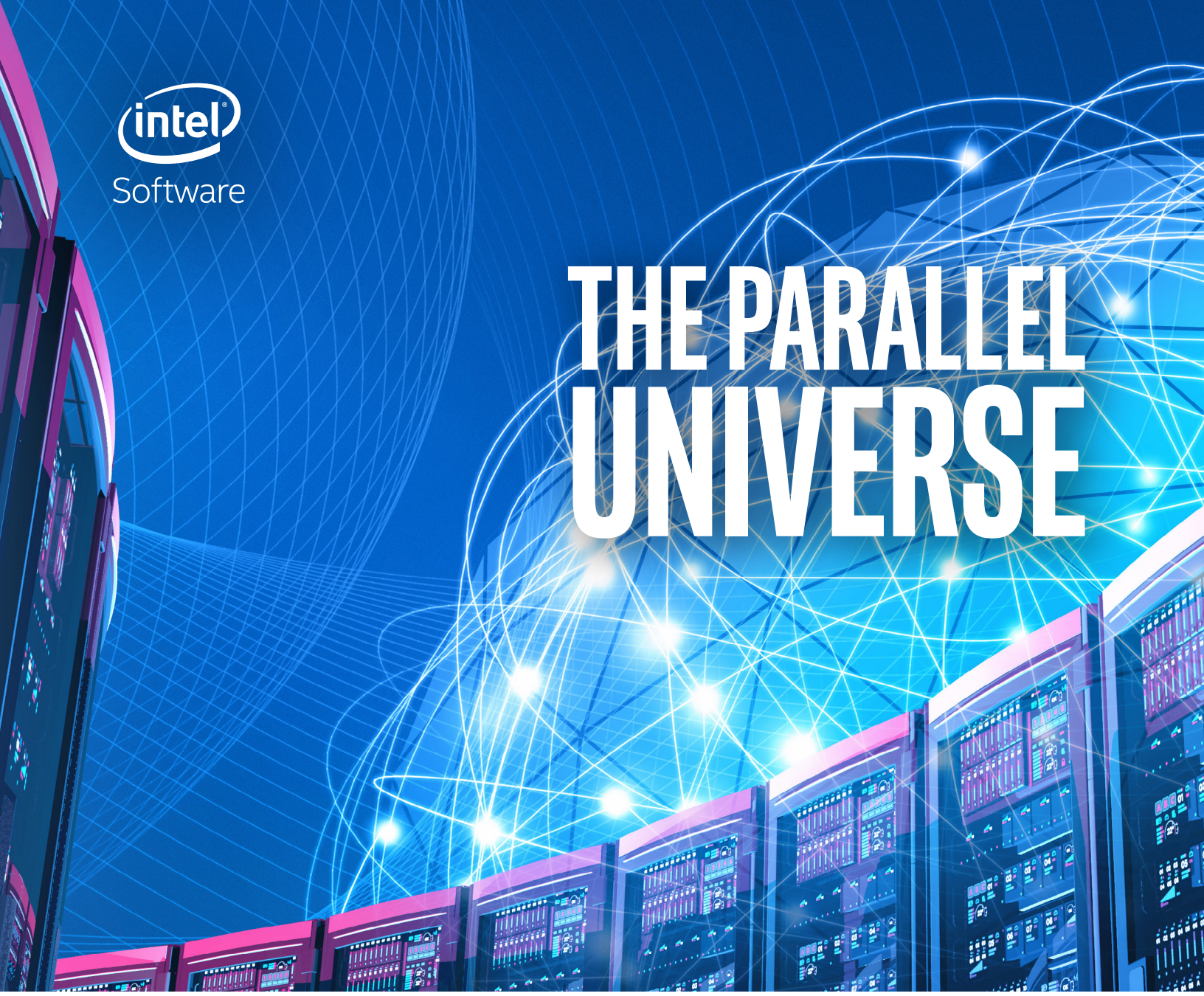


Transform Sequential C++ Code to Parallel with Parallel STL

High-Performance with R: Satisfying the Need for Speed

Solving Real-World Machine Learning Problems

Issue
28
2017

CONTENTS

LETTER FROM THE EDITOR

Henry A. Gabb, *Senior Principal Engineer* at Intel Corporation, is a long-time high-performance and parallel computing practitioner and has published numerous articles on parallel programming. He was editor/coauthor of "Developing Multithreaded Applications: A Platform Consistent Approach" and was program manager of the Intel/Microsoft Universal Parallel Computing Research Centers.

Parallel Languages, Language Extensions, and Application Frameworks

Back in the days of nonstandard programming languages and immature compilers, parallel computing as we know it today was still far over the horizon. It was still a niche topic, so practitioners were content with language extensions and libraries to express parallelism (e.g., OpenMP*, Intel® Threading Building Blocks, MPI*, pthreads*). Programming language design and parallel programming models were separate problems, so they continued along distinct research tracks for many years. These tracks would occasionally cross with varying degrees of success (e.g., High-Performance Fortran*, Unified Parallel C*), and there were frequent debates about whether the memory models of popular languages even allowed parallelism to be implemented safely. However, much was learned during this time of debate and experimentation.

Today, parallel computing is so ubiquitous that we're beginning to see parallelism become a standard part of mainstream programming languages. This issue's feature article, **Parallel STL: Boosting Performance of C++ STL Code**, gives an overview of the Parallel Standard Template Library in the upcoming C++ standard (C++17) and provides code samples illustrating its use.

Though it's not a parallel language in and of itself, we're still celebrating 20 years of OpenMP, the gold standard for portable, vendor-neutral parallel programming directives. In the last issue of **The Parallel Universe**, Michael Klemm (the current CEO of the OpenMP Architecture Review Board) gave an overview of the newest OpenMP features. In this issue, industry insider Rob Farber gives a retrospective look at OpenMP's development and its modern usage in **Happy 20th Birthday, OpenMP**.

I rely on R for certain tasks but I won't lie to you, it's not my favorite programming language. I would never have thought to use R for high-performance computing (HPC) but Drew Schmidt from the University of Tennessee Knoxville makes the case for using this popular statistics language in **HPC with R: The Basics**. Drew's article is helping to make an R believer out of me.

New Software for Machine Learning

There's no denying that machine learning, and its perhaps-more-glamorous nephew, deep learning, are consuming a lot of computing cycles these days. Intel continues to add solutions to its already robust machine learning portfolio. The latest offering, **BigDL**, is designed to facilitate deep learning within big data environments. **BigDL: Optimized Deep Learning on Apache Spark*** will help you get started using this new framework. **Solving Real-World Machine Learning Problems with the Intel® Data Analytics Acceleration Library** walks through classification and clustering using this library. Two problems taken from the **Kaggle** predictive modeling and analytics platform are used to illustrate, and comparisons to Python* and R alternatives are shown.

Coming Attractions

Future issues of *The Parallel Universe* will contain articles on a wide range of topics. Stay tuned for articles on the Julia* programming language, working with containers in HPC, fast data compression for cloud and IoT applications, Intel® Cluster Checker, and much more.

Henry A. Gabb
April 2017

PARALLEL STL: BOOSTING PERFORMANCE OF C++ STL CODE

C++ and the Evolution Toward Parallelism

Vladimir Polin, *Application Engineer,* and Mikhail Dvorskiy, *Senior Software Development Engineer,* Intel Corporation

Computing systems have evolved rapidly from single-threaded SISD architectures to modern multi- and many-core SIMD architectures, which are used in various fields and form factors. C++ is a general-purpose, performance-oriented language widely used on these modern systems. However, until recently, it didn't provide any standardized instruments to fully utilize these modern systems. Even the latest version of C++ has limited features to extract parallelism. Over time, vendors invented a variety of specifications, techniques, and software to support parallelism[1] (**Figure 1**). The upcoming version of the C++ standard (C++17) introduces Parallel STL, which makes it possible to transform existing, sequential C++ code to parallel in order to take advantage of hardware capabilities like threading and vectorization.

	C++11/14	C++17	C++2X	Other approaches
High Level (Message, Event Flow Graph)	std::async, std::future	std::async, std::future	resumable functions	MPI*, Microsoft AAL* Intel TBB flow graph Qualcomm Symphony*, etc.
Middle Level (fork-join, threads)	std::thread + manual sync	Parallel STL (par)	Task Block for_loop	OpenMP*, Intel TBB, Cilk*, Nvidia Thrust, Microsoft PPL*, Qualcomm Symphony*, OpenCL*, OpenACC*, etc.
Innermost Level (SIMD)	No	Parallel STL (par_unseq)	Parallel STL (unseq, vec) for_loop	Intrinsics, auto-vectorization, Intel Cilk™ Plus, OpenCL*, Nvidia CUDA*, OpenMP* 4, OpenACC*, etc.

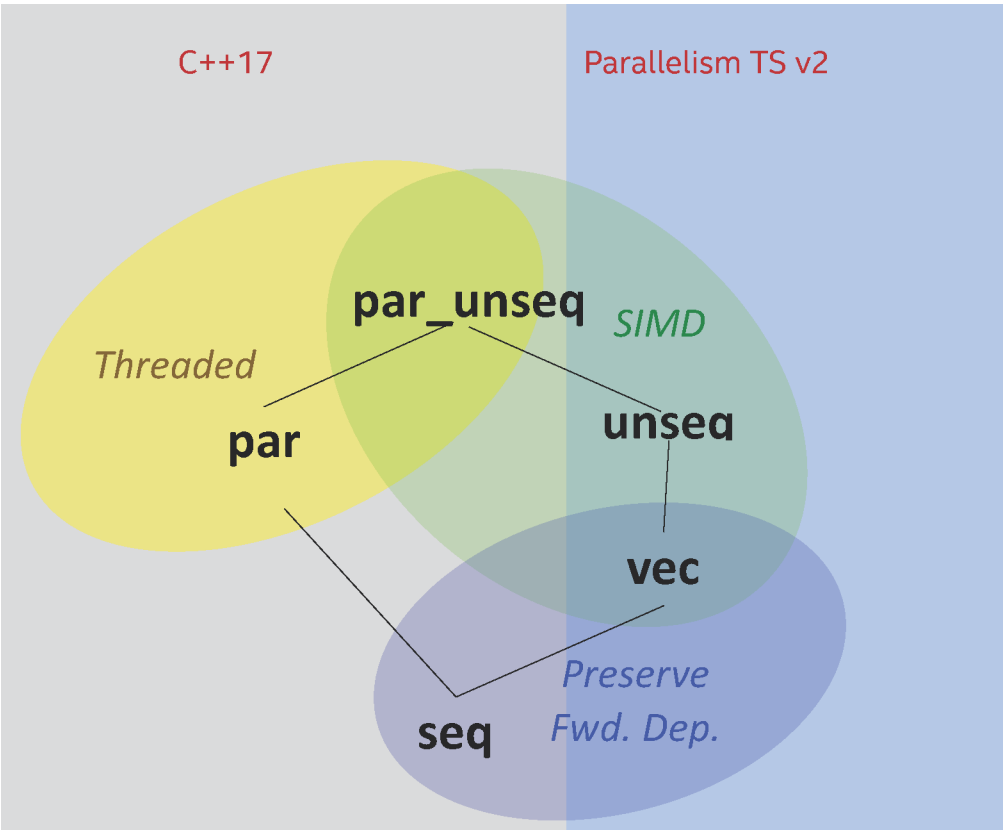
1 Landscape of parallelism in C++

Enter Parallel STL

Parallel STL extends the C++ Standard Template Library with the execution policy argument. An execution policy is a C++ class used as a unique type to disambiguate function overloading for STL algorithms. For convenience, the C++ library also defines an object of each class that can be used as the policy argument. Policies can be used with well-known algorithms (e.g., transform, for_each, copy_if), as well as new algorithms (e.g., reduce, transform_reduce, variations of scan [prefix sum]). Support for parallel execution policies was developed over several years as the Technical Specification for C++ Extensions for Parallelism (Parallelism TS). Now it's been adopted as the standard and included in the current C++17 standard draft (document n4640[2]). Support for vectorization policies has been proposed for the second version of the Parallelism TS (documents p00753[3] and p00764[4]). Overall, these documents describe five different execution policies (**Figure 2**):

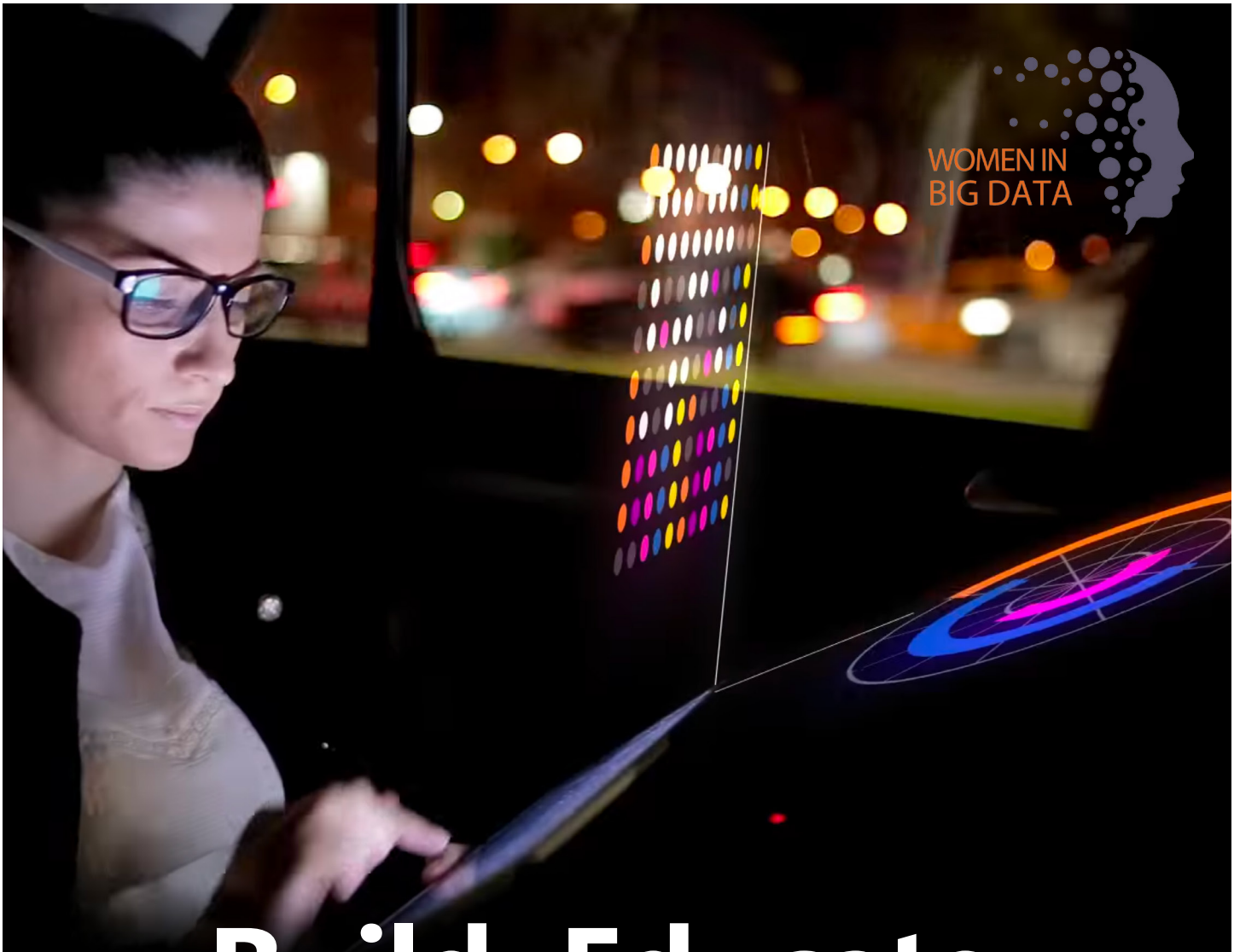
- The class `sequenced_policy (seq)` requires that an algorithm's execution may not be `parallelized`.[2]
- The class `parallel_policy (par)` indicates that an algorithm's execution may be parallelized.[2] Any user-specified functions invoked during the execution should not contain data races.
- The class `parallel_unsequenced_policy (par_unseq)` suggests that execution may be parallelized and vectorized.[2]

- The class `unsequenced_policy (unseq)` is a proposal in Parallelism TS v2[4] of an execution policy to indicate that an algorithm's execution may be vectorized but not parallelized. This policy requires that all functions provided are SIMD safe.
- The `class vector_policy (vec)` is a proposal[4] of an execution policy type to indicate that an execution may be vectorized in a way that preserves forward dependency between elements.



2 Execution policies for the C++ Standard Template Library

Figure 2 shows relations between these execution policies. The higher a policy is in the lattice, the more execution freedom it allows—but also, the more requirements it puts on the user code. An implementation of Parallel STL is allowed to substitute an execution policy with a more restrictive one that is lower in the lattice.



Build, Educate, and Inspire

Women in Big Data is a grassroots community championing women's leadership and success in big data/analytics. Women in Big Data seeks to strengthen diversity in this field by encouraging and attracting more female talent and helping them connect, engage and grow.

Be part of the solution.

Become a member ›

Follow us on Twitter **@DataWomen**

Simplified equivalents of the STL and Parallel STL algorithms can be written as follows:

```
#include <execution>
#include <algorithm>

void increment_seq( float *in, float *out, int N ) {
    using namespace std;
    transform( in, in + N, out, []( float f ) {
        return f+1;
    });
}

void increment_unseq( float *in, float *out, int N ) {
    using namespace std;
    using namespace std::execution;
    transform( unseq, in, in + N, out, []( float f ) {
        return f+1;
    });
}

void increment_par( float *in, float *out, int N ) {
    using namespace std;
    using namespace std::execution;
    transform( par, in, in + N, out, []( float f ) {
        return f+1;
    });
}
```

Where

```
std::transform( in, in + N, out, foo );
```

would be as simple as the following loop

```
for (x = in; x < in+N; ++x) *(out+(x-in)) = foo(x);
```

and

```
std::transform( unseq, in, in + N, out, foo );
```

would be as simple as the following loop (our implementation uses `#pragma` omp simd on the innermost level; other Parallel STL implementations might use different approaches to implement `unseq` policy)

```
#pragma omp simd
for (x = in; x < in+N; ++x) *(out+(x-in)) = foo(x);
```

and

```
std::transform( par, in, in + N, out);
```

would be as simple as the following parallel loop

```
tbb::parallel_for (in, in+N, [=] (x) {
 *(out+(x-in)) = foo(x);
});
```

Overview of Parallel STL Implementation in Intel® Parallel Studio XE 2018 Beta

The Parallel STL implementation is a part of **Intel® Parallel Studio XE 2018 Beta**. It offers efficient support for both parallel and vectorized execution of algorithms on Intel® processors. Under the hood, it uses an available implementation of the C++ standard library for sequential execution, **Intel® Threading Building Blocks** (Intel® TBB) for parallelism with par and par_unseq execution policies, and OpenMP* vectorization for unseq and par_unseq policies.

The Parallel STL implementation in Intel Parallel Studio XE 2018 beta is prerelease code, which may not be fully functional and which Intel may substantially modify in future versions.

After installing Parallel STL, you need to set up the environment following the instructions in the “Getting Started with Parallel STL” document provided in the package. The document also contains the up-to-date list of algorithms that have parallel and vector implementations. For all other algorithms, execution policies are accepted but fall back to sequential implementation. We plan to enable parallelism in more algorithms in future releases based on feedback and demand.

To achieve best results with Parallel STL, we recommend using Intel® C++ Compiler 2018. Other compilers can also be used, provided they support C++11. For vectorization policies to be effective, the compiler should also support OpenMP 4.0 SIMD constructs. Use of parallel execution policies requires Intel TBB.

Follow these steps to add Parallel STL to your application:

1. Add `#include "pstl/execution"` line to your code. Then add one or more of the following lines, depending on the algorithms you intend to use:
   a. `#include "pstl/algorithm"`
   b. `#include "pstl/numeric"`
   c. `#include "pstl/memory"`

Note that `"pstl"` should be used as part of the header name. This is done intentionally to avoid conflicts with the C++ standard library header files.

2. When using algorithms and execution policies, specify the namespaces `std` and `std::execution`, respectively.
3. Compile the code as C++11 or later. Use a proper compiler option to enable OpenMP vectorization; e.g., for the Intel® C++ Compiler, use `-qopenmp-simd` (`/Qopenmp-simd` for Windows*).
4. To get good performance, specify the target platform. For the Intel C++ Compiler, some of the relevant options are `-xHOST`, `-xCORE-AVX2`, `-xMIC-AVX512` for Linux* or `/QxHOST`, `/QxCOREAVX2`, `/QxMIC-AVX512` for Windows.
5. Link with Intel TBB, if required. On Windows, it is done automatically; on other platforms, add `-ltbb` to the linker options.

Intel Parallel Studio XE 2018 beta contains the gamma correction example that can be used to try Parallel STL.

Efficient Vectorization, Parallelization, and Composability using Parallel STL

In theory, Parallel STL was invented as a highly intuitive way for C++ developers to program a parallel random-access machine (PRAM). Let's consider several ways the theory correlates with the best practices of parallelizing loop hierarchies, such as the "vectorize innermost, parallelize outermost" (VIPO) approach.[5] (Take into account that we are using prerelease software, so results can be different in further versions of the software.) Consider the classic example of image gamma correction (or simply gamma)—a nonlinear operation used to encode and decode the luminance of each image pixel. Note that we have to disable compiler auto-vectorization for sequential cases to show the difference between sequential and unsequenced execution. (Otherwise, this difference can be seen only with compilers that do not support automatic vectorization but do support OpenMP-based vectorization.)

Here is a simple serial implementation:

```
#include <algorithm>
void ApplyGamma(Image& rows, float g) {
    using namespace std;
    for_each(rows.begin(), rows.end(), [g](Row &r) {
        transform(r.cbegin(), r.cend(), r.begin(),
            [g](float v) {
            return pow(v, g);
        });
    });
}
```

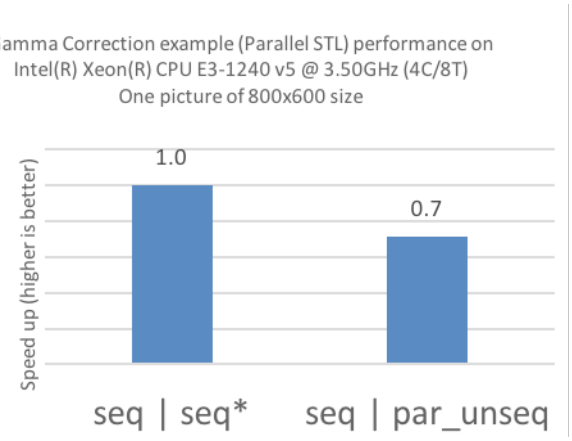
The function `ApplyGamma` gets an image passed by reference as a set of rows and uses `std::for_each` to iterate over them. The lambda function called for each row iterates over the pixels in the row with `std::transform` to modify the luminance of each pixel.

As described previously, Parallel STL provides parallelized and vectorized versions of the `for_each` and `transform` algorithms enabled with the execution policies. In other words, an execution policy passed as the first argument into a standard algorithm leads to execution of the parallelized and/or vectorized version of the algorithm.

Returning to the example above, you may notice that all calculations are in the lambda function called from the `transform` algorithm. So, let's try killing two birds with one stone and rewrite the example using the `par_unseq` policy as follows:

```
void ApplyGamma(Image& rows, float g) {

    using namespace std::execution;
    std::for_each(rows.begin(),rows.end(), [g](Row &r) {
        // Inner parallelization and vectorization
        std::transform(par_unseq, r.cbegin(), r.cend(), r.begin(),
            [g](float v) {
            return pow(v, g);
        });
    });
}
```



3 Inner parallelization and vectorization

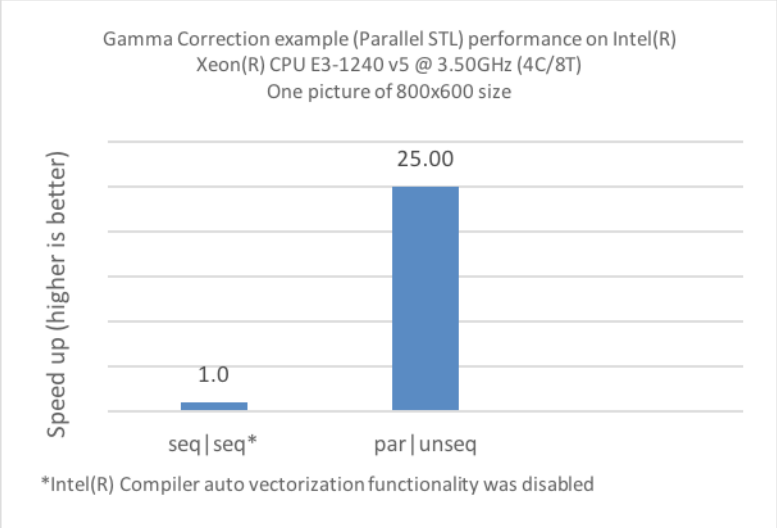
Surprisingly, the miracle has not happened (**Figure 3**). Performance with `par_unseq` is worse than serial execution. So this is a great example of how *not* to use Parallel STL. If you profile the code with tools like **Intel® VTune™ Amplifier XE**, you might see a lot of cache misses caused by threads on different cores writing to the same cache lines, as well as high thread synchronization and scheduling overhead. [Editor's note: This is known as false sharing. For more information, see "**Avoiding and Identifying False Sharing Among Threads**."]

As we noticed before, Parallel STL helps us to express the middle and innermost level parallel patterns. The middle level means parallelization with system threads; the innermost level means vectorization with SIMD. In general, to get the best speedup, estimate execution time for an algorithm and compare it with parallelization and vectorization overheads. We recommend that serial execution time be at least two times more than the overheads on each parallelization level. Additionally:

- Parallelize at the outermost level; seek the maximum amount of work to execute in parallel.
- If that provides sufficient parallelism, stop. Don't oversubscribe. Otherwise, parallelize an additional inner level.
- Make sure the algorithm is cache efficient.
- Try to vectorize the innermost level. Ensure minimal control flow divergence and memory access uniformity.
- Find more recommendations.[5]

These recommendations suggest that specifying the parallel and vector policies on different levels may provide better performance, i.e.,

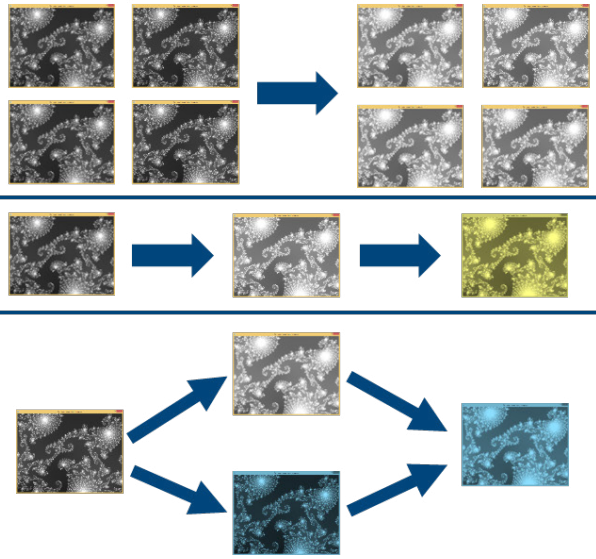
```
void ApplyGamma(Image& rows, float g) {
    using namespace std::execution;
    // Outer parallelization
    std::for_each(par, rows.begin(), rows.end(), [g](Row &r) {
        // Inner vectorization
        std::transform(unseq, r.cbegin(), r.cend(), r.begin(),
            [g](float v) {
            return pow(v, g);
        });
    });
}
```



4 Outer parallelization and inner vectorization

We now have efficient parallel processing for one image **(Figure 4)**, but real applications typically process multiple images or apply several corrections at once **(Figure 5)**. The parallelism at that higher level might not work well using the standard algorithms. In this case, we can use Intel TBB with Parallel STL in a composable way.

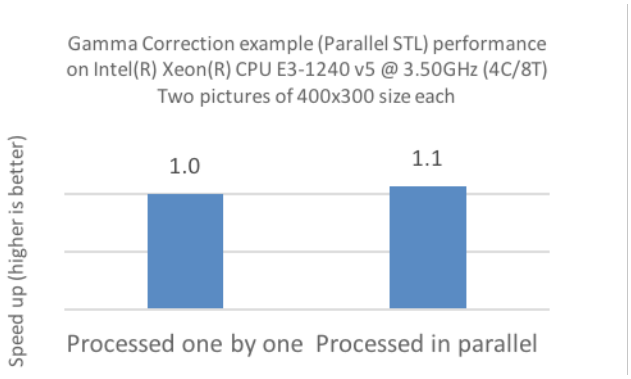
- The application can process multiple pictures simultaneously
  - Batch
  - Pipeline
  - Graph



5 Composability cases

Composability means you can express the topmost parallelism with Intel TBB parallel constructs (e.g., flow graph, pipeline) or tasks, and call Parallel STL algorithms at inner levels—all without worrying about oversubscribing your system, i.e.,

```
void Function() {
    Image img1, img2;
    // Prepare img1 and img2
    tbb::parallel_invoke(
        [&img1] { img1.ApplyGamma(gamma1); },
        [&img2] { img2.ApplyGamma(gamma2); }
    );
}
```



6 Composability between Intel® TBB and Parallel STL

As shown in **Figure 6**, processing two images simultaneously with Intel TBB doesn't reduce the performance—and even increases it a bit. This indicates that the expressing of inner and innermost parallelism fully utilizes the CPU cores.

Now let's consider the situation where we have a larger set of images to process and more CPU cores available.

```
tbb::parallel_for(images.begin(), images.end(),
[](image* img) {applyGamma(img->rows(), 1.1);}
);
```

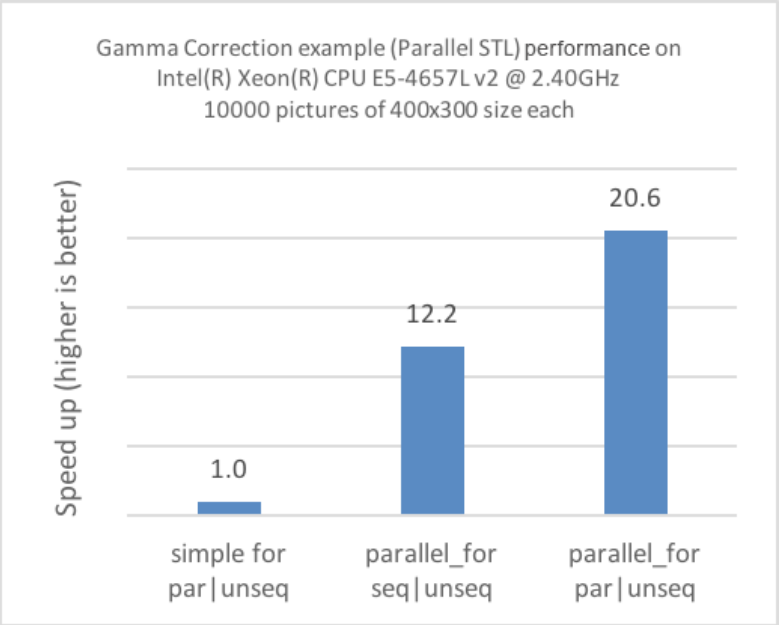
BLOG HIGHLIGHTS

Supercomputing 2016 HPC Server Demos

BY **MIKE P. (INTEL)** >

SC16 brought into sharp focus the powerful impact HPC is having on everything from life sciences and research to machine learning. Software-defined infrastructure, visualization, analytics, simulation, and DNA sequencing were just some of the use cases demonstrated in the Intel Corporation booth this year in the HPC Life Sciences Experience. Read the Demo Guide and follow the videos through a journey explaining how Intel HPC technologies, encapsulated in the Intel® Scalable System Framework (Intel® SSF), are helping to fuel a revolution in cancer cures with innovations that break down barriers in supercomputing!

Read more >



7 Composability between Intel® TBB and Parallel STL

As shown in **Figure 7**, processing a set of images simultaneously with Intel TBB (`parallel_for`) drastically increases speedup. Indeed, have a look at the first bar where we iterate sequentially through the images, and each image is processed by the inner and innermost parallelism patterns. Adding just the topmost level of parallelism (`parallel_for`) without the inner parallelism (`par`) significantly improves performance, but this is not enough to fully utilize all of the CPU core. The third bar shows that expressing all levels of parallelism increases the performance drastically. This illustrates the great composability between Intel TBB and our Parallel STL implementation.

Summary

Parallel STL is a significant step in the evolution of C++ toward parallel execution, making it easily applied to STL algorithms during code modernization and the development of new applications. It adds to C++ vectorization and parallelization capabilities without resorting to nonstandard or proprietary extensions, and its execution policies provide control over the use of these capabilities while abstracting hardware details. Parallel STL lets developers focus on expressing the parallelism in their applications without worrying about the low-level details of managing that parallelism. In addition to efficient, high-performing implementations of the most commonly used high-level parallel algorithms, the Parallel STL implementation in Intel Parallel Studio XE 2018 showcases the great composability with Intel Threading Building Blocks parallel patterns. However, Parallel STL is not a silver bullet. It must be used wisely. To achieve great performance, follow best-known practices when modernizing your code.[5]

Learn More

Here's where to find recent Parallel STL and Intel TBB versions and additional information:

- **Intel TBB official site**
- **Intel TBB open source site**
- **Intel TBB documentation**

Your feedback is important and there are several ways to provide it:

- **Intel TBB Forum**
- **Intel TBB developers' mail group**

References

1. Alex Katranov, Mikhail Dvorskiy, Parallelizing C++ Standard Algorithms for Intel® Processors, Intel Software Development Conference, London 2016.

2. ISO/IEC N4640, working draft, Standard for Programming Language C++.

3. P0075r1, **Template Library for Parallel For Loops**, Programming Language C++ (WG21).

4. P0076r3, **Vector and Wavefront Policies**, Programming Language C++ (WG21).

5. Robert Geva, **Code Modernization Best Practices: Multi-level Parallelism for Intel® Xeon® and Intel® Xeon Phi™ Processors**, IDF15 – Webcast.

BLOG HIGHLIGHTS

Just Released! Intel® Media SDK for Embedded Linux*

BY JEFFERY M. (INTEL) >

Smart camera and connected car developers can now deliver fast and efficient video processing in media applications with the Intel® Media SDK for Embedded Linux*. This opens the door for optimizing video streaming to and from smart camera usages across a host of devices: drones, phones, robotics, videocams, cars, players and editors, and more.

Just released, the Intel® Media SDK for Embedded Linux supports the latest Intel Atom®, Pentium®, and Celeron® processors as well as Yocto Project* (a comprehensive embedded Linux development environment). Together, this SDK along with Intel® processor-based platforms bring real-time computing in digital surveillance and new in-vehicle experiences.

Read more ›

HAPPY 20TH BIRTHDAY, OPENMP*

Making Parallel Programming Accessible to C/C++ and Fortran Programmers—and Providing a Software Path to Exascale Computation

Rob Farber, *Global Technology Consultant*, TechEnablement

In October 1997, the **OpenMP* Architecture Review Board (ARB)** published the v1.0 version of the OpenMP Fortran specification, with the C/C++ specification following nearly a year later. At that time, the fastest supercomputer in the world, **ASCI Red**, was based on computational nodes containing two 200 MHz Intel® Pentium® Pro processors. (Yes, leadership-class supercomputing at that time considered a single-core 200 MHz Intel Pentium processor to be fast.) Built at a cost of USD 46 million (roughly USD 68 million in today's dollars), ASCI Red was the first supercomputer to deliver a trillion floating-point operations per second on the TOP500 LINPACK benchmark. It was also the first supercomputer to consume a megawatt of power—foretelling a trend to come. In contrast, a modern dual-socket Intel® Xeon® processor v4 family is positively a steal, in terms of both teraflop computing capability and power consumption.

Dr. Thomas Sterling, director of the **Center for Research in Extreme Scale Technologies**, observes, "OpenMP has provided a vision of single-system programming and execution that emphasizes simplicity and uniformity. It challenges producers of system software to address

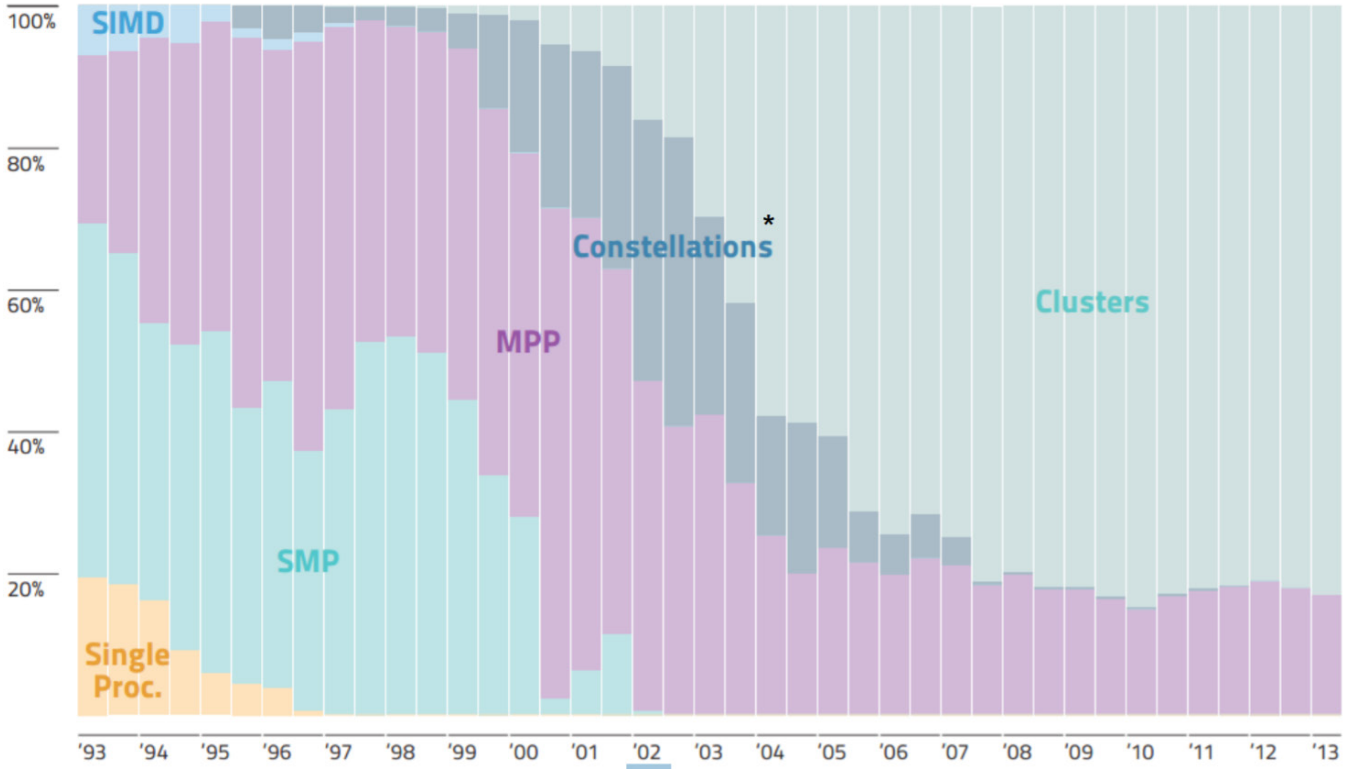
asynchrony, latency, and overhead of control while encouraging future hardware system designers to achieve user productivity and performance portability in the era of exascale. The last two decades have seen remarkable accomplishment that will lead the next 20 years of scalable computing."

OpenMP: A Forward-Thinking, Developer-Motivated Effort

The OpenMP initiative was motivated by the developer community. There was increasing interest during that time for a standard that programmers could reliably use to move code between different parallel, shared-memory platforms.

Before OpenMP, programmers had to explicitly use a threading model such as pthreads, or a distributed framework such as MPI, to create parallel codes. (The first MPI standard was completed in 1994.) The convenience of simply adding an OpenMP pragma to exploit parallelism in a shared-memory model was revolutionary in its convenience. But, at that time, thread-based computing models were of limited interest, since clusters of single-threaded processors dominated the high-performance computing world. It was possible on some hardware platforms to purchase extra plug-in CPUs that could provide hardware-based multithreaded performance. But, generally, threads were considered more of a software trick to emulate asynchronous behavior using OS time slices rather than a route to scalable parallel performance. At that time, the thread debate centered more on the use of heavyweight threads (e.g., processes created with fork/join) rather than lightweight threads that shared memory. Hardware parallelism inside a node was limited to dual- or quad-core processor systems, so OpenMP scaling was a nonissue.

Thus, the 1997 OpenMP specification was very forward thinking, since distributed-memory MPI computing was "the" route to parallelism. Basically, it was cheaper and easier to connect lots of machines via a network. In a world where **Dennard scaling** laws applied, faster application performance could be achieved by either adding MPI nodes or purchasing machines containing a higher–clock rate processor that could run serial software faster. Thus, the big advances around that time came from using commodity off-the-shelf (COTS) hardware to build clusters, which dominated the parallel computing world (**Figure 1**). For example, the original 1998 Beowulf how-to explains that, "Beowulf is a technology of clustering computers to form a parallel, virtual supercomputer," which "behaves more like a single machine rather than many workstations." There really was no mass scientific or commercial demand for multicore processors—hence, multithreaded parallel computing was more a very interesting HPC project than a mainstream programming model. The brief, massively parallel single instruction, multiple data (SIMD) interlude shown in **Figure 1** was short-lived and basically disappeared with the demise of Thinking Machines Corporation, the company that manufactured the SIMD architecture CM-2 supercomputer and later the CM-5 MIMD (multiple instruction, multiple data) massively parallel processor (MPP) supercomputer. The SGI Challenge is an example of an SMP (shared-memory multiprocessor in this context) from that era.

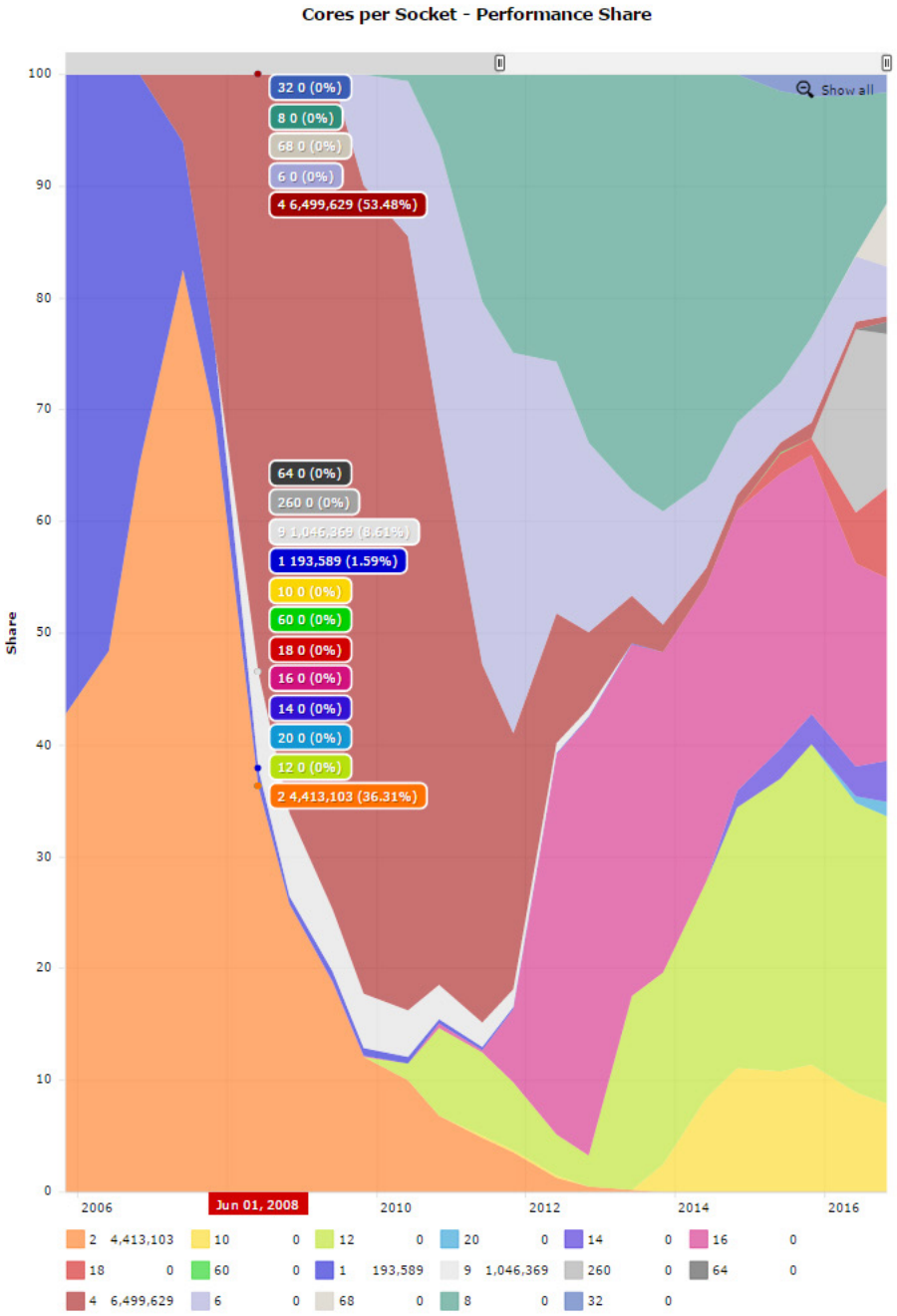


1 Percentage of system architectures per year in the TOP 500 (source: **top500.org**) (Constellations are clusters of large SMP systems)

OpenMP Moves into the Spotlight: Dennard Scaling Breaks and the Rise of Multicore

Between 2005 and 2007, it became clear that Dennard scaling had broken down and we started to see the first modern multicore processors. Since it was no longer possible to achieve significant performance increases by boosting the clock rate, manufacturers had to start adding processor cores to generate significant performance increases (and a reason to upgrade). This broke the comfortable status quo where codes would automatically run faster on the next generation of hardware due to clock rate increases. As a result, people started to seriously investigate using thread-based computing as a means to increase application performance. Even so, most applications exploited parallelism by simply running one serial MPI rank per core on the multiprocessor.

In the 2007 to 2008 timeframe, multicore processors began to dominate the performance landscape as illustrated by **Figure 2**, a performance share graph from the TOP500 organization. You can clearly see that the trend since then has been toward increasing core counts.



2 Trend toward increasing core counts to get performance in the TOP500 (source: **top500.org**)

Code Modernization with OpenMP

Increasing core counts benefited both OpenMP and MPI programs through greater parallelism. But the phoenix-like rise of vector parallelism, coupled with higher-core-count processors, has really turned OpenMP into a first-class citizen.

Many legacy applications utilized the one MPI rank per processor core because parallelism was the path to performance on COTS hardware when they were written. This is not to say that vectorization was not utilized—especially in HPC codes—but rather to highlight that small vector widths in the processors used for COTS clusters bounded the performance benefits. Also, programming the vector units was difficult. As a result, many programmers continued to rely on increased MPI parallelism to achieve higher application performance. Any benefits of vectorized loops in the code that ran inside each MPI rank were a nice additional benefit.

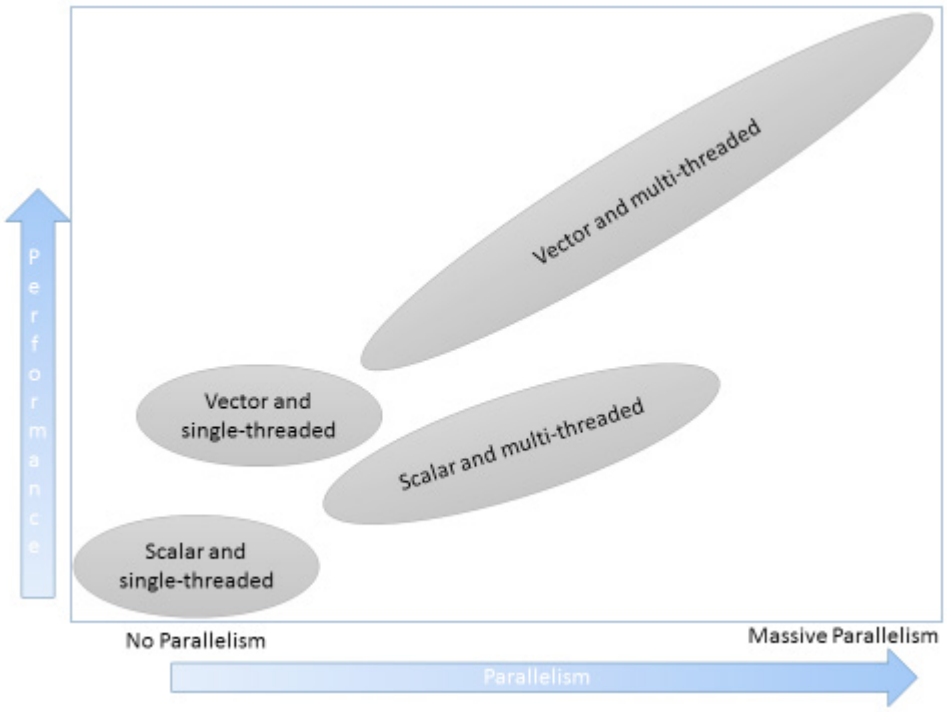
A resurgence in SIMD and data parallel programming, starting around 2006, showed that rewriting legacy codes to exploit hardware thread parallelism could deliver significant performance increases across a wide variety of applications and computational domains.

This trend accelerated as it was realized that power efficiency was a key stumbling block on the road to petascale—and eventually exascale—computers. The **Green500** list debuted in 2007, marking the end of the "performance at any cost" era in large-scale computing.

OpenMP was suddenly well positioned to exploit the focus on energy-efficient computing and data parallelism. Succinctly, CPUs are general-purpose MIMD devices that can run SIMD codes efficiently. Even better, SIMD codes map very nicely onto hardware vector units. Meanwhile, MIMD-based task parallelism was simply a loop construct away.

To increase both performance and power efficiency, ever-wider vector instructions have been added to the x86 ISA (instruction set architecture). Similar efforts are underway for other ISAs. Succinctly, hardware vector units consume relatively small amounts of space on the silicon of the chip, yet they can deliver very power-efficient floating-point performance. As a result, the floating-point capability of general-purpose processors increased dramatically and the era of high-core-count (or many-core) vector parallel processors was born. Examples include the many-core Intel Xeon and Intel® Xeon Phi™ processors.

Code modernization became a buzzword as people realized that programming one MPI rank per core was an inefficient model because it didn't fully exploit the performance benefits of SIMD, data parallel, and vector programming. Making efficient use of the AVX-512 vector instructions on the latest generation of hardware, for example, can increase application performance by 8x for double-precision codes and by 16x for single-precision codes. Many programming projects have switched, or are in the process of switching, to a combined OpenMP/MPI hybrid model to fully exploit the benefits of both MPI and OpenMP. The resulting performance increase can be the product of the number of cores and vector performance as shown in **Figure 3**. In fact, the latest Intel Xeon Phi processors have two AVX-512 vector units per core.



3 The highest performance is in the top right quadrant where programmers exploit both vector and parallel hardware (Image courtesy of Morgan Kaufmann, imprint of Elsevier, all rights reserved)

OpenMP: State of the Art

The OpenMP standard recognized the importance of SIMD programming and the SIMD clause was added to the OpenMP 4.0 standard in October of 2013. Additional clauses were added to the OpenMP 4.0 specification so that the offload mode programming of coprocessors and accelerators like GPUs is also now supported. OpenMP continues to grow and adapt to the changing hardware landscape.

OpenMP for the Exascale Era

As we look to an exascale future, power consumption is king. The trend for exascale computing architectures is to link power-efficient serial cores with parallel hardware—essentially, a hardware instantiation of **Amdahl's Law**. NERSC notes that the latest Cori supercomputer represents the first time users will run on a leadership-class supercomputer where their programs will run slower if they don't do anything to the code. Such is the inescapable consequence of increased power efficiency, since power-efficient serial cores for exascale supercomputers simply require more time to run sequential code. This trend will likely spill over to the data center, where power consumption is crucial to the bottom line and profits, yet it is expected that 5G will increase data volumes by up to 1,000x (source: *Forbes*).

Happily, OpenMP is now a tried-and-true veteran that gives us performance while still meeting the original design goal of a standard that programmers can reliably use to move code between different parallel, shared-memory platforms. Performance plus portability: what a lovely combination.

Rob Farber is a global technology consultant and author with an extensive background in HPC. He is an active advocate for portable parallel performant programming.
Reach him at **info@techenablement.com**.

SOLVING REAL-WORLD MACHINE LEARNING PROBLEMS WITH INTEL® DATA ANALYTICS ACCELERATION LIBRARY

Oleg Kremnyov, *Technical Intern;* Ivan Kuzmin, *Software Engineering Manager;* and Gennady Fedorov, *Software Technical Consulting Engineer;* Intel Corporation

Machine learning plays an important and growing role in the fields of statistics, data mining, and artificial intelligence. With the rapid growth of data, there are good reasons to believe that learning from data will become even more pervasive—and a necessary ingredient for future business growth. At the same time, choosing the right algorithms and libraries to solve a given problem depends on many factors, including:

- Class of problem (e.g., classification, regression)
- Input data
- Required performance
- Prediction accuracy
- Model interpretability

This creates barriers for the wider adoption of machine learning, which requires varied skill sets.

In this article, we will talk about criteria you can use to select correct algorithms based on two real-world machine learning problems that were taken from the well-known **Kaggle** platform used for predictive modeling and from analytics competitions where data miners compete to produce the best models. We'll use libraries that implement the algorithms from:

- **Scikit-learn***, the most popular library among Python* data scientists,
- **R**, the language of data analytics and statistical computing, and
- **Intel® Data Analytics Acceleration Library (Intel® DAAL)**, a performance library that provides optimized building blocks for data analysis and machine learning on Intel® platforms.

Real-world machine learning usually has high CPU and memory requirements, which makes Intel® Xeon Phi™ processors an ideal platform. Intel DAAL provides a quick way of building machine learning applications optimized for Intel® Xeon® and Intel Xeon Phi processors. We will demonstrate how to use KNN (K-nearest neighbors), boosting, and support vector machines (SVM) with Intel DAAL on two real-world machine learning problems, both from **Kaggle: Leaf Classification** and **Titanic: Machine Learning from Disaster** and compare results with the same algorithms from scikit-learn and R.

Why Kaggle?

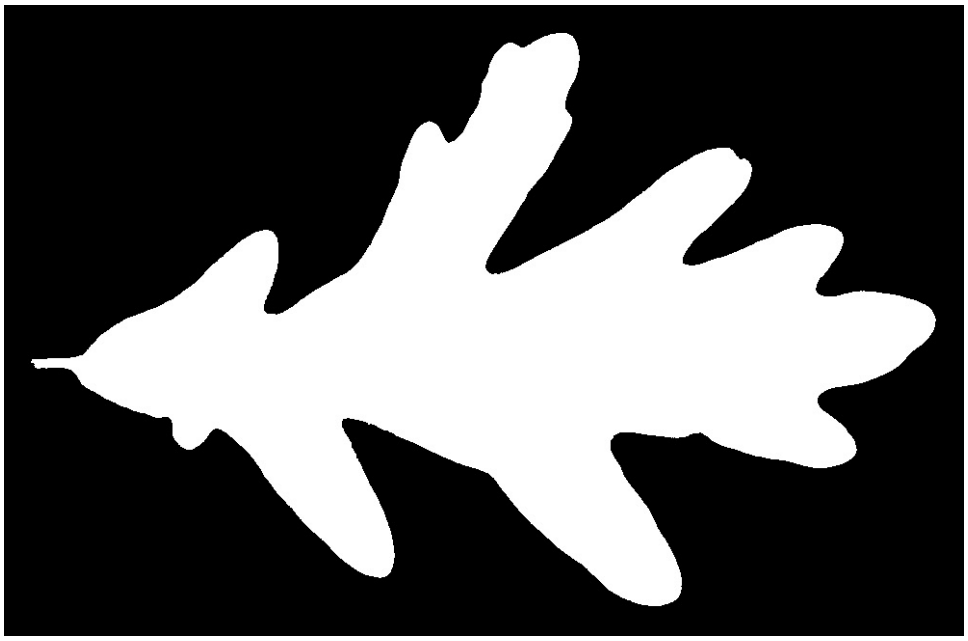
Kaggle is a platform for predictive modeling and analytics competitions in which companies and researchers post their data, and statisticians and data miners from all over the world compete to produce the best models.[1] As of May 2016, Kaggle had more than 536,000 registered users, or "Kagglers." Spanning 194 countries, the community is one of the largest and most diverse in the world. Kaggle has run over 200 data science competitions since it was founded.

The goal of each competition is to produce the best model for a given real-world problem. The model is often evaluated by analyzing its prediction accuracy on a test data set. You can evaluate your model in an instance by submitting your prediction on a test data set and seeing your result on a leaderboard.

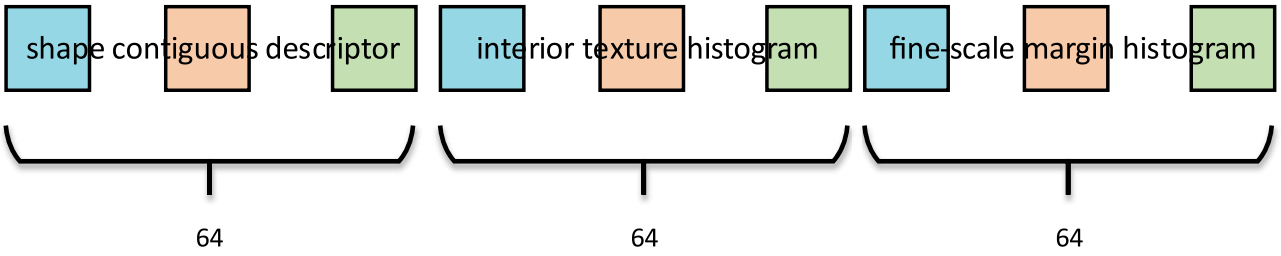
We will evaluate the models, produced by different algorithms and libraries, to see how they perform in Kaggle competitions.

Leaf Classification

There are nearly half a million species of plants in the world. Classifying species has been historically problematic, often resulting in duplicate identifications. This Kaggle challenge is to accurately identify 99 species of plants using leaf images and extracted features (e.g., shape, margin, and texture) to train a classifier. The training data contains 990 leaf images, and the test data contains 594 images (**Figure 1**). Three sets of features are also provided per image: a shape contiguous descriptor, an interior texture histogram, and a fine-scale margin histogram (**Figure 2**). For each feature, a 64-attribute vector is given per leaf sample.



1 Leaf image



2 Features overview

One approach is to apply a list of machine learning algorithms to the training data, evaluate their accuracy on validation data, and find optimal algorithms and hyperparameters. Scikit-learn, the most popular machine learning library among Python data scientists, provides a wide range of algorithms. In the **Kaggle kernel**, we analyzed the prediction accuracy of 10 algorithms. The linear discriminant analysis and KNN algorithms proved to be the best on validation data (see the Kaggle kernel for detailed results).

Here is the linear discriminant analysis in Python (scikit-learn):

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
clf = LinearDiscriminantAnalysis()
clf.fit(X_train, y_train)
test_predictions = favorite_clf.predict(X_test)
```

And KNN in Python (scikit-learn):

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(k=4)
clf.fit(X_train, y_train)
test_predictions = favorite_clf.predict(X_test)
```

In R, you can also apply linear discriminant analysis and KNN.

```
library (MASS)
r <- lda(formula = Species ~ ., data = train)
plda = predict(object = r, newdata = test)
test_predictions = plda$class
```

KNN in R:

```
library(class)
test_predictions = knn(X_train, X_test, y_train, k=4)
```

Intel DAAL provides a scalable version of KNN[2] that uses the KD-tree algorithm and low-level optimizations to make it extremely fast on Intel® architectures while also providing better accuracy.

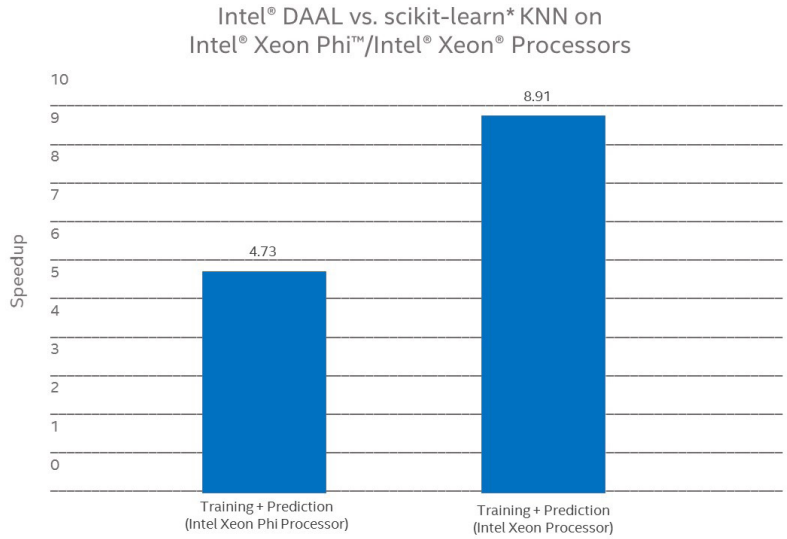
KNN training stage in Python (Intel DAAL):

```
from daal.algorithms.kdtree_knn_classification import training, prediction
from daal.algorithms import classifier, kdtree_knn_classification
trainAlg = kdtree_knn_classification.training.Batch()
trainAlg.input.set(classifier.training.data, X_train)
trainAlg.input.set(classifier.training.labels, y_train)
trainAlg.parameter.k = 4
trainingResult = trainAlg.compute()
```

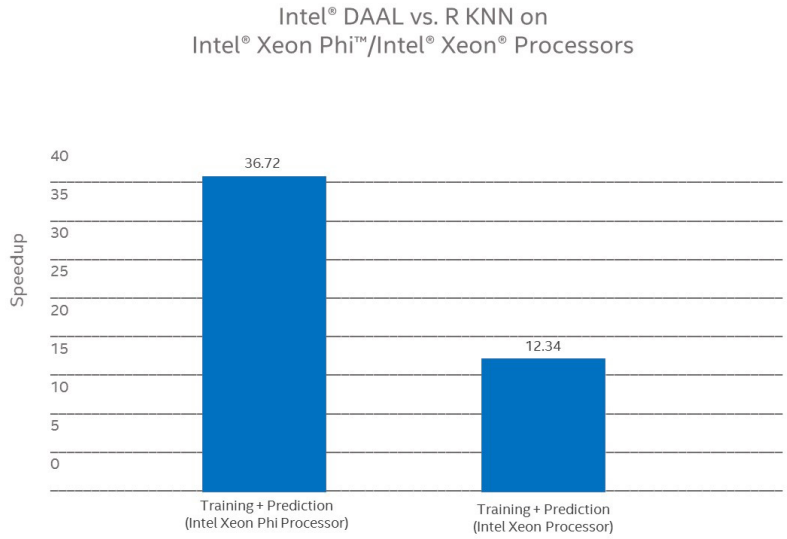
KNN prediction stage in Python (Intel DAAL):

```
predictAlg = kdtree_knn_classification.prediction.Batch()
predictAlg.input.setTable(classifier.prediction.data, X_test)
predictAlg.input.setModel(classifier.prediction.model,
↳ trainingResult.get(classifier.training.model))
predictAlg.compute()
predictionResult = predictAlg.getResult()
test_predictions = predictionResult.get(classifier.prediction.prediction)
```

Figures 3 and 4 show performance comparison graphs. For details on system configurations used for benchmarking, see Configurations and Tools Used at the end of this article.

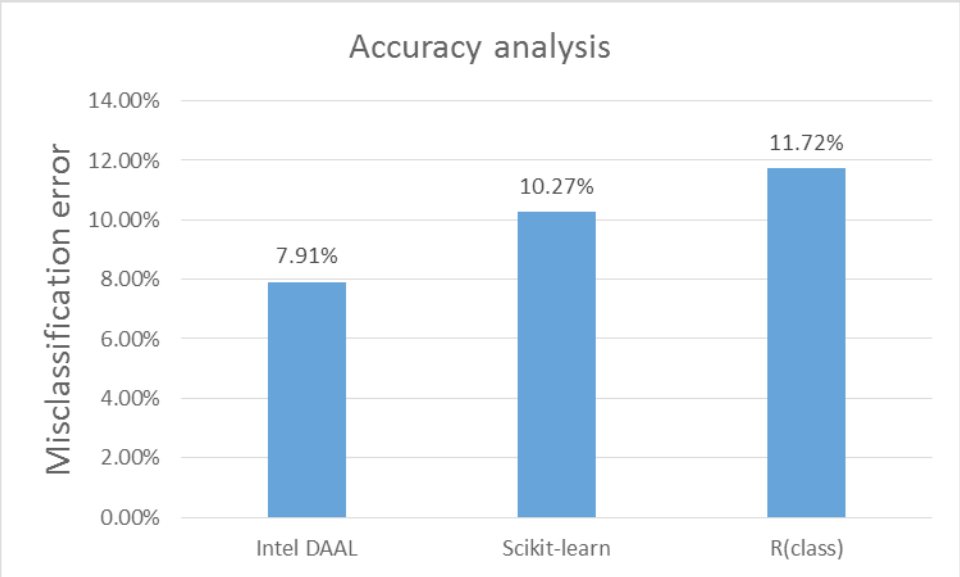


3 Intel® DAAL speedup relative to scikit-learn



4 Intel® DAAL speedup relative to R

Figure 5 shows accuracy comparison graphs:



5 Accuracy comparison, KNN

It is possible to improve KNN accuracy if we apply it to data with fewer dimensions. According to statistical decision theory, if we know the conditional (discrete) distribution $P(G|X)$, where G is a label to predict, and we use the 0-1 loss function, then we predict $\hat{G}(x)=G_k$ if $P(G_k|X=x)=max_{g\in G}P(g|X=x)$. KNN classification assumes that $P(G_k|X=x)$ is constant in the neighborhood of x. Obviously, the larger the number of dimensions, the larger the neighborhood of x containing k training samples. In this problem, we do not have a large number of samples, so settling for the neighborhood as a surrogate for conditioning will fail miserably. The convergence still holds, but the rate of convergence decreases as the dimension increases. See Section 2.4 to 2.5 of *The Elements of Statistical Learning*[3] for a more detailed explanation.

We can improve KNN accuracy by preprocessing the original data using the linear discriminant analysis (LDA) algorithm. In our approach, we preprocessed input data with LDA (nComponents=40) and trained the KNN model on the preprocessed data.

Preprocessing with LDA (Python):

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis(n_components = 40)
X_train_reduced = lda.fit_transform(X_train, y_train)
X_test_reduced = lda.transform(X_test)
```

KNN in Python (scikit-learn):

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(k=4)
clf.fit(X_train_reduced, y_train)
test_predictions = favorite_clf.predict(X_test_reduced)
```

Preprocessing with LDA (R):

```
library (MASS)
r <- lda(formula = Species ~ ., data = train)
plda = predict(object = r, newdata = train)
X_train_reduced = plda$x
plda = predict(object = r, newdata = test)
X_test_reduced = plda$x
```

KNN in R (class):

```
library (class)
test_predictions = knn(X_train_reduced, X_test_reduced, y_train, k=4)
```

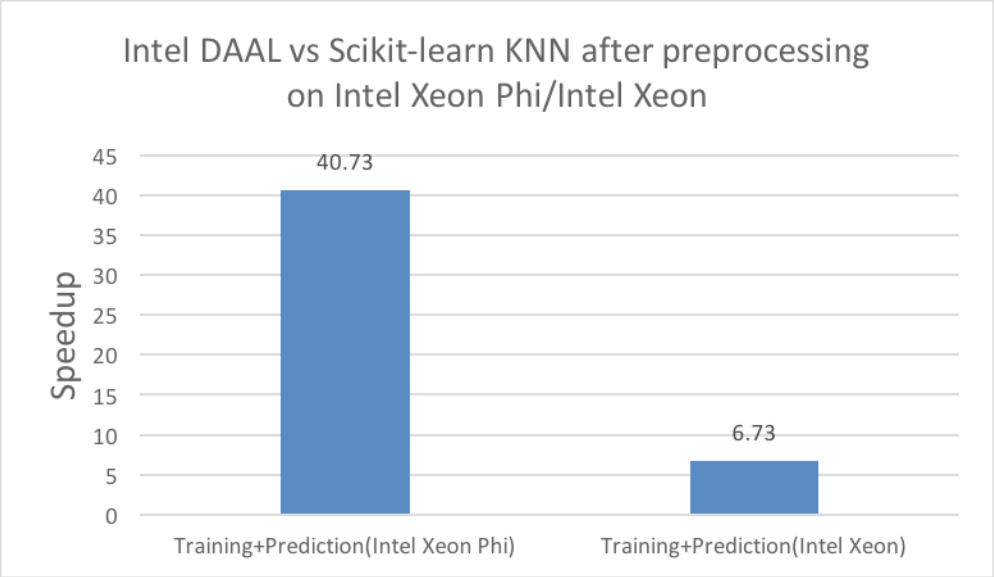
KNN training stage in Python (Intel DAAL):

```
from daal.algorithms.kdtree_knn_classification import training, prediction
from daal.algorithms import classifier, kdtree_knn_classification
trainAlg = kdtree_knn_classification.training.Batch()
trainAlg.input.set(classifier.training.data, X_train_reduced)
trainAlg.input.set(classifier.training.labels, y_train_reduced)
trainAlg.parameter.k = 4
trainingResult = trainAlg.compute()
```

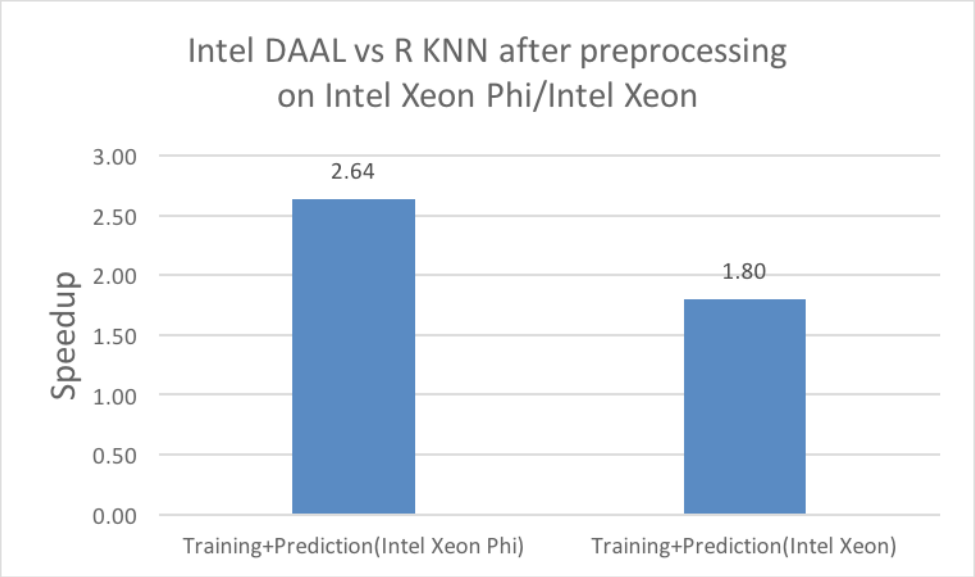
KNN prediction stage in Python (Intel DAAL):

```
predictAlg = kdtree_knn_classification.prediction.Batch()
predictAlg.input.setTable(classifier.prediction.data, X_test_reduced)
predictAlg.input.setModel(classifier.prediction.model,
↳ trainingResult.get(classifier.training.model))
predictAlg.compute()
predictionResult = predictAlg.getResult()
test_predictions = predictionResult.get(classifier.prediction.prediction)
```

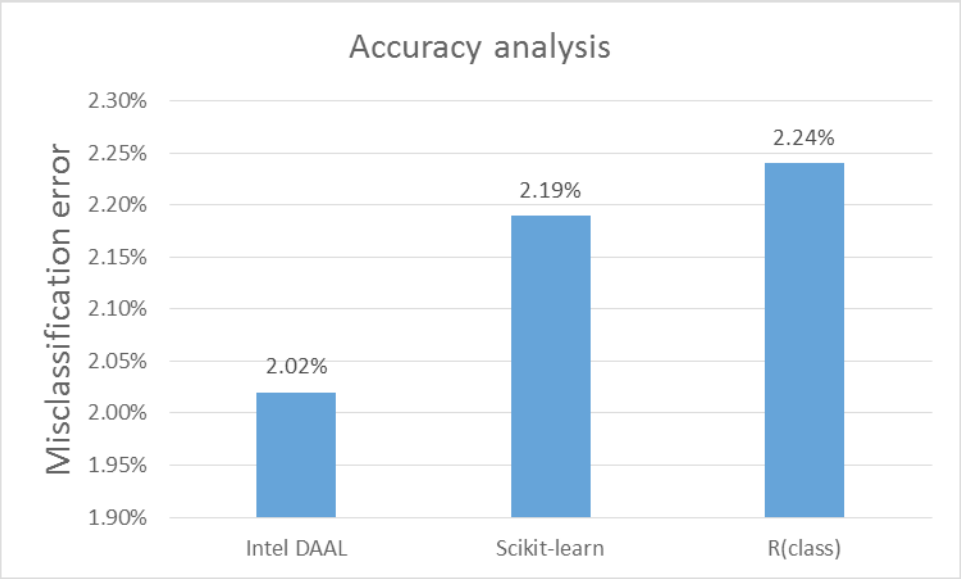
Figures 6, 7, and 8 show the result of applying data preprocessing.



6 Intel® DAAL speedup relative to scikit-learn



7 Intel® DAAL speedup relative to R

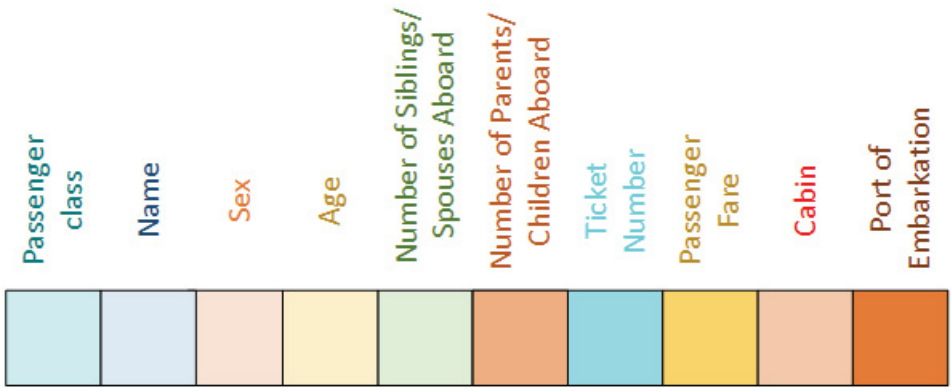


8 Accuracy comparison of KNN after preprocessing

Obviously, feature engineering plays a key role in machine learning, and good feature selection is critical to achieving accurate predictions. Moreover, Intel DAAL achieves the best accuracy and performance among the libraries tested.

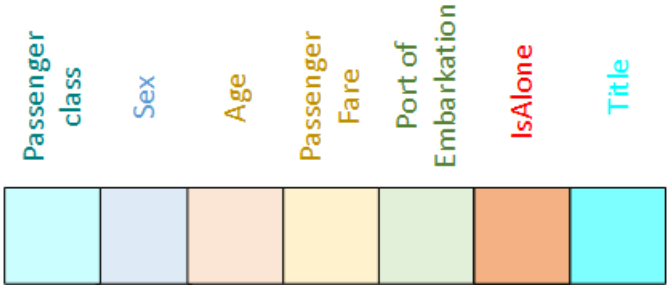
Titanic: Machine Learning from a Disaster

Another Kaggle competition is based on the sinking of the *RMS Titanic*, one of the most infamous shipwrecks in history. On April 15, 1912, during her maiden voyage, the *Titanic* sank after colliding with an iceberg, killing 1,502 out of 2,224 passengers and crew members. This sensational tragedy shocked the world and led to better safety regulations for ships. The challenge here is to analyze the different classes of passengers and crew and predict who among them survived the tragedy. The input data contains the features shown in **Figure 9**. See the **data overview** on Kaggle for details.



9 *Titanic* original features overview

One approach is to preprocess the data into informative feature vectors that can be used to train the machine learning models. Then several classifiers on the preprocessed data should be tried to find out which algorithms perform best. In this **Kaggle kernel**, feature engineering is performed. The following features were constructed from the original ones (see **Figure 10**): passenger class, sex, age (transformed with feature binning), passenger fare (transformed with feature binning), port of embarkation, is alone (true if person has no siblings/spouse/children/parents on *Titanic*), title (Mrs./Miss/Mr./Master). Then, 10 algorithms from scikit-learn were tested and their prediction accuracy was compared. The SVM classifier with the Gaussian kernel gave the best accuracy (see Kaggle kernel for detailed results). SVM parameters are obtained with cross-validation.



10 *Titanic* preprocessed features overview

SVM in Python (scikit-learn):

```
from sklearn.svm import SVC
clf = SVC(C = 5, gamma = 1.5)
clf.fit(X_train, y_train)
test_predictions = favorite_clf.predict(X_test)
```

SVM in R:

```
library(e1071)
model <- svm(X_train, y_train, gamma=1.5, cost=5)
test_predictions <- predict(model, X_test)
```

SVM with the Gaussian kernel involves a lot of time-consuming exponential computations. In Intel DAAL, these computations are highly optimized for Intel architectures, enabling us to quickly create an SVM model.

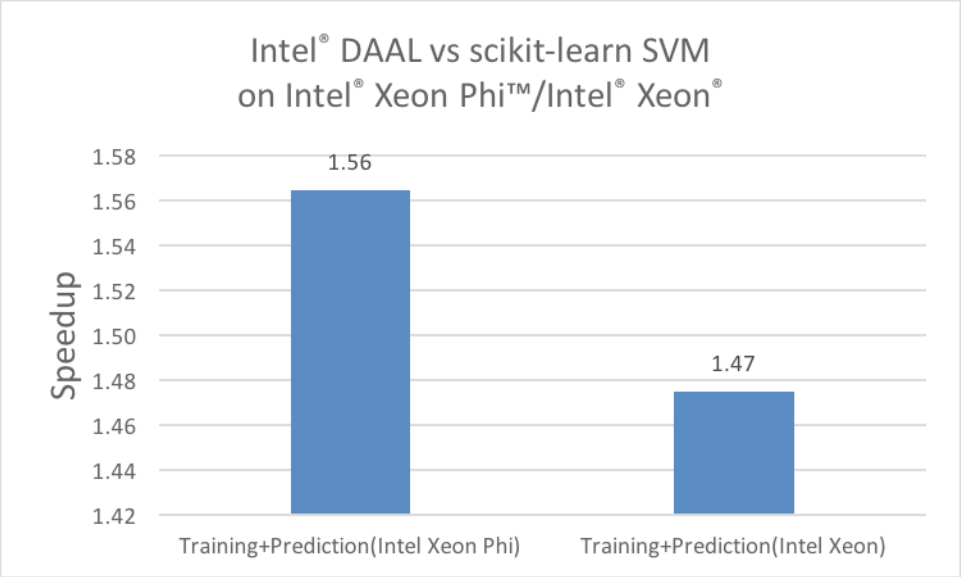
SVM training stage in Python (Intel DAAL):

```
from daal.algorithms.svm import prediction, training
from daal.algorithms import kernel_function, classifier
import daal.algorithms.kernel_function.rbf
trainAlg = svm.training.Batch()
trainAlg.input.set(classifier.training.data, X_train)
trainAlg.input.set(classifier.training.labels, y_train)
kernel = kernel_function.rbf.Batch()
kernel.parameter.sigma = 1.5
trainAlg.parameter.C = 5
trainAlg.parameter.kernel = kernel
trainAlg.parameter.cacheSize = 60000000
trainingResult = trainAlg.compute()
```

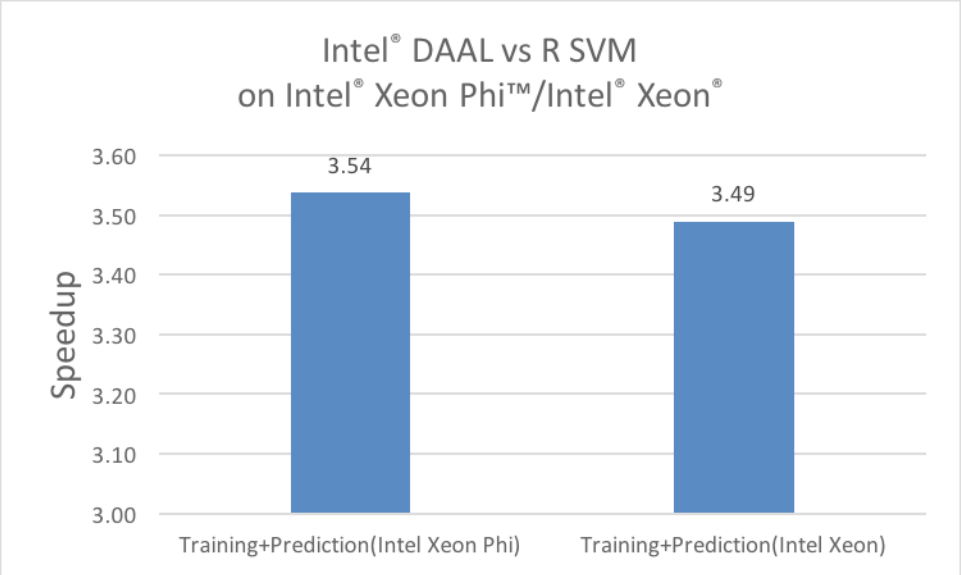
SVM prediction stage in Python (Intel DAAL):

```
predictAlg = svm.prediction.Batch()
predictAlg.input.setTable(classifier.prediction.data, X_test)
predictAlg.input.setModel(classifier.prediction.model,
trainingResult.get(classifier.training.model))
predictAlg.parameter.kernel = kernel
predictAlg.compute()
predictionResult = predictAlg.getResult()
test_predictions = predictionResult.get(classifier.prediction.prediction)
```

Figures 11 and 12 show performance comparison graphs.

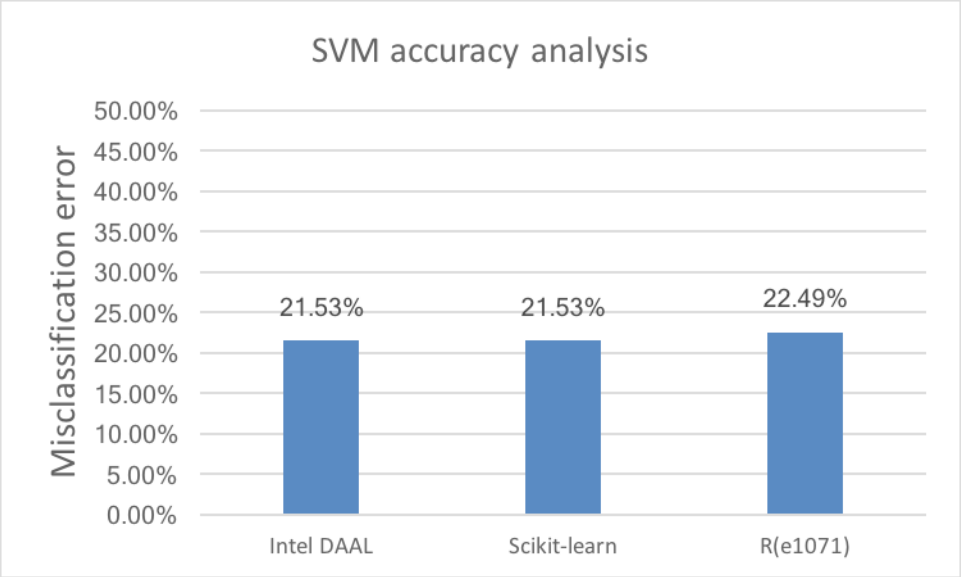


11 Intel® DAAL speedup relative to scikit-learn



12 Intel® DAAL speedup relative to R

Figure 13 shows accuracy comparison graphs:



13 SVM accuracy analysis

We see that Intel DAAL and scikit-learn produced the best accuracy and that Intel DAAL has the best performance.

We will now apply boosting classifiers to this classification problem. Boosting is one of the most powerful learning ideas introduced in the last 20 years. The idea behind boosting is to combine the outputs of many weak classifiers to produce a powerful committee.[3] We will consider the following boosting algorithms:

- AdaBoost*
- BrownBoost*
- LogitBoost*
- Gradient boosting

Numerous resources are available[4, 5, 6] with detailed explanations of these algorithms.

Python (scikit-learn), AdaBoost:

```
from sklearn.ensemble import AdaBoostClassifier
clf = AdaBoostClassifier(n_estimators=1000)
clf.fit(X_train, y_train)
test_predictions = favorite_clf.predict(X_test)
```

Python (scikit-learn), gradient boosting:

```
from sklearn.ensemble import GradientBoostingClassifier
clf = GradientBoostingClassifier(n_estimators=1000)
clf.fit(X_train, y_train)
test_predictions = favorite_clf.predict(X_test)
```

Python (Intel DAAL), AdaBoost (training):

```
from daal.algorithms.adaboost import prediction, training
from daal.algorithms import classifier
trainAlg = training.Batch()
trainAlg.input.set(classifier.training.data, X_train)
trainAlg.input.set(classifier.training.labels, y_train)
trainAlg.parameter. maxIterations = 1000
trainingResult = trainAlg.compute()
```

Python (Intel DAAL), AdaBoost (prediction):

```
predictAlg = prediction.Batch()
predictAlg.input.setTable(classifier.prediction.data, X_test)
predictAlg.input.setModel(classifier.prediction.model,
↳ trainingResult.get(classifier.training.model))
predictAlg.compute()
predictionResult = predictAlg.getResult()
test_predictions = predictionResult.get(classifier.prediction.prediction)
```

Python (Intel DAAL), BrownBoost (training):

```
from daal.algorithms.brownboost import prediction, training
from daal.algorithms import classifier
trainAlg = training.Batch()
trainAlg.input.set(classifier.training.data, X_train)
trainAlg.input.set(classifier.training.labels, y_train)
trainAlg.parameter. maxIterations = 1000
trainingResult = trainAlg.compute()
```

Python (Intel DAAL), BrownBoost (prediction):

```
predictAlg = prediction.Batch()
predictAlg.input.setTable(classifier.prediction.data, X_test)
predictAlg.input.setModel(classifier.prediction.model,
↳ trainingResult.get(classifier.training.model))
predictAlg.compute()
predictionResult = predictAlg.getResult()
test_predictions = predictionResult.get(classifier.prediction.prediction)
```

Python (Intel DAAL), LogitBoost (training):

```
from daal.algorithms.brownboost import prediction, training
from daal.algorithms import classifier
trainAlg = training.Batch()
trainAlg.input.set(classifier.training.data, X_train)
trainAlg.input.set(classifier.training.labels, y_train)
trainAlg.parameter. maxIterations = 1000
trainingResult = trainAlg.compute()
```

Python (Intel DAAL), LogitBoost (prediction):

```
predictAlg = prediction.Batch()
predictAlg.input.setTable(classifier.prediction.data, X_test)
predictAlg.input.setModel(classifier.prediction.model,
↳ trainingResult.get(classifier.training.model))
predictAlg.compute()
predictionResult = predictAlg.getResult()
test_predictions = predictionResult.get(classifier.prediction.prediction)
```

AdaBoost R (fastAdaBoost):

```
library(fastAdaBoost)
model <- adaboost(Survived ~ ., train, 1000)
pred <- predict(model, newdata=test)
```

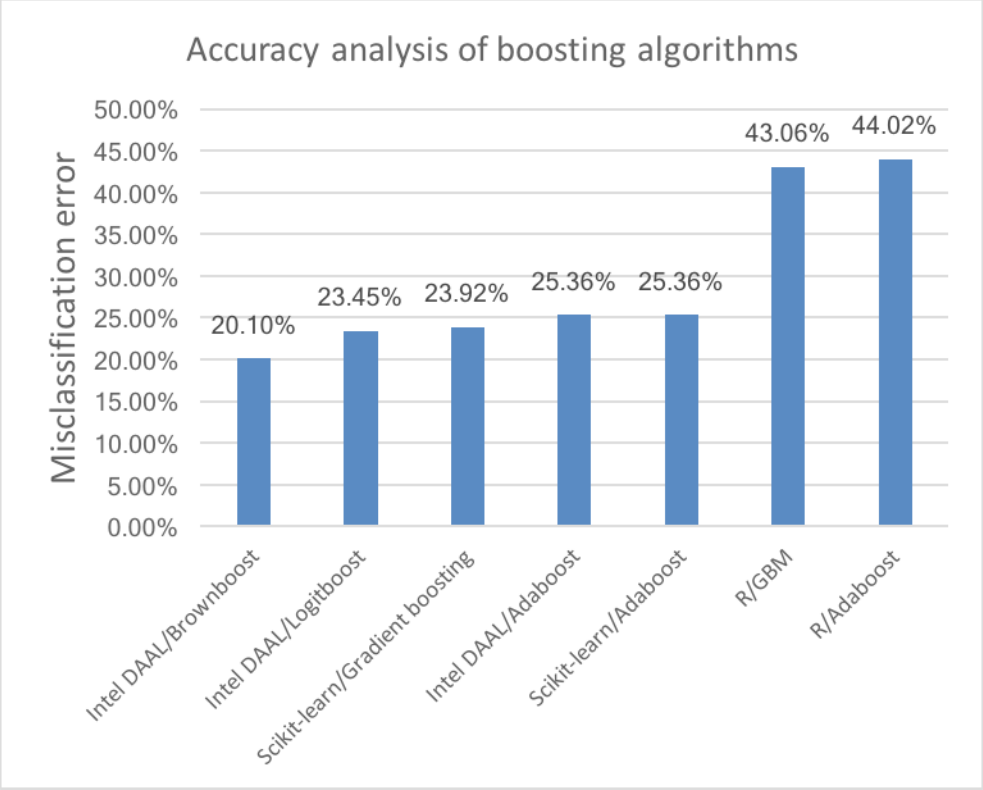
LogitBoost R (caTools):

```
library(caTools)
model <- LogitBoost(X_train, Y_train, nIter=1000)
pred <- predict(model, X_test)
```

Gradient boosting R (gbm):

```
library(gbm)
model <- gbm(Survived ~ ., data=train, n.tree = 1000, shrinkage = 1)
predict(model, test, n.trees = 1000)
```

Figure 14 shows the accuracy of different boosting algorithms.



14 Boosting algorithms accuracy analysis

As we see, the BrownBoost algorithm from Intel DAAL demonstrates the best prediction accuracy.

Solving Data Analytics Problems Using Machine Learning and Intel DAAL

Selecting an algorithm to solve machine learning problems is a nontrivial problem and requires a lot of thought. Libraries, like Intel DAAL or scikit-learn, provide a wide variety of machine learning algorithms, so the user can choose the one that best suits the user's problem.

We demonstrate how you can use Intel DAAL to get all the power of Intel platforms to obtain faster model training and prediction. Our benchmarks show that Intel DAAL has a performance advantage over scikit-learn and R implementations while also producing more accurate models.

References

1. **Overview of Kaggle on Wikipedia**.

2. Patwary, Md. Mostofa Ali; Satish, Nadathur Rajagopalan; Sundaram, Narayanan; Liu, Jialin; Sadowski, Peter; Racah, Evan; Byna, Suren; Tull, Craig; Bhimji, Wahid; Prabhat, Dubey, Pradeep. 2016. "**PANDA: Extreme Scale Parallel K-Nearest Neighbor on Distributed Architectures**," IEEE International Parallel and Distributed Processing Symposium.

3. Hastie, Trevor; Tibshirani, Robert; and Friedman, Jerome. 2009. *The Elements of Statistical Learning, Second Edition.* Springer International Publishing AG.

4. Freund, Yoav, and Schapire, Robert E. 1999. "Additive Logistic Regression: A Statistical View of Boosting," *Journal of Japanese Society for Artificial Intelligence* (14[5]), pp. 771–780.

5. Friedman, Jerome; Hastie, Trevor; and Tibshirani, Robert. 2000. "Additive Logistic Regression: A Statistical View of Boosting," *The Annals of Statistics*, 28(2), pp. 337–407.

6. Friedman, Jerome. 2001. "Greedy Function Approximation: A Gradient Boosting Machine," *The Annals of Statistics* 29(5), pp. 1189–1232.

BLOG HIGHLIGHTS

The Right Mindset for Testing (Testing Theory Part 1)

BY **JAKOB ENGBLOM** ›

A recent blog post I wrote about the ESA Schiaparelli crash triggered a discussion about testing, execution tools for testing, and the right mindset for testing. If you look back at what I have written in the past on **this blog** and the **Wind River blog**, there is a recurring theme of expanding testing beyond the obvious and testing what cannot be easily tested in the real world (by using simulation). In this two-part series of blog posts about testing theory, I will attempt to summarize my thoughts on testing, and share some anecdotes along the way.

Read more ›

Configurations and Tools Used

System configurations used for benchmarking:

Intel® Xeon®:

Model name:	Intel® Xeon®CPU E5-2699 v4 @ 2.20 GHz
Core(s) per socket:	22
Socket(s):	2
MemTotal:	256 GB

Intel® Xeon Phi™:

Model name:	Intel® Xeon Phi™ Processor 000A @ 1.40 GHz
Core(s) per socket:	68
Socket(s):	1
RAM:	16 GB

Software tools used in this example:

- Intel® DAAL 2017 Beta update 2
- R version 3.3.2
- scikit-learn version 0.19.1
- Class package version 7.3-14
- MASS package version 7.3-45
- e1071 package version 1.6-8
- fastAdaBoost package version 1.0.0
- caTools package version 1.17.1
- gbm package version 2.1.1

HPC WITH R: THE BASICS

Drew Schmidt, *Graduate Research Assistant*, University of Tennessee, Knoxville

Some say R is not fit for high-performance computing (HPC). Others say, "No ... wait ... you're actually serious? R in HPC?"

However, the world is changing. Data analytics is the new hip, cool thing. And whether you see tangible benefits in data science, or just dollar signs, the fact is that R excels here. The HPC landscape is changing to better accommodate data analytics applications and users. Therefore, R is the natural candidate on which to focus your attention due to its overwhelming popularity. Fortunately, many have already taken up the call to embed R in HPC environments.

I'm making some assumptions about my audience here. I'm guessing you may not know much about R, but are at least curious. Maybe this curiosity is being driven by an application need, a fear of missing out (all the cool kids are programming in R), or maybe you have clients increasingly asking you about R solutions. Whatever your motivations, welcome. I'm glad to have you. I'm also assuming that you're otherwise fairly well versed in HPC sorts of things. You know what a compiler is, and Intel® compilers are your favorites.

Given these assumptions, this article will be a bit different. I want to introduce you to the basics and introduce some of the competing ideas in the R landscape without trying too hard to pick sides. Maybe it's more exciting to talk about distributed computing with R on thousands of nodes and getting interactive speeds on terabytes of data—and we can do that. But given the relative obscurity of R in HPC circles, I feel compelled to take some time learning to crawl before we fire ourselves out of a cannon.

We'll begin with a little history and some basics that everyone who picks up R and cares about performance ought to know. We'll spend a little more time talking about integrating compiled code into R and then close off with a discussion of parallel computing. It's a bit of a whirlwind, and this article won't make you an expert on any one topic. But the hope is to give you enough to get you started working with R more seriously.

Background

R can trace its roots back to 1976, when John Chambers, then of Bell Labs, began working on S. S was originally designed as an interactive interface to a bunch of Fortran code. And, try as you might, you just can't get rid of Fortran, so that's also pretty much how R works today. R itself was released in the early 1990s, created by Ross Ihaka and Robert Gentleman as a free S. Strictly speaking, R is a dialect of the S language, which is a very snooty way of saying that there's a lot of S code written in the '80s that still runs in R.

R is a strange language. One of my favorite examples to demonstrate this is:

```
typeof(1)
## [1] "double"

typeof(2)
## [1] "double"

# 1:2 is the vector of numbers "1 2"
typeof(1:2)
## [1] "integer"
```

It sort of makes sense if you stare at it long enough. It's pretty reasonable to assume that `1:2` is probably going to be an index of some kind. But it's still pretty weird. In general, `:` is a bit unpredictable. If you do `"1":"2"`, then it will return the vector of ints 1 and 2. So it just does an ASCII conversion with chars to ints, right? Except that `"A":"B"` errors. And `:` doesn't always produce integers; you can do `1.5:2.5`, for example.

R has a very vibrant community, boasting over 10,000 contributed packages on its Comprehensive R Archive Network (CRAN). These packages represent everything from hardcore numerical computing, to cutting-edge statistics and data science methodology, to building an interactive data analysis website (it's called **Shiny** and it's amazing). And R's package infrastructure is unparalleled as far as I'm concerned. It's about as "it just works" a thing as you will ever find. So for all the Python* fans who've been giggling and nodding "so true" up until this point, now might be a good time to explain how a bunch of statisticians who you think have no idea what they're doing somehow managed to create the only packaging framework that isn't abject misery to use.

Now, when I said R was popular, I wasn't kidding. In 2016, the **IEEE Spectrum** programming language rankings placed R in the number five spot, beating out C# and JavaScript*. What is especially interesting about this is that the rankings are of programming languages. And even the people who love R will tell you that it's a terrible programming language. R is just so good for data analysis that people are willing to overlook all of its peculiarities to see the really beautiful gem hiding underneath.

Said another way, R is a bit like Jack Sparrow from the *Pirates of the Caribbean* films: it may be the worst (language) you've ever heard of … but you have heard of it.

Free Improvements

American comedian W. C. Fields once said, "The laziest man I ever met put popcorn in his pancakes so they would turn over by themselves." I suspect this apocryphal man Fields speaks of would have made a fantastic engineer. After all, why work hard when we can let others do the hard work for us?

In R, there are a few ways to engage in this kind of giant-shoulder-standing. First—and this should hardly come as any surprise—if you compile R with good compilers, you can expect to see some nontrivial performance gains. R is written in C, Fortran, and R, so using Intel's icc and ifort on Intel® hardware is a good place to start. And, lucky you, Intel has a **very nice article** on how to build R with Intel compilers.

That is a strong first step in getting good performance out of R. But what about all that R code making up base R? R has had a bytecode compiler since version 2.13.0, and it compiles R internals for 2.14.0 and later. For code that you write, you have traditionally needed to go somewhat out of your way to use the bytecode compiler. However, in version 3.4.0 (due for release shortly at the time of this writing), R will include a JIT, making many of the old recommendations for using the compiler fairly moot.

Now, it's worth pointing out that the bytecode compiler is not nearly as nice as a real compiler. If your code has bad design, like unnecessarily computing something, then it's still going to be there; the computation is just executed in its bytecode form. It's also not an R-to-C translator or anything like that. It does best on loop-heavy code (not counting implicit loops), and (from a performance standpoint) tends to do next to nothing otherwise. I have seen it improve a loop body by the order of 10 percent, and I have seen it affect the performance by 0.01 percent. But hey, it doesn't take any work on your part, so we'll take what we can get.

These improvements are all fine and will definitely help with the runtime of your R code, but it won't blow your socks off. Now, if you're looking to buy a new pair of socks, then you can get really impressive performance gains by choosing good LAPACK and BLAS libraries. These are de facto standard numerical libraries for matrix operations, and R uses them to power its low-level linear algebra, and most of its statistical operations. Ironically, perhaps the most important operation in statistics, linear regression, does not use LAPACK. Instead, it uses a highly modified version of LINPACK. No, not the benchmark that runs on supercomputers. I'm talking about the '70s predecessor to LAPACK. The reasons for this are a bit complicated, but there are reasons. So your fancy tuned LAPACK won't help with linear regression, but it can still take advantage of good level-one BLAS.

R ships with the so-called "reference" BLAS, which is bone-achingly slow. The above example notwithstanding, if you link R with good BLAS and LAPACK implementations, then you can expect to see significant performance improvements. And, as luck would have it, Intel has a very high-quality implementation in the **Intel® Math Kernel Library (Intel® MKL)**. Microsoft offers a distribution of R, which, to my understanding, is R compiled with Intel compilers and shipped with Intel MKL. They call it **Microsoft R Open***, and it is freely available. They also maintain a detailed collection of **benchmarks** demonstrating the power of Intel MKL. Or, if you prefer, you can follow Intel's own **documentation** for linking R with Intel MKL.

And for those of you with all the newest, fanciest toys: yes, this applies to MIC accelerators as well. A lot of the early work on this comes from the fine folks at the **Texas Advanced Computing Center (TACC)**, who have done quite a bit of experimenting with using Intel MKL Automatic Offload. To say that things work well is a bit of an understatement, and R users with a few Intel® Xeon Phi™ processors lying around should seriously consider trying this for themselves. If you aren't sure where to start, Intel has also produced a very handy **guide** to help you with exactly this kind of thing.

Leveraging Compiled Code

One of the interesting revolutions happening in the R world today is the increasing use of C++ in R packages. Most of the credit for this belongs to Dirk Eddelbuettel and Romain Francois, who created the **Rcpp** package. Rcpp makes it significantly easier to incorporate C++ code into an R analysis pipeline. Yes, somehow they managed to convince a bunch of statisticians who thought Python was too complicated to program in C++. I'm just as amazed as you are. But however they managed to pull it off, we are all the beneficiaries. This means that CRAN packages are only getting faster and consuming less memory. And those using the Intel compilers stand to benefit the most from this revolution, because, as we discussed earlier: better compiler, faster code.

Now, for those who prefer vanilla C, R has a first-class C API. In fact, it's on this foundation that Rcpp is built; although I think it's fair to say that Rcpp goes to much greater lengths. But getting back to the C API, this is also convenient for those who can tolerate Fortran and are willing to write a C wrapper and don't want to bring the C++ linker to the party. This API is mostly documented in the **Writing R Extensions** manual, which is an indispensable resource for anyone working with R. However, to answer some questions that arise, you may find yourself poking around R's header files if you go this route.

For the Rcpp route, you install it as you would any other R package, namely:

```
install.packages("Rcpp")
```

For a quick example of the power of this package, let's take a look at the "numerical Hello World," the Monte Carlo integration, to find π that you've seen a million times before. In R, you might write something like this:

```
mcpi <- function(n)
{
  r <- 0L

  for (i in 1:n){
    u <- runif(1)
    v <- runif(1)

    if (u*u + v*v <= 1)
      r <- r + 1L
  }

  return(4*r/n)
}
```

Now when I say "you might write," I am again assuming you're not that familiar with R. Probably no experienced R user would ever write such a thing. One could reasonably argue that it looks a bit like C. The best advice I could give to anyone who ever inherits an R codebase for the purposes of making it faster is: the more it looks like C, the worse it will run in R, but the easier it is to convert to C/C++. The inverse is also true, in that the less it looks like R, the harder it is to convert. A more natural R solution is the following vectorized gibberish:

```
mcpi_vec <- function(n)
{
  x <- matrix(runif(n * 2), ncol=2)
  r <- sum(rowSums(x^2) <= 1)

  return(4*r/n)
}
```

Like in every other high-level language, the use of **vectorization** will improve the runtime performance, but also gobble up a lot more RAM. So instead, let's forget all this R business and just write a version in C++:

```
#include <Rcpp.h>

// [[Rcpp::export]]
double mcpi_rcpp(const int n)
{
  int r = 0;

  for (int i=0; i<n; i++){
    double u = R::runif(0, 1);
    double v = R::runif(0, 1);

    if (u*u + v*v <= 1)
      r++;
  }

  return (double) 4.*r/n;
}
```

Now, except for that mysterious `Rcpp::export` bit, that probably looks like very readable C++. Well, it turns out that the mysterious bit will handle the generation of all of the "boilerplate" code. In R, there are no scalars (hey, I told you it was a weird language), only vectors of length 1. So, behind the scenes, Rcpp is actually handling this mental overhead for you and, in its wrapper, will create a length 1 double vector for you. As a general rule, we can play this game with integers and doubles, and vectors of these basic types. More complicated things involve more complications. But hey, that's pretty nice, right?

To compile/link/load and generate the various boilerplates, we need only call `sourceCpp()` to make the function immediately available to R:

```
Rcpp::sourceCpp(file="mcpi.cpp")
mcpi_rcpp(10000)
## [1] 3.1456
```

Eagle-eyed readers may be wondering, "Isn't '10000' here a double?" And you'd be correct in thinking so, because it is. We could demand an integer by calling with `10000L`—that's an ordinary 32-bit integer, mind you—but Rcpp will automatically handle type conversions for you. It's actually handling the conversion exactly as the R code versions are. This has fairly obvious pros and cons, but it's the approach they took, and is worth noting and being aware of.

We can easily compare the performance of the three using the rbenchmark or microbenchmark packages. I happen to prefer rbenchmark, which goes something like:

```
library(rbenchmark)

n <- 100000
cols <- c("test", "replications", "elapsed", "relative")
benchmark(mcpi(n), mcpi_vec(n), mcpi_rcpp(n), columns=cols)
##           test replications elapsed relative
## 1      mcpi(n)          100  49.901  214.167
## 2  mcpi_vec(n)          100   1.307    5.609
## 3 mcpi_rcpp(n)          100   0.233    1.000
```

And hey, that's pretty good! Now, of course, this opens up opportunities for things like OpenMP* or **Intel® Threading Building Blocks (Intel® TBB)**. But speaking of parallelism...

Parallel Programming

Since version 2.14.0, R ships with the parallel package. This allows for pretty simple task-level parallelism by offering two separate APIs, one using sockets, and one using the OS fork. The reason for the two interfaces is one part historical, in that they are derived from the older contributed packages, multicore and snow. But the desire to keep both is probably best explained by R core's desire to support all platforms, even Windows* (which lacks `fork`). On a non-Windows platform, the function of interest is `mclapply()`, and it's the multicore `lapply()`—so named because it applies a function and returns a list. Here R flexes some of its functional programming muscles:

```
lapply(my_data, my_function)
parallel::mclapply(my_data, my_function)
```

The data can be an index or convoluted list of very large, complex objects. So long as the supplied function can handle the inputs, it'll work.

Now that's one of the two officially supported interfaces. The other is more complicated and generally only used by Windows programmers. This creates a bit of a rift for R users. This is purely my own opinion, but I feel that R users don't really like having multiple options as much as your regular programmer working in another language. They want *one* good way to do things and for that to be the end of the discussion. It's to this end that (ironically) several projects have emerged to try to unify all of the disparate interfaces. These include the older and more established foreach package, as well as the newer BiocParallel from the Bioconductor project.

You might wonder what the big deal is about having two separate interfaces. Well, in fact, there are many more packages that enable parallelism in R. The **HPC Task View** is a good resource to discover the many options.

If it sounds to you like most of the focus and interest has been on shared-memory parallelism, you'd be right. The R community in general has been a bit resistant to caring much about performance until relatively recently. I think this is largely because the R mind-space is still dominated by the statistics side of data science. Most big data problems in statistics, frankly, aren't. You can still manage to get a lot done by just downsampling your data and using classical statistics techniques. It's not the right tool for every job, but it certainly has its place—and if anything, these tools are underappreciated.

But none of that involves supercomputers, so forget that nonsense. Let's talk about MPI. The Rmpi package dates all the way back to 2002. More recently, the **Programming with Big Data in R (pbdR)** project has been developing packages for doing large-scale computing with R in supercomputing environments. Now, full disclosure: I work on that project. As such my opinions on it are naturally biased. But I think we have a few interesting things to show you.

We maintain quite a few packages, and generally try to bring the best of HPC to R for data analysis and profiling. For the sake of brevity, let's just focus on direct MPI programming. We'll briefly compare Rmpi and our package, pbdMPI. First, and this is a big one, Rmpi can be used interactively, but pbdMPI cannot. This is because pbdMPI is designed to be used exclusively in single program, multiple data (SPMD) style, whereas Rmpi is really meant to work in a manager/worker style. If you've ever submitted a batch job on a cluster using MPI, you were almost certainly writing in SPMD. It's one of those ideas that's so intuitive, if you've never heard of it before, you'll be surprised it even has a name. So for those coming to R from the HPC world, pbdMPI should feel right at home.

Beyond the programming style, there are some serious differences between the APIs of the two packages. In Rmpi, you need to specify the type of the data. For example:

```
library(Rmpi)
mpi.allreduce(x, type=1) # int
mpi.allreduce(x, type=2) # double
```

Remember our type example from the beginning? In pbdMPI, we use a lot of R's object-oriented facilities to try to automatically handle these and other low-level details:

```
library(pbdMPI)
allreduce(x)
```

It's a small example but a good demonstration of our philosophy. We think the HPC community does great work, but we also think HPC tools are too hard to use for most people and should be made simpler.

For a slightly more substantive example, let's take a quick look at parallel "Hello World." Now, we mentioned that pbdMPI has to be used in batch mode. The downside is that R users have trouble thinking in terms of batch rather than interactive processing. The upside is that it plays well with all of the HPC things like resource managers and job schedulers that you already know about. Say we wanted to run our "Hello World" example:

```
library(pbdMPI)

comm.print(paste("Hello from rank", comm.rank(), "of", comm.size()), all.
rank=TRUE)

finalize()
```

We just need to make the appropriate call to `mpirun` (or your system's equivalent):

```
mpirun -np 2 Rscript hello_world.r
```

Which gives the output you would expect:

```
[1] “Hello from rank 0 of 2”
[1] “Hello from rank 1 of 2”
```

Summary

There’s a famous saying in statistics circles, attributed to George Box: All models are wrong, but some are useful. Well, I posit that all programming languages are bad, but some are useful. R is perhaps the ultimate expression of this idea. After all, there has to be something to it if it’s held up for 40 years (counting S) and is currently ranked fifth among programming languages. And while R has a reputation for being slow, there are definitely strategies to mitigate this. Use a good compiler. Good BLAS and LAPACK will improve the performance of many data science operations. Embedding compiled kernels in your R analysis pipeline can greatly enhance performance. And, when in doubt, throw more cores at your problem.

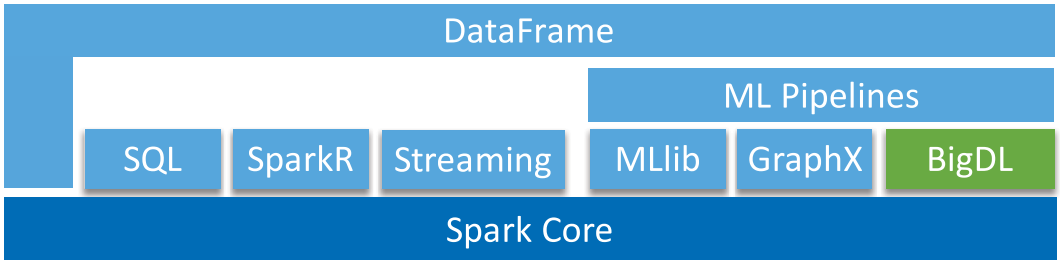
BIGDL: OPTIMIZED DEEP LEARNING ON APACHE SPARK*

Making Deep Learning More Accessible with an Open Source, Distributed Deep Learning Framework

Jason Dai, *Senior Principal Engineer,* and Radhika Rangarajan, *Technical Program Manager,* Intel Corporation

Artificial intelligence (AI) plays a central role in today's smart and connected world—and is continuously driving the need for scalable, distributed big data analytics with deep learning capabilities. There is also an increasing demand to conduct deep learning in the same cluster along with existing data processing pipelines to support feature engineering and traditional machine learning. To address the need for a unified platform for big data analytics and deep learning, Intel recently released **BigDL**, an open source distributed deep learning framework for Apache Spark*. In this article, we'll discuss BigDL features and how to get started building models using BigDL.

BigDL is implemented as a library on top of Spark (**Figure 1**), allowing easy scale-out computing. With BigDL, users can write their deep learning applications as standard Spark programs, which can directly run on top of existing Spark or Hadoop* clusters.



1 BigDL implementation

Overview of BigDL

BigDL brings native support for deep learning functionalities to big data and Spark platforms by providing:

- **Rich, deep learning support.** Modeled after Torch, BigDL provides comprehensive support for deep learning, including numeric computing (e.g., Tensor) and high-level neural networks. In addition, users can load pretrained Caffe* or Torch models into Spark programs using BigDL.
- **Extremely high performance.** To achieve high performance, BigDL uses **Intel® Math Kernel Library (Intel® MKL)** and multithreaded programming in each Spark task. Consequently, it is orders of magnitude faster than out-of-box open source Caffe, Torch, or TensorFlow* on a single-node Intel® Xeon® processor (i.e., comparable with mainstream GPU).
- **Efficient scale-out.** BigDL can efficiently scale out to perform data analytics at big data scale by leveraging Apache Spark, as well as efficient implementations of synchronous SGD and all-reduce communications on Spark.

Native integration with Spark is a key advantage for BigDL. Since it is built on top of Spark, it is easy to distribute model training, the computationally intensive part of deep learning. Rather than requiring the user to explicitly distribute the computation, BigDL automatically spreads the work across the Spark cluster.

BigDL supports Hadoop and Spark as unified data analytics platforms (for data storage, data processing and mining, feature engineering, classical machine learning, and deep learning) and makes deep learning more accessible to big data users and data scientists.

Typical BigDL use cases include:

- **Analyzing a large amount of data** using deep learning technologies, on the same big data (Hadoop and/or Spark) cluster where the data are stored (in, say, HDFS*, HBase*, Hive*, etc.) to eliminate a large volume of unnecessary data transfer between separate systems.
- **Adding deep learning functionalities** (either training, fine-tuning, or prediction) to the big data (Spark) programs and/or workflow to reduce system complexity and the latency for end-to-end learning.
- **Leveraging existing Hadoop and/or Spark clusters** to run the deep learning applications, which can then be dynamically shared with other workloads (e.g., ETL, data warehousing, feature engineering, classical machine learning, graph analytics, etc.).

Getting Started with BigDL

BigDL provides comprehensive support for deep learning, including numeric computing (e.g., Tensor), high-level neural networks, as well as distributed stochastic optimizations (e.g., synchronous minibatch SGD and all-reduce communications on Spark). **Table 1** summarizes the abstractions and APIs provided by BigDL.

Name	Descriptions
`Tensor`	Multidimensional array of numeric types (e.g., Int, Float, Double)
`Module`	Individual layers of the neural network (e.g., ReLU, Linear, SpatialConvolution, Sequential)
`Criterion`	Given input and target, computing gradient per given loss function
`Sample`	A record consisting of `feature` and `label`, each of which is a tensor
`DataSet`	Training, validation, and test data; one may use `Transformer` to perform series of data transformations (w/ -> operators) on DataSet
`Engine`	Runtime environment for the training (e.g., node#, core#, spark versus local, multithreading)
`Optimizer`	Stochastic optimizations for local or distributed training (using various `OptimMethod` such as SGD, AdaGrad)

Table 1. BigDL abstractions and APIs

A BigDL program can run either as a local Scala/Java* program or as a Spark program. [Editor's Note: Python* support will be available shortly and may even be available by the time this article is published.] To quickly experiment with BigDL code as a local Scala/Java program using the interactive Scala shell (REPL), one can first type:

```
$ source PATH_To_BigDL/scripts/bigdl.sh
$ SPARK_HOME/bin/spark-shell --jars bigdl-0.1.0-SNAPSHOT-jar-with-dependencies.
jar
```

The Scala shell will show something like:

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_79)
Spark context available as sc.
scala>
```

One can then experiment with the Tensor APIs in BigDL as follows:

```
scala> import com.intel.analytics.bigdl.tensor.Tensor
import com.intel.analytics.bigdl.tensor.Tensor

scala> Tensor[Double](2,2).fill(1.0)
res9: com.intel.analytics.bigdl.tensor.Tensor[Double] =
1.0   1.0
1.0   1.0
[com.intel.analytics.bigdl.tensor.DenseTensor of size 2x2]
```

One can also experiment with the Module APIs in BigDL as follows:

```
scala> import com.intel.analytics.bigdl.numeric.NumericFloat // import global
float tensor numeric type
import com.intel.analytics.bigdl.numeric.NumericFloat

scala> import com.intel.analytics.bigdl.nn._
import com.intel.analytics.bigdl.nn._

scala> val f = Linear(3,4) // create the module
mlp: com.intel.analytics.bigdl.nn.Linear[Float] = nn.Linear(3 -> 4)

// let's see what f's parameters were initialized to. ('nn' always inits to
something reasonable)
scala> f.weight
res5: com.intel.analytics.bigdl.tensor.Tensor[Float] =
-0.008662592  0.543819    -0.028795477
-0.30469555   -0.3909278   -0.10871882
0.114964925   0.1411745    0.35646403
-0.16590376   -0.19962183   -0.18782845
[com.intel.analytics.bigdl.tensor.DenseTensor of size 4x3]
```

Building a Simple Text Classifier with BigDL

Going beyond APIs, let's see how to build a text classifier using a simple **convolutional neural network (CNN) model**.

A BigDL program starts with import `com.intel.analytics.bigdl._` and then initializes the engine (including the number of executor nodes, the number of physical cores on each executor, and whether it runs on Spark or as a local Java program):

```
val sc = new SparkContext(
 Engine.init(param.nodeNum, param.coreNum, true).get
  .setAppName("Text classification")
  .set("spark.akka.frameSize", 64.toString)
  .set("spark.task.maxFailures", "1"))
```

After that, the example broadcasts the pretrained word embedding and loads the input data using RDD transformations (`vectorizedRdd`):

```
 // For large dataset, you might want to get such RDD[(String, Float)] from
HDFS
 val dataRdd = sc.parallelize(loadRawData(), param.partitionNum)
 val (word2Meta, word2Vec) = analyzeTexts(dataRdd)
 val word2MetaBC = sc.broadcast(word2Meta)
 val word2VecBC = sc.broadcast(word2Vec)
 val vectorizedRdd = dataRdd
   .map {case (text, label) => (toTokens(text, word2MetaBC.value), label)}
   .map {case (tokens, label) => (shaping(tokens, sequenceLen), label)}
   .map {case (tokens, label) => (vectorization(
    tokens, embeddingDim, word2VecBC.value), label)}
```

It then converts the processed data (`vectorizedRdd`) to an RDD of Sample, and then randomly splits the sample RDD (`sampleRDD`) into training data (`trainingRDD`) and validation data (`valRDD`):

```
val sampleRDD = vectorizedRdd.map {case (input: Array[Array[Float]],
                     label: Float) =>
  Sample(
   featureTensor = Tensor(input.flatten, Array(sequenceLen, embeddingDim))
    .transpose(1, 2).contiguous(),
   labelTensor = Tensor(Array(label), Array(1)))
 }

val Array(trainingRDD, valRDD) = sampleRDD.randomSplit(
 Array(trainingSplit, 1 - trainingSplit))
```

After that, the example builds the CNN model by calling `buildModel`:

```
def buildModel(classNum: Int): Sequential[Float] = {
 val model = Sequential[Float]()
 model.add(Reshape(Array(param.embeddingDim, 1, param.maxSequenceLength)))
 model.add(SpatialConvolution(param.embeddingDim, 128, 5, 1))
 model.add(ReLU())
 model.add(SpatialMaxPooling(5, 1, 5, 1))
 model.add(SpatialConvolution(128, 128, 5, 1))
 model.add(ReLU())
 model.add(SpatialMaxPooling(5, 1, 5, 1))
 model.add(SpatialConvolution(128, 128, 5, 1))
 model.add(ReLU())
 model.add(SpatialMaxPooling(35, 1, 35, 1))
 model.add(Reshape(Array(128)))
 model.add(Linear(128, 100))
 model.add(Linear(100, classNum))
 model.add(LogSoftMax())
 model
}
```

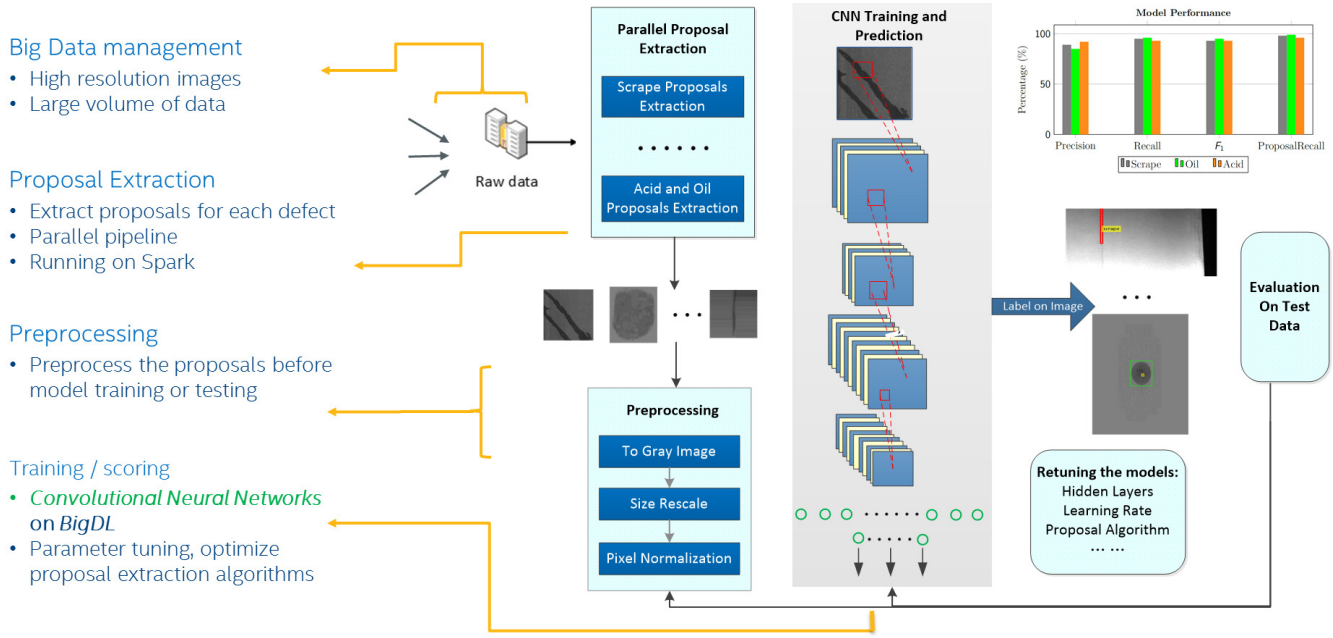
It then creates the `Optimizer`, passes the RDD of training data (`trainingRDD`) to the Optimizer (with specific batch size), and finally trains the model (using `Adagrad` as the optimization method, and setting relevant hyperparameters in state):

```
 val optimizer = Optimizer(
  model = buildModel(classNum),
  sampleRDD = trainingRDD,
  criterion = new ClassNLLCriterion[Float](),
  batchSize = param.batchSize
 )
 val state = T("learningRate" -> 0.01, "learningRateDecay" -> 0.0002)
 optimizer
   .setState(state)
   .setOptimMethod(new Adagrad())
.setValidation(Trigger.everyEpoch, valRDD, Array(new Top1Accuracy[Float]),
         param.batchSize)
   .setEndWhen(Trigger.maxEpoch(2))
   .optimize()
```

Building an End-to-End Application with BigDL

With BigDL, users can build end-to-end AI applications using a single analytics pipeline based on Spark, including data management, feature management, feature transformations, model training and prediction, and results evaluation. We have worked with customers in different domains and developed end-to-end solutions using BigDL for fraud detection and defect detection, to name a

couple. **Figure 2** illustrates an end-to-end image recognition and object detection pipeline built using BigDL on Spark, which collects and processes large volumes of images from manufacturing pipelines and automatically detects product defects from these images (using convolutional neural network models on BigDL).



2 End-to-end image recognition and object detection pipeline

Making Deep Learning Accessible

In this article, we discussed BigDL, an open source distributed deep learning framework for Apache Spark. BigDL makes deep learning more accessible to big data users and data scientists by allowing users to write their deep learning applications as standard Spark programs, and to run these deep learning applications directly on top of existing Spark or Hadoop clusters. As a result, it makes Hadoop/Spark a unified platform for data storage, data processing and mining, feature engineering, traditional machine learning, and deep learning workloads, which can provide better economy of scale, higher resource utilization, ease of use/development, and better TCO.

GET STARTED
GET BIGDL FROM GITHUB* ›


intel
Software
OPTIMIZE YOUR CODE
FOR TODAY AND TOMORROW
Technical Webinars
Spring 2017
Create faster applications—faster.
Watch our free webinars on:
• Optimized, vectorized code
• High-performance Python*
• And lots more
Learn more and register ›
Wednesday, May 3
9:00 to 10:00 a.m. PDT
Healthy, Happy Performing Clusters with Intel® Cluster Checker 2017
Register ›
Wednesday, May 10
9:00 to 10:00 a.m. PDT
Snapshot Your Application Performance and Improve!
Register ›
Wednesday, May 17
9:00 to 10:00 a.m. PDT
HPC Applications Need High-Performance Analysis
Register ›
Wednesday, May 24
9:00 to 10:00 a.m. PDT
Parallel STL: Boosting Application Performance with Standard C++ Algorithms
Register ›
Wednesday, May 31
9:00 to 10:00 a.m. PDT
Accelerate Application Performance with OpenMP* and SIMD Parallelism
Register ›
Wednesday, June 7
9:00 to 10:00 a.m. PDT
CPUs, GPUs, FPGAs: Managing the Alphabet Soup
with Intel® Threading Building Blocks
Register ›
Wednesday, June 14
9:00 to 10:00 a.m. PDT
High-Performance Machine Learning and Data Science
with Julia and Intel® Math Kernel Library
Register ›
Wednesday, June 21
9:00 to 10:00 a.m. PDT
Shipping your HPC Applications with Containers
Register ›
For more complete information about compiler optimizations, see our Optimization Notice.
Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.
© Intel Corporation

THE PARALLEL UNIVERSE