J Supercomput (2018) 74:1461–1472
https://doi.org/10.1007/s11227-017-2231-3

# Language-based vectorization and parallelization using intrinsics, OpenMP, TBB and Cilk Plus

**Przemysław Stpiczyński**[1]

Published online: 11 January 2018


**Abstract** The aim of this paper is to evaluate OpenMP, TBB and Cilk Plus as basic language-based tools for simple and efficient parallelization of recursively defined computational problems and other problems that need both task and data parallelization techniques. We show how to use these models of parallel programming to transform a source code of *Adaptive Simpson's Integration* to programs that can utilize multiple cores of modern processors. Using the example of *Belman–Ford algorithm* for solving single-source shortest path problems, we advise how to improve performance of data parallel algorithms by tuning data structures for better utilization of vector extensions of modern processors. Manual vectorization techniques based on Cilk array notation and intrinsics are presented. We also show how to simplify such optimization using Intel SIMD Data Layout Template containers.

**Keywords** Multicore · Manycore · Recursive algorithms · Shortest path problems · Intrinsics · OpenMP · Cilk Plus · TBB · SDLT containers

## 1 Introduction

Recently, multicore and manycore computer architectures have become very attractive for achieving high-performance execution of scientific applications at relatively low costs [5,13,17]. Modern CPUs and accelerators achieve performance that was recently reached by supercomputers. Unfortunately, the process of adapting existing software to such new architectures can be difficult if we expect to achieve reasonable performance

✉ Przemysław Stpiczyński
przem@hektor.umcs.lublin.pl

[1] Institute of Mathematics, Maria Curie–Skłodowska University, Pl. Marii Curie-Skłodowskiej 1, Lublin 20-031, Poland

without putting much effort into software development. However, sometimes the use of high-level language-based programming interfaces devoted to parallel programming can get satisfactory results with rather little effort [15].

Software development process for modern Intel multicore CPUs and manycore coprocessors such as Xeon Phi [5,13] requires special optimization techniques to obtain codes that would utilize the power of underlying hardware. Usually it is not sufficient to parallelize applications. For such computer architectures, efficient vectorization is crucial for achieving satisfactory performance [5,17]. Unfortunately, very often compiler-based automatic vectorization is not possible because of some non-obvious data dependencies inside loops [1]. Moreover, the performance of vectorized programs can be improved by the use of proper memory layout. On the other hand, people expect parallel programming to be easy. The use of simple and powerful programming constructs that can utilize underlying hardware is highly desired.

Intel C/C++ compilers and development tools offer many language-based extensions that can be used to simplify the process of developing high-performance parallel programs [6,17]. OpenMP [3,18] is the most popular, but one can consider using Threading Building Blocks (TBB for short) [6,12] or Cilk Plus [5,13]. More sophisticated language-based optimization can be done by using *intrinsics*, which allow to utilize Intel Advanced Vector Extensions (i.e., SIMD extensions) explicitly [6]. SDLT template library can be applied to introduce SIMD-friendly memory layout transparently [6].

In this paper, we evaluate OpenMP, Intel TBB and Cilk Plus as language-based tools for simple and efficient parallelization of recursively defined computational problems and other problems that need both task and data parallelization techniques. We also advise how to improve the performance of such algorithms by tuning data structures manually and automatically using SDLT. We also explain how to explicitly utilize vector units of modern multicore and manycore processors using intrinsics. We show how to parallelize recursively defined *Adaptive Simpson's Integration Rule* [7] using OpenMP, Intel TBB and Cilk Plus [9], and we consider various implementations of Belman–Ford algorithm for solving the single-source shortest path problem [4] and examine their performance. These two computational problems have been chosen to demonstrate the most important features of the considered language-based tools.

## 2 Short overview of selected language-based tools

In this section, we present a short overview of the considered language-based tools for parallel and vector programming.

**OpenMP** is a well-known standard for shared memory parallel programming in C/C++ and Fortran [3,11]. It is based on compiler directives used to specify parallel execution of selected parts of computer programs (i.e., loops and code sections). Directives are also used to provide explicit synchronization constructs. Such directives may contain clauses that define data sharing between threads. Additionally, OpenMP consists of a small set of library routines and environment variables that influence runtime behavior. Newer versions of the standard define more advanced programming constructs such as tasking and SIMD support [18].

**TBB (Threading Building Blocks)** is a C++ template library supporting task parallelism on Intel multicore platforms [5,12]. It provides a rich collection of components for parallel programming, and a scheduler which manages and schedules threads to execute parallel tasks. TBB also provides low-level services for memory allocation and atomic operations. TBB programs can be combined with OpenMP pragmas specifying compiler supported vectorization; thus, it can exploit both task and data parallelism.

**Cilk Plus** offers several powerful extensions to C/C++ that allow to express both task and data parallelism [5,8,13,14]. The most important constructs are useful to specify and handle possible parallel execution of tasks. `_Cilk_for` followed by the body of a `for` loop tells that iterations of the loop can be executed in parallel. Runtime applies the *divide-and-conquer* approach to schedule tasks among active workers to ensure balanced workload of available cores. `_Cilk_spawn` permits a given function to be executed asynchronously with the rest of the calling function. `_Cilk_sync` tells that all tasks spawned in a function must complete before execution continues. Another important feature of Cilk Plus is the array notation which introduces vectorized operations on arrays. Expression `A[start:len:stride]` represents an array section of length `len` starting from `A[start]` with the given `stride`. Omitted `stride` means 1. The operator `[:]` can be used on both static and dynamic arrays. There are also several built-in functions to perform basic computations among elements in an array such as sum, min and max. It should be noticed that the array notation can also be used for array indices. For example, `A[x[0:len]]` denotes elements of the array `A` given by indices from `x[0:len]`. Intel Cilk Plus also supports *Shared Virtual Memory* which allows to share data between the CPU and the coprocessor. It is perfect for exchanging irregular data with limited size, when explicit synchronization is not used frequently [16].

**Intrinsics** for SIMD instructions allow to take full advantage of Intel Advanced Vector Extensions (AVX, AVX2, AVX-512) what cannot always be easily achieved due to limitations of programming languages and compilers [6]. They allow programmers to write constructs that look like C/C++ function calls corresponding to actual SIMD instructions. Such calls are replaced with assembly code inlined directly into programs. The disadvantage of this solution is the lack of the code portability between different versions of vector extensions.

**SDLT (SIMD Data Layout Template)** is a C++11 template library which provides containers with SIMD-friendly data layouts [6]. The use of such containers allows for a transparent transition of data structures of the *Array of Structures* (AOS) type to *Structure of Arrays* (SOA) or *Arrays of Structure of Arrays* (ASA) forms. Such conversions can improve vectorization and increase the efficiency of programs executed on modern processors.

## 3 Two examples of computational problems

Now we will present two exemplary problems which can be easily parallelized and optimized using considered language-based tools. All implementations have been tested on a server with two Intel Xeon E5-2670 version 3 (totally 24 cores with

**Listing 1** Cilk version of Adaptive Simpson's method

```
1  double cilkASAux(double (*f)(double),double a,double b,
2         double eps,double S,double fa,double fb,double fc,int
               depth)
3  {
4    double c = (a + b)/2, h = b - a;
5    double d = (a + c)/2, e = (c + b)/2;
6    double fd = f(d), fe = f(e);
7    double Sleft = (h/12)*(fa + 4*fd + fc);
8    double Sright = (h/12)*(fc + 4*fe + fb);
9    double S2 = Sleft + Sright;
10   if (depth <= 0 || fabs(S2 - S) <= 15*eps)
11     return S2 + (S2 - S)/15;
12   double din1 =
13     _Cilk_spawn cilkASAux(f,a,c,eps/2,Sleft,fa,fc,fd,depth-1);
14   double din2 = cilkASAux(f,c,b,eps/2,Sright,fc,fb,fe,depth-1);
15   _Cilk_sync;
16   return  din1+din2;
17 }
18 double cilkAS(double (*f)(double),double a,double b,double eps,
       int depth)
19 { double c = (a + b)/2, h = b - a;
20   double fa = f(a), fb = f(b), fc = f(c);
21   double S = (h/6)*(fa + 4*fc + fb);
22   return cilkASAux(f,a,b,eps,S,fa,fb,fc,depth);
23 }
```

hyperthreading, 2.3 GHz), 128GB RAM, with Intel Xeon Phi Coprocessor 7120P (61 cores with multithreading, 1.238 GHz, 16GB RAM), running under CentOS 6.5 with Intel Parallel Studio version 2017, C/C++ compiler supporting Cilk Plus, TBB and SDLT. Experiments on Xeon Phi have been carried out using its native mode.

### 3.1 Adaptive Simpson's Integration Rule

Let us consider the following recursive method for numerical integration called *Adaptive Simpson's Rule* [7]. We want to find the approximation of $I(f) = \int_a^b f(x)\mathrm{d}x$ with a user-specified tolerance $\epsilon$. Let $S(a,b) = \frac{h}{6}(f(a) + 4f(c) + f(b))$, where $h = b - a$ and $c$ is a midpoint of the interval $[a,b]$. The method uses Simpson's rule to the halves of the interval in recursive manner until the stopping criterion $\frac{1}{15}|S(a,c) + S(c,b) - S(a,b)| < \epsilon$. is reached [10].

Listing 1 shows our Cilk version of the straightforward recursive implementation of the method [2]. Note that we have only included keywords `_Cilk_spawn` and `_Cilk_sync`. The first one specifies that `cilkASAux()` can execute in parallel with the remainder of the calling kernel. `_Cilk_sync` tells that all spawned calls in the current call of the kernel must complete before execution continues. Our OpenMP implementation (Listing 2) assumes the use of tasks [18], where the keywords `_Cilk_spawn` and `_Cilk_sync` are simply replaced with `task` and `taskwait` constructs. The main function `ompAS()` is a little bit more complicated. One should create a parallel region and call `ompASAux()` as a single task. Listing 3 presents our TBB version of the method, which is analogous to the Cilk version but calls `tbbASAux()`.

**Listing 2** OpenMP version of Adaptive Simpson's method

```
1  double ompASAux(double (*f)(double),double a,double b,
2         double eps,double S,double fa,double fb,double fc,int
               depth)
3  { // as in Cilk version
4    // ...
5    double din1,din2;
6    #pragma omp task shared(din1)
7    { din1 = ompASAuxAux(f,a,c,epsilon/2,Sleft,fa,fc,fd,depth-1);
         }
8    din2 = ompASAuxAux(f,c,b,epsilon/2,Sright,fc,fb,fe,depth-1);
9    #pragma omp taskwait
10   return  din1 + din2;
11 }
12 double ompAS(double (*f)(double),double a,double b,double eps,
       int depth)
13 { // as in Cilk version
14   // ...
15   double y;
16   #pragma omp parallel
17   {#pragma omp single
18      y=ompASAux(f,a,b,epsilon,S,fa,fb,fc,maxRecursionDepth);
19   }
20   return y;
21 }
```

**Listing 3** TBB version of Adaptive Simpson's method

```
1  double tbbASAux(double (*f)(double),double a,double b,
2         double eps,double S,double fa,double fb,double fc,int
               depth)
3  { // as in Cilk version
4    // ...
5    double din1,din2;
6    tbb::task_group g;
7    g.run([&]{din1=tbbASAux(f,a,c,epsilon/2,Sleft,fa,fc,fd,depth
         -1);});
8    din2 = tbbASAux(f,c,b,epsilon/2,Sright,fc,fb,fe,depth-1);
9    g.wait();
10   return  din1+din2;
11 }
```

Table 1 shows the execution time of our three parallel implementations applied for finding the approximation of $\int_{-4.4}^{4.4} \exp(x^2)\mathrm{d}x$ with $\epsilon = 1.0e-7$ and $depth = 40$. We can observe that `cilkAS()` outperforms `ompAS()` significantly (about four times faster for CPU and three times for Xeon Phi). `cilkAS()` is also about $2.0\times$ faster than `tbbAS()`. It should be noticed that the execution time (s) of the sequential version of the method is 62.8 for CPU and 638.04 for Xeon Phi. Thus, the speedup achieved by our Cilk implementation is 14.35 (CPU) and 70.66 (Xeon Phi), respectively. The Cilk version scales very well when the number of Cilk workers increases up to 24 for CPU and 60 for Xeon Phi, respectively, i.e., to the number of physical cores. The further increase in the number of workers results in smaller and rather marginal gains.

**Table 1** Execution time (s) of `cilkAS()`, `ompAS()` and `tbbAS()` for $\int_{-4.4}^{4.4} \exp(x^2)\mathrm{d}x$

| 2× E5-2670 | | | | | | |
|---|---|---|---|---|---|---|
| Number of threads/workers | 2 | 4 | 6 | 12 | 24 | 48 |
| `cilkAS()` | 61.99 | 31.06 | 20.64 | 10.57 | 5.39 | 4.32 |
| `ompAS()` | 202.71 | 101.43 | 68.03 | 34.36 | 17.43 | 15.45 |
| `tbbAS()` | 141.98 | 71.64 | 48.29 | 25.46 | 12.21 | 9.37 |
| Xeon Phi 7120P (native mode) | | | | | | |
| Number of threads/workers | 2 | 30 | 60 | 120 | 180 | 240 |
| `cilkAS()` | 478.11 | 32.44 | 16.71 | 10.51 | 9.33 | 9.03 |
| `ompAS()` | 1355.67 | 92.60 | 45.57 | 31.13 | 29.22 | 28.52 |
| `tbbAS()` | 839.71 | 55.32 | 27.74 | 18.94 | 18.26 | 17.46 |

### 3.2 Bellman–Ford algorithm for the single-source shortest path problem

Let $G = (V, E)$ be a directed graph with $n$ vertices labeled from 0 to $n - 1$ and $m$ arcs $\langle u, v\rangle \in E$, where $u, v \in V$. Each arc has its weight $w(u, v) \in \mathbf{R}$ and we assume $w(u, v) = \infty$ when $\langle u, v\rangle \notin E$. For each path $\langle v_0, v_1, \ldots, v_p\rangle$, we define its length as $\sum_{i=1}^{p} w(v_{i-1}, v_i)$. We also assume that $G$ does not contain negative cycles. Let $d(s, t)$ denote the length of the shortest path from $s$ to $t$ or $d(s, t) = \infty$ if there are no paths from $s$ to $t$. Algorithm 1 is the well-known *Belman–Ford* method for finding shortest lengths of paths from a given source $s \in V$ to all other vertices [4].

**Algorithm 1:** Bellman–Ford Algorithm

**Data**: $G = (V, E)$, $|V| = n$, $s \in V$, $w(u, v)$ for all $u, v \in V$
**Result**: $\mathrm{D}[v] = d(s, v)$ for all $v \in V$

```
1  for v ∈ V do
2  |  D[v] ← w(s, v);
3  end
4  D[s] ← 0;
5  for k = 1, ..., n − 2 do
6  |  for v ∈ V \ {s} do
7  |  |  for u ∈ V such that ⟨u, v⟩ ∈ E do
8  |  |  |  D[v] ← min (D[v], D[u] + w(u, v));
9  |  |  end
10 |  end
11 end
```

The most common basic implementations of the algorithm assume AOS (i.e., *Array of Structures*) representations of graphs. It means that a graph is represented as an array that describes its vertices. Each vertex is described by an array containing information about incoming arcs. Each arc is represented by the initial vertex and arc's weight. It is also necessary to store the length of arrays describing vertices. In order to parallelize such a basic implementation using OpenMP (see Listing 4), we should notice that

**Listing 4** OpenMP implementation of Algorithm 1: AOS, loop

```
1  void ompBF1(DGraph & g,float *d1,float *d2)
2  {float dist;
3   #pragma omp parallel
4   {#pragma omp for schedule(runtime)
5    for(int i=1;i<g.n;i++)
6     {dist=INFTY;
7      if((g.node[i].degIn>0)&&(g.node[i].in[0].v==0))
8         dist=g.node[i].in[0].weight;
9      d1[i]=dist;
10    }
11   #pragma omp single
12   { d2[0]=d1[0]=0; }
13   for(int k=1;k<g.n-1;k++)
14    {#pragma omp for schedule(runtime)
15     for(int i=1;i<g.n;i++)
16       {float t=d1[i];
17        #pragma omp simd reduction(min:t)
18        for(int j=0;j<g.node[i].degIn;j++)
19          t=std::min(t,d1[g.node[i].in[j].v]+g.node[i].in[j].
                weight);
20        d2[i]=t;
21        }
22      float *temp=d1; d1=d2; d2=temp;
23     }
24 }
```

the entire algorithm should be within the `parallel` construct. The loops 5–10 and 15–23 can be parallelized using `for` constructs. The assignment in line 12 needs to be a single task (i.e., defined by the `single` construct). Moreover, we need two copies of the array `D` for storing current and previous updates within each iteration of the loop 13–23. The most inner loop 18–19 is automatically vectorized by compiler. For the sake of simplicity, we also assume that the vertex labeled as 0 is the source.

In order to introduce SIMD-friendly memory layout for better utilization of Intel Advanced Vector Extensions, we assume that each vertex of a given graph is represented by two arrays of the same size. The first one (i.e., `inv`) sorted in increasing order contains labels of initial vertices of incoming arcs. The next one (i.e., `inw`) stores weights of corresponding arcs. Thus, such a representation of graphs is of the SOA (i.e., *Structure of Arrays*) type. These arrays should be allocated using `_mm_malloc()` to ensure proper memory alignment [17]. Listing 5 shows the most inner loop of our another implementation of the algorithm. Note that the loop can be replaced with the corresponding array expression (Listing 6). Listing 7 presents another possible modification. The most inner loop is replaced with a simple call to the function `vecmin()`, which uses AVX2 intrinsics explicitly. Note that the first loop works on 8-element vectors, while the second one processes the remainder of input data sequentially.

Listing 8 presents the most interesting improvements. It uses SDLT template library to move from AOS to SOA transparently. Each arc is represented as in our first implementation. By the use of `SDLT_PRIMITIVE`, the data structure `Arc` is declared as a primitive and its data members are identified. Then we can define a data structure for arcs. It should be an array of containers of the type `sdlt::soa1d_container<Arc>`. Internally, such containers are represented as

**Listing 5** The most inner loop of Algorithm 1: SOA, loop

```
float t=d1[i];
#pragma omp simd reduction(min:t)
for(int j=0;j<g.node[i].degIn;j++)
  t=std::min(t,d1[g.node[i].inv[j]]+g.node[i].inw[j]);
d2[i]=t;
```

**Listing 6** The most inner loop of Algorithm 1: SOA, array notation

```
int deg=g.node[i].degIn;
d2[i]=std::min(d1[i],__sec_reduce_min(d1[g.node[i].inv[0:deg]]+
                                      g.node[i].inw[0:deg]))
                                           ;
```

**Listing 7** The most inner loop of Algorithm 1: SOA, intrinsics for AVX2

```
float vecmin(int n,float *d,int *inv,float *inw)
{ float minv=0; int i,j=0; float *pd=d;
  float *pinw=inw; int *pinv=inv; float t[8];
__m256 xmin = _mm256_set1_ps(1.0e+20);
t[0]=1.0e+20;
for(i=0;i<(n/8)*8;i+=8)
{ __m256 xw,x1,x2;  __m256i idx;
  idx=_mm256_load_si256((__m256i*)pinv);
  x1=_mm256_i32gather_ps(pd,idx,4);
  xw=_mm256_load_ps(pinw);
  x1=_mm256_add_ps(xw,x1);
  x2=_mm256_permute_ps(x1,0x39);
  x1=_mm256_min_ps(x1,x2);
  x2=_mm256_permute_ps(x1,0x4E);
  x1=_mm256_min_ps(x1,x2);
  x2=_mm256_permute2f128_ps(x1,x1,0x1);
  x1=_mm256_min_ps(x1,x2);
  xmin=_mm256_min_ps(xmin,x1);
  pinw+=8; pinv+=8;
}
_mm256_store_ps(t,xmin); minv=t[0];
for(i=(n/8)*8;i<n;i++)
   minv=std::min(minv,d[inv[i]]+inw[i]);
return min v;
}
```

SOA with elements aligned properly to improve efficiency of AVX instructions. Vectorization of the loop is enforced by the use of "`#pragma omp simd reduction`." To ensure the compiler respects that the methods `x()` and `weight()` are inlined, we use "`#pragma force inline`."

Now let us consider the results of experiments performed to compare five considered implementations of Belman–Ford algorithm: **BF1** (basic using AOS), **BF2** (using SOA with the most inner loop vectorized by compiler), **BF3** (using SOA and the array notation), **BF4** (using SOA and intrinsics) and **BF5** (using SDLT). All results have been obtained for graphs generated randomly for a given number of vertices and a maximum degree (i.e., the maximum number of incoming arcs). Fig-

**Listing 8** The most inner loop of Algorithm 1: SDLT

```
struct Arc // data structures
{ unsigned int v;
  float weight;
};
SDLT_PRIMITIVE(Arc,v,weight);
struct DGraph
{ int n;
  sdlt::soa1d_container<Arc> *node;
};
// ...
// the most inner loop of the algorithm
float t=d1[i];
auto arc = g.node[i].const_access(); // get access to the
    container
#pragma forceinline recursive
{ #pragma omp simd reduction(min:t)
  for(int j=0;j<g.node[i].get_size_d1();j++)
     t=std::min(t,d1[arc[j].v()]+arc[j].weight());
  d2[i]=t;
}
```

ure 1 shows the speedup of five parallel implementations with outer loops parallelized using `schedule(static)` against the sequential version of **BF1**. We can observe that the parallel implementations are much faster than the basic sequential implementation. For sufficiently large graphs, all parallel implementations utilize multiple cores achieving reasonable speedup. It happens when vectorized loops are sufficiently long. Indeed, the speedup grows when the maximum degree (i.e., the length of the arrays grows). Usually **BF5** is much faster than other parallel versions. **BF2**, **BF3** and **BF4** outperform **BF1** for larger and wider graphs, and their performance is comparable; however, **BF4** is slightly faster on Xeon Phi. Unexpectedly, for Xeon E5-2670, the performance of **BF5** drops for larger graphs. Such a behavior has not been observed for Xeon Phi. For this platform, the implementation using SDLT always achieves the best performance.

Figure 2 presents the execution time of the SDLT version of the algorithm parallelized using OpenMP (for "`static`" and "`dynamic,ChS`" values of the clause `schedule`, respectively), TBB and Cilk Plus. In case of our OpenMP implementations, Fig. 2 shows the best results chosen after several tests for various values of `ChS`. Thus, our OpenMP "dynamic" version has been manually tuned. We have observed that the best performance is achieved when the value of `ChS` is about 40 for Xeon E5-2670 and 20 for Xeon Phi. In case of TBB and Cilk, the runtime system has been responsible for load balancing. The parallel loops in the Cilk version have been parallelized using `_Cilk_for` construct. In case of TBB, we have used `tbb::parallel_for` template. It should be noticed that TBB requires more changes in the source code than in the case of OpenMP and Cilk Plus. The use of Cilk seems to be the easiest.

We can observe that for almost all cases, the OpenMP "static" version achieves the best performance. In case of Xeon Phi, the situation is slightly different. For smaller graphs the OpenMP "dynamic" version is really better. Both OpenMP versions outperform TBB and Cilk Plus significantly. Usually, the Cilk version achieves the worst performance. Generally, the use of technologies utilizing dynamic allocation

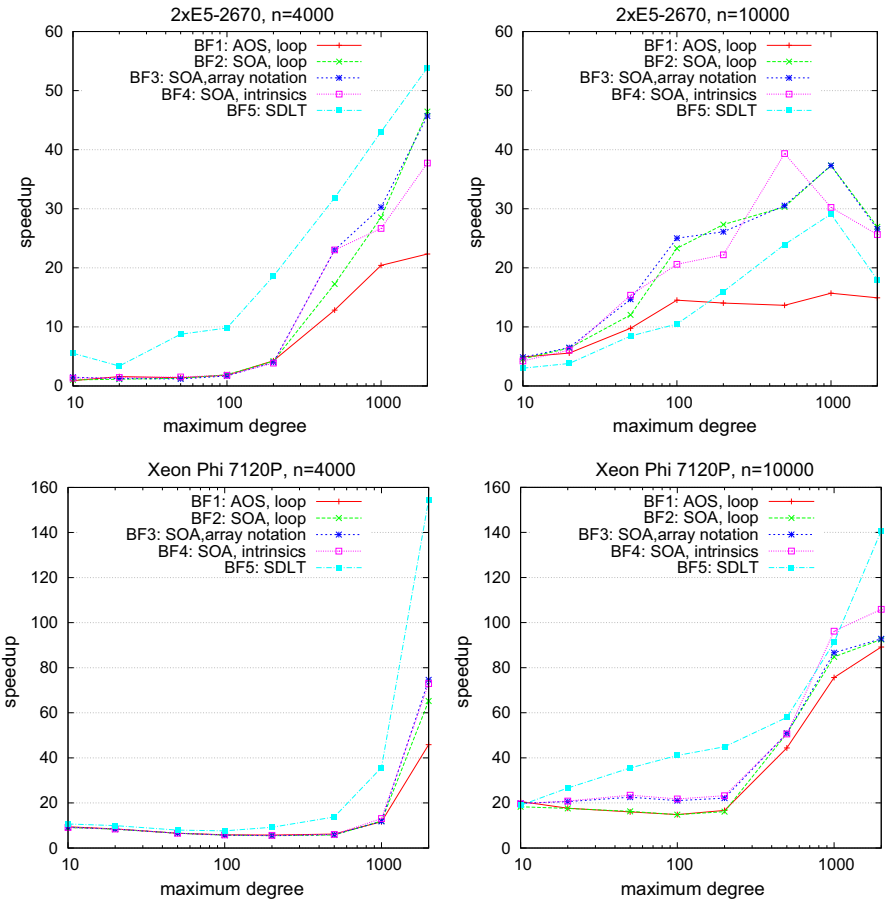


**Fig. 1** Speedup of the considered implementations against the sequential version of **BF1** (static schedule of for loops)

of computational tasks is associated with significant overheads. However, for large graphs both OpenMP and TBB versions achieve almost the same performance.

## 4 Conclusions

We have compared OpenMP, TBB and Cilk Plus as basic language-based tools for simple and efficient parallelization of computational problems. We have shown that Cilk Plus can be very easily applied to parallelize recursively defined *Adaptive Simpson's Integration Rule* and such implementation can also utilize coprocessors such as Intel Xeon Phi. Unexpectedly, the OpenMP implementation using tasks achieves much worse performance. The efficiency of the TBB implementation is also worse than Cilk, but TBB is still better than OpenMP.

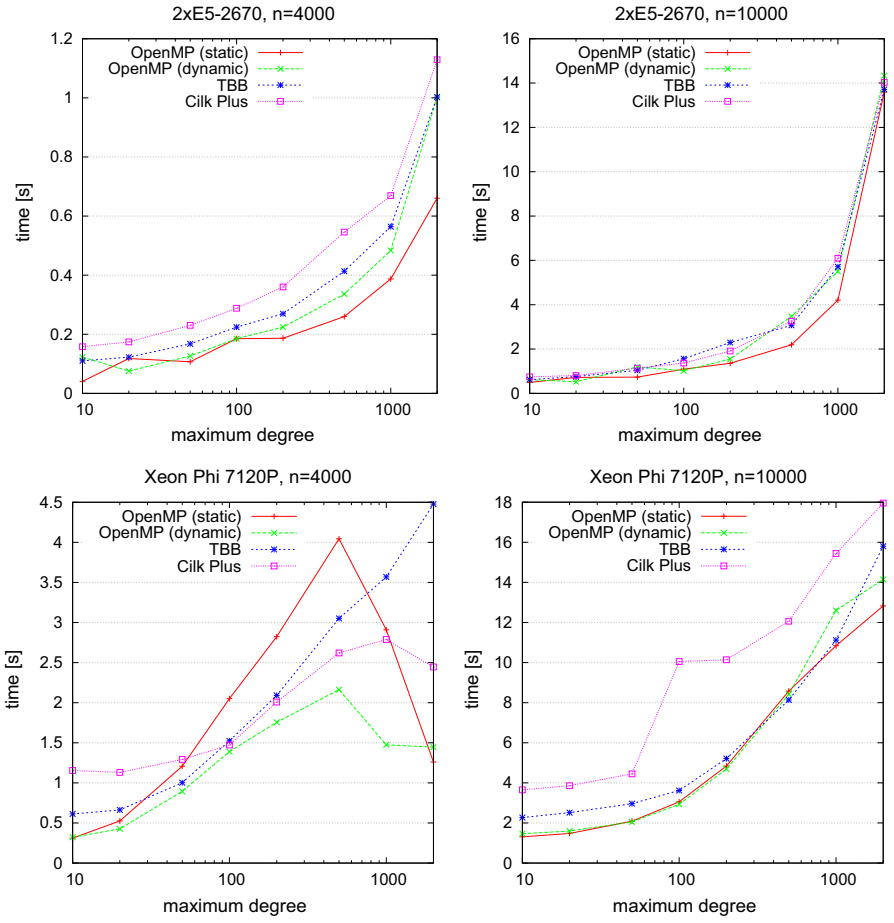


**Fig. 2** Execution time of the SDLT version of the algorithm parallelized using OpenMP (static and dynamic), TBB and Cilk Plus

Using the example of *Belman–Ford algorithm* for solving single-source shortest path problems, we have shown that OpenMP is the best language-based tool for efficient parallelization of simple loops. We have also demonstrated that in case of computational problems that need both task and data parallelization techniques, efficient vectorization is crucial for achieving reasonable performance. Vector extensions of modern multicore processors can be efficiently utilized by using Cilk array notation, intrinsics or even automaticly by compilers, but on condition that data structures are properly aligned in memory. In general, SOA data structures are much more SIMD-friendly than data structures of the AOS form. The transition from AOS to SOA can be made transparently by using SDLT template library. Usually, such semiautomatic optimization results in better performance than rather complicated manual process. In particular, programming using intrinsics involves a much greater effort and results are not so impressive.

**Acknowledgements** The use of computer resources installed at Institute of Mathematics, Maria Curie-Skłodowska University, Lublin, is kindly acknowledged.



## References

1. Allen R, Kennedy K (2001) Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann, Burlington
2. Cameron M (2010) Adaptive integration. http://www2.math.umd.edu/~mariakc/teaching/adaptive.pdf
3. Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R (2001) Parallel programming in OpenMP. Morgan Kaufmann Publishers, San Francisco
4. Cormen T, Leiserson C, Rivest R (1994) Introduction to algorithms. MIT Press, Cambridge
5. Jeffers J, Reinders J (2013) Intel Xeon Phi coprocessor high-performance programming. Morgan Kaufman, Waltham
6. Jeffers J, Reinders J, Sodani A (2016) Intel Xeon Phi processor high-performance programming. Knights landing edition. Morgan Kaufman, Cambridge
7. Kuncir GF (1962) Algorithm 103: Simpson's rule integrator. Commun ACM 5(6):347. https://doi.org/10.1145/367766.368179
8. Leiserson CE (2011) Cilk. In: Padua DA (ed) Encyclopedia of parallel computing. Springer, Berlin, pp 273–288. https://doi.org/10.1007/978-0-387-09766-4_2339
9. Leist A, Gilman A (2014) A comparative analysis of parallel programming models for C++. In: Proceedings of ICCGI 2014: The Ninth International Multi-Conference on Computing in the Global Information Technology, IARIA, pp 121–127
10. Lyness JN (1969) Notes on the adaptive Simpson quadrature routine. J ACM 16(3):483–495. https://doi.org/10.1145/321526.321537
11. Marowka A (2007) Parallel computing on any desktop. Commun ACM 50(9):74–78. https://doi.org/10.1145/1284621.1284622
12. Marowka A (2012) Tbbench: a micro-benchmark suite for intel threading building blocks. J Inf Process Syst 8(2):331–346. https://doi.org/10.3745/JIPS.2012.8.2.331
13. Rahman R (2013) Intel Xeon Phi coprocessor architecture and tools: the guide for application developers. Apress, Berkely
14. Robison AD (2013) Composable parallel patterns with intel cilk plus. Comput Sci Eng 15(2):66–71. https://doi.org/10.1109/MCSE.2013.21
15. Stpiczyński P (2016) Semiautomatic acceleration of sparse matrix-vector product using OpenACC. In: Parallel Processing and Applied Mathematics, 11th International Conference, PPAM 2015, Kracow, Poland, September 6–9, 2015, Revised Selected Papers, Part II, Springer, Lecture Notes in Computer Science, vol 9574, pp 143–152. http://dx.doi.org/10.1007/978-3-319-32152-3_14
16. Stpiczyński P (2018) Efficient language-based parallelization of computational problems using cilk plus. In: Parallel Processing and Applied Mathematics, 12th International Conference, PPAM 2017, Lublin, Poland, September 10–13, 2017 (accepted)
17. Supalov A, Semin A, Klemm M, Dahnken C (2014) Optimizing HPC applications with intel cluster tools. Apress, Berkely
18. van der Pas R, Stotzer E, Terboven C (2017) Using OpenMP—the next step. Affinity, accelerators, tasking, and SIMD. MIT Press, Cambridge