

The Parallel Universe



Full throttle: **OpenMP* 4.0**

Introducing
Intel® Cluster Studio XE 2013 SP1



Issue
16
2013

CONTENTS

Letter from the Editor **3** **Performance Hits the Streets**

By James Reinders

Introducing Intel® Cluster Studio XE 2013 SP1 **4**

By James Tullos

Provides a quick reference to the newest features of this HPC software development tool suite. These include the latest improvements to the Intel® MPI Library and the Intel® Trace Analyzer and Collector to help distributed memory programs run faster and more effectively.

FEATURE

Full throttle: OpenMP* 4.0 **6**

By Michael Klemm and Christian Terboven

OpenMP takes a quantum leap with new features supporting OpenMP tasks, SIMD instructions, and the effective integration of application code, third-party libraries, and hardware to achieve a highly efficient solution.

Profiling MPI Communications—Techniques for High Performance **17**

By James Tullos

Focuses on Intel® Trace Analyzer and Collector (ITAC), a performance analysis tool which is part of Intel® Cluster Studio XE SP1. ITAC provides the ability to profile and analyze MPI applications to find areas for performance improvement.

Pexip Speeds Videoconferencing with Intel® Parallel Studio XE **30**

By Stephen Blair-Chappell

See how Pexip has been able to match, and even exceed, the performance of traditional conferencing systems—designing a cost-effective alternative with expanded processing capabilities.



LETTER FROM THE EDITOR

James Reinders, Director of Parallel Programming Evangelism at Intel Corporation.

James is a coauthor of two new books from Morgan Kaufmann, [Intel® Xeon Phi™ Coprocessor High Performance Programming](#) (2013), and [Structured Parallel Programming](#) (2012). His other books include [Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism](#) (O'Reilly Media, 2007, available in English, Japanese, Chinese, and Korean), and [VTune™ Performance Analyzer Essentials](#) (Intel Press, 2005).



Performance Hits the Streets

High-performance computing (HPC) is no longer the provenance of research clusters and high-speed stock trades. HPC capabilities are now in demand by a broad range of vertical industries and global enterprises seeking to leverage the competitive advantage of big data and the performance opportunity of real-time *everything*. This issue focuses on the HPC software tool capabilities and programming models that are both challenging and inspiring developers and software industry decision makers.

Our feature article, *Full throttle: OpenMP* 4.0*, explores a quantum leap in OpenMP features, supporting task-based, heterogeneous programming models applicable beyond the scientific and research community.

Introducing Intel® Cluster Studio XE 2013 SP1 provides a brief overview of the newest features of this HPC software development tool suite. Find out how to apply capabilities from distributed memory and MPI programming to enhanced support for Infiniband* fabrics, while maximizing performance with the Intel® Xeon Phi™ coprocessor.

Profiling MPI Communications—Techniques for High Performance takes a close look at optimization techniques for MPI application performance. You'll find techniques for finding—and correcting—HPC application bottlenecks and imbalances.

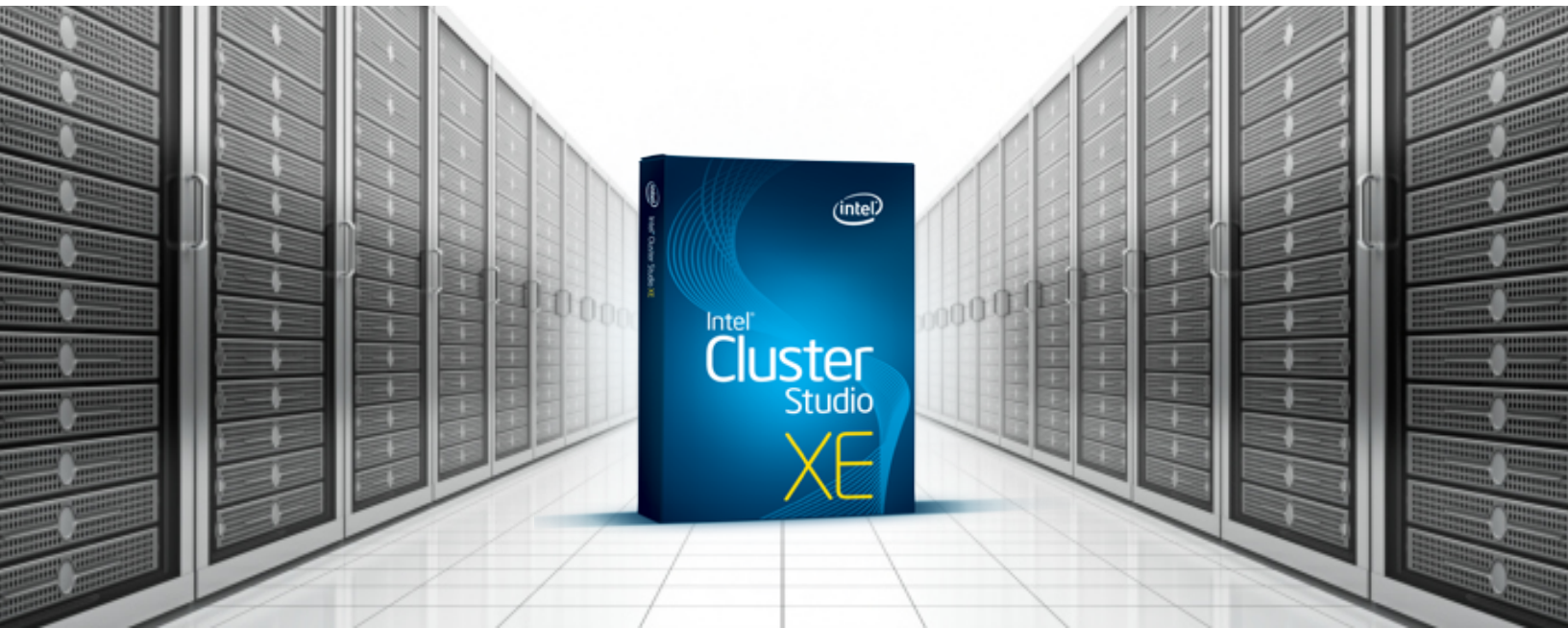
Pexip Speeds Videoconferencing with Intel® Parallel Studio XE provides a case study on the optimization of enterprise-class productivity tools. This innovative start-up company, founded by former Cisco and Tandberg executives, is tapping into the performance gains made possible, in part, by Intel® software tools to differentiate its product and support real-time data processing.

Whether you are designing the next wave of NASA exploration, creating analytics algorithms, or reshaping business productivity tools such as videoconferencing, we hope you will find useful techniques that support your participation in a world on performance overdrive.

James Reinders

November 2013





Introducing Intel® Cluster Studio XE 2013 SP1

by James Tullos, *Technical Consulting Engineer, Intel*

Intel® Cluster Studio XE 2013 SP1 contains all features found in Intel® Parallel Studio XE 2013 SP1, while also supporting distributed memory programming and, in particular, MPI programming. This means it includes the latest improvements to the Intel® MPI Library and the Intel® Trace Analyzer and Collector to help your distributed memory programs run faster and more effectively. (For more information, see: <http://software.intel.com/en-us/articles/intel-parallel-studio-xe-2013-sp1-release-notes>.)

The latest version of the Intel MPI Library includes enhanced support for all InfiniBand* fabrics. Specifically, we have improved the process for automatically selecting DAPL providers, improved the scalability for OFA, added support for TMI when using the Intel® Xeon Phi™ coprocessor, and added support for Microsoft Network Direct*.

If you are running MPI jobs with the Intel Xeon Phi coprocessor in either native or symmetric mode, you now have up to three DAPL providers for different situations. The first DAPL provider is used for small messages, the second is used for large messages within a single physical node (including between different Intel Xeon Phi coprocessors on the same node), and the third is used for large messages involving multiple physical nodes. Use `I_MPI_DAPL_PROVIDER_LIST` to define a comma-separated list of DAPL providers to use.

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



In addition to InfiniBand improvements, the product features improved pinning and job management capabilities. If using the checkpoint-restart functionality, setting `I_MPI_RESTART=1` will enable your application to restart from a stored checkpoint. Improved pinning support for both traditional and non-uniform memory access (NUMA) systems has been added. For Windows*, we have added Hydra as an experimental scalable process manager. You can try it out by replacing `mpiexec` with `mpiexec.hydra` for launching your application. Support for programs utilizing the offload model with the Intel Xeon Phi coprocessor is also now available for host systems running Windows.

In the newest version of the Intel Trace Analyzer and Collector, we have made significant enhancements. The graphical user interface (GUI) now includes a Trace Map, which always displays the full execution time, to help easily navigate through a large set of data, even when zoomed. The Trace Map also gives a visual representation of how many ranks are performing MPI calls at any given time. All of the timeline settings are now available as preferences to allow customization of how traces are opened and viewed. Finally, you can instantaneously access context-sensitive help from anywhere in the GUI with the push of a button.

Please visit www.intel.com/go/clustertools to get more information about Intel Cluster Studio XE 2013 SP1 and all of the suite components. Release notes detailing the new features can be found at <http://software.intel.com/en-us/articles/intel-cluster-studio-xe-2013-sp1-release-notes>. While visiting the site, please sign up for the evaluation software and try out the new features to see the benefit for your applications. ●

BLOG HIGHLIGHTS

Common Vectorization Tips

BY WENDY DOERNER >>

Compiler Methodology for Intel® Manycore Integrated (Intel® MIC) Architecture

Handling user-defined function calls inside vector-loops

If you want to vectorize a loop that has a user-defined function call, possibly refactor the code and make the function-call a vector-elemental function.

Specifying unit-stride accesses inside elemental functions

If your SIMD-enabled function accesses memory in unit stride, these are the two ways you can write:

- > Uniform pointer indexed by linear integer
- > Linear pointer

```
__declspec(vector(uniform(a),linear(i:1)))
float foo(float *a, int i){
    return a[i]++;
}
__declspec(vector(linear(a:1)))
float foo1(float *a){
    return (*a)++;
}
```

Handling memory disambiguation inside vector loops

Consider vectorization for a simple loop:

More





Full Throttle: OpenMP* 4.0

By **Michael Klemm**, *Senior Application Engineer, Intel*
and **Christian Terboven**, *Deputy Head of HPC Group, RWTH Aachen University*

Introduction

“Multicore is here to stay.” This single sentence accurately describes the situation of application developers and the hardware evolution they are facing. Since the introduction of the first dual-core CPUs, the number of cores has kept increasing. The advent of the Intel® Xeon Phi™ coprocessor has pushed us into the world of manycore—where up to 61 cores with 4 threads each impose new requirements on the parallelism of applications to exploit the capabilities of the hardware.

It is not only the ever-increasing number of cores that requires more parallelism in an application. Over the past years, the width of SIMD (Single Instruction Multiple Data) registers has been growing. While the early single instruction multiple data (SIMD) instructions of Intel® MMX™ technology used 64-bit registers, our newest family member, Intel® Advanced Vector Instructions 512 (Intel® AVX-512), runs with 512-bit registers. That’s an awesome 16 floating-point numbers in single precision, or eight double-precision numbers that can be computed in one go. If your application does not exploit these SIMD capabilities, you can easily lose a factor of 16x or 8x compared to the peak performance of the CPU.

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

[Sign up for future issues](#)

[Share with a friend](#)



It is key for application developers to keep up with the evolution of the hardware in terms of both number of cores and SIMD capabilities. Today's applications must exploit multiple levels of parallelism to make use of the compute power of today's and tomorrow's CPUs. Multithreading alone will not scale to the future. In addition, applications increasingly make use of plugins and libraries that are also written with parallelism in mind. Programmers now need to orchestrate the cooperation between their application code, third-party libraries, and hardware to achieve an efficient solution.

One possible solution to this are task-based programming models to describe how the application can be decomposed into concurrent tasks that can be independently executed. In contrast to traditional thread-based programming, benefits include a much more flexible mapping of application tasks to the execution units, and a much easier interaction between components. Each component can just create tasks, which then automatically intermix with all other tasks of the application. Programming models such as Intel® Cilk™ Plus, Intel® Threading Building Blocks (Intel® TBB), or C++11's **async** feature are good examples.

My Name is OpenMP*

Since its introduction in 1997, the OpenMP API emerged as a de-facto standard for shared-memory parallel programming. With a focus on technical and scientific computing, it supports C, C++, and Fortran. OpenMP compilers can be found for all mainstream platforms. OpenMP is "open," since anyone can implement parts or the full specification without any licensing costs.

OpenMP consists of directives that describe how the code should be parallelized by the OpenMP compiler. It also defines an API and environment variables to control the runtime behavior of the code. The directives are implemented through pragmas in C/C++ or special comments in Fortran. Because of this, OpenMP code can usually also be compiled into a sequential version by ignoring the pragmas or comments. OpenMP makes it possible to incrementally add parallelism to existing code as well as to focus on the compute-intensive parts of the code.

The fundamentals of OpenMP are parallel regions, in which the "master thread" is joined by so-called "worker threads." Outside of the parallel regions, the code runs sequentially in the master thread. Worksharing constructs provide a means to distribute work across the team of threads (**Figure 1**). In this example, the **for** loop is cut into the equal-size chunks which are assigned to the threads of the team. If this static distribution does not fit,

```

1 const int N = 1000000;
2 double A[N], B[N], C[N];
3 // initialize A, B, and C
4 // ...
5
6 #pragma omp parallel for
7 for (int i = 0; i < N; i++) {
8     A[i] = B[i] + C[i];
9 }
    
```

1

Simple example of a parallel region with worksharing of a **for** loop



because it creates a load imbalance, the schedule clause can change the default distribution scheme. For instance, `schedule(dynamic, c)` creates chunks of size `c`, and idle threads grab the next available chunk.

OpenMP also provides mechanisms to describe the visibility of data to the threads (“scoping”). It is important to tell the OpenMP compiler what data must remain in the shared memory domain and what data needs to be private to the individual threads. OpenMP defines clauses that can be added to the directives to control the scope of variables. The `shared` clause keeps a variable in the shared space, while the `private` clause creates a thread-private copy of a variable. Shared scope is the default for variables that are declared outside of a parallel region. In **Figure 1**, this applies to the variables `A`, `B`, `C`, and the constant `N`. Private variables can store different values for different threads, as needed by the loop counter `i`, in the example. Private copies are by default created without initializing; `firstprivate` can be used to assign the value of the variable outside of the parallel region.

The latest version 4.0 of the OpenMP API specification not only includes minor bug fixes and improvements to existing features, it now supports a good share of features introduced with Fortran 2003. OpenMP affinity defines a common way to express thread affinity to execution units of the hardware. Version 4.0 also comes with major feature enhancements, some of which will be discussed in more detail here. Task groups improve tasking by providing a better way to express synchronization of a set of tasks and to handle cancellation, which allows to stop parallel execution. SIMD pragmas extend the thread-parallel execution to data-parallel SIMD machine instructions, while user-defined reductions let programmers specify arbitrary reduction operations. Possibly the biggest addition to OpenMP is support for offloading computation to coprocessor devices.

Talk the Talk, Task the Task

The growing number of cores (and threads) make it harder to fully utilize the cores with traditional worksharing constructs for parallel loops. Irregular algorithms, such as recursions and traversals of graphs, require a completely different approach to parallelism. Task-based models blend well with the requirements of these algorithms, since tasks can be created in a much more flexible way.

An OpenMP task may be treated as a small package that consists of a piece of code to be executed and all the data needed for execution. An OpenMP task is created through the `#pragma omp task` directive to mark a piece of code and data for concurrent execution. The OpenMP runtime system takes care of mapping the created tasks to the threads of a parallel region. It may defer the task for later execution by adding it to a task queue or it may execute the task immediately.

Figure 2 shows a task-parallel version of a very simple, brute-force Sudoku* solver. The idea of the algorithm is:

1. Find an empty field without a number
2. Insert a number
3. Check the Sudoku board
4. If the solution is invalid, try the next possible number
5. If the solution is valid, go to the next field and start over




```

1 void main() {
2 // setup data structures
3 // ...
4 #pragma omp parallel // start parallel region
5 {
6 #pragma omp single // limit to one thread
7 {
8 #pragma omp taskgroup // group all tasks
9 {
10 solve_parallel(0, 0, sudoku);
11 }
12 }
13 } // end omp parallel
14 }
15
16 void solve_parallel(int x, int y, CSudokuBoard* sudoku, CSudokuBoard* & solution) {
17 if (x == sudoku->getFieldSize()) { // end of Sudoku line
18 y++; x = 0;
19 if(y == sudoku->getFieldSize()) // end of Sudoku field
20 return true;
21 }
22
23 if (sudoku->get(y, x) > 0) { // field already set
24 return solve_parallel(x+1, y, sudoku); // tackle next field
25 }
26
27 for (int i = 1; i <= sudoku->getFieldSize(); i++) { // try all possible numbers
28 if (!sudoku->check(x, y, i)) {
29 #pragma omp task firstprivate(i,x,y,sudoku) // create new solver task
30 {
31 CSudokuBoard* new_sudoku = new CSudokuBoard(*sudoku);
32 new_sudoku->set(y, x, i); // if number fits, set it
33 if (solve_parallel(x+1, y, new_sudoku)) { // tackle next field
34 #pragma omp critical // protect Sudoku solution
35 if (!solution) { // if new solution, save it
36 solution = new_sudoku;
37 #pragma omp cancel taskgroup // request cancellation
38 }
39 }
40 delete new_sudoku; // clean up
41 }
42 }
43 }
44 #pragma omp taskwait // await completion of child tasks
45
46 sudoku->set(y, x, 0); // no solution found, reset field
47 }

```

Sudoku* solver example with OpenMP* tasks and cancellation



To execute tasks, a parallel region first creates the team of threads. In our example, only one thread needs to start with the Sudoku algorithm, as the algorithm starts task creation when fired up. The algorithm creates tasks in step four above. Trying to solve the Sudoku board for different configurations of a number of a given field can be parallelized: each task can try a different number and check the board for a valid solution.

The example also shows that combining C++ classes and OpenMP is easy. The variable `sudoku` is a pointer to instances of the `CSudokuBoard` class and the `firstprivate` indicates that each task received a private copy of that pointer. The tasks can then create a copy of the instance by invoking its copy constructor.

All threads of a team participate in executing deferred tasks from the task queue. If threads run into a barrier or some other synchronization construct, they can check for available tasks and execute them. It is explicitly allowed by OpenMP that undersupplied threads steal tasks from overloaded threads.

OpenMP offers several synchronization constructs to synchronize the execution of tasks. Barriers guarantee that all tasks created before reaching a barrier have been executed when the barrier is left by the team. The `taskwait` construct waits for the completion of all child tasks created by a parent task. Finally, the `taskgroup` construct logically groups all tasks created within the construct and establishes an implicit `taskwait` for all tasks of the group at the end of the construct.

Stopping Midstream

Sometimes you may want to abort a parallel computation because of an unforeseen situation (e.g., a failure or error) that prevents execution from continuing cleanly. Another reason might be that the result has been computed and it does not make much sense to continue execution. Examples are numerical algorithms or search algorithms. With OpenMP 3.1, programmers could not easily implement such algorithms with OpenMP, since it did not support stopping a parallel execution once it had been started. For some OpenMP constructs, there have been workarounds. For instance, stopping a `for` worksharing construct involved an `if` statement that turned the `for` loop into an empty loop that ran to its natural end without doing any more work. Despite being a dirty hack, the loop continued to run, which costs precious energy and consumes time.

OpenMP 4.0 solves this problem by introducing directives to cleanly abort parallel execution. The key directive is the `cancel` directive to request termination of the current OpenMP region. The requesting thread immediately stops execution and notifies the remaining threads about the request for termination. If they receive a notification, the threads check at so-called “cancellation points” and stop execution.

The `cancel` directive supports termination of `parallel` regions, worksharing constructs, and task groups. Cancellation points are automatically inserted at barriers and the `cancel` directive. Programmers can add additional cancellation points to the code through the `cancellation point` directive. When cancellation occurs, the OpenMP runtime does not release any acquired resources such as allocated memory, locks, or open files. It is the programmer’s responsibility to clean up before cancellation is requested, or before a thread hits a cancellation point that triggers cancellation.



The Sudoku example in **Figure 2** uses cancellation to stop searching for a solution of the Sudoku board. If one of the solver tasks found a new solution, it first checks whether some other task has already found another solution. If it is the first solution found, the task saves the solution and requests cancellation. We use a **critical** region to avoid a potential race condition on the solution and to avoid two tasks asking for cancellation at the same time.

In the example, we rely on the behavior of task cancellation. All tasks that have started execution may run to completion and are not aborted unless they contain a cancellation point. All other tasks sent to the waiting queue are discarded and considered completed. This explains why each task must always check for a solution already found. We cannot know if a task might start execution ahead of time. Hence, we need a safety net to avoid storing a duplicate solution if a task slips into execution shortly after one task has found solution and requested cancellation.

SIMD Me

OpenMP 3.0 and earlier versions focused solely on multithreading and left other topics, such as data-parallel SIMD instructions, to other paradigms. If a programmer wanted to exploit the SIMD features of modern processors, she was left hoping for the compiler’s auto-vectorizer to be smart enough to insert appropriate SIMD instructions. Otherwise, she had to use vendor-specific extensions, which are not easily portable and were found to be problematic in combination with OpenMP’s parallelization directives.

OpenMP 4.0 aims to improve this situation. It defines new constructs, allowing for the portable description of SIMD expressions and their combination with parallelization directives. The main building block for this is the **simd** construct to vectorize loops (**Figure 3**). It advises the compiler to introduce appropriate SIMD instructions

3

```

1  #pragma omp declare simd aligned(a,b) notinbranch
2  float min(float a, float b) {
3      return a < b ? a : b;
4  }

5  #pragma omp declare simd aligned(x) uniform(y) notinbranch
6  float distance(float x, float y) {
7      return (x - y) * (x - y);
8  }

9  void distance_update(float *a, float *b, float *y, int vlen) {
10     float *ptr = b;
11     #pragma omp parallel for simd safelen(16) linear(ptr:1) aligned(a,b,y)
12     for (int i=0; i<vlen; i++) {
13         y[i] = min(sqrt(distance(a[i], 1.0)), ptr);
14         ptr += 1;
15     }
16 }

```

Vectorization of functions and loop with OpenMP* SIMD directives



into the serial code. The **for simd** and **parallel for simd** constructs combine this aspect with the well-known loop-level thread parallelization, allowing for an efficient combination of both paradigms simultaneously.

The syntax of the new construct closely matches the existing OpenMP worksharing constructs and supports a large set of their established clauses (such as **private**, **reduction**, **collapse**, etc.), albeit with slightly revised semantics. Additionally, there are some new clauses that aid the compiler in creating efficient SIMD code. The **safelen** clause defines the maximum possible vector length for a loop, for example, in the presence of dependencies between loop iterations with a specific stride. Similarly important is **linear**, expressing a linear dependence of a variable to the loop counter. The **aligned** clause specifies data alignment to help the compiler choose optimal load and store instructions.

A challenge of vectorizing codes is when the loop body contains function calls, as shown in **Figure 3**. If there is no SIMD version of the functions **min** and **distance**, then the compiler cannot generate SIMD instructions. For many routines of the standard library, modern compilers already offer vectorized versions (e.g., **sqrt**, **sin**, **cos**). For all other functions, the programmer has to help again. In **Figure 3**, the **declare simd** construct instructs the compiler to generate an additional vector version by promoting scalar arguments to vectors. If necessary, the **uniform** clause avoids SIMD promotion for certain arguments. The **notinbranch** clause assures that the function will never be called from within a conditional branch (e.g., in the body of an **if** statement), hence allowing further compiler optimizations. The **inbranch** clause asserts the opposite.

Reducers Everywhere

Reductions are involved whenever a team of threads cooperatively work on a problem and have to produce a single, global result. Each thread receives a private copy of a variable to collect their local intermediate results. Shortly before parallel execution ends, the OpenMP runtime system collects all the intermediate results from all threads and reduces them into the global result. Before OpenMP 4.0, only predefined reductions operations, such as addition or minimum/maximum on primitive types of the base language (e.g., **int** or **float**), have been supported. Programmers previously had to write their own reduction code if derived data types or more complex operations had to be used. This resulted in more or less complex code patterns that had to be maintained by programmers.

OpenMP 4.0 includes support for user-defined reductions. Programmers can now define arbitrary reduction operations on arbitrarily complex data types. **Figure 4** uses the new feature to implement a parallel algorithm to compute the bounding box of a cloud of 2D points. The bounding box is the smallest rectangle that contains all points. It is computed in 2D by determining the left-most and right-most, as well as lowest and highest, point of the cloud. These locations give you the lower left and upper right corner of the rectangle.



```

1  #include <algorithm>
2
3  #pragma omp declare reduction(minp : Point2D : \
4      omp_out.setX(std::min(omp_in.getX(), omp_out.getX())),
5      omp_out.setY(std::min(omp_in.getY(), omp_out.getY())) )
6      initializer(omp_priv = Point2D(MAX_X, MAX_Y))
7  #pragma omp declare reduction(maxp : Point2D : \
8      omp_out.setX(std::max(omp_in.getX(), omp_out.getX())),
9      omp_out.setY(std::max(omp_in.getY(), omp_out.getY())) )
10     initializer(omp_priv = Point2D(MIN_X, MIN_Y))
11
12 Rectangle bounding_box(std::vector<Point2D> points) {
13     Point2D lb(MAX_X, MAX_Y);
14     Point2D ub(MIN_X, MIN_Y);
15     std::vector<Point2D>::iterator it;
16 #pragma omp parallel for reduction(minp:lb) reduction(maxp:ub)
17     for (it = points.begin(); it != points.end(); it++) {
18         Point2D &p = *it;
19         lb.setX(std::min(lb.getX(), p.getX()));
20         lb.setY(std::min(lb.getY(), p.getY()));
21         ub.setX(std::max(ub.getX(), p.getX()));
22         ub.setY(std::max(ub.getY(), p.getY()));
23     }
24     return Rectangle(lb, ub);
25 }

```

User-defined reductions to compute the bounding box of a cloud of 2D points

For **Figure 4**, we rely on a simple class **Point2D** that stores the x and y coordinates of a 2D point, and a class **Rectangle** that stores a rectangle consisting of two 2D points. The **declare reduction** directives in the example declare new user-defined reduction operations. Separated by colons, the directive introduces names for the reduction operation (**minp** and **maxp**) and defines the data types for which the reduction operation is effective (**Point2D**). The last part contains a C/C++ expression or Fortran statement that describes how to combine two local results into a new intermediate result. This expression or statement is applied repeatedly until all intermediate results have been combined into the global result. The **initializer** clause specifies how the thread-private copy of the reduction variable is initialized.

The **for** loop iterates all elements of a **std::vector** that stores all the points of the cloud. For each point, it computes the running minimum and maximum of **x** and **y** coordinates to find the corners of the minimal bounding box. To parallelize the code, we introduce a **parallel for** construct to distribute the loop across a team of threads. Thus, each thread computes only a local bounding box that contains only the points assigned to the thread. The missing piece is how to combine all the local bounding boxes into the global one. The **reduction** clause at the **parallel for** construct performs this operation by applying the new reduction operations **minp** and **maxp**. The **omp_in** and **omp_out** variables in the combiner expression refer to the left and right operand when applying the reduction expression. In the example, these operands are passed to the minimum/maximum operation to enlarge the bounding box.



High Speed

Support to offload computation to attached devices, such as the Intel Xeon Phi coprocessor, is probably the most groundbreaking feature of OpenMP 4.0. OpenMP 4.0 introduces a new device model that extends the traditional threading model for shared memory to support offload computations. Having an OpenMP specification for offloading provides an industry-wide solution superior in every way to the OpenACC* it supersedes. Unlike OpenACC, OpenMP allows for the full use of a wide variety of devices instead of restricting what can be offloaded. Unlike OpenACC, there is no restriction on the number of devices supported, though all devices used need to be of the same architecture. OpenMP defines new constructs and directives to facilitate transfer of control between the host and the devices, as well as to issue data transfers.

The `target` construct (see **Figure 5**) transfers control from the host thread to the coprocessor device, and creates a device data environment to contain all the data needed to execute code on the target device. The `map` clause at the target construct controls the allocation of data in the data environment, as well as the direction of data transfers. **Figure 6** shows the different transfer types supported by `map`. In the example, we use `map(to:x[:N])` to transfer the array `x` from the host to the target device. The syntax `x[:N]` is shorthand for `x[0:N]` and describes `N` array elements starting from index `0`, that is, all array elements in `x`.

5

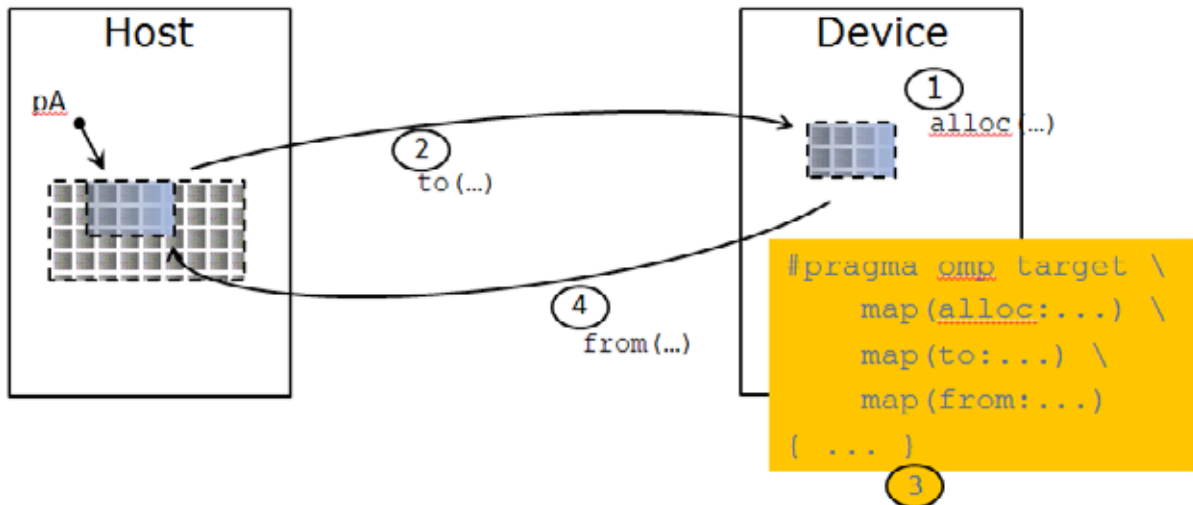
```

1  int n = 10240; float a = 2.0f; float b = 3.0f;
2  float *x = (float*) malloc(n * sizeof(float)); // init x
3  float *y = (float*) malloc(n * sizeof(float)); // init y
4
5  #pragma omp target data map(to:x[:N])
6  {
7      int num_blcks = 61;
8      int num_thrds = 4;
9      #pragma omp target map(tofrom:y[:N])
10     #pragma omp teams num_teams(num_blcks) num_threads(num_thrds)
11     #pragma omp distribute
12     for (int b = 0; b < n; b += num_blcks) {
13         #pragma omp parallel for
14         for (int i = b; i < b + num_blcks; ++i){
15             y[i] = a*x[i] + y[i];
16         }
17     }
18
19     // do something with y
20     // ...
21
22     #pragma omp target map(tofrom:y[:N])
23     #pragma omp teams distribute parallel for \
24         num_teams(num_blcks) num_threads(num_thrds)
25     for (int i = 0; i < n; ++i){
26         y[i] = b*x[i] + y[i];
27     }
28
29     free(x); free(y);

```

Offloading computation from the host to a coprocessor device





6 Transfer kinds of the map clause

Since coprocessors are using the PCI Express* bus to talk to the host system, avoiding unnecessary data transfers is very important. The **target data** construct establishes a device data environment, but does not transfer the control flow. With this construct, programmers can transfer data to the target device and keep it there, while the control flow is sent back and forth through several **target** constructs. If the **target** construct notices that data is already present on the device, it will then avoid the data transfer. The **target update** directive can be used to issue data exchanges, if data on either the host or the devices is outdated.

Let's dissect the example of **Figure 5** and explain how the different constructs interact. The example implements the well-known SAXPY operation $y = a \cdot x + y$. Since the code executes the operation two times, the code creates a device data environment for safe transfers of **x** across different invocations. The **target** construct in the example will notice that **x** is already available on the coprocessor, and only issue a data transfer for **y**. The same is true for the second **target** construct in the example.

The constructs mentioned so far only transfer control or data between the host and the devices. They do not automatically parallelize the code that is transferred over. Programmers need to use standard OpenMP features to create parallelism on the target devices. In our example, a simple **parallel for** construct would be sufficient to create a team of threads to execute the SAXPY loop in parallel. However, this may be inefficient, since the overhead of team creation and synchronization for the full coprocessor with 244 threads will be considerable.

To make the solution more efficient, the parallelization exploits the hierarchical architecture of four hyper-threads per physical coprocessor core. OpenMP 4.0 offers new constructs to map these hierarchies to program code:



- > The **team's** constructs creates a league of independent thread teams, whose master thread executes the code of the construct.
- > The **distribute** construct is a new worksharing construct to distribute a loop across the master threads of a league. The **distribute** construct does not have an implicit barrier at the end.
- > As usual, the existing **parallel for** construct distributes a loop within a thread team.

The SAXPY code creates one thread team for each of the 61 physical cores of the coprocessor. As a first level of parallelism, it then distributes the outer loop over **b** across the created thread teams. The second level of parallelism consists of a **parallel for** to create four threads for each hyper-thread of the physical cores. The second invocation of SAXPY shows the syntactic sugar of the combined **teams distribute parallel for** constructs to make the code shorter.

The transfer of control from the host to a device is a blocking operation. That is, the host thread waits until the control flow on the target has finished and all data has been transferred. If both host and device need to fulfill concurrent tasks, programmers may use existing OpenMP features. For instance, programmers can wrap a **target** or **target update** construct in an OpenMP task to execute these constructs in another OpenMP thread.

Conclusion

With version 4.0, OpenMP makes a quantum leap forward. The newly introduced features open up a new world of heterogeneous programming. OpenMP 4.0 provides new features that make it an interesting programming model with a wider reach than technical and scientific computing. Intel continues its commitment to OpenMP as a parallel programming model. Intel® Composer XE 2013 fully supports the previous OpenMP API specification 3.1 and future versions will supply a first-class, high-performance implementation of OpenMP 4.0. With SP1, Intel Composer XE 2013 for C, C++, and Fortran supports a subset of new features of OpenMP 4.0, including OpenMP target constructs, SIMD directives, and the OpenMP affinity feature. Future versions of Intel Composer XE will ship with support for user-defined reductions and task dependencies, as well as support for cancellation. The work for OpenMP version 4.1 and 5.0 has already begun by the OpenMP Architecture Review Board, and Intel is on board to support the development of the next, even better, OpenMP. ●

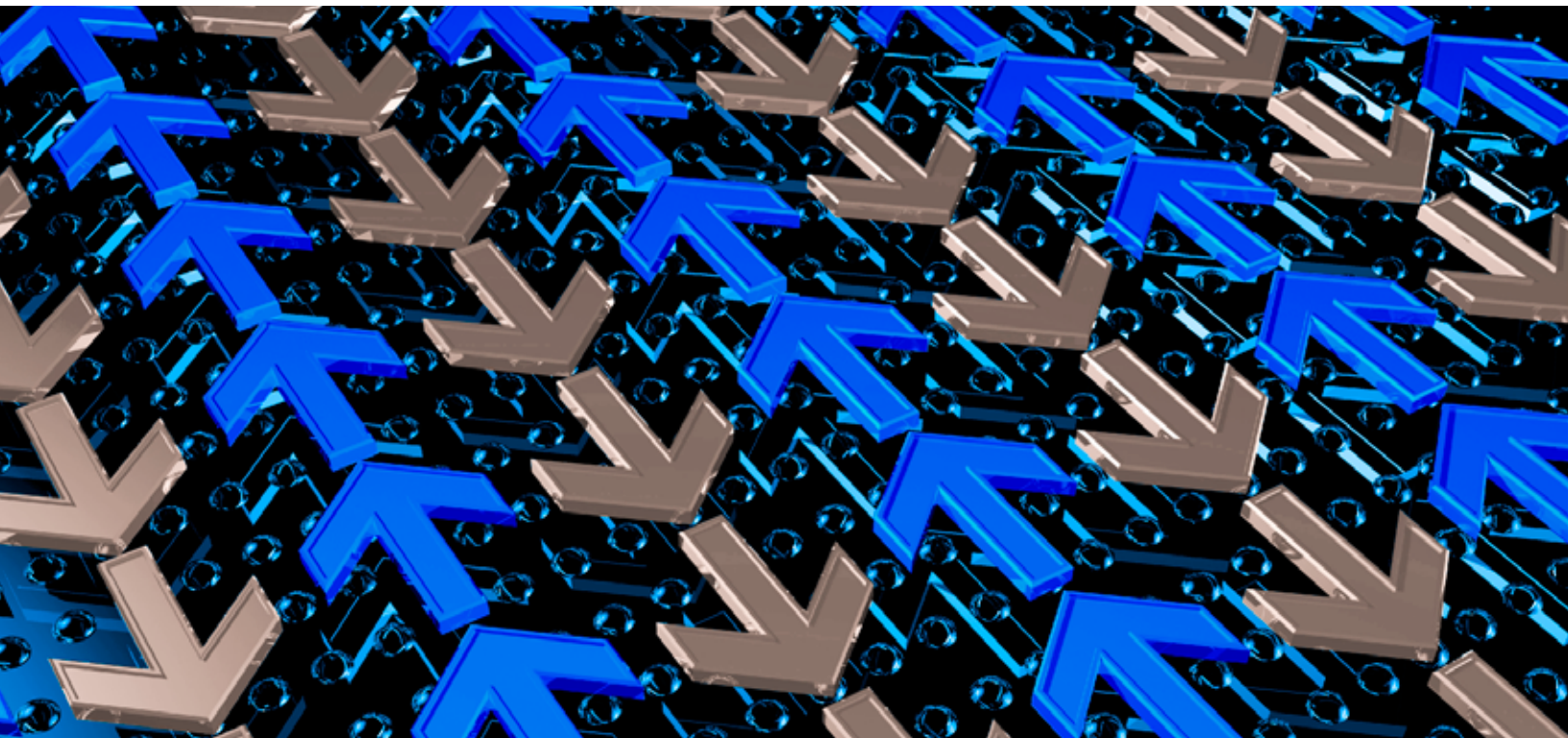


See the impact.
30-day trial software

[Evaluate the latest Intel® Parallel Studio XE and Intel® Cluster Studio XE >](#)

[Evaluate Intel® System Studio for embedded and mobile system developers >](#)





Profiling MPI Communications— Techniques for High Performance

by **James Tullos**, *Technical Consulting Engineer, Intel*

Introduction

The Message Passing Interface (MPI) provides an API for passing data between multiple instances (called ranks) of an application. MPI enables a high level of parallelism, but good performance is not automatic. This article presents techniques for improving MPI application performance, focusing on two aspects of optimization:

- › **Bottlenecks:** Portions of your program that delay execution
- › **Load Imbalance:** Uneven workload distribution, forcing some execution units to wait

Here, we'll focus on Intel® Trace Analyzer and Collector (ITAC), a performance analysis tool which is part of Intel® Cluster Studio XE. ITAC provides the ability to profile and analyze MPI applications to find areas for performance improvement.



Collecting Performance Data with ITAC

ITAC consists of two components: the Intel® Trace Collector (ITC) and the Intel® Trace Analyzer (ITA). ITC is a library that collects data from your application for use in ITA. The ITC output, called a trace, contains routine entry and exit data, a record of MPI parameters, and communication versus waiting time in MPI.

To use ITC with your MPI application, the simplest method is to load the library at runtime. When using the Intel® MPI Library, just add `-trace` to the `mpirun` arguments. As your application executes, ITC will generate the trace, which you can then analyze in ITA.

Bottlenecks in MPI

Communication bottlenecks cause your application to spend idle time waiting in communication routines, reducing overall performance. ITAC can show when your application is waiting for an MPI call to complete, helping to identify bottlenecks.

The example below uses a Poisson solver, which implements an `MPI_Sendrecv` to exchange data between neighboring ranks and an `MPI_Allreduce` to aggregate that data across ranks. These communications occur at every iteration. The example code is distributed with ITAC.

Before collecting the trace, you need to set some environment variables. Scripts are provided for this:

```
source /opt/intel/impi/4.1.1.036/intel64/bin/mpivars.sh
source /opt/intel/itac/8.1.3.037/intel64/bin/itacvars.sh
```

By default, ITC will distribute the collected data amongst multiple structured trace files (`stf`) for better scalability. If your run is small, you can tell ITC to combine data into a single trace file:

```
export VT_LOGFILE_FORMAT=singlestf
```

Now you are ready to collect the trace with:

```
mpirun -f myhostfile -n 16 -ppn 4 -trace ./poisson.sendrecv
```




This will run the solver across 4 nodes, with 4 ranks per node, and generate a trace file containing all of the MPI calls made by the application. You can use ITA to open the generated trace file and begin the analysis:

```
traceanalyzer ./poisson.sendrecv.single.stf
```



Finding Signs of a Problem

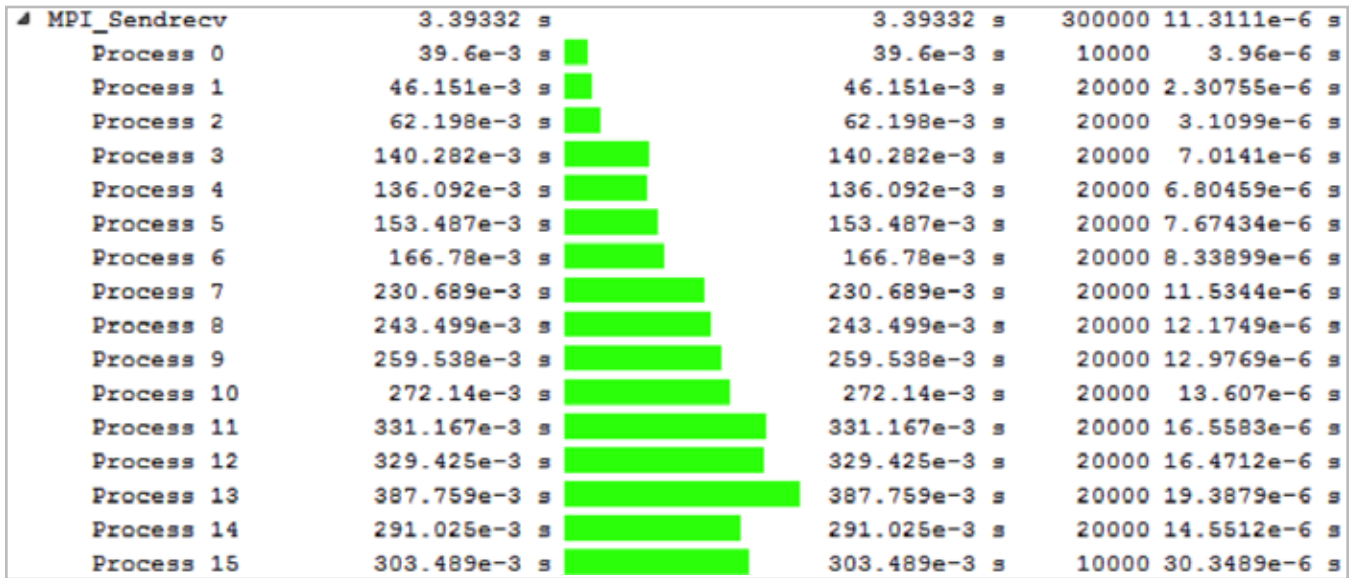
The initial view shows the Trace Map and the Function Profile Chart, set to the Flat Profile tab. The Flat Profile initially shows two groups: Group Application and Group MPI. Since we're focusing on optimizing MPI performance, we'll use Group MPI. Right-click on the Group MPI line and select Ungroup MPI. This will separate it into individual MPI calls. Each MPI function is given a default color, which has been changed here for better visibility.

Flat Profile	Load Balance	Call Tree	Call Graph			
Group All_Processes						
Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call	
Group All_Processes						
Group Application	10.6901 s		17.7119 s	16	668.134e-3 s	
MPI_Comm_size	56e-6 s		56e-6 s	16	3.5e-6 s	
MPI_Comm_rank	56e-6 s		56e-6 s	16	3.5e-6 s	
MPI_Errhandler_free	41e-6 s		41e-6 s	16	2.5625e-6 s	
MPI_Errhandler_get	109e-6 s		109e-6 s	16	6.8125e-6 s	
MPI_Errhandler_set	89e-6 s		89e-6 s	16	5.5625e-6 s	
MPI_Finalize	81.3279e-3 s		81.3279e-3 s	16	5.083e-3 s	
MPI_Bcast	5.316e-3 s		5.316e-3 s	16	332.25e-6 s	
MPI_Errhandler_create	616e-6 s		616e-6 s	16	38.5e-6 s	
MPI_Wtime	61e-6 s		61e-6 s	32	1.90625e-6 s	
MPI_Sendrecv	3.39332 s		3.39332 s	300000	11.3111e-6 s	
MPI_Allreduce	3.54073 s		3.54073 s	80000	44.2591e-6 s	

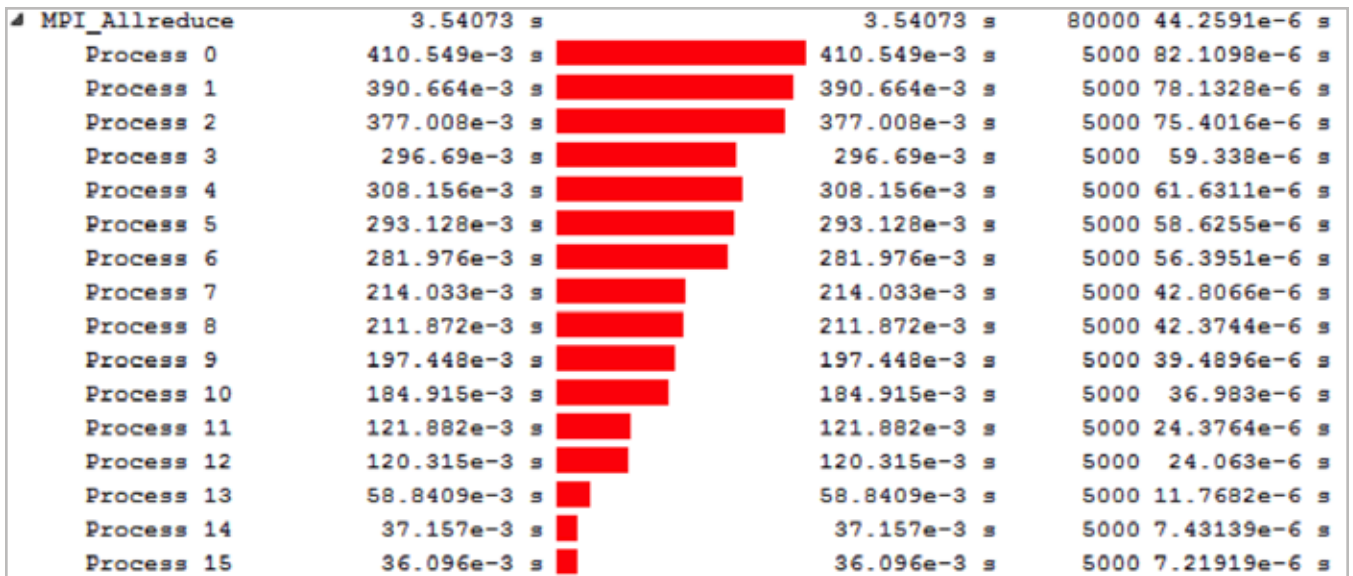
1 Flat Profile

In the Flat Profile, you can see that two MPI functions are the largest time consumers: MPI_Sendrecv and MPI_Allreduce. Each of these functions appears to take approximately the same amount of time. Click on the Load Balance tab to get more information. Notice in **Figure 2** that MPI_Sendrecv is consuming more time in the higher numbered ranks, and MPI_Allreduce is consuming more time in the lower numbered ranks (**Figure 3**). This indicates a potential performance issue, as communications should be balanced across all ranks.





2 MPI_Sendrecv Load Balance

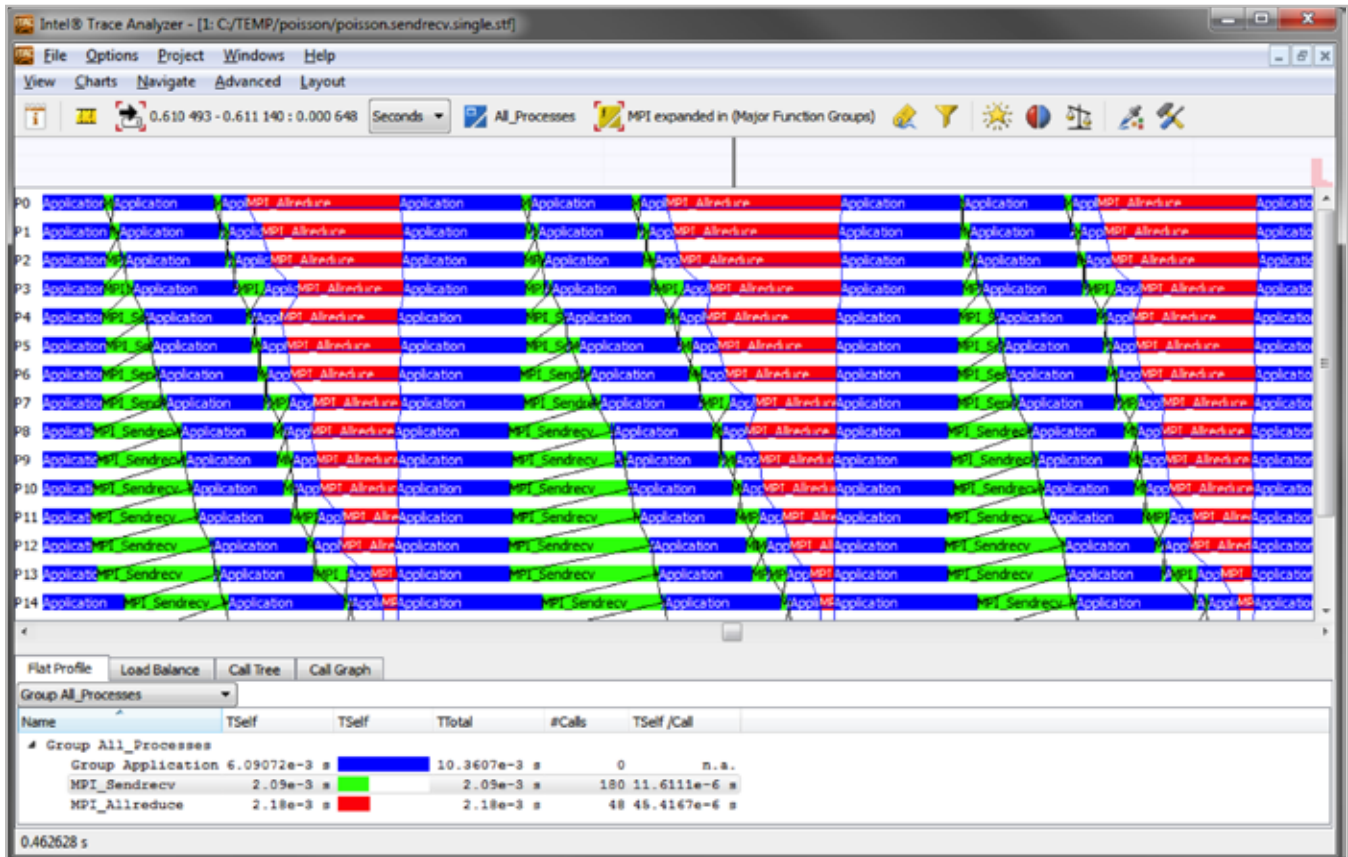


3 MPI_Allreduce load balance



Finding the Root Cause

Now that you've established the presence of a problem, it's time to determine the cause. Open the Event Timeline Chart by navigating to the **Charts->Event Timeline** menu. This shows a breakdown of what each rank is doing over the execution time of the application. As it is, the default view is too cluttered to be useful. Zoom (just click and drag) on the Event Timeline to focus on a few iterations. Notice that, when zooming, all charts except the Trace Map will alter the data shown to only display information for the selected time range. The Trace Map always shows the entire time, highlighting the selected range.



4 Event Timeline

Because MPI_Sendrecv is a blocking call, it cannot return until the exchange is completed. Ranks 0 and 1 must complete their exchange before ranks 1 and 2 can begin, forcing rank 2 to wait. This effect cascades through the ranks, creating the MPI_Sendrecv imbalance. This delay leads to earlier ranks reaching MPI_Allreduce sooner and being blocked until later ranks complete, creating the MPI_Allreduce imbalance.



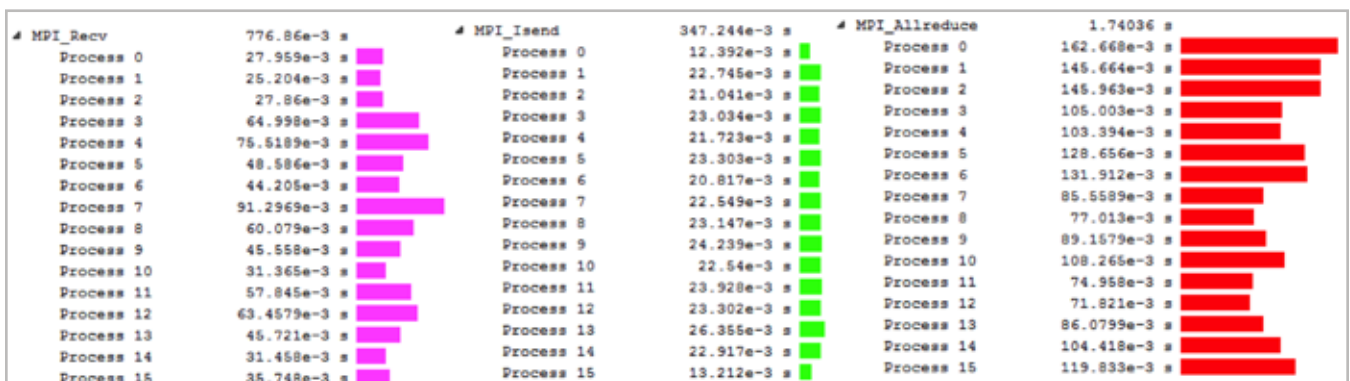
Removing the Bottleneck

This bottleneck appears to be a good target for improvement. Changing the blocking communication to non-blocking seems like an intuitive fix. You can replace the MPI_Sendrecv calls with calls to MPI_Isend and MPI_Recv, allowing each rank to send data to its neighbors without being blocked by earlier communications. Run the modified application and load the new trace in ITA. Open the Flat Function Profile. As you can see, the communications in the improved code consume less total time.

Flat Profile					
Load Balance					
Call Tree					
Call Graph					
Group All_Processes					
Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	11.0252 s		14.1075 s	16	689.073e-3 s
MPI_Comm_size	29e-6 s		29e-6 s	16	1.8125e-6 s
MPI_Comm_rank	71e-6 s		71e-6 s	16	4.4375e-6 s
MPI_Errhandler_free	37e-6 s		37e-6 s	16	2.3125e-6 s
MPI_Errhandler_get	62e-6 s		62e-6 s	16	3.875e-6 s
MPI_Errhandler_set	77e-6 s		77e-6 s	16	4.8125e-6 s
MPI_Finalize	92.357e-3 s		92.357e-3 s	16	5.77231e-3 s
MPI_Bcast	4.896e-3 s		4.896e-3 s	16	306e-6 s
MPI_Recv	776.86e-3 s		776.86e-3 s	300000	2.58953e-6 s
MPI_Errhandler_create	653e-6 s		653e-6 s	16	40.8125e-6 s
MPI_Isend	347.244e-3 s		347.244e-3 s	300000	1.15748e-6 s
MPI_Wtime	66e-6 s		66e-6 s	32	2.0625e-6 s
MPI_Waitall	119.583e-3 s		119.583e-3 s	160000	747.393e-9 s
MPI_Allreduce	1.74036 s		1.74036 s	80000	21.7546e-6 s

5 Flat Function Profile, non-blocking communications

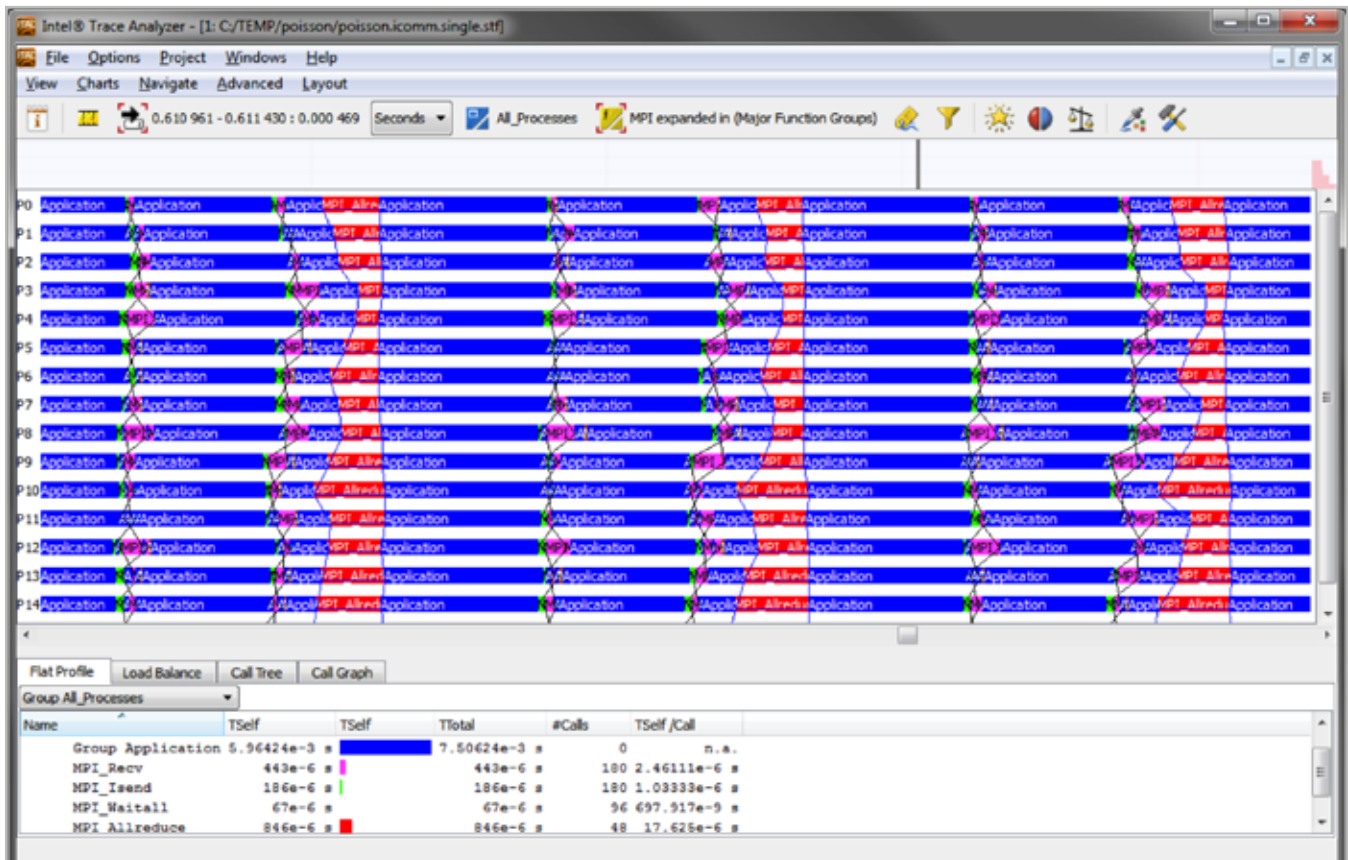
If we look at the Load Balance profiles of the most time-consuming communications, we can see that these functions are now more balanced.



6 Load Balance, non-blocking communications



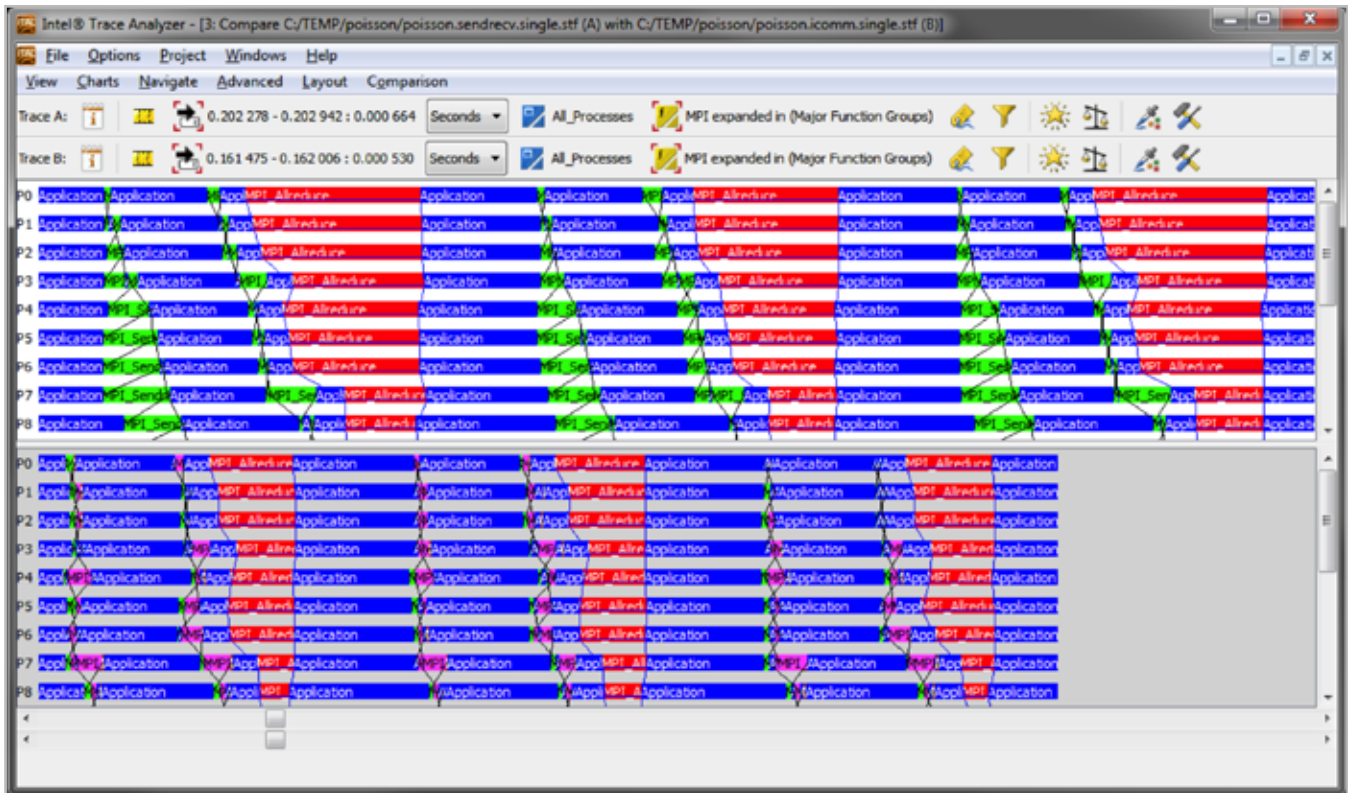
The Event Timeline gives the best indication of improvement. The serialized communications are gone, and communication is more parallel.



7 Event Timeline, non-blocking communications

Click **View->Compare** to compare the new trace to the original. You can synchronize navigation keys and/or mouse zoom, as well as match time scaling between the two traces. You can see the improvements here, showing that the solver is now spending more time doing actual computation (the blue regions), rather than waiting in communications.





8 Comparison of traces

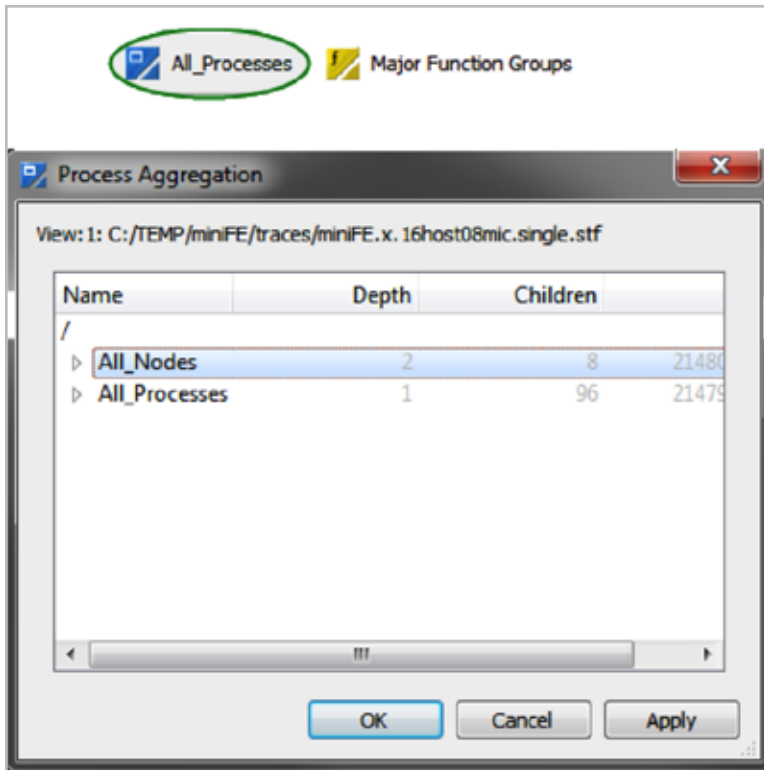
Finding Load Imbalance

Load balancing is the art of distributing work across your application. Next, we'll look at distributing load across heterogeneous MPI ranks.

The following example uses the miniFE* benchmark from Sandia Labs. It simulates the operations of a finite element solver, and is available at <http://www.nersc.gov/systems/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/minife/>. MiniFE uses strong scaling, dividing the load among ranks. This example incorporates Intel® Xeon Phi™ coprocessors to show load balancing across heterogeneous nodes. The job uses 4 host nodes, each with 16 MPI ranks, and 4 coprocessor nodes, with varying rank counts, and a matrix size of 256x512x512.

For this example, results are grouped by node rather than rank. To do this, click the Process Aggregation button in the toolbar, select All_Nodes in the Process Aggregation dialog, and click OK.



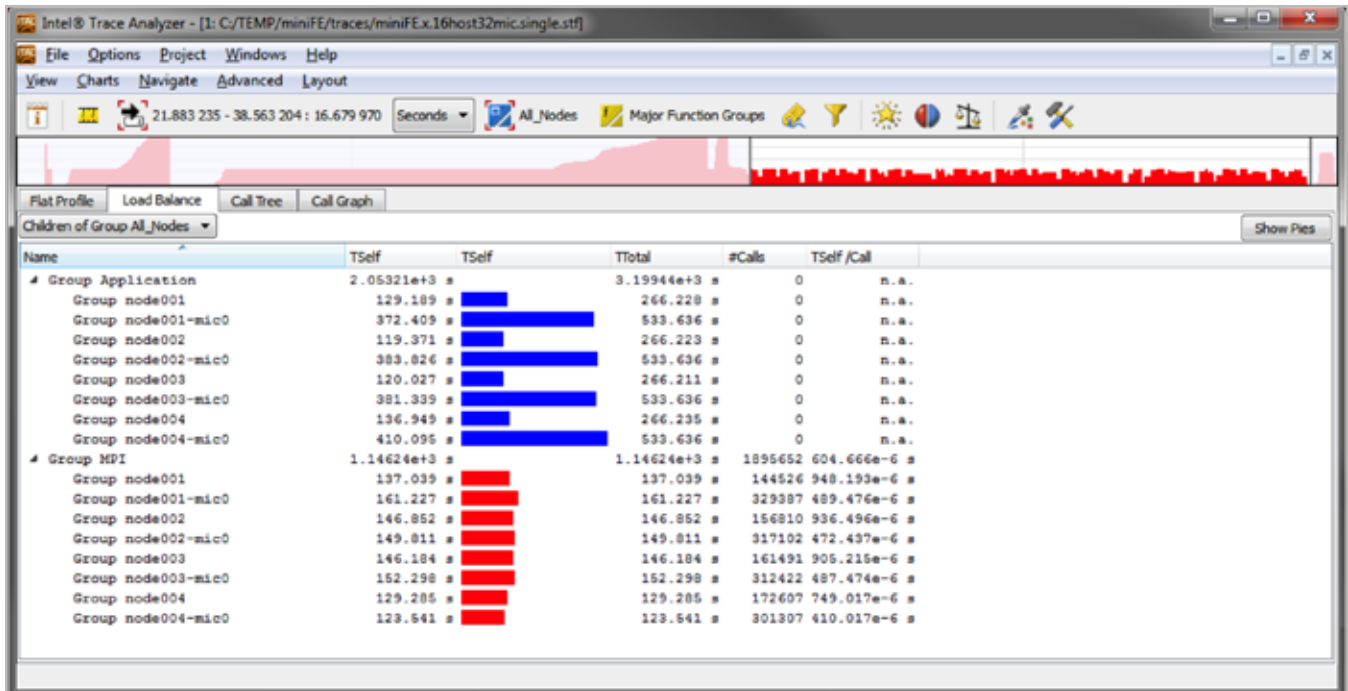


Imbalance in the Function Profile

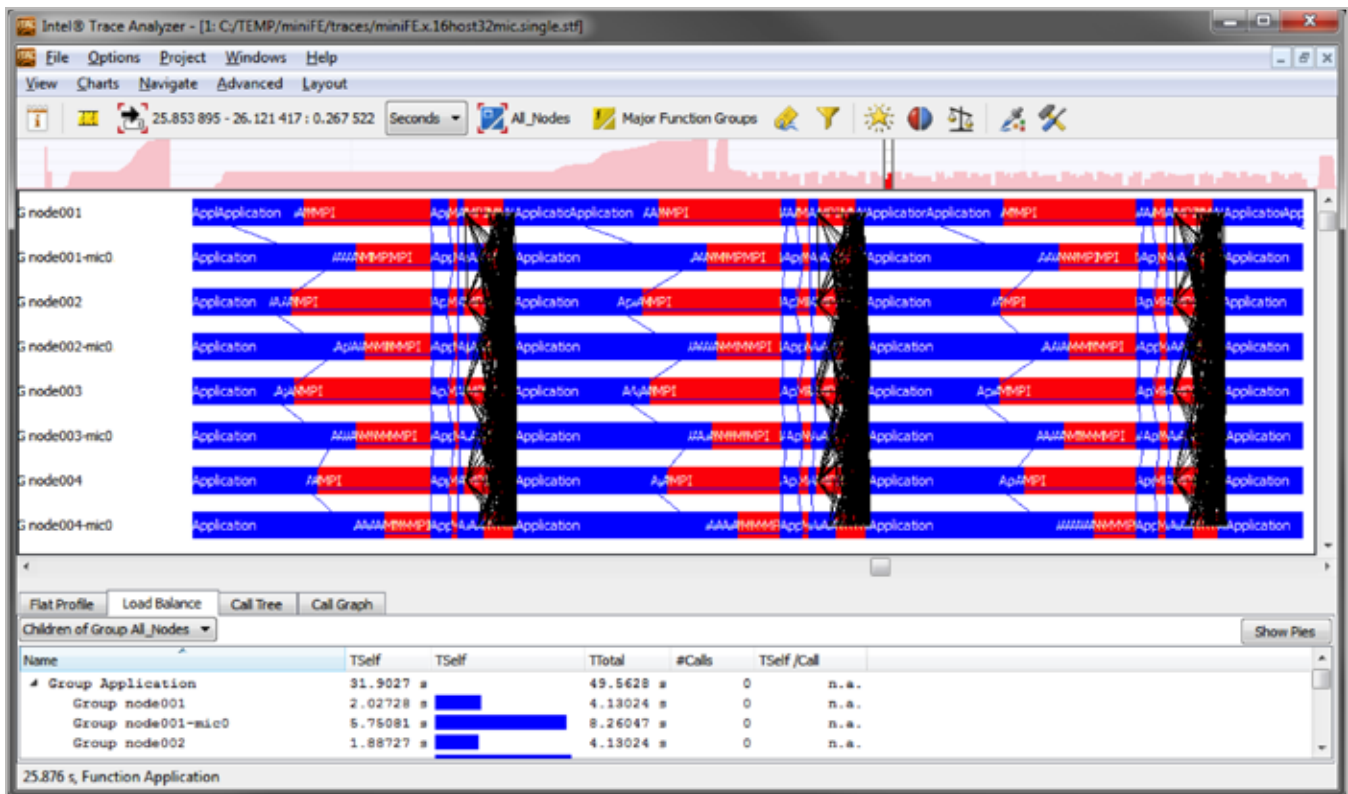
For the first run, miniFE has an initial distribution of 16 ranks per host and 32 ranks per coprocessor. Open the trace in ITA and select the Load Balance Profile. Notice that the MPI communications are fairly well balanced, but the workload is higher on the coprocessor ranks.

ITAC consists of two components: the Intel® Trace Collector (ITC) and the Intel® Trace Analyzer (ITA). ITC is a library that collects data from your application for use in ITA. The ITC output, called a trace, contains routine entry and exit data, a record of MPI parameters, and communication versus waiting time in MPI.





9 Load Balance Profile, 16 host, 32 coprocessor ranks per node

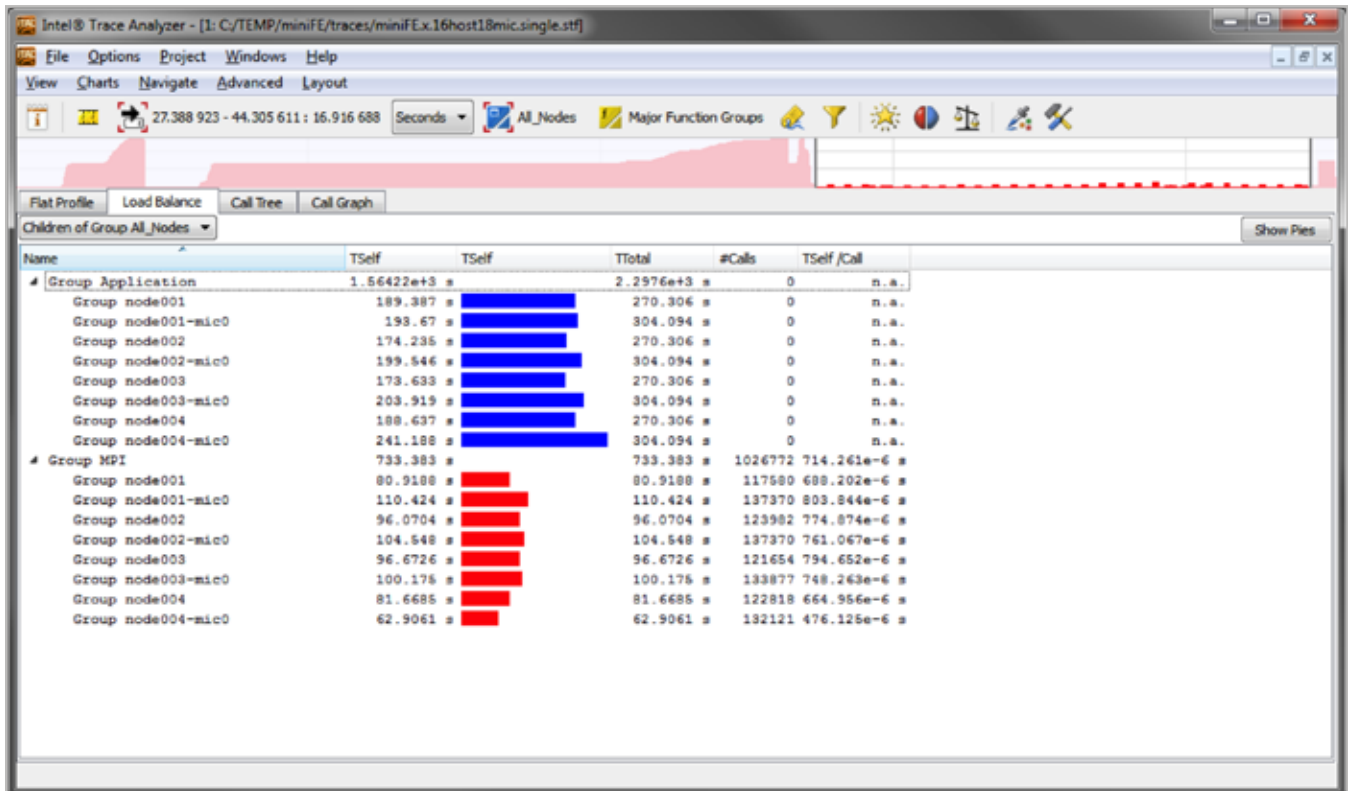


10 Load Balance Profile, 16 host, 32 coprocessor ranks per node



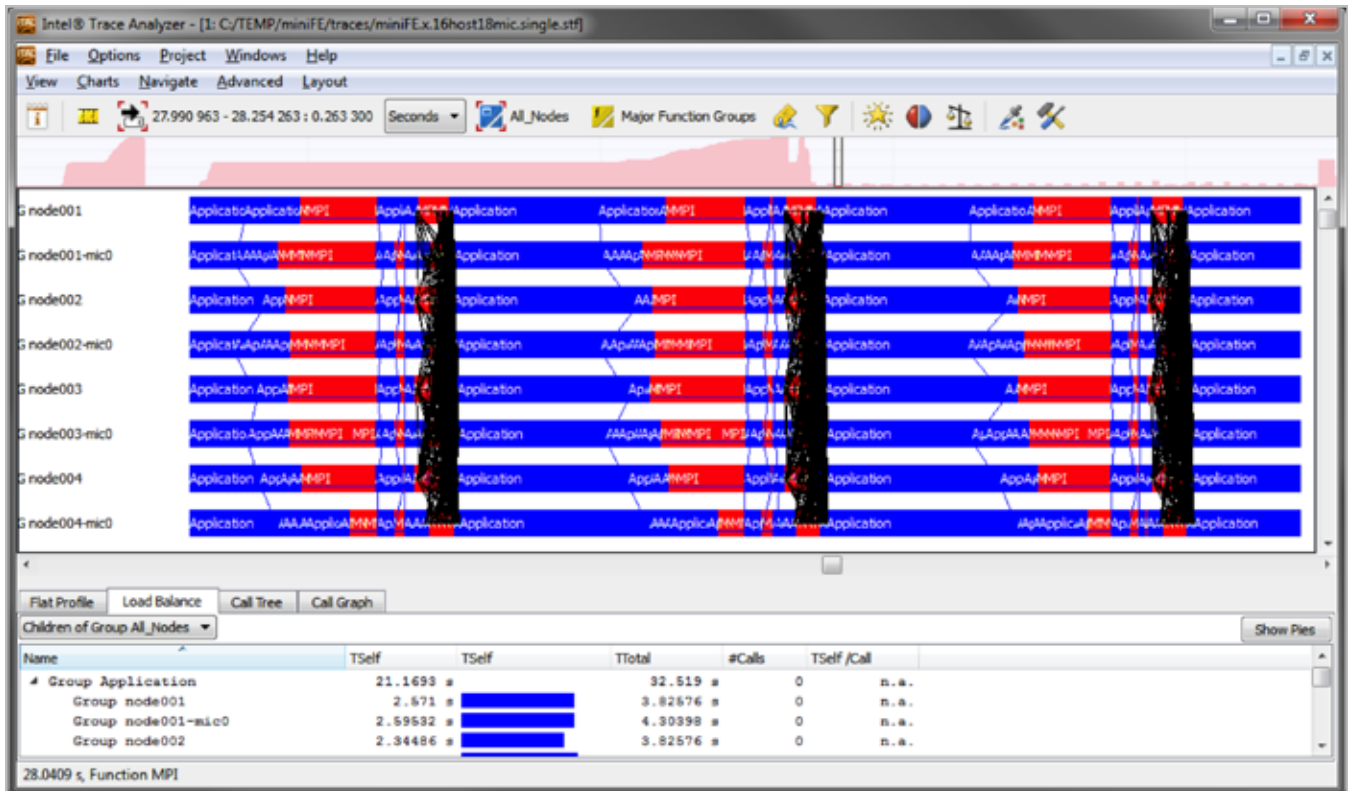
The Event Timeline also shows the imbalance. Each of the coprocessor nodes spends more time in application code than the corresponding host nodes.

Simply adjusting the number of ranks per coprocessor will not give perfect balance across all MPI ranks. Other factors, such as number of threads and total rank distribution influence the load balance and should also be adjusted. However, since this is a heterogeneous system, adjusting the number of coprocessor ranks can improve performance. A new distribution of 16 host ranks and 18 coprocessor ranks significantly improves the balance on this system, leading to improved application performance.



11 Load Balance Profile, 16 host, 18 coprocessor ranks per node





12 Event Timeline, 16 hosts, 18 coprocessors per node

Applying to Large Applications

We've seen how to use Intel Trace Analyzer and Collector to find two common problems in simple applications. In large applications, problems are often less obvious. However, the techniques provided here still apply.

Trace File Size

As your applications grow, it is easy to imagine the corresponding trace files increasing proportionally. To help alleviate this potential problem, ITC provides several techniques to manage the trace file size. You can trace only desired ranks, filter results for specific functions, or use the API to directly control trace collection. For more information about filtering in ITC, visit <http://software.intel.com/en-us/articles/intel-trace-collector-filtering>.

Summary

Now that you have the techniques to find bottlenecks and imbalance in MPI applications, you can use them to improve your own MPI applications. Visit <http://www.intel.com/go/itac> for more information about the Intel Trace Analyzer and Collector. ●



System Configuration

Both examples in this article were run on the same cluster.

Head node:

Intel® Server Board S5520UR
Intel® Xeon® Processor X5670
4GB Memory
500GB HDD

Compute nodes 0-3:

Intel® Workstation Board W2600CR2
Beta SNB-EP (Sandy Bridge) processors, 3.1 GHz
32GB Memory
1TB HDD
1 B1/C0 Intel® Xeon Phi™ Coprocessor 5120A each

Compute node 4:

Intel® Server Board S2600GZ
Intel® Xeon® Processor E5-2680
32GB Memory
128GB SSD
1 B1 Intel® Xeon Phi™ Coprocessor 5120A

CentOS* 6.3

Warewulf* 3.4 beta distribution system
Mellanox Technologies MT25418

Intel® Composer XE for Linux* 2013 Update 5
Intel® MPI Library for Linux* 4.1 Update 1
Intel® Trace Analyzer and Collector for Linux* Version 8.1 Update 1 (collection)
Intel® Trace Analyzer and Collector for Windows* Version 8.1 Update 3 (analysis)





Pexip Speeds Videoconferencing with Intel® Parallel Studio XE

by **Stephen Blair-Chappell**, *Technical Consulting Engineer, Intel*

Over the last 18 months, Pexip’s software engineers have been optimizing Pexip Infinity*—their videoconferencing software—using Intel® Parallel Studio XE. This article is based on interviews with Pexip’s lead optimization engineer, Lars Petter Endresen.

As organizations rely more heavily on global teams, videoconferencing allows teams to meet together without the corresponding expenses and lost productivity of traveling. It is important for a good user experience that any video content must be of reasonably high definition, and displayed without any significant delay. Both the quantity of data, and the real-time nature of videoconferencing, means that any host system must have copious amounts of processing power.

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

[Sign up for future issues](#)

[Share with a friend](#)



Traditionally, digital signalling processors (DSPs) were the obvious choice when designing video conferencing servers. The downside to such solutions is that DSP-based designs can come with an expensive price tag, with some servers costing as much as \$180k.

Here, we'll look at how Pexip, a videoconferencing start-up company founded 18 months ago by former Cisco and Tandberg executives, has replaced DSP designs with a software-only solution that can run on off-the-shelf Intel®-based servers. **By using Intel Parallel Studio XE, along with the processing power of the Intel® Core™ architecture, Pexip has been able to match, and even exceed, the performance of traditional conferencing systems.** Figure 1 shows a videoconferencing session being run from a laptop using Pexip's flagship product Pexip Infinity.



1 A Pexip Infinity* Videoconferencing session
Picture source: pexip.com

Analyze and Optimize

Pexip's development cycle consisted of a repeated sequence of analysis and incremental development:

- > Analyze the code for hotspots using Intel® VTune™ Amplifier XE
- > Examine the code in the hotspots and look for ways of improving the algorithms
- > Build the code with the Intel® compiler
- > Compare the optimized version with the original

All the development work was carried out on a PC with a 2nd generation Intel® Core™ microprocessor (code-named Sandy Bridge).

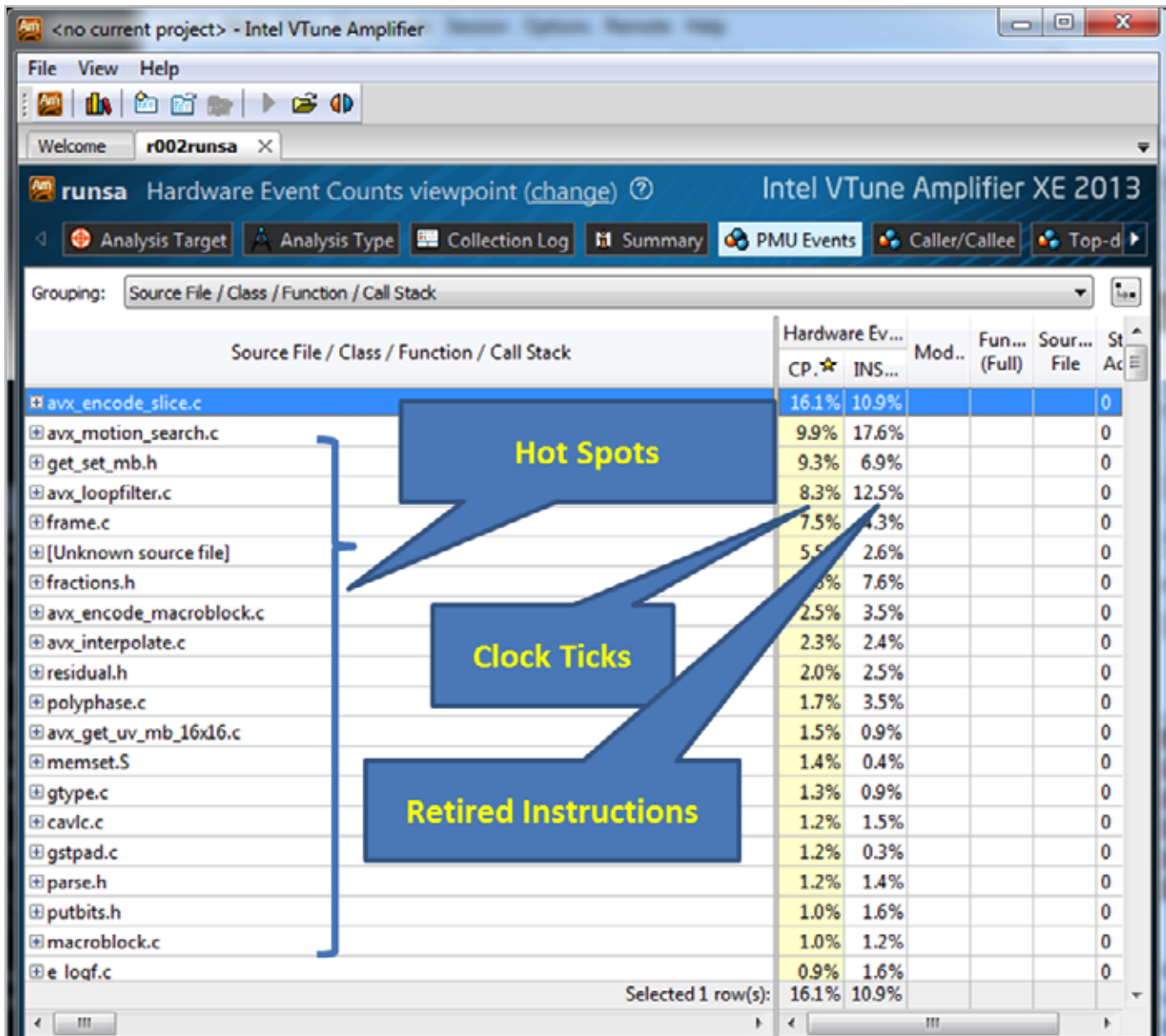
In addition to this work, Pexip also extracted code snippets from the most performance-sensitive parts of the code and analyzed them with the Intel® Architecture Code Analyzer (**Figure 2**) to see how the code would perform on 3rd and 4th generation Intel Core microprocessors.



Profiling with Intel VTune Amplifier XE

An advanced hotspot analysis was carried out using Intel VTune Amplifier. Each of the hotspots was then examined for optimization opportunities.

Figure 2 shows the results of the analysis of Pexip Infinity. The hotspots are displayed in the first column, with the most significant hotspots being at the top of the list. The second column shows the number of clock ticks there were for each hotspot. The third column shows how many retired instructions were used. Retired instructions are instructions that made it all the way through the CPU pipeline.



2 An Intel® VTune™ advanced hotspot analysis



The Main Hotspots

The analysis showed that there were five hotspots in the code (**Figure 3**).

Hotspot	Percentage of Time
EncodeSlice	16.1
MotionSearch	9.9
FastDCT	9.3
LoopFilter	8.3
Frame	7.5

3 The five top hotspots

All the hotspots are in code that implements the H.264 Advanced Video Coding (AVC) video compression standard.

To reduce the amount of data being streamed, the images are broken down into small macroblocks. Various compression and prediction algorithms are then applied to a sequence of macroblocks with only the differences between the macroblocks being placed in the compressed stream.

When the image stream is later decompressed, there is a tendency for the outline of the macroblocks to be visible on the reconstructed image. To overcome this flaw, the H.264 standard includes a loop filter that is used to remove artifacts in a process known as deblocking. This is implemented in the hotspot function **LoopFilter**.

Deblocking can be one of the costliest parts of image decoding, although in this particular test case the hotspot only consumed 8 percent of the runtime; it is not unusual for over 30 percent of the total CPU time be taken up with this task.

While we'll focus next on the **LoopFilter** bottleneck, our discussion is equally relevant for the other hotspots.

Optimizing for the Latest Generation Intel Core Architecture

Two optimization features of the Intel® compiler—*compiler vectorization* and *idiom recognition*—play an important role in boosting the performance of the deblocking filter.



Vectorization

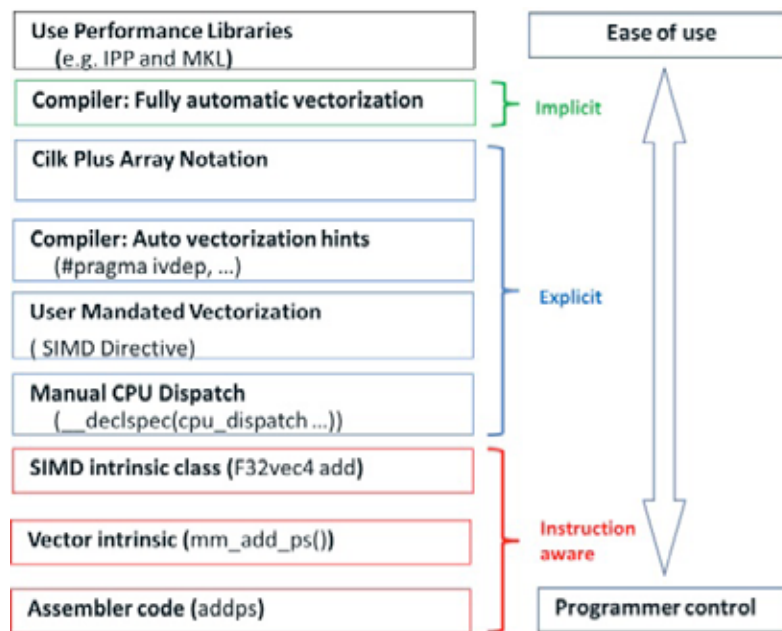
In 1997, Intel introduced a multimedia extension, MMX™ technology, providing a new class of instructions capable of doing multiple calculations in one instruction. These new SIMD (Single Instruction Multiple Data) instructions acted on extra-wide 64-bit registers holding several data items or vectors. Intel® MMX was followed by the Streaming SIMD Extensions (SSE), which was introduced on the Intel® Pentium® III processor in 1999. Since then, there have been 11 major revisions. Each revision has brought new capabilities and additional instructions. In three of these revisions, the width of the SIMD register has doubled, significantly increasing the amount of data parallelism that can be achieved.

Creating Vectorized Code

The Intel compiler provides a number of different ways that vectorization can be introduced into code (Figure 4). The simplest way to include vectorization is to use one of the optimization libraries, such as the Intel® Math Kernel Library (Intel® MKL) or the Intel® Integrated Performance Primitives Library (Intel® IPP).

The most difficult way to add vectorization is by hand-crafting your code to include assembler or vector intrinsics. This has the advantage that you can finely control the smallest detail of your algorithm, but it requires a deep working knowledge of the instruction set architecture. One of the problems in writing such low-level code is that for each new generation of the CPU architecture the code has to be rewritten.

Where possible, Pexip’s developers have avoided hard-coded implementations by writing their code in fairly generic C and C++, and then relying on the Intel compiler to automatically complete the work of vectorization.



4 Different ways of introducing vectorization



While code can be *implicitly* vectorized using automatic-vectorization, *explicit* vectorization is growing in popularity. Explicit vectorization allows programmers to place vectorization pragmas and constructs such as array notation in their code, without resorting to the use of instruction-aware programming.

Figure 5 shows an example using array notation. A detailed explanation of this and other examples can be found at: <https://www.cilkplus.org/tutorial-array-notation>.

Array Notation	Equivalent Scalar C and C++ Code
<pre>C[:, :] = 12;</pre> <p style="text-align: center;">(a) Set all the elements of the two-dimensional array C to 12.</p>	<pre>for (i = 0; i < 10; i++) for (j = 0; j < 10; j++) C[i][j] = 12;</pre>
<pre>func (A[:]);</pre> <p style="text-align: center;">(b) Pass all elements one by one into function "func()".</p>	<pre>for (i = 0; i < 10; i++) func (A[i]);</pre>
<pre>if (5 == a[:]) results[:] = "Matched"; else results[:] = "Not Matched";</pre> <p style="text-align: center;">(c) For each element of array A, print "Matched" if the value is 5. Otherwise print "Not Matched".</p>	<pre>for (int i = 0; i < array_size; i++) { if (5 == a[i]) results[i] = "Matched"; else results[i] = "Not Matched"; }</pre>

Examples using array notation

Example Vectorization: The Deblocking Filter

When the `LoopFilter` code deblocks an image, a weighting algorithm is used to decide whether to apply a *normal* or a *strong* filter to each edge of each macroblock. The file in **Figure 6**, `strong.c` contains an example of the strong filter.

Source: <https://www.cilkplus.org/tutorial-array-notation>



```

1: //strong.c
2: #include <stdint.h>
3: #define abs(x) ((x)<0)?-(x):(x)
4: #define max(x,y) ((x)>(y)?(x):(y))
5: #define min(x,y) ((x)<(y)?(x):(y))
6: #define norm(x,y) ((uint8_t)abs(x-y))
7: #define sat(x) max(min((int16_t)x,255),0)
8:
9: void strong(uint8_t *restrict m,
10: uint8_t Alpha,
11: uint8_t Beta,
12: uint8_t Gamma)
13: {
14:     for(int i=0;i<16;i++)
15:     {
16:         uint8_t L3=m[0*16+i];
17:         uint8_t L2=m[1*16+i];
18:         uint8_t L1=m[2*16+i];
19:         uint8_t L0=m[3*16+i];
20:         uint8_t R0=m[4*16+i];
21:         uint8_t R1=m[5*16+i];
22:         uint8_t R2=m[6*16+i];
23:         uint8_t R3=m[7*16+i];
24:         if ((uint8_t)((norm(R0,L0)<Alpha)
25:             &(norm(R0,R1)<Beta)
26:             &(norm(L0,L1)<Beta)))
27:         {
28:             m[3*16+i]=sat((2*L1+L0+R1+2)>>2);
29:             m[4*16+i]=sat((2*R1+R0+L1+2)>>2);
30:             if (norm(R0,L0)<Gamma)
31:             {
32:                 if(norm(L0,L2)<Beta)
33:                 {
34:                     m[1*16+i]=sat((2*(L3+L2)+L2+L1+L0+R0+4)>>3);
35:                     m[2*16+i]=sat((L2+L1+L0+R0+2)>>2);
36:                     m[3*16+i]=sat((R1+2*(L1+L0+R0)+L2+4)>>3);
37:                 }
38:                 if(norm(R0,R2)<Beta)
39:                 {
40:                     m[4*16+i]=sat((L1+2*(R1+L0+R0)+R2+4)>>3);
41:                     m[5*16+i]=sat((R2+R0+L0+R1+2)>>2);
42:                     m[6*16+i]=sat((2*(R3+R2)+R2+R1+L0+R0+4)>>3);
43:                 }
44:             }
45:         }
46:     }
47: }

```

Code for the strong filter



If you compile the code using the `/Qvec-report` flag, the Intel compiler will report that the main for loop at line 14 has been vectorized:

```
icl /QxAVX /O2 /Qstd=c99 /Qvec-report2 strong.c
Intel(R) C++ Intel(R) 64 Compiler XE for applications running on
Intel(R) 64 . . .

strong.c
c:\strong.c(14): (col. 3) remark: LOOP WAS VECTORIZED.
```

What is remarkable about this compilation, is the fact that the compiler has vectorized code containing three levels of if-tests, as well as dealing with mixed 8-bit unsigned and 16-bit signed data types.

If you build the code with the option `-S` to generate an assembler listing, you will see that the code is vectorized by the existence of packed instructions such as `vpaddw`, `vpsraw`, and `vmovups`:

```
icl /QxAVX /O2 /Qstd=c99 -S strong.c

vpaddw    xmm15, xmm15, xmm6                ;34.21
vpsraw    xmm12, xmm12, 3                    ;34.21
vpaddw    xmm15, xmm15, xmm2                ;34.21
vpsraw    xmm15, xmm15, 3                    ;34.21
vpackuswb xmm12, xmm12, xmm15               ;34.21
vblendvb  xmm4,  xmm4,  xmm12,  xmm14       ;34.11
vmovups   XMMWORD PTR [16+rcx], xmm4        ;34.11
```

Often, you can guess which instructions are packed by the existence of the letter “p” in the instruction name. Packed instructions are instructions that perform across the full set of vectors held in the registers. Full details of what each instruction does can be found in the *Intel® Architecture Instruction Set Extensions Programming Reference* book.

Idiom Recognition

The Intel compiler automatically recognizes various computational patterns or idioms, and generates performance-efficient code. Idiom recognition has been supported since the early days of the compiler, and is implemented in different compiler phases. With each new version of the compiler, new idioms continue to be added.

An example of one such idiom is the pattern to calculate the sum of absolute differences (SAD), which is implemented in the vectorization phase of the Intel compiler. **Figure 7** shows three pieces of source code. The first, 7(a), is a simple C code snippet, where the absolute difference of each element of **a** and **b** are added together and stored in the variable **sad**.



The second, 7(b), is the same SAD algorithm implemented using array notation. The reduction operator `__sec_reduce_add` is applied to each element of the arrays `a` and `b`, having first calculated the absolute difference.

Figure 7(c) shows the assembler generated by the Intel compiler from **Figure 7(a)**. Because the compiler is able to recognize the SAD idiom, it uses the instruction `psadbw` – which calculates the packed sum of absolute differences on unsigned byte integers. When the array notation code is compiled, the compiler generates assembler code almost identical to the 7(c).

You can generate this assembler for yourself using the `-S` option:

Windows: `icl -S /QxAVX sad.c`

Linux: `icc -S -xAVX sad.c`

```

// SAD calculation in C
#define abs(a) (((a)<0)?-(a):(a))
uint8_t a[16],b[16];
int sad=0;

for(int i=0;i<16;i++)
    sad+=abs(a[i]-b[i]);
    
```

7a

The Intel® compiler recognizes the SAD idiom

```

// SAD calculation using array notation
#define abs(a) (((a)<0)?-(a):(a))
uint8_t a[16],b[16];
int sad=0;

sad = __sec_reduce_add(abs(a[:]-b[:]));
    
```

7b

```

;; the compiler has recognised the
;; sad idiom and used the psadbw
;; instruction

movdqa    xmm0, XMMWORD PTR [ ]
pxor     xmm3, xmm3

psadbw   xmm0, XMMWORD PTR [ ]

padd     xmm3, xmm0
movdqa  xmm1, xmm3
psrldq  xmm1, 8
padd     xmm3, xmm1
movdqa  xmm2, xmm3
psrldq  xmm2, 4
padd     xmm3, xmm2
movd     eax, xmm3
    
```

7c



Intel Architecture Code Analyzer

The Intel Architecture Code Analyzer helps you conduct quick analyses for various ISA extensions before processors with these instructions are actually available. The Analyzer can be used to compare the relative performance of code snippets on different microarchitectures, but it does not provide absolute performance numbers.

The Analyzer performs a static analysis of code that lies between special markers inserted into the source by the programmer. **Figure 8** shows a sample output from the Analyzer. For more details, refer to whatif.intel.com.

```

iaca.exe -64 -arch IVB strong.avx.exe

Intel(R) Architecture Code Analyzer Version - 2.1
Analyzed File - strong.avx.exe
Binary Format - 64Bit
Architecture - IVB
Analysis type - throughput

Throughput Analysis Report
-----
Block Throughput: 83.50 Cycles      Throughput Bottleneck: Port1, Port5

Port Binding in Cycles per iteration:
-----
| Port | DV | 1 | 2 | 3 | 4 | 5 |
-----
| Cycles | 42.0 | 0.0 | 83.5 | 27.0 | 15.5 | 27.0 | 15.5 | 23.0 | 83.5 |
-----

.
.
.
| Num Of | Ports pressure in cycles
| Users | DV | 1 | 2 | 3 | 4 | 5 |
-----
| 1 | 1.0 | | | | | | | |
| 1 | 1.0 | | | | | | |
| 1 | | | 0.5 | 0.5 | 0.5 | 0.5 | | |
| 1 | | | 0.5 | 0.5 | 0.5 | 0.5 | | |
| 1 | | | 0.5 | 0.5 | 0.5 | 0.5 | | |
-----
| MOV_SAX, 0x36363636
| MOV_SDX, 0x1010101
| VMOVDQU xmm13, XMMWORD PTR [rsi+0x30]
| VMOVDQU xmm7, XMMWORD PTR [rsi+0x40]
| VMOVDQU xmm3, XMMWORD PTR [rsi+0x50]

```

8 A sample output from the Intel® Architecture Code Analyzer

Figure 9 shows the results of analyzing the strong filter code on 2nd, 3rd, and 4th generation Intel Core microarchitectures. The same cycle count is expected for Sandy Bridge and Ivy Bridge, as both share many of the same architectural features. The biggest difference is that Ivy Bridge uses a 22nm manufacturing process, whereas Sandy Bridge uses 32nm.



Microarchitecture Generation	Codename	Compiler Options	Analyzer Options	Block Throughput
2nd gen Intel® Core™	Sandy Bridge	/QxAVX	-64 -arch SNB	83.50
3rd gen Intel® Core™	Ivy Bridge	/QxAVX	-64 -arch IVB	83.50
4th gen Intel® Core™	Haswell	/QxCORE-AVX2	-64 -arch HSW	55.60

9 Cycle throughput of the strong filter on different generations of Intel® Core™ microarchitecture

By using the Intel Architecture Code Analyzer—and before having access to real hardware—Pexip was able to confirm that the code would benefit from the 4th generation Intel Core microarchitecture.

Results

As a result of using the vectorization and idiom recognition features of the Intel compiler, all the main hotspots in the code performed better as seen in **Figure 10**.

Hotspot	Number of Cycles x 10 ⁹		Speedup
	No Vectorization (/no-vec)	With Vectorization (/QxAVX)	
FastDct	69.66	3.64	19.14
MotionSearch	33.16	6.16	5.38
LoopFilter	14.62	3.54	4.13
EncodeMacroblock	12.62	1.64	7.70
EncodeSlice	12.30	6.78	1.81
Interpolate	6.30	1.28	4.92

10 The speedup of the different hotspots

Because Pexip chose to write their performance-sensitive codes in C, rather than assembler, they were able to migrate their application to the latest Intel Core architecture without having to recode everything.

At the start of the optimization process, the conferencing software was able to encode 1080p video with 7 frames per second (FPS). On completion of the work, it could comfortably handle 100 FPS: a speedup of more than 14x. ●

Learn More

Pexip is based in Oslo, Norway, with offices in the U.S. and United Kingdom. To learn more about Pexip, visit: www.pexip.com.

Download the Intel® Architecture Code Analyzer at: whatif.intel.com.



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

BLOG HIGHLIGHTS

Transactional Memory Support: The speculative_spin_mutex

BY CHRISTOPHER HUS >>

Intel recently released the 4th Generation Intel® Core™ processors, which have [Intel® Transactional Synchronization Extensions \(Intel® TSX\)](#) enabled. Intel TSX can improve the performance of applications that use lock-based synchronization to protect data structure updates. This feature allows multiple non-conflicting lock-protected changes to data to occur in parallel.

I've found over the years one of the best ways to improve parallel performance is to get rid of coarse-grained locks, replacing them with multiple, fine-grained locks to protect the structure being modified. While this technique can yield better performance, it is difficult to get right, and increases the number of locks that must be acquired to perform an operation. There may also be subtle performance problems if multiple

locks occupy the same cache line; though the changes themselves may not overlap, locking adjacent mutexes in the same cache line will result in false sharing.

Transactional memory addresses the problem a different way, by allowing multiple threads to access or update the protected data, and guaranteeing the updates appear atomically to all other threads. This gives some of the benefits of fine-grained locking without having to make changes to the code beyond replacing the locks.

The two interfaces for Intel TSX are Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM).

More





The Parallel
UNIVERSE