ComPar: Optimized Compiler for Automatic OpenMP Source-to-Source Parallelization using Code Segmentation and Hyperparameters Tuning

Idan Mosseri[1,3], Re'em Harel[2,4], Lee-or Alon[2,3], Reuven Regev Farag[5], Gilad Guralnik[5], Yoni Cohen[5], May Hagbi[5], Shlomi Tofahi[5], Yoel Vaizman[5], and Gal Oren*[1,3]

[1] Department of Physics, Nuclear Research Center - Negev, P.O.B. 9001, Be'er-Sheva, Israel
[2] Israel Atomic Energy Commission, P.O.B. 7061, Tel Aviv, Israel
[3] Department of Computer Science, Ben-Gurion University of the Negev, P.O.B. 653, Be'er Sheva, Israel
[4] Department of Physics, Bar-Ilan University, IL52900, Ramat-Gan, Israe.
[5] Department of software engineering, Sami Shamoon College of Engineering, P.O.B. 950, Be'er Sheva, Israel
* Corresponding Author

Introduction

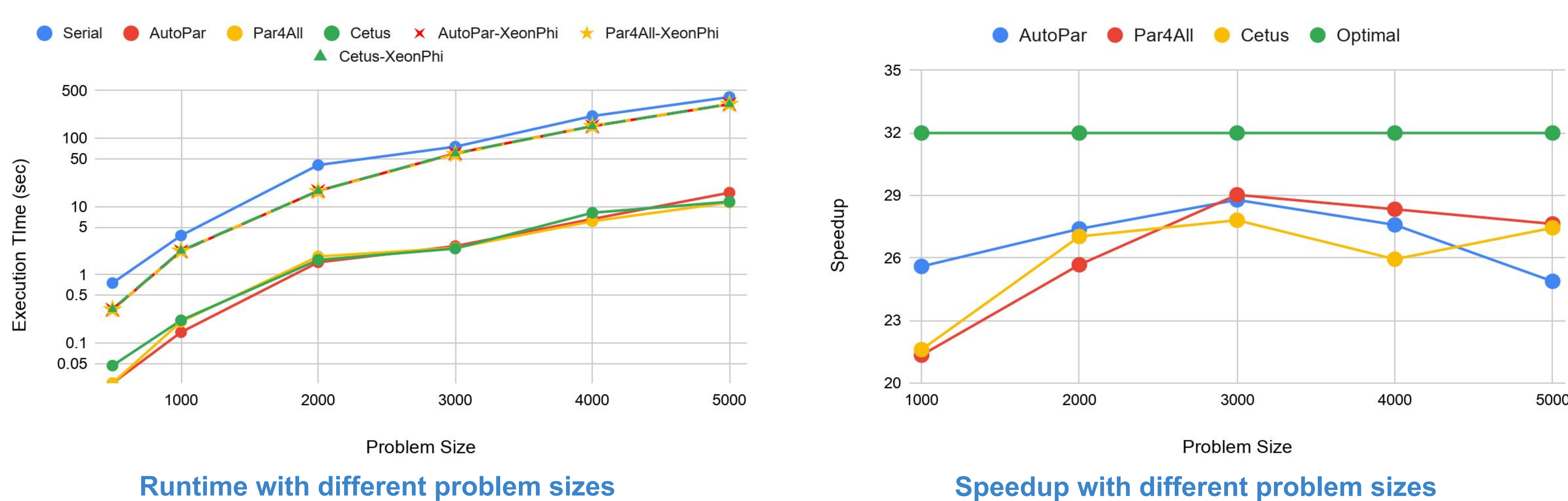
- **Parallelization is essential** to exploit the full benefits of multi-core architectures
- Designing valid parallelization for applications is **not always a simple nor cheap task**
- **Automatic parallelization** source-to-source (S2S) compilers were proposed to **ease this process, while keeping the code readable for the user**
- Each compiler has its pros and cons. We wish to **enjoy the best of every compiler**

There is NO Best Compiler

We compare **AutoPar**, **Par4All** and **Cetus** on different exemplary tests, **each test emphasizing a different** parallel shared memory management **pitfall**

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

Matrix Multiplication Problem



Runtime with different problem sizes — Speedup with different problem sizes

ComPar - Fusion of Optimizations

Each compiler has its advantages and disadvantages (as can be seen from our performance analysis). "Wisly" fusing the compilers' output while further optimizing their performances, should **produce superior results**

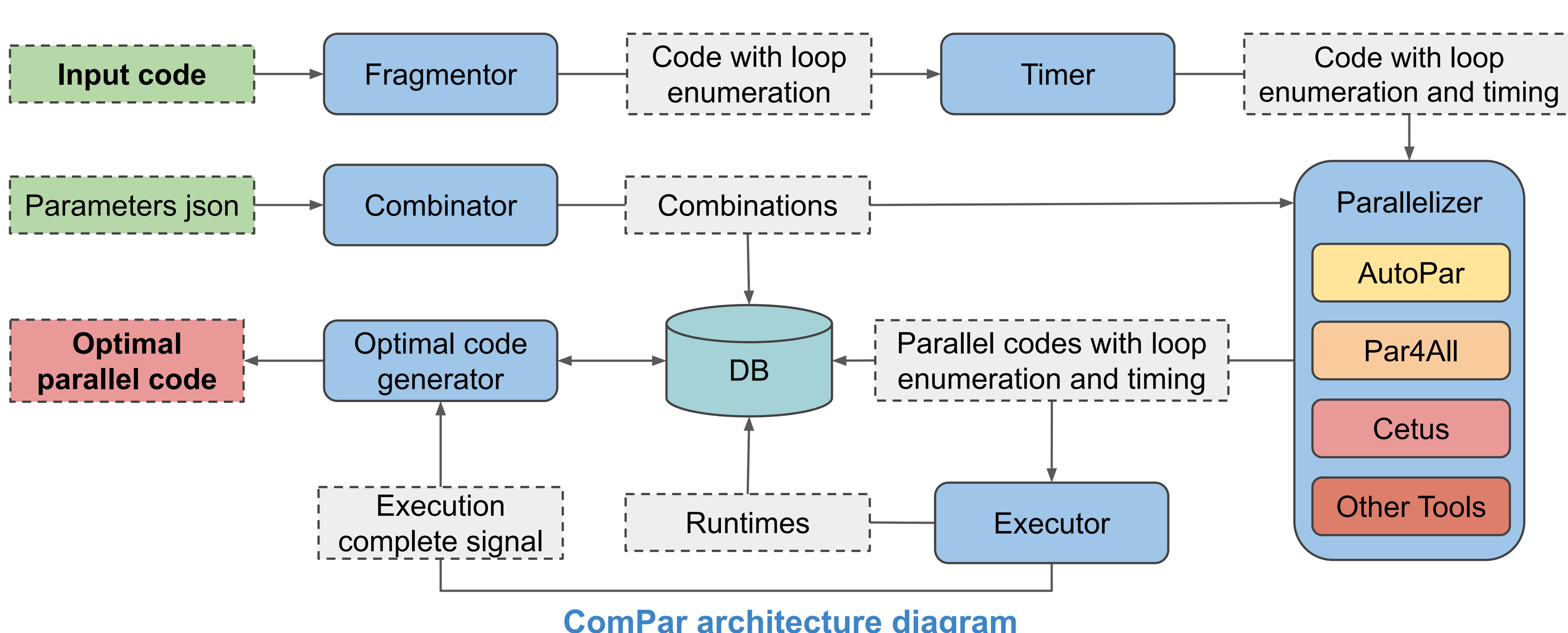
In order to achieve the above objective, we designed and built a new parallelization framework called ***ComPar***, which is based on current S2S automatic parallelizers

- ***ComPar* adapts the automated parallelism** scheme according to the performances of a collection of representative runs, over varying hardware. Automatically choosing the preferred parallelization scheme for each loop individually
- ***ComPar* automatically chooses** different scheduling methods, chunk sizes, thread-affinity strategies, thread-placement options, number of threads and so forth

ComPar Architecture

***ComPar* workflow is composed of the following components:**

- **Combinator** creates all possible combinations of compilers and flags
- **Fragmentor** finds and enumerates all loops in the input source code
- **Timer** adds timing code around previously enumerated loops
- **Parallelizer** creates a parallel code for each compiler and compilation flag combination
- **Executor** runs the combinations on available compute nodes
- **Optimal code generator** fuses fastest code fragments, creating *ComPar* output
- **DB** holds all combinations, metadata and runtime information



ComPar architecture diagram

ComPar is under development in Python3 with OOP methodology, supports C source code and uses MongoDB database and Python Flask framework frontend.

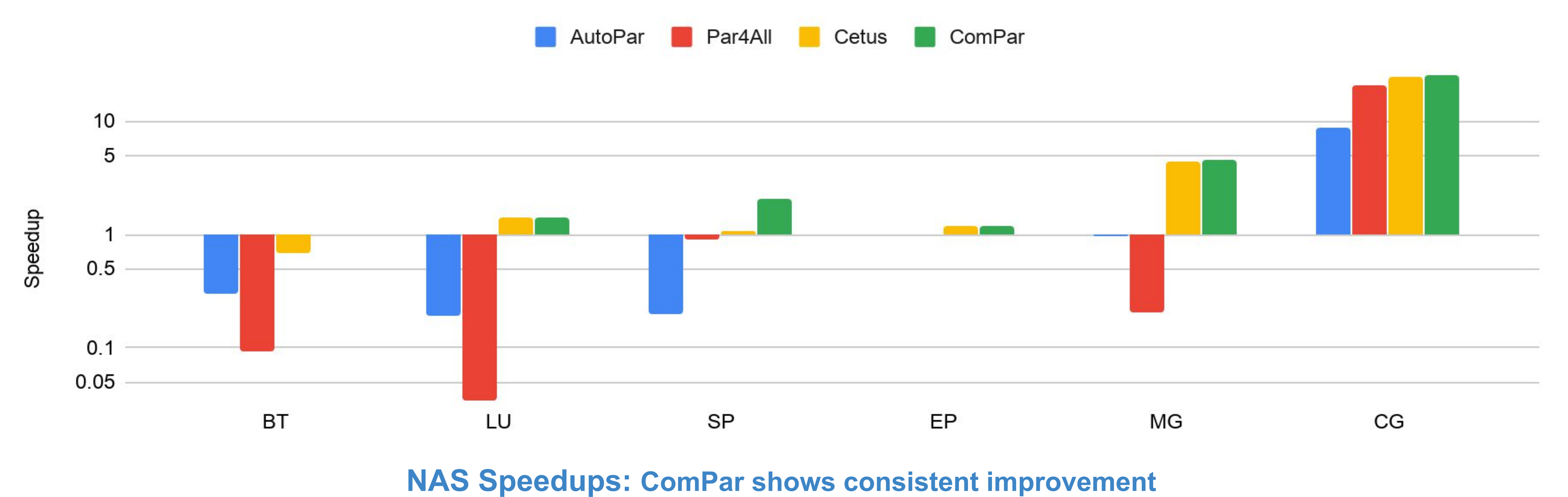
Performance Analysis

To show that **each S2S parallelization compiler has it's advantages and disadvantages**, and how ***ComPar* overcomes** them, **We tested *ComPar*'s performance** against said compilers on Numerical Aerodynamics Simulations (NAS) and PolyBench benchmarks. ***ComPar* always achieved the best speedups, or at least the same ones as the best S2S compiler (which is different for each benchmark)**

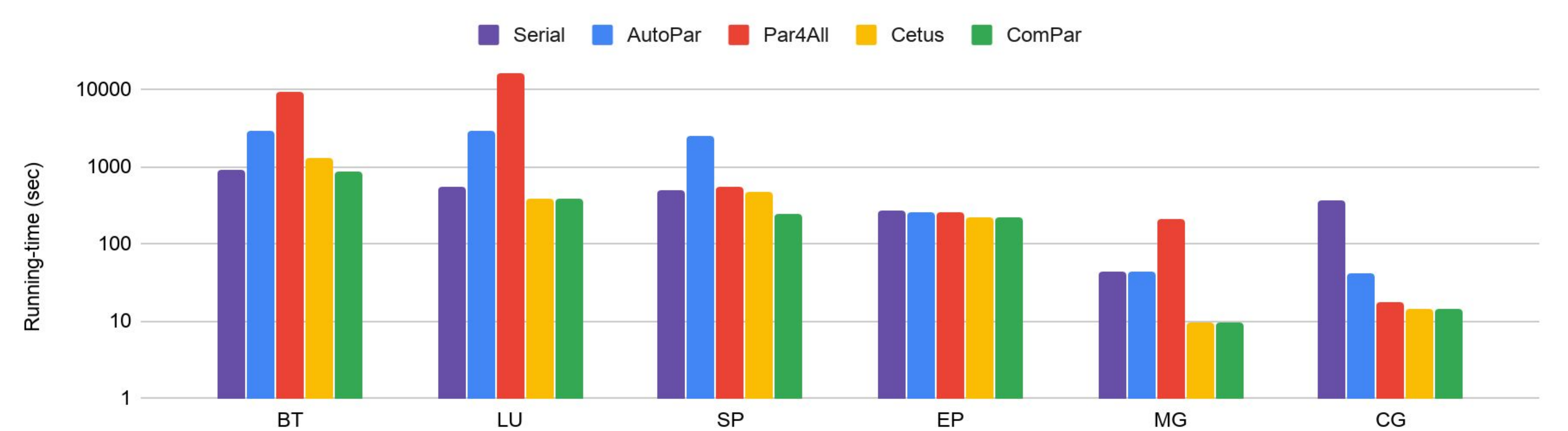
Combinations Table: parameters used in our tests

Compiler Flags	autopar	--keep_going, --enable_modeling, --no_aliasing, --unique_indirect_index
	par4all	-O, --fine-grain, --com-optimization, --no-pointer-aliasing
	cetus	-parallelize-loops=[1, 2], -reduction=[0, 2], -privatize=[0, 2], -alias=[1, 3]
OMP Directive Clauses	schedule	dynamic, static (2, 4, 8, 16, 32)
OMP Runtime Library Routines	omp_set_num_threads	2, 4, 8, 16, 32

NAS Parallel Benchmarks

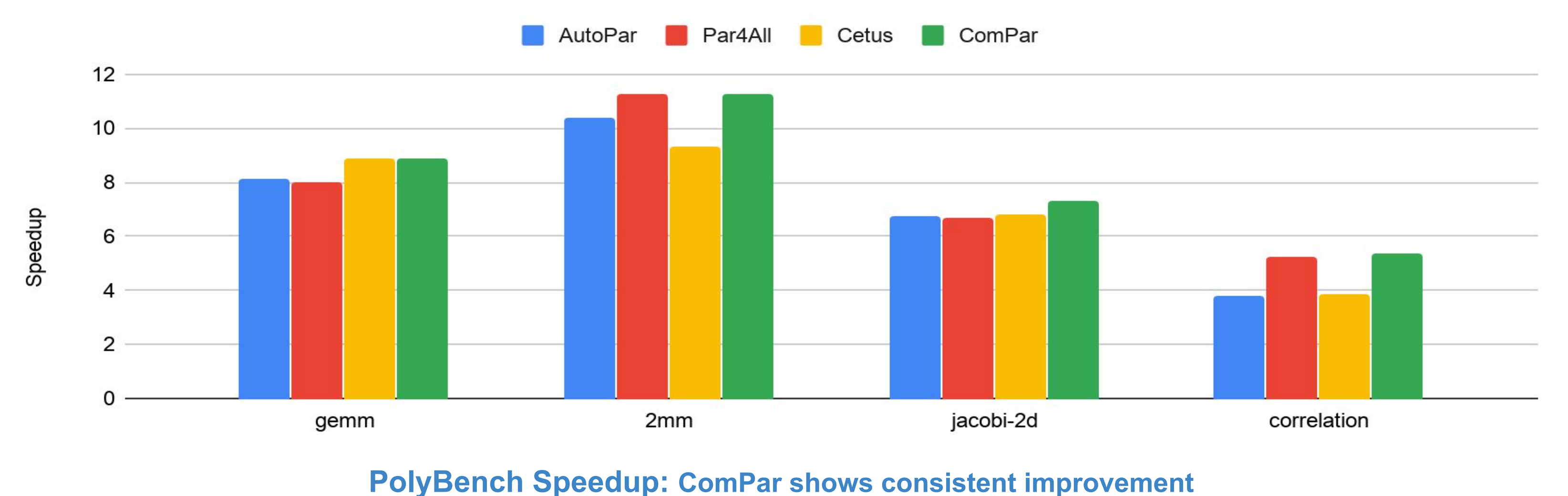


NAS Speedups: ComPar shows consistent improvement

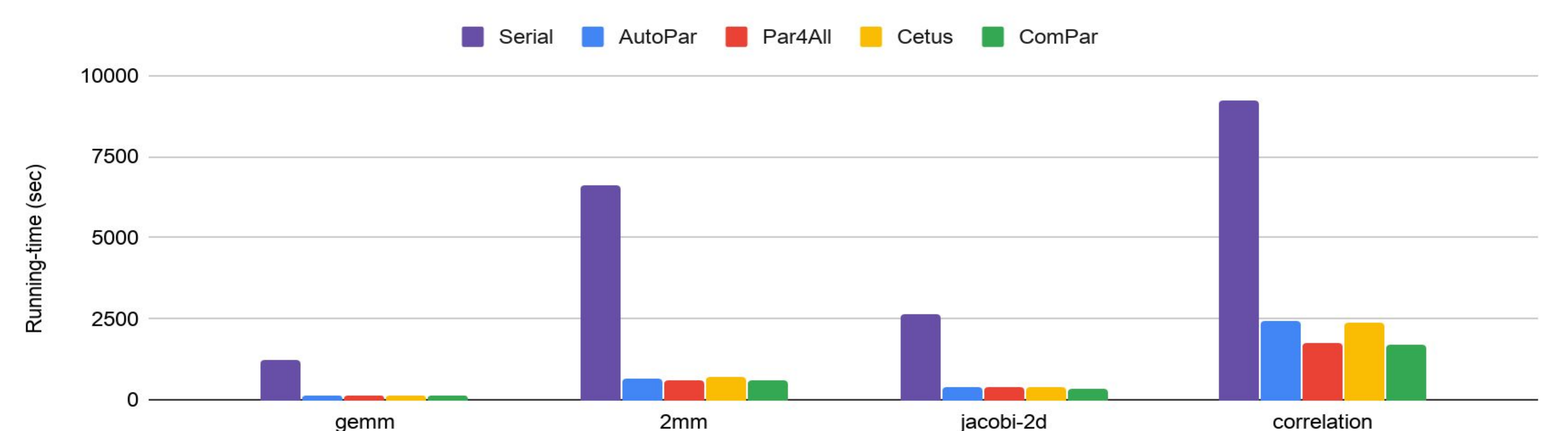


NAS Running-times [sec]: Absolute running-times are also important as the tasks grow expensive

PolyBench Benchmarks



PolyBench Speedup: ComPar shows consistent improvement



PolyBench Running-time [sec]: Absolute running-times are also important as the tasks grow expensive

Conclusion & Future Directions

- As we assumed, ***ComPar*'s results show that it is possible to increase the speedup** by combining several compilers with a mixture of compilation flags and environment parameters
- **All compilers are effective** to some extent, some more than others
- We hope to increase the performances by **adding more compilers in the future**
- In order to minimize the amount of runs **we will implement several search optimizations that could reduce the amount of combinations executed**
- Using **Machine Learning models** we hope to **learn the best hyperparameters for each specific hardware** and further narrow down the search phase

Acknowledgments: This work was supported by the Lynn and William Frankel Center for Computer Science. Computational support was provided by the ***NegevHPC project***.