THE PARALLEL UNIVERSE

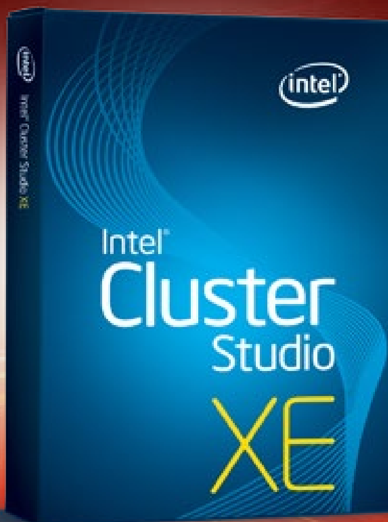
Issue 12
November 2012

Shedding Light on Cluster Performance with LAMPS

by Walter Shands

Walter Shands
Software Development Engineer

The efficient path to increased performance



Intel® Cluster Studio XE 2013 combines proven cluster and performance profiling tools with advanced threading and memory correctness analysis to boost HPC application performance and reliability.

- **Industry-leading Intel® MPI Library:** New levels of performance, scalability and flexibility for cluster applications on Intel® platforms
- **Intel® C, C++, and Fortran compilers:** Built-in optimization technologies and multithreading support
- **Intel® VTune™ Amplifier XE and Intel® Inspector XE:** Efficient thread and memory analysis
- **Intel® Cilk™ Plus and Intel® Threading Building Blocks:** Support open, standard parallel programming models

©2012, Intel Corporation. All rights reserved. Intel, the Intel logo, Cilk Plus, and VTune 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®
Software

CONTENTS

Letter from the Editor
When Complex Is the Baseline, BY JAMES REINDERS 4

Shedding Light on Cluster Performance with LAMMPS,
BY WALTER SHANDS 6

Highlights the features of Intel® Cluster Studio XE by using them to build and analyze LAMMPS (http://lammps.sandia.gov/), a complex cluster application and benchmark used in Spec MPI*.

Checklist for Programming Intel® Xeon Phi™ Coprocessors,
BY JAMES REINDERS 22

Offers key tips for programming a high degree of parallelism, while using familiar programming methods and the latest Intel® tools supporting the Intel® Xeon Phi™ coprocessor.

Advanced Vectorization, BY GEORG ZITZLSBERGER 28

Applying some of the vectorization techniques enabled by Intel compilers and their Intel® Cilk™ Plus technologies to an example application.

Optimizing Correlation Analysis of Financial Market Data Streams Using Intel® Math Kernel Library,
BY ZHANG ZHANG, ANDREY NIKOLAEV, AND VICTORIYA KARDAKOVA 40

Demonstrates the performance advantages of Intel® Math Kernel Library in the implementation of the online noise filtration algorithm on a correlation analysis of financial market data.

© 2012, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Core, Intel Inside, Cilk, Pentium, VTune, VPro, Xeon and Xeon Phi 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.

Sign up for future issues | Share with a friend

The Parallel Universe is a free quarterly magazine. **Click here** to sign up for future issue alerts and to share the magazine with friends.

When COMP Is the Baseline

LETTER FROM THE EDITOR

James Reinders Director of Parallel Programming Evangelism at Intel Corporation. James is a co-author of a new book, *Structured Parallel Programming*, from Morgan Kaufmann, 2012. His other books include *Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism*, available in English, Japanese, Chinese, and Korean.

Complexity is every day in high performance and cluster applications. Applications are utilized to solve the increasingly complex problems that we pose. Parallelism is the natural vocabulary for developers trying to ensure that large data stores and demanding workloads run fast, run flawlessly, and helps us get the results we seek.

The conjunction of Intel® Cluster Studio XE 2013 and the Intel® Xeon Phi™ coprocessor makes this an ideal time to explore some of our more interesting development challenges.

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

PLEX

We start by seeing Intel Cluster Studio XE 2013 in action—in *Using Intel® Software Development Tools to Analyze the Performance of LAMMPS* —as we use new features of this advanced toolset to build and analyze LAMMPS, one of the Spec MPI* benchmarks.

Next, in *Checklist for Programming Intel® Xeon Phi™ Coprocessors*, we offer programming tips for applications running on and taking advantage of the capabilities of Intel® Xeon Phi™ coprocessors. Scaling, vector usage, and memory usage can all be improved, and these benefits are preserved when the applications run on Intel® Xeon® processors.

Advanced Vectorization uses an example application to present practical, proven techniques enabled by Intel compilers and their Intel® Cilk™ Plus technologies.

And finally, *Optimizing Correlation Analysis of Financial Market Data Streams Using Intel® Math Kernel Library*, is at once a case study of extreme complexity—applying an online noise filtration algorithm to correlation analysis in the finance industry—and a proof of the performance gains possible with Intel® Math Kernel Library.

We all live in an increasingly complexity environment—and I hope you are inspired by seeing some ways others are tackling their complex challenges.

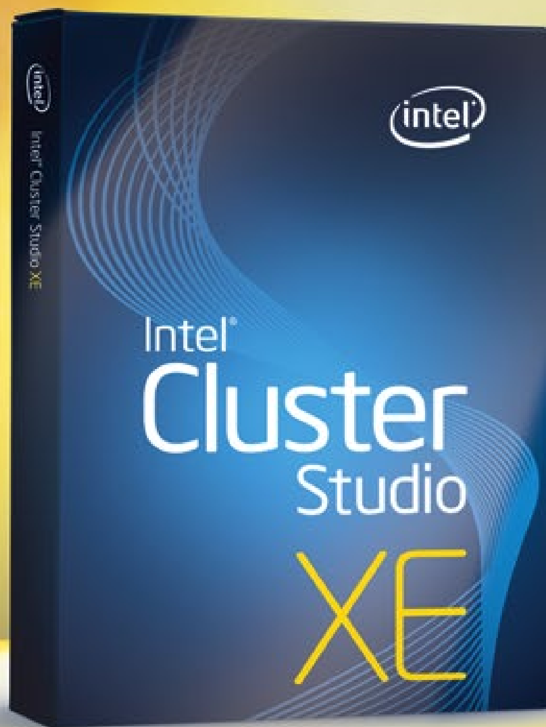
James Reinders
November 2012

Sign up for future issues | Share with a friend

Using Intel® Software Development Tools to Analyze the Performance of LAMMPS

by Walter Shands,
Software Development Engineer, Intel

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.


intel
Intel® Cluster Studio XE
Intel®
Cluster
Studio
XE

Sign up for future issues | Share with a friend

Introduction

This article highlights the features of Intel® Cluster Studio XE by using them to build and analyze LAMMPS (http://lammps.sandia.gov/), a benchmark used in Spec MPI. We will describe build settings for the Intel® C++ Compiler that optimize performance and how to use the Intel® MPI message library to deliver best-in-class performance for LAMMPS on Intel® architecture-based clusters. We will use Intel® Trace Analyzer and Intel® Trace Collector to illuminate the use of MPI APIs that cause performance problems in LAMMPS, and show how to compare trace files with the Intel Trace Analyzer GUI to get detailed analysis of message passing with aligned timelines. We will also show how to use the Intel® MPI correctness checking library to look for MPI coding errors. Additionally, we will show how to use Intel® VTune™ Amplifier XE to visualize application scaling on individual nodes.

The techniques described in this article may be applied to similar types of complex cluster applications by using diverse technology such as MPI and OpenMP* across multiple machines.

Our objective is to build LAMMPS, analyze the application with respect to MPI API performance and scaling on individual nodes, and make it run faster. In general, we will want to make sure we use an optimizing compiler like the Intel® C++ Compiler, that compiler settings are optimized for the architecture the application will run on, and that we are using an optimized MPI implementation like Intel MPI. In addition, we want to make sure MPI API call time is a reasonably small percentage of total computation time, and that individual MPI API calls are independent across ranks so MPI call time is minimized. We will also want to ensure that the application running on each node scales to take advantage of all of the CPU cores, so that we use CPU resources efficiently. Intel Cluster Studio XE 2013 provides the tools to build faster MPI applications and analyze MPI application performance.

According to LAMMPS documentation, "LAMMPS is a classical molecular dynamics code that models an ensemble of particles in a liquid, solid, or gaseous state. It can model atomic, polymeric, biological, metallic, granular, and coarse-grained systems using a variety of force fields and boundary conditions."

Building LAMMPS with the Intel C++ Compiler and Intel MPI Library

To build LAMMPS we created a custom make file by copying one of the provided make files and editing the contents so the LAMMPS build used the Intel C++ Compiler and the Intel MPI Library. Here are some of the settings **(Fig. 1)**:

`mpiicpc` is the Intel-specific command for building an application using the Intel C++ Compiler and the Intel MPI Library. The MPI Library focuses on making applications perform better on IA-based clusters by implementing the high-performance MPI-2 specification on multiple fabrics. It enables you to quickly deliver maximum end-user performance, even if you change or upgrade to new interconnects, without requiring major changes to the software or operating environment.

To analyze MPI message traffic in an application with Intel Trace Analyzer, we need to collect data into trace files. Intel Trace Collector for MPI applications produces trace files that can be analyzed with the Intel Trace Analyzer performance analysis tool. It records all calls to the MPI library and all transmitted messages, and allows arbitrary user-defined events to be recorded. Instrumentation can be switched on or off at runtime, and a powerful filtering mechanism helps to limit the amount of the generated trace data.

Intel Trace Collector is an add-on for existing MPI implementations; using it merely requires relinking the application with the Intel Trace Collector profiling library. This will enable the tracing of all calls to MPI routines, as well as all explicit message passing. On some platforms, calls to user-level subroutines and functions will also be recorded.

We can use a fully optimized build and capture a trace that will allow us to drill down to source code with Intel Trace Analyzer. The same is true of Intel VTune Amplifier XE, so we use full optimizations in the compiler with the `–O3` switch. We want to use the most advanced instruction set for our target nodes, which in our case supports SSE4.2, so we include the –SSE4.2 switch since the default for the compiler is SSE2.

We need symbol information to drill down to source code with Intel VTune Amplifier XE and Intel Trace Analyzer, so we include the `–g` switch. A new feature in Intel VTune Amplifier XE 2013 is that it allows us to drill down to source code even if functions are inlined, so we don't need to include the –fno-inline-functions switch as was required for previous versions. Intel Trace Collector requires normal stack frames but the Intel® compiler does not use normal stack frames by default if optimization is enabled, so we must use -fno-omit-frame-pointer to enable the use of normal stack frames.

```
# compiler/linker settings
# specify flags and libraries needed for your compiler

CC =          mpiicpc
CCFLAGS =     -g –O3 -xSSE4.2 -fno-alias -fno-omit-frame-pointer -vec-report5 -opt-report3
-opt-report-phase=all
DEPFLAGS =    -M
LINK =        mpiicpc
LINKFLAGS =   -O
LIB =         -lstdc++ -lpthread -liomp5
ARCHIVE =     ar
ARFLAGS =     -rc
SIZE =        size
```

Figure 1

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

Running a LAMMPS Benchmark

LAMMPS runs by reading commands from stdin, and you can write an input script of commands to set up and simulate particle dynamics. There is a set of input scripts for tests and benchmarks provided in the LAMMPs distribution. We used the script for a standard Lennard-Jones benchmark.

This is an example of running LAMMPS with a Lennard-Jones benchmark script **(Fig. 2)**:

```
> mpirun -f ~/cluster.hosts -trace -genv VT_PCTRACE 4 –perhost 1 -bootstrap slurm -n8
~/LAMMPS/lammps-25Jul12/src/lmp_walt -var x 2 -var y 2 -var z 2 < in.lj
```

Figure 2

`mpirun` is the command used to launch an MPI job. Here it runs the LAMMPS application on the nodes listed in the cluster.hosts file; in this case my build is called "lmp_walt". The `–trace` switch causes Intel Trace Collector to profile the LAMMPS application by preloading the Intel Trace Collector library, and the environment variable `VT_PCTRACE` sets the call stack depth.

The `–bootstrap slurm` option selects a built-in bootstrap server to use, which is the basic remote node access mechanism provided on the nodes, and `–n 8` says to use eight processes in the run. The `–perhost 1` option is important: it says to create 1 MPI process on each node in a round-robin fashion. Without it, an MPI process is started for every core available on the first node. For example, if the nodes each have 12 cores and we use the `–n 8` option, but not the `–perhost` option (or equivalently the `–rr` option), then eight MPI processes would be started on one node.

The `–var x 2 –var y 2 –var z 2` LAMMPS flags are for scaling the problem 2x in each x, y, and z direction, where the data to work on is arrayed as a grid of x * y * z 3d subdomains. We set the variables to two so that there is a 3D subdomain assigned to each of the eight processors used.

We created eight MPI processes to run in total, each on a separate node for our first run of the Lennard-Jones benchmark, and analyzed the trace with Intel Trace Analyzer.

Analyzing MPI Communications

When you develop MPI applications it is important to look at how much time the MPI calls are consuming compared to application time and the load balance for each MPI call. Intel Trace Analyzer provides this data through various charts and profiles.

The Flat Profile

The Flat Profile view tells us how much time user code took and can tell us how much time each MPI API took during the application run. It shows that the total amount of time taken by MPI calls over the whole run of the program is fairly significant compared to application time **(Fig. 3)**, and we can use Intel Trace Analyzer to see more details.

Ungrouping MPI reveals that the MPI_Send and MPI_Wait calls are the most expensive in terms of time **(Fig. 4)**.

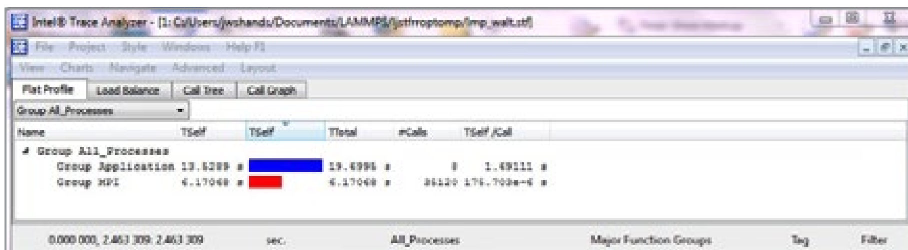


Figure 3

The Load Balance Chart

In addition, MPI API call time and application time are not well balanced across nodes, as shown in the Load Balance chart **(Fig. 5)**.

If we look at the MPI function time in finer detail, there are varying times for individual MPI calls across the compute nodes, with most of the time taken on the 0, 4, 5, and 9 compute nodes. This can be an indication of message dependencies that result in reduced application performance **(Fig. 6)**. The Event Chart provides information on whether MPI API calls are interdependent.

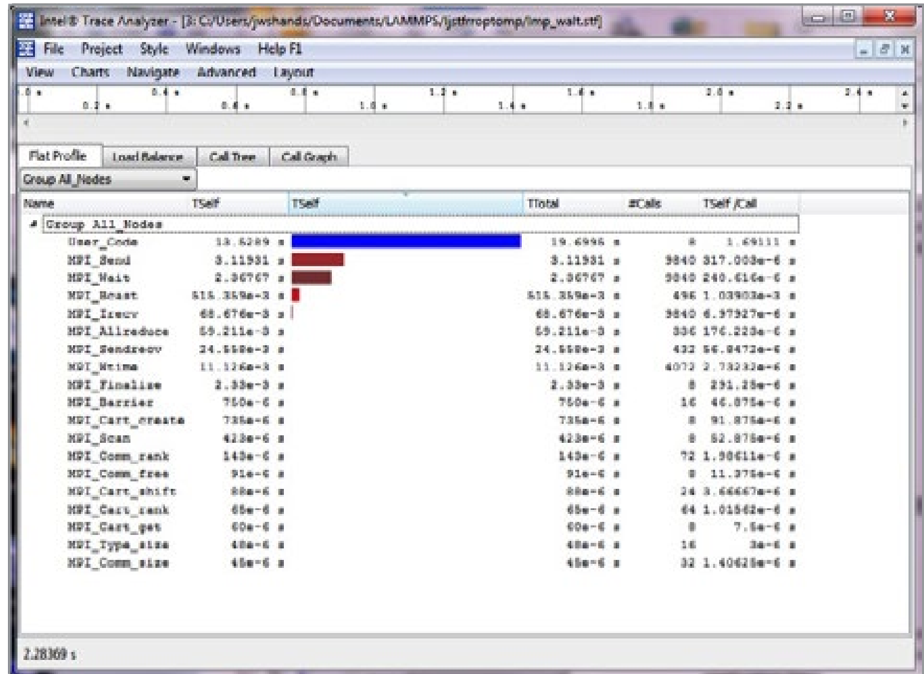


Figure 4

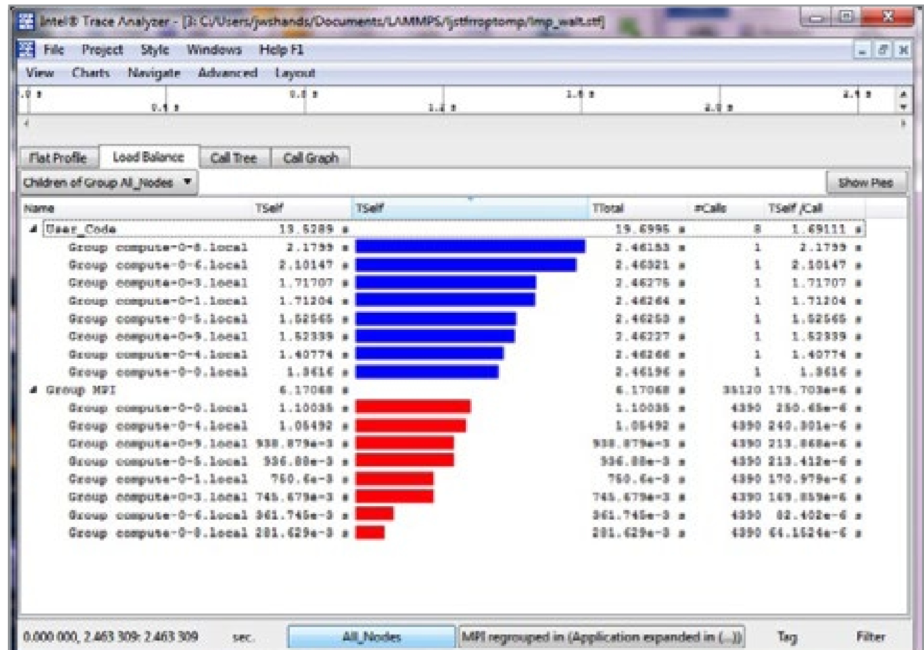


Figure 5

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

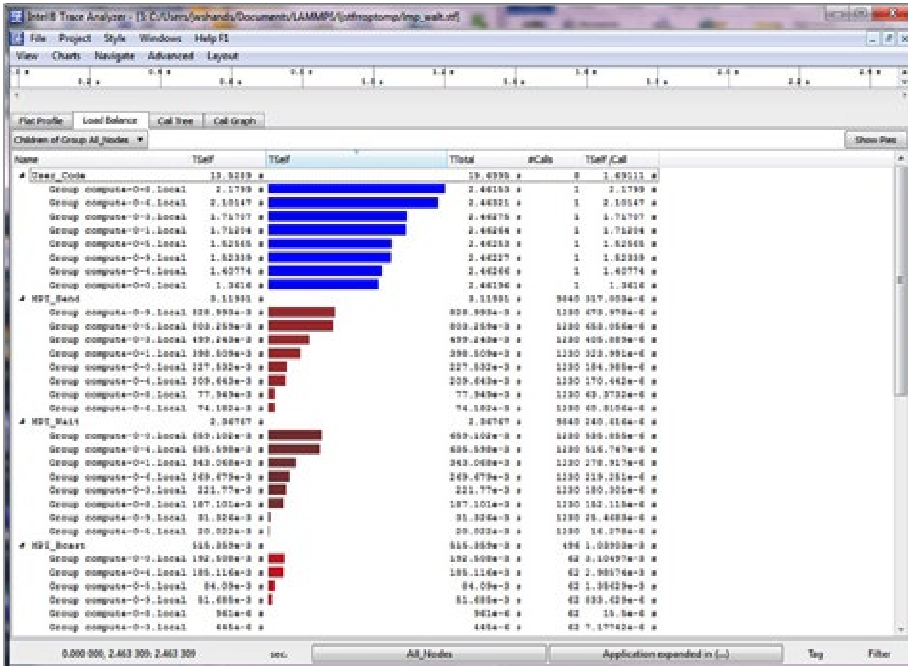


Figure 6

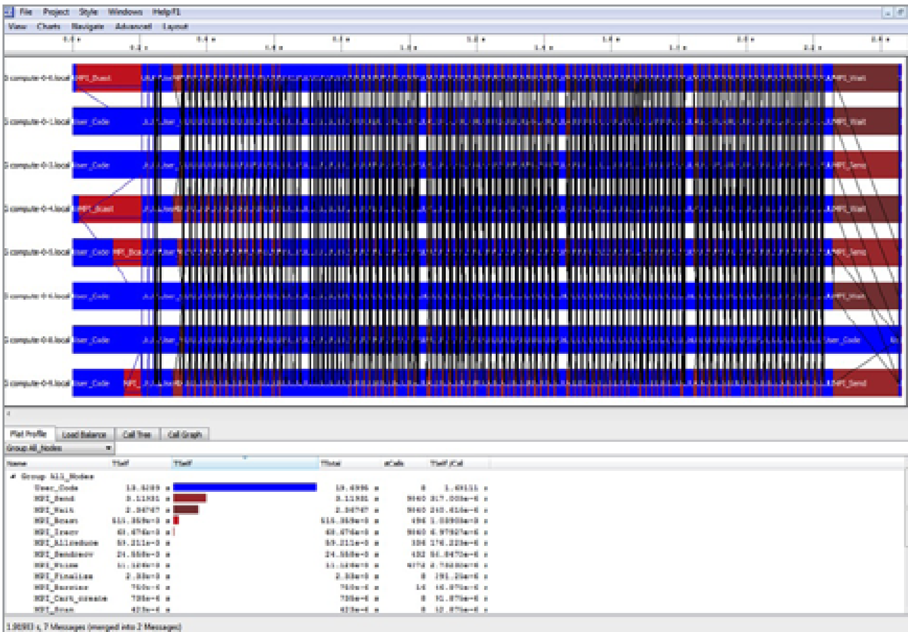


Figure 7

The Event Chart

If we open the Event Chart from the Chart menu, it shows us a timeline and horizontal bars with segments representing application code and MPI messages. This gives us an idea of which message APIs were called at a particular time and how long they took to complete. **(Fig. 7)**

LAMMPS documentation states "For computational efficiency, LAMMPS uses neighbor lists to keep track of nearby particles ... On parallel machines, LAMMPS uses spatial-decomposition techniques to partition the simulation domain into small 3D subdomains, one of which is assigned to each processor. Processors communicate and store "ghost" atom information for atoms that border their sub-domain .... The Comm class performs interprocessor communication, typically of ghost atom information. This usually involves MPI message exchanges with six neighboring processors in the 3D logical grid of processors mapped to the simulation box."

Our input script called for 100 time steps with neighbor lists of particles rebuilt after 20 time steps. The communication of atom information occurs at each time step, and the Event Chart allows us to see the communication graphically. We can see the five groups of 20 atom information exchanges show up as black lines.

If we zoom into the event chart, we can get details on the message passing calls. Here is the zoom into one of the five groups **(Fig. 8)**.

A further zoom into an atom information exchange provides a look at the individual MPI API calls over time. The black lines indicate which ranks or processes exchanged messages, and there are two distinct time periods within a time step where messages are exchanged **(Fig. 9)**. The messages between ranks look fairly independent, since we do not see a stair-step pattern, which is indicative of message interdependence—although some of the MPI APIs in the first message exchange sequence take quite a bit longer to complete than the same MIP APIs in the second message exchange sequence during this application time step.

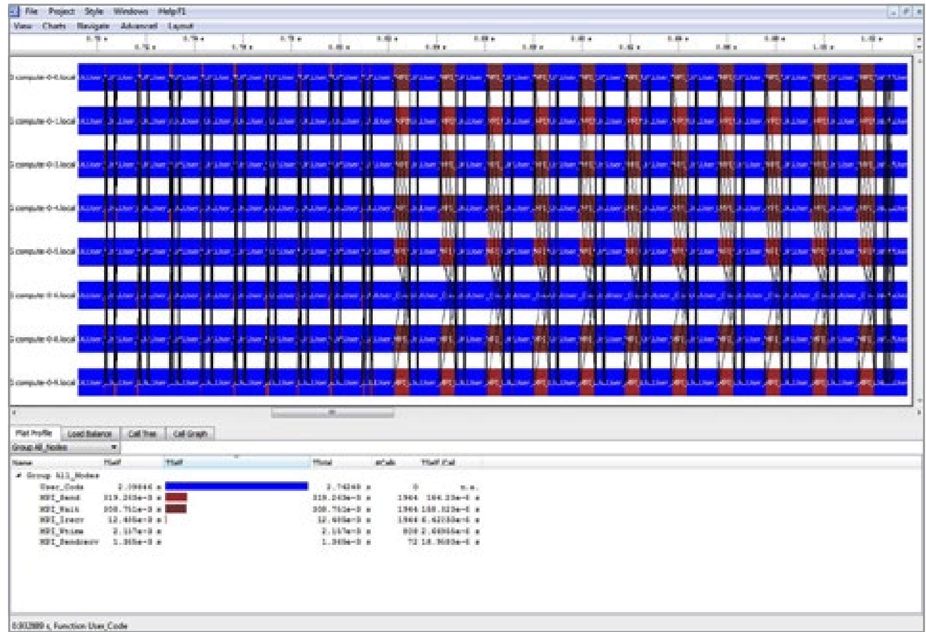


Figure 8

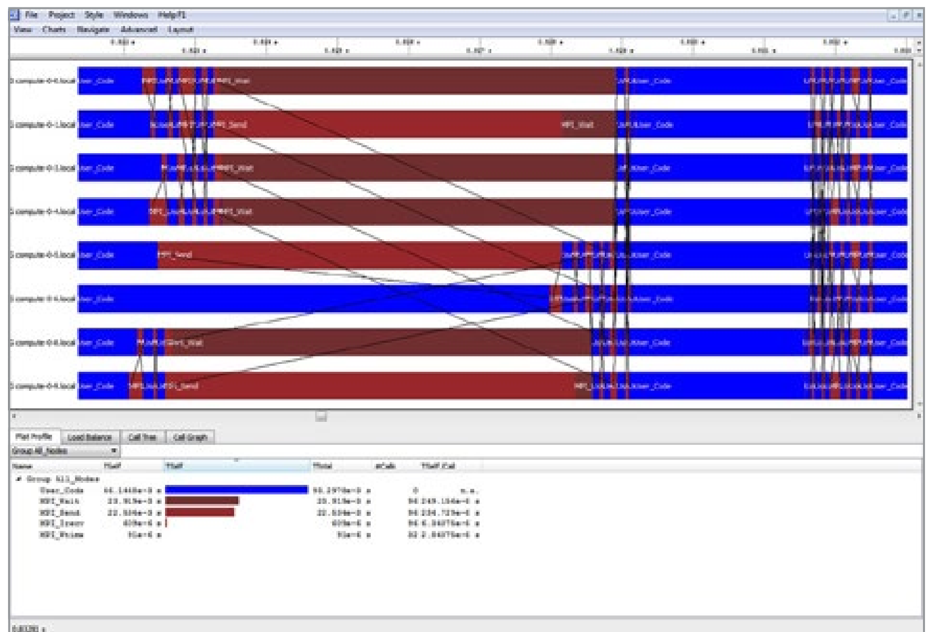


Figure 9

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

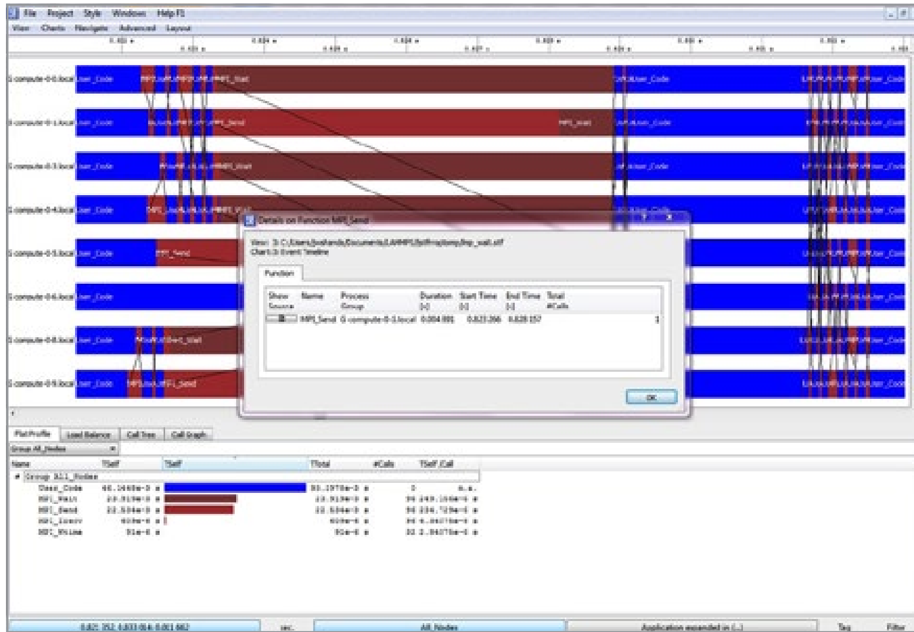


Figure 10

If we want to get information on a message or MPI API call, we can simply right-click on a black message line or on the MPI call segment to bring up the message details dialog **(Fig. 10)**.

If we want to look at the source code where the MPI call originated and we have debug symbols, we can click on the Show Source button in the dialog box to display the location of the MPI call and the call stack **(Fig. 11)**. In our case, this confirms that in the first message exchange sequence the Comm class is calling reverse_comm to exchange ghost atom information or, more specifically, forces on ghost atoms are communicated and summed back to their corresponding owned atoms.

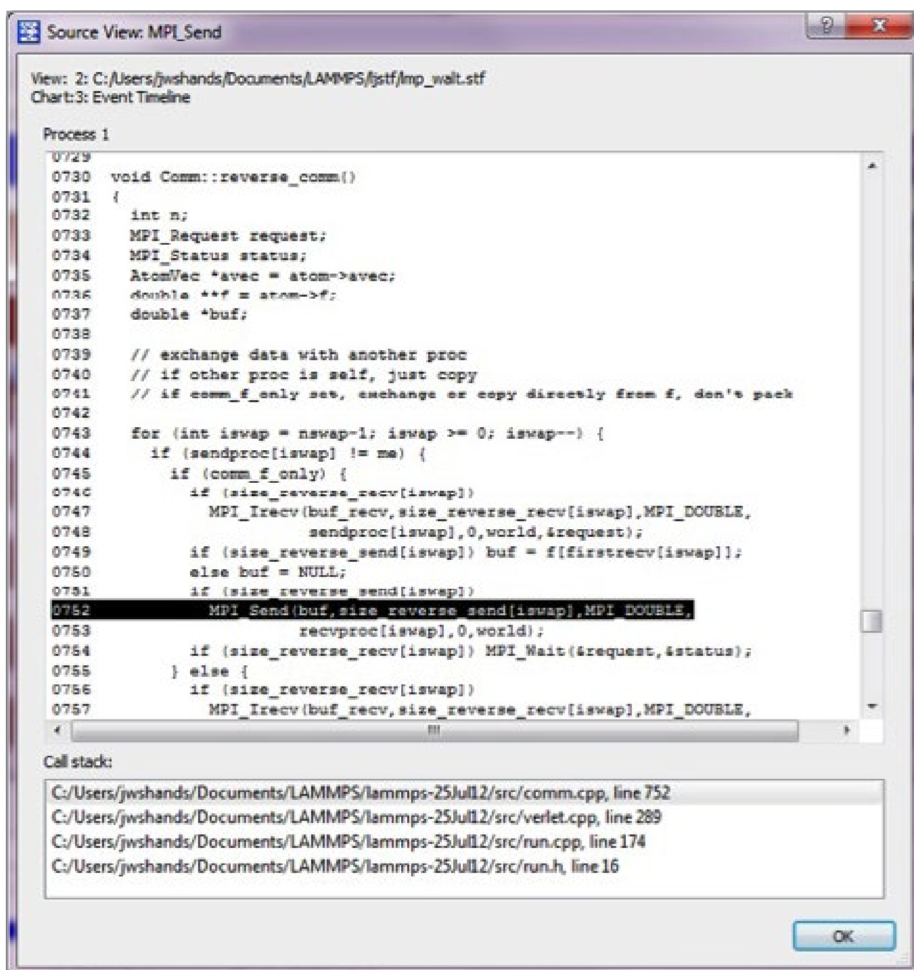


Figure 11

The same source code analysis confirms that the second message exchange sequence is where the Comm class calls forward_comm to distribute coordinates of ghost atoms to each process. This is done at each time step **(Fig. 12)**.

The Qualitative Timeline

Another metric we may want to examine is the data volume that is being transmitted with each message over time. To see this, we use the Qualitative Timeline, selected from the Chart menu, and right-click on the window to select Messages and Data Volume from the pop-up menu as the items to display **(Fig. 13)**.

The data volume per message is fairly consistent over time—with a maximum at about 93 KB. Additionally, we can see the duration and transfer rate by picking those attributes for the display.

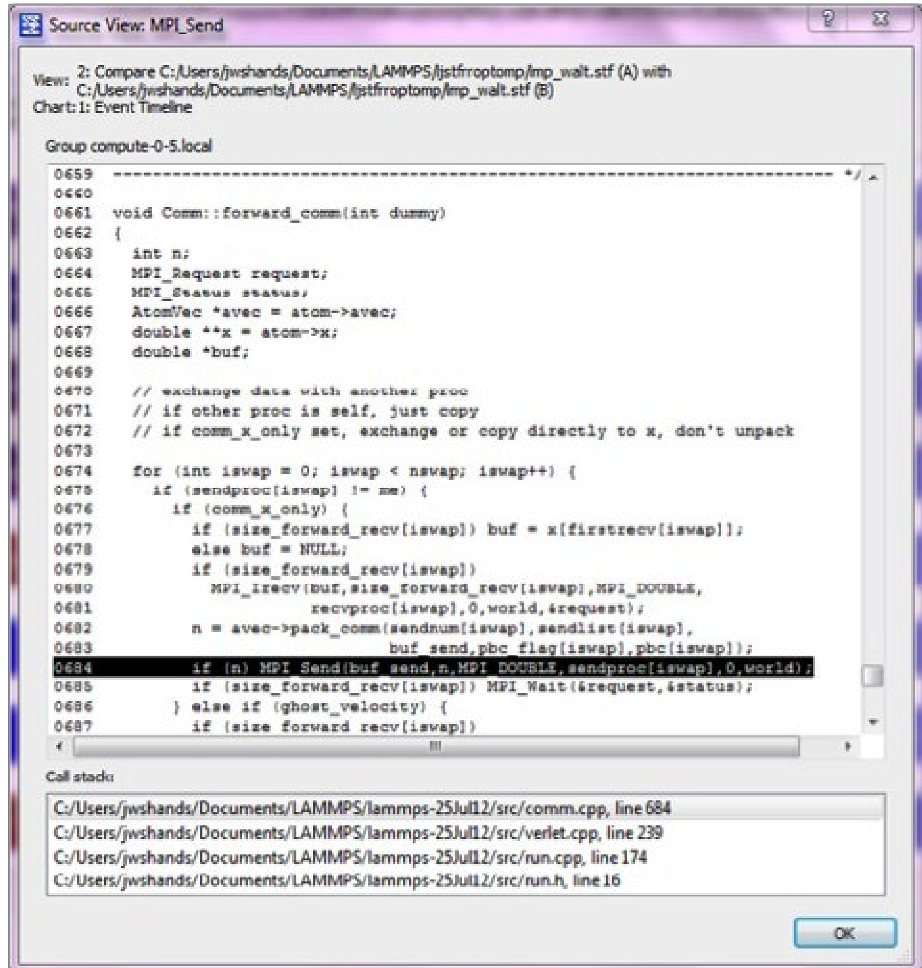


Figure 12

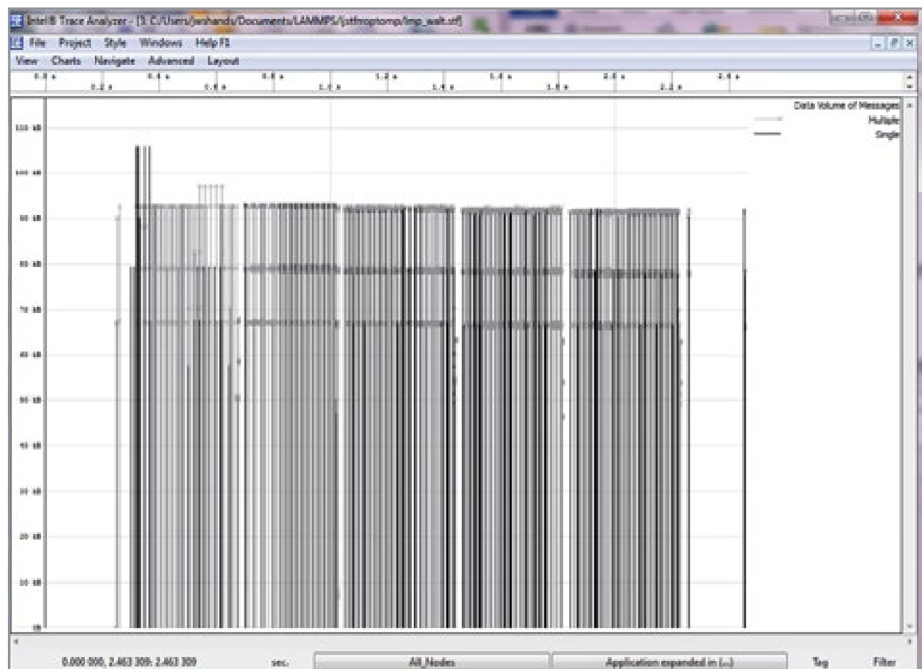


Figure 13

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

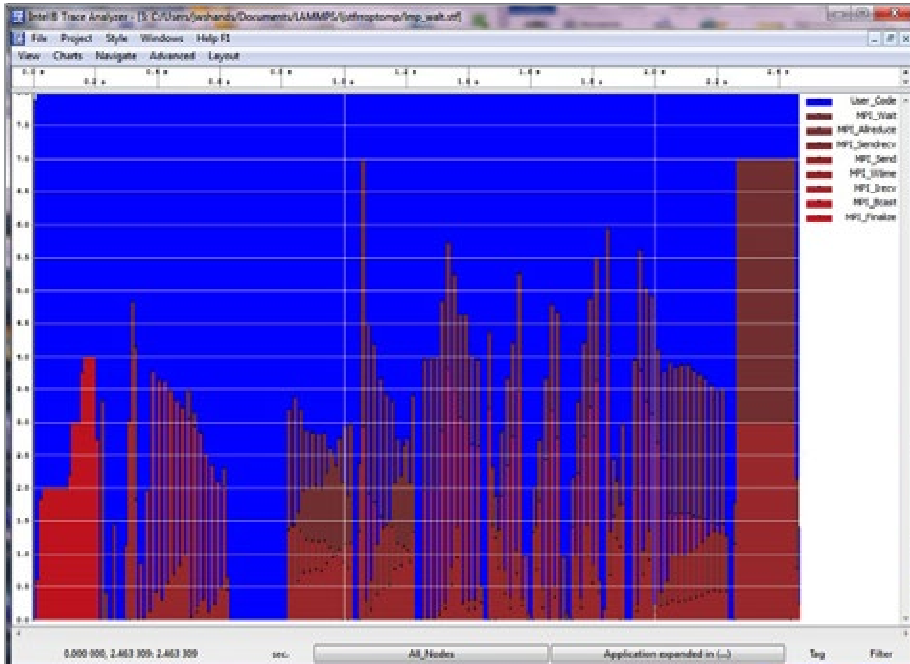


Figure 14

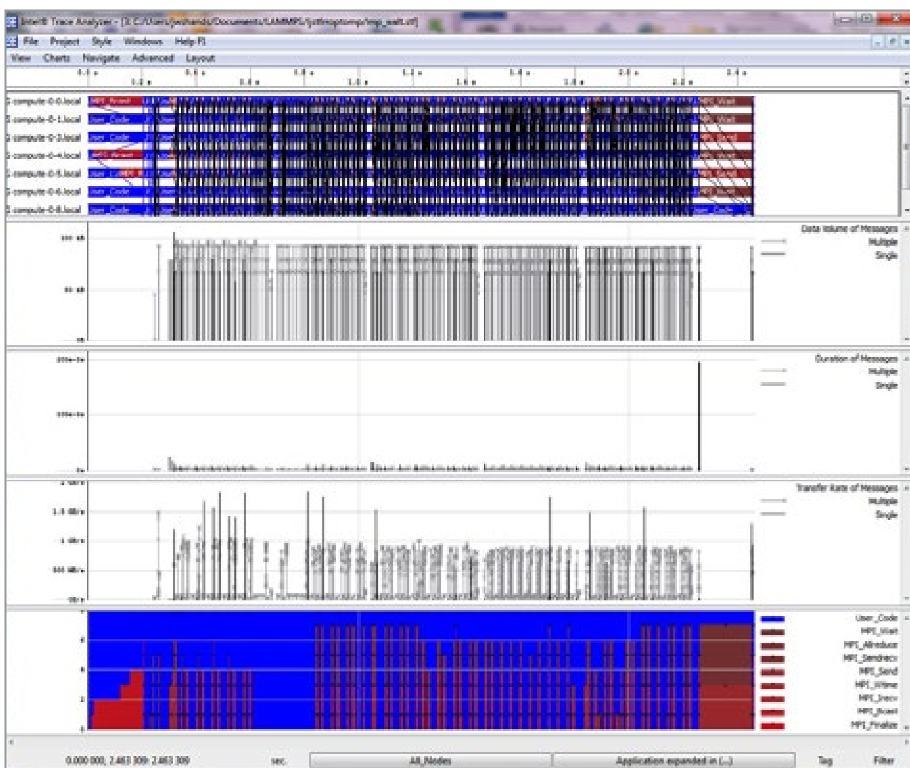


Figure 15

The Quantitative Timeline

The Quantitative Timeline gives an overview of the parallel behavior of the application. It shows over time how many processes or threads are involved in which function. Along the time axis, the different functions are presented as vertically stacked color bars. The height of these bars is proportional to the number of processes that are currently within the respective function **(Fig. 14)**.

There is also the option to view all of these profiles and charts in a single window working on a common timeline, so patterns and relationships in the data are easier to see **(Fig. 15)**.

We can also couple the mouse zoom and navigation keys for all of the displayed charts, so zooming in by highlighting a section of the timeline provides a detailed view of the data described above over the same time interval **(Fig. 16)**.

The Message Profile Chart

To get another view on messages we can use the Message Profile Chart, which categorizes messages by groupings in a matrix and shows the value of several attributes in each cell. By default, the matrix is square with the sending processes as row labels and the receiving processes as column labels. It shows in cell (i, j) the total time spent in transferring messages from sender i to receiver j. This chart also includes per row and per column statistics, which give the sum, the average, and the standard deviation for the respective row or column **(Fig. 17)**.

The chart shows that the most time-consuming messages originated with compute nodes 0, 1, 3, and 4 sending messages to compute nodes 5, 6, 8, and 9 respectively, while messages sent in the opposite direction are not a performance problem.

MPI Message Correctness Checking

During development of an MPI application you will need to check for errors. You can do this with the correctness checking feature in Intel Trace Collector by replacing the mpirun command –trace switch with –check. This will not generate a trace file, but will report errors to stdout. Errors include MPI local memory errors, message data type mismatches, message corruption, pending messages, deadlocks, invalid parameters, and others.

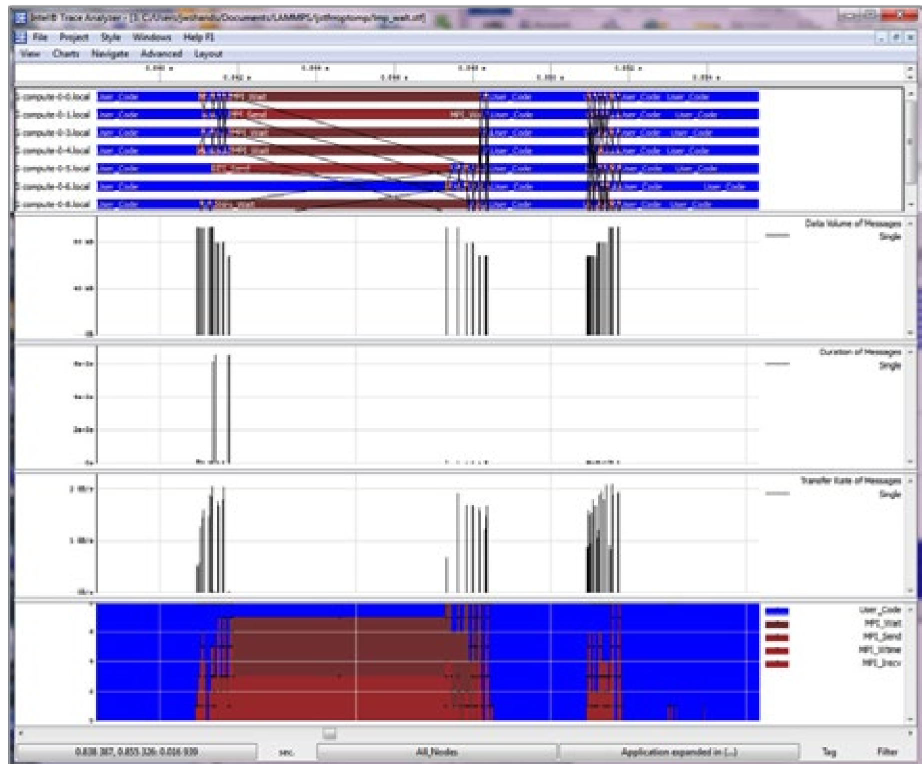


Figure 16

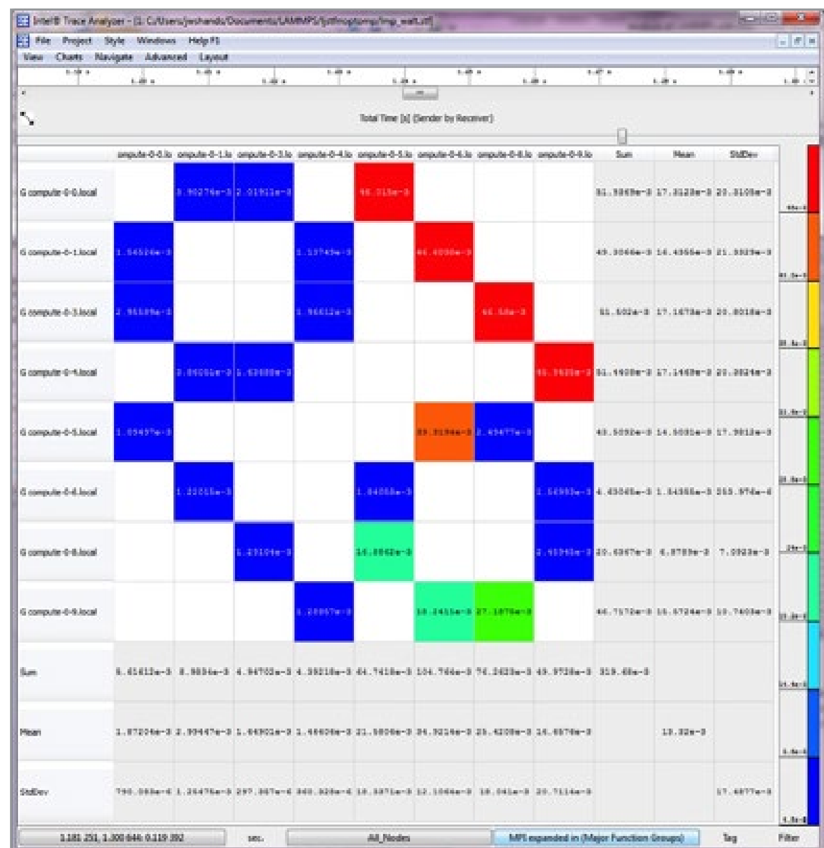


Figure 17

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

Here is an example of how the correctness checking feature was used to check LAMMPS and reported no errors: **(Fig. 18)**:

```
~/LAMMPS/lammps-25Jul12/bench]> mpirun -f ~/cluster.hosts -check -perhost 1 -bootstrap
slurm -n 8 ~/LAMMPS/lammps-25Jul12/src/lmp_walt -var x 2 -var y 2 -var z 2 < in.lj
```

Figure 18

Application Scaling and Performance Profiling with Intel VTune Amplifier XE

It is important to balance the scalability of each MPI process with the number of MPI processes started on each node. For example, if we have an MPI application that scales to 12 cores and have 12 cores on each node, then we can start one MPI process per node and utilize all the CPUs effectively.

To check this on LAMMPS, we run Intel VTune Amplifier XE 2013 and a single LAMMPS process on each node. This allows us to collect performance data on each MPI process to see how well LAMMPS utilizes all of the cores on the node.

Here is the command line to collect hotspot data for the LAMMPS process on each node using Intel VTune Amplifier XE **(Fig. 19)**.

```
~/LAMMPS/lammps-25Jul12/bench]> mpirun -f ~/cluster.hosts –perhost 1 -bootstrap slurm -n 8
amplxe-cl -collect hotspots ~/LAMMPS/lammps-25Jul12/src/lmp_o3 -var x 2 -var y 2 -var z 2 < in.lj
```

Figure 19

Essentially, we are using the mpirun command to run Intel VTune Amplifier XE, which in turn launches an instance of LAMMPS. Intel VTune Amplifier XE will run on each node and collect data for the LAMMPS process. The results data is stored in separate folders for each node on the machine where the mpirun command was executed.

Intel VTune Amplifier XE results tell us that the application is using only one thread for computation, and that most of the run time is concentrated in a function called `PairLJCut::compute` in file `pair_lj_cut.cpp`. **(Fig. 20)**

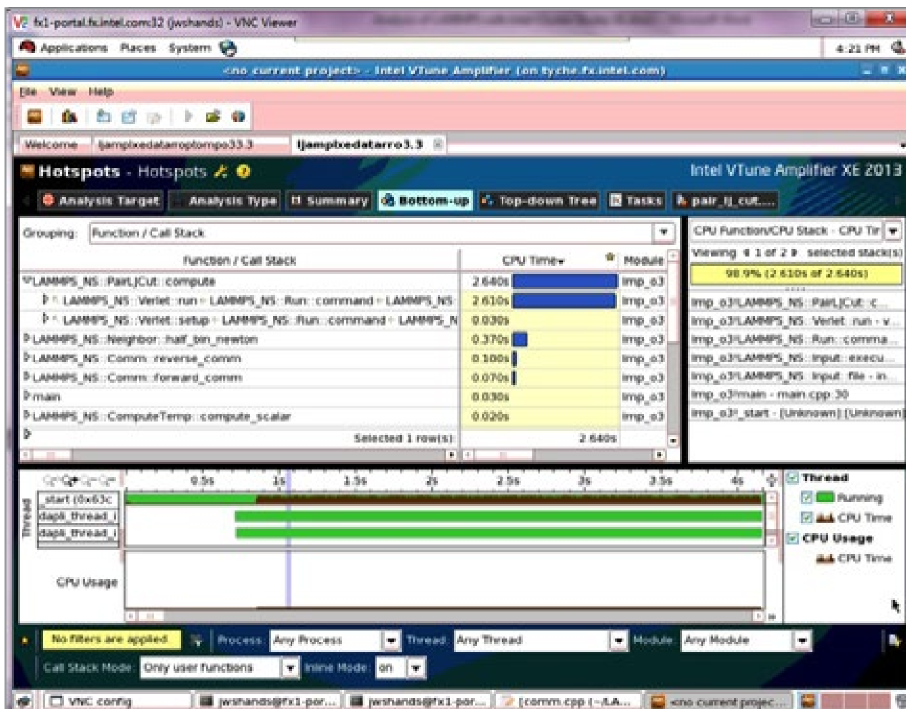


Figure 20

The runtime was about 3.15 seconds in default mode **(Fig. 21)**.

```
mpirun -f ~/cluster.hosts -perhost 1 -bootstrap slurm -n 8 ~/LAMMPS lammps-
25Jul12/src/lmp_o3 -var x 2 -var y 2 -var z 2 < in.lj
LAMMPS (25 Jul 2012)
 using 24 OpenMP thread(s) per MPI task
Lattice spacing in x,y,z = 1.6796 1.6796 1.6796
Created orthogonal box = (0 0 0) to (67.1838 67.1838 67.1838)
 2 by 2 by 2 MPI processor grid
Created 256000 atoms
Setting up run . . .
Memory usage per processor = 55.2672 Mbytes
Step Temp E_pair E_mol TotEng Press
     0      1.44   -6.7733681      0   -4.6133765   -5.019674
   100   0.75865617   -5.7603259      0   -4.6223461   0.19586104
Loop time of 3.14944 on 192 procs (8 MPI x 24 OpenMP) for 100 steps with 256000 atoms
```

Figure 21

Next, we want to include threading by using an optimization package provided in the LAMMPS distribution to see if performance improves. The amount of scaling we get will determine how many processes we will start on each node to get optimal use of the computational resources.

We include an optimization package that provides threading via OpenMP called USER-OMP.

Intel VTune Amplifier XE shows multiple threads of execution with significant CPU activity **(Fig.22)**.

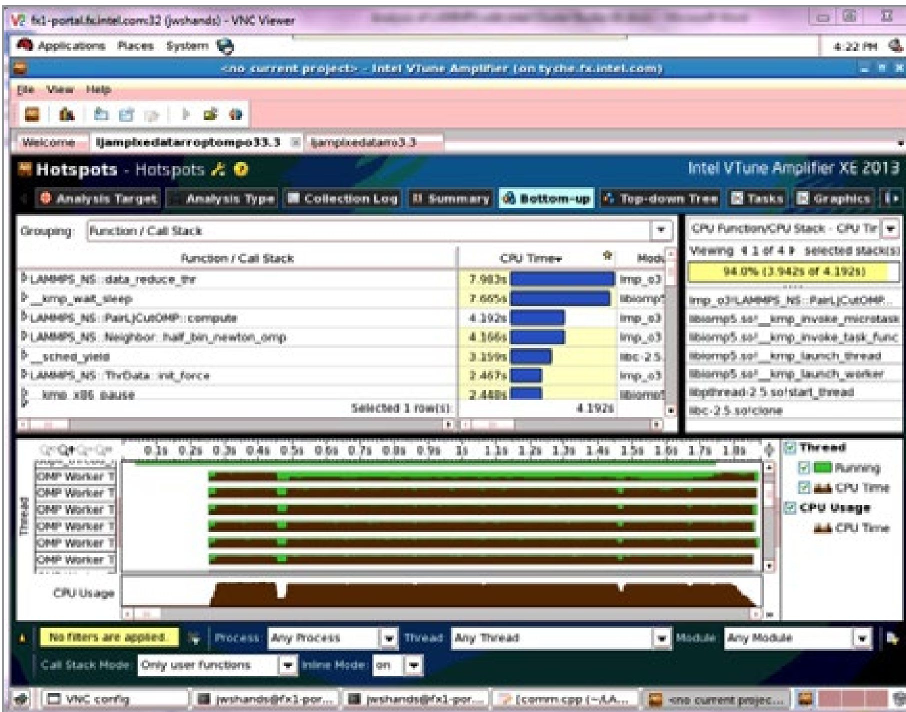


Figure 22

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

Intel VTune Amplifier XE Thread Concurrency data shows that the application threads are consuming significant CPU time over the time of the run **(Fig. 23)**.

The histograms below **(Fig. 24)** show threads and CPUs are running simultaneously most of the time, indicating the application is utilizing computational resources effectively.

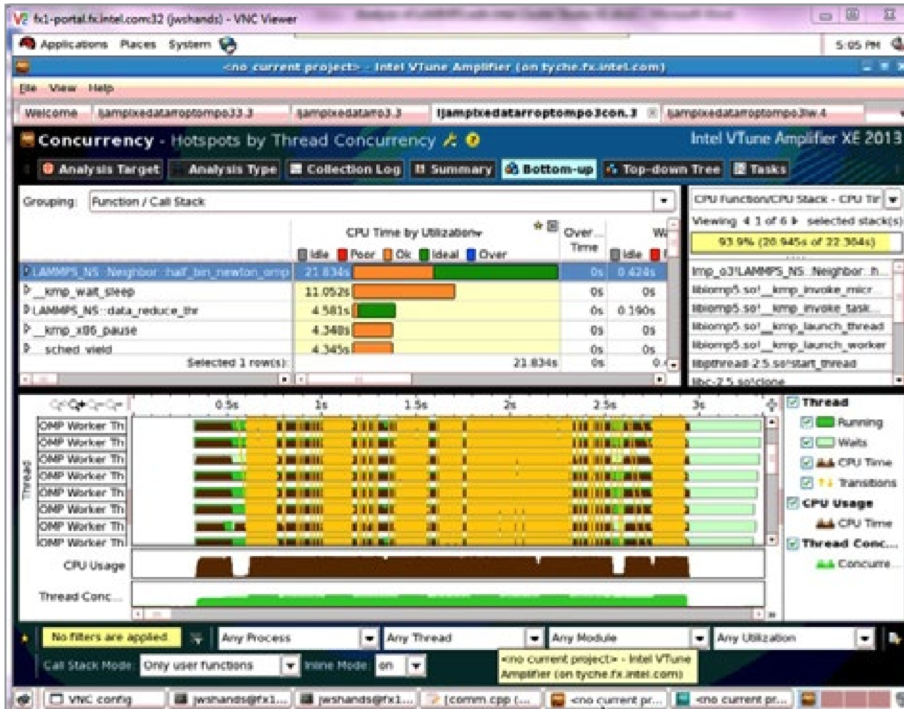


Figure 23

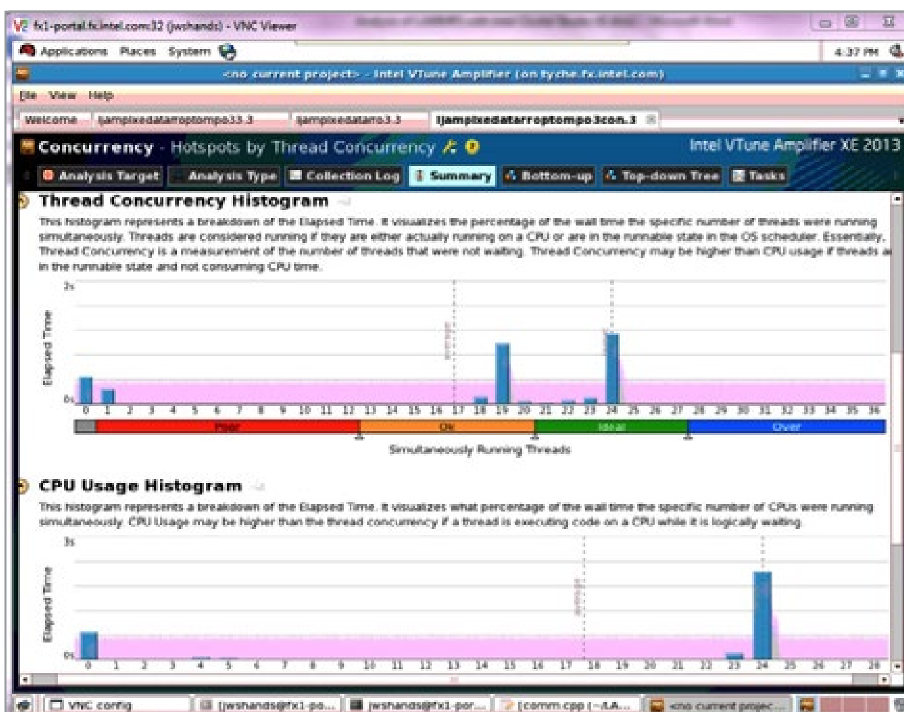


Figure 24

Sign up for future issues | Share with a friend

The runtime was about 1.20 seconds with the optimization package, much less than without the package, even though there was a significant amount of OpenMP wait time introduced **(Fig. 25)**.

```
> mpirun -f ~/cluster.hosts -perhost 1 -bootstrap slurm -n 8 ~/LAMMPS/lammps-
25Jul12/src/lmp_o3 -sf omp -var x 2 -var y 2 -var z 2 < in.lj
LAMMPS (25 Jul 2012)
 using 24 OpenMP thread(s) per MPI task
Lattice spacing in x,y,z = 1.6796 1.6796 1.6796
Created orthogonal box = (0 0 0) to (67.1838 67.1838 67.1838)
 2 by 2 by 2 MPI processor grid
Created 256000 atoms
Last active /omp style is pair_style lj/cut/omp
Setting up run ...
Memory usage per processor = 59.4847 Mbytes
Step Temp E_pair E_mol TotEng Press
    0      1.44   -6.7733681      0   -4.6133765   -5.019674
   100  0.75865617   -5.7603259      0   -4.6223461  0.19586104
Loop time of 1.19531 on 192 procs (8 MPI x 24 OpenMP) for 100 steps with 256000 atoms
```

Figure 25

Memory and Threading Error Checking of MPI Processes with Intel® Inspector XE

We can also use Intel® Inspector XE to check for threading errors or memory errors in an MPI application. Here is the command line we used to check for memory errors and to pinpoint their location **(Fig. 26)**.

```
> mpirun -f ~/cluster.hosts -perhost 1 -bootstrap slurm -n 8 inspxe-cl -collect mi3
-r=~/LAMMPS/lammps-25Jul12/bench/ljinspxe/ljinspxedatarroptompo3con5
~/LAMMPS/lammps-25Jul12/src/lmp_o3 -sf omp -var x 2 -var y 2 -var z 2 < in.lj
```

Figure 26

Data results are collected and stored for each node in a manner similar to Intel VTune Amplifier XE. We can open the results in the Intel Inspector XE GUI just as we did for Intel VTune Amplifier XE results.

Support of Manycore Architectures: Intel® Xeon Phi™ Coprocessors

One of the newest additions to high performance computing is the Intel® Xeon Phi™ coprocessor, which is designed to provide efficient performance for highly parallel applications. Common programming models for Intel® Xeon processors extend to Intel Xeon Phi coprocessors—so as developers embrace high degrees of parallelism, they don't need to rethink the entire problem.

The practical result of this, in our case, is that it is very easy to run the LAMMPS application on an Intel Xeon Phi coprocessor. All we have to do if we are using the Intel C/C++ Compiler is to rebuild LAMMPS with the –mmic switch for both the compiler and linker. Here is the relevant part of the makefile **(Fig. 27)**:

```
# compiler/linker settings
# specify flags and libraries needed for your compiler

CC =        mpiicpc
CCFLAGS =      -mmic -g –O3 -openmp -fno-alias-fno-omit-frame-pointer

DEPFLAGS =    -M

LINK =      mpiicpc
LINKFLAGS =   -mmic
LIB =       -lstdc++ -lpthread -liomp5
…
```

Figure 27

 For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

You will also need to have libiomp5.so and libVT.so on the coprocessor to use OpenMP and Intel Trace Collector.

There are a couple of choices on where to run the application. To run four LAMMPS processes on the host and eight LAMMPS processes on the Intel Xeon Phi coprocessor, use a command similar to the following. We use our original LAMMPS build for the host and the one built with the –mmic switch on the Intel Xeon Phi coprocessor **(Fig. 28)**.

```
$ mpiexec -host sc-mic -n 4 ../src/lmp_o3 -sf omp -var x 2 -var y 2 -var z 2 < in.lj : -host mic0 -n 8
~/lmp_mic -sf omp -var x 2 -var y 2 -var z 2 < in.lj
```

Figure 28

To run all eight LAMMPS processes on the Intel Xeon Phi coprocessor, use the following command line indicating the working directory on the coprocessor **(Fig. 29)**:

```
$ mpiexec -wdir ~/. -host mic0 -n 8 ~/lmp_mic -sf omp -var x 2 -var y 2 -var z 2 < in.lj
```

Figure 29

In addition, we can still use Intel Trace Collector to get MPI information from applications running on Intel Xeon Phi coprocessors, and the command line switch is the same **(Fig. 30)**.

```
$ mpiexec –trace -host sc-mic -n 4 ../src/lmp_o3 -sf omp -var x 2 -var y 2 -var z 2 < in.lj : -wdir ~/.
-host mic0 -n 8 ~/lmp_mic -sf omp -var x 2 -var y 2 -var z 2 < in.lj
```

Figure 30

Or, to run and collect traces only on the Intel Xeon Phi coprocessor **(Fig. 31)**:

```
$ mpiexec –trace -wdir ~/. -host mic0 -n 8 ~/lmp_mic -sf omp -var x 2 -var y 2 -var z 2 < in.lj
```

Figure 31

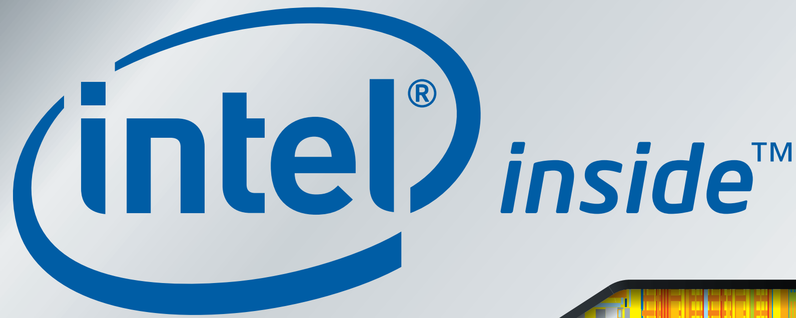
Conclusion

Intel Trace Collector and Intel Trace Analyzer give us insight into MPI application information like message timing, load balancing, and data volume so we can more easily optimize MPI applications. In the case of LAMMPS, we could see the most time-consuming APIs and where they were called.

Intel VTune Amplifier XE provides CPU utilization and hotspot information for each MPI process, which is vital to ensuring those processes scale on each node. Intel VTune Amplifier XE showed us that the USER-OMP package for LAMMPS provided significant CPU resource utilization on a node, allowing the application to scale well on a two-socket 12-core node.

Intel Inspector XE and the check feature of Intel MPI help find MPI, memory, and threading errors in our application, and the Intel C++ Compiler and Intel MPI provide highly optimized binaries and MPI infrastructure that results in a high performance application.

This is only a glimpse into the power of Intel® Software Development tools to create and analyze complex software applications using diverse technology such as MPI and OpenMP across multiple machines. For more information on Intel Software Development tools see www.intel.com/software/products. ☐



CHECKLIST FOR PROGRAMMING

Intel® Xeon Phi™ Coprocessors

by James Reinders,
Director, Software Evangelist, Intel

 For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

The Intel® Xeon Phi™ coprocessor extends the reach of the Intel® Xeon® family of computing products into higher realms of parallelism. This article offers the key tips for programming such a high degree of parallelism, while using familiar programming methods and the latest Intel® Parallel Studio XE 2013 and Intel® Cluster Studio XE 2013—which both support the Intel Xeon Phi coprocessor.

Checklist

A checklist for programming for an Intel Xeon Phi coprocessor looks like this:

- [x] Make sure your application scales beyond 100 threads (usually 150 or more).
- [x] Make sure your application either:
    - Does most computations as efficient vector instructions (requires vectorization)
    - Uses a lot of memory bandwidth with decent locality of reference
- [x] The application is written using your favorite programming languages and parallel models to achieve the above.
- [x] Use your favorite tools, the same ones you use for programming for Intel® Xeon® processors. Get the latest versions that include support for Intel® Xeon Phi™ coprocessor support (such as Intel® Cluster Studio XE 2013, Intel® Parallel Studio XE 2013, Rogue Wave TotalView* debugger and IMSL* Library, NAG Library*, or Allinea DDT* debugger)

It is worth explaining this checklist in more depth, and that is the purpose of this article. You can see that preparing for Intel Xeon Phi coprocessors is primarily about preparing for a 50+ core x86 SMP system with 512-bit SIMD capabilities. That work can happen on most any large, general purpose system, especially one based on Intel Xeon processors. Intel Parallel Studio XE 2013 and Intel Cluster Studio XE 2013 will support your work on an Intel Xeon processor-based system with or without Intel Xeon Phi coprocessors. All the tools you need are in one suite.

Introduction

Intel Xeon Phi coprocessors are designed to extend the reach of applications that have demonstrated the ability to reach the scaling limits of Intel Xeon processor-based systems, and have also maximized usage of available vector capabilities or memory bandwidth. For such applications, the Intel Xeon Phi coprocessors offer additional power-efficient scaling, vector support, and local memory bandwidth, while maintaining the programmability and support associated with Intel Xeon processors.

Advice for successful programming can be summarized as: "Program with lots of threads that use vectors with your preferred programming languages and parallelism models." Since most applications have not yet been structured to take advantage of the full magnitude of parallelism available in an Intel Xeon Phi coprocessor, understanding how to restructure to expose more parallelism is critically important to enable the best performance. This restructuring itself will generally yield benefits on most general purpose computing systems—a bonus due to the emphasis on common programming languages, models, and tools across the Intel Xeon family of products. You may refer to this bonus as the dual-transforming-tuning advantage.

A system that includes Intel Xeon Phi coprocessors will consist of one or more nodes (a single node computer is "just a regular computer"). A typical node consists of one or two Intel Xeon processors, plus one to eight Intel Xeon Phi coprocessors. Nodes cannot consist of only coprocessors.

The First Intel Xeon Phi Coprocessor, Codename Knights Corner

While programming does not require deep knowledge of the implementation of the device, it is definitely useful to know some attributes of the coprocessor. From a programming standpoint, treating it as an x86-based SMP-on-a-chip with over 50 cores, over 200 hardware threads, and 512-bit SIMD instructions is the key.

The cores are in-order, dual-issue x86 processor cores (which trace some history to the original Intel® Pentium® design). But with the addition of 64-bit support, four hardware threads per core, power management, ring interconnect support, 512 bit SIMD capabilities, and other enhancements, these are hardly the Intel Pentium cores of 20 years ago. The x86-specific logic (excluding L2 caches) makes up less than 2 percent of the die for an Intel Xeon Phi coprocessor.

Here are key facts about the first Intel Xeon Phi coprocessor product:

- It is a coprocessor (requires at least one processor in the system); in production in 2012
- Boots and runs Linux* (source code available at http://intel.com/software/mic)
- It is supported by standard tools including Intel Parallel Studio XE 2013. Listings of additional tools available can be found online (http://intel.com/software/mic).
- It has many cores:
  - More than 50 cores (This will vary within a generation of products, and between generations. It is good advice to not hard code applications to a particular number.)
  - In-order cores support 64-bit x86 instructions with uniquely wide SIMD capabilities.
  - Four hardware threads on each core (resulting in more than 200 hardware threads on a single device) are primarily used to hide latencies implicit in an in-order microarchitecture. As such, these hardware threads are much more important for HPC applications to utilize than hyperthreads on an Intel Xeon processor.
  - Cache coherent across the entire coprocessor.
  - Each core has a 512K L2 cache locally with high-speed access to all other L2 caches (making the collective L2 cache size over 25M).
- Special instructions in addition to 64-bit x86:
  - Uniquely wide SIMD capability via 512-bit wide vectors instead of MMX, SSE or AVX.
  - High performance support for reciprocal, square root, power, and exponent operations
  - Scatter/gather and streaming store capabilities for better effective memory bandwidth
  - Performance monitoring capabilities for tools like Intel® VTune™ Amplifier XE 2013

Maximizing Parallel Program Performance

The choice whether to run an application solely on Intel Xeon processors, or to extend an application run to utilize Intel Xeon Phi coprocessors, will always start with two fundamentals:

1. Scaling: Is the scaling of an application ready to utilize the highly parallel capabilities of an Intel Xeon Phi coprocessor? The strongest evidence of this is generally demonstrated scaling on Intel Xeon processors.
2. Vectorization and Memory Locality: Is the application either:
   - Making strong use of vector units?
   - Able to utilize more local memory bandwidth than available with Intel Xeon processors?

If both of these fundamentals are true for an application, then the highly parallel and power-efficient Intel Xeon Phi coprocessor is most likely worth evaluating.

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

Ways to Measure Readiness for Highly Parallel Execution

To know if your application is maximized on an Intel Xeon processor-based system, you should examine how your application scales, as well as how it uses vectors and memory. Assuming you have a working application, you can get some impression of where you are with regards to scaling and vectorization by doing a few simple tests.

To check scaling, create a simple graph of performance as you run with various numbers of threads (from one up to the number of cores, with attention to thread affinity) on an Intel Xeon processor-based system. This can be done with settings for OpenMP*, Intel® Threading Building Blocks (Intel® TBB) or Intel® Cilk™ Plus (e.g., OMP_NUM_THREADS for OpenMP). If the performance graph indicates any significant trailing off of performance, you have tuning work you can do to improve your application before trying an Intel Xeon Phi coprocessor.

To check vectorization, compile your application with and without vectorization. If you are using Intel compilers: disable vectorization via compiler switch: -no-vec, use at least –O2 xhost for vectorization. Compare the performance you see. If the performance difference is insufficient, you should examine opportunities to increase vectorization. Look again at the dramatic benefits vectorization may offer as illustrated in Figure 7. If you are using libraries, such as the Intel® Math Kernel Library (Intel® MKL), you should consider that time in Intel MKL routines offer vectorization invariant to the compiler switches. Unless your application is bandwidth limited, effective use of Intel Xeon Phi coprocessors should be done with most cycles executing having computations utilizing the vector instructions. While some may tell you that "most cycles" needs to be over 90 percent, we have found this number to vary widely based on the application and whether the Intel Xeon Phi coprocessor needs to be the top performance source in a node or just to contribute to performance.

The Intel® VTune™ Amplifier XE 2013 can help measure computations on Intel Xeon processors and Intel Xeon Phi coprocessor to assist in your evaluations.

Aside from vectorization, being limited by memory bandwidth on Intel Xeon processors can indicate an opportunity to improve performance with an Intel Xeon Phi coprocessor. In order for this to be most efficient, an application needs to exhibit good locality of reference and utilize caches well in its core computations.

The Intel VTune Amplifier XE product can be utilized to measure various aspect of a program, and among the most critical is "L1 Compute Density." This is greatly expanded upon in a paper titled *Using Hardware Events for Tuning on Intel® Xeon Phi™ Coprocessor (codename: Knights Corner)*.

When using MPI, it is desirable to see a communication vs. computation ratio that is not excessively high in terms of communication. Because programs vary so much, this has not been well characterized other than to say that, like other machines, Intel Xeon Phi coprocessors favor programs with more computation vs. communication. Programs are most effective using a strategy of overlapping communication and I/O by computation. Intel® Trace Analyzer and Collector, part of Intel Cluster Studio XE 2013, is very useful for profiling. It can be used to profile MPI communications to help visualize bottlenecks and understand the effectiveness of overlapping with computation to characterize your program.

Compiler and Programming Models

No popular programming language was designed for parallelism. In many ways, Fortran has done the best job adding new features, such as DO CONCURRENT, to address parallel programming needs, as well as benefiting from OpenMP. C users have OpenMP, as well as Intel Cilk Plus. C++ users have embraced Intel Threading Building Blocks and, more recently, have Intel Cilk Plus to utilize as well. C++ users can use OpenMP as well.

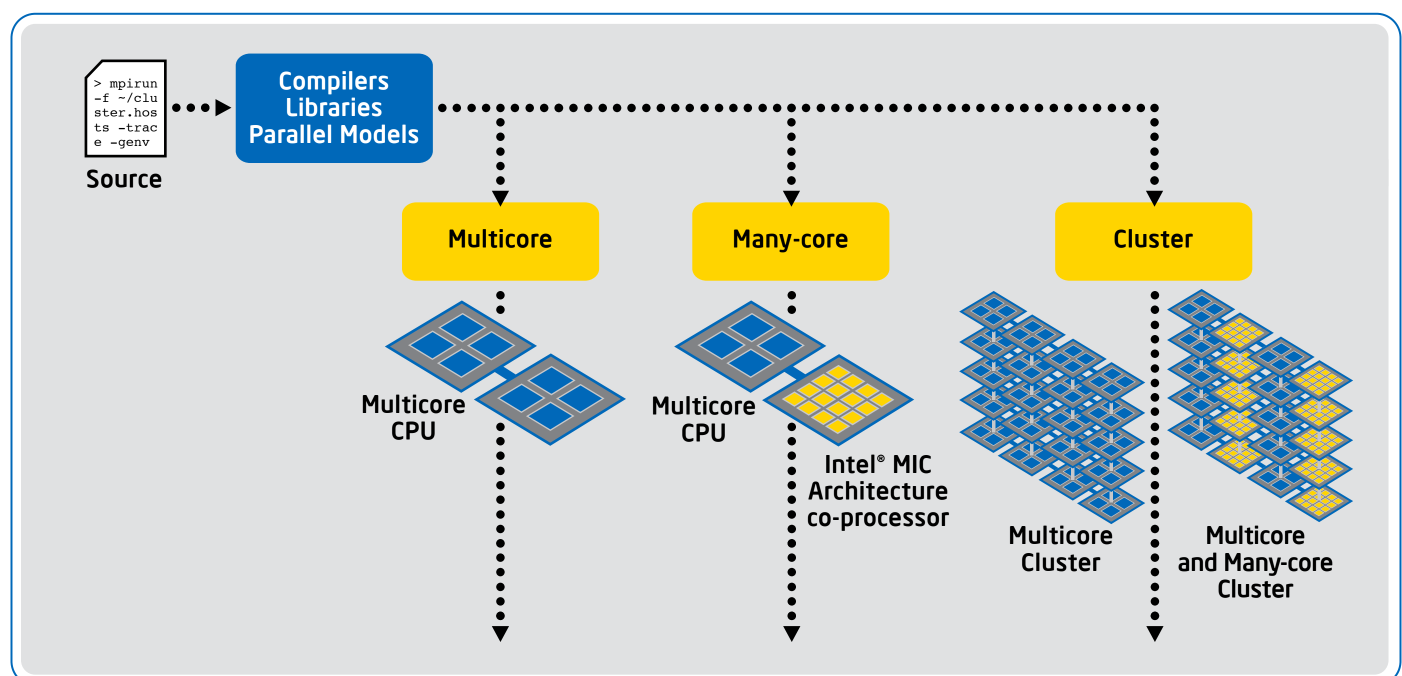


Figure 1: The double advantage of transforming-and-tuning means that optimizations are shared across the Intel®Xeon® family of products, Capabilities of Intel Xeon processors are extended by Intel® Xeon Phi™ coprocessors.

Intel Xeon Phi coprocessors offer the full capability to use the same tools, programming languages, and programming models as an Intel Xeon processor. However, with this coprocessor designed for high degrees of parallelism, some models are more interesting than others.

In essence, it is quite simple: an application needs to deal with having lots of tasks (call them "workers" or "threads" if you prefer), and deal with vector data efficiently (a.k.a., vectorization).

There are some recommendations we can make based on what has been working well for developers. For Fortran programmers, use OpenMP, DO CONCURRENT, and MPI. For C++ programmers, use Intel TBB, Intel Cilk Plus, and OpenMP. For C programmers, use OpenMP and Intel Cilk Plus. Intel TBB is a C++ template library that offers excellent support for task-oriented load balancing. While Intel TBB does not offer vectorization solutions, it does not interfere with any choice of solution for vectorization. Intel TBB is open source and available on a wide variety of platforms supporting most operating systems and processors. Intel Cilk Plus is a bit more complex in that it offers both tasking and vectorization solutions. Fortunately, Intel Cilk Plus fully interoperates with Intel TBB. Intel Cilk Plus offers a simpler set of tasking capabilities than Intel TBB, but uses keywords in the language to enable full compiler support for optimizing.

Intel Cilk Plus also offers elemental functions, array syntax, and "#pragma SIMD" to help with vectorization. The best use of array syntax is implemented along with blocking for caches, which unfortunately means naïve use of constructs such as A[:] = B[:] + C[:]; for large arrays may yield poor performance. The best use of array syntax ensures that the vector length of single statements is short (some small multiple of the native vector length, perhaps only 1X). Finally, and perhaps most important to programmers today, Intel Cilk Plus offers mandatory vectorization pragmas for the compiler called "#pragma SIMD." The intent of "#pragma SIMD" is to do for vectorization what OpenMP has done for parallelization. Intel Cilk Plus requires compiler support. It is currently available from Intel for Windows*, Linux*, and Apple OS* X. It is also available in a branch of gcc.

If you are happy with OpenMP and MPI, you are in great shape to use Intel Xeon Phi coprocessors. Additional options may be interesting to you over time, but OpenMP and MPI are enough to get great results. Your key challenge will remain vectorization. Auto-vectorization may be enough for you, especially if you code in Fortran, with the possible additional considerations for efficient vectorization, such as alignment and unit-stride accesses. The "#pragma SIMD" capability of Intel Cilk Plus (available in Fortran, too) is worth a look. In time, you may find it has become part of OpenMP.

Dealing with tasks means specification of tasks, and load balancing amongst them. MPI has provided this capability for decades with full flexibility and control given to the programmer. Shared memory programmers have Intel TBB and Intel Cilk Plus to assist them. Intel TBB has widespread usage in the C++ community. Intel Cilk Plus extends Intel TBB to offer C programmers a solution, as well as help with vectorization in C and C++ programs.

Coprocessor Major Usage Model: MPI vs. Offload

Given that we know how to program the Intel Xeon processors in the host system, the question arises of how to involve the Intel Xeon Phi coprocessors in an application. There are two major approaches: (1) "offload" selective portions of an application to the Intel Xeon Phi coprocessors, and (2) run an MPI program where MPI ranks can exist on Intel Xeon processors cores, as well as on Intel Xeon Phi coprocessor cores with connections made by MPI communications. The first is call "offload mode" and the second "native mode." The second does not require MPI to be used, because any SMP programming model can be employed, including just running on a single core. There is no machine "mode" in either case, only a programming style that can be intermingled in a single application if desired. Offload is generally used for finer-grained parallelism and, as such, generally involves localized changes to a program. MPI is more often done in a coarse-grained manner, often requiring more scattered changes in a program. RDMA support for MPI is available.

The choice is certain to be one of considerable debate for years to come. Applications that already utilize MPI can actually use either method by either limiting MPI ranks to Intel Xeon processors and use offload to the coprocessors, or distributing MPI ranks across the coprocessors. It is possible that the only real MPI ranks be established on the coprocessor cores, but if this leaves the Intel Xeon processors unutilized then this approach is likely to give up too much performance in the system.

Being separate and on a PCIe bus creates two additional issues: (1) the limited memory on the coprocessor card, and (2) the benefits of minimizing communication to and from the card. It is worth noting as well, that the number of MPI ranks used on an Intel Xeon Phi coprocessor should be substantially less than the number of cores—in no small part because of limited memory on the coprocessor. Consistent with parallel programs in general, the advantages of overlapping communication (e.g., MPI messages or offload data movement) with computation are important to consider, as well as techniques to load balance work across all available cores. Of course, involving Intel Xeon processor cores and Intel Xeon Phi coprocessor cores adds the dimension of "big cores" and "little cores" to the balancing work, even though they share x86 instructions and programming models. While MPI programs often already tackle the overlap of communication and computation, the placement of ranks on coprocessor cores still requires dealing with the highly parallel programming needs and limited memory. This is why an offload model can be attractive, even within an MPI program.

The offload model for Intel Xeon Phi coprocessors is quite rich. The syntax and semantics of the Intel® Language Extensions for Offload are generally a superset of other offload models including OpenACC. This provides for greater interoperability with OpenMP; ability to manage multiple coprocessors (cards); and the ability to offload complex program components that an Intel Xeon Phi coprocessor can process, but that a GPU could not (and hence, OpenACC does not allow). We expect that a future version of OpenMP will include offload directives that provide support for these needs, and Intel plans to support such a standard for Intel Xeon Phi coprocessors

 For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

as part of our commitment to providing OpenMP capabilities. Intel Language Extensions for Offload also provides for an implicit sharing model that is beyond what OpenMP will support. It rests on a shared memory model supported by Intel Xeon Phi coprocessors that allow a shared memory programming model (Intel calls this "MYO") between Intel Xeon processors and Intel Xeon Phi coprocessors. This is most similar to partitioned global address space (PGAS) programming models; not an extension provided by OpenMP. The Intel "MYO" capability offers a global address space within the node, allowing sharing of virtual addresses for select data between processors and coprocessor on the same node. It is offered in C and C++, but not Fortran, since future support of coarray will be a standard solution to the same basic problem. Offloading is available as Fortran offloading via pragmas, C/C++ offloading with pragmas, and optionally shared (MYO) data. Use of MPI can also distribute applications across the system.

Summary: Transforming-and-Tuning Double Advantage

Programming should not be called easy, and neither should *parallel programming*. However, we can work to keep the fundamentals the same: maximizing parallel computations and minimizing data movement. Parallel computations are enabled through scaling (more cores and threads) and vector processing (more data processed at once). Minimal data movement is an algorithmic endeavor, but can be eased through the higher bandwidth between memory and cores that is available with the Intel® Many Integrated Core (Intel® MIC) architecture used by Intel Xeon Phi coprocessors. This leads to *parallel programming* using the same programming languages and models across the Intel Xeon family of products, which are generally also shared across all general purpose processors in the industry. Languages such Fortran, C, and C++ are fully supported. Popular programming methods such as OpenMP, MPI, and Intel TBB are fully supported. Newer models with widespread support such as Coarray Fortran, Intel Cilk Plus, and OpenCL* can apply as well.

Tuning on Intel Xeon Phi coprocessors for scaling, and vector and memory usage, also benefits the application when run on Intel Xeon processors. Maintaining a value across the Intel Xeon family is critical, as it helps preserve past and future investments. Applications that initially fail to get maximum performance on Intel Xeon Phi coprocessors generally trace problems back to scaling, vector usage, or memory usage. When these issues are addressed, the improvements to the application usually have a related positive effect when run on Intel Xeon processors. This is the double advantage of "transforming-and-tuning," and developers have found it to be among the most compelling features of the Intel Xeon Phi coprocessors. □

Learn More

Additional material regarding programming for Intel Xeon Phi coprocessors can be found at http://intel.com/software/mic.

Parallel Programming Community: http://software.intel.com/en-us/parallel/

Advanced Vector Extensions: http://software.intel.com/en-us/avx/

Intel Guide for Developing Multithreaded Applications: http://software.intel.com/en-us/articles/intel-guide-for-developing-multithreaded-applications/

BLOG highlights

ispc: Intel® Xeon® and Intel® Xeon Phi™ support now

JAMES REINDERS, (Intel)
Director of Parallel Programming Evangelism

Vectorization is an industry-wide challenge—and if you are interested in seeing some one of the industry-leading exploration projects (and trying it on your code), then you may want to look at ispc.

ispc is an R&D compiler for a C-based language that is targeted to exploring the performance available from doing SPMD (single program, multiple data) computation on SIMD units found on CPUs and on Intel® Xeon Phi™ coprocessors (using the Intel® Many Integrated Core [MIC] architecture). It has delivered performance competitive with hand-coded SSE and AVX for a variety of graphics and throughput kernels, and typically delivers a 3x to 4x speedup vs. scalar C and C++ code on SSE and a 5 to 7x speedup on AVX (for computations that are amenable to SPMD implementation), while still providing the ease of use of a C-like language.

The paper "ispc: A SPMD Compiler for High-Performance CPU Programming" by Matt Pharr and Bill Marks won *Best Paper Award* at InPar 2012. It is an excellent paper that articulates the challenges of vectorization and explains the important context very well. It also advances a solid demonstration of what is possible when you think about SPMD on SIMD models clearly...

SEE THE REST OF JAMES' BLOG:

Visit **Go-Parallel.com**

Browse other blogs exploring a range of related subjects at Go Parallel: Translating Multicore Power into Application Performance.

ADVANCED Vectoriz

by Georg Zitzlsberger,
Technical Consulting Engineer, Intel

Issue 10 of the *Parallel Universe Magazine* featured a basic introduction to vectorization. However, this parallelization technique using Intel® compilers is neither black nor white. There are many variations that impact the degree of vectorization, and the performance of the final application. For large-scale systems like compute clusters, even small improvements are desirable—vector computation as a per-core feature of the processor quickly amplifies through core and node count.

Here, we demonstrate the most important features and best-known methods by applying some of the vectorization techniques enabled by Intel® compilers and their Intel® Cilk™ Plus technologies to an example application implementing an image processing algorithm.

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

ation

Example Application

Our example application reads a motion-blurred JPEG image, calling a filter function that undoes the motion blur (i.e., deconvolution) and eliminates low frequencies, before saving it back to a JPEG file. **Figure 1** shows the basic workflow.

To keep the implementation simple, we have decided to implement the filter function in the frequency domain. Thus, before executing the function a forward DFT takes place, transforming the image into two single-precision floating point (32-bit FP) arrays: magnitude and phase. The same is also applied to the (de-)convolution kernel. Initial implementations of the filter function are shown in **Figure 2**, in both C and Fortran versions. After executing the filter function, both output arrays are recombined to an image by inverse DFT and stored as JPEG.

Our example operates on three channel (RGB) images, so magnitude and phase are clustered in the same way. This adds some complexity to the filter regarding the DC values, which does not make it trivial to vectorize (see modulo operation).

The filter is provided as a separate compilation unit. To measure the quality of the vectorization, we record the time it takes to execute it. To lower the noise, the filter is executed multiple times with the same input data. Throughout this article, we only focus on the vectorization of the filter itself. For the detailed implementation and reproducibility you can download the example from our blog.

Baseline

For our measurements, we are using Ubuntu* 11.04 (64-bit) running on an Intel® Core™ i7 processor (i7-2600). We're also using 64-bit C/C++ and Fortran compilers shipped with Intel® Composer XE 2013. Since we're working with arrays of single-precision FP values, we also want to make full use of Intel® Advanced Vector Extensions (Intel® AVX), which can process vectors with up to eight such elements at once.

Starting with the baseline implementations for C and Fortran shown in Listing 1, we measure the runtime of repetitive calls to the filter and denote them as baseline. We get those results with standard build options and can see that there's improvement left for vectorization.

Looking at this example, the benefit of vectorization seems quite obvious. But did the baseline already take full advantage of it? A common, though tedious, way to verify this is by analyzing the produced assembly code. For complex algorithms, this easily becomes a challenge and also requires advanced knowledge about the underlying architecture. A better solution would be using Intel® VTune™ Amplifier XE to count executed instructions using SIMD vectors. However, this still requires some knowledge about the architecture. Fortunately there is yet another, much faster, and easier way, that is used here—the Intel compiler vectorization reports (**Table 1**).

First, we start with the C implementation and apply the vectorization report with `n=3` to the source file "feature.c." It unveils rather dramatic reasons that turn out to have hindered the compiler from proper vectorization:

Boiling down the redundant output, mostly caused by loop unrolling and permutation of dependencies, reveals that the loop cannot be vectorized because of dependencies between elements of the arrays, including:

1. "imgMag"and "imgPhase"
2. "imgMag" and "filterMag"
3. "imgMag" and "filterPhase"
4. "imgMag" and "gaussian"
5. "imgPhase" and "filterMag"
6. "imgPhase" and "gaussian"

If we list the dependencies this way it turns out that only two arrays, "imgMag" and "imgPhase," are involved. Those are the output parameters from our filter. The report assumes that any write access also influences the other arrays. Why?

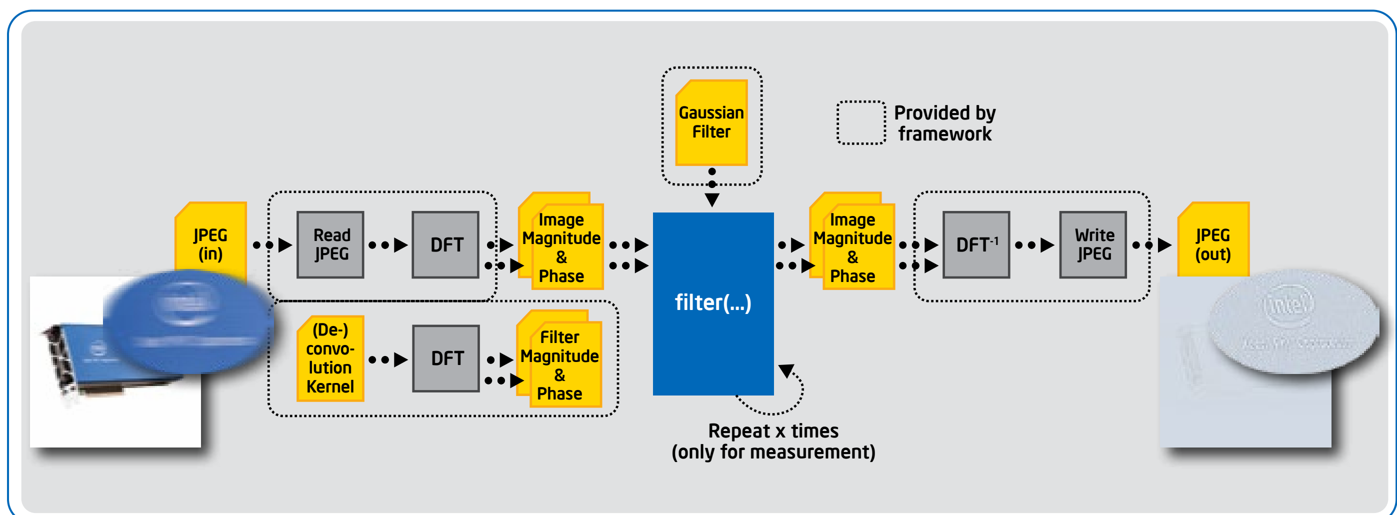


Figure 1: Block diagram of the example used. Focus is the implementation of the filter.

For more information regarding performance and optimization choices in Intel® software products, visit http://software.intel.com/en-us/articles/optimization-notice.

filter.c:

```
#include "filter.h"

void filter(
    unsigned int size,
    float *imgMag, float *imgPhase,       // in & out
    float *filterMag, float *filterPhase,   // in
    float *gaussian)                      // in
{
  for(int idx = 0; idx < size; idx++)
  { // idx%3: RGB values from DC, first 3 elements of "filterMag"
    imgMag[idx] /= (filterMag[idx] * 1/filterMag[idx%3] + filterMag[idx%3]);
    imgMag[idx] *= gaussian[idx];
    imgPhase[idx] -= filterPhase[idx];
  }
}
```

filter.f90:

```
subroutine filter(sze, imgMag, imgPhase, filterMag, filterPhase, gaussian) &
  bind(C, name="filter")
  use, intrinsic :: ISO_C_BINDING
  implicit none
  integer(kind=C_INT), VALUE                 :: sze
  real(C_FLOAT), dimension(*), intent(inout) :: imgMag
  real(C_FLOAT), dimension(*), intent(inout) :: imgPhase
  real(C_FLOAT), dimension(*), intent(in)    :: filterMag
  real(C_FLOAT), dimension(*), intent(in)    :: filterPhase
  real(C_FLOAT), dimension(*), intent(in)    :: gaussian
  integer                           :: idx
! mod(idx, 3) + 1: RGB values from DC, first 3 elements of "filterMag"
  do idx = 1, sze
    imgMag(idx) = imgMag(idx) / (filterMag(idx) *                &
                               1/filterMag(mod(idx, 3) + 1) + &
                               filterMag(mod(idx, 3) + 1));
    imgMag(idx) = imgMag(idx) * gaussian(idx)
    imgPhase(idx) = imgPhase(idx) - filterPhase(idx)
  end do
end subroutine
```

Figure 2

Syntax		
Linux* and Mac OS* X: `-vec-report[n]` **Windows*:** `/Qvec-report[n]`		
Arguments		
n	Is a value denoting which diagnostic messages to report. **Possible values are:**	
	0	Tells the vectorizer to report no diagnostic information.
	1	Tells the vectorizer to report on vectorized loops.
	2	Tells the vectorizer to report on vectorized and non-vectorized loops.
	3	Tells the vectorizer to report on vectorized and non-vectorized loops and any proven or assumed data dependencies.
	4	Tells the vectorizer to report on non-vectorized loops.
	5	Tells the vectorizer to report on non-vectorized loops and the reason why they were not vectorized.

Table 1: Compiler option to turn on the Intel® compiler vectorization reports

```
filter.c(9): (col. 3) remark: loop was not vectorized: existence of vector dependence.
filter.c(13): (col. 5) remark: vector dependence: assumed FLOW dependence
between imgPhase line 13 and filterMag line 11.
filter.c(11): (col. 5) remark: vector dependence: assumed ANTI dependence
between filterMag line 11 and imgPhase line 13.
filter.c(13): (col. 5) remark: vector dependence: assumed FLOW dependence
between imgPhase line 13 and filterMag line 11.
filter.c(11): (col. 5) remark: vector dependence: assumed ANTI dependence
between filterMag line 11 and imgPhase line 13.

...

filter.c(13): (col. 5) remark: vector dependence: assumed ANTI dependence
between imgPhase line 13 and imgMag line 11.
filter.c(11): (col. 5) remark: vector dependence: assumed OUTPUT dependence
between imgMag line 11 and imgPhase line 13.
filter.c(13): (col. 5) remark: vector dependence: assumed OUTPUT dependence
between imgPhase line 13 and imgMag line 11.
```

Figure 3

```
filter.c:

# include "filter.h"

void filter(
    unsigned int size,
    float * restrict imgMag, float * restrict imgPhase, // in & out
    float *filterMag, float *filterPhase,               // in
    float *gaussian)                                    // in
{
  for(int idx = 0; idx < size; idx++)
  { // idx%3: RGB values from DC, first 3 elements of "filterMag"
    imgMag[idx] /= (filterMag[idx] * 1/filterMag[idx%3] + filterMag[idx%3]);
    imgMag[idx] *= gaussian[idx];
    imgPhase[idx] -= filterPhase[idx];
  }
}
```

Figure 4

"For large-scale systems like compute clusters, even small improvements are desirable—vector computation as a per-core feature of the processor quickly amplifies through core and node count."

In C and C++, a pointer to a memory location is assumed to be overlapped by other pointers. Even strict ANSI aliasing rules, which prohibit reference of memory locations by pointers of different types, are not strict enough. They still allow pointers of the same type to overlap.

In our example, this is the case for all pointers. Hence the compiler has to assume dependencies among them. It should be emphasized that these are "assumed" dependencies, meaning they must not occur during runtime, and the compiler cannot disambiguate because the filter is implemented in a separate compilation unit. Nevertheless, any language-standard compliant compiler needs to assume the worst case—with pointers overlapping—and handle them correctly.

In most cases, such assumed dependencies may not be desired, and add unnecessary complexity to optimizations such as vectorization. However, for our example, every pointer from the parameter list references its own memory location. As a result these are not expected to overlap at all, which breaks up dependencies and increases the likelihood of successful vectorization. There are two ways to tell the compiler to ignore such assumed dependencies, the "restrict" keyword and the IVDEP pragma/directive.

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

"Restrict" Keyword & IVDEP Pragma/Directive

The "restrict" keyword is a feature of the C99 standard. It can be attributed to pointers to guarantee that no other pointer overlaps the referenced memory location. Using the Intel® C++ compiler does not only limit it to C99. It makes the keyword available for C89 and even for different incarnations of C++, simply by enabling a dedicated option: "-restrict" (Linux* and Mac OS* X) or "/Qrestrict" (Windows*).

Figure 4 shows a possible implementation by using this keyword. It should be noted that we only need to apply the keyword to the two output parameters to break up the dependencies.

Another approach is the IVDEP pragma/directive, provided by all Intel compilers (C++ and Fortran). This pragma is used in front of block scopes, such as loops, to tell the compiler to ignore all assumed dependencies therein. Applied to our C example (**Figure 5**), the speedup is the same as with the "restrict" keyword (**Figure 6**). Nevertheless, the pragma still has some advantages over it. First, pragmas not known by other compilers are ignored. Thus this is less intrusive than using the keyword approach, which might not work for other compilers and non-C99 code. Second, the pragma can be used locally (e.g., for one loop), and leave the rest unchanged, while the keyword has impact on the entire function body. And lastly, the locality of the pragma allows it to selectively ignore overlapping memory locations. For example, it might be legal to ignore dependencies if certain conditions are met, but it would be incorrect to generally ignore these throughout the entire function body.

```
filter.c:

#include "filter.h"

void filter(
    unsigned int size,
    float *imgMag, float *imgPhase,       // in & out
    float *filterMag, float *filterPhase,  // in
    float *gaussian)                      // in
{
#pragma ivdep
   for(int idx = 0; idx < size; idx++)
   { // idx%3: RGB values from DC, first 3 elements of "filterMag"
     imgMag[idx] /= (filterMag[idx] * 1/filterMag[idx%3] + filterMag[idx%3]);
     imgMag[idx] *= gaussian[idx];
     imgPhase[idx] -= filterPhase[idx];
   }
}

filter.f90:

subroutine filter(sze, imgMag, imgPhase, filterMag, filterPhase, gaussian) &
   bind(C, name="filter")
   use, intrinsic :: ISO_C_BINDING
   implicit none
   integer(kind=C_INT), VALUE                 :: sze
   real(C_FLOAT), dimension(*), intent(inout) :: imgMag
   real(C_FLOAT), dimension(*), intent(inout) :: imgPhase
   real(C_FLOAT), dimension(*), intent(in)    :: filterMag
   real(C_FLOAT), dimension(*), intent(in)    :: filterPhase
   real(C_FLOAT), dimension(*), intent(in)    :: gaussian
   integer                                    :: idx
! mod(idx, 3) + 1: RGB values from DC, first 3 elements of "filterMag"
!DEC$ IVDEP
   do idx = 1, sze
     imgMag(idx) = imgMag(idx) / (filterMag(idx) *                &
                                1/filterMag(mod(idx, 3) + 1) +   &
                                filterMag(mod(idx, 3) + 1));
     imgMag(idx) = imgMag(idx) * gaussian(idx)
     imgPhase(idx) = imgPhase(idx) - filterPhase(idx)
   end do
end subroutine
```

Figure 5

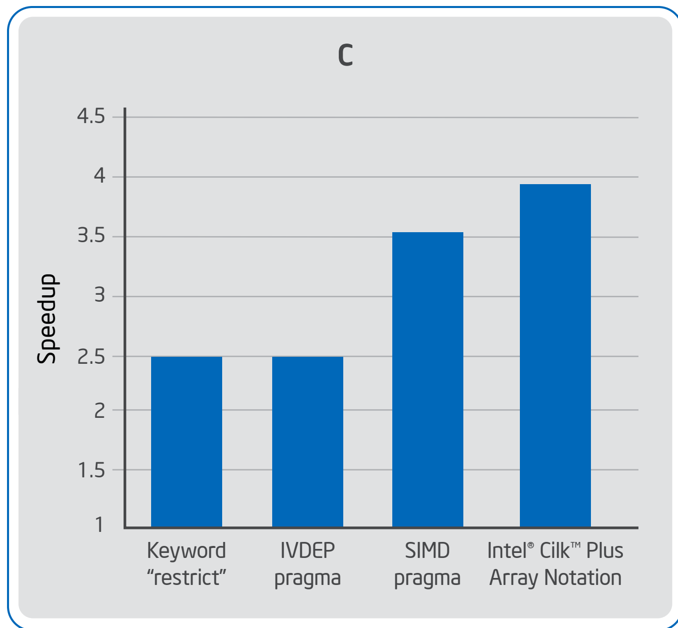


Figure 6: Speedup for the example against the baseline of the C version (higher is better)

You might have noticed already that the result for the IVDEP directive (**Figure 5**) is not shown in **Figure 7**. That's true because Fortran is much stricter regarding overlapping memory regions than C, and because there's no improvement in our example. There are exceptions when using Fortran pointers.

Numerous other cases can also produce assumed dependencies for both C and Fortran. Two examples are access of array elements through indirection at runtime and conditional memory accesses in a loop. All such cases justify the availability of the IVDEP directive for Fortran as well.

In any case, both "restrict" keyword and IVDEP pragma/directive can change semantics of the code if applied incorrectly. For our example, both inform the compiler that no other pointer overlaps the referenced memory location. If this is violated, for instance, by sharing memory among different pointers of the function parameters, the compiler might reorder or otherwise optimize accesses in a way that yields different results. Therefore, it is crucial to verify the correctness once the "restrict" keyword and IVDEP pragma/directive are used. In our case, we can do so easily by comparing the results of the different versions we have created. If the resulting image was corrupted, further analysis would be required. Since our example uses disjunctive memory locations, all optimizations work well.

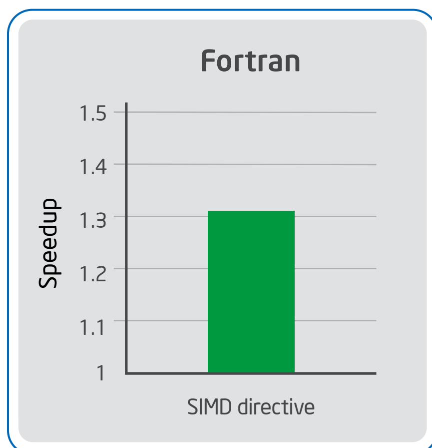


Figure 7: Speedup for the example against the baseline of the Fortran version (higher is better)

Intel® Cilk™ Plus SIMD Pragma/Directive

For the next step, we apply any of the above proposals to our C implementation. For the Fortran version, we still use the baseline. After consulting the vectorization report once again, vectorization still does not look optimal, for either the C or Fortran version (**Figure 8**):

It seems like the loop was not fully vectorized. This reflects the conclusion from the compiler's efficiency heuristic. It only regards a fraction of the loop body as being meaningfully vectorized. This result is based on general knowledge about the underlying architecture and on following language standards. Here, the compiler regards only a small fraction as not being vectorized, because of gather accesses caused by the modulo operations. There can be other cases where the compiler might not vectorize at all because of alleged inefficiency. In such cases there will be a clear message (**Figure 9**):

In such cases, it may make sense to ignore such heuristics, and also to partly ignore language standards to "enforce" vectorization. It should be noted that vectorization can only be enforced if it is technically possible. A loop which prints strings to the terminal is a trivial example which cannot be vectorized.

Intel Cilk Plus—an integral part of Intel compilers—comes with a SIMD pragma/directive that does what we are looking for: it ignores any compiler heuristic regarding efficiency, as well as any restrictions induced by the language to enforce vectorization. It also ignores all dependencies; not just assumed ones like the IVDEP pragma/directive. Due to this fact, it is important to mention that it can change semantics of vectorized code as a side effect. Hence, it is mandatory to verify the correctness once more. For our example, we simply compare the output among the different versions, as we already did for the changes with the "restrict" keyword and IVDEP pragma/directive.

Applying the SIMD pragma/directive to our example (**Figure 10**) indeed shows an additional improvement (**Figure 11**). Verifying the results also holds true; thus being a prime solution for us.

```
C:
filter.c(9): (col. 3) remark: PARTIAL LOOP WAS VECTORIZED.
filter.c(9): (col. 3) remark: PARTIAL LOOP WAS VECTORIZED.

Fortran:
filter.f90(13): (col. 3) remark: PARTIAL LOOP WAS VECTORIZED.
filter.f90(13): (col. 3) remark: PARTIAL LOOP WAS VECTORIZED.
```

Figure 8

```
remark: loop was not vectorized: vectorization possible but
seems inefficient.
```

Figure 9

For more information regarding performance and optimization choices in Intel® software products, visit http://software.intel.com/en-us/articles/optimization-notice.

filter.c:

```
#include "filter.h"

void filter(
    unsigned int size,
    float * restrict imgMag, float * restrict imgPhase, // in & out
    float *filterMag, float *filterPhase,               // in
    float *gaussian)                                    // in
{
#pragma simd vectorlength(8) assert
  for(int idx = 0; idx < size; idx++)
  { // idx%3: RGB values from DC, first 3 elements of "filterMag"
    imgMag[idx] /= (filterMag[idx] * 1/filterMag[idx%3] + filterMag[idx%3]);
    imgMag[idx] *= gaussian[idx];
    imgPhase[idx] -= filterPhase[idx];
  }
}
```

filter.f90:

```
subroutine filter(sze, imgMag, imgPhase, filterMag, filterPhase, gaussian) &
  bind(C, name="filter")
  use, intrinsic :: ISO_C_BINDING
  implicit none
  integer(kind=C_INT), VALUE                   :: sze
  real(C_FLOAT), dimension(*), intent(inout) :: imgMag
  real(C_FLOAT), dimension(*), intent(inout) :: imgPhase
  real(C_FLOAT), dimension(*), intent(in)    :: filterMag
  real(C_FLOAT), dimension(*), intent(in)    :: filterPhase
  real(C_FLOAT), dimension(*), intent(in)    :: gaussian
  integer                                      :: idx
! mod(idx, 3) + 1: RGB values from DC, first 3 elements of "filterMag"
!DEC$ SIMD VECTORLENGTH(8) ASSERT
  do idx = 1, sze
    imgMag(idx) = imgMag(idx) / (filterMag(idx) *                &
                                1/filterMag(mod(idx, 3) + 1) +  &
                                filterMag(mod(idx, 3) + 1));
    imgMag(idx) = imgMag(idx) * gaussian(idx)
    imgPhase(idx) = imgPhase(idx) - filterPhase(idx)
  end do
end subroutine
```

Figure 10

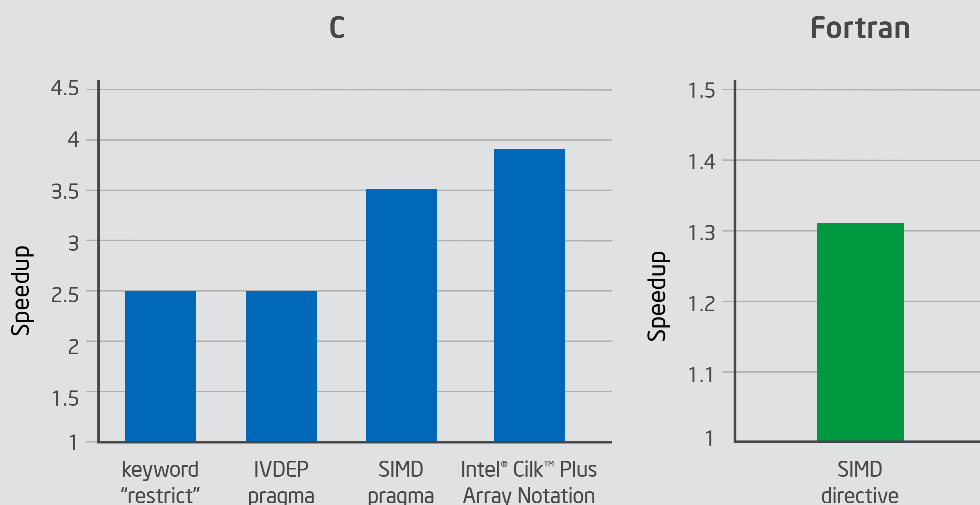


Figure 11

Syntax		
C/C++: `#pragma simd [clause[ [,] clause]...]` Fortran: `cDEC$ SIMD [clause[, clause]...]`		
Arguments		
`clause`	**Can be any of the following:**	
	`vectorlength(n1[, n2]...)`	Vector length to use (power of 2)
	`vectorlengthfor(data type)`	Only for C/C++; same as above but uses built-in types to calculate the vector length

	`private(var1[, var2]...)`
	`firstprivate(var1[, var2]...)`
	`lastprivate(var1[, var2]...)`
	`reduction(oper:var1 [,var2]...)`
		Same as in OpenMP* work-sharing construct: see the OpenMP* 3.1 specification, section 2.5.1

	`linear(var1:step1 [,var2:step2]...)`	Specify additional induction variables
	`[no]assert`	Compile time error in case vectorization fails

Table 2: SIMD pragma/directive and clauses

```
filter.hpp:

...
class FreqDomain {
  public:
    FreqDomain(float *mag, float *phase, unsigned int size) :
      mMag(mag), mPhase(phase), mSize(size) {}

    unsigned int getSize() { return mSize; }
    float getMag(unsigned int idx) { return mMag[idx]; }
    void setMag(unsigned int idx, float val) { mMag[idx] = val; }
    float getPhase(unsigned int idx) { return mPhase[idx]; }
    void setPhase(unsigned int idx, float val) { mPhase[idx] = val; }

  private:
    const unsigned int mSize;
    float *mMag;
    float *mPhase;
};
...

filter.cpp:

include "filter.hpp"

void filter(FreqDomain &img, FreqDomain &filter, float *gaussian)
{ //                    ^in & out      ^in              ^in
// Uncomment either of the following for IVDEP or SIMD results
// #pragma ivdep
// #pragma simd vectorlength(8) assert
  for(int idx = 0; idx < img.getSize(); idx++)
  { // idx%3: RGB values from DC, first 3 elements of magnitude from "filter"
    img.setMag(idx, img.getMag(idx) / (filter.getMag(idx) *
                                   1/filter.getMag(idx%3) +
                                   filter.getMag(idx%3)));
    img.setMag(idx, img.getMag(idx) * gaussian[idx]);
    img.setPhase(idx, img.getPhase(idx) - filter.getPhase(idx));
  }
}
```

Figure 12

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

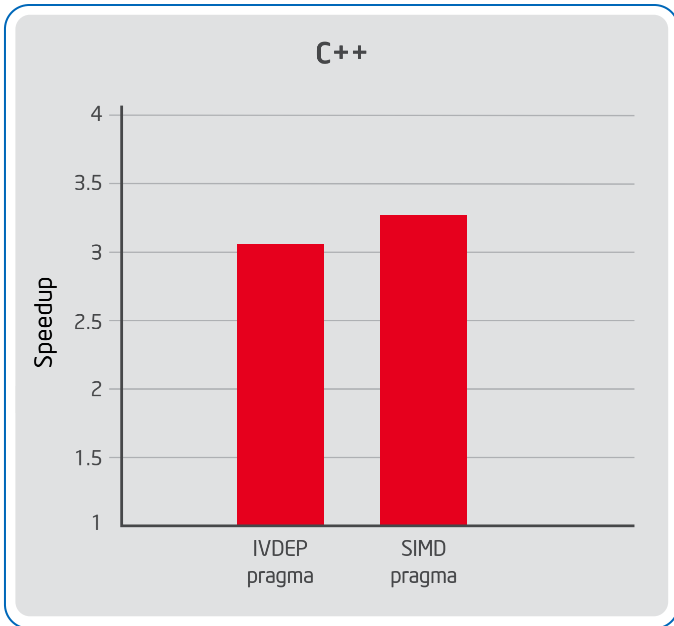


Figure 13: Speedups for the example against the baseline of the C++ version (higher is better))

For our example we also provided additional clauses to the pragma/directive: "vectorlength(8)" and "assert." The former guarantees the use of vectors with eight elements—in this case an Intel AVX 256 bit vector with eight single-precision FP elements. The latter yields a compile time error in case the loop cannot be vectorized at all. This can be useful during development in order to guarantee that only vectorized code is generated, thus detecting unwanted changes early. There are additional clauses (**Table 2**). Most of them are derived from the OpenMP* work-sharing construct. This highlights another aspect of the SIMD pragma/directive: it maps existing OpenMP paradigms for concurrency to vectorization.

References in C++

In combination with C++, IVDEP and SIMD pragmas have another advantage. For this, we convert the C version from our example to C++ (**Figure 12**). We use the dedicated class "FreqDomain" for the frequency domain, containing pointers to arrays for magnitude and phase. Instead of passing such pointers to the filter, we provide them encapsulated in wrapper objects. Those objects are forwarded by reference. And that's the pinpoint: with passing references we lose the option to use the keyword "restrict," because it is only specified for pointers, not references.

```
filter_cilk.c:

#include "filter.h"

void filter(
    unsigned int size,
    float imgMag[size], float imgPhase[size],        // in & out
    float filterMag[size], float filterPhase[size],   // in
    float gaussian[size])                             // in
{
// Note: 24 is least common multiple of 3 RGB elements and 8 elements per vector
// (3 vectors in total)
//                              R  G  B  R  G  B   ...
unsigned int index[24] = { 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2,
                           0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2 };
int idx = 0;
for(; idx < size; idx += 24)
    imgMag[idx:24] /= (filterMag[idx:24] * 1/filterMag[index[:]] +
                       filterMag[index[:]]) * 1/gaussian[idx:24];

for(; idx < size; idx++) // Rest (< 24)
    imgMag[idx] /= (filterMag[idx] * 1/filterMag[idx%3] +
                    filterMag[idx%3]) * 1/gaussian[idx];

imgPhase[:] -= filterPhase[:];
```

Figure 14

"Intel® Cilk™ Plus—an integral part of Intel® compilers—comes with a SIMD pragma/directive that does what we are looking for: it ignores any compiler heuristic regarding efficiency, as well as any restrictions induced by the language to enforce vectorization."

Fortunately, both IVDEP and SIMD pragmas can still be applied. Albeit we encapsulated the arrays in objects and only access them via member functions, both pragmas can improve the situation for our example (**Figure 13**). And, all that is possible just by adding a single line.

Intel Cilk Plus Array Notations

Lastly, we'll take a completely different approach and use another facet of Intel Cilk Plus technology. We refactor our example to make use of Intel Cilk Plus array notations for C and C++ (see the "Learn More" section below). **Figure 14** shows a potential implementation, using the C version of our example as baseline.

The changes seem a bit complex at first glance. However, besides using the extended array notation syntax, we only split the arrays into blocks of 24. We use precisely this block size, because it is the least-common multiple of three (RGB values) and eight (maximum single-precision FP elements per Intel AVX vector). This enables the compiler to generate code using three SIMD vectors (or multiples thereof) at once. Both the array notation and the blocking provide enough information to improve performance even further (**Figure 7**).

Summary

IVDEP and SIMD pragmas/directives are easy to apply for C/C++ and Fortran. While changes are kept minimal, they can help improve

Intel® Xeon Phi™ Coprocessor

Compute clusters are large scale systems. Intel® Xeon Phi™ coprocessors can also be seen this way. The huge amount of cores, each with their own vector execution unit, makes it more beneficial to invest in optimization. Even small gains quickly increase overall performance by a magnitude. All features mentioned throughout this article can also be applied to Intel Xeon Phi coprocessors. The application of the examples is identical for native mode use of an Intel Xeon Phi coprocessor. If using the coprocessor in an offload mode, a little additional control is needed to specify the particulars. Enabling our example for an Intel Xeon Phi coprocessor is quite simple. Among the different explicit and implicit offloading models that the coprocessor offers, we show a basic, explicit offloading implementation. A possible C version could look like this:

```
void filter(
    unsigned int size,
    float * restrict imgMag, float * restrict imgPhase, // in & out
    float *filterMag, float *filterPhase,               // in
    float *gaussian)                                    // in
{
#pragma offload target(mic:0)              \
                inout(imgMag:length(size))    \
                inout(imgPhase:length(size))  \
                in(filterMag:length(size))    \
                in(filterPhase:length(size))  \
                in(gaussian:length(size))
#pragma simd vectorlength(16) assert
  for(int idx = 0; idx < size; idx++)
  { // idx%3: RGB values from DC, first 3 elements of "filterMag"
    imgMag[idx] /= (filterMag[idx] * 1/filterMag[idx%3] + filterMag[idx%3]);
    imgMag[idx] *= gaussian[idx];
    imgPhase[idx] -= filterPhase[idx];
  }
}
```

The explicit offloading pragma declares the loop to be executed on the first coprocessor target (mic:0) of the host system. All the data is transferred to the target before entering the loop (in/inout attributes), but only the two resulting image components are transferred back (inout attribute). We also applied the SIMD pragma here with increased vector length: Intel Xeon Phi coprocessors can handle up to 16 single-precision FP elements per vector.

The implementation above is just an example. In addition, offloading can:

- Transfer invariant data once and keep it on the target
- Asynchronous data transfer (double buffering)
- Conditional allocation and deallocation on the target
- Implicit offloading which shares memory between host and target, i.e., virtual shared memory (VSHM)
- And, much more (see documentation for Intel® MIC)

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

vectorization of an application and thus increase the overall performance. Intel compiler vectorization reports provide potential locations of where to apply the pragmas/directives. Combining all of that provides a powerful feature set to increase the quality of vectorization.

Intel Cilk Plus array notations on the other side require a small change in the existing paradigm. The benefit is that a different methodology can be a better solution for using the compiler to aid in vectorization. Intel Cilk Plus, part of Intel compilers, offers more technologies than covered here and addresses both vectorization and concurrency. It is an open specification that will be adopted in additional compilers in the future (there is already an experimental GCC branch which implements it). Please refer to the Intel Cilk Plus product page for a full description.

Further information about vectorization is provided online with our compiler documentation, knowledge base articles, and blogs. □

Learn More

Blog: http://software.intel.com/en-us/blogs/2012/11/01/parallel-universe-magazine-12-advanced-vectorization/

Intel® Cilk™ Plus: http://cilkplus.org/

Intel® Xeon Phi coprocessors and the Many Integrated Core (MIC) Architecture: http://intel.com/software/mic/

Intel® Composer XE: http://software.intel.com/en-us/intel-composer-xe/

DID YOU KNOW?

Strict ANSI aliasing: For historical reasons, the Intel® C and C++ compiler does not enable strict ANSI aliasing per default. Obeying strict ANSI aliasing rules provides more room for optimization. Hence, it is highly recommended to enable for ANSI-conforming code via "-ansi-alias" (Linux* and Mac OS* X) or "/Qansi-alias" (Windows*). This is already the default for the Intel® Fortran Compiler.

No aliasing of arguments: On Linux* and Mac OS* X the option "-fargument-noalias", "/Qalias-args-" on Windows* acts in the same way as applying the keyword "restrict" to all pointers of all function parameters throughout a compilation unit. For those platforms, this would have been another option for our example.

Alignment: In our example, no information about alignment of the arrays is provided. In such a case, the compiler generates multiple versions of function or loop bodies and code additions to select the most appropriate one during runtime. Such a test usually adds minimal overhead, but is already slightly noticeable in our example. In more complex scenarios, any missing information about aligned data can also hinder vectorization entirely. Hence, it is recommended to use pointer attributes for alignment (e.g. __declspec(align(…)), !DEC$ ASSUME_ALIGNED, etc.) or the vector pragma/directive (see below). See the Intel® compiler documentation for more information.

Vector pragma/directive: Allows some control over the vectorization of a loop. As with the SIMD pragma/directive, it can assert vectorization during compilation time. It also controls use of streaming stores, thus bypassing the cache. Alignment can also be ignored throughout the loop, freeing the compiler from another burden to create multiple code versions. Finally, it can also ignore compiler internal efficiency heuristics by applying the "always" clause. For our simple example, applying the vector pragma/directive would provide the same performance as when applying the SIMD pragma/directive, yet it has to obey the language standards and hence is less powerful. See the Intel® compiler documentation for more information.

BLOG highlights

Structured Parallel Programming with Deterministic Patterns

DR. MICHAEL MCCOOL, (Intel)

Many-core processors target improved computational performance by making available various forms of architectural parallelism, including but not limited to multiple cores and vector instructions. However, approaches to parallel programming based on targeting these low-level parallel mechanisms directly lead to overly complex, non-portable, and often unscalable and unreliable code.

A more structured approach to designing and implementing parallel algorithms is useful to reduce the complexity of developing software for such processors, and is particularly relevant for many-core processors with a large amount of parallelism and multiple parallelism mechanisms. In particular, efficient and reliable parallel programs can be designed around the composition of deterministic algorithmic skeletons, or patterns. While improving the productivity of experts, specific patterns and fused combinations of patterns can also guide relatively inexperienced users on developing efficient algorithm implementations that have good scalability.

The approach to parallelism described in this document includes both collective "data-parallel" patterns, such as map and reduce, as well as structured "task-parallel" patterns, such as pipelining and superscalar task graphs…

SEE THE REST OF MICHAEL'S BLOG:

Visit **Go-Parallel.com**

Browse other blogs exploring a range of related subjects at Go Parallel: Translating Multicore Power into Application Performance.

Optimizing

Correlation Analysis of Financial Market Data Streams Using Intel® Math Kernel Library

by Zhang Zhang, *Technical Consulting Engineer, Intel,*
Andrey Nikolaev, *Software Architect, Intel,*
and Victoriya Kardakova, *Software Development Engineer, Intel*

 For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

Intel® Math Kernel Library

Introduction

Online data analysis is becoming highly important in the financial industry—it supports real-time decision making and responsiveness to fluctuating market conditions. Here, we demonstrate that advanced algorithms and the right combination of hardware and software technologies lead to a high performance implementation, which is the key to any practical use of such analysis. As an example, we'll consider online detection of dependencies in the price movements of a large stock portfolio. This is an important component of technical financial analysis. The purpose is to find correlation patterns among the stocks, i.e., to see how the price movements of some stocks influence the price movements of others.

A noise filtration algorithm[1] has been developed for this type of analysis. The algorithm is compute-intensive. It requires high-performance software building blocks running on powerful hardware to produce results in a timely fashion. We describe an implementation using the Intel® Math Kernel Library (Intel® MKL). The advantages of an Intel MKL-based implementation are that Intel MKL readily provides all math functions needed by this algorithm, as well as the high performance we can achieve thanks to the highly tuned statistical and linear algebra functions in Intel MKL. We demonstrate that this implementation can attain ~29x speedup on an Intel® Xeon® E5-2600 system over a reference implementation based on non-optimized statistics functions and NetLib LAPACK*. We also show that the implementation can be easily extended to make use of the Intel® Xeon® Phi™ coprocessor.

Section 2 of this article discusses the methodology of the online noise filtration algorithm. Section 3 provides an overview of the math building blocks in the algorithm, describes the Intel MKL-based implementation, and then discusses performance results. Section 4 covers a straightforward port to the Intel Xeon Phi coprocessor. Section 5 summarizes the work and relates the techniques used in this article to similar statistical analyses.

Dependency Detection for Financial Market Data Streams

Computation of correlations is an important and basic instrument in stock data analysis. Its purpose is to reveal any statistical dependencies among different stocks. Straightforward computation of correlations, without any post-processing, can result in biased results as the price data that comes in the form of data streams is generally noisy, the number of observations in each data block is small, and the underlying statistical distribution is unknown. Thus, noise filtration is one of the early and basic stages in reliable data processing and analysis.

The algorithmic scheme proposed[1] resolves these issues and results in a more accurate estimation of dependencies and patterns in joint behavior of stock prices. The methodology is to split a correlation matrix—representing the overall dependencies in data—into two components, a "signal" matrix and a "noise" matrix. The signal matrix gives an accurate estimate of dependencies between stocks. The algorithm relies on an Eigen state-based approach that separates noise from useful information by considering the Eigenvalues of the correlation matrix for the accumulated data. In addition, it organizes the computation in a fashion that is particularly suitable for online and real-time analysis: it only needs information in the current data block to update the estimates of the signal and noise correlation matrices. It does not need historical price data.

We start with a general description of the algorithm and then provide additional details related to the steps of the algorithm.

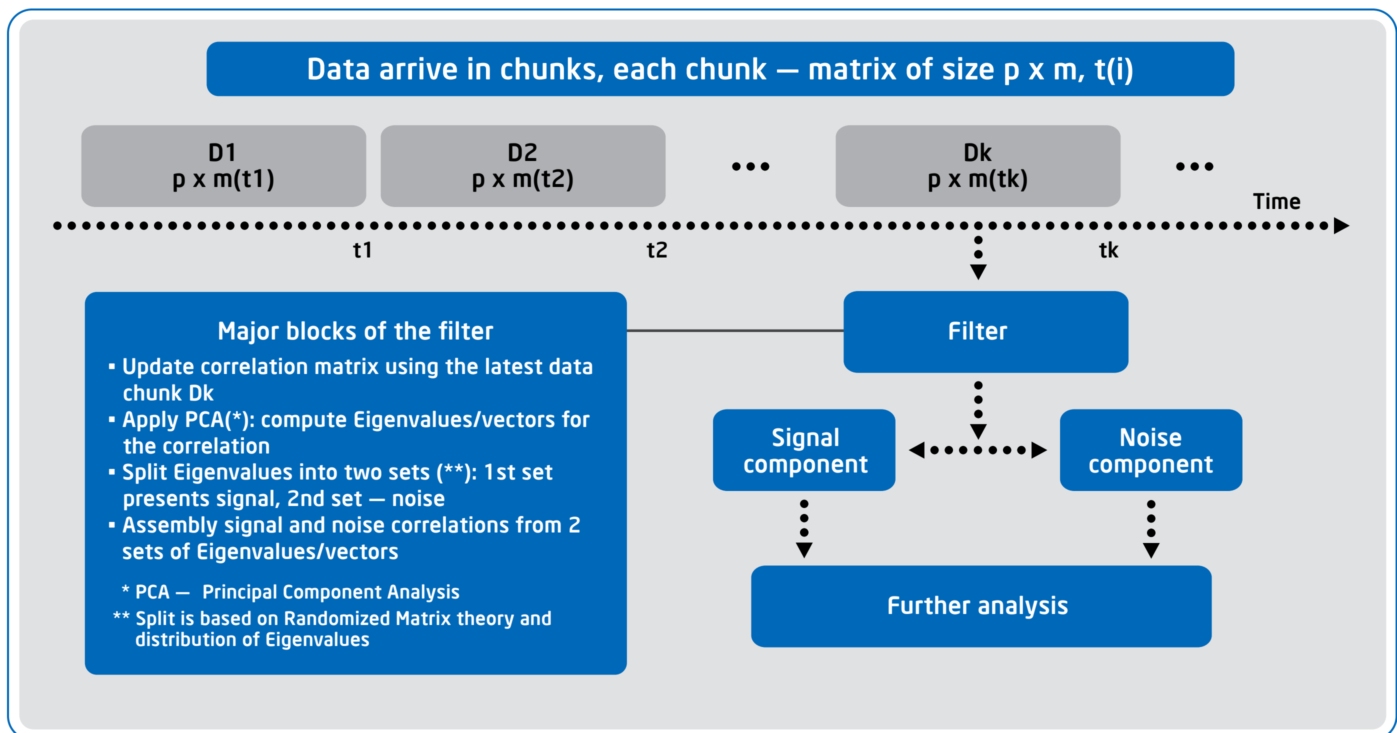
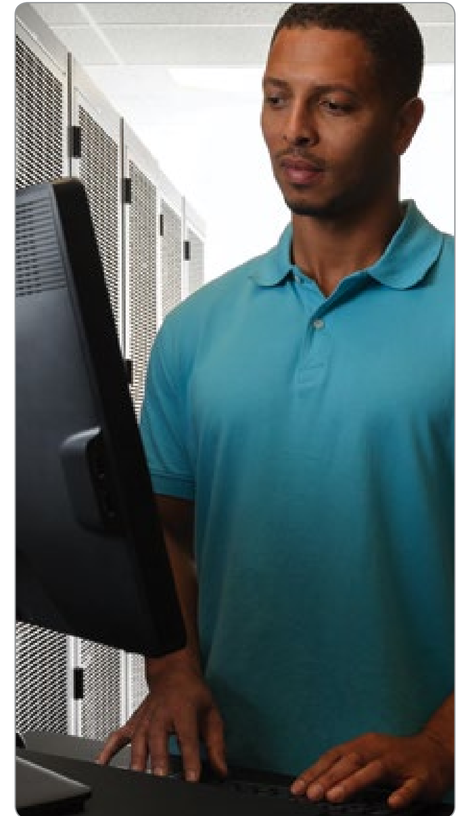


Figure 1: The online noise filtration algorithm.

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.



Online Noise Filtration Algorithm

The online noise filtration algorithm can be summarized as follows:

Step 1 Get a new block of data from the data stream.

Step 2 Update the correlation matrix using the latest data block.

Step 3 Compute the Eigenvalues/Eigenvectors that define the noise component, by searching those Eigenvalues of the correlation matrix belonging to the interval [$\lambda min, \lambda max$].

Step 4 Compute the correlation matrix of the noise component by combining Eigenvalues/Eigenvectors computed in Step 3.

Step 5 Compute the correlation matrix of the signal component by subtracting the noise component from the overall correlation matrix.

Step 6 If there is more data, go back to Step 1.

Figure 1 schematically depicts the algorithm and its main elements.

From algorithmic perspective, the computation of correlation matrix, Eigenvalues/Eigenvectors, and matrix operations are three essential components. Computational complexity of the components means that powerful hardware and fast software are key to the systems intended for practical use.

Let's consider the major elements of the algorithm in more detail. The data block arriving at time *t* is organized as a *p*-by-*m* matrix *D(t)*, where *p* is the number of stocks, and *m* the number of readings (observations) of the *p* stock prices. The information contained in *D(t)* is noisy in general. The signal/noise filtration in the algorithm is based on principal component analysis (PCA), which transforms a set of related variables into a smaller set of principal, linearly independent components. These components capture a good approximation of the original data with fewer variables by throwing out statistically insignificant information. To apply PCA we compute the correlation matrix *Cor(D)* for data block *D*: $Cor(D_c) = \frac{1}{(m-1)} D_c D_c^T$, where D_c is the normalized *D* (subtracting the mean and dividing by the standard deviation). Next, on this correlation matrix we compute Eigenvectors representing the basis of the source data and find the Eigenvalues defining the statistical importance of the corresponding vectors. The Eigenvalues are then used to separate signal and noise.

The step of filtering out noise is based on the theory of random matrices, which assumes that noise in the price data is represented by independent and identically distributed random variates. Under this assumption the Eigenvalues of the noise correlation matrix are known to follow statistical distribution described by probability density function[1] (see Karupta, 2002 for more details):

$$f_Q(x) = \begin{cases} \frac{Q\sqrt{(x-\lambda_{\min})(\lambda_{max}-x)}}{2\pi x}, \lambda_{min} < x < \lambda_{max} \\ 0, otherwise \end{cases}, Q = \frac{m}{p}, \lambda_{min} = \left(1-\frac{1}{\sqrt{Q}}\right)^2, \lambda_{max} = \left(1+\frac{1}{\sqrt{Q}}\right)^2.$$

Figure 2

Simply speaking, it means that the correlation matrix of the noise is constructed from the Eigenvalues belonging to the interval [$\lambda min, \lambda max$] and the corresponding Eigenvectors by using matrix operations. The correlation matrix of the signal is obtained by subtracting the noise component from the original correlation matrix.

"Intel® Math Kernel Library provides a set of functions for computing statistical estimates of multidimensional datasets."

Optimizing Online Noise Filtration Using Intel MKL

Intel MKL[2] is the industry-leading computational math library for applications that require maximum performance. It provides advanced performance optimizations for past, present, and future Intel® and compatible processors. The library provides rich collections of the algorithms that address a wide spectrum of problems in finance, engineering, and science. The high speed of Intel MKL functions relies on the latest advances in hardware and the intensive applications of instruction-level, data-level, and task-level parallelisms available in modern multicore CPUs. All kernels necessary to implement the online noise filtration algorithm are available in Intel MKL. These kernels include statistical functions, functions for solving symmetric Eigenvalue problems, and matrix operations.

Intel MKL Statistics Functions

Intel MKL provides a set of functions for computing statistical estimates of multidimensional datasets. Those functions rely on cutting-edge parallel algorithms of computational statistics, and provide simple interfaces that allow almost any statistical analysis task to be performed with only four steps. The functions that are of particular interest for this article are the ones that compute correlation matrix. Let's take a brief look at the usage model:

1. Initialize a summary statistics task and define the objects for our analysis: dataset x, its sizes (number of variables `p` and number of observations `m`), and the storage format `x_storage`:

```
status = vsldSSNewTask( &task, &p, &m, x, &x_storage, 0,0 );
```

2. Specify task parameters:
   - The memory intended to hold the correlation matrix:

```
status = vsldSSEditCovCor( task, mean, 0, 0, cor, cor_storage );
```

   - A two-element array intended to hold accumulated weights of observations processed so far (necessary for correct computation of estimates for data streams):

```
W[0] = W[1] = 0.0;
status = vsldSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, W );
```

3. Call the major compute driver by specifying computation type `VSL_SS_COR`, and computation method, `VSL_SS_METHOD_FAST`:

```
status = vsldSSCompute( task, VSL_SS_COR, VSL_SS_METHOD_FAST);
```

4. De-allocate resources associated with the task:

```
status = vslSSDeleteTask( &task );
```

Refer to related sections in the Intel MKL Reference Manual,[3] as well as the *Summary Statistics Library Application Notes,*[4] for detailed information on these functions.

"Straightforward computation of correlations, without any post-processing, can result in biased results as the price data that comes in the form of data streams is generally noisy, the number of observations in each data block is small, and the underlying statistical distribution is unknown. Thus, noise filtration is one of the early and basic stages in reliable data processing and analysis."

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

Intel MKL Eigenvector/Eigenvalue Functions

Step 3 of the algorithm involves solving an Eigenvalue problem for a symmetric matrix, one of the fundamental problems handled by the LAPACK package. Intel MKL offers a parallelized and highly optimized set of LAPACK functions that is API-compatible with the open source LAPACK library. In particular, it contains a set of drivers and computational routines to compute Eigenvalues and Eigenvectors for symmetric matrices of various properties and storage formats.

The online noise filtration algorithm requires computation of Eigenvalues that belong to the predefined interval, λmin and λmax. These Eigenvalues define noise in the data. The LAPACK river routine syevr is the default choice to solve this kind of problem. The syevr interface allows the caller to specify a pair of values, in our case corresponding to λmin and λmax, as the lower and upper bounds of the interval to be searched for Eigenvalues.

Intel MKL Matrix Operations

The Eigenvectors found are returned as columns of an orthogonal matrix A, and the Eigenvalues are returned in a diagonal matrix *Diag*. The correlation matrix for the noise component can be obtained as $ADiagA^T$. Instead of constructing a noise correlation matrix using two general matrix multiplications, this can be more efficiently computed with one diagonal matrix multiplication and one rank-n update operation:

$$Cor_{noise}(D) = ADiagA^T = A\sqrt{Diag}\,\sqrt{Diag}^T A^T = \left(A\sqrt{Diag}\right)\left(A\sqrt{Diag}\right)^T$$

Figure 3

For the rank-n update operation, Intel MKL provides the BLAS function syrk.

Source Code

The source code of our implementation comes with two versions, a baseline implementation and an optimized implementation. The baseline uses the open source Netlib LAPACK and BLAS

```
Initialization

#define P 450 /* # of stocks*/
*/  to #define M 1000 /* # of observations in block */
...
VSLSSTaskPtr task;
double x[P*M], mean[P], cor[P*P], W[2];
MKL_INT p, m, x_storage, cor_storage;

/* Initialize VSL Summary Stats task */
p = P; m = M;
x_storage = VSL_SS_MATRIX_STORAGE_COLS;
vsldSSNewTask( &task, &p, &m, x, &x_storage, 0,0 );

/* Set-up parameters of the task */
/* Specify memory for correlation estimate in task */
cor_storage = VSL_SS_MATRIX_STORAGE_FULL;
vsldSSEditCovCor( task, mean, 0, 0, cor, cor_storage );

/* Specify the parameter for progressive estimation of
 correlation */
W[0] = W[1] = 0.0;
vsldSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, W );
...
```

Figure 4: Initialize a correlation analysis task and its parameters

```
Computation

/* Set threshold that define noise component */
l1 = ( 1.0 – sqrt ( (double)p / m ) );
l1 = l1*l1;
l2 = ( 1.0 + sqrt ( (double)p / m ) );
l2 = l2*l2;
/* Loop over data blocks */
for ( nblock = 0; ; nblock ++ )
{
 /* Get the next chunk of size p x m into x */
    GetNextChunck( p, m, x );

 /* Update correlation estimate in cor */
 vsldSSCompute( task, VSL_SS_COR, VSL_SS_METHOD_FAST );

 /* Apply PCA and compute eigen-values that
  belong to (l1, l2) and define noise */
 dsyevr(...,&l1, &l2, ..., &evect_n);

 /* Assembly correlation matrix of noise */
 ...
 dsyrk( &evect_n, ..., cor_n,... );

 /* Compute correlation matrix of signal
  by substracting cor_n from cor */
}
```

Figure 5: Important steps involved in noise filtration for each block of data

```
De-Initialization

vslSSDeleteTask( task );
MKL_Free_Buffers();
...
```

Figure 6: Delete the task and release associated resources

libraries for the Eigen solvers and matrix operations. NetLib BLAS* routines are also used to build the correlation algorithm. The optimized version is developed by combining building blocks from Intel MKL, as discussed above.

The accuracy and performance of the implementations is tested using a dataset containing historical closing stock prices for 450 companies from the S&P 500 for a range of 9,608 trading days. In both implementations, we emulate data streaming by reading a block of 1,000 price vectors of size 450 at every time step. The structure of the source code, and some of the important steps in the optimized implementation are shown in **Figures 4–6**.

> "Although the problem of interest in this article is correlation analysis of financial market data, the principles and statistical analysis techniques used can find applications in many other fields, such as data mining, machine learning, pattern recognition, and bioinformatics."

For more information regarding performance and optimization choices in Intel® software products, visit http://software.intel.com/en-us/articles/optimization-notice.

Performance

The performance of the two implementations was measured on a 16-core Intel® Xeon® E5-2690 processor. The configuration of the test platform is:

- Intel Math Kernel Library (Intel MKL) 11.0
- Hardware: Intel Xeon Processor E5-2690, two eight-core CPUs (20Mb L3 Cache, 2.9GHz), 32GB of RAM
- Operating System: RHEL 6 GA x86_64
- Benchmark Source: Intel Corporation

The benchmarking results show that the optimized implementation runs ~29 times faster than the baseline implementation. Intel MKL functions tuned for multicore architectures and their effective exploiting of data and instruction level parallelisms are helping to enable this huge performance difference.

	Seconds per Block	Speedup
Baseline implementation	0.883	1.0
Optimized implementation (using Intel® MKL)	0.031	28.9

Table 1

Using an Intel Xeon Phi Coprocessor

The highly parallel online nature of the noise filtration algorithm is a great fit for an efficient, natural implementation using the Intel Xeon Phi coprocessor. There are several key advantages for using a coprocessor for this type of data analysis:

- The programming models for processors and the coprocessor are the same. That is, the coprocessor does not require new development of the algorithm. The same set of software development tools, such as the Intel® C++ compiler and Intel MKL, support both platforms and the transparent communication between them;
- The number of cores on the coprocessor is substantially larger and the vector registers are wider. These help Intel MKL to get additional performance advantages on the coprocessor;
- Offloading computations to the coprocessor frees up resources on the host for other tasks, and just one thread is necessary to support communication between host and coprocessor.

```
Offload Noise Filtration with Asynchronous Data Transfer

...
for (k = 0; k < nBlocks; k++)
{
  /* Get the next data block */
  status = nfReadNextChunk(..., xblock );

  /* Start asynchronous transfer of block xshared to coprocessor */
  #pragma offload_transfer target(mic:0) wait(res) \
    in(xblock:length(m*p) alloc_if(0) free_if(0))signal(xblock)
  {}

  /* Offload noise filtration computations to the coprocessor */
  #pragma offload target(mic:0) in(params) in(res_buf_sz)    \
    wait(xblock)                         \
    nocopy(ssTask)                       \
    nocopy(xblock:length(m*p) alloc_if(0) free_if(0))     \
    nocopy(buf:length(buf_sz) alloc_if(0) free_if(0))     \
    out(res:length(res_sz) alloc_if(0) free_if(0))      \
    signal(res)
  {
    NoiseFiltr(&params, xblock, buffer, res, ssTask);
  }
}
/* Wait for the result of the last iteration */
#pragma offload_wait target(mic:0) wait(res)
{}
```

Figure 7: Offload noise filtration with asynchronous data transfer

Several approaches are available to arrange communication between host and coprocessor. In our implementation, we rely on asynchronous data transfer. This mechanism allows us to get additional performance benefits by overlapping computation and communication. Upon offloading the data block and initiating computation on the coprocessor, the host immediately moves to get and preprocess the next block of data. The host gets back the results once the coprocessor signals its availability. Asynchronous data transfer is supported by Intel C++ compiler and can be arranged using relevant Intel-specific pragmas. The skeleton of the code for offloading with asynchronous data transfer is shown in **Figure 7**.

The algorithm implemented in `NoiseFiltr` is identical to the one discussed earlier. The computational flow is mostly the same. To control data transfer and offloading, we add corresponding compiler pragmas to the necessary elements of the loop. It is worth noting that the pragmas can be easily disabled, so the code can be recompiled to run on the host.

When the *i-th* data block is obtained through the function `nfReadNextChunk` the host comes to a state of waiting for the filtration results to arrive in buffer `res`. The results were computed on the previous iteration, for the *(i-1)-th* block, by the coprocessor. Once the results arrive, the host resumes and transfers the *i-th* data block `xblock` to the coprocessor and moves to the next statement. The data transfer is achieved using a single pragma `#pragma offload_transfer`. Next, the host initiates noise filtration on the coprocessor using `#pragma offload`. The `signal` clause in the pragma dictates the host to immediately move to reading the *(i+1)-th* data block, in parallel with the computation on the coprocessor. Note that filtration starts only after the *i-th* data block is received by the coprocessor. The results of analysis of the *i-th* block are expected by iteration *(i+1)*.

This coprocessor-oriented implementation has advantages only if the cost of computation dominates data transfer overheads. To some extent, asynchronous data transfer helps to hide the overhead. The key is to choose a right data block size, such that there is enough parallelism to be exploited in each block to achieve the optimal speedup, while still keeping data transfer overhead relatively low.

“Here, we demonstrate that advanced algorithms and the right combination of hardware and software technologies lead to a high performance implementation.”

Conclusion

This article demonstrates the superb performance advantages of Intel MKL in the implementation of the online noise filtration algorithm on Intel Xeon processors. It also demonstrates a straightforward port to the Intel Xeon Phi coprocessor. Although the problem of interest in this article is correlation analysis of financial market data, the principles and statistical analysis techniques used can find applications in many other fields, such as data mining, machine learning, pattern recognition, and bioinformatics. A common problem in these applications is transforming data in a highly dimensional space to a space with a reduced number of dimensions, i.e., dimensionality reduction. Principal component analysis (PCA) and the closely related linear discriminant analysis (LDA) are the most frequently used methods of dimensionality reduction. These methods require computation of statistical estimates for the raw data, such as variance, covariance, and correlation. They also typically involve linear transformation of large datasets. Intel MKL offers highly optimized and extensively threaded summary statistics functions and linear algebra functions on both Intel Xeon architectures and Intel Xeon Phi coprocessors, and should be considered the math library of choice in powering these data-oriented and compute-intensive applications. □

Learn More

Source Code: http://software.intel.com/sites/default/files/article/327618/nf-bench6.zip

1. H. Kargupta, K. Sivakumar, S. Ghost (2002). A Random Matrix-Based Approach for Dependency Detection from Data Streams. In Proceedings of PKDD'2002 (pp. 250-262). Springer-Verlag.

2. Intel® Math Kernel Library Product Page: http://software.intel.com/en-us/articles/intel-mkl/

3. Intel® Math Kernel Library Reference Manual: http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mklman/index.htm

4. Summary Statistics Library Application Notes: http://software.intel.com/sites/products/documentation/hpc/mkl/sslnotes/index.htm

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

RESOURCES AND SITES OF INTEREST

Go Parallel

The mission of Go Parallel is to assist developers in their efforts toward "Translating Multicore Power into Application Performance." Robust and full of helpful information, the site is a valuable clearinghouse of multicore-related blogs, news, videos, feature stories, and other useful resources.

"What If" Experimental Software

What if you could experiment with Intel's advanced research and technology implementations that are still under development? And then what if your feedback helped influence a future product? It's possible here. Test drive emerging tools, collaborate with peers, and share your thoughts via the What If blogs and support forums.

Intel® Software Network

Check out a range of resources on a wide variety of software topics for a multitude of developer communities ranging from manageability to parallel programming to virtualization and visual computing. This content-rich collection includes Intel® Software Network TV, popular blogs, videos, tools, and downloads.

Step Inside the Latest Software

See these products in use, with video overviews that provide an inside look into the latest Intel® software. You can see software features firsthand, such as memory check, thread check, hotspot analysis, locks and waits analysis, and more.

Intel® Inspector XE

Intel® VTune™ Amplifier XE

Intel® Software Evaluation Center

The Intel® Software Evaluation Center makes 30-day evaluation versions of Intel® Software Development Products available for free download. For high-performance computing products, you can get free support during the evaluation period by creating an Intel® Premier Support account after requesting the evaluation license, or via Intel® Software Network Forums. For evaluating Intel® Parallel Studio, you can access free support through Intel Software Network Forums ONLY.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Free updates and fast downloads on even more new software technologies, tools, and best practices for smart coding and innovative user experiences.

> Join Intel® Software Dispatch.

Sign up for future issues | Share with a friend

The Parallel Universe is a free quarterly magazine. **Click here** to sign up for future issue alerts and to share the magazine with friends.

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

Design
Tune
Build
Verify
Intel Parallel Studio XE

CodeBook
Intel® Parallel Studio XE 2013

intel
Software

Boost performance and accuracy

This downloadable CodeBook provides "how-to" guidance and a comprehensive resource toolkit to help you efficiently produce fast, scalable, reliable applications throughout the development lifecycle.

Look for guidance and techniques for C++ and Fortran developers:

Tools and techniques across the development lifecycle

Technical guides, white papers, articles, and blogs

Features for accelerated performance

And much more

DOWNLOAD THE FREE CODEBOOK NOW

intel®
Software

©2012, Intel Corporation. All rights reserved. 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® SOFTWARE ADRENALINE

Introducing Intel® Software Adrenaline

Enter a world of doers, dreamers, and software industry luminaries.

This new magazine puts you on the forefront of the software industry. Roadmaps, R&D, industry pioneers and game changers, news, and more.

FREE SUBSCRIPTION
softwareadrenaline.intel.com

© 2012, Intel Corporation. All rights reserved. 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.