Automatic Parallelization for Shared Memory Scientific Multiprocessing: An Analysis Comparison

Idan Mosseri

September 26, 2018

Ben-Gurion University of the Negev
Nuclear Research Center - Negev

idanmos@post.bgu.ac.il

Table of contents

Introduction

Introduction

- Parallelization schemes are essential in order to exploit the full benefits of multi-core architectures, which have become widespread in recent years.
- The introduction of correct parallelization to applications is not always a simple task.
- Automatic parallelization tools were created to ease this process.

Tools Listing

Table 1: Listing of Auto-Parallelization Tools.

Tool	License	Supported Language	Last Updated
AutoPar (ROSE)	Free	C, C++	May, 2017
Par4All (PIPS)	Free	C, Fortran, CUDA, OpenCL	May, 2015
Cetus	Free	C	Feb, 2017
SUIF compiler	Free	C, Fortran	2001
ICC	Proprietary	C, Fortran, C++	Jan, 2017
Polaris compiler	Free	Fortran 77	Unknown
S2P	Proprietary	C	Unknown

Automatic Parallelization

Automatic parallelization tools allow the programmer to focus on the application rather than on its parallelization.

To achieve their goal, these tools usually implement the following algorithm:

1. Scan and parse the code.
2. Turn it in to an AST (Abstract Syntax Tree).
3. Analyze this AST and find data dependencies.
4. Add parallel directives to code segments that may benefit from parallelization accompanied by other optimizations.
5. Finally, the AST is converted back to code in the original programming language.

Steps 3-4 may repeat several times.

Our Focus

In this study we overview and compare a subset of free up-to-date tools that were found to be most suitable for scientific code parallelization. Among these we can include:

- AutoPar (based on ROSE compiler)
- Par4All (based on PIPS)
- Cetus

We omit from our forward review the SUIF and Polaris compliers since they are outdated and only operate on older versions of Fortran compilers. The ICC compiler is not included in our review since it is not a source to source compiler. We omit S2P because it is a commercial software (as well as ICC).

Tools Specifications

AutoPar

- AutoPar is a module within the ROSE compiler.
- ROSE is under ongoing development by Lawrence Livermore National Laboratory (LLNL).
- Verifies the correctness of existing parallel codes.
- Supports C and C++.
- Supports Object Oriented Programming.
- Uses annotation files to handle complex data-structures.

AutoPar Pros and Cons

Pros

+ Suitable for OOP.
+ Verifies existing OpenMP directives.
+ Can be directed to add OpenMP directives regardless of errors.
+ Modifications are accompanied by clear explanation and reasoning in its' output.

Cons

- May Requires programmer intervention to handle function side-effects, classes etc. (via annotation file).
- May add incorrect OpenMP directives when given the "No-aliasing" option.

Par4All

- Par4All is an automatic parallelization tool built upon the PIPS source-to-source compiler.
- PIPS was developed in 1988 by several teams lead by SILKAN.
- The development of Par4All was shut-down by 2015.
- Specializes in inter-procedural analysis.
- Supports C, Fortran, CUDA and OpenCL.
- Supports parallelization for both multi-core (OpenMP) processors and GPGPUs (OpenCL, CUDA).
- Might Change the code to allow parallelization.

Par4All Pros and Cons

Pros

+ Suitable for GPUs.
+ Automatically analyzes function side effects and pointer aliasing.
+ Supports many data types.
+ Supports Fortran.

Cons

- Not under development
- Dead code will not be parallelized.
- May change the code structure.

Cetus

- Cetus is a compiler infrastructure for the source-to-source transformation of software programs.
- Developed by ParaMount research group at Purdue U.
- Supports C.
- Optimizations: data dependent analysis, pointer alias analysis, array privatization, reduction recognition.
- Ensures parallelization only on loops with 10,000+ iter'.
- Contains a graphical user interface (GUI).

Cetus Pros and Cons

Pros

+ Loop size dependent parallelization
+ Provides cross-platform interface.
+ Verifies existing OpenMP directives.
+ Modifications are accompanied by clear explanation and reasoning in its' output.

Cons

- Adds Cetus's pragmas which create excess code.
- May create uncompilable reduction clauses.
- Does not support function calls.

Comparison

Method

An input source code was made in order to show the differences between the output files generated by AutoPar, Par4All and Cetus, in order to both test and compare their capabilities.

The basic source code calculates Matrix Multiplication, and several variants of this problem were created, with each variant pointing to a different parallel shared memory management pitfall/obstacle.

Nested Loops and Array Reduction

Array Reduction/Privatization: How does the tool behave when a parallel directive can only be added with an array reduction clause or with an array declared as private?

Serial matrix multiplication code

```
1  void mat_mul(int n, int **a, int** b, int** c) {
2    int i,j,k;
3    for (i = 0; i < n; i++) {
4      for (j = 0; j < n; j++) {
5        for (k = 0; k < n; k++) {
6          c[i][j] += a[i][k] * b[k][j]; }}}
7    return;}
```

Nested Loops and Array Reduction

AutoPar inserted OpenMP directives to the two outermost loops.

AutoPar output

```
1  void mat_mul(...) {
2    int i, j, k;
3  #pragma omp parallel for private (i,j,k) firstprivate
    ↪  (n)
4    for (i = 0; i <= n - 1; i += 1) {
5  #pragma omp parallel for private (j,k)
6      for (j = 0; j <= n - 1; j += 1) {
7        c[i][j] = 0;
8        for (k = 0; k <= n - 1; k += 1) {
9          c[i][j] += a[i][k] * b[k][j]; }}}
10   return ; }
```

Nested Loops and Array Reduction

Par4All inserted OpenMP directive only to the outermost loop.

Par4All output

```
1  void mat_mul(...) {
2    int i, j, k;
3    #pragma omp parallel for private(j, k)
4    for(i = 0; i <= n-1; i += 1)
5      for(j = 0; j <= n-1; j += 1) {
6        for(k = 0; k <= n-1; k += 1)
7          c[i][j] += a[i][k]*b[k][j]; }
8    return; }
```

Nested Loops and Array Reduction

Cetus insert OpenMP directives to all three loops.

Cetus output

```
1   void mat_mul(...) {
2     int i, j, k;
3   #pragma cetus private(i, j, k)
4   #pragma loop name mat_mul#0
5   #pragma cetus parallel
6   #pragma omp parallel for if((10000<(((1L+(3L*n))+((4L*n)*n))+(((3L*n)*n)*n)))) private(i, j, k)
7     for (i=0; i<n; i ++ ) {
8   #pragma cetus private(j, k)
9   #pragma loop name mat_mul#0#0
10  #pragma cetus parallel
11  #pragma omp parallel for if((10000<((1L+(4L*n))+((3L*n)*n)))) private(j, k)
12      for (j=0; j<n; j ++ ) {
13        c[i][j]=0;
14  #pragma cetus private(k)
15  #pragma loop name mat_mul#0#0#0
16  #pragma cetus reduction(+: c[i][j])
17  #pragma cetus parallel
18  #pragma omp parallel for if((10000<(1L+(3L*n)))) private(k) reduction(+: c[i][j])
19        for (k=0; k<n; k ++ ) {
20          c[i][j]+=(a[i][k]*b[k][j]); }}}
21    return ; }
```

Pointer Aliasing

Verify Alias Dependence: Does the tool check for dependencies other then pointer aliasing when the No-Aliasing option is turned on?

Serial code with pointer aliasing

```
1  void mat_mul_pointer_alias(...) {
2    int i,j,k;
3    for (i = 0; i < N; i++) {
4      for (j = 0; j < N; j++) {
5        for (k = 0; k < N; k++) {
6          (*(c+i))[j] += (*(a+i))[k] * b[k][j]; }}}
7    return;}
```

- Par4All added a directive to the outer most loop.
- Cetus stepped into an internal error in this test case.

Pointer Aliasing

When given the No-Aliasing option, AutoPar ignored all data dependencies and inserted an incorrect OpenMP directive to the innermost loop.

AutoPar output

```
1  void mat_mul_pointer_alias(...) {
2    int i,j,k;
3  #pragma omp parallel for private (i,j,k) firstprivate
     ↪  (n)
4    for (i = 0; i <= n - 1; i += 1) {
5  #pragma omp parallel for private (j,k)
6      for (j = 0; j <= n - 1; j += 1) {
7        c[i][j] = 0;
8  #pragma omp parallel for private (k)
9        for (k = 0; k <= n - 1; k += 1) {
10         ( *(c + i))[j] += ( *(a + i))[k] * b[k][j];
             ↪  }}}
11   return; }
```

Loop Unrolling

Loop Unroll support: Will the tool insert OpenMP directives to loops containing loop unroll?

Serial code with loop unrolling

```
1   void mat_mul_loop_unroll(...) {
2     int i,j,k;
3     for (i = 0; i < N-1; i+=2) {
4       for (j = 0; j < N-1; j+= 2) {
5         for (k = 0; k < N-1; k += 2) {
6           c[i][j] += (a[i][k] * b[k][j] + a[i][k+1] * b[k+1][j]);
7           c[i][j+1] += (a[i][k] * b[k][j+1] + a[i][k+1] * b[k+1][j+1]);
8           c[i+1][j] += (a[i+1][k] * b[k][j] + a[i+1][k+1] * b[k+1][j]);
9           c[i+1][j+1] += (a[i+1][k] * b[k][j+1] + a[i+1][k+1] *
              ↪  b[k+1][j+1]); }}}
10    return; }
```

- AutoPar did not insert any OpenMP directives.
- Par4All added an OpenMP directive only to the outer-most loop as in the first test.

Loop Unrolling

- As before, Cetus added OpenMP directives to all three loops.
- However, Cetus's output for the innermost loop is invalid since it contains a reduction clause for multiple array cells.
- This kind of reduction can not be compiled - as far as is known - by any compiler.

The innermost loop of Cetus Output for the loop unrolling test-case.

```
1  #pragma cetus private(k)
2  #pragma loop name mat_mul_loop_unroll3#0#0#0
3  #pragma cetus reduction(+: c[i+1][j+1], c[i+1][j],
    ↪  c[i][j+1], c[i][j])
4  #pragma cetus parallel
5  #pragma omp parallel for if((10000<(1L+(6L*((-2L+n)/2L)))))
    ↪  private(k) reduction(+: c[i+1][j+1], c[i+1][j],
    ↪  c[i][j+1], c[i][j])
6  for (k=0; k<(n-1); k+=2 {
7    ... };
```

Function Calls

Function call support: Will the tool insert OpenMP directives to loops containing function calls with/without side effects?

Serial Code with function calls

```
1  void compute_cijk(int i, int j, int k, int a[N][N],
    ↪  int b[N][N], int c[N][N]) {
2    c[i][j] += a[i][k] * b[k][j];}
3
4  void mat_mul_function_calls(...) {
5    ...
6          compute_cijk(i,j,k,a,b,c);}}}
7    return;}
```

- AutoPar could not parallelize the code without an Annotation file.
- Cetus did not add any OpenMP directives as well.
- Par4All Inserted an OpenMP directive to the outermost loop as before.

Quick recap

The following table summarizes the three tools by pointing out their key features.

Table 2: Table summary of the three tools key features

Feature	AutoPar (ROSE)	Par4All (PIPS)	Cetus
Supported Languages	C, C++	C, Fortran, CUDA	C
Loop Unroll	No	Yes	Yes
Verify 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

Runtime Analysis

Runtime Analysis

All test-cases were compiled

- Using Intel(R) C Compiler XE 2018.
- Update 5 for Linux*.
- Using internal optimizations (i.e. -O3). Unless stated otherwise.

And executed on

- Machine with two Intel(R) Xeon(R) CPU E5-2683 v4 processors.
- Intel(R) Xeon-Phi co-processor 5100 series (rev 11) in native mode.

Runtime Analysis

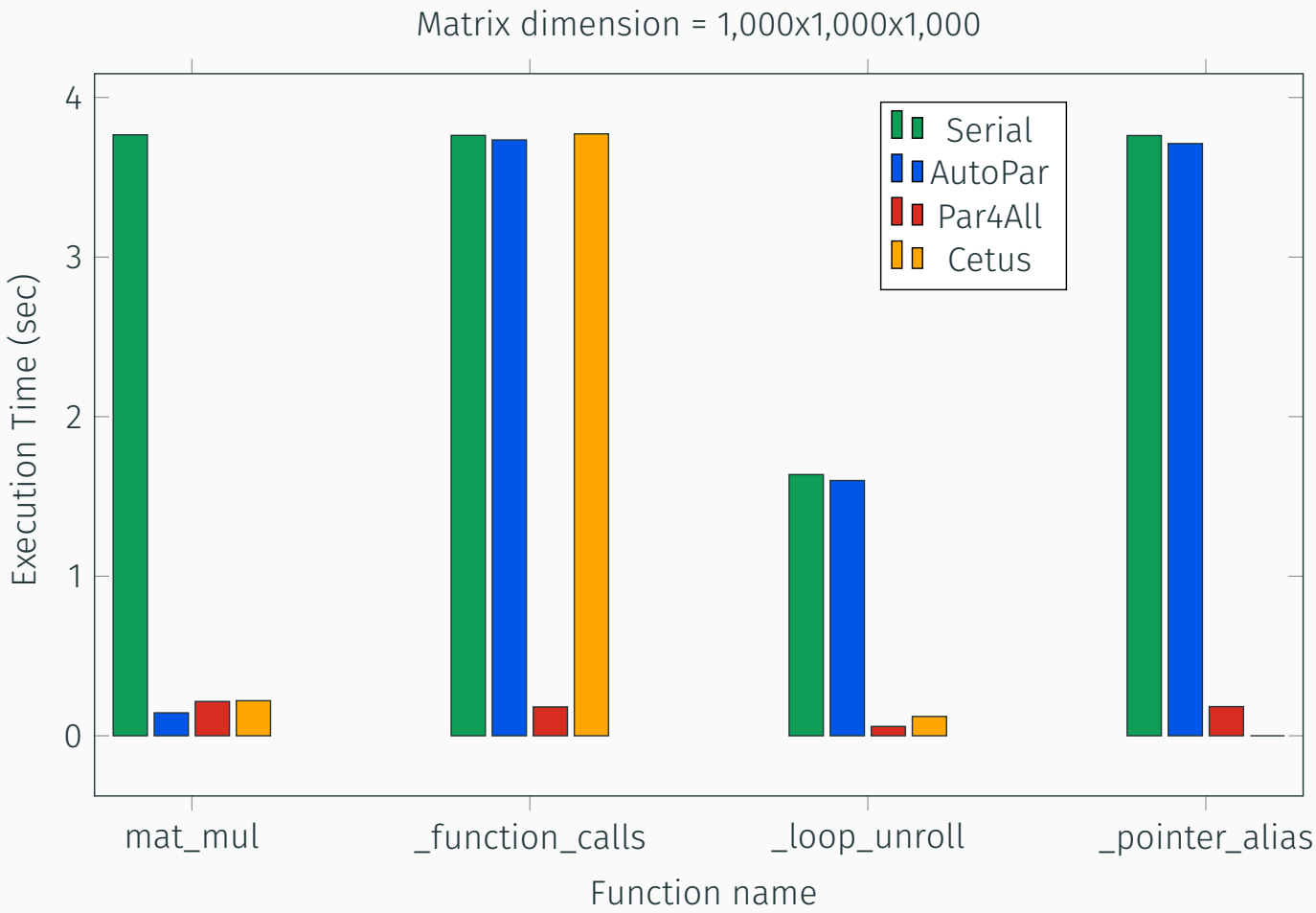

Figure 1: Functions execution time generated by the three tools and serial execution.

Runtime Analysis

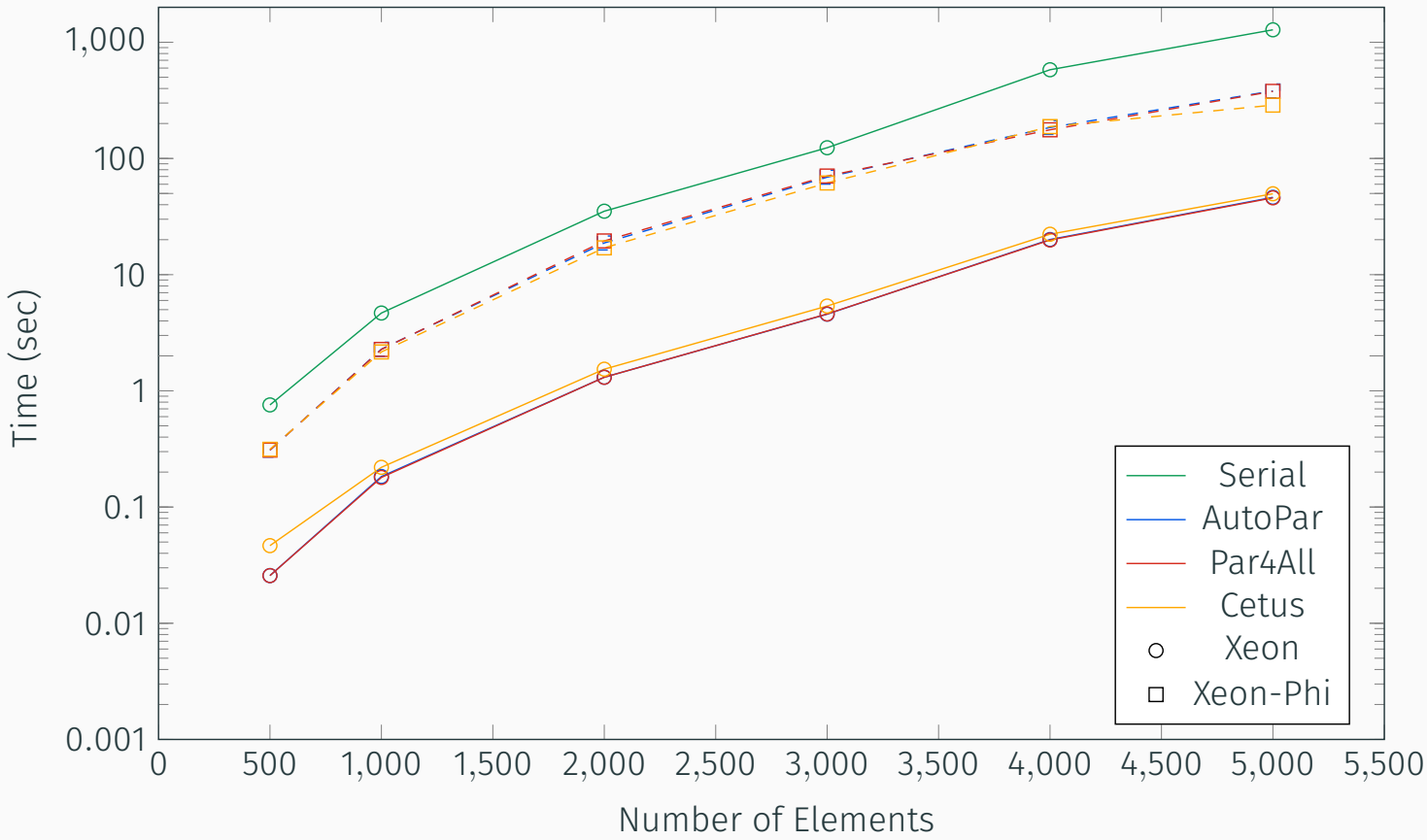

Figure 2: *mat_mul* execution time with different number of elements in log-scale.

Runtime Analysis

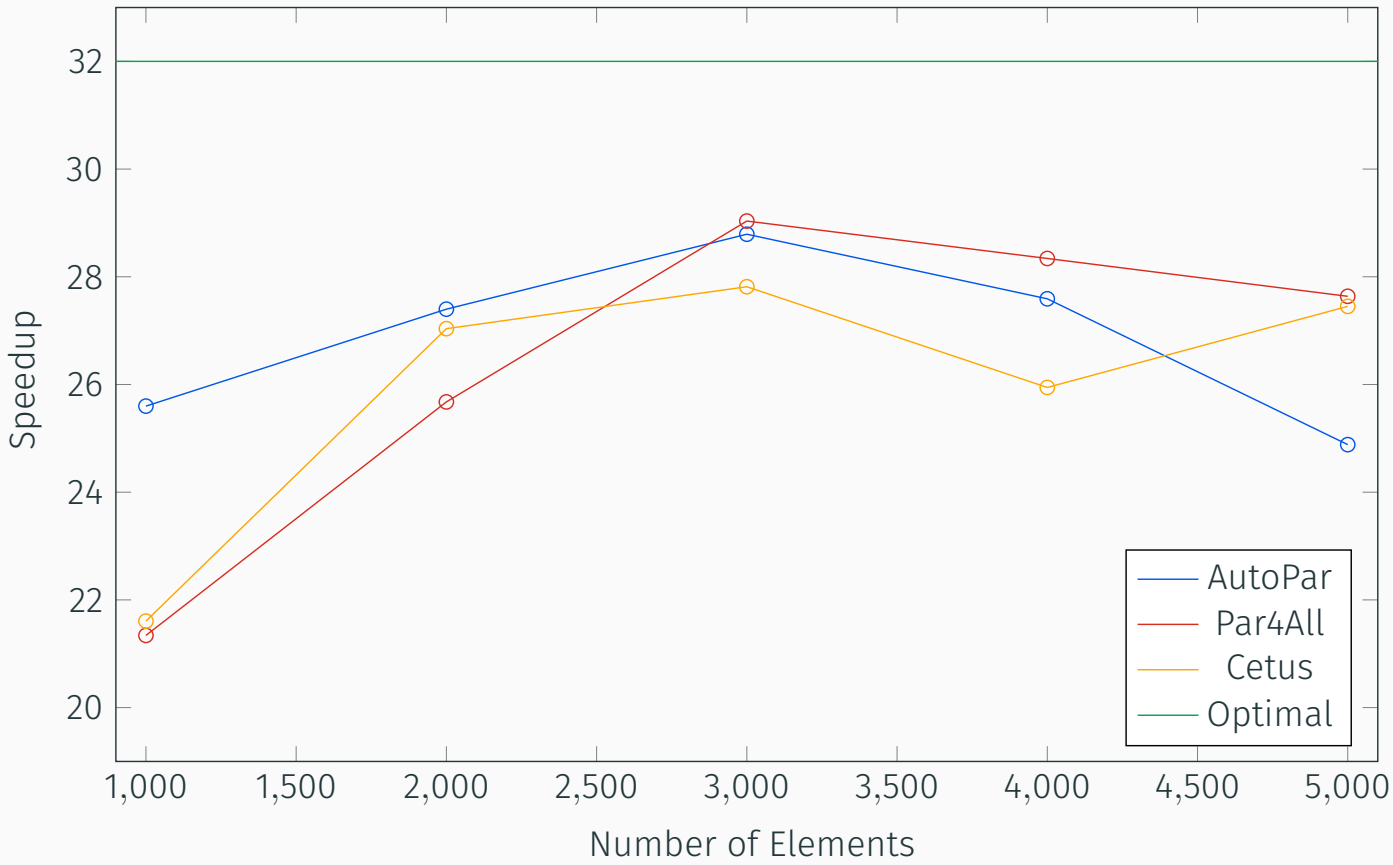

Figure 3: *mat_mul* speedup with different number of elements in log-scale.

Runtime Analysis

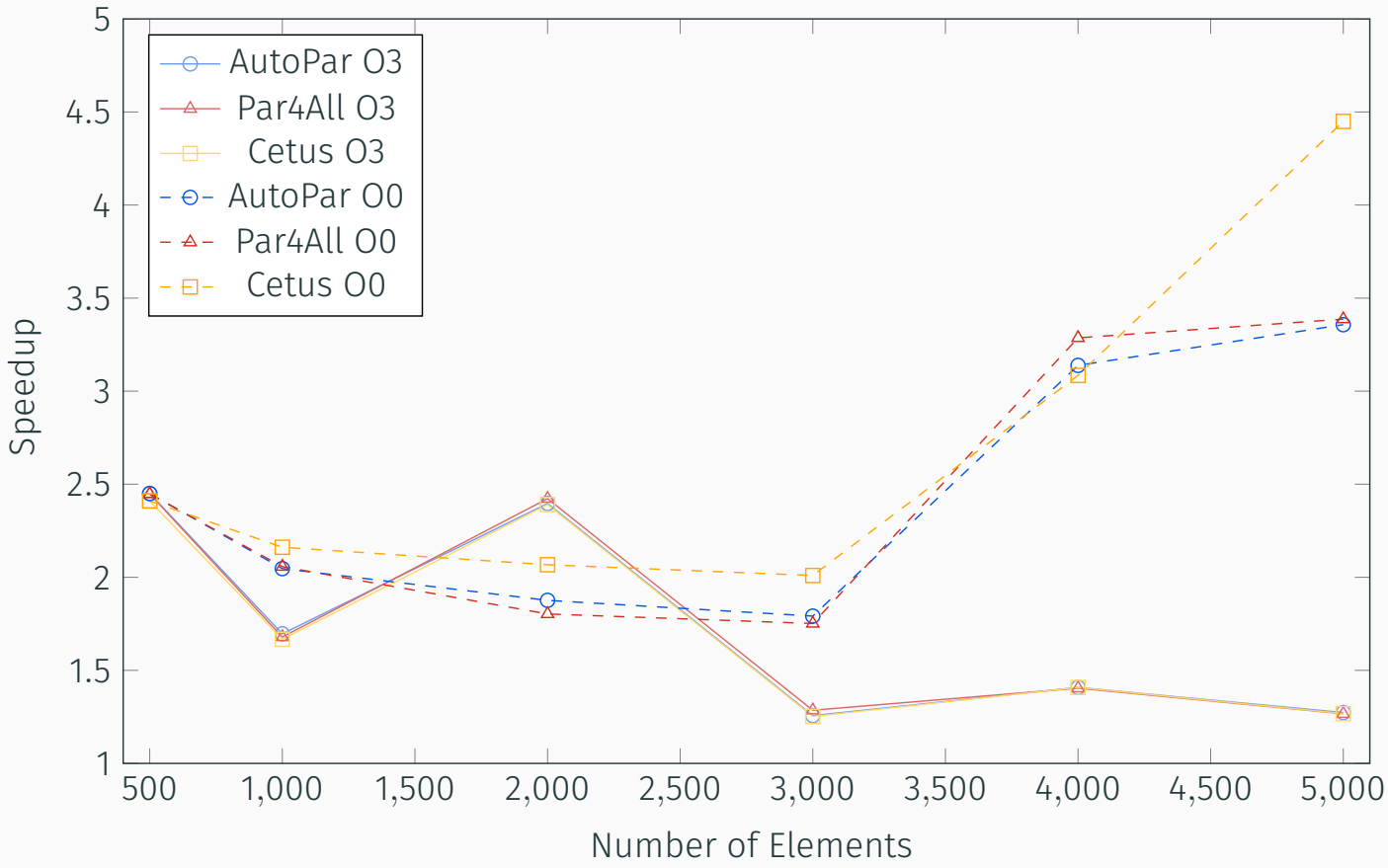

Figure 4: *mat_mul* speedup on Intel Xeon-Phi compared to serial run (on Intel Xeon) with both -O3 and -O0.

Runtime Analysis

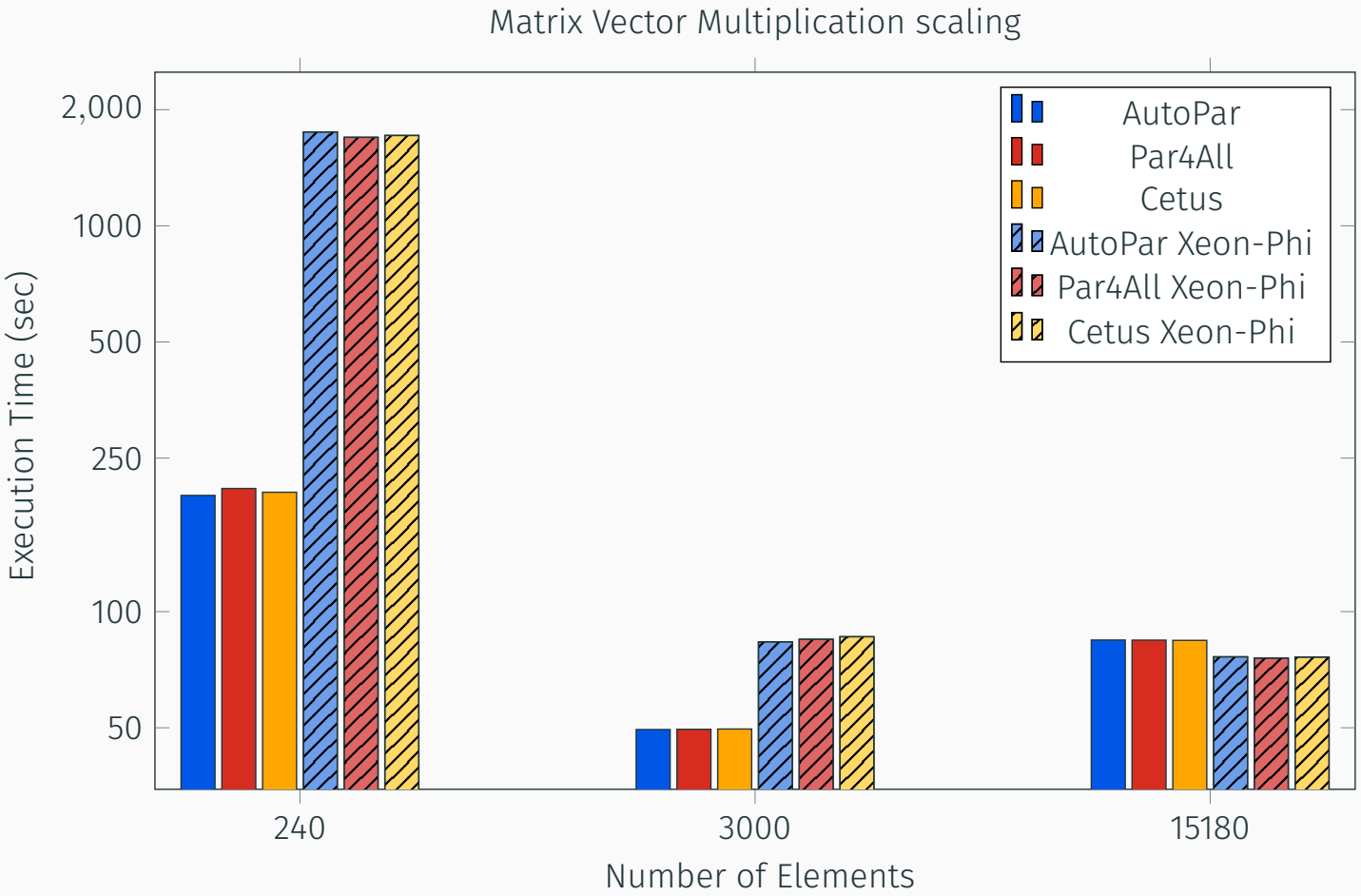

Figure 5: Functions execution time generated by the three tools and serial execution.

NAS Parallel Benchmarks

NAS Parallel Benchmarks

To further evaluate the tools capabilities, we introduce the Numerical Aerodynamics Simulations (NAS) Parallel Benchmarks, developed by NASA.

NAS benchmarks include:

- Block Tri-diagonal solver (BT).
- Conjugate Gradient (CG).
- Embarrassingly Parallel (EP).
- Lower-Upper Gauss-Seidel solver (LU)
- Multi-Grid (MG)
- Scalar Penta-diagonal solver (SP)
- Unstructured Adaptive mesh (UA)

NAS Parallel Benchmarks

AutoPar - Failed to gain any speedup in most of the benchmarks (BT, LU, MG, SP, EP). This can be explained by the insertion of OpenMP directives to computationally small and inner nested loop.

AutoPar's directives on computationally small loops EP/ep.c.

```
1  #pragma omp parallel for private (i)
2    for (i = 0; i <= 9; i += 1) {
3      q[i] = 0.0; }
4  ...
5  #pragma omp parallel for private (gc,i) reduction
    ↪  (+:gc)
6    for (i = 0; i <= 9; i += 1) {
7      gc = gc + q[i]; }
```

NAS Parallel Benchmarks

AutoPar - Failed to gain any speedup in most of the benchmarks (BT, LU, MG, SP, EP). This can be explained by the insertion of OpenMP directives to computationally small and inner nested loop.

AutoPar's directives on nested loop and low iterative loops SP/rhs.c.

```
1  #pragma omp parallel for private (i,j,k,m)
2    for (k = 0; k <= grid_points[2] - 1; k += 1) {
3  #pragma omp parallel for private (i,j,m)
4      for (j = 0; j <= grid_points[1] - 1; j += 1) {
5  #pragma omp parallel for private (i,m)
6        for (i = 0; i <= grid_points[0] - 1; i += 1) {
7  #pragma omp parallel for private (m)
8          for (m = 0; m <= 4; m += 1) {
9            rhs[k][j][i][m] = forcing[k][j][i][m]; }  }
               ↪  }  }
```

NAS Parallel Benchmarks

Par4All - Unlike AutoPar, Par4All did not insert multiple directives in nested loops. However, Par4All did insert many of its directives on the innermost loops and on computationally small loops

Par4All's directives on computationally small and nested loops

```
1   #pragma omp parallel for
2       for(m = 0; m <= 4; m += 1)
3           errnm[m] = 0.0;
4
5       for(k = 1; k <= nz-1-1; k += 1)
6           for(j = jst; j <= jend-1; j += 1)
7               for(i = ist; i <= iend-1; i += 1) {
8                   exact(i, j, k, u000ijk);
9   #pragma omp parallel for private(tmp)
10                  for(m = 0; m <= 4; m += 1) {
11                      tmp = u000ijk[m]-u[k][j][i][m];
12                      errnm[m] = errnm[m]+tmp*tmp; } }
```

NAS Parallel Benchmarks

Cetus - Cetuss loop size dependent parallelization combined with the option to parallelize only the outermost-parallelizable loop in the loop nest explains the speedup gained in the CG, EP, LU, MG, SP benchmarks.

Cetus directive in BT/x_solve.c.

```
1  #pragma omp parallel for
     ↪ if((10000<(-78L+(79L*isize)))) private(i)
     ↪ lastprivate(tmp1, tmp2a omp parallel for
     ↪ if((10000<(56L+(55L*isize)))) private(i)
     ↪ lastprivate(tmp1, tmp2, tmp3)
2      for (i=0; i<=isize; i ++ ) ...
```

We expand our analysis and evaluate the impact of minimal human intervention on the tools output, by removing unnecessary directives.

NAS Parallel Benchmarks

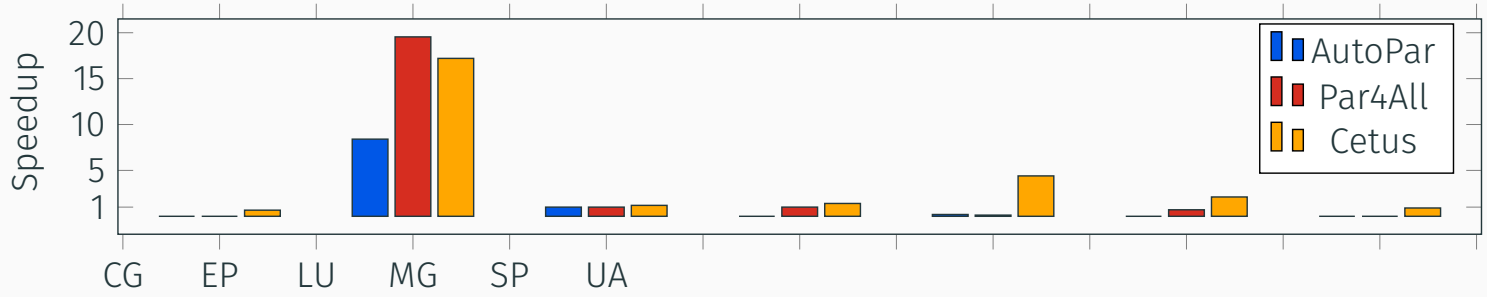

Figure 6: Speedup gained from the tools alone on the tested benchmarks compared to serial execution.

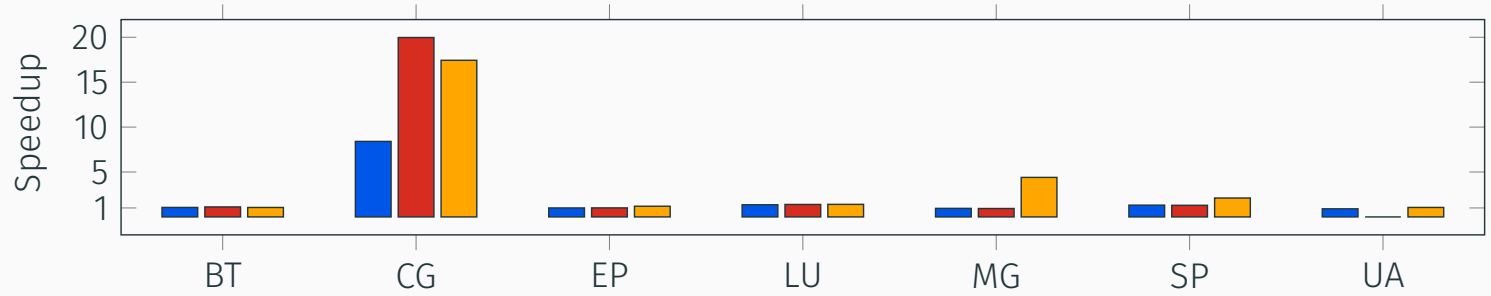

Figure 7: Speedup gained from removing the unnecessary OpenMP directives compared to serial execution.

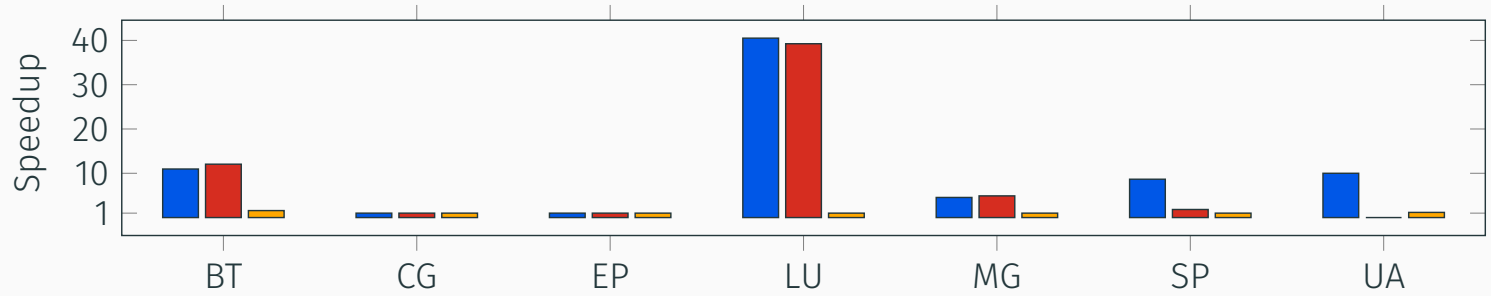

Figure 8: Speedup gained from removing the unnecessary OpenMP directives compared to non-removal.

Accelerators & Co-processors

Accelerators & Co-processors

Par4All's GPU accelerator support relies on an API called P4A_ACCEL.

- P4A_ACCEL provides an encapsulation of CUDA's API.
- Data-parallel loops are automatically transformed into CUDA kernels that are executed on GPUs.
- Ad hoc communications between the host memory and the GPU memory are generated to enable kernel execution.

We test Par4All's ability to transform C code to CUDA for GPUs. Modifications to the matrix multiplication test were required for Par4all to support the code transformation to CUDA and to enable better compiler optimization and memory management.

Matrix multiplication code for the GPU case study

```
1  void mat_mul(int* a, int* b, int* c) {
2      int i,j,k;
3      for (i = 0; i < N; i++) {
4          for (j = 0; j < N; j++) {
5              c[i*N+j] = 0;
6              for (k = 0; k < N; k++) {
7                  c[i*N+j] += a[i*N+k] * b[j*N+k];
8              } } } return; }
```

Accelerators & Co-processors

The native compilation scheme is to

- Allocate the desired space on the GPU.
- Transfer the computation data to the GPU.
- Launch the kernel.
- Copy back the results from the GPU.

Matrix multiplication code for the GPU transformed by Par4All to Accel (CUDA)

```
1  void mat_mul(int *a, int *b, int *c) {
2        int (*p4a_var_c0)[1000000] = (int (*)[1000000]) 0, (*p4a_var_b0)[1000000] =
          ↪  (int (*)[1000000]) 0, (*p4a_var_a0)[1000000] = (int (*)[1000000]) 0;
3        P4A_accel_malloc((void **) &p4a_var_a0, sizeof(int)*1000000); // Same for
          ↪  b0, c0
4        P4A_copy_to_accel_1d(sizeof(int), 1000000, 1000000, 0, &a[0], *p4a_var_a0);
          ↪  // Same for b0, c0
5        p4a_launcher_mat_mul(*p4a_var_a0, *p4a_var_b0, *p4a_var_c0);
6        P4A_copy_from_accel_1d(sizeof(int), 1000000, 1000000, 0, &c[0],
          ↪  *p4a_var_c0);
7        P4A_accel_free(p4a_var_a0); // Same for b0, c0
8     return; }
```

Accelerators & Co-processors

Matrix multiplication code for the GPU transformed by Par4All to Accel (CUDA) wrappers

```
1  void P4A_accel_malloc(void **address, size_t size);
2  void P4A_copy_to_accel_1d(size_t element_size, size_t d1_size, size_t
    ↪  d1_block_size, size_t d1_offset, const void *host_address, void
    ↪  *accel_address);
3  void P4A_copy_from_accel_1d(size_t element_size, size_t d1_size, size_t
    ↪  d1_block_size, size_t d1_offset, void *host_address, const void
    ↪  *accel_address);
4  void P4A_accel_free(void *address);
5  P4A_accel_kernel_wrapper p4a_wrapper_mat_mul(int i, int j, int *a, int *b, int *c)
    ↪  {
6     i = P4A_vp_1; // Index has been replaced by P4A_vp_1
7     j = P4A_vp_0; // Index has been replaced by P4A_vp_0
8     p4a_kernel_mat_mul(i, j, a, b, c); }
9  P4A_accel_kernel p4a_kernel_mat_mul(int i, int j, int *a, int *b, int *c) {
10    int k; // Declared by Pass Outlining
11    if (i<=999&&j<=999) {
12       c[i*1000+j] = 0;
13       for(k = 0; k <= 999; k += 1)
14          c[i*1000+j] += a[i*1000+k]*b[j*1000+k]; } }
15 void p4a_launcher_mat_mul(int *a, int *b, int *c) {
16    int i, j;  // Declared by Pass Outlining
17    P4A_call_accel_kernel_2d(p4a_wrapper_mat_mul, 1000, 1000, i, j, a, b, c); }
```

Accelerators & Co-processors

We test the Par4All's CUDA output on a GPU against Par4Alls OpenMP output on a Xeon processor and a Xeon-Phi Co-processor.

The following architectures were used to test this study case:

- *GPU*: NVIDIA(R) Tesla(R) P100-PCIE-16GB.
- *Xeon*: Intel(R) Xeon(R) CPU E5-2683 v4 2 process units with 16 cores each.
- *Xeon-Phi*: Intel(R) Xeon-Phi co-processor 5100 series (rev 11).

Accelerators & Co-processors

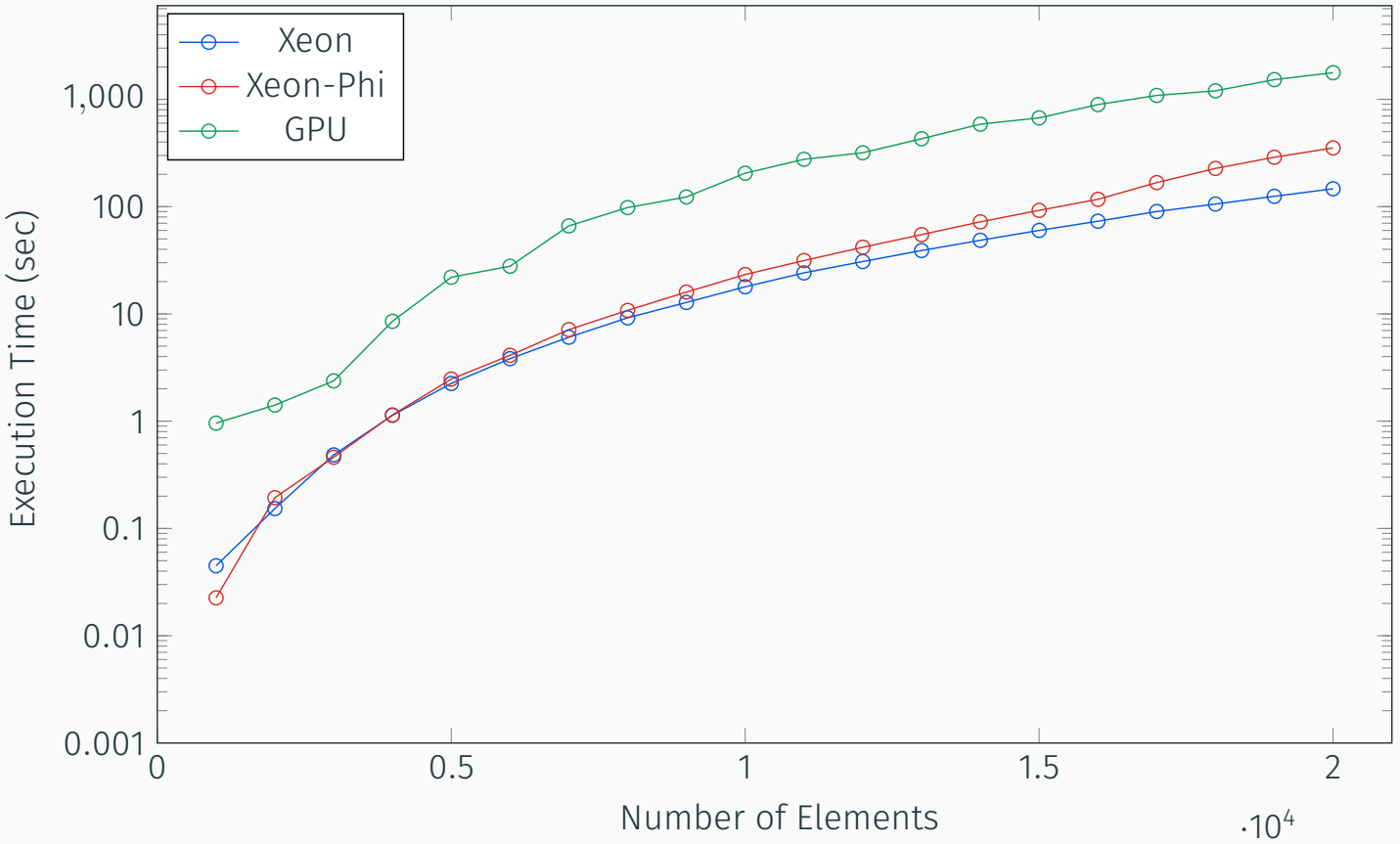

Figure 9: Execution time comparison of Par4All output on Xeon and Xeon-Phi and GPU.

Accelerators & Co-processors

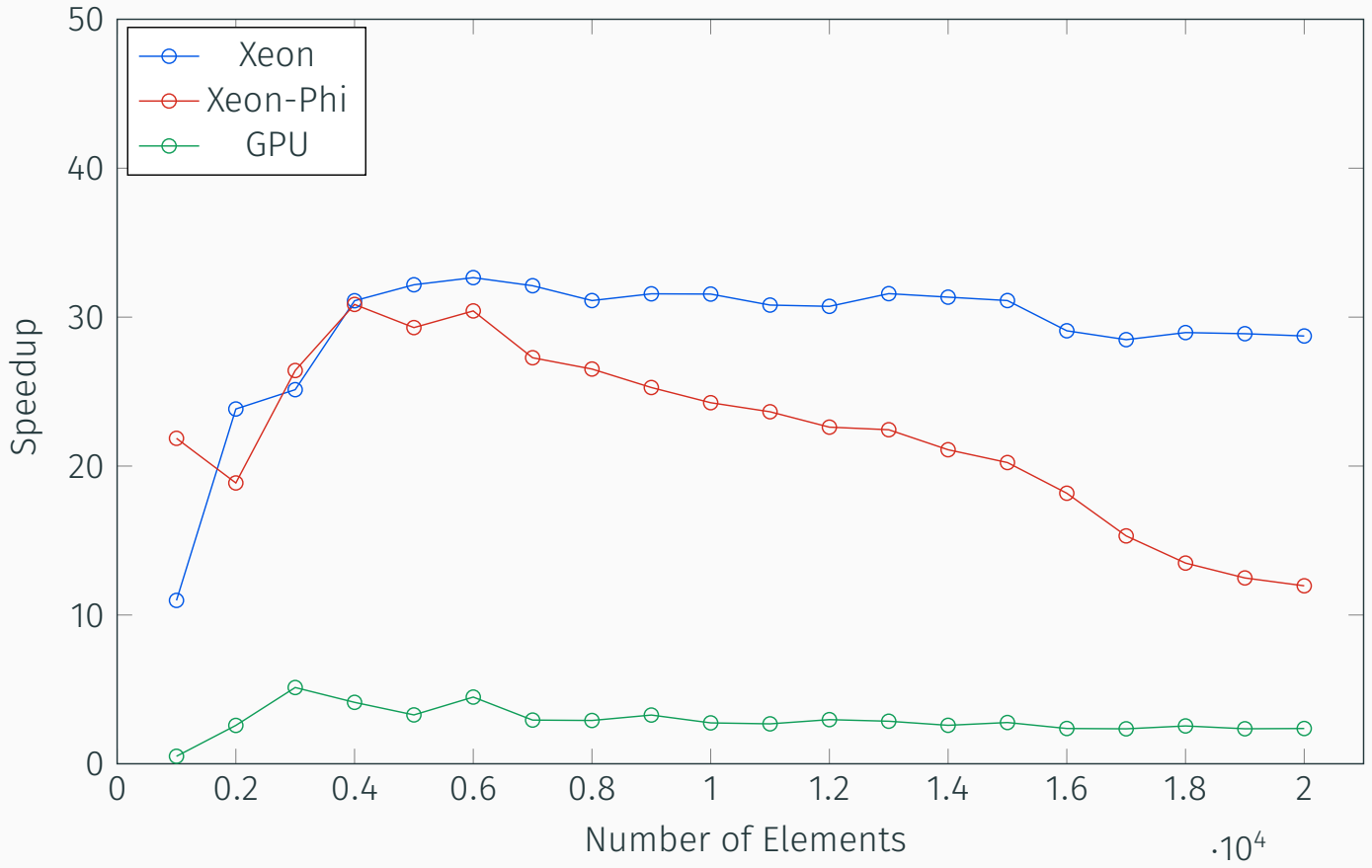

Figure 10: Speedup comparison of Par4All output on Xeon and Xeon-Phi and GPU.

Conclusions

Conclusions

- Each tool has its strengths and weaknesses.
- Automatic parallelization can't replace developers yet.
- but in some cases it gives good results.
- minimal human intervention may improve the tools results even more.
- automatic parallelizion isn't quite adapted for complex architectures like accelerators & co-processors yet.

Future Work

Future Work

We can still do better by:

- combining the tools output in some manner.
- remove and change some of their outputs.
- and optimize run-time parameters like chunk size and scheduling type.

We plan on creating a smi learning system that performs all of the above based on diagnostics from many runs of the tools outputs on different architectures and with different parameters.

Future Work

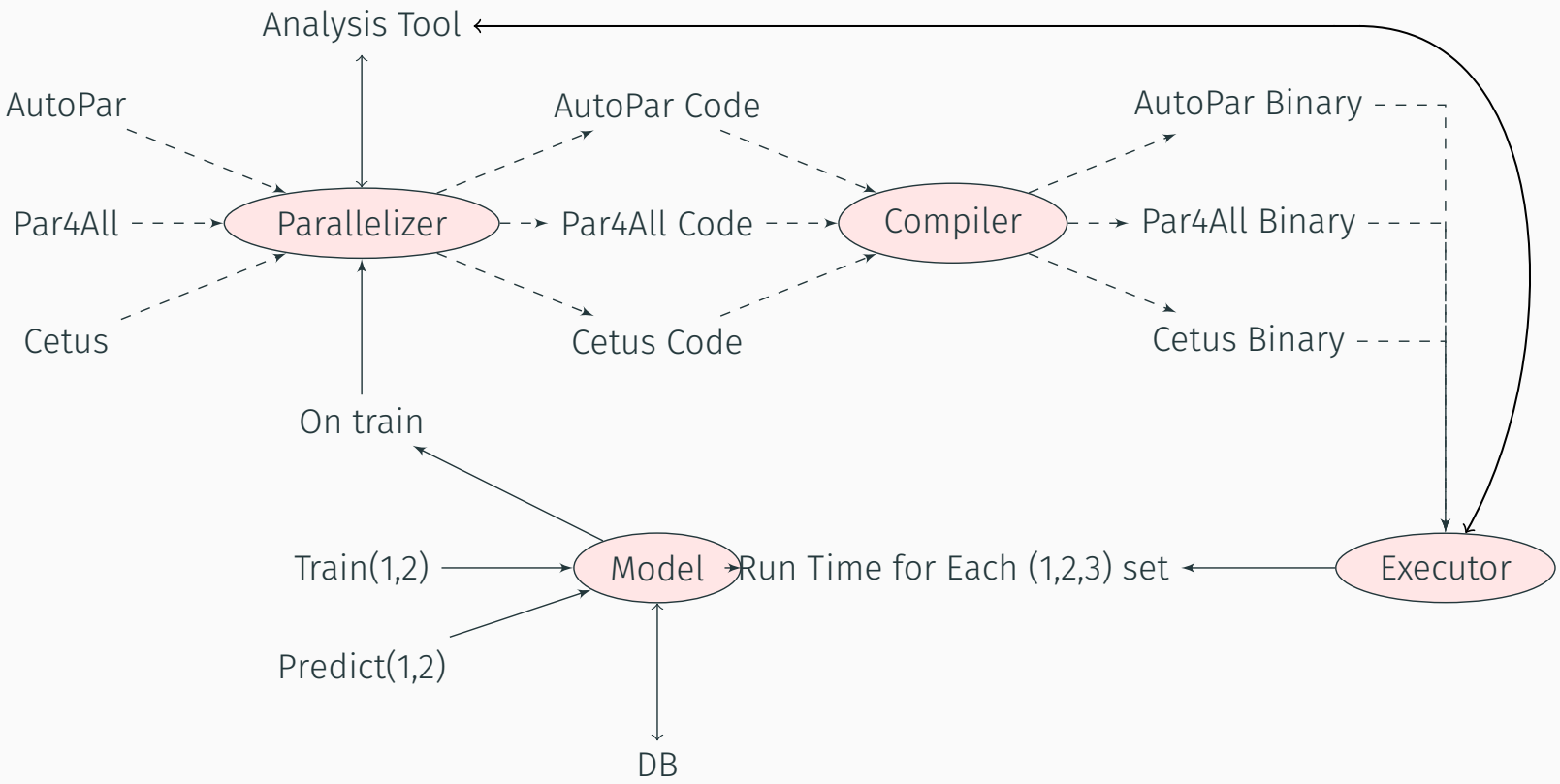

Figure 11: Scheme of the desired Parallelizator.

Questions?