Efficient Language-Based Parallelization of Computational Problems Using Cilk Plus

Przemysław Stpiczyński(✉)

Institute of Mathematics, Maria Curie–Skłodowska University,
Pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin, Poland
przem@hektor.umcs.lublin.pl

Abstract. The aim of this paper is to evaluate Cilk Plus as a language-based tool for simple and efficient parallelization of recursively defined computational problems and other problems that need both task and data parallelization techniques. We show that existing source codes can be easily transformed to programs that can utilize multiple cores and additionally offload some computations to coprocessors like Intel Xeon Phi. We also advise how to improve simplicity and performance of data parallel algorithms by tuning data structures to utilize vector extensions of modern processors. Numerical experiments show that in most cases our Cilk Plus versions of *Adaptive Simpson's Integration* and *Belman-Ford Algorithm* for solving single-source shortest-path problems achieve better performance than corresponding OpenMP programs.

Keywords: Cilk Plus · Multicore · Xeon Phi · Vectorization
Offload · Recursive algorithms · Shortest-path problems

1 Introduction

Recently, multicore and manycore computer architectures have become very attractive for achieving high performance execution of scientific applications at relatively low costs [8,17,20]. 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 without putting much effort into software development. For example, the use of OpenCL [10] leads to a substantial increase of software complexity. However, sometimes the use of high-level language-based programming interfaces devoted to parallel programming can get satisfactory results with rather little effort [19].

Software development process for modern Intel multicore CPUs and manycore coprocessors like Xeon Phi [8,17] requires special optimization techniques to obtain codes that would utilize the power of underlying hardware. Usually it is not sufficient to parallelize applications because in case of such computer architectures efficient vectorization is crucial for achieving satisfactory performance

R. Wyrzykowski et al. (Eds.): PPAM 2017, LNCS 10778, pp. 254–263, 2018.
https://doi.org/10.1007/978-3-319-78054-2_24

[8,20]. Unfortunately, very often compiler-based automatic vectorization is not possible because of some non-obvious data dependencies inside loops [1,21]. On the other hand, people expect parallel programming to be easy and they prefer to concentrate on algorithms and use simple and powerful programming constructs that can utilize underlying hardware.

Cilk Plus introduces new extensions to C/C++ programming languages to express task and data parallelism using high-level constructs [8,9,12,18]. Although Cilk Plus has more usability than OpenMP [6], it is not very popular (several interesting applications can be found in [2,3,13,15]).

In this paper we show that Cilk Plus can be very easily applied to parallelize recursively defined adaptive Simpson's integration rule [11] and such implementation can be easily transformed to utilize coprocessors like Intel Xeon Phi. We also advise how to simplify move from OpenMP to Cilk Plus and improve the performance of such algorithms by tuning data structures to utilize hardware (i.e. vector units) of modern multicore and manycore processors. As an example we consider our Cilk Plus implementation of Belman-Ford algorithm for solving the single-source shortest-path problem [7] which achieves better performance than the corresponding simple OpenMP version of the algorithm. These two computational problems have been chosen to demonstrate the most important features of Cilk Plus that can be easily added to sequential C/C++ programs.

2 Short Overview of Cilk Plus

Cilk Plus offers several powerful extensions to C/C++ that allow to express both task and data parallelism [8,17]. 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, max etc. 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 what is promising especially

for complex data structures [8,17]. Such shared variables are declared using `_Cilk_shared` keyword. It also allows to declare functions that should be available for CPU and coprocessors. Computations can be offloaded to coprocessors for asynchronous execution using `_Cilk_spawn _Cilk_offload` construct. In such a case all necessary data are moved to the coprocessor. Memory synchronization between the CPU and the coprocessor takes place when an offloaded function is called by CPU or an offloaded function returns (i.e. when `cilk_sync` is used). The description of other features of Cilk Plus (like reducers) can be found in [17].

3 Two Examples of Computational Problems

Now we will present two exemplary problems which can be easily parallelized and optimized using Cilk Plus. All implementations have been tested on a server with two Intel Xeon E5-2670 v3 (totally 24 cores with hyperthreading, 2.3 GHz), 128 GB RAM, with Intel Xeon Phi Coprocessor 7120P (61 cores with multithreading, 1.238 GHz, 16 GB RAM), running under CentOS 6.5 with Intel Parallel Studio ver. 2017, C/C++ compiler supporting Cilk Plus. Experiments on Xeon Phi have been carried out using its native and offload modes.

3.1 Adaptive Simpson's Integration Rule

Let us consider the following recursive method for numerical integration called *Adaptive Simpson's Rule* [11]. We want to find the approximation of

$$I(f) = \int_a^b f(x)dx \tag{1}$$

with a user-specified tolerance ϵ. Let $S(a,b) = \frac{h}{6}\left(f(a) + 4f(c) + f(b)\right)$, 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 following stopping criterion is reached [14]:

$$\frac{1}{15}|S(a,c) + S(c,b) - S(a,b)| < \epsilon. \tag{2}$$

Figure 1 shows our parallel version of the straightforward recursive implementation of the method [4]. Note that we have only included keywords `_Cilk_spawn` and `_Cilk_sync`. The first one specifies that `cilkAdaptiveSimpsonsAux()` 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. For comparative purposes we have also implemented the method using OpenMP tasks [16], where the keywords `_Cilk_spawn` and `_Cilk_sync` are simply replaced with `task` and `taskwait` constructs.

Another Cilk implementation of the method assumes that some computations can be offloaded to a coprocessor (i.e. Xeon Phi, if available). The auxiliary kernel `cilkAdaptiveSimpsonsAux()` should be declared with the keyword

```
double cilkAdaptiveSimpsonsAux(double (*f)(double),double a,double b,
                 double eps,double S,double fa,double fb,double fc,int depth)
{
  double c = (a + b)/2, h = b - a;
  double d = (a + c)/2, e = (c + b)/2;
  double fd = f(d), fe = f(e);
  double Sleft = (h/12)*(fa + 4*fd + fc);
  double Sright = (h/12)*(fc + 4*fe + fb);
  double S2 = Sleft + Sright;
  if (depth <= 0 || fabs(S2 - S) <= 15*eps)
    return S2 + (S2 - S)/15;
  double din1 =
    _Cilk_spawn adaptiveSimpsonsAux(f,a,c,eps/2,Sleft,fa,fc,fd,depth-1);
  double din2 = adaptiveSimpsonsAux(f,c,b,eps/2,Sright,fc,fb,fe,depth-1);
  _Cilk_sync;
  return  din1+din2;
}
double cilkAdaptiveSimpsons(double (*f)(double),double a,double b,double eps,
                                                                  int depth)
{ double c = (a + b)/2, h = b - a;
  double fa = f(a), fb = f(b), fc = f(c);
  double S = (h/6)*(fa + 4*fc + fb);
  return adaptiveSimpsonsAux(f,a,b,eps,S,fa,fb,fc,depth);
}
```

Fig. 1. Parallel version of Adaptive Simpson's method

`_Cilk_shared`, what makes the function available for CPU and coprocessors. In the main kernel `cilkAdaptiveSimpsonsOff()`, the integration over the first half of the interval $[a, b]$ can offloaded to Xeon Phi using `_Cilk_spawn _Cilk_offload` construct, while the rest is to be done by CPU.

Table 1 shows the execution time of our three parallel implementations applied for finding the approximation of $\int_{-4.4}^{4.4} \exp(x^2)dx$ with $\epsilon = 1.0e-7$ and $depth = 40$ (namely OpenMP with tasks, Cilk, and Cilk with offload). We can observe that `cilkAdaptiveSimpsons()` outperforms `ompAdaptiveSimpsons()` significantly (about four times faster for CPU and three times for Xeon Phi). It should be noticed that the execution time (seconds) 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.

Our non-offloaded 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. For `cilkAdaptiveSimpsonsOff()`, we can observe that the shortest execution time is achieved for twelve workers. Then the execution time of `cilkAdaptiveSimpsonsAux()` on CPU and Xeon Phi working on the halves of the interval is approximately the same.

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 $w(u, v) \in \mathbf{R}$

Table 1. Execution time (s) of `ompAdaptiveSimpsons()`, `cilkAdaptiveSimpsons()` and `cilkAdaptiveSimpsonsOff()` for $\int_{-4.4}^{4.4} \exp(x^2)dx$

2x E5-2670 and Xeon Phi 7120P (coprocessor)						
number of threads/workers (CPU)	2	4	6	12	24	48
`ompAdaptiveSimpsons()`	202.71	101.43	68.03	34.36	17.43	15.45
`cilkAdaptiveSimpsons()`	61.99	31.06	20.64	10.57	5.39	4.32
`cilkAdaptiveSimpsonsOff()`	34.28	17.06	11.33	5.67	5.78	5.95
Xeon Phi 7120P (native mode)						
number of threads/workers	2	30	60	120	180	240
`ompAdaptiveSimpsons()`	1355.67	92.60	45.57	31.13	29.22	28.52
`cilkAdaptiveSimpsons()`	478.11	32.44	16.71	10.51	9.33	9.03

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 $d(s,t)$ denotes 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 [7].

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 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 Fig. 2, left), we should notice that the entire algorithm should be within the `parallel` construct. Then the loops 7–13 and 18–26

can be parallelized using `for` construct with clause `schedule(dynamic,ChS)`. Thus, iterations are divided into pieces having a size specified by chunk size `ChS` and such pieces are dynamically dispatched to threads. The assignment in line 4 needs to be a single task (i.e. defined by `single`). Moreover, we need two copies of the array `D` for storing current and previous updates within each iteration of the loop 20–25. It should be noticed that this loop is automatically vectorized by the compiler. For the sake of simplicity, we also assume that the vertex labeled as 0 is the source.

```
1  void
2  ompBF1(DGraph & g,float *d1,float *d2)
3  {
4   float dist;
5   #pragma omp parallel
6   {#pragma omp for schedule(dynamic,ChS)
7    for(int i=1;i<g.n;i++)
8     {dist=INFTY;
9      if((g.node[i].degIn>0)&&
10                (g.node[i].in[0].v==0))
11         dist=g.node[i].in[0].weight;
12     d1[i]=dist;
13    }
14    #pragma omp single
15    { d1[0]=0; }
16    for(int k=1;k<g.n-1;k++)
17     {#pragma omp for schedule(dynamic,ChS)
18      for(int i=1;i<g.n;i++)
19       {d2[i]=d1[i];
20        for(int j=0;j<g.node[i].degIn;j++)
21        {
22         int u=g.node[i].in[j].v;
23         dist=g.node[i].in[j].weight;
24         d2[i]=std::min(d2[i],d1[u]+dist);
25        }
26       }
27      float *temp=d1; d1=d2; d2=temp;
28     }
29   }
30 }
```

```
void
cilkBF(DGraph & g,float *d1,float *d2)
{
 float dist;


 cilk_for(int i=1;i<g.n;i++)
 {dist=INFTY;
  if((g.node[i].degIn>0)&&
            (g.node[i].inv[0]==0)
     dist=g.node[i].inw[0];
  d1[i]=dist;
 }

 d1[0]=0;
 for(int k=1;k<g.n-1;k++)
  {
   cilk_for(int i=1;i<g.n;i++)
    {
     int deg=g.node[i].degIn;
     dist=__sec_reduce_min
         (d1[g.node[i].inv[0:deg]]+
                g.node[i].inw[0:deg]);
     d2[i]=std::min(d1[i],dist);

    }
   float *temp=d1; d1=d2; d2=temp;
  }

}
```

Fig. 2. Belman-Ford algorithm implemented using OpenMP and Cilk Plus

In our Cilk Plus implementation (see Fig. 2, right), the loops 7–13 and 18–26 are parallelized using `cilk_for` construct. We also 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. Then (lines 21–24) we can simply vectorize the body of the loop using built-in function `__sec_reduce_min()` to find minimum among elements in the array given by the sum of the array `inw` and necessary elements from the array `d1` given by indices from `inv`. This is a very fine example of using the array notation.

Table 2 shows the results of experiments performed for the considered implementation of Belman-Ford algorithm, namely **basic**, **ompBF1**, **ompBF2** and **cilkBF**. Note that **ompBF2** is another implementation that uses OpenMP and

Table 2. Execution time (in seconds) of three implementations of Algorithm 1

	2x E5-2670				Xeon Phi 7120P			
max deg	basic	ompBF1	ompBF2	cilkBF	basic	ompBF1	ompBF2	cilkBF
	The number of nodes $n = 4000$							
10	0.20	0.16	0.16	0.19	4.73	0.48	0.43	0.99
20	0.30	0.21	0.16	0.20	6.24	0.84	0.87	1.03
50	0.57	0.25	0.19	0.25	10.04	1.76	1.63	1.38
100	1.05	0.25	0.26	0.35	15.29	2.34	2.05	1.50
200	2.07	0.38	0.31	0.32	27.37	2.26	2.08	1.94
500	5.13	0.67	0.35	0.42	54.97	1.75	1.93	1.70
1000	10.34	0.71	0.92	0.59	100.37	2.60	2.87	2.40
2000	22.22	1.51	1.49	1.07	200.16	5.68	6.22	4.16
	The number of nodes $n = 10000$							
10	1.70	0.75	0.83	0.52	31.89	1.02	1.10	3.07
20	2.20	0.80	0.74	0.55	39.66	1.16	1.23	3.26
50	3.74	0.82	0.79	0.71	61.50	1.48	1.54	3.60
100	6.72	0.98	0.96	0.78	97.62	2.31	2.41	4.84
200	13.09	1.22	1.35	1.08	168.09	3.72	3.60	4.98
500	32.99	2.53	2.38	1.61	369.67	7.68	8.90	9.34
1000	71.88	3.90	4.87	4.01	684.38	14.49	16.01	10.85
2000	156.82	12.65	12.68	11.63	1331.26	27.82	30.62	18.25

the same data layout as **cilkBF**. All results have been obtained for graphs generated randomly for a given number of vertices and maximum degree (i.e. the maximum number of incoming arcs). We can observe that the parallel implementations are much faster than the basic (i.e. non-parallelized) implementation of Algorithm 1. Usually **ompBF2** is faster than **ompBF1**. **cilkBF** outperforms **ompBF1** and **ompBF2** for larger and wider graphs. However, in case of our OpenMP implementations, Table 2 shows the best results chosen from several runs for various values of `ChS`. Thus, one can say that our OpenMP versions have been manually tuned. In case of **cilkBF**, the runtime system has been responsible for load balancing.

We can observe that for sufficiently large graphs all parallel implementations utilize multiple cores achieving reasonable speedup (see Fig. 3). Moreover, **cilkBF** outperforms **ompBF** significantly, especially on Xeon Phi. This is the effect of the efficient and explicit vectorization of the loop 7–9 in Algorithm 1. For this architecture it is also important to vectorize sufficiently long loops. Indeed, the speedup grows when the maximum degree (i.e. the length of the arrays `inv` and `inw`) grows.

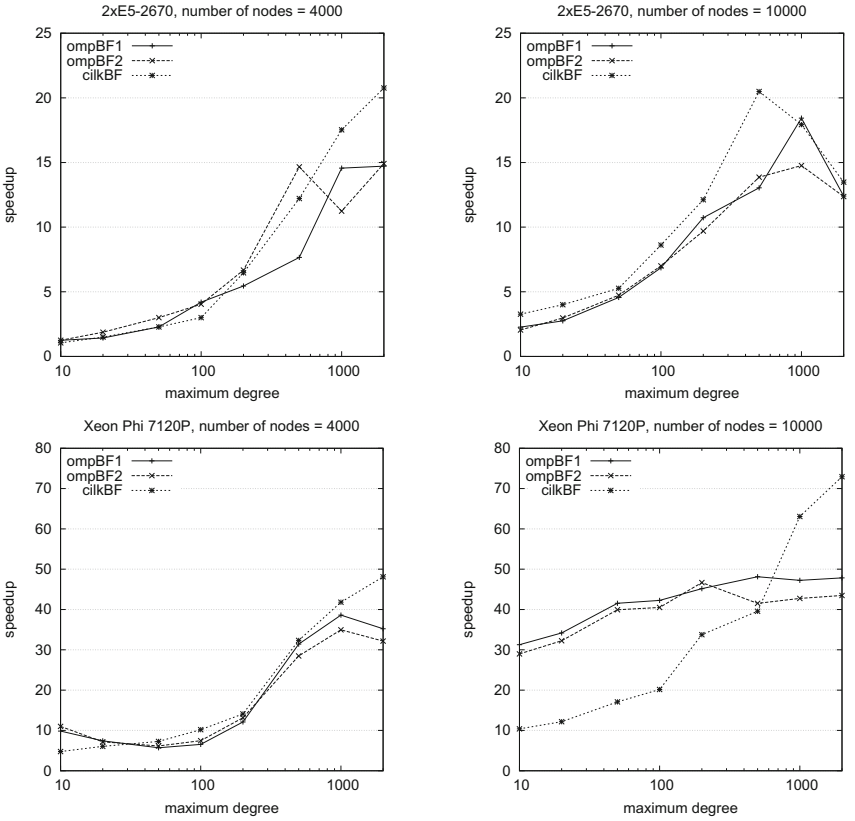

Fig. 3. Speedup of OpenMP and Cilk Plus implementations versus non-parallelized basic version of Belman-Ford algorithm

It should be noticed that we have also tested another version of **cilkBF** that uses `_Cilk_spawn _Cilk_offload` construct and where all data structures have been shared between CPU and coprocessors. Unfortunately, the need for synchronization of *Shared Virtual Memory* at the end of each iteration (i.e. the loop 16–28) leads to a very large increase in processing time and our implementation with offloading is over 10× slower than **cilkBF**. However, *Shared Virtual Memory* is perfect for exchanging irregular data with limited size, when explicit synchronization is not used frequently. Both sides (CPU and coprocessor) should operate on memory allocated locally. Local data can be persisted using more sophisticated techniques (the use of Cilk Plus together with `#pragma offload`).

4 Conclusions and Future Work

We have shown that Cilk Plus can be very easily applied to parallelize recursively defined problems like *Adaptive Simpson's Integration Rule* and such implementation can be easily modified to utilize coprocessors like Intel Xeon Phi. It is also easy to move from OpenMP to Cilk Plus and improve the performance of such algorithms by tuning data structures to utilize hardware (i.e. vector units) of modern multicore and manycore processors. For sufficiently large graphs, our Cilk implementation of Belman-Ford algorithm for solving the single-source shortest-path problem achieves really better performance than corresponding OpenMP versions of the algorithm. Thus, Cilk Plus is a good choice for people who want to concentrate on algorithms and prefer to use simple high-level programming constructs to express parallelism. Of course, it is clear that the use of OpenMP together with more advanced programming tools allows to fine-tune programs for a particular architecture [17,20]. However, this involves a much greater effort.

In the future, we plan to implement some other important computational problems using Cilk Plus. It would also be interesting and important to find problems that can benefit from using *Shared Virtual Memory*.

Acknowledgements. This work was partially supported by the National Centre for Research and Development under MICLAB Project POIG.02.03.00-24-093/13. The use of computer resources installed at Institute of Mathematics, Maria Curie-Skłodowska University in Lublin is kindly acknowledged.

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann, Burlington (2001)
2. Asai, R., Vladimirov, A.: Intel Cilk Plus for complex parallel algorithms: "enormous fast Fourier transforms" (EFFT) library. Parallel Comput. **48**, 125–142 (2015). https://doi.org/10.1016/j.parco.2015.05.004
3. Basseda, R., Chowdhury, R.A.: A parallel bottom-up resolution algorithm using Cilk. In: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, 4–6 November 2013, pp. 95–100. IEEE Computer Society (2013). https://doi.org/10.1109/ICTAI.2013.24
4. Cameron, M.: Adaptive integration (2010). http://www2.math.umd.edu/~mariakc/teaching/adaptive.pdf
5. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel Programming in OpenMP. Morgan Kaufmann Publishers, San Francisco (2001)
6. Coblenz, M.J., Seacord, R., Myers, B.A., Sunshine, J., Aldrich, J.: A course-based usability analysis of Cilk Plus and OpenMP. In: 2015 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2015, Atlanta, GA, USA, 18–22 October 2015, pp. 245–249. IEEE (2015). https://doi.org/10.1109/VLHCC.2015.7357223
7. Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. MIT Press, Cambridge (1994)

8. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High-Performance Programming. Morgan Kaufman, Waltham (2013)
9. Khaldi, D., Jouvelot, P., Ancourt, C., Irigoin, F.: Task parallelism and data distribution: an overview of explicit parallel programming languages. In: Kasahara, H., Kimura, K. (eds.) LCPC 2012. LNCS, vol. 7760, pp. 174–189. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37658-0_12
10. Kowalik, J.S., Puzniakowski, T.: Using OpenCL - Programming Massively Parallel Computers, Advances in Parallel Computing, vol. 21. IOS Press, Amsterdam (2012). http://ebooks.iospress.nl/volume/using-opencl
11. Kuncir, G.F.: Algorithm 103: Simpson's rule integrator. Commun. ACM **5**(6), 347 (1962). https://doi.org/10.1145/367766.368179
12. Leiserson, C.E.: Cilk. In: Padua, D.A. (ed.) Encyclopedia of Parallel Computing, pp. 273–288. Springer, Boston (2011). https://doi.org/10.1007/978-0-387-09766-4_2339
13. Lewin-Berlin, S.: Exploiting multicore systems with Cilk. In: Proceedings of the 4th International Workshop on Parallel Symbolic Computation, PASCO 2010, 21–23 July 2010, Grenoble, France, pp. 18–19. ACM (2010). https://doi.org/10.1145/1837210.1837214
14. Lyness, J.N.: Notes on the adaptive Simpson quadrature routine. J. ACM **16**(3), 483–495 (1969). https://doi.org/10.1145/321526.321537
15. Musaev, M., Khujayarov, I., Buriboev, A.: Accelerate the solution of problems of digital signal processing technology based INTEL CILK PLUS. Asian J. Comput. Inf. Syst. **3**, 48–51 (2015). https://doi.org/10.24203/ajcis.v3i2.2507
16. van der Pas, R., Stotzer, E., Terboven, C.: Using OpenMP - The Next Step. Affinity, Accelerators, Tasking, and SIMD. MIT Press, Cambridge (2017)
17. Rahman, R.: Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers. Apress, Berkely (2013)
18. Robison, A.D.: Composable parallel patterns with Intel Cilk Plus. Comput. Sci. Eng. **15**(2), 66–71 (2013). https://doi.org/10.1109/MCSE.2013.21
19. Stpiczyński, P.: Semiautomatic acceleration of sparse matrix-vector product using OpenACC. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015 Part II. LNCS, vol. 9574, pp. 143–152. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_14
20. Supalov, A., Semin, A., Klemm, M., Dahnken, C.: Optimizing HPC Applications with Intel Cluster Tools. Apress, Berkely (2014)
21. Wolfe, M.: High Performance Compilers for Parallel Computing. Addison-Wesley, Boston (1996)