

Hybrid Programming Model

- Three levels of parallelism is required on Trinity test beds: (1) distributed memory parallelism supported through the Intel MPI library and (2) shared memory thread level parallelism on the MIC device using OpenMP, and (3) vectorization for the 512-bit SIMD Vector Processing Unit (VPU) of KNC.
- Distributed parallelism would require optimal domain decomposition considering load balancing and MPI communication cost.
- Thread level parallelism on 57 or 61 core KNC would require loop-level data parallelism via a threading library. In this study, Kokkos layer was used on top of OpenMP library to achieve performance portability.
- Intel VectorAnalyzer and compiler flags/reports can be used to vectorize the code to achieve fine-grained parallelism on VPU.

Performance Portability

- Performance portability and preserving the source code from potentially detrimental parallel directives for multiple architectures are important for software maintenance.
- Kokkos provides a minimal overhead abstract layer that isolates user code from device specific hardware architectures. Goal is to write one implementation which compiles and runs on multiple architectures.
- Kokkos supports MPI+“X” programming model to scale on both KNC MIC-based and GPU-based next generation platforms.
- Kokkos provides performant memory access patterns across multiple architectures and leverage architecture-specific features where possible.
- Kokkos currently uses device specific backend libraries such as CUDA, pthreads, and OpenMP for thread-level parallelism.

Kernel (Hot Spot)

In this case study, CAMAL’s Laplace volume smoothing algorithm was used as the hotspot kernel. Laplace smoothing is given by:

$$x_{i+1,k} = \frac{1}{N} \sum_{j=1}^N x_{i,j} \quad \text{----- (1)}$$

where N is the number of adjacent nodes to node k , $x_{i,j}$ is the coordinate of the j^{th} adjacent node of node k in the i^{th} iteration, and $x_{i+1,k}$ is the new $i+1^{\text{th}}$ iteration coordinate of node k .

Kokkos pseudo code is give below:

```
// data parallelism on N nodes
MyClass::class_method( function arguments )
{ ...
  // 1st argument: number of nodes
  // 2nd argument: this object
  Kokkos::parallel_for( N, *this);
  ..
}
// operator() for Kokkos::parallel_for
MyClass::operator()( int k ) const
{ ...
  // Laplace smoothing at node k given by Eq (1)
  laplacian_smooth_at_node(k);
}
```

Test Case

Kernel: Laplace volume smoothing
 Iterations: 10
 Data size: 5 million nodes
 No. of MPI processes = 4
 No. of threads per process = 1 to 64

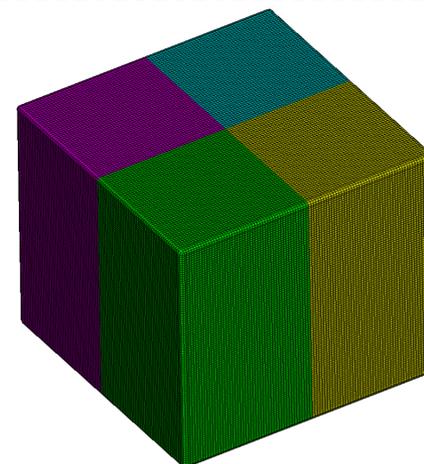


Fig 3: 5 million node mesh

Case Study Results

- One of the studies focused on node-level threading performance on a KNC MIC device. Therefore, MPI-related parameters were kept constant as shown in Table 1. The MPI-only version shown in row 1 is regarded as the baseline application.
- As we increase the threads per process, the deviation from the linear scaling increases due to thread startup and overhead costs as shown in Figure 3.
- On a MIC device, 95% reduction in runtime (a 20X speedup) is observed, as the single process runtime of 278.88 seconds is dropped to 14.24 seconds.

Table 1. 5-trial average result of volume smoothing on a 5 million node hex mesh

Number of Processes	Thread per Process	Process X Thread	Actual Runtime (sec)	Ideal Runtime (sec)	Percentage Deviation
4	1	4	278.88	278.88	0%
4	2	8	140.48	139.44	0.74%
4	4	16	73.22	69.72	5.02%
4	8	32	40.23	34.86	15.40%
4	16	64	24.16	17.43	38.61%
4	32	128	23.64	8.71	171.25%
4	64	256	14.24	4.35	226.79%

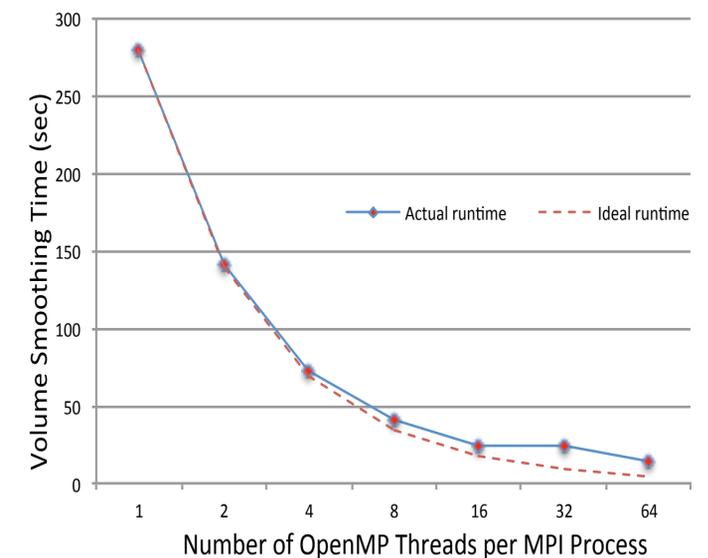


Fig 4: Linear scaling and actual scaling graphs