International Journal of Parallel Programming
https://doi.org/10.1007/s10766-019-00640-3

Source-to-Source Parallelization Compilers for Scientific Shared-Memory Multi-core and Accelerated Multiprocessing: Analysis, Pitfalls, Enhancement and Potential

Re'em Harel[1,2] · Idan Mosseri[3,4] · Harel Levin[4,5] · Lee-or Alon[2,6] · Matan Rusanovsky[2,3] · Gal Oren[3,4]

Received: 14 November 2018 / Accepted: 31 July 2019

Abstract
Parallelization schemes are essential in order to exploit the full benefits of multi-core architectures, which have become widespread in recent years, especially for scientific applications. In shared memory architectures, the most common parallelization API is OpenMP. However, the introduction of correct and optimal OpenMP parallelization to applications is not always a simple task, due to common parallel shared memory management pitfalls and architecture heterogeneity. To ease this process, many automatic parallelization compilers were created. In this paper we focus on three source-to-source compilers—AutoPar, Par4All and Cetus—which were found to be most suitable for the task, point out their strengths and weaknesses, analyze their performances, inspect their capabilities and suggest new paths for enhancement. We analyze and compare the compilers' performances over several different exemplary test cases, with each test case pointing out different pitfalls, and suggest several new ways to overcome these pitfalls, while yielding excellent results in practice. Moreover, we note that all of those source-to-source parallelization compilers function in the limits of OpenMP 2.5—an outdated version of the API which is no longer in optimal accordance with nowadays complicated heterogeneous architectures. Therefore we suggest a path to exploit the new features of OpenMP 4.5, as it provides new directives to fully utilize heterogeneous architectures, specifically ones that have a strong collaboration between CPUs and GPGPUs, thus it outperforms previous results by an order of magnitude.

Keywords Parallel programming · Automatic parallelism · Cetus · AutoPar · Par4All

✉ Gal Oren
orenw@post.bgu.ac.il

Extended author information available on the last page of the article

1 Introduction

1.1 Parallel Programming

Multi-core architectures have become very common in recent years [1]. These architectures are no longer exclusive to the HPC industry, and are now implemented in most personal computers, laptops, smart-phones and even wearable devices [2]. In order to exploit the benefits of these architectures, programming methods should evolve from a sequential to a parallel work fashion [3]. A parallel program is a program that is being executed by several processing units simultaneously. Thus, dividing the program's workload between these processing units.

There are two main parallel programming models [4]: the shared memory model and the distributed memory model.

1. In the *shared memory model*, several processing units can exchange data and communicate through a common address space. All processing units can read from and write to this address space simultaneously, which may induce synchronization problems. Most shared memory programs consist of a single process which manages several threads, usually a single thread per core. A common way to implement this model is with OpenMP [5] usage. OpenMP is a pragma oriented library for shared memory parallelization. Using OpenMP, the programmer can mark code segments which will then be divided and executed concurrently between several threads. Marking these code segments is done by wrapping them with compiler directives (pragmas), which dictate how the parallelization should be implemented. The programmer can control the parallel execution using OpenMP's runtime subroutines. OpenMP's environment variables may be used to customize runtime specification such as scheduling and thread count, prior to the process execution.
2. In the *distributed memory model*, multiple processes coordinate in order to perform a task. In this model, each process runs on a separate processing unit and has its own individual memory. These processes can reside on the same machine or distributed over several different ones. An inter-process communication infrastructure has to be established between these processes to share data and synchronize execution. A common way to implement this model is with the MPI (Message Passing Interface) standard [6,7]. MPI implementations provide an API with which a set of processes can exchange data by sending and receiving both synchronous and asynchronous messages. All implementations consist of the same set of routines defined by the MPI standard. One common implementation of the MPI standard is OpenMPI [8]. Note that these two models can be combined such that each MPI process will implement several OpenMP threads. This model is called the Hybrid Parallelization model.

In this paper we will focus on the shared memory model, mainly because most, if not all automatic parallelization compilers tend to address this model, due to its relative ease of implementation—as directives mostly keep the code structure as it is—unlike other programming methods (such as MPI and CUDA [9]) which require a more fundamental reconstruction of the desired code, thus creating a much harder problem for automation.

1.2 Automatic Parallelization Techniques

The intricate relationships between different memory structures in the program, accompanied by the need to synchronize simultaneous reads and writes to these structures, and the ambition to distribute the workload evenly across the systems resources, make it difficult to write a parallel program over the shared memory model. Similarly, integrating parallelism into existing legacy codes may be tedious, as it requires a comprehensive understanding of these codes and their intricacies, such as in AutOMP compiler, which was designed for a particular kind of Fortran legacy code loops parallelization [10]. In common code scenarios, parallelizing legacy codes may entail rewriting them entirely. These concerns led to the creation of several automatic parallelization compilers, which allow the programmer to focus on the application rather than on its parallelization. To achieve their goal, these compilers first scan and parse the code and turn it into an AST (Abstract Syntax Tree) [11]. Afterwards, they analyze this AST, find data dependencies and add parallel directives to code segments that may benefit from parallelization accompanied by other optimizations. The analysis and optimization steps may be repeated several times until convergence. Finally, the AST is converted back to code in the original programming language. All of the above mentioned process is done while maintaining both the program's correctness and data coherence implied by its data dependencies.

Over the years many automatic parallelization compilers were created based on these techniques. Each have their own supported programming languages as well as their own pros and cons. Table 1 briefly points out the most well-established automatic-parallelization compilers to this day.

In this study we overview and compare a subset of free up-to-date Compilers that were found to be most suitable for scientific code parallelization. Among these we included AutoPar (based on ROSE) [12–14], Par4All (based on PIPS) [15–17] and Cetus [18,19]. We omit from our forward review the SUIF [20] and Polaris [21] compilers since they are outdated and only operate on older versions of Fortran compilers. The ICC [22] 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). Pluto [23]

Table 1 Listing of auto-parallelization compilers

Criteria	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
Pluto	Free	C	2015
Polaris compiler	Free	Fortran 77	Unknown
S2P	Proprietary	C	Unknown

is not presented because it requires manual intervention, which makes it irrelevant for legacy codes or large codes in general.

It's worth stating that the reader should keep in mind there is no *best* compiler, rather a *preferable* compiler for each case. The target of this paper is to conclude which compilers are the most preferable ones based on their evaluation and performances comparison.

2 Related Work

As previously mentioned, choosing the most suitable automatic parallelization compiler is not a simple task, and previous work was done in order to ease this decision accordingly. Prema et al. [24] briefly introduced and compared the following automatic parallelization compilers: Cetus, Par4All, Pluto, Parallware [25], ROSE, and ICC. They discussed their different aspects of work fashion including dependence analysis techniques, supported languages, frameworks etc. In their work, they showed the speedup and points of failure of the compilers on ten NAS (Numerical Aerodynamics Simulation) Parallel Benchmarks [26] using the Gprof tool [27]. In order to deal with these points of failure, they proposed possible solutions that require manual intervention. Furthermore, they concluded that the compilers which require no manual intervention are Par4All and ICC; that the compilers that require minimal manual intervention are Cetus and AutoPar; and that the compilers that failed to parallelize the benchmark are Parallware and Pluto. Another work was done by Prema et al. [28], which reviewed two automatic parallelization compilers: Cetus and Par4All. Several study cases were introduced to analyze and evaluate both of these compilers' performances such as Matrix Multiplication. Execution time and speedup gained were compared between the automatic parallelization results of Cetus and Par4All and a manually OpenMP directed parallel program execution. The results showed that the manually OpenMP directed parallel program produced the best execution time and speedup. Furthermore, the authors concluded that Par4All and Cetus are not suited for nested loops. Both Aditi Athavale et al. [29] and Prema et al. [28] reviewed Cetus and Par4All. However, Aditi Athavale et al. [29] include the S2P compiler with a detailed explanation of its key features in their comparison, while Cetus and Par4All are explained briefly. The three compilers were tested with Perfectly Parallel NAS Parallel Benchmarks and Matrix Multiplication. The comparison criteria they used were performance, scalability, memory, time complexity, and also parallelization overhead. The authors concluded that although Par4All and Cetus managed to insert OpenMP directives, additional effort was needed in order to achieve the best performance.

For our study, after a broad examination of the current available parallelization compilers, we concluded that AutoPar, Par4All and Cetus are the most suitable compilers for automatic parallelization of scientific codes, which usually hold a large amount of arrays and array operations under multiple loops. The benefit of each compiler in service to this goal is explained in detail, and a pros and cons list is included. Our case study consists of Matrix Multiplication and NAS parallel benchmark, which are fundamental and representative scientific operation, and discussed thoroughly in Sect. 4. The three compilers' key features are shown by the table in Sect. 7.

3 Compilers Specifications

3.1 AutoPar

AutoPar is a module within the ROSE compiler [30], which is under ongoing development by Lawrence Livermore National Laboratory (LLNL). ROSE [13] is a source-to-source compiler which is implemented in a modular fashion such that additional specialized translators may be constructed and added to it. ROSE's front-end parses the input code and then builds an abstract syntax-tree, which can be manipulated by the translation modules. ROSE's Intermediate Representation (IR), also named SAGE III, is based upon SAGE II and uses Edison Design Group (EDG) front-end. SAGE II was developed specifically to C and C++, which means that ROSE internally supports Object Oriented Programming. ROSE currently supports C, C++, Fortran, Java, Python and PHP programming languages.

AutoPar [12] is an open-source automatic parallelization compiler for C and C++, included within ROSE. AutoPar can be used either to add new OpenMP directives to a serial code or to check the correctness of existing ones in a given parallel code. Although AutoPar can be used in an unmanaged manner, there is some information which is usually hard to automatically extract from the code such as aliases and possible side effects of functions. This information is crucial for optimizing the parallelization performances of the code. For these cases, the programmer may want to provide AutoPar with an annotation file describing the code's features.

AutoPar Pros and Cons

+ Inherently suitable for OOP.
+ Handles nested loops.
+ 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.

- Requires programmer intervention to handle function side-effects, classes etc. (via annotation file).
- 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 (for further explanation see Sect. 4.2.2).

3.2 Par4All

Par4All [15] is an open-source compilation framework which can be utilized for analysis and manipulation of C and Fortran programs. Par4All was developed by SILKAN, MINES ParisTech and Institute Télécom as a merge of some open-source development projects. The development of Par4All was shut-down by 2015. It specializes in inter-procedural analysis which can be used to understand data dependencies within the code and to validate correctness of code manipulations. This framework may also be used as a way to enforce coding standards, to ease the process of debugging, and to make code more maintainable. Par4All may be useful for migrating serial programs to both multi-core processors and GPGPUs [31]. Within a single command, Par4All automatically transforms C and Fortran sequential programs to parallel ones. It offers code execution optimization on multi-core and many-core architectures regardless of a particular programming language. Par4All supports CUDA paradigms.

Par4All includes some of PIPS [16] commands. PIPS is a source-to-source compiler that was built for scientific and signal processing applications. PIPS was developed in 1988 by the same teams from MINES ParisTech, CEA-DAM, Southampton University, Télécom Bretagne, Télécom SudParis, SRU (Slippery Rock University), RPI (Rensselaer Polytechnic Institute) and ENS Cachan.

Par4All Pros and Cons

+ Automatically analyzes function side effects and pointer aliasing.
+ Suitable for GPUs.
+ Supports many data types.
+ Supports Fortran, thus making it more suitable for scientific legacy codes.

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

3.3 Cetus

Cetus [18] is a compiler infrastructure for the source-to-source transformation of software programs. Cetus was developed by ParaMount research group at Purdue University. It currently supports C programs. Cetus compiler includes data dependent analysis, pointer alias analysis, array privatization, reduction recognition etc. (for further reading please refer to [32] or see below in Sect. 4). Cetus transformation phases include induction variable substitution pass, loop dependent analysis, loop parallelizer based on some of the compiler passes and more.

Automatic parallelization capability is an integral part of the Cetus compiler itself, unlike ROSE and PIPS—in which the automatic parallelization function is an external part of the compiler. Cetus is written in Java, and it contains a graphical user interface (GUI) and a client-server model, allowing users to transform sequential C code to parallel one via the server. Moreover, the client-server model gives the opportunity for non-Linux users to run Cetus automatic parallelization compiler. Cetus adds a condition which ensures that parallelization is done only for loops above 10,000 iterations, thus preventing parallelization overhead where a sequential run is enough. In cases of nested loop, the number of iterations of each loop segment will also include the number of iterations of its inner loops.

Cetus Pros and Cons

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

- Adds Cetus's pragmas which create excess code.
- May create reduction clauses that are unknown for standard compilers.
- Does not insert OpenMP directives to loops that contain function calls.

4 Comparison

In this section we present relevant definitions and criteria for the automatic parallelization compilers. We then compare the three compilers' outcomes, and point to the strengths and weaknesses of each compiler. Lastly, we present a runtime analysis of the parallelized code segments using different suitable hardware platforms.

4.1 Criteria and Terminology

Relevant definitions and criteria for automatic parallelization compilers comparison include—but are not limited to—the following:

- Definitions:
  - *Pointer Aliasing* Refers to the situation where the same memory location can be accessed using multiple different names (i.e pointers).
  - *Function Side Effect* A function is said to have a side effect if it modifies addresses accessible outside its scope (i.e has any interaction with the rest of the program besides a return value).
  - *Array Privatization* The act of declaring an array as a private variable in a parallel section.
  - *Loop Unrolling/Unwinding* A loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size (excess code). An example to this optimization is shown in loop unroll test (Listing 7)
  - *Reduction Clause* An OpenMP clause that performs a reduction operator on a variable or an array.
- Criteria:
  - *No-Aliasing Option* A flag notifying the source-to-source compiler that there are no pointer aliasing in the code.
  - *Supported Language* The set of programing languages on which the source-to-source compilers can operate.

- *Array Reduction/Privatization* How does the source-to-source compiler behave when a parallel directive can only be added with an array reduction clause or with an array declared as private?
- *Loop Unroll Support* Will the source-to-source compiler insert OpenMP directives to unrolled loops?
- *Double Check Alias Dependence* Does the source-to-source compiler check for dependencies other than pointer aliasing when the No-Aliasing option is turned on?
- *Function Call Support* Will the source-to-source compiler insert OpenMP directives to loops containing function calls with/without side effects?
- *Nested Parallelism* How does the source-to-source compiler behave on nested loops?
- *OOP Compatible* Does the source-to-source compiler support Object Oriented Programming languages and concepts?

4.2 Test-Cases

Several input source codes were 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 codes are Matrix Vector Multiplication and Matrix Multiplication [33], the latter's source code is presented below. Several variants of these problems were created, with each variant pointing to a different parallel shared memory management pitfall/obstacle for the purpose of the compilers' dependence analysis. The Matrix Vector Multiplication source code was not included due to the similarity in the OpenMP directives to the Matrix Multiplication source code.

Listing 1 Serial matrix multiplication code for the nested loop and reduction test-case.

```
void mat_mul(int n, int **a, int** b, int** c) {
  int i,j,k;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      for (k = 0; k < n; k++) {
        c[i][j] += a[i][k] * b[k][j]; }}}
  return;}
```

Listing 2 Par4All output for the nested loop and reduction test-case.

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

4.2.1 Nested Loops and Array Reduction

The test on Listing 1 contains a nested loop which implements a simple naïve Matrix Multiplication calculation. While Par4All (Listing 2) inserted OpenMP directive solely to the outermost loop, AutoPar (Listing 3) was able to insert OpenMP directives to the two outermost loops, and Cetus (Listing 4) managed to insert OpenMP directives to all of the loops.

Listing 3 AutoPar output for the nested loop and reduction test-case.

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

Listing 4 Cetus output for the nested loop and reduction test-case.

```
void mat_mul(...) {
  int i, j, k;
#pragma cetus private(i, j, k)
#pragma loop name mat_mul#0
#pragma cetus parallel
#pragma omp parallel for if((10000<(((1L+(3L*n))+((4L*n)*n))+(((3L*n)*n)*n))))
↪   private(i, j, k)
  for (i=0; i<n; i ++ ) {
#pragma cetus private(j, k)
#pragma loop name mat_mul#0#0
#pragma cetus parallel
#pragma omp parallel for if((10000<((1L+(4L*n))+((3L*n)*n)))) private(j, k)
    for (j=0; j<n; j ++ ) {
      c[i][j]=0;
#pragma cetus private(k)
#pragma loop name mat_mul#0#0#0
#pragma cetus reduction(+: c[i][j])
#pragma cetus parallel
#pragma omp parallel for if((10000<(1L+(3L*n)))) private(k) reduction(+: c[i][j])
      for (k=0; k<n; k ++ ) {
        c[i][j]+=(a[i][k]*b[k][j]); }}}
  return ; }
```

4.2.2 Pointer Aliasing

Listing 5 contains a pointer dereference test-case. This specific test-case caused Cetus to step into an internal error. When given the No-Aliasing option, AutoPar ignored all data dependencies and inserted an incorrect OpenMP directive to the innermost loop (Listing 6). Par4All did not manage to insert any OpenMP directive (even with the No-Aliasing option). However, if one changes the arrays from a static allocation to a dynamic one, i.e. a pointer to a matrix, Par4All will insert OpenMP directive when given the No-Aliasing option. Since Cetus and Par4All did not manage to insert the directives in both allocation modes, their codes are not presented.

Listing 5 Serial Code for the pointer dereference test-case.

```c
void mat_mul_pointer_alias(...) {
  int i,j,k;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      for (k = 0; k < n; k++) {
        (*(c+i))[j] += (*(a+i))[k] * b[k][j]; }}}
  return;}
```

Listing 6 The innermost loop of AutoPar's output for the pointer dereference test-case which is wrong.

```c
#pragma omp parallel for private (k)
for (k = 0; k <= n-1; k += 1) {
  ( *(c + i))[j] += ( *(a + i))[k] * b[k][j]; }
```

4.2.3 Loop Unrolling

In Listing 7 we created a version of the original *mat_mul* function where in each iteration we changed four consequent array cells. AutoPar did not manage to insert any OpenMP directives, thus the code it has generated is not included. Par4All parallelized the outermost loop as before and managed to insert the correct directive, therefore the code generated by it is not presented for convenience reasons. As before, Cetus added OpenMP directives to all three loops. However, Cetus's output for the innermost loop (Listing 8) 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. On the runtime analysis section we omitted this invalid OpenMP directive.

Listing 7 Serial code for the loop unrolling test-case.

```c
void mat_mul_loop_unroll(...) {
  int i,j,k;
  for (i = 0; i < n-1; i+=2) {
    for (j = 0; j < n-1; j+= 2) {
      for (k = 0; k < n-1; k += 2) {
        c[i][j] += (a[i][k] * b[k][j] + a[i][k+1] * b[k+1][j]);
        c[i][j+1] += (a[i][k] * b[k][j+1] + a[i][k+1] * b[k+1][j+1]);
        c[i+1][j] += (a[i+1][k] * b[k][j] + a[i+1][k+1] * b[k+1][j]);
        c[i+1][j+1] += (a[i+1][k] * b[k][j+1] + a[i+1][k+1] * b[k+1][j+1]); }}}
  return; }
```

Listing 8 The innermost loop of Cetus Output for the loop_unrolling test-case.

```c
#pragma cetus private(k)
#pragma loop name mat_mul_loop_unroll3#0#0#0
#pragma cetus reduction(+: c[i+1][j+1], c[i+1][j], c[i][j+1], c[i][j])
#pragma cetus parallel
#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])
for (k=0; k<(n-1); k+=2 {
 ... };
```

4.2.4 Function Calls

On another test-case (Listing 9), the innermost instruction was isolated into a function and replaced with a function call. Using this code arrangement, the matrix multiplication contains a function-call side effect. For this test-case, AutoPar couldn't parallelize the code without being fed with an annotation file. Cetus always assumes that function-calls will cause unexpected data manipulations and tries to evade these side-effects by avoiding any OpenMP directives. Meanwhile, Par4All handles function side-effects automatically and manages to insert the OpenMP directive to the outermost loop as before (Listing 10). Since AutoPar and Cetus did not insert OpenMP directives, their code is not presented.

Listing 9 Serial code for the function call test-case.

```
void compute_cijk(int i, int j, int k, int a[N][N], int b[N][N], int c[N][N]) {
  c[i][j] += a[i][k] * b[k][j];}
void mat_mul_function_calls(...) {...
        compute_cijk(i,j,k,a,b,c);
    return;}
```

Listing 10 Par4All output code for the function call test-case.

```
#pragma omp parallel for private(j, k)
  for(i = 0; i <= n-1; i += 1)
    for(j = 0; j <= n-1; j += 1) {
      for(k = 0; k <= n-1; k += 1)
        compute_cijk(i, j, k, a, b, c); }
```

4.3 Runtime Analysis

All the above-mentioned test-cases were compiled using Intel(R) C Intel(R) 64 Compiler for applications running on Intel(R) 64, Version 18.0.1.163 Build 20171018 using internal optimizations (i.e. -O3), and executed on a machine with two Intel(R) Xeon(R) CPU E5-2683 v4 process units with 16 cores each. We also executed these tests on an Intel(R) Xeon-Phi co-processor 5100 series (rev 11) in native mode. As far as we know, this kind of performance comparison of the of automatic parallelization compilers over Xeon-Phi co-processors is the first of its kind.

Figure 1 compares the runtime of each output file generated by the three automatic parallelization compilers (AutoPar, Par4All, Cetus) and the original sequential source code. The functions (y-axis) are the code segments described in Sect. 4. As seen in 1, Par4All's overall performance is the best. This is because AutoPar and Cetus generated OpenMP directives in the inner-loops which creates a slight overhead. Cetus inserts a reduction clause on an array which also creates overhead. However, one can disable this feature and avoid inserting a reduction clause on arrays. AutoPar has the worst performance in *_loop_unroll* because the compiler did not insert any OpenMP directives on loops that have been manually unrolled.

Figure 2a compares the execution time of the parallelized and serial *mat_mul* test-case on Xeon processors (represented by circle markers on the figure) and Xeon-Phi

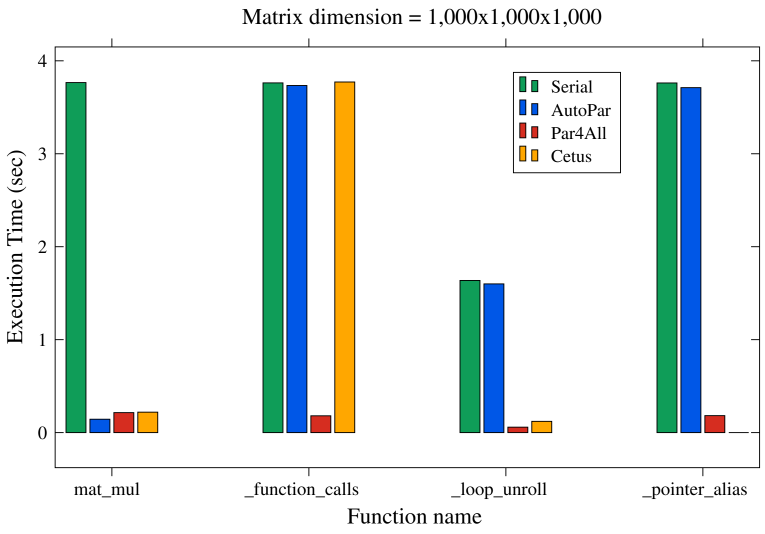


Fig. 1 Functions execution time generated by the three compilers and serial execution

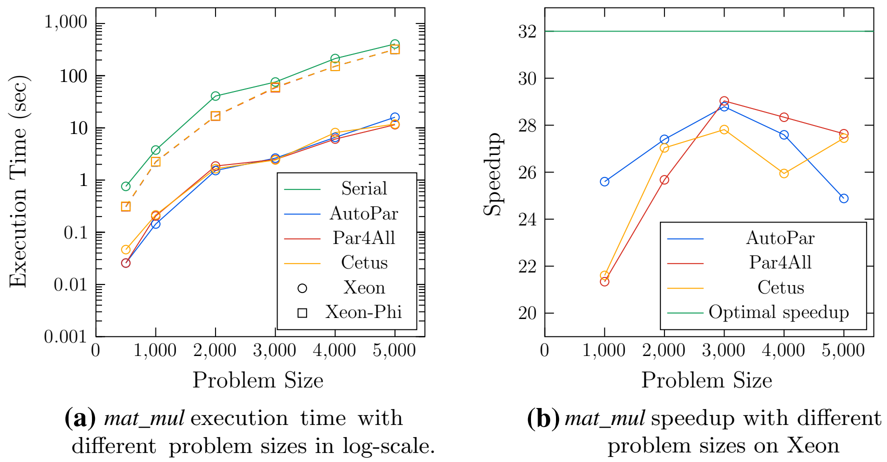


Fig. 2 Execution time and speedup comparison on Xeon and Xeon-Phi

co processors (represented by square markers on the figure) with the "-O3" option. As one may notice, the performance of the parallel execution on MIC architecture is worse than on x86_64. Figure 2b compares the speedup gained by the parallel execution on Xeon to the serial execution, the results scale well as they are on average above x25 while the optimum is x32 which is the number of threads.

The next Fig. 3 compares the speedup gain from executing the parallel source codes on a Xeon-Phi co-processor with the options "-O3" and "-O0". The reason for compiling with the "-O0" option is to demonstrate the source-to-source compilers' capabilities without any source-to-machine language compiler optimization. The difference between the "-O3" and "-O0" and other options, is further discussed in Chae

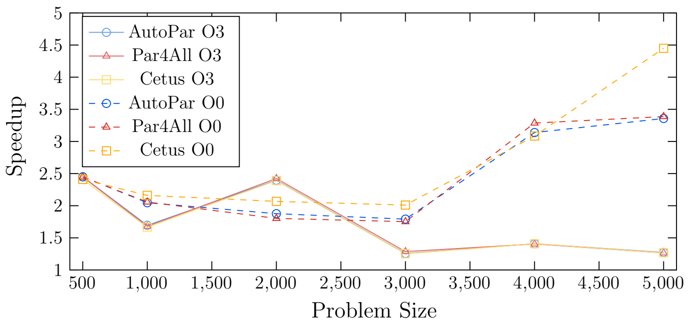


Fig. 3 *mat_mul* speedup on Intel Xeon-Phi compared to serial run (on Intel Xeon) with both -O3 and -O0 optimizations

Jubb paper [34]. The speedup gain with the option "-O3" is lower than the option "-O0", this is because the option "-O3" improves the execution time of the serial run more that it improves the parallel. The performance imbalance with the option "-O0" happens mainly because the overhead that is caused by creating many threads for each loop, outweighs the benefit we gain from the parallelization itself. Once the benefit of parallelizing the computation over many threads overcomes the overhead of the creation of a thread itself, we can see an increase in speedup (which occurs when N $>$ 3000). The results from Figs. 2a and 3 imply that one should not solely rely on automatic parallelization compilers when optimizing code for execution on Xeon-Phi. Previous work showed that efficient optimization for native MIC involves vectorization [35], tailoring of the perfect algorithm [36] and careful instructions arrangement [37]. These optimizations are yet beyond the capabilities of the automatic parallelization compilers.

To further understand the abilities of the compilers to maximize performance both on co-processors and processors, we present a Vector Matrix Multiplication test case. In this test we balanced the problem size with the number of repetitions to allow a fixed number of floating point operations in all tests. The focus of this test is to show the impact of memory usage over the L1, L2 and L3 cache sizes while maintaining the same workload operation-wise. The first execution size contains one matrix size of 240 elements which sums to 233 kB, while our L1 cache is 64 kB and L2 cache size is 256 kB. The second execution size contains 3000 elements which sums up to 3604 kB, while our L3 cache is 41 mB. The third execution size contains 15,180 elements which sums to 1 gB. The results show in Fig. 4 that the fastest execution time for Xeon is with 3000 elements. This is because the number of cache misses is the smallest (the problem's size is smaller than the L3 cache size). We can see that for 15,180 elements test case the Xeon-Phi outperformed the Xeon, which can indicate that Xeon-Phi is beneficial for larger inputs.

5 Accelerators and Co-processors

In this section we test Par4All's ability to transform C code to CUDA for GPUs. To better suite this study case, we slightly modified the previous matrix multiplication code see Listing 11.

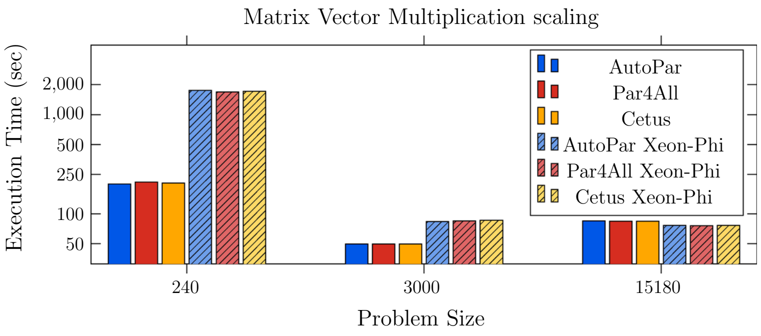


Fig. 4 Functions execution time generated by the three compilers and serial execution

- The new matrix is now allocated as a N^2 dimensional vector of integers instead of a N dimensional vector of pointers to N dimensional vectors of integers.
- We replaced the dynamic parameter n that holds the dimension of the matrix, with a static definition of N known at compilation time (i.e #define).
- We assume the matrix B is transposed, which allows us to access it by row order, as is the access to the matrix A.

The above changes are required for Par4all to support the code transformation to CUDA and to enable better compiler optimization and memory management. Par4All's GPU accelerator support relies on an API called P4A_ACCEL that 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 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) V100-PCIE-32GB.
- *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).

Listing 11 Matrix multiplication code for the GPU case study

```
void mat_mul(int* a, int* b, int* c) {
    int i,j,k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            c[i*N+j] = 0;
            for (k = 0; k < N; k++) {
                c[i*N+j] += a[i*N+k] * b[j*N+k];
            } } } return; }
```

The native compilation scheme is to firstly allocate the desired space on the GPU for each parallel loop nest, transfer the computation data to the GPU and finally launch the kernel and copy back the results from the GPU. Par4Alls output for GPU accelerators is shown in Listing 12. Par4All's code transformation to OpenMP remains the same as before see Listing 2. The runtime and speedup results of these tests are shown in Fig. 5.

The results below show a great differentiation between the different platforms. Although the GPU V100 is about seven-eight times stronger in terms of flop/s than the Xeon-Phi KNC, the KNC outperformed the V100. We can also see that since the automatic code parallelization cannot transform the code into a vectorized code, performances did not improve in comparison to the Xeon processor, and were even worsen over an increase of the matrix size. We can also notice that the Xeon processor's speedups stay steady as we increase the matrix size, but they are worsen in the Xeon-Phi in a semi-linear fashion.

Therefore, we conclude that though these source-to-source automatic parallelization compilers can achieve improvements to shared-memory platforms, they still cannot achieve the same kind of improvements to more complicated architectures such as accelerators or co-processors. This is mostly due to the lack of vectorization support by these compilers. These architectures are still required to be programmed in an ad-hoc fashion to a specific problem by an expert programmer that can take many parameters into account as parallelization is being performed.

Listing 12 Matrix multiplication code for the GPU transformed by Par4All to Accel (CUDA)

```
void P4A_accel_malloc(void **address, size_t size); //PIPS generated
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); //PIPS generated
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); //PIPS generated
void P4A_accel_free(void *address); //PIPS generated
P4A_accel_kernel_wrapper p4a_wrapper_mat_mul(int i, int j, int *a, int *b, int *c) {
   i = P4A_vp_1; // Index has been replaced by P4A_vp_1
   j = P4A_vp_0; // Index has been replaced by P4A_vp_0
   p4a_kernel_mat_mul(i, j, a, b, c); }
P4A_accel_kernel p4a_kernel_mat_mul(int i, int j, int *a, int *b, int *c) {
   int k; // Declared by Pass Outlining
   if (i<=999&&j<=999) {
      c[i*1000+j] = 0;
      for(k = 0; k <= 999; k += 1)
         c[i*1000+j] += a[i*1000+k]*b[j*1000+k]; } }
void p4a_launcher_mat_mul(int *a, int *b, int *c) {
   int i, j;  // Declared by Pass Outlining
   P4A_call_accel_kernel_2d(p4a_wrapper_mat_mul, 1000, 1000, i, j, a, b, c); }
void mat_mul(int *a, int *b, int *c) {
      int (*p4a_var_c0)[1000000] = (int (*)[1000000]) 0, (*p4a_var_b0)[1000000] = (int
      ↪  (*)[1000000]) 0, (*p4a_var_a0)[1000000] = (int (*)[1000000]) 0;
      P4A_accel_malloc((void **) &p4a_var_a0, sizeof(int)*1000000); // Same for b0, c0
      P4A_copy_to_accel_1d(sizeof(int), 1000000, 1000000, 0, &a[0], *p4a_var_a0); //
      ↪  Same for b0, c0
      p4a_launcher_mat_mul(*p4a_var_a0, *p4a_var_b0, *p4a_var_c0);
      P4A_copy_from_accel_1d(sizeof(int), 1000000, 1000000, 0, &c[0], *p4a_var_c0);
      P4A_accel_free(p4a_var_a0); // Same for b0, c0
   return; }

```

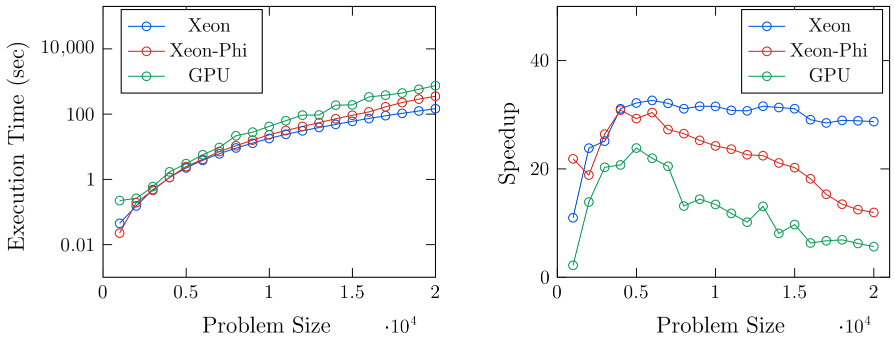


Fig. 5 Execution time and speedup comparison of Par4All output on Xeon and Xeon-Phi and GPU

6 NAS Parallel Benchmark

To further evaluate the compilers capabilities, we introduce a more practical test case. The Numerical Aerodynamics Simulations (NAS) Parallel Benchmarks [26], developed and maintained by NASA, are a group of applications created to evaluate the performance of HPCs. The NAS Parallel Benchmarks include ten different benchmarks [38]. The benchmarks that were included to evaluate the compilers are 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) and Unstructured Adaptive mesh (UA). The benchmarks Fourier Transform (FT) and Integer Sort (IS) were excluded from this study due to the inability of AutoPar and Par4All to process them. Each of the benchmarks was processed by the three compilers and compiled using Intel(R) C Intel(R) 64 Compiler for applications running on Intel(R) 64, Version 18.0.1.163 Build 20171018 with a fixed class size defined by Class=C.[2]

6.1 NAS Results

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 loops, such as Listing 13 and the insertion of OpenMP directives in a nested loop manner, such as Listing 14.

Listing 13 AutoPar's directives on computationally small loops EP/ep.c.

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

[2] Source code of relevant sections can be found at: *github.com/reemharel22/AutomaticParallelization_NAS*

Listing 14 AutoPar's directives on nested loop and low iterative loops SP/rhs.c.

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

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 Listing 15.

Listing 15 Par4All's directives on computationally small and nested loops

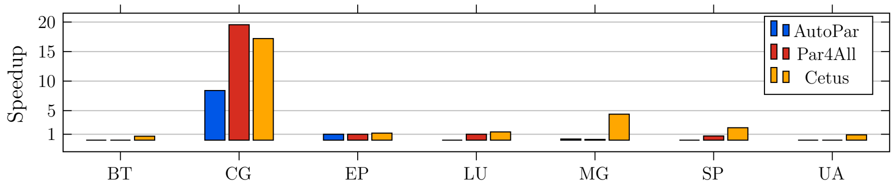
```
#pragma omp parallel for
   for(m = 0; m <= 4; m += 1)
      errnm[m] = 0.0;

   for(k = 1; k <= nz-1-1; k += 1)
      for(j = jst; j <= jend-1; j += 1)
         for(i = ist; i <= iend-1; i += 1) {
            exact(i, j, k, u000ijk);
#pragma omp parallel for private(tmp)
            for(m = 0; m <= 4; m += 1) {
               tmp = u000ijk[m]-u[k][j][i][m];
               errnm[m] = errnm[m]+tmp*tmp; } }
```

Cetus As mentioned above, Cetus ensures that the number of loop iterations is above ten thousand, combining this feature 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. Although this feature was found to be powerful in CG, EP, LU, MG, SP benchmarks, this is not the case in UA and BT benchmark, in which it was found to be insufficient in cases of a single loop such as Listing 16. UA and BT benchmark with the removed directives (single loops) speedup results are shown in Fig. 6b.

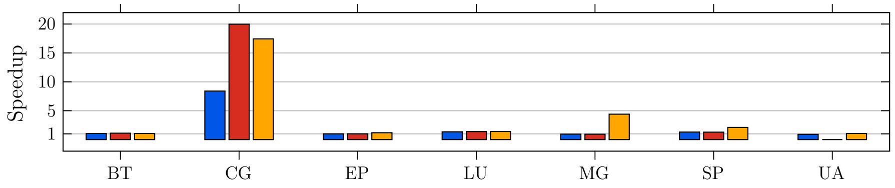
Listing 16 Cetus directive in BT/x_solve.c.

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

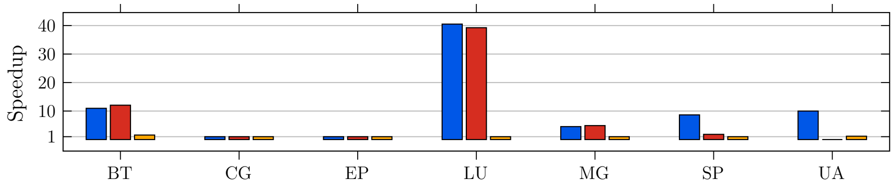
In order to show a more realistic scenario which involves the compilers' output and minimal human intervention. The unnecessary OpenMP directives, such as Listing 14, created by the three compilers were **manually removed**. Figure 6a presents the compilers' speedup compared to the serial run of the benchmarks whereas Fig. 6b and 6c compare the compilers' performance after removing the unnecessary directives to the serial run and to the compilers' performance with those directives respectively. We can see that by removing the unnecessary OpenMP directive we gain speedup in most test-cases. These results imply that although the compilers results performed worse than the serial execution in some cases, we can still gain speedup with minimal human



(a) Speedup gained from the compilers alone on the tested benchmarks compared to serial execution.



(b) Speedup gained from removing the unnecessary OpenMP directives compared to serial execution.



(c) Speedup gained from removing the unnecessary OpenMP directives compared to non-removal.

Fig. 6 NAS benchmark results

intervention by removing redundant directives. We can see that the compiler with the best results is Cetus. As one can see CG and EP benchmarks are the exceptions that disprove the rule, meaning without human intervention we still achieve the compilers' maximal speedup.

6.2 OpenMP NAS Benchmark

In order to maximize the compilers' performance on the NAS benchmark—and by that to fully understand the parallelization compilers performances—we introduced AutoPar, Par4All and Cetus to a parallel-suitable NAS benchmark [39], which was code-designed to achieve peak performances using OpenMP. The OpenMP NAS benchmarks tackle the same NAS serial benchmark physical and mathematical problems, with differences in the code structure, such as including OpenMP directives in the most suitable spots, pre-declaring global variables as shared or private if necessary, changing a loop to be more suitable for a reduction clause and so forth. The notion for testing the compilers on a code that was re-designed to be more suitable for shared-memory parallel run is obvious, as serial code not always enables achieving peak performances without the necessary adaptations. Therefore, the compilers should yield better performances on the NAS parallel-suitable codes. However, the performances results were almost identical to the results of the compilers with the serial benchmark. The following Fig. 7 compares the compilers' speedup on the OpenMP

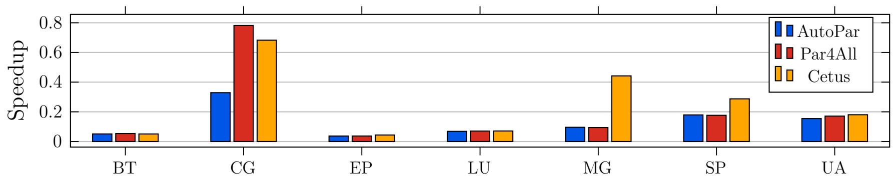


Fig. 7 Speedup gained from removing the unnecessary OpenMP directives compared to the man-OpenMP code

benchmark, after removing unnecessary OpenMP directives, as was discussed above in Sect. 6.1, to the OpenMP benchmark ($\frac{Tool\ Runtime}{OpenMP\ Runtime}$).

Although the code is different and more suitable for shared-memory parallelization, the compilers failed to take advantage of this parallelization adaptation in the code structure and did not gain any significant performance improvements. A comprehensive analysis of the causes to the gap between the original OpenMP code to the compiler's OpenMP code can be found in the next section.

6.3 The Gap Between and Pitfalls of the Compilers

As known, the NAS OpenMP parallel benchmark contains many nested loops with an OpenMP directive on the outermost loop, which ensures minimal overhead from thread creation while maximizing the workload of each thread (for example, Listing 17). In some of these cases, the compilers failed to insert an OpenMP directive to the outermost loop or insert an OpenMP directive at all, as shown in Listing 16. The insertion of the directives into the inner loops resulted in smaller workloads for each thread, and therefore it increased the overhead penalty to the total runtime. The reasons for the gap between the original OpenMP parallelized code and the compilers' OpenMP code and between the compilers themselves, can be summarized in the following points:

Internal Logic of the Code The OpenMP parallel benchmark contains code segments that are wrapped within a parallel region directive. Inside the parallel region, a team of threads is created once, and then is being executed on the same code segment. This is helpful in cases of consecutive loops that can be parallelized at once, thus minimizing the overhead caused by creating threads over and over again. Furthermore, this one-time-thread-creating parallel region allows the usage of the *nowait* clause, an example of this is shown in Listing 18. Another example is when there is an assignment of individual work for each individual thread, as shown in Listing 19. Cases like Listing 18 and Listing 19 are hard to predict without knowing the internal logic of the code, which is the reason for the failure of the compilers to insert an OpenMP directive at all. The lack of the internal logic of the code might cause additional failures in producing OpenMP directives due to the difficulty of predicting or understanding the pattern of the code. For example, in Listing 20 the compilers can't predict the values of the array *rowstr* thus, the compilers must assume the worst case which is the values of the array contain overlapping range.

Array Privatization Many of the OpenMP directives in the NAS OpenMP code include an array privatization. As mentioned, unlike Cetus, AutoPar and Par4All support only scalar variable privatization and not array privatization. Thus, AutoPar and Par4All failed to insert OpenMP directives into *for*-loops that can be parallelized with the array privatization option while Cetus managed to do so successfully. Thanks to this feature, Cetus managed to produces the best results among the three compilers in most of the NAS benchmarks (Fig 6a, 7). The only exception test case is the CG test case, as the original CG OpenMP code did not include array privatization options.

Function Calls Although Cetus and Par4All analyze function side effects automatically, in the NAS benchmark the analysis of function side effects were found to be insufficient. To analyze function side effects, the compiler needs the source code of the function and not the compiled form of the function at the compiler's processing stage. This is due to the fact that these compilers are source-to-source compilers. Even if the function's source code is provided to the compiler, in some cases it fails to analyze the side effect correctly, thus fails to insert an OpenMP directive entirely.

Listing 17 OpenMP parallel region in BT/x_solve.c.

```
#pragma omp parallel for default(shared) shared(isize) private(i,j,k,m,n)
  for (k = 1; k <= grid_points[2]-2; k++) {
    for (j = 1; j <= grid_points[1]-2; j++) {
      for (i = 0; i <= isize; i++) {
        ....}}}
```

Listing 18 OpenMP parallel region in MG/mg.c.

```
#pragma omp parallel default(shared) private(i1,i2,i3) {
    #pragma omp for
    for (i3 = d3; i3 <= mm3-1; i3++) {
        ...}
    #pragma omp for nowait
    for (i3 = 1; i3 <= mm3-1; i3++) {
        ...}
        }
```

Listing 19 OpenMP parallel region in EP/ep.c.

```
#pragma omp parallel default(shared) private(k,kk,t1,t2,t3,t4,i,ik,x1,x2,l) {
    for (i = 0; i < NQ; i++)
      qq[i] = 0.0;
    #pragma omp for reduction(+:sx,sy) nowait
    for (k = 1; k <= np; k++) {
      kk = k_offset + k;
      ... } ...}
```

Listing 20 Array values defined in runtime in CG.

```
#pragma omp parallel for
for (j = 0; j < lastrow - firstrow + 1; j++) {
      for (k = rowstr[j]; k < rowstr[j+1]; k++) {
        colidx[k] = colidx[k] - firstcol; } }
```

6.4 Overcoming the Pitfalls in The Limits of OpenMP 2.5

In order to overcome the pitfalls that were encountered by the compilers, and to minimize the gap between the original OpenMP code to the code produced by the compilers, we suggest several changes in the original OpenMP code that will help the compilers produce better results, i.e additional directives to additional loops, with minimal human intervention. However, there are some pitfalls that the presented compilers cannot overcome due to their inability to produce the directive such as defining a parallel region, the *nowait* directive, array privatization option (although in specific cases loop transformations can be applied to eradicate the array or, if possible, by declaring the array inside the loop thus, implicitly declaring the array as private). It is worth noting that the changes we suggest can be used for the general case of these pitfalls and not only for the NAS benchmark.

Array Privatization Although Cetus can produce an OpenMP directive with array privatization, it is still difficult to predict an array privatization case. Thus, if one has prior knowledge whether the loop can be parallelized with the array privatization option, the array can be declared inside the loop. Thus, implicitly declaring the array as a private array. By implicitly declaring the array as private, we also overcome Par4All's and AutoPar's pitfall - in which these compilers cannot produce an array privatization option at all. An example of producing an OpenMP directive by implicitly declaring an array as a private is shown in Listing 21.

Function Calls As mentioned, analyzing function side effects can be difficult and not all the compilers support this option. In order to overcome the function side effect pitfall, it might be helpful to 'inline' the function's source code to the loop itself. As seen in Listing 22, by inserting the function's source code Cetus managed to produce the correct OpenMP directive.

Listing 21 Implicitly declaring the array as private in BT/x_solve.c.

```
#pragma omp parallel for if((10000<((((((((1940L*grid_points[0L]-1195L+))+(592L *
↪   grid_points[1L]))+(598L*grid_points[2L]))+((grid_points[0L]*(-970L))
↪   *grid_points[1L]))+((grid_points[0L]*(-970L))*grid_points[2))+((grid_points[1L]*(-296L))
    *grid_points[2L]))
↪   +(((485L*grid_points[0L])*grid_points[1L])*grid_points[2L])))) private(i, ii, j, k,
↪   m, n) lastprivate(tmp1, tmp2, tmp3)
    for (k=1; k<=(grid_points[2]-2); k ++ ) {
        double lhs_r[(162+1)][3][5][5];
        double fjac_r[(162+1)][5][5];
        double njac_r[(162+1)][5][5];
        ...}
```

Listing 22 Inserting binvcrhs source code to BT/x_solve.c main loop.

```
    ...
/*  binvcrhs( lhs_r[i][BB], lhs_r[i][CC], rhs[k][j][i] ); */
    pivot=(1.0/lhs_r[i][1][0][0]);
    lhs_r[i][1][1][0]=(lhs_r[i][1][1][0]*pivot);
    lhs_r[i][1][2][0]=(lhs_r[i][1][2][0]*pivot);
    lhs_r[i][1][3][0]=(lhs_r[i][1][3][0]*pivot);
    lhs_r[i][1][4][0]=(lhs_r[i][1][4][0]*pivot);
    ...
```

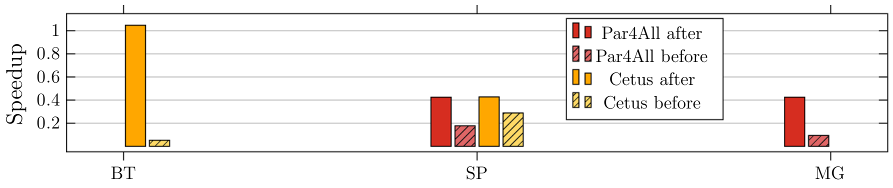


Fig. 8 Speedup gained before and after applying the two techniques, compared to the humanmade OpenMP code

In order to test the techniques discussed above, we apply the two techniques discussed above on three NAS benchmarks—BT, SP and MG. We note that the targeted functions were chosen by identifying them as their high execution time with the scalasca tool [40] and the work done by Prema et al. [24]. We targeted the three source codes in BT benchmark—*x_solve*, *y_solve*, *z_solve* due to their high execution time. By applying the two techniques while maintaining the correctness of the code, Cetus managed to insert the correct and optimal directive in the new source codes while Par4All and AutoPar failed to produce the same result due to long analyzing time which resulted in the compilers failing in mid-process. A similar procedure was done for the two other benchmarks—SP and MG. In SP, Par4All and Cetus managed to insert the correct and optimal directive while AutoPar failed to produce the same directive. In MG applying the techniques did not help Cetus to produce better results, because the compiler already inserted the correct directives and avoided these pitfalls. Nonetheless, Par4All did overcome the pitfalls and managed to insert additional OpenMP directives while AutoPar failed to overcome the pitfalls. Figure 8 presents the speedup of the compilers compared to the human-made OpenMP code. The bold bars represent the speedup gained after applying the discussed techniques, while the striped bars represent the speedup gained before applying these techniques (as in Fig. 7). As one can notice, Cetus's BT execution time is shorter than the human's. This is due to the minimization of function calls in the new source code, which resulted in faster execution time.

7 Conclusions and Future Work

7.1 Conclusions

In this paper, we first described the need for parallelism in modern architectures and the difficulty of integrating it in existing codes. By that, we stressed the need for automatic parallelization compilers. We then presented the current automatic parallelization compilers and briefly discussed their history, work-fashion and pros and cons. Afterward, we chose three compilers which we found to be most suitable for parallelization of scientific codes. We also presented several test-cases, with different variations such as the Matrix Multiplication and the NAS benchmark, and presented the relevant output files generated by the three compilers. We also suggested some changes to the code base to help the three compilers insert more OpenMP directives. Finally, the compilers were compared based on their performance in different test-

Table 2 Table summary of the three compilers key features

Feature	AutoPar (ROSE)	Par4All (PIPS)	Cetus
Loop unrolling	No	Yes	Yes
Supported language	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

cases on three different suitable hardware architectures. We summarize the compilers' work fashion as follows:

1. AutoPar's annotation file is a powerful compiler, especially for function side effect, structures and object-oriented programming, which makes it more suitable for bigger projects or OOP-based projects. However, AutoPar's No-Aliasing option is dangerous and should be used with caution.
2. Although Par4All is somewhat out-dated, it is still a forceful compiler that can handle most of the cases automatically with minimal user intervention. Par4All also supports Fortran, thus making it more suitable for scientific legacy codes.
3. Cetus can provide a great service not only to Linux users but also for other operating system users via their GUI and client-server model. Cetus is able to generate reduction clauses on arrays as seen in Listing 8, though this reduction clause may be invalid, resulting in a crucial problem to compile the code.

The following table summarizes the three compilers by pointing out their key features (Table 2).

7.2 Future Work

7.2.1 Adapt to OpenMP 4.5

The three automatic parallelization compilers produce OpenMP directives that utilize only the features of version 2.5. However, over the years OpenMP evolved from a simple API that is suited for regular for-loops to an API that includes hundreds of different directives that support many different architectures. These new directives, specifically the ones included since OpenMP version 4.0 [41], attempt to utilize complicated architectures such as NUMA, co-processors, and accelerators. Thus, the compilers can greatly benefit and enhance their performance by introducing the new OpenMP features to their base code.

A new directive added in OpenMP 4.0 is the *SIMD* directive which allows the execution of the same operation on multiple data elements simultaneously. Using this directive, the processing time of the operation can be potentially reduced by a factor of the vector length. The compilers can greatly benefit from this directive since it can be added regardless of any other OpenMP directive, i.e it can be added to low-iteration loops (without the parallel-for directive) and still gain speedup as vector units are an integral part of new CPUs and obviously of co-processors and accelerators.

Although compilers can identify and execute *SIMD* loops themselves, they still may encounter challenges that will cause them to not automatically execute the *SIMD* directive such as imprecise dependence information, conditional execution, calls to functions, loop bounds that are not always a multiple of the vector length etc [42,43]. To overcome these challenges, the manual *SIMD* directive was introduced, allowing the compilers to take advantage of this directive by helping (or forcing) the compiler to execute the *SIMD* instruction.

Another directive in which the compilers can greatly benefit from is the *offload* directive. With the offload directive, the specified code can be easily offloaded to an accelerator or to a co-processor in cases where the *SIMD* code is intense, thus making the compiler produce codes that are suited for heterogeneous architectures. As seen in Sect. 5 Par4All can produce a code suited for accelerators in CUDA-Nvidia. Since OpenMP 4.0 the directives suited for accelerators (CUDA) can be replaced by simple OpenMP directives, thus making it easier to identify and insert codes suited for accelerators. This also allows the parallelization compilers to be familiar with only OpenMP directives. Introducing OpenMP offload also allows combining existing OpenMP directives such as the *SIMD* instruction, reduction clause, etc with the directive itself. The offload instruction—copying the memory and the code to the accelerator, executing, copying the memory back to the host—is an expensive task. Therefore, it is advisable to include with the offload some sort of if-clause (which Cetus already includes for parallel-for directives), thus avoiding the overhead caused by offloading in some cases. In addition, the compiler can easily detect which parameters should be copied to the device and which should be copied and returned from the device, and put the *to* and *tofrom* clauses respectively. Thus, avoiding the overhead of redundant copies from the device.

Listing 23 Example of loop suited for parallelization with *SIMD* and the *target* directive

```
void initialize_x_and_y(double *x, double *y, int N){
        for (int i=0; i<N; ++i) {x[i] = i*0.1; y[i] = i*0.2; }}

int main(int R) {
    double *x, *y;
    double alpha = 2.0;
    int N = 1024*1024*256;
    int R = 8192;
    int k,i;

    x = (double *) malloc (N * sizeof(double));
    y = (double *) malloc (N * sizeof(double));

    initialize_x_and_y(x,y,N);

    for(k=0; k<R; k++)
        for (i=0; i<N; i++)
            y[i] = alpha * x[i]; }
```

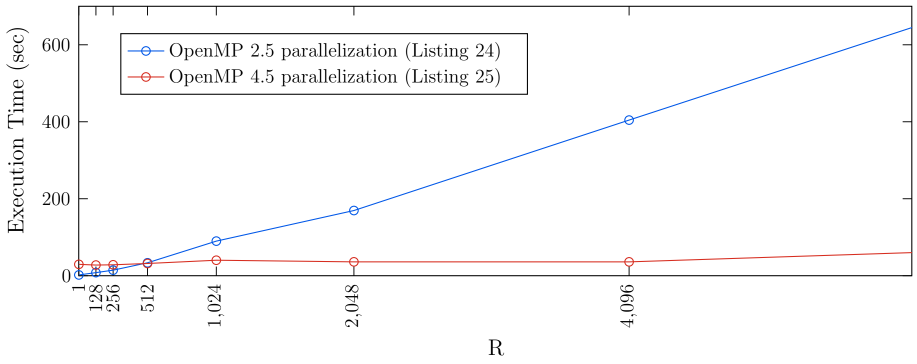


Fig. 9 Execution time of the programs presented in Listing 24 and in Listing 25

Listing 23 shows a serial code example in which an adaptation to OpenMP 4.5 directives can greatly elevate the entire performances. Currently, applying the parallelizators will only create a (i) *#pragma omp parallel for* between lines 1 and 2, and between 16 and 17, achieving simple loop parallelization (Listing 24 by Cetus, for example). However, introducing *SIMD* can create in this case a (ii) *#pragma omp for simd* between line 16 and line 17 instead, and by that to obtain much better performances. In cases where the integer N is big enough and/or the amount of vectored operations is substantial, one can offload the code to an available co-processor/accelerator using (iii) *#pragma omp target teams distribute parallel for simd* instead of (ii).

Listing 25 demonstrates (with the needed clarity for presentation) how the code in listing 24 would look like if the compilers have implemented OpenMP 4.5 directives. In order to demonstrate the potentially gained performances by the introduction of OpenMP 4.5 to the compilers, we compare the performances of the two similar codes (in the same setting as in Sect. 5), which perform a simple vector-valued function ($y = \alpha * x$), with differentiate amount of iterations.

As can be seen in Fig. 9, the *target*, *teams* and *simd* directives creates an overhead when the number of iterations is relatively small, but in all of the other cases they produce increased performances, up to an order of magnitude. Therefore, this small, easy to implement changes, can elevate the performances of the compilers.

Listing 24 OpenMP 2.5 (Listing 23)

```
void initialize_x_and_y(double * x, double * y, int N){{
        int i = 0;
        #pragma loop name initialize_x_and_y#0
        for (; i<N;  ++ i){
            x[i]=(i*0.1);
            y[i]=(i*0.2);
        }}}

int main(){
    double * x, * y;
    double alpha = 2.0;
    int N = (1024*1024)*256;
    int R = 8192;
    int k, i;

    x=((double * )malloc(N*sizeof(double)));
    y=((double * )malloc(N*sizeof(double)));

    initialize_x_and_y(x, y, N);

    #pragma cetus firstprivate(y)
    #pragma cetus private(i, k)
    #pragma cetus lastprivate(y)
    #pragma loop name main#0
    #pragma cetus parallel
    #pragma omp parallel for if((10000<((1L+(3L*R)) + ((3L*N)*R)))) private(i, k) firstprivate(y) lastprivate(y)
    for (k=0; k<R; k ++ ){
        #pragma cetus private(i)
        #pragma loop name main#0#0
        for (i=0; i<N; i ++ ){
            y[i]=(alpha*x[i]);
        }}}
```

Listing 25 OpenMP 4.5 (Listing 23)

```c
void initialize_x_and_y(double *x, double *y, int N){

        #pragma omp target teams distribute parallel for map(from:x[0:N], y[0:N])
        for (int i=0; i<N; ++i) {x[i] = i*0.1; y[i] = i*0.2;}
}

int main() {
    double *x, *y;
    double alpha = 2.0;
    int N = 1024*1024*256;
    int R = 8192;
    int k,i;

    x = (double *) malloc (N * sizeof(double));
    y = (double *) malloc (N * sizeof(double));

    initialize_x_and_y(x,y,N);

    #pragma omp target teams distribute parallel for simd map(to:x[0:N]) map(tofrom:y[0:N])
        for(k=0; k<R; k++)
            for (i=0; i<N; i++)
                y[i] = alpha * x[i];
}
```

7.2.2 Fusion of Advantageous

As mentioned and thoroughly discussed in Sects. 4 and 6, each compiler has its advantages and disadvantages. Each compiler had the best performance in at least one of the tests, but no compiler had better performance than other compilers in all tests. This could indicate that a combination of the compilers' output in some manner might produce better performances, as the compilers vary in their ability to parallel different code structures. In a way, by combining the compilers' output wisely we can exploit their advantages and minimizing their disadvantages. One can push this idea further and include minimal human intervention by removing unnecessary OpenMP directives produced by the combined output.

In order to achieve the above objective it might be beneficial to design and build a new parallelization framework, based on current automatic parallelizers, which will be able to adapt the automated parallelism scheme according to the performances of a collection of representative runs, and the varying hardware in a given cluster - meaning that the parallelization directives will constantly be optimized based on the simulation

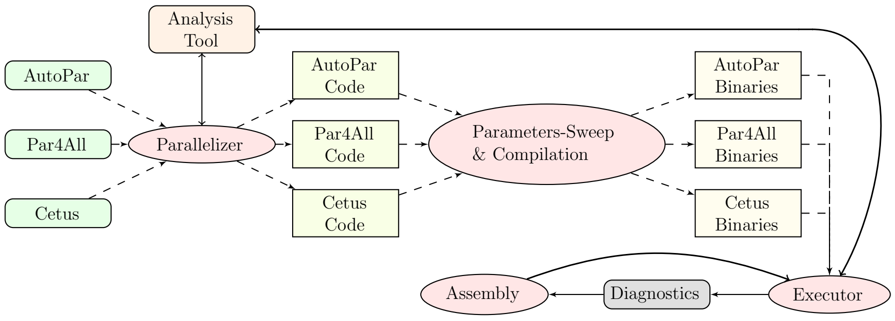


Fig. 10 Scheme of the desired parallelizator

diagnostics and scaling results. For example, the optimization will be done by choosing different scheduling methods, chunk size, thread-affinity, thread-placement, number of threads and so forth. *Thread affinity* allows the user to define the way the OpenMP runtime environment will spread the threads across the machine. This can be useful in cases of high memory/bandwidth, in which the user will want to spread the threads across the machine as much as possible to maximize the throughput alternatively. In cases of low memory usage and many instructions, the user will want to make sure the threads are working on the same cache-line. The three main options for thread spreading are *master*, *close* and *spread*. Once the user define the *spread* policy it is also useful to provide the thread-placement policy that defines on which hardware specifically (on core-level, on socket-level, or on hyper-thread level) the threads will run. The new framework will also differentiate those parallel command styles based on the given cluster architecture. For example, the usage of OpenMP on NUMA architecture vs. the native run, or by offloading with OpenMP 4.5 to accelerators and co-processors. Figure 10 outlines the basic architecture of this new framework, and we can describe it as follows. The software will first apply the appropriate automatic parallelization compilers on a given source code. Then, a runtime analysis tool will insert its queries before and after every parallelized loop. Every feasibly appropriate combination of the parallelized loop, runtime variables, and input sizes will then be executed with varying problem sizes while the individual execution time of each loop will be analyzed by the analysis tool. Any loop with a speedup below a given threshold will be removed. Finally, the framework will produce a new output code that contains the automated parallelism scheme with the best performance. This framework can help significantly to both experienced and novice parallel code programmers.

Acknowledgements This work was supported by the Lynn and William Frankel Center for Computer Science. Computational support was provided by the NegevHPC project [44].

References

1. Geer, D.: Chip makers turn to multicore processors. Computer **38**(5), 11–13 (2005)
2. Blake, G., Dreslinski, R.G., Mudge, T.: A survey of multicore processors. IEEE Signal Process. Mag. **26**(6), 26–37 (2009)

3. Pacheco, P.: An Introduction to Parallel Programming. Elsevier, Amsterdam (2011)
4. Leopold, C.: Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches. Wiley, Hoboken (2001)
5. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
6. Gropp, W., Thakur, R., Lusk, E.: Using MPI-2: Advanced Features of the Message Passing Interface. MIT Press, Cambridge (1999)
7. Snir, M., Otto, S., Huss-Lederman, S., Dongarra, J., Walker, D.: MPI-the Complete Reference: The MPI Core, vol. 1. MIT press, Cambridge (1998)
8. Boku, Taisuke, Sato, Mitsuhisa, Matsubara, Masazumi, Takahashi, Daisuke: Openmpi-openmp like tool for easy programming in mpi. In Sixth European Workshop on OpenMP, pages 83–88, (2004)
9. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. In: ACM SIGGRAPH 2008 Classes, p. 16. ACM (2008)
10. Oren, G., Ganan, Y., Malamud, G.: Automp: an automatic openmp parallization generator for variable-oriented high-performance scientific codes. Int. J. Comb. Optim. Probl. Inform. **9**(1), 46–53 (2018)
11. Neamtiu, I., Foster, J.S., Hicks, M.: Understanding source code evolution using abstract syntax tree matching. ACM SIGSOFT Softw. Eng. Notes **30**(4), 1–5 (2005)
12. AutoPar documentations. http://rosecompiler.org/ROSE_HTML_Reference/auto_par.html. Accessed 8 Aug 2019
13. ROSE homepage. http://rosecompiler.org. Accessed 8 Aug 2019
14. Dever, M.: AutoPar: automating the parallelization of functional programs. PhD thesis, Dublin City University (2015)
15. Par4All homepage. http://par4all.github.io/. Accessed 8 Aug 2019
16. PIPS homepage. https://pips4u.org/. Accessed 8 Aug 2019
17. Ventroux, N., Sassolas, T., Guerre, A., Creusillet, B., Keryell, R.: SESAM/Par4all: a tool for joint exploration of MPSoC architectures and dynamic dataflow code generation. In: Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, pp. 9–16. ACM (2012)
18. Cetus homepage. https://engineering.purdue.edu/Cetus/. Accessed 8 Aug 2019
19. Dave, C., Bae, H., Min, S.-J., Lee, S., Eigenmann, R., Midkiff, S.: Cetus: a source-to-source compiler infrastructure for multicores. Computer **42**(12), 36–42 (2009)
20. Hall, M.W., Anderson, J.M., Amarasinghe, S.P., Murphy, B.R., Liao, S.-W., Bugnion, E., Lam, M.S.: Maximizing multiprocessor performance with the suif compiler. Computer **29**(12), 84–89 (1996)
21. Pottenger, B., Eigenmann, R.: Idiom recognition in the Polaris parallelizing compiler. In: Proceedings of the 9th International Conference on Supercomputing, pp. 444–448. ACM (1995)
22. Tian, X., Bik, A., Girkar, M., Grey, P., Saito, H., Su, E.: Intel® openmp c++/fortran compiler for hyper-threading technology: implementation and performance. Intel Technol. J. **6**(1), 36–46 (2002)
23. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: PLUTO: a practical and fully automatic polyhedral program optimization system. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008). Citeseer (2008)
24. Prema, S., Jehadeesan, R., Panigrahi, B.K.: Identifying pitfalls in automatic parallelization of NAS parallel benchmarks. In: 2017 National Conference on Parallel Computing Technologies (PARCOMPTECH), pp. 1–6. IEEE (2017)
25. Arenaz, M., Hernandez, O., Pleiter, D.: The technological roadmap of parallware and its alignment with the openpower ecosystem. In: International Conference on High Performance Computing, pp. 237–253. Springer (2017)
26. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., et al.: The nas parallel benchmarks. Int. J. Supercomput. Appl. **5**(3), 63–73 (1991)
27. Graham, S.L., Kessler, P.B., McKusick, M.K.: Gprof: a call graph execution profiler. ACM SIGPLAN Not. **39**(4), 49–57 (2004)
28. Prema, S., Jehadeesan, R.: Analysis of parallelization techniques and tools. Int. J. Inf. Comput. Technol. **3**(5), 471–478 (2013)
29. Sohal, M., Kaur, R.: Automatic parallelization: a review. Int. J. Comput. Sci. Mob. Comput. **5**(5), 17–21 (2016)

30. Quinlan, D.: ROSE: compiler support for object-oriented frameworks. Parallel Process. Lett. **10**(02n03), 215–226 (2000)
31. Amini, M., Creusillet, B., Even, S., Keryell, R., Goubier, O., Guelton, S., McMahon, J.O., Pasquier, F.-X., Péan, G., Villalon, P.: Par4all: from convex array regions to heterogeneous computing. In: IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012 (2012)
32. Lee, S.-I., Johnson, T.A., Eigenmann, R.: Cetus—an extensible compiler infrastructure for source-to-source transformation. In: International Workshop on Languages and Compilers for Parallel Computing, pp. 539–553. Springer (2003)
33. Liang, X., Humos, A.A., Pei, T.: Vectorization and parallelization of loops in C/C++ code. In: Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS). The Steering Committee of The World Congress in Computer Science, Computer, pp. 203–206 (2017)
34. Jubb, C.: Loop optimizations in modern c compilers (2014)
35. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High Performance Programming. Newnes, Oxford (2013)
36. Lu, M., Zhang, L., Huynh, H.P., Ong, Z., Liang, Y., He, B., Goh, R.S.M., Huynh, R.: Optimizing the mapreduce framework on Intel Xeon Phi coprocessor. In 2013 IEEE International Conference on Big Data, pp. 125–130. IEEE (2013)
37. Heinecke, A., Vaidyanathan, K., Smelyanskiy, M., Kobotov, A., Dubtsov, R., Henry, G., Shet, A.G., Chrysos, G., Dubey, P.: Design and implementation of the linpack benchmark for single and multi-node systems based on Intel® Xeon Phi coprocessor. In: 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), pp. 126–137. IEEE (2013)
38. Bailey, D.H.: NAS parallel benchmarks. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1254–1259. Springer, Heidelberg (2011)
39. NPB in C homepage. http://aces.snu.ac.kr/software/snu-npb/. Accessed 8 Aug 2019
40. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The scalasca performance toolset architecture. Concurr. Comput. Pract. Exp. **22**(6), 702–719 (2010)
41. Van der Pas, R., Stotzer, E., Terboven, C.: Using OpenMP The Next Step: Affinity, Accelerators, Tasking, and SIMD. MIT Press, Cambridge (2017)
42. Sui, Y., Fan, X.I., Zhou, H., Xue, J.: Loop-oriented array-and field-sensitive pointer analysis for automatic SIMD vectorization. In: ACM SIGPLAN Notices, vol. 51, pp. 41–51. ACM (2016)
43. Zhou, H.: Compiler techniques for improving SIMD parallelism. PhD thesis, University of New South Wales, Sydney, Australia (2016)
44. NegevHPC Project. https://www.negevhpc.com. Accessed 8 Aug 2019

Affiliations

Re'em Harel[1,2] · Idan Mosseri[3,4] · Harel Levin[4,5] · Lee-or Alon[2,6] · Matan Rusanovsky[2,3] · Gal Oren[3,4]

Re'em Harel
reemharel22@gmail.com

Idan Mosseri
idanmos@post.bgu.ac.il

Harel Levin
harellevin@gmail.com

Lee-or Alon
lee-or@mail.com

Matan Rusanovsky
matanru@post.bgu.ac.il

[1] Department of Physics, Bar-Ilan University, 52900 Ramat Gan, Israel

[2] Israel Atomic Energy Commission, P.O.B. 7061, 61070 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, Nuclear Research Center-Negev, P.O.B. 9001, Be'er-Sheva, Israel

[5] Department of Mathematics and Computer Science, The Open University of Israel, P.O.B. 808, Ra'anana, Israel

[6] Department of Computer Science, Bar-Ilan University, 52900 Ramat Gan, Israel