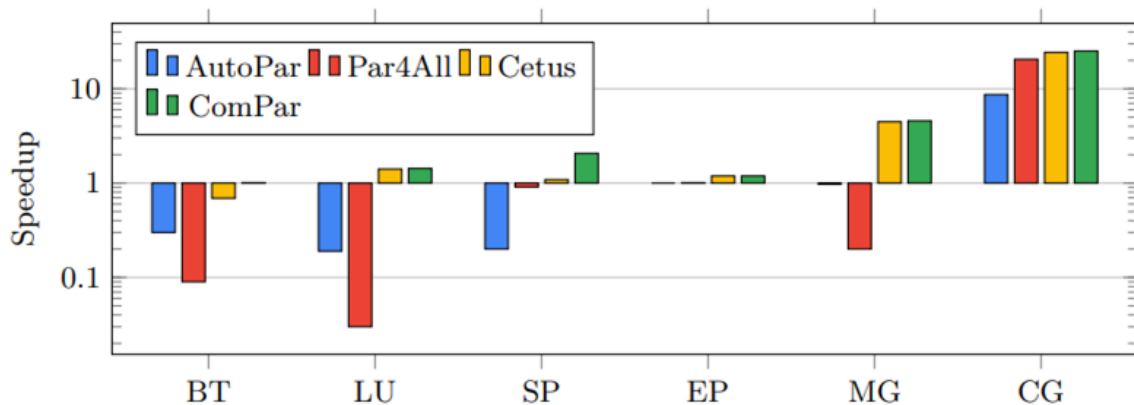
Parameters used in
the experiments

Compilers' Flags	
<i>Compiler</i>	<i>Flag</i>
Cetus	parallelize-loops, reduction, private, alias
AutoPar	keep_going, enable_modeling, no_aliasing, unique_indirect_index
Par4All	O, fine-grain, com-optimization, no-pointer-aliasing
OMP <i>parallel</i> for Directive Clauses	
<i>Clause</i>	<i>Kind</i>
schedule	static [2,4,8,16,32], dynamic
Runtime Library Routines	
<i>RTL Routine</i>	<i>Argument</i>
omp_set_num_threads	2,4,8,16,32

Experiments

NAS

The results are the best each S2S compiler achieved using different flags combinations -- **not a "vanilla" execution**





Experiments

PolyBench

- 30 representative potentially compute-intensive benchmarks
- Attempts to make the kernels' execution as uniform and consistent as possible
- We enlarged the (already LARGE) problem size by x8



Experiments

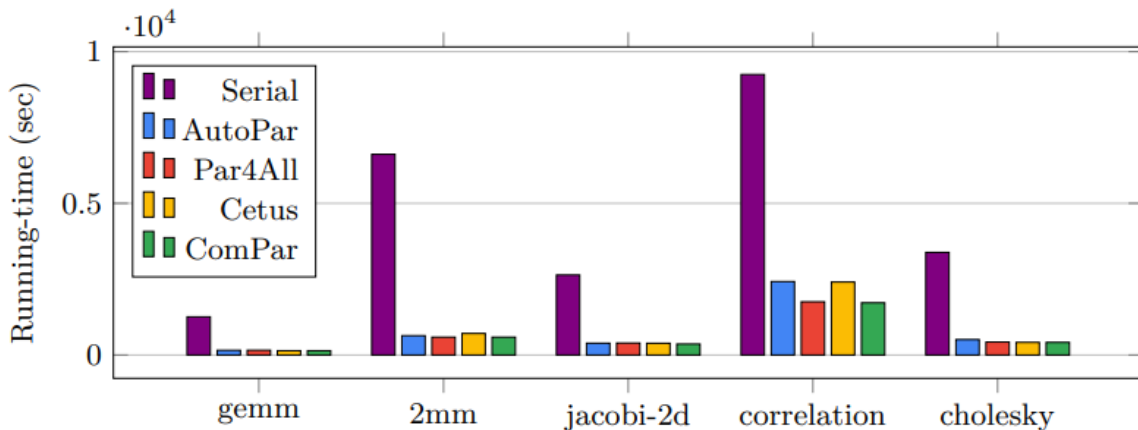
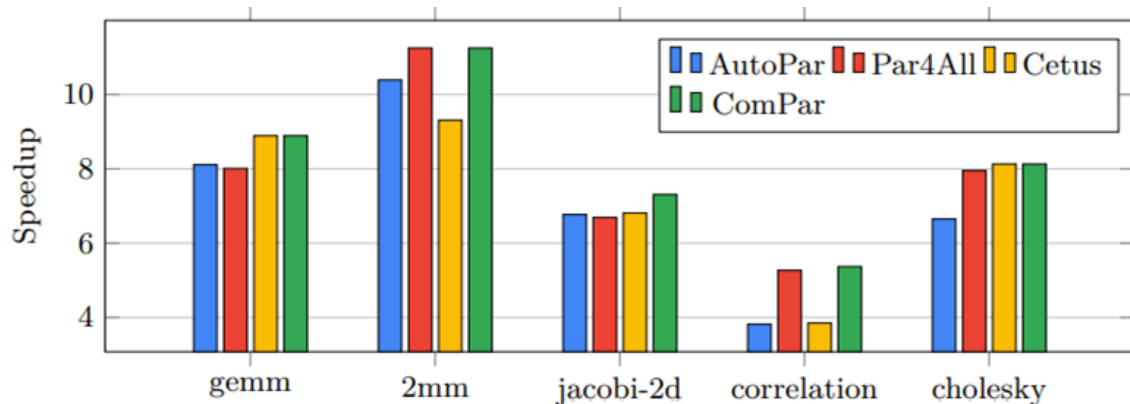
PolyBench

Linear Algebra Kernels		Medley
2mm	3mm	deriche
atax	bicg	floyd-warshall
doitgen	mvt	nussinov
Linear Algebra Solvers		BLAS Routines
cholesky	durbin	gemm
gramschmidt	lu	gemver
ludcmp	trisolv	gesummv
Stencils		symm
adi	ftdt-2d	syrk
heat-3d	jacobi-1d	syr2k
jacobi-2d	seidel-2d	trmm
Data Mining	covariance	correlation

Experiments

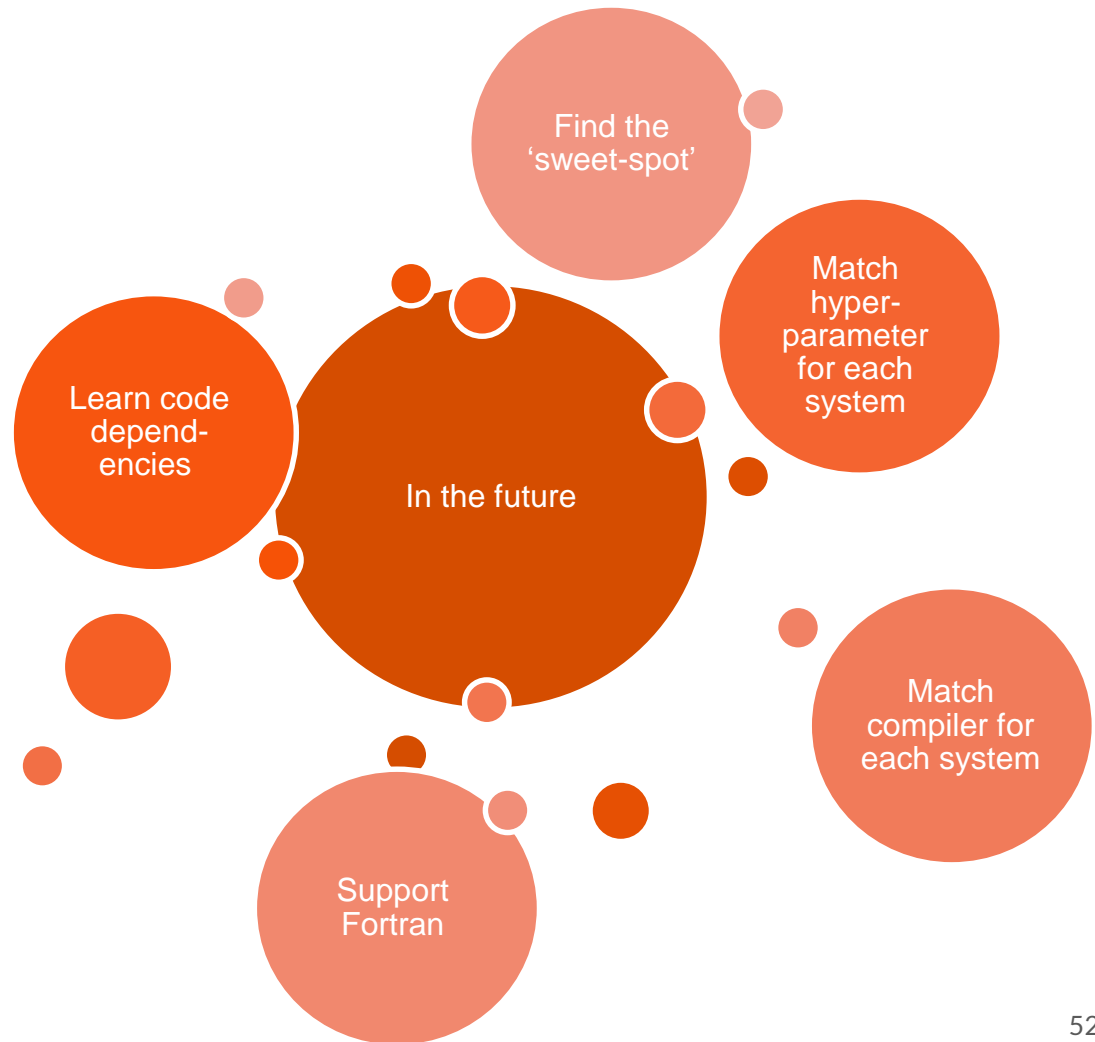
PolyBench

The results are the best each S2S compiler achieved using different flags combinations -- **not a "vanilla" execution**



Future Work

Much work is left...






Summary

What have we seen today?

orenw@post.bgu.ac.il

- 
- To enjoy multi-core architectures, one must adjust its code
 - **Very complicated!**

- 
- To ease this problem, automatic S2S parallelization compilers were introduced

- 
- No compiler is superior to all other compilers in all tests

- 
- Carefully fuse the abilities of all compilers

ComPar allows users to enjoy the advantages of these compilers, while avoiding, when possible, from their disadvantages