ISSN 0361-7688, Programming and Computer Software, 2018, Vol. 44, No. 1, pp. 35–42. © Pleiades Publishing, Ltd., 2018.
Original Russian Text © N.I. V'yukova, V.A. Galatenko, S.V. Samborskii, 2018, published in Programmirovanie, 2018, Vol. 44, No. 1.

Support for Parallel and Concurrent Programming in C++

N. I. V'yukova*, V. A. Galatenko, and S. V. Samborskii*****

Scientific Research Institute for System Analysis, Russian Academy of Sciences, Nakhimovskii pr. 36/1, Moscow, 117218 Russia

**e-mail: niva@niisi.ras.ru,*

***e-mail: galat@niisi.ras.ru,*

****e-mail: sambor@niisi.ras.ru*

Received September 1, 2017

Abstract—C++ was originally designed as a sequential programming language. For development of multithreaded applications, libraries, such as Pthreads, Windows threads, and Boost, are traditionally used. The C++11 standard introduced some basic concepts and means for developing parallel and concurrent programs, but the direct use of these low-level means requires high programming skills and significant efforts. The absence of high-level models of parallelism in C++ is somewhat compensated for by various parallel libraries and directive parallelization tools (such as OpenMP), as well as by language extensions supported by some compilers (Intel CilkPlus). Nevertheless, we still require more advanced means to express parallelism in programs at the level of language standard and language library. In this survey, we consider the means for parallel and concurrent programming that are included into the C++17 standard, as well as some capabilities that are to be expected in the future standards.

DOI: 10.1134/S0361768818010073

1. INTRODUCTION

For quite a long time, increase in CPU clock rate allowed application performance to be automatically improved by employing a faster processor. However, technological limits on increasing clock rate forced developers to find other ways to speed up programs, which gave rise to various types of parallelism in modern microprocessor architectures. It is conventional to distinguish among the following types of parallelism, which can be variously combined in real systems.

• *Bit-level parallelism* speeds up operations on values due to an increased length of a machine word. For example, 64-bit values are summed up by one instruction if the length of a word is 64 bits; if the length is shorter, then a library routine may need to be called.

• *Instruction-level parallelism* means that the processor can simultaneously execute several instructions from one instruction flow. This property is inherent in pipelined, superscalar microprocessors, as well as in microprocessors with explicit parallelism at the instruction level and microprocessors with very long instruction word (VLIW) [1].

• *Data-level parallelism* means the support of instructions that operate with vectors of single-type values [2].

• *Execution-level parallelism* in multiprocessor, manycore, multicore, or hybrid systems.

Execution-level parallelism is associated with two programming paradigms: parallel and concurrent programming. These paradigms are closely related and can use the same software and hardware resources, but, essentially, they have significant differences. Parallel programming is aimed at speeding up the execution of a certain task by taking advantage of multiple equipment items (processors, cores, or computers in a distributed system). For this purpose, the task is divided into several subtasks of the same type that operate mostly with independent data fragments; once all subtasks are complete, their results are merged. In this case, programmers often try to use other levels of parallelism as well (instruction level and data level). In contrast, concurrent programming is aimed at structuring a program by allocating multiple control flows in it. Conceptually, these flows are executed concurrently, although, in fact, they can be executed on a set of processors (cores) or on one processor with context switching.

In practice, these two approaches can be combined to complement each other. For example, allocation of independent concurrent subtasks can help both structure a program and improve its performance. And yet, it is useful to distinguish between these paradigms.

To improve computational performance, modern compilers enable, to one extent or another, the automatic use of all types of parallelism mentioned above (see, for example, [3]). There are also various external parallelization tools at the source code level [4, 5]. However, the capabilities of automatic parallelization (and automatic vectorization) of programs in compilers and external tools remain quite limited because of many difficulties in implementation of code analysis and transformation that are associated with this problem.

That is why the development and standardization of software models and linguistic means at various levels is one of the main directions for exploiting the parallelism of multitask and multithread execution. These models and means would allow programmers to effectively and efficiently develop programs with a given type of parallelism, as well as port sequential software to parallel systems.

This survey is devoted to the support means for parallel and concurrent programming in the current and future standards of C++.

2. CHARACTERISTICS OF PARALLEL SYSTEMS

Applicability of different tools and models of parallel and concurrent programming significantly depends on the memory organization of the computer system. From this perspective, there are two main classes of systems:

• systems with *distributed memory*, where each node has its own memory and remote data are accessed via corresponding nodes;

• systems with *shared memory*, where all nodes have access to common memory with a single address space.

Combinations of these two approaches are *distributed shared memory* (DSM) and *virtual shared memory* (VSM). In such systems, memory is physically distributed but with support of the single address space abstraction; in this case, access to memory of another node is more "expensive" than access to node's own memory. Organization of nonuniform memory access (NUMA) differs from that of uniform memory access (UMA), which is typical of symmetric multiprocessor systems and multicore processors.

In recent years, hybrid configurations became very popular, e.g., general-purpose computing on graphics processing units (GPGPU). Another example of a hybrid system is the Intel Xeon Phi processor that incorporates a coprocessor with the many integrated core (MIC) architecture. According to some estimates, the future of high-performance computing belongs to these systems, which integrate classical CPUs and special-purpose manycore coprocessors. An advantage of hybrid systems is their high performance in combination with energy efficiency. These qualities are important for mobile device industry and other application domains, including on-board systems, robotics, and classical server-based high-performance computing.

At the same time, software development for hybrid systems faces significant challenges. Since CPUs and coprocessors have separate memory and different address spaces, it is required to copy data and results between the devices, which increases overhead costs when organizing parallel computing. Software development is also complicated by the necessity of using special dialects of C/C++ (for CUDA and OpenCL), as well as by difficulties in software porting among different hybrid systems.

To develop and standardize architectures for heterogeneous systems, the Heterogeneous System Architecture (HSA) Foundation was established in 2012 by AMD, ARM, Imagination Technologies, MediaTek, Texas Instruments, Samsung Electronics, and Qualcomm. In early 2015, the HSA specification version 1.0 was released; version 1.1 followed in 2016. The purpose of the HSA specifications is to support effective models of parallel and concurrent programming for heterogeneous systems with the use of the corresponding software and hardware means. Below are some of the key HSA features.

• Memory coherence, i.e., shared address space with homogeneous properties for CPU and accelerators. This type of memory is called the heterogeneous uniform memory access (hUMA). Other HSA memory requirements are a single byte order and support of atomic operators as in C/C++11.

• A unified low-level intermediate language (HSAIL) into which program fragments are translated to be executed on accelerators. Translation from the HSAIL into the language of a target device is carried out by specific (for the device) components (finalizers) during compilation, loading, or execution.

• A task queue management mechanism at the user level and context switching with a possibility to preempt tasks for all types of computing elements.

Thus, the HSA integrates CPUs and accelerators into a system with the common principle of organized computations, which significantly facilitates software development and porting among different HSA systems. A survey of HSA specifications can be found in [7].

The parallelism support means for C++ considered below are used in shared memory systems and heterogeneous systems.

3. MEANS OF PARALLEL AND CONCURRENT PROGRAMMING IN THE C++ STANDARDS

3.1. Support of Parallelism in the C++11 Standard

C and C++ were originally designed as sequential programming languages. For development of multithreaded applications, libraries such as Pthreads

(which implements threads according to the POSIX standard), MS Windows threads, Boost, etc. are traditionally used. Insufficiency of libraries was thoroughly substantiated in [8], where some examples were presented of commonly used code optimizations that may result in data race errors. Another disadvantage of the "library" approach noted in [8] is the absence of means for programming lock-free and wait-free algorithms.

Taking into account the analysis conducted in [8], specifications of multithreading support means for C++ were developed. The C++11 standard included the shared memory model, concepts of multithread execution and data race, multithreading support means (storage-class for the data local to the thread, `thread_local`, and the thread library, `std::thread`), library of atomic operators (`std::atomic`), and library to support asynchronous execution (`std::future`). Later, most of these means (with the corresponding modifications) were carried over to the C11 standard. For more information about the support of parallelism in C11, see [9, 10]. The use of `std::future` objects was discussed in [11].

3.2. Extended Support of Parallelism in the C++17 Standard

An important step toward the development of parallelism in C++ was the inclusion of the parallel standard template library (PSTL) into the C++17 standard. The PSTL contains the means for parallel processing of generalized containers. The PSTL specification was developed based on the experience obtained in the following projects: Intel Threading Building Blocks (TBB) [12], Microsoft C++ AMP [13], and NVidia Thrust [14].

The PSTL interfaces have an additional parameter: "parallelization policy." C++17 supports three policies:

`seq` (sequential execution);

`par` (thread-level parallelization);

`par_unseq` (thread-level parallelization and vectorization).

Listing 1 shows some examples of calling a sort algorithm.

```
std::vector<int> v = …
// Traditional call of the sort algorithm
std::sort(vec.begin(), vec.end());
// Explicit specification of the
// sequential policy
// in the sort call
std::sort(seq, vec.begin(), vec.end());
// Explicit call of parallel sorting
std::sort(par, vec.begin(), vec.end());
```

Listing 1. Variants of calling the sequential and parallel sort algorithms.

We expect that the next version of the standard, C++2x, will include two additional policies—`unseq` and `vec`—that correspond to vectorized execution without thread-level parallelization. The difference is that `unseq` allows one to change the order of computations as compared to purely sequential execution, and `vec` does not. In particular, the `unseq` policy enables effective vectorization of reduction loops (summation of vectors, scalar product of vectors, etc.), though the computational result can differ from that obtained by a sequential algorithm.

Although the support of the PSTL offers an easy way for programmers to employ parallelism of multithread and vector execution, the selection of a suitable parallelization policy in each particular case may require profiling and performance analysis.

The following example from [15] demonstrates the use of different policies. Listing 2 shows a loop of image gamma-correction. The `transform` algorithm that processes one image line is used with the `par_unseq` policy. Profiling shows that this causes execution to slow down as compared to the sequential case. This policy proves to be inadequate when the dimension of a problem is low because the overhead costs of thread starting outweigh the benefits of parallel execution.

```
void image::ApplyGamma (float g) {
  using namespace std::execution;
  for_each (image_.begin(), image_.end(),
  [g](Row &r) {
    transform(par_unseq, r.cbegin(),
    r.cend(), r.begin(),
      [g](float v) {
        return pow(v, g);
      })
  });
}
```

Listing 2. Using the `par_unseq` policy in the inner loop of image processing.

Listing 3 demonstrates a joint use of policies: processing of the whole image is paralleled at the thread level (`par` policy) and processing of each line is vectorized (`unseq` policy). This approach improves performance almost by an order of magnitude.

```
void image::ApplyGamma (float g) {
  using namespace std::execution;
  for_each (par, image_.begin(), image_.end(),
    [g](Row &r){
      transform(unseq, r.cbegin(), r.cend(),
      r.begin(),
        [g](float v) {
          return pow(v, g);
        })
      });
}
```

Listing 3. Joint use of parallelization policies.

Presently, the PSTL is implemented in the icc compiler [16] (based on Intel TBB). Its experimental implementations by Microsoft [17] and Khronos-Group [18] (based on SYCL specifications [19]) are also available. The gcc compiler, beginning with its sixth version, supports an experimental version of the

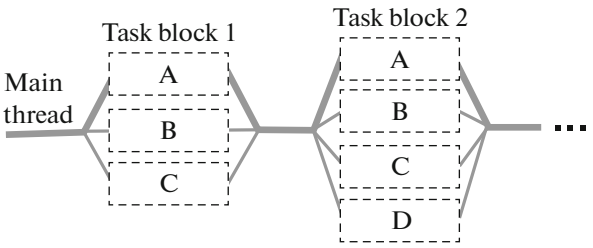

Fig. 1. Parallel execution of task blocks.

PSTL [20] implemented based on OpenMP [21]. We can expect that, in the future versions of the gcc compiler, the PSTL interface will be modified to comply with the C++17 standard.

4. PROSPECTS OF PARALLELISM SUPPORT IN C++

The next version of the standard, C++2x, is expected to include a number of new means to support parallelism: task blocks, parallel for-loops, transactional memory, and coroutines. In addition, interfaces of already existing STL classes can undergo certain modifications. In the longer term, the C++ standard may also include development tools for heterogeneous systems.

4.1. Task Blocks

The task block mechanism [22] is based on the parallel patterns library (PPL) by Microsoft and the threading building blocks (TBB) by Intel, as well as on the experience of using the Intel CilkPlus extension for C++. Task blocks are a library for thread-level parallelization of computations in a fork-join manner (see Fig. 1). The basic elements of the task block library are the class `task_block` and the function template `define_task_block`.

Consider an example of a program for parallel tree processing (Listing 4) from [22] in which all tree nodes are processed by the user-defined function compute.

```
template<typename Func>
int traverse(node *n, Func&& compute)
{
  int left = 0, right = 0;
  define_task_block([&](task_block& tb) {
    if (n->left)
      tb.run([&] {left = traverse(n->left,
      compute);});
    if (n->right)
      tb.run([&] {right = traverse(n->right,
      compute);});
  });
  return compute(n) + left + right;
}
```

Listing 4. Parallel tree processing.

A call of `define_task_block` defines a parallel block that can potentially contain calls of the method `task_block::run` for starting threads. The method `run` starts a task (certain computation) that can be executed in parallel with the thread that called `run`. When exiting the block `define_task_block`, all threads created in it by using `run` are merged into one thread.

The method run launches the user-defined functional object `f` asynchronously. This means that the exit from `run` can occur before `f` is executed. The scheduler can start the thread immediately or wait until the necessary recourses are deallocated.

An object of the `task_block` class can be created only in `define_task_block` because this class has no `public` constructors. Hence, the method `run` can be called (directly or indirectly) only from a user-defined function that is set as an argument when calling `define_task_block`.

4.2. Parallel for-Loops [23]

The C++17 PSTL offers a loop structure of the form `parallel::for_each`, which implements parallel processing of sequences (see Listing 3). This structure, however, has a limited application context and does not support many popular templates of loop computations, namely,

• concurrent processing of several sequences: `A[i] = B[i];`

• access to previous or subsequent elements of a sequence: `iter[0] = iter [1];`

• using the number of the current element of a sequence in computations:

`A[i] += i % 2 ? 1 : -1.`

Note that many of these templates allow effective vector implementation. The library of parallel for-loops offers templates for a number of parallel loop structures (see examples below).

1. Loop for sequential array processing.

```
for_loop(seq, 0, n, [&](int i) {
  y[i] += a*x[i];
});
```

This loop is equivalent to the common loop

```
for (int i=0; i<n; ++i)
  y[i] += a*x[i];
```

To parallelize this loop, it is sufficient to replace the `seq` policy with `par`.

2. Strided loop.

```
for_loop_strided(par,10,20,3,[&](int k) {
  c[k] = true;
});
```

In this example, the value "`true`" is assigned to the elements `c [10]`, `c [13]`, `c [16]`, and `c [19]`.

3. Reduction loop.

```
float dot_saxpy(int n, float a, float
x[], float y[]) {
  float s = 0;
  for_loop(par, 0, n,
```

```
    reduction(s, 0.0f, std::plus<float>()),
    [&](int i, float& s_) {
       y[i] += a*x[i];
       s_+= y[i]*y[i];
    });
    return s;
}
```

Here, the call of `reduction` defines a reduction object for the addition operator. For convenience, the library of parallel `for`-loops provides standard reductions for addition, multiplication, bit operations, minimum, and maximum.

It also provides means to describe inductive variables, which linearly depend on a loop variable. Once `for_loop` terminates, their values are set in the same manner as with sequential loop execution.

4.3. Transactional Memory

Transactional memory is a thread synchronization technology that allows groups of instructions to be executed as atomic operators. Threads can execute transactions in parallel until they begin to modify the same memory area. In this case, the transaction is rolled back, i.e., all objects modified by it in shared memory return to the previous state (before transaction execution), and an attempt is made to rerun the transaction.

The traditional thread synchronization method using locks is based on a pessimistic assumption that threads are more likely to conflict, so, in the critical interval, there must be no more than one thread at a time. In contrast, the use of transactions is based on an optimistic assumption that threads conflict only rarely and, therefore, can simultaneously execute operations on a shared object in a speculative mode; in the case of a conflict, the result is rolled back. Transactions can be more effective than locks, e.g., when threads modify a shared container while rarely accessing the same elements.

The current version of the technical specification [24] introduces atomic operators, i.e., code blocks preceded by one of three key words: `atomic_noexcept`, `atomic_commit`, or `atomic_cancel`. The difference among the three types of atomic blocks is in their response to exceptions.

• `atomic_noexcept`: in the case of an exception, `std::abort` is executed and a program terminates.

• `atomic_cancel`: if an exception does not support transaction rollback, then `std::abort` is executed; otherwise, transaction rollback (return to the initial state) is carried out with exit from the atomic block (in this case, an exception is thrown).

• `atomic_commit`: in the case of an exception, the transaction is fixed.

Listing 5 demonstrates the use of atomic blocks, where atomic transactions implement the methods "`deposit`" and "`withdraw`" (from an account), as well as the function "`transfer`" (between accounts).

```
class Account {
   int bal;
 public:
   Account(int initbal) { bal = initbal; };
   void deposit(int x) {
     atomic_noexcept {
       this.bal += x;
     }
   };
   void withdraw(int x) {
     deposit(-x);
   };
   int balance() { return bal; }
}
void transfer(Account a1, Account a2, int x) {
  atomic_noexcept {
    a1.withdraw(x);
    a2.deposit(x);
  }
}
```

Listing 5. Use of atomic transactions.

The standard specifies the constraints imposed on operations that can be used in atomic blocks by introducing the concepts of transaction-safe functions, operators, and expressions. By definition, transaction-unsafe computations are those the result of which cannot be rolled back (e.g., output to a console). Formally, transaction-unsafe computations are operations on volatile data, calls of transaction-unsafe functions, and use of transaction-unsafe assembler insertions. Declarations of transaction-safe functions must have the specifier `transaction_safe`.

In addition to atomic blocks, the specification [24] defines `synchronized` code blocks, which are executed as if each such block is surrounded by the acquire and release of a common mutex. Computations in `synchronized` blocks do not have to be transaction-safe. The mechanism of synchronized blocks offers a simple (but not always efficient) way to synchronize operations on shared data. Listing 6 shows an example from [24].

```
int f() {
  static int i = 0;
  synchronized {
    printf("before %d\n", i);
    ++i;
    printf("after %d\n", i);
    return i;
  }
}
```

Listing 6. Example of a synchronized block.

Beginning with version 4.7, the GCC compiler supports an experimental implementation of transactions that is so far based on an earlier version of the transactional memory specification for C++.

4.4. Coroutines [25]

A coroutine is a generalization of a subroutine. The execution of a coroutine can be paused and then continued from the position of the last pause. A subroutine can return control to a caller routine only once the execution of this subroutine is complete. In contrast, a coroutine can return control without termination, preserving its current state; then, its execution can be continued from arbitrary context, possibly, from another thread. Coroutines facilitate the implementation of some typical programming idioms, such as event-driven programming, cooperative multitasking, and generators.

Consider an example of a function that is used as an integer generator and is implemented using the class `vector<int>` (see Listing 7).

```
std::vector<int> getInts(int start, int stop,
int inc= 1){
  std::vector<int> ints;
  for (int i= start; i < stop; i += inc)
    ints.push_back(i);
  return ints;
}
int main(){
  auto numbers= getInts(-10, 11);
  for (auto n: numbers) std::cout << n << " ";
  std::cout << "\n";
}
```

Listing 7. Data generation using a container.

Compare this approach with the following implementation that uses a coroutine (see Listing 8).

```
1   generator<int>   generatorForInts(int
start,
2     int inc= 1){
3 for (int i= start;; i += inc)
4    co_yield i;
5 }
6 int main(){
7   auto nums= generatorForInts(-10);
8   for (int i= 1; i <= 20; ++i)
9       std::cout << nums << " ";
10   std::cout << "\n\n";
11 }
```

Listing 8. Data generation using a coroutine.

The use of `generatorForInts` (line 6) returns a generator, i.e., an object of the class `generator<int>`, which is assigned to the variable nums. Each use of nums in the loop (line 9) continues the execution of the coroutine `generatorForInts` from the position where it was paused. When executing the operator `co_yield` (line 4), the next result of the coroutine is computed and it is then paused to return control to the caller. Thus, in this case, the use of the coroutine implements lazy evaluations as they are requested. Another feature of coroutines demonstrated by this example is that they can return a sequence of values (in this case, a potentially infinite one) rather than a single value. For more examples, see [26].

Syntactically, a coroutine differs from a subroutine only in that its body includes the operator `co_yield`, `co_await`, or `co_return` (or a loop `range_for` with the operator `co_await`). The type of a value returned by a coroutine is a class that must contain certain elements and methods specified by the standard. Whereas the existing coroutines are quite easy to use, the development of new coroutines and the corresponding classes requires fairly high programming skills.

Coroutines were expected to be included into the C++17 standard; however, their current specification gave rise to a number of objections by members of the working group on C++ standardization [27]. In particular, the introduction of new key words into the language is subjected to criticism, as this may require a significant redesign of existing code to enable the use of coroutines. It is also noted that this approach has not yet undergone proper verification in high-performance computing, which is traditionally based on Linux platforms. So far, coroutines are implemented only in the compiler of Microsoft Visual Studio. The LLVM compiler provides their experimental support [28] only in version 4.0.0 (released March, 2017), while the corresponding project for the gcc compiler has not yet been officially announced. That is why, for now, it was proposed to leave coroutines at the stage of technical specification.

4.5. Other Suggestions for Development of Parallelism in C++

The working group on C++ standardization is also considering a number of other suggestions for development of parallelism [29], which are briefly described below.

1. Supplementing the class `future` with the method `then`. In asynchronous programming, once a certain asynchronous task is complete, it is often required to launch another one while providing it with the result obtained by the first task. The class `future` is supplemented with the method `then` in order to build such chains of asynchronous actions. The corresponding example is shown in Listing 9.

```
#include <future>
using namespace std;
int main() {
  future<int> f1 = async([]() {return 123;});
  future<string> f2 = f1.then([](future<int> f) {
    return to_string(f.get());
  });
}
```

Listing 9. Organizing a chain of asynchronous actions by using the method `then` of the class `future`.

2. Latches and barriers. These classes allow one to lock a group of threads until a certain operation is complete. A latch can be used only once, while a bar-

rier can be used repeatedly. Barriers are useful to synchronize threads in a repeatedly executed parallelized task.

3. Atomic pointers. New classes `atomic_shared_ptr` and `atomic_weak_ptr` are thread-safe atomic analogs of the classes `shared_ptr` and `weak_ptr`. For these classes, standard atomic operators are defined.

4.6. Support of Heterogeneous Computations in C++

CUDA [30] and OpenCL [31] are basic high-performance computing technologies for heterogeneous systems. CUDA is a proprietary technology that supports only NVIDIA graphics cards; OpenCL supports a wider range of devices, including digital signal processors and programmable logic devices. Both technologies support some dialects of C and C++. Nevertheless, both CUDA and OpenCL imply low-level programming that involves many details of mapping a desired computation onto a target configuration.

The OpenMP [21] and OpenACC [32] standards significantly facilitate programming for heterogeneous systems. Advantages of directive parallelization are the uniformity of directives in different languages and a possibility of incremental parallelization of legacy code, beginning with the most critical sections of a program. However, directive means are not always harmoniously combined with C++ means, particularly, with templates. Directives make the program difficult to understand because they can alter the order of computations as compared to its source code. Software debugging gets complicated as well, because it becomes difficult for the compiler to conduct proper diagnostics, e.g., of semantic errors in directives, and to generate adequate debugging information.

There are projects aimed at developing high-level programming abstractions for heterogeneous systems in C++: Kokkos [33], C++ AMP [13], SYCL [19], etc. Meanwhile, software developers in various application domains (video games, financial trading, embedded systems, etc.) become increasingly insistent in requiring a parallel programming model for heterogeneous systems to be included into the C++ standard. In [34], it was proposed that the future C++ standards should support programming based on the SYCL specification developed by Khronos™ Group. The SYCL consists of two interconnected components: C++ library and SYCL compilers for devices. The SYCL project also includes an implementation of the C++17 PSTL [18] with a possibility to use GPUs and additional CPUs through the corresponding parallelization policy (sycl).

As compared to direct programming in OpenCL, the advantages of the SYCL are the support of all modern C++ programming capabilities for heterogeneous systems and unified source code, including the code executed on the CPU and computational cores for accelerators, which are specified as lambda expressions. Parallelism of computations is expressed explicitly using constructions like `parallel_for`.

In [34], a model of separate address spaces was proposed in which the data transferred between CPU memory and device memory must be defined explicitly in the program. Meanwhile, other possible memory models (virtual shared memory with and without cache coherence), as well as their possible support in the future versions of the standard, are being considered. Examples of SYCL programs and their comparisons with the corresponding OpenCL programs can be found in [35].

CONCLUSIONS

The purpose of this work was to analyze and illustrate the trends and prospects for development of parallel programming means for multicore and heterogeneous architectures in C++ both at the language level and at the STL level. Before the C++11 standard, multithreaded programs in C++ could be developed only based on external libraries. The absence of a shared memory model and concepts related to execution parallelism often resulted in the fact that a debugged multithreaded program worked incorrectly upon porting to a different operating system or even to a compiler of a different version.

The C++11 standard laid the foundation of parallelism: it introduced the basic concepts (memory model, multithread execution, thread progress, and data race), storage class `thread_local`, basic means for thread control and synchronization, and means for asynchronous execution. Direct use of these basic low-level means for application development, however, requires high programming skills. The concepts underlying these means are quite difficult to comprehend. There is an opinion that only several dozen people in the world thoroughly understand the C++ memory model in all its details.

The need for high-level parallelism is more or less satisfied with directive parallelization tools (based on the OpenMP standard, etc.), which are supported by modern compilers, as well as with effective parallel libraries for many application domains. This, however, cannot fully replace high-level means for parallel and concurrent programming at the language and STL levels.

Thus, for a wide use of parallelism in C++ on the basis of the means introduced in C++11, a system of high-level buildups that implement typical templates of parallelism in the form adequate for use in a wide range of algorithms should be developed [36]. An important step toward "democratization" of parallelism in C++ was the introduction of the PSTL in the C++17 standard. The means considered in this survey, which we expect to be included into the future versions of the standard, significantly facilitate parallel and concurrent programming in C++ and extend its capa-

bilities. These and other means are already available in various business systems and open-source projects.

The progress in and increasing application of heterogeneous systems created an urgent need for standardization of memory models and linguistic means that allow one to develop parallel programs for these systems. We can expect that the future versions of the C++ standard will support programming based on the SYCL/OpenCL or HSAIL models for heterogeneous systems.

REFERENCES

1. Shumakov, S.M., *Obzor metodov optimizatsii koda dlya protsessorov s podderzhkoi parallelizma na urovne komand* (Survey of Code Optimization Methods for Processors Supporting Instruction-Level Parallelism), Moscow: Inst. Sistemnogo Program. Ross. Akad. Nauk, 2002, pp. 4–59.
2. V'yukova, N.I., Galatenko, V.A., and Samborskii, S.V., Use of vector extensions for modern processors, *Program. Inzheneriya,* 2016, no. 4, pp. 147–157.
3. Volkonskii, V.Yu., Breger, A.V., Buchnev, A.Yu., Grabezhnoi, A.V., Ermolitskii, A.V., Mukhanov, L.E., Neimanzade, M.I., Stepanov, P.A., and Chetverina, O.A., Software parallelization methods in an optimizing compiler, *Vopr. Radioelektron.,* 2012, vol. **4**, no. 3, pp. 63–88.
4. Cetus: A Source-to-Source Compiler Infrastructure for C Programs. https://engineering.purdue.edu/Cetus.
5. Bakhtin, V.A., Zhukova, O.F., Kataev, N.A., Kolganov, A.S., Kryukov, V.A., Podderyugina, N.V., Pritula, M.N., Savitskaya, O.A., and Smirnov, A.A., Automating the parallelization of software complexes, *Tr. XVIII Vseros. nauchn. konf. Nauchnyi servis v seti Internet* (Novorossiisk, 2016) (Proc. XVIII All-Rus. Sci. Conf. Scientific Service on the Internet), Moscow: Inst. Prikl. Mat. Keldysha, 2016, pp. 76–85.
6. HSA Foundation. http://www.hsafoundation.com/standards.
7. Paltashev, T. and Perminov, I., Heterogeneous architecture for CPU, GPU, and DSP, *Otkrytye sist. SUBD,* 2013, no. 8, pp. 12–15. http://www.osp.ru/os/2013/08/13037850.
8. Boehm, H.-J., Threads cannot be implemented as a library, *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* 2005, pp. 261–268.
9. V'yukova, N.I., Galatenko, V.A., and Samborskii, S.V., Multithreading support in the C11 standard, *Program. Inzheneriya,* 2013, no. 9, pp. 2–8.
10. V'yukova, N.I., Galatenko, V.A., and Samborskii, S.V., Memory model of multithreaded programs in the C11 standard, *Program. Inzheneriya,* 2013, no. 10, pp. 2–9.
11. Williams, A., *C++ Concurrency in Action: Practical Multithreading,* Manning Publications, 2012.
12. Intel Developer Zone, Intel Threading Building Blocks. https://software.intel.com/en-us/intel-tbb.
13. Microsoft Developer Network, C++ AMP Overview. https://msdn.microsoft.com/ru-ru/library/hh265136. aspx.
14. NVIDIA Accelerated Computing, Thrust. https://developer.nvidia.com/thrust.
15. Dvorskii, M., Parallel algorithms of the standard template library. https://events.yandex.ru/lib/talks/4347.
16. Parallel STL: Parallel algorithms in standard template library. https://software.intel.com/en-us/articles/parallel-stl-parallel-algorithms-in-standard-template-library.
17. Parallel STL. http://parallelstl.codeplex.com.
18. Open source parallel STL implementation. https://github.com/KhronosGroup/SyclParallelSTL.
19. Khronos Group, SYCL Overview. http://www.khronos.org/sycl.
20. The GNU C++ library manual, Chapter 18: Parallel mode. https://gcc.gnu.org/onlinedocs/libstdc++/manual.
21. The OpenMP API specification for parallel programming. http://www.openmp.org/specifications.
22. Task block (formerly task region) R4. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf.
23. Template library for parallel for-loops. http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0075r1.pdf.
24. Technical specification for C++ extensions for transactional memory. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf.
25. Technical specification for C++ extensions for coroutines. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4649.pdf.
26. C++ User Group, Russia, C++ coroutines: A negative overhead abstraction. http://cpp-russia.ru/?page_id=991.
27. Coroutines belong in a TS. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0158r0.html.
28. LLVM Coroutines. http://releases.llvm.org/4.0.0/docs/leaseNotes.html#llvm-coroutines.
29. Technical specification for C++ extensions for concurrency. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4538.pdf.
30. CUDA parallel computing. http://www.nvidia.ru/object/cuda-parallel-computing-ru.html.
31. Khronos Group, OpenCL Overview. http://www.khronos.org/opencl.
32. What is OpenACC? http://www.openacc.org.
33. Edwards, H.C. and Sunderland, D., Kokkos array performance-portable manycore programming model, *Proc. Int. Workshop Programming Models and Applications for Multicores and Manycores,* New Orleans, 2012, p. 1–10.
34. Wang, M., Richards, A., Rovatsou, M., and Reyers, R., Khronos's OpenCL SYCL to support heterogeneous devices for C++. http://www.open-std.org/jtc1/sc22/wg21/docs/pers/2016/p0236r0.pdf.
35. SYCL Tutorial 1: The Vector Addition. http://www.codeplay.com/portal/sycl-tutorial-1-the-vector-addition.
36. Kukanov, A., Parallelism in C++: Manage the application and not the threads. https://events.yandex.ru/lib/talks/1795.

Translated by Yu. Kornienko