A solution to the problem of parallel programming

Edward Givelberg
ed.givelberg@gmail.com

November 26, 2018

The problem of parallel programming is the most important open problem of computer engineering. We show that object-oriented languages, such as C++, can be interpreted as parallel programming languages, and standard sequential programs can be parallelized automatically. Parallel C++ code is typically more than ten times shorter than the equivalent C++ code with MPI. The large reduction in the number of lines of code in parallel C++ is primarily due to the fact that communications instructions, including packing and unpacking of messages, are automatically generated in the implementation of object operations. We believe that implementation and standardization of parallel object-oriented languages will drastically reduce the cost of parallel programming. This work provides the foundation for building a new computer architecture, the multiprocessor computer, including an object-oriented operating system and more energy-efficient, and easily programmable, parallel hardware architecture. The key software component of this architecture is a compiler for object-oriented languages. We describe a novel compiler architecture with a dedicated back end for the interconnect fabric, making the network a part of a multiprocessor computer, rather than a collection of pipes between processor nodes. Such a compiler exposes the network hardware features to the application, analyzes its network utilization, optimizes the application as a whole, and generates the code for the interconnect fabric and for the processors. Since the information technology sector’s electric power consumption is very high, and rising rapidly, implementation and widespread adoption of multiprocessor computer architecture will significantly reduce the world’s energy consumption.

1 Introduction

Parallel programming is considered to be very difficult. Over the past several decades countless parallel programming languages, specialized systems, libraries and tools have been created. Some of these, such as Google’s MapReduce, provided adequate answers for specific problem domains, however no general solution has emerged. In this paper we show that, surprisingly, a simple and general solution to this problem exists based on parallel interpretation of established and widely used object-oriented languages, such as C++, Java and Python. We propose to incorporate the parallel interpretation into the standards of object-oriented programming languages. Programmers competent in these languages will be able to write parallel code with ease, and to translate it to an executable object simply by specifying an appropriate compiler flag.

Practically all computing devices today have multiple processors. By 2004 the speed of individual processors reached a peak and today faster computation is possible only by deploying multiple processors in parallel on the given task. This has proved to be very difficult, even when the task is in principle trivially parallelizable. Parallel programming is slow, tedious and requires additional expertise; debugging is extremely difficult. In short, the cost of parallel programming is very high. It is now the main obstacle to the solution of computational problems. We are still seeing improvements in hardware, and although the speed of individual processors remains bounded, networking bandwidth is expected to increase [2], and the cost of hardware components continues to decline. It is possible to build computing systems with very large number of cheap components, but the hardware cannot be adequately utilized without parallel programming. Almost all of software is sequential. We do not know how to build easily programmable parallel computers. This paper
lays out a plan for doing this.

After 2004 CPU manufacturers started producing multi-core CPUs in an attempt to boost performance with parallel computing, but their primary focus moved to improving the energy efficiency of computation. The energy cost can be significantly reduced when multiple cores and shared memory are placed on the same integrated circuit die. The processing cores and the RAM are typically interconnected using a bus, which becomes a bottleneck if the number of cores is increased. In practice, this architecture is limited to about a dozen cores. This problem is resolved by building a network on the chip (NoC), which makes CPUs with thousands of cores possible. Such systems have been shown to be very energy efficient, but programming them is difficult, and they are still in the early stages of research and development. Presently, the information technology sector consumes approximately 7% of the global electricity [7], and this number is growing rapidly. A solution to the problem of parallel programming is likely to lead to substantial improvement in the energy efficiency of computing systems, and to significantly affect the world’s energy consumption.

It is probably impossible to survey all of the ideas that have been proposed to tackle the problem of parallel programming, but the two most important models are shared memory and message passing. The shared memory model is based on the naive idea that the computation should be constructed using common data structures, which all computing processes use simultaneously. Today, most of the parallel computing software uses threads, which are sequential processes that share memory. The problems with threads are eloquently described in [15]. Threads is a terrible programming model. Despite this, and the fact that shared memory is not scalable, programming frameworks with artificially generated global address space have been proposed.

The message passing programming paradigm, on the other hand, acknowledges that processes generally reside on distant CPUs and need to exchange messages. Most of the scientific computing software uses the Message Passing Interface (MPI), which over the years has become a bloated collection that includes both low-level messaging routines of various kinds and library functions encoding complex algorithms on distributed data structures. Large-scale computations attempting to maximize the performance of a cluster of multicore CPUs need to combine shared memory and message passing, incurring significant additional programming cost.

Shared memory and message passing share a common fundamental flaw: both approaches use processes, and co-ordinating multiple concurrent processes in a computation is a nearly impossible programming task. We draw an analogy to the problem with the goto statement, which was discredited in the 1960s and 1970s. Unrestrained use of goto leads to an intractable number of possible time-ordered sequences of events. The concept of a computational process is not a suitable abstraction for parallel programming.

The solution lies in object-level parallelism. We perceive the world primarily as a world of objects, a world where a multitude of objects interact simultaneously. From the programmer’s point of view interactions between objects are meaningful and memorable, unlike interactions of processes exchanging messages. Object-oriented languages have been around for decades and today they are the most widely used programming languages in practice, yet, remarkably, all existing object-oriented programming languages are sequential. The inherent natural parallelism of the object-oriented programming paradigm has not been understood. We do not know of any proposal to interpret an existing object-oriented programming language as a parallel programming language.

In section 2.1 we define an abstract framework for parallel object-oriented computation. Throughout the rest of this paper we use C++, but our results apply to a variety of object-oriented languages. In section 3 we show that C++ can be interpreted as a parallel programming language within this framework without any change to the language syntax. Parallel interpretation of C++ differs from the standard sequential C++, but in section 3.3 we describe a new technique for automatic parallelization of sequential code. Standard sequential C++ code can be ported to run on parallel hardware either automatically or with a relatively small programming effort.

In section 4 we show that parallel C++ is a powerful and intuitive language where object-oriented features are much more naturally expressed than in standard sequential C++. In our experience programs written in parallel C++ are at least ten times shorter than the equivalent programs written in C++ with MPI. The large reduction in the number of lines of code in parallel C++ is primarily due to the fact that communications instructions, including packing and unpacking of messages, are automatically generated in the implementation of object operations.

For decades the central processing unit (CPU) was the most important component of the computer. Continuous improvements in application performance came mainly as a result of engineering increasingly faster CPUs. As a result, the complexity of applications is typically measured by estimating the number of multiplications they perform. In a multiprocessor system the CPU is no longer central, and the cost of moving data is not negligible with respect to the cost of multiplication, yet the prevailing view continues to be that all of the important work done by an application is carried out inside the CPUs, and the network is merely a collection of pipes between them. This state of technology is in stark contrast to observations in neuroscience which suggest that the network is much more important than the individual processors. Yet, the mechanism of computation in the brain is not under-
An abstract framework for object-oriented computing

“Objects are like people. They’re living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things.”

(Steve Jobs)

2.1 The Model

An object is an abstract autonomous parallel computing machine. An application is a collection of objects that perform a computation by executing methods on each other. An object is represented by an agent, which is a collection of processes that receive incoming method execution requests, execute them and send the results back to the client objects. An agent can process multiple method execution requests simultaneously. It can also represent several objects simultaneously, effectively implementing a virtual host where these objects live.

An object is accessed via a pointer (sometimes referred to as a remote pointer, or a generalized pointer), which contains the address of the virtual host representing the object, as well as the address of the object within the virtual host.

An application is started as a single object by the operating system, which first creates a virtual host and then constructs the application object on it. The objects of the application may request the operating system to create new virtual hosts and construct new objects on them.

2.2 Related concepts

The abstract framework of section 2.1 is the foundation for all of the results of this paper. It is instructive to examine the shortcomings of some of the previous approaches. The C++ standard defines an object as a region of storage [13]. Standards of other object-oriented languages avoid defining an object directly. In Python an object is described as an “abstraction for data” [3]. The wikipedia entry for object [5] describes it as follows: “an object can be a variable, a data structure, a function, or a method, and as such, is a value in memory referenced by an identifier”.

We learned about the existence of the actor model only after we completed the research of this paper. The actor model [12] is an abstract model for distributed computing. Actors perform computations and exchange messages, and can be used for distributed computing with objects. A number of programming languages employ the actor model, and many libraries and frameworks have been implemented to permit actor-style programming in languages that don’t have actors built-in [4]. Yet, the usefulness of the actor
model has been limited, and it does not provide a solution to the problem of parallel programming. Again, the reason is that processes exchanging messages is not a suitable model for parallel programming (see Introduction).

The difficulty in identifying the relevant abstract concepts and understanding their significance can be appreciated by the fact that 45 years of research have failed to clarify the relationship between the actor model and object-oriented computing. The object-oriented model is a higher level of abstraction. The network, the computational processes and the messages are not included in the model. They are "implementation details". Yet, an object can do everything that an actor can, so actors are useful only because they can implement objects. In section 3.3 we show that in object-oriented languages remote objects can be constructed naturally, making the actor-model libraries obsolete. The object-oriented model supersedes the actor model.

3 Language interpretation

We show that C++ code, without any changes to the language syntax, can be executed in parallel using the abstract framework described in section 2.1.

3.1 Remote objects

```cpp
Host * host = new Host("machine1");
Object * object = new(host) Object(parameters);
result = object->ExecuteMethod(some, parameters);
```

The code in Figure 1 is compliant with the standard C++ syntax. It creates a virtual host, constructs an object on it and executes a remote method on that object. We interpret all pointers as generalized pointers. The virtual host object is provided by the operating system, and is associated with a physical device. We use the "placement new" operator to construct the object on a given virtual host. Using a generalized pointer a method is executed on a remote object. By-value parameters are serialized and sent over the network to the remote host. Once the remote execution completes, the result is sent back. The treatment of by-reference parameters is more complicated: the simplest solution is to serialize the parameter, send it to the remote host and, upon completion of the method execution, to serialize it and to send it back. When the parameter is a complex object, and the changes made by method execution are relatively small, there may be a more efficient method to update the original parameter object.

3.2 Causal asynchronous execution

Sequential interpretation of an object-oriented language precludes parallel computation with remote objects: whenever an object executes a method on another (remote) object, it is obliged to wait for the completion of this operation before executing the next statement. While it is possible for several objects to simultaneously execute methods on a given object, this will never happen if the application is started as a single object.

Imagining the object as an intelligent, living, breathing thing, it could proceed with its computation immediately after initiating remote method execution, and stop to wait for its completion only when its results are needed. We call this causal asynchronous execution. Causal asynchronous execution enables parallel computation and provides a natural way to co-ordinate objects, as shown in Figure 2.

```cpp
bool completed = remote_object->ExecuteMethod();
// do something while the method is being executed
SomeComputation();
// wait for (remote) method completion
if (completed) {
  // method execution has completed
  AnotherComputation();
}
```

In this example the purpose of the if statement is to suspend the execution of AnotherComputation until the value of the variable completed is set by the remote method.

Despite its simplicity, parallel C++, i.e. C++ with causal asynchronous interpretation, has great expressive power and is sufficiently rich to implement the most complex parallel computations. The programmer constructs a parallel computation by co-ordinating high-level actions on objects, while the underlying network communications are generated by the compiler. Computation and communication overlap naturally, as in the example in Figure 2 and large, complex objects can be sent over the network as parameters of remote object methods. This is ideally suited to utilizing high network bandwidth and avoiding the latency penalty incurred by small messages. In section 3.4 we introduce additional mechanisms for fine-grained control of parallelism.

3.3 Automatic code parallelization

The construction of virtual hosts described in section 3.1 allows the programmer to control the placement of remote objects, but this can be done automatically by the operating system. Virtual hosts can be created
implicit by the compiler, and the operating system can assign virtual hosts to physical processors, and construct the application’s objects on these hosts at run time.

The remote placement of objects can be applied to all programs running on the system. This transformation alone does not parallelize program execution, but it is likely to improve the overall utilization of a multiprocessor system.

We can expect that any sufficiently complex sequential program will contain code sections that parallelize automatically. In order to parallelize a given program remote object placement must be combined with causal asynchronous code execution. An object can avoid causality violation at runtime by simply never using results of remote operations before they become available. Such design does not prevent potential deadlocks. Furthermore, remote pointers can be abused, so ultimately the programmer is responsible for the causality correctness of the code. There is, however, a wide range of use cases where causality can be enforced by the compiler using control flow graph analysis (see examples in section 4).

Automatic parallelization is a difficult and active area of research. Much of this work has focused on parallelizing loop execution. Task parallelization typically requires the programmer to use special language constructs to mark the sequential code, which the compiler can then analyze for parallelization. The object-level parallelization we introduced here requires no new syntax, and, arguably, well-structured serial object-oriented code can be parallelized either automatically, or with minimal programming effort.

### 3.4 Detailed control of parallelism

We now extend causal asynchronous execution to enable a more fine-grained control of parallelism. In this model compound statements and iteration statements are also executed asynchronously, and subject to causality. In addition, we define the nested compound statement to be a barrier statement. This means that prior to its execution the preceding statements in the parent compound statement must finish execution, and its own execution must finish before the execution of the following statement starts (see example in Figure 3). Similarly, we require that when the barrier statement is the first statement in the iteration, the preceding iteration must finish before the barrier statement is executed. These definitions allow the programmer to describe parallelism in detail, as shown by the examples in Figure 4.

Clearly, the price for the introduction of mechanisms for detailed control of parallelism is the increased distance in the semantic interpretation between parallel C++ and standard sequential C++.

### 4 Parallel C++ examples

We present examples illustrating the expressive power of parallel C++. Our parallel C++ research started with the problem of finding an efficient way to carry out data-intensive computations, i.e. computations where disks are used like RAM. With the changing ratio between the cost of arithmetic operations and the cost of data movement (see [2]), new applications become possible, but the challenge is to deploy the hardware in parallel. The computation of a large 3D Fourier transform (see section 4.4) is a natural test problem where a significant part of the computation is the movement of data. Our code was initially written in C++ with MPI, but we found that the natural solution to the problem was to arrange the computation using large objects that can be deployed on remote devices. With the introduction of parallel C++ the time for an implementation of such a solution is reduced from many months to a few weeks.

#### 4.1 Array objects

The syntax of array operations applies naturally to remote pointers. The array in the following example is allocated on a remote host, and the array operations require sending the values of x and a[24] over the network.

```cpp
double * a = new (remote_host) double([1024];
a[2] = 22.22 + x;
double z = a[24] + 3.1;
```

#### 4.2 MapReduce

A basic example of MapReduce functionality can be implemented with only a few lines of parallel C++ code, as shown in Figure 5.
A solution to the problem of parallel programming

```cpp
for (int i = 0; i < N; i ++)
{
    objectA[i]->computation();
    objectB[i]->computation();
}
```

(a) parallel

```cpp
for (int i = 0; i < N; i ++)
{
    objectA[i]->computation();
    objectB[i]->computation();
}
```

(b) sequential iterations

```cpp
for (int i = 0; i < N; i ++)
{
    objectA[i]->computation();
}
```

(c) parallel iterations

```cpp
for (int i = 0; i < N; i ++)
{
    objectA[i]->computation();
    {
        objectB[i]->computation();
    }
}
```

(d) sequential

Figure 4: Iteration statements examples.
- (a): potentially all 2N statements are executed in parallel.
- (b): all iterations are sequential, but each iteration has two potentially parallel statements.
- (c): potentially N iterations are executed in parallel, but each iteration has 2 statements that are executed sequentially.
- (d): all 2N computations are executed sequentially.

```cpp
int NumberOfWorkers = 44444444;
Worker * workers[NumberOfWorkers];
for (int i = 0; i < NumberOfWorkers; i ++)
    workers[i] = new (host[i]) Worker();
for (int i = 0; i < NumberOfWorkers; i ++)
    result[i] = workers[i]->compute(data[i]);
double total = 0.0;
for (int i = 0; i < NumberOfWorkers; i ++)
    total += result[i];
```

Figure 5: Example: MapReduce. The workers array is assigned in parallel, with each worker being constructed on its virtual host. The compute methods are also executed in parallel. We rely on the compiler to enforce causality in the execution of the reduction loop. It starts executing only after result[0] becomes available, and it executes sequentially, according to the definition of causal asynchronous execution.

The master process allocates workers on remote hosts, initiates a method execution on each worker and sums up the result. If the data[i] object is not located on host[i], it will be copied there over the network. This code is shorter and easier to write than the code that uses Google's library. Moreover, as we show in section 5, the parallel C++ compiler may be able to generate more efficient code by optimizing network operations.

4.3 Breadth-First Search on a large graph

Distributed BFS on a large graph is a standard benchmark problem [1]. We implemented a straightforward algorithm in C++/MPI using over 2000 lines of code, and in parallel C++ with less than 200 lines of code.

The graph data is divided into N objects, each containing an array of vertices with a list of edges for each vertex. We create N virtual hosts, one for each available processor, and allocate a graph object on each host. The main object initiates the BFS by invoking the BuildTree method on each graph object (see Figure 6). The computation proceeds with several (typically

```cpp
void Graph::BuildTree(VertexId root_id)
{
    int root_owner = VertexOwner(root_id);
    if (this->id() == root_owner)
        frontier.push_back(v[root_id]);
    EdgeList * E = new EdgeList[N];
    bool finished = false;
    while (!finished)
    {
        SortFrontierEdges(E);
        // remote, asynchronous, in parallel
        for (int i = 0; i < N; i ++)
            graph[i]->SetParents(E[i]);
        // finish BFS when all frontiers are empty
        finished = true;
        for (int i = 0; i < N; i ++)
            if (finished & graph[i]->isEmptyFrontier())
                finished = false;
    }
}
```

Figure 6: Building the BFS tree. The frontier is initialized with the root vertex by its owner. Iterations continue until all frontiers are empty. In each iteration the local frontier edges are sorted into N EdgeList lists, one for each graph object. Communication of all the lists begins thereafter. After the execution of all SetParents methods has finished all graph objects are asked if their frontier is empty.
We used 15,000 lines of C++ code with MPI. The equivalent variables. This could be more conveniently achieved visited yet. The graph object that owns the root ver-
text initializes its local frontier with the root vertex. In
each iteration the local frontier edges are sorted into N lists, one for each graph object. The vertex on the
other end of each frontier edge becomes the child in the
tree, unless it was visited before. The new frontier set consists of the new children. To set the parent
links and to update the frontier set every graph object executes a method on every other graph object, sending
it the corresponding list of edges. This is done in the SetParents method, whose parameter is a large
object of type EdgeList, which is serialized and sent over the network. The calls to SetParents execute in parallel after the completion of SortFrontierEdges.

BFS iterations stop when all local frontiers are empty. We used \( N^2 \) messages to set the values of the finished variables. This could be more conveniently achieved using an allreduce library function, like those implemented in MPI. In parallel C++ such functions could be, for example, implemented in the standard library using specialized containers for collective operations. Notice that the while iterations are executed sequentially because they causally depend on the value of finished.

4.4 3D Fourier transform

We computed the Fourier transform of a 64 TB array of \( 16384^3 \) complex double precision numbers on an 8-node cluster shown in Figure 7. The total computation time was approximately one day, and it could be significantly improved with code optimization. More importantly, the hardware system could be redesigned to achieve a better balance between the components. Using more powerful hardware components a similar computation can be carried out inexpensively with a 2 PB array on a suitably configured small cluster. We implemented the Fourier transform using approximately 15,000 lines of C++ code with MPI. The equivalent parallel C++ code is about 500 lines.

We used 4 of the cluster nodes to store the input array, dividing it into \( 128^3 \) pages of \( 128^3 \) numbers each.

We used the other 4 nodes to run 16 processes of the Fourier transform, 4 processes per node. An Array object (see Figure 9) is logically a pointer: it is a small object which is copied to all processes working on the array. Array pages are allocated on 96 hard drives, using virtual hosts for storing persistent objects, which are implemented in the Device class. We ran two Device agents on each CPU core, each Device using a single hard drive. In order to maximize the utilization of CPUs, network bandwidth and the total available disk throughput, array pages were allocated in circul-

The Fourier transform is computed by loading, and when necessary transposing, lines of 128 pages into 4GB RAM buffers, performing \( 128^2 \) one-dimensional transforms in each buffer, and writing the contents back to hard drives (see Figure 8). We illustrate the computation of the Fourier transform in the first dimension. The processes computing the Fourier transform in the first dimension are implemented in the SlabFFT1 class. Each of the 16 SlabFFT1 objects was assigned an array slab of \( 128 \times 8 \times 128 \) pages to transform it line by line in \( 8 \times 128 = 1024 \) iterations (see Figure 10). The SlabFFT1 objects are independent of each other (see Figure 11), but they compete for service from the 96 hard drives, and they share the network bandwidth. The SlabFFT1 process overlaps reading a page line with FFT1 function, which computes \( 128^2 \) 1D FFTs using the FFTW library \[10\]. The 16 SlabFFT1 processes use the ReadPageLine method in parallel, and each ReadPageLine call reads 128 pages in parallel from the hard drives storing the array, and copies them over the network into the RAM buffer of the SlabFFT1 object. Figure 12 shows two implementations of ReadPageLine, demonstrating a very easy way to shift computation among processors. We used the first implementation in our computation in order to offload some of the work from the SlabFFT1 processes.
A solution to the problem of parallel programming

class Array
{
    public:
        Array(Domain * ArrayDomain, Domain * PageDomain);
    ~Array();
    void allocate(int number_of_devices, Device * d);
    void FFT1(int number_of_cpus, Host ** cpus);
    private:
        Domain * ArrayDomain;
        Domain * PageDomain;
        ArrayPage * *** page; // 128^3 pointers
};

Figure 9: The Array class. Domain is a helper class describing 3D subdomains of an array. ArrayPage is a small 3D array, which implements local array operations, such as transpose12 and transpose13 methods. These operations are needed in the Fourier transform computation. Global array operations are implemented using the local methods of ArrayPage. Array pages are allocated in circulant order. The allocate method constructs N1 × N2 × N3 array pages of size n1 × n2 × n3 on a list of virtual devices. The dimensions are obtained from ArrayDomain and PageDomain, and in our case are all equal 128.

void Array::allocate(int number_of_devices, Device * d)
{
    for (int j1 = 0; j1 < N1; j1++)
        for (int j2 = 0; j2 < N2; j2++)
            for (int j3 = 0; j3 < N3; j3++)
            {
                int k = (j1 + j2 + j3) % number_of_devices;
                page[j1][j2][j3] =
                    new(d[k]) ArrayPage(n1, n2, n3);
            }
}

void Array::FFT1(int number_of_cpus, Host ** cpus)
{
    int slab_width = N2 / number_of_cpus;
    SlabFFT1 ** slab_fft =
        new SlabFFT1 *[number_of_cpus];
    for (int i = 0; i < number_of_cpus; i++)
        slab_fft[i] =
            new(cpus[i]) SlabFFT1(this, i, slab_width, (i + 1) * slab_width);
    for (int i = 0; i < n1; i++)
        slab_fft[i]->ComputeTransform();
}

void SlabFFT1::ComputeTransform()
{
    ReadPageLine(page_line, N20, 0);
    for (int i2 = N20; i2 < N21; i2++)
        for (int i3 = 0; i3 < N3; i3++)
        {
            if (L3 == N3)
                { L3 = 0; L2 ++; }
            if (L2 != N21)
                ReadPageLine(next_page_line, L2, L3);
            FFTW1(page_line);
            WritePageLine(page_line, i2, i3);
            page_line = next_page_line;
        }
}

Figure 10: Fourier transform of an array. SlabFFT1 objects are constructed (in parallel) on remote processors, each SlabFFT1 is assigned a slab of the array. The 16 SlabFFT1 objects compute the transforms in parallel.

Figure 11: Fourier transform of a slab. page_line and next_page_line are 2 local RAM buffers, 4 GB each. The iterations are sequential, and next_page_line is read while page_line is being transformed using the FFTW1 function, which computes 128^2 1D FFTs using the FFTW library.

The Fourier transform main (see Figure 13) creates 96 virtual devices for array storage, one on each hard drive of the 4 storage nodes. The array object is created and array pages are allocated. Next, 16 virtual hosts are created on the 4 computing nodes and the Fourier transform computation is performed using these virtual hosts.

5 Compilation and Runtime

5.1 Compiler architecture

The object-oriented framework of section 2.1 implicitly restricts network communications to implementation of object operations. Object operations can be described by an intermediate representation (IR) language. We devised a rudimentary IR for our compiler prototype (see section 5.2), where most of the IR instructions were generated by the compiler. Here are three examples of IR instructions: remote copy a block of memory, initiate a method execution on an object, notify an agent that a remote execution has completed. IR code can be translated by the compiler into instructions for the network hardware and for the CPUs. It can also be
A solution to the problem of parallel programming

```c
void SlabFFT1::ReadPageLine(ArrayPage * page_line, int i2, int i3)
{
    for (int i1 = 0; i1 < N1; i1++)
    {
        page[i1][i2][i3]->transpose13();
        page_line[i1] = *page[i1][i2][i3];
    }
}
```

```c
void SlabFFT1::ReadPageLine(ArrayPage * page_line, int i2, int i3)
{
    for (int i1 = 0; i1 < N1; i1++)
    {
        page_line[i1] = *page[i1][i2][i3];
        page_line[i1]->transpose13();
    }
}
```

Figure 12: Two possible implementations of ReadPageLine. In our computation we used the first implementation, where the transpose is performed “close to the data” by the agent storing the page. In the second implementation the transpose would be performed by SlabFFT1 after it reads the page.

```c
int main()
{
    int number_of_disks = 96;
    Device ** hdd = new Device *[number_of_disks];
    for (int i = 0; i < number_of_disks; i++)
        hdd[i] = new Device("hard drive i");

    int n = 128;
    Domain page_domain(n, n, n);
    int N = n * n;
    Domain array_domain(N, N, N);
    Array * a = new Array(array_domain, page_domain);
    a->allocate(number_of_disks, hdd);
    int number_of_cpus = 16;
    Host ** cpu = new Host *[number_of_cpus];
    for (int i = 0; i < number_of_cpus; i++)
        cpu[i] = new Host("address of cpu i");
    a->FFT1(number_of_cpus, cpu);
}
```

Figure 13: Fourier transform main constructs in parallel 96 virtual devices for array storage, one on each hard drive of the 4 storage nodes. The array object is created and array pages are allocated. The 16 virtual hosts are created on the 4 computing nodes after the page allocation has completed, and the Fourier transform computation starts after the construction of virtual hosts.

used for compile-time analysis of network utilization, as well as optimization of the system’s performance as a whole. It is therefore natural to implement a dedicated compiler backend for the interconnect fabric.

The interconnect hardware instruction set is not restricted to sending and receiving messages. In the Mellanox InfiniBand, for example, processing is done in network interface cards and network switches. The following two examples illustrate the potential advantages of compiler-generated networking instructions for this network.

Applications must use large messages to avoid the latency penalty and to utilize the network bandwidth. As a result, a lot of code (and some processing power) is devoted to packing and unpacking messages. The User-mode Memory Registration (UMR) feature of Mellanox InfiniBand can support MPI derived datatype communication, which may reduce some of this overhead [16], but it requires the programmer to duplicate datatype definitions, in order to inform the MPI library about the datatypes used in the program. In parallel C++ this information is available to the compiler, which can generate the UMR instructions.

Another example of in-network processing is the Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) of Mellanox InfiniBand [11] that off-loads the computation of collective operations, such as barrier and broadcast, to the switch network, eliminating the need to send data multiple times between endpoints. The SHARP hardware capabilities are currently accessed by the user only indirectly via a communications library, like MPI. However, the compiler is potentially capable of generating detailed and efficient routing and aggregation instructions for very complex code. Perhaps the simplest example is the following variant of the broadcast statement, where a large number of objects a[i] are located on some subset of the system’s processors:

```c
// a[i] are remote objects
for (int i = 0; i < N; i++)
    a[i] = b;
```

The last example suggests that the development of an optimizing compiler targeting network hardware may lead to improved network hardware design. In that respect, an especially important example of a compilation target is a many-core processor with a network-on-chip (NoC), such as the Tile processor [8]. Such processors can now be designed to optimally execute IR code.

Presently, computations on distributed systems use processes that exchange messages, which are arbitrary collections of bits. The programming framework provides the communication libraries with almost no meaningful information about the messages. The derived data types mechanism in MPI, mentioned in the example above, is merely an awkward attempt to extract a small amount of such information from the programmer. We have described a software architecture where
the application’s communications are an integral part of the computation, which is analyzed and mapped by the compiler onto the interconnect fabric. The network is not merely a collection of passive data pipes between processing nodes, but is a key component, which together with the processors, makes up a computer. Based on this architecture it is now possible to design an operating system, develop new hardware and build a multiprocessor computer (see section 6).

5.2 The prototype

We built a prototype compiler, called PCPP, and a runtime system (see Figure 14) for parallel C++. The compiler translates parallel C++ into C++ code, which is compiled and linked against the runtime library to obtain an executable.

5.2.1 The runtime library

The runtime library implements virtual hosts as agents that execute IR instructions. All messages between agents are serialized IR instructions, and for that purpose the runtime library contains a simple serialization layer. An agent is implemented as an MPI process with multiple threads: a dispatcher thread and a pool of worker threads. The dispatcher thread receives an incoming message, unserializes it into an IR instruction and assigns it to a worker thread for execution. Each worker thread maintains a job queue of IR instructions, however the pool of worker threads is not limited, and can grow dynamically. Every worker thread is either processing its job queue, is suspended and waiting to be resumed, or is idle and available to work. An execution of an IR instruction typically involves execution of the application’s code and may result in new IR instructions being sent over the network. We used one dedicated worker thread in every agent to serialize and send IR instructions to their destination agents.

We used a small number of basic MPI commands to implement a transport library for agents’ communications, and to launch agents on remote hosts as MPI processes. All of the MPI functionality used in the prototype is encapsulated in the transport library and can be easily replaced.

5.2.2 The PCPP compiler

PCPP is source-to-source translation tool which works with a subset of the C++ grammar. It is built using the Clang library tools. (Clang is the front end of the LLVM compiler infrastructure software.)

PCPP transforms the main of the input program into a stand-alone class, the application’s main class. It generates a new main program which initializes the runtime system, constructs a virtual host and constructs the application’s main object on it. Next, the new main reverses these actions, destroying the application’s main object on the virtual host, destroying the virtual host and shutting down the runtime system. PCPP translates all pointers to remote pointer objects. For every class of the application PCPP generates IR instructions for its object operations (constructors, destructors and methods). Additionally, PCPP replaces calls to object operations with code that serializes the parameters and sends them with the corresponding instruction to the destination agent. For example, when a constructor is invoked, one of the serialized parameters is a remote pointer containing the address of the result variable, which is a remote pointer variable that should be assigned with the result of the constructor. The PCPP-generated IR instruction is a serializable class, derived from the base instruction class defined in the runtime library. When this instruction is received by the destination agent, it is unserialized and its execute method is invoked. This method constructs a local object using the unserialized parameters and generates an IR instruction to copy the object pointer to the result variable on the source agent.

For causality enforcement we implemented a simple guard object, based on the condition_variable of the C++11 standard library. PCPP generates a guard object for every output variable of a remote operation. A wait method on the guard object suspends the executing thread until a release method is called on the same guard object by another thread. A remote pointer to this guard object is sent to the destination agent. When the destination agent completes the operation it sends an IR instruction to the source agent to release the guard. The wait call is inserted in the application code just before the value of the output variable is used.

We used PCPP to design the runtime system architecture and to run experiments with parallel C++. Substantial work needs to be done to produce a fully functional compiler. A full-fledged formal IR language has to be developed and a multiple of compiler back
ends implemented for various hardware architectures. Because the programming model makes no distinction between local and remote pointers, compiler optimization of the code is needed to make local computations efficient. It is likely that this problem can also be addressed with an appropriate hardware design. Causality enforcement requires implementation of control flow graph analysis and loop optimization, such as those implemented in parallelizing compilers. A conservative implementation would inhibit parallel computation and would alert the user whenever dependence analysis fails, allowing the user to change the code accordingly.

6 Conclusion: the road to a multiprocessor computer

We have defined a framework for object-oriented computing and have shown that object-oriented languages can be interpreted in this framework as parallel programming languages. Parallel C++, for example, is a very powerful language, which is accessible and natural for programmers who are proficient in standard C++. We believe that implementation and standardization of parallel C++ will drastically reduce the cost of parallel programming. Furthermore, we have shown that standard sequential C++ programs can be parallelized automatically, potentially sped up and ported to more energy efficient parallel computing hardware.

The object-oriented computing framework provides the foundation for a new computer architecture, which we call a multiprocessor computer. A parallel C++ compiler with a back end that generates code for the interconnect fabric is a first step towards developing an object-oriented operating system and designing new hardware architecture.

Many object-oriented operating systems have been built [6], but these projects used standard sequential object-oriented languages, and were not aimed at designing an operating system for a multiprocessor computer. Files and processes are the fundamental building blocks of operating systems used today. Both of these abstractions need to be replaced with the concept of an object; files should be replaced with persistent objects. Objects, virtual hosts, virtual devices and applications, defined in section 2.1, are some of the fundamental entities an object-oriented operating system needs to be based on. It is now possible to design an operating system that enables multiple applications to simultaneously share the network, the processors, and all of the system’s resources. This will significantly improve hardware utilization and reduce the energy cost of computations.

Processors with a large number of cores and a network on chip (NoC) are very energy efficient [9], but are very difficult to program [17]: the typical approach is to use a communication library with message passing processes. We propose to use lightweight cores, communicating asynchronously, and designed with hardware support for running virtual hosts. The NoC has very high bandwidth, but congestion control is still a problem. It has been shown that application awareness significantly improves congestion control in a NoC because better throttling decisions can be implemented [18]. In present systems such application awareness is very difficult to implement, but the object-oriented framework provides the concept of an application, and the operating system can be tasked with tracking application’s objects and throttling the application when appropriate.

Building a multiprocessor computer is an enormous task. It is not possible to discuss all of the important aspects of this project in a few pages, but for the first time in this paper we have described the key ideas that make it possible.
A solution to the problem of parallel programming

References

[1] graph500 benchmark. http://www.graph500.org/ Accessed: 2018-07-30.

[2] InfiniBand® Roadmap. http://www.infinibandta.org/content/pages.php?page=technology_overview Accessed: 2018-07-27.

[3] The Python Language Reference. https://docs.python.org/2/reference/index.html Accessed: 2018-11-04.

[4] Wikipedia entry: Actor model. https://en.wikipedia.org/wiki/Actor_model Accessed: 2018-11-04.

[5] Wikipedia entry: Object (computer science). https://en.wikipedia.org/wiki/Object_(computer_science) Accessed: 2018-11-04.

[6] Wikipedia entry: Object oriented operating system. https://en.wikipedia.org/wiki/Object-oriented_operating_system Accessed: 2018-11-14.

[7] Maria Averinou, Paolo Bertoldi, and Luca Castelazzi. Trends in data centre energy consumption under the european code of conduct for data centre energy efficiency. Energies, 10(10):1470, Sep 2017.

[8] B. Edwards, D. Wentzlaff, L. Bao, H. Hoffmann, C. Miao, C. Ramey, M. Mattina, P. Griffin, A. Agarwal, and J. F. Brown III. On-chip interconnection architecture of the tile processor. IEEE Micro, 27:15–31, 10 2007.

[9] Emilio Francesquini, Márcio Castro, Pedro H. Penna, Fabrice Dupros, Henrique C. Freitas, Philippe O.A. Navaux, and Jean-François Méhaut. On the energy efficiency and performance of irregular application executions on multicore, numa and manycore platforms. J. Parallel Distrib. Comput., 76(C):32–48, February 2015.

[10] Matteo Frigo and Steven G Johnson. The design and implementation of fftw3. Proceedings of the IEEE, 93(2):216–231, 2005.

[11] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievska, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim, and Eitan Zahavi. Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction. In Proceedings of the First Workshop on Optimization of Communication in HPC, COM-HPC '16, pages 1–10, Piscataway, NJ, USA, 2016. IEEE Press.

[12] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI’73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[13] Programming Language C++. [Working draft]. Standard, International Organization for Standardization (ISO), Geneva, Switzerland, (2014).

[14] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO ’04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Edward A. Lee. The problem with threads. Computer, 39(5):33–42, May 2006.

[16] M. Li, K. Hamidouche, X. Lu, J. Zhang, J. Lin, and D. K. Panda. High performance OpenSHMEM strided communication support with infiniband umr. In 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), pages 244–253, Dec 2015.

[17] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumin. Programming the intel 80-core network-on-a-chip terascale processor. In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC ’08, pages 38:1–38:11, Piscataway, NJ, USA, 2008. IEEE Press.

[18] George Nychis, Chris Fallin, Thomas Moscibroda, and Onur Mutlu. Next generation on-chip networks: What kind of congestion control do we need? In Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, Hotnets-IX, pages 12:1–12:6, New York, NY, USA, 2010. ACM.