Next Generation Platforms (NGP)

- The next generation platforms built towards exascale computing will have heterogeneous architectures in order to take advantage of the revolution in many integrated core (MIC) and GPU devices.
- Trinity supercomputer to be hosted at Los Alamos National Laboratory in 2016 will have Cray XC30 platform architecture containing Intel Knights Landing (KNL) MIC processors.
- Sierra supercomputer to be hosted at Lawrence Livermore National Laboratory in 2017 will have IBM platform with GPU accelerators.

NGP Challenge

- 20 year “just recompile” free ride is over! “Just recompile” approach would result in approximately 10x slow down on NGP platforms.
- MPI-only is no longer possible because not all cores can run MPI.
- Compute nodes are heterogeneous in both cores and memory. Scaling requires leveraging node-level heterogeneous parallelism.
- Compute node architectures can be characterized by increasing thread count and decreasing memory per thread. Therefore, threading is critical for high scaling.
- Performance portability on multiple advanced architectures is a challenge.
- High performance computing on these advanced architectures requires hybrid programming models containing inter-node and on-node parallelism.

Procedure to Scale Legacy Code

- **STEP 1 - Profile legacy code to identify characteristics**
  Profile to identify number of hotspots, distribution of hotspots, etc. to decide rewrite or refactor. TAU has been used to profile CUBIT, and Laplace volume smoothing in CAMAL was taking 30% of the overall Sculpt meshing runtime on our test cases.
- **STEP 2: Choose a suitable programming model**
  The next generation platforms would require both distributed and shared memory programming models. Also, determine if data parallelism is enough or if task parallelism is required? In this case study, hybrid MPI + Kokkos programming model has been selected.
- **STEP 3: Implement selected programming model**
  First, convert the serial code to parallel using the MPI for distributed level parallelism. Next, make the code thread safe at hotspots for shared memory parallelism. In this study, a performance portable layer such as Kokkos as been used for thread level parallelism.
- **STEP 4: Determine optimal runtime parameters**
  Determine the optimal number of MPI processes and threads per MPI process based on the underlying hardware architecture. In this study, a series of studies were performed to find optimal parameters.
- **STEP 5: Optimize the code for higher scaling**
  Optimize the code for efficient memory access, reduced communication, vectorization, etc. to achieve higher scaling.

Test Beds

- In this case study a test bed for Trinity supercomputer was used. The test bed contains Intel Knight Corners (KNC) instead of Intel KNL.
- Two nodes were allocated in this study. Each node contains a socket with two KNC coprocessors.
- Each KNC has 57 or 61 many integrated cores.

**Fig 1: NGP compute node with heterogeneous cores and memory [courtesy of https://github.com/kokkos]**

**Fig 2: Test bed contains KNC**
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 performs 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:

\[ x_{i+1,k} = \frac{1}{N} \sum_{j=1}^{N} x_{ij} \]  \hspace{1cm} (1)

where \( N \) is the number of adjacent nodes to node \( k \), \( x_{ij} \) 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 \( i+1 \)th iteration coordinate of node \( k \).

Kokkos pseudo code is given below:

```cpp
// 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

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

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Thread per Process</th>
<th>Process X Thread</th>
<th>Actual Runtime (sec)</th>
<th>Ideal Runtime (sec)</th>
<th>Percentage Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>278.88</td>
<td>278.88</td>
<td>0%</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td>140.48</td>
<td>139.44</td>
<td>0.74%</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>16</td>
<td>75.22</td>
<td>69.72</td>
<td>5.02%</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>32</td>
<td>40.25</td>
<td>34.86</td>
<td>15.40%</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64</td>
<td>24.16</td>
<td>17.43</td>
<td>35.81%</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>128</td>
<td>23.64</td>
<td>8.71</td>
<td>171.25%</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>256</td>
<td>14.24</td>
<td>4.35</td>
<td>226.79%</td>
</tr>
</tbody>
</table>

Fig 3: 5 million node mesh

Fig 4: Linear scaling and actual scaling graphs