OpenFOAM: Computational Fluid Dynamics

1. Flow analysis of Motor Bike and Car.
2. Gauss Siedel Solver used for solving $AXx = b$.
3. Transform $(L + D + U) * x = b$ $=> (L + D) * x = b - U * x$.

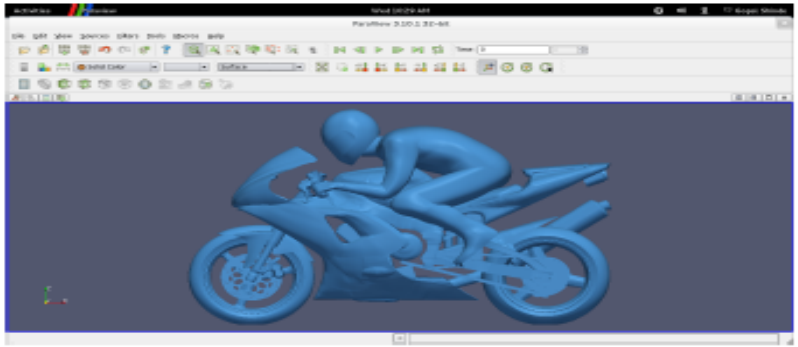


Figure: Motor Bike CFD*

*Source: Openfoam tutorial

Gauss Siedel iteration : $(L + D) * x^{new} = b - U * x^{old}$

What's unique about my tuning work

- **The OpenFOAM®(Open Field Operation and Manipulation) CFD Toolbox is a free, open source CFD software package which has a large user base across most areas of engineering and science, from both commercial and academic organizations.**

- **Application domain:** Computational Fluid Dynamics

- **Execution mode:** Symmetric MPI, Cluster

- **The following tools were used in the analysis and optimization cycles:**

  - Intel vTune Analyzer, Intel Trace Analyzer and Collector, idb debugger, Intel compiler parallelization reports

Insights

- **What I learned:**

  - Effective load distribution Xeon & Xeon Phi in symmetric mode give better performance. This is due to difference in CPU frequency between Xeon:Xeon Phi and total number of cores present in both of them
  - Reduction in neighboring communication in unstructured mesh between Xeon & Xeon Phi gives better performance
  - IO penalty is high on Xeon Phi as compare to Xeon
  - System tuning like Huge Page = 2MB, compiler flags gives signification improvements.

- **What I recommend:** Effective use of tools to diagnose performance problems.

- **Most effective optimization: Vectorization Intrinsics**

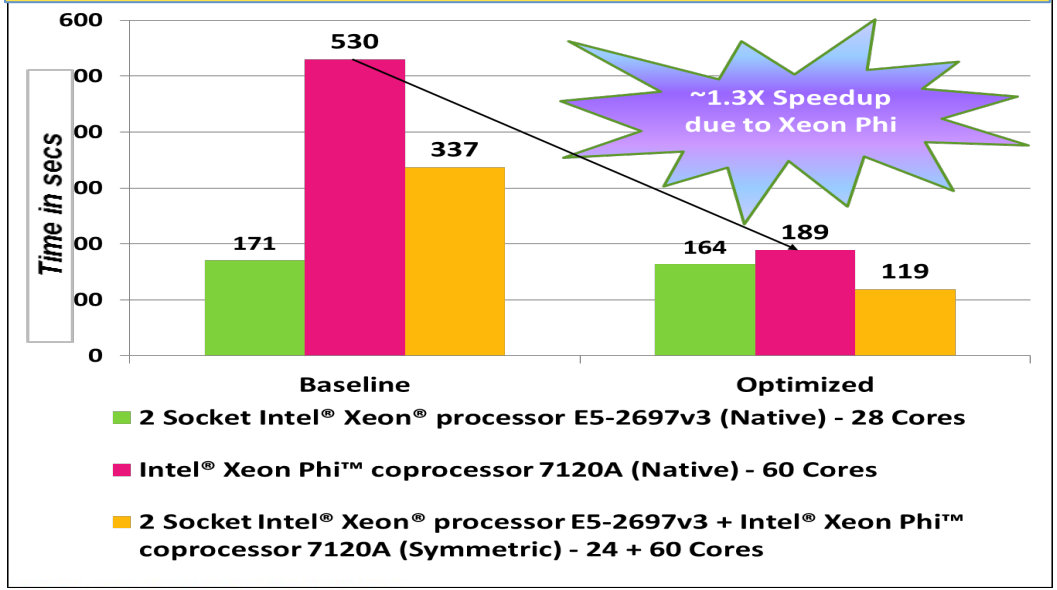
  - Changes to get efficient vectorization and avoid compiler overhead due to VGATHER & VSCATTER instructions
  - Unrolling
  - Prefetching
  - Explored decomposition algorithm change for weighted decomposition support

- **Surprises:** Oversubscription works better in native Xeon Phi mode when IO is turned off.

Performance – Motorbike Case

- **Compelling Performance :**
  - **Haswell optimized : 164 s**
  - **Haswell + Xeon Phi optimized : 119s (164/119 = 1.38 X)**

Runtime(secs) of Motorbike Case, 4.2M Workload (lower is better)



- **Competitive performance:**
  - **There are no published numbers for GPU's for OpenFOAM benchmark application.**

- **Results:**
  - **1.38 X Speedup due to Xeon Phi addition w.r.t Xeon optimized result**

Vectorization Effectiveness for GaussSeidel

	Total % vectorization gain	Total vectorization intensity	Number of vectorized loops	Total Time (s)
Original workload baseline	1 X	4.1	1	707
Original workload baseline with disabled auto-vectorization	1.62 X	NA	0	436
Tuned workload (vectorized using intrinsics)	2.25 X	4.2	2	314

Issues with GaussSeidelSmoother Loop

```
for (register label celli=0; celli<nCells; celli++)
{
    // Start and end of this row
    fStart = fEnd;
    fEnd = ownStartPtr[celli + 1];

    // Get the accumulated neighbour side
    psii = bPrimePtr[celli];

    // Accumulate the owner product side
    for (register label facei=fStart; facei<fEnd; facei++)
    {
        psii -= upperPtr[facei]*psiPtr[uPtr[facei]];
    }

    // Finish psi for this cell
    psii /= diagPtr[celli];

    // Distribute the neighbour side using psi for this cell
    for (register label facei=fStart; facei<fEnd; facei++)
    {
        bPrimePtr[uPtr[facei]] -= lowerPtr[facei]*psii;
    }

    psiPtr[celli] = psii;
}
```

Outer loop is not parallelizable because of dependencies

Low trip count, on average ~4, not good for vectorization

Indirect Referencing :
- Inefficient use of cacheline
- Scatter/Gather overhead

Overhead of scalar "DIV" operation inside loop

Unaligned data structures throughout the application

Identify vector assembly overhead

1st Loop

```
for (facei=fStart; facei<fEnd; facei++)
      psii -= upperPtr[facei]*psiPtr[uPtr[facei]];
```

By default : vectorized and unrolled by 4

1.

```
vgatherdpdq  (%r14,%zmm14,8), %k4, %zmm15
vgatherdpdq  (%r13,%zmm11,8), %k1, %zmm16
vgatherdpsl  (%r13,%zmm10,4), %k3, %zmm14
vgatherdpsl  (%r12,%zmm0,4), %k2, %zmm11
```

... (Multiple gathers, due to indirect reference, unrolling and peel & remainder loops)

2. The FOR loop overhead is high as it has only a small trip count in most of the cases.

3. Compiler performing reduction operation seems costlier as visible in assembly.

4. Peel and remainder loops are introducing overheads since trip count is < 8 for 99% of the time.

These issues suggest that manual vectorization using intrinsics might be beneficial.

Vtune profiling & ICC compiler vec-report leads to bottleneck identification

Unrolling For loop with intrinsics and multi-versioning the code

1st Loop

```
for (facei=fStart; facei<fEnd; facei++)
        psii -= upperPtr[facei]*psiPtr[uPtr[facei]];
```

Baseline assembly : for loop overhead

```
for()
{
    ......
}
```

```
add $0x8, %r15
lea 0x8(%r10), %r11d
jnl 0x3e5451 <Block 145>
cmpl  0x4(%rax,%r8,4), %edi
.....
```

Time = ~110 s

Intrinsic assembly

```
if (faces < 8)
    intrinsics
else
    for()
```

Hit 99%

Hit 1%

```
sub %r12d, %eax
lea (%rax,%rdi,1), %ecx
cmp $0x8, %eax
jnle 0x3f0dce <Block 109>
.....
```

Time = ~20 s

For loop was unrolled and replaced by if-else condition check. 99% time IF condition is hit.

Baseline code
Intrinsic code

Time comparison of 1st and 2nd loop : Baseline vs Intrinsic

First Loop	Time (s)	Remarks
Baseline	486	By default vectorized
Intrinsic (vectorization)	186	

Speedup 2.6 X

Second Loop	Time (s)	Remarks
Baseline	221	By default non-vectorized
Baseline + #pragma ivdep	412	Enabling vectorization degrades performance
Intrinsic (vectorization)	128	

Speedup 1.7 X

Vtune profiling & ICC compiler vec-report/opt-report used for detailed analysis

Note: Time taken from Vtune by aggregating assembly

Vtune Profiles of Baseline & Optimized version

Baseline:

Function / Call Stack	CPU Time	Clockticks	Instructions Retired	CPI Rate	Cache Usage		
					L1 Misses	L1 Hi...	Estimate...
func@0xe72d6	18.7%	1,599,780,000,000	801,430,000,000	1.996	42,000,000	1.000	16,728.0 ...
Foam::GaussSeidelSmoother::smooth	15.3%	1,309,000,000,000	587,930,000,000	2.226	5,634,300,000	0.963	144.771
[vmlinux]	9.5%	808,150,000,000	142,030,000,000	5.690	3,780,000,000	0.928	151.576
Foam::lduMatrix::residual	4.1%	354,410,000,000	128,240,000,000	2.764	2,068,150,000	0.961	117.601
Foam::lduMatrix::Amul	3.0%	253,680,000,000	100,380,000,000	2.527	1,561,000,000	0.959	118.702
Foam::fv::gaussGrad<double>::gradf	2.8%	243,460,000,000	134,890,000,000	1.805	1,065,400,000	0.978	102.178
Foam::surfaceInterpolationScheme<double>::	2.2%	186,060,000,000	67,970,000,000	2.737	1,715,000,000	0.958	59.751
Foam::fvc::surfaceIntegrate<double>	1.7%	149,170,000,000	68,950,000,000	2.163	1,204,000,000	0.967	57.140
Foam::fvc::surfaceIntegrate<Foam::Vector<do	1.6%	139,720,000,000	40,040,000,000	3.490	847,000,000	0.964	92.190

19.4 %

Optimized:

Function / Call Stack	CPU Time	Clockticks	Instructions Retired	CPI Rate	Cache Usage		
					L1 Misses	L1 Hi...	Estimate...
[vmlinux]	8.4%	664,660,996,990	63,700,000,000	10.434	690,000,000	0.980	732.458
Foam::GaussSeidelSmoother::smooth	7.2%	570,550,855,825	244,050,000,000	2.338	2,542,500,000	0.962	159.935
Foam::GaussSeidelSmoother::smooth	3.4%	272,840,409,260	131,450,000,000	2.076	1,454,000,000	0.962	135.146
[libc-2.14.90.so]	3.2%	252,590,378,885	107,750,000,000	2.344	635,750,000	0.985	197.867
func@0xe7334	2.7%	214,210,321,315	144,950,000,000	1.478	0	1.000	0.000
Foam::fv::gaussGrad<double>::gradf	2.1%	164,070,246,105	66,800,000,000	2.456	1,100,000,000	0.958	85.950
Foam::linearUpwindV<Foam::Vector<double>	2.0%	160,180,240,270	39,200,000,000	4.086	810,000,000	0.959	138.420
Foam::surfaceInterpolationScheme<Foam::Ter	1.7%	135,670,203,505	25,500,000,000	5.320	1,245,000,000	0.928	77.382
func@0x235d20	1.6%	127,760,191,640	77,350,000,000	1.652	193,750,000	0.993	270.042
Foam::surfaceInterpolationScheme<Foam::Ve	1.6%	127,210,190,815	32,150,000,000	3.957	1,546,000,000	0.923	60.242
Foam::surfaceInterpolationScheme<double>:	1.6%	126,640,189,960	36,700,000,000	3.451	1,405,250,000	0.924	59.815
Foam::multiply	1.5%	115,160,172,740	13,900,000,000	8.285	1,275,250,000	0.733	78.451
Foam::fvc::surfaceIntegrate<double>	1.4%	113,300,169,950	50,200,000,000	2.257	569,500,000	0.975	175.126
Foam::lduMatrix::Amul	1.4%	110,040,165,060	72,000,000,000	1.528	716,250,000	0.977	90.403
Foam::fvc::surfaceIntegrate<Foam::Vector<d	1.3%	104,010,156,015	33,500,000,000	3.105	532,000,000	0.972	121.778
Foam::lduMatrix::negSumDiag	1.2%	91,630,137,445	25,750,000,000	3.558	670,250,000	0.957	85.642
Foam::lduMatrix::residual	1.1%	85,930,128,895	43,850,000,000	1.960	232,250,000	0.991	129.745

11.7 %

DRAM Latency Sensitivity of OpenFOAM (on KNC)

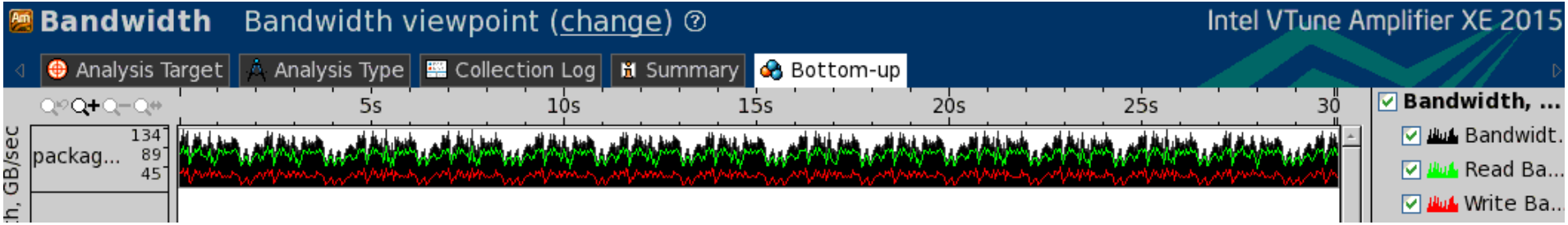
The Estimated Latency Impact is **215 cycles** which is >145, indicating that the application is **memory bound**.

Cache Usage:

L1 Misses:	77,904,750,000
L1 Hit Ratio:	0.955
Estimated Latency Impact:	215.626

Estimated Latency Impact value is high, which likely indicates that the majority of L1 data cache misses are not being serviced by the L2 cache. Software prefetching is one strategy for improving this on the Intel Xeon Phi coprocessor. Data reorganization or traditional techniques to increase data locality (such as cache blocking/tiling, using streaming stores, and...

The peak bandwidth is **134 GB/s** which is close to the threshold on KNC (140 GB/s).



DRAM Latency Sensitivity metrics of OpenFOAM

Xeon Phi :

Application	CPI Rate	L1 Hit Ratio	Estimated Latency Impact	Peak Bandwidth (GB/s)	Memory BW bound? (Y/N)	Potential Memory Latency Sensitive? (Y/N)
OpenFOAM	4.319	0.955	215.6	134	Y	Y

Xeon (HSW-EP) :

Application	CPI Rate	Backend Bound	Memory Bound	Core Bound	Peak Bandwidth (GB/s)	Memory BW bound? (Y/N)	Potential Memory Latency Sensitive? (Y/N)
OpenFOAM	1.629	0.803	1.000	0.057	52	Y	N

Targeting Prefetches based on L1 Hit Ratio and Est. Lat. Impact

First at the function level – those functions with low L1 Hit Ratio and high Estimated Latency Impact are targeted for prefetching :

General Exploration General Exploration viewpoint (change) Intel VTune Amplifier XE 2013

Analysis Target | Analysis Type | Summary | Bottom-up | Top-down Tree | Tasks and Frames | GaussSeidel...

Grouping: Function / Call Stack

	Cache Usage			Vectorization Usage			TLB Usage			Hardware Ev...	Hardware E
	L1 Misses	L1 Hi...	Estimate...	Vecto...	L1 Co...	L2 Co...	L1 T...	L2 T...	L1 TLB ...	L2_DATA_REA...	L2_DATA_W
func@0xe72d6	42,000,000	1.000	16728.000	0.000	0.000	0.000	0.039	0.000	4162.500	11,900,000	
Foam::GaussSeidelSmoother::smooth	5,634,300,000	0.963	144.771	4.507	7.117	194.063	0.002	0.000	0.000	11,200,000	
[vmlinux]	3,780,000,000	0.928	151.576	0.000	0.003	0.037	0.009	0.000	30.667	587,300,000	95,200,0
Foam::lduMatrix::residual	2,068,150,000	0.961	117.601	1.325	1.012	27.385	0.004	0.000	0.000	2,800,000	
Foam::lduMatrix::Amul	1,561,000,000	0.959	118.702	1.374	1.163	35.674	0.001	0.000	0.000	2,800,000	
Foam::fv::gaussGrad<double>::gradf	1,065,400,000	0.978	102.178	3.413	3.177	144.605	0.003	0.000	0.000	1,400,000	
Foam::surfaceInterpolationScheme<double>::interpolate	1,715,000,000	0.958	59.751	0.916	0.415	9.796	0.004	0.000	0.000	2,800,000	
Foam::fvc::surfaceIntegrate<double>	1,204,000,000	0.967	57.140	1.957	1.030	31.512	0.001	0.000	0.000	1,400,000	
Foam::fvc::surfaceIntegrate<Foam::Vector<double>>	847,000,000	0.964	92.190	0.920	0.613	17.107	0.001	0.000	0.000	0	
Foam::lduMatrix::negSumDiag	805,000,000	0.972	93.643	0.802	0.381	13.739	0.002	0.000	0.000	0	
Selected 1 row(s):	5,634,300,000	0.963	144.771	4.507	7.117	194.063	0.002	0.000	0.000	11,200,000	

Then at the code level : the lines with a load/store operation and low L1 Hit Ratio or high Estimated Latency Impact are targeted for prefetching for the buffers involved.

Source	Cache Usage: Self		
	L1 Misses	L1 Hit Ratio	Estimated Lat...
`// Distribute the neighbour side using psi for this cell`			
`for (register label facei=fStart; facei<fEnd; facei++)`	0	1.000	0.000
`{`			
`bPrimePtr[uPtr[facei]] -= lowerPtr[facei]*psii;`	1,928,1 ...	0.931	104.026
`}`			

Prefetching using Intrinsics : Example

```
forAll (faceCells, elemI)
{
   result[faceCells[elemI]] -= coeffs[elemI]*scalarReceiveBuf_[elemI];
}
```

No vprefetch instructions were generated in the assembly, even after giving pragma hints.

```
vprefetch1 (%r8,%r11,8)
vprefetch0 (%r8,%r11,8)
vprefetch1 0x210(%rdi,%rsi,8)
vprefetch0 0x98(%rdi,%rsi,8)
....
```

```
register const label len = faceCells.size();
#pragma noprefetch
for (register label elemI=0; elemI<len; elemI++)
{
  _mm_prefetch((char *)&result[faceCells[elemI+48]], _MM_HINT_T1);
  result[faceCells[elemI]] -= coeffs[elemI]*scalarReceiveBuf_[elemI];
  _mm_prefetch((char *)&result[faceCells[elemI+8]], _MM_HINT_T0);

  _mm_prefetch((char *)&coeffs[elemI+64], _MM_HINT_T1);
  _mm_prefetch((char *)&coeffs[elemI+16], _MM_HINT_T0);

  _mm_prefetch((char *)&scalarReceiveBuf_[elemI+64], _MM_HINT_T1);
  _mm_prefetch((char *)&scalarReceiveBuf_[elemI+16], _MM_HINT_T0);
}
```

Prefetching using Intrinsics : Vtune Snapshot

Baseline:

So. Li.	Source	CPU Time: Total	Cache Usage: Self		
			L1 Misses	L1 Hit Ratio	Estimated Lat...
409					
410	`// Multiply the field by coefficients and add into the result`				
411	`forAll(faceCells, elemI)`	6.446s	0	1.000	0.000
412	`{`				
413	`result[faceCells[elemI]] -= coeffs[elemI]*scalarReceiveBuf_[elemI];`	58.748s	798,000,000	0.934	56.860
414	`}`				

Optimized:

So. Li.	Source	CPU Time: Total	Cache Usage: Self		
			L1 Misses	L1 Hit Ratio	Estimated Lat...
414	`register const label len = faceCells.size();`				
415					
416	`#pragma noprefetch`				
417	`for (register label elemI=0; elemI<len; elemI++)`	0.565s	0	1.000	0.000
418	`{`				
419	`_mm_prefetch((char *)&result[faceCells[elemI+48]], _MM_HINT_T1);`	6.163s	14,350,000	0.987	727.000
420	`result[faceCells[elemI]] -= coeffs[elemI]*scalarReceiveBuf_[elemI];`	31.381s	139,300,000	0.984	195.071
421	`_mm_prefetch((char *)&result[faceCells[elemI+8]], _MM_HINT_T0);`	11.309s	35,700,000	0.993	165.500
422					
423	`_mm_prefetch((char *)&coeffs[elemI+64], _MM_HINT_T1);`	5.541s	10,850,000	0.987	0.000
424	`_mm_prefetch((char *)&coeffs[elemI+16], _MM_HINT_T0);`	3.901s	3,850,000	0.997	0.000
425					
426	`_mm_prefetch((char *)&scalarReceiveBuf_[elemI+64], _MM_HINT_T1);`	4.523s	0	1.000	0.000
427	`_mm_prefetch((char *)&scalarReceiveBuf_[elemI+16], _MM_HINT_T0);`	3.845s	5,250,000	0.996	0.000
428	`}`				

Functions improved – Vtune counters

Function	Baseline		Optimized	
	L1 Misses	L1 Hit Ratio	L1 Misses	L1 Hit ratio
GaussSeidelSmoother::smooth	6194650000	0.963597	3486700000	0.97056
lduMatrix::Amul	2143750000	0.958503	1422050000	0.979037
lduMatrix::sumA	566300000	0.958976	476700000	0.984022
lduMatrix::sumMagOffDiag	266000000	0.967213	301350000	0.980977
GAMGSolver::agglomerateMatrix	280000000	0.937792	110950000	0.98937
fvc::surfaceSum	154000000	0.966514	70350000	0.975185
processorFvPatchField::updateInterfaceMatrix	616000000	0.92781	91700000	0.978203
processorGAMGInterfaceField::updateInterfaceMatrix	700000000	0.952449	143850000	0.977784

L1 hit ratio has improved throughout, and L1 misses have decreased.

Prefetch Efficiency (Custom Counter Analysis in Vtune)

Formula for **Prefetch Efficiency** :

```
L1 Prefetch Miss Ratio = L1_DATA_PF1_MISS/L1_DATA_PF1
L2 Prefetch Miss Ratio = L2_DATA_PF2_MISS/L2_DATA_PF2
```

Function	Baseline		Optimized		Improvement (L1 Prefetch miss ratio)	Improvement (L2 Prefetch miss ratio)
	L1 Pf Miss Ratio	L2 Pf Miss Ratio	L1 Pf Miss Ratio	L2 Pf Miss Ratio		
Amul	0.115882875	0.644026549	0.138617445	0.399846213	0.83 X	1.6 X
lduMatrx::H	0.129310345	0.708333333	0.927835052	0.314814815	0.14 X	2.25 X
sumA	0.102265769	0.610859729	0.928057554	0.111111111	0.11 X	5.5 X
relax	0.24	0.591549296	0.619047619	0.463414634	0.4 X	1.28 X
gradF	0.335664336	0.767955801	0.07329105	0.146540027	4.5 X	5.24 X
surfaceInterpolationScheme	0.481481481	0.384615385	0.209183673	0.389285714	2.3 X	1 X

Huge Pages improvement via libhugetlbfs library

Huge memory pages (2MB) are often necessary for memory allocations on the coprocessor. With 2MB pages, TLB misses and page faults may be reduced, and there is a lower allocation cost.

Without huge pages vs huge pages : difference analysis using VTune

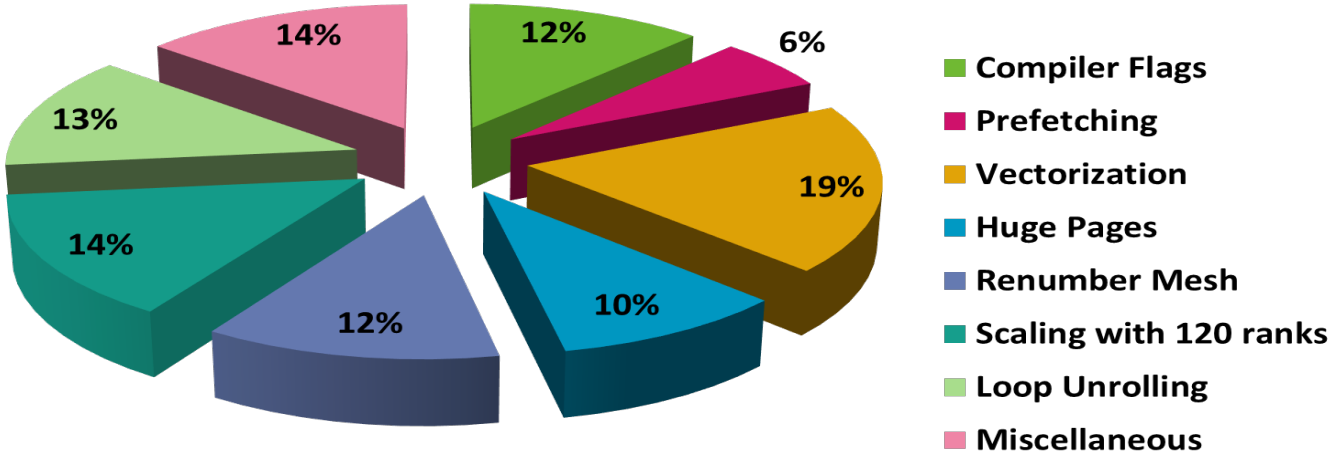
Function / Call Stack	CPU Time: Difference	CPU Time: r005ge	CPU Time: r004ge
[libOpenFOAM.so]	5818.818s	5818.818s	
func@0xe7325	4912.119s	4912.119s	
[vmlinux]	911.528s	2354.130s	1442.602s

Hardware Metrics

Clockticks:	8,319,150,000,000 - 9,175,320,000,000 = -856,169,999,999
Instructions Retired:	3,033,800,000,000 - 3,247,510,000,000 = -213,709,999,999
CPI Rate:	2.742 - 2.825 = -0.083
L1 Misses:	56,477,050,000 - 49,516,600,000 = 6,960,450,000
L1 Hit Ratio:	0.956 - 0.966 = -0.010
Estimated Latency Impact:	98.744 - 137.408 = -38.664
L1 TLB Miss Ratio:	0.010 - 0.019 = -0.010
L2 TLB Miss Ratio:	Not changed, 0.000
L1 TLB Misses per L2 TLB Miss:	774.667 - 422.932 = 351.735
Vectorization Intensity:	2.750 - 2.878 = -0.129
L1 Compute to Data Access Ratio:	1.768 - 2.145 = -0.377
L2 Compute to Data Access Ratio:	46.503 - 81.565 = -35.063

Performance Optimizations on Xeon Phi (Native mode)

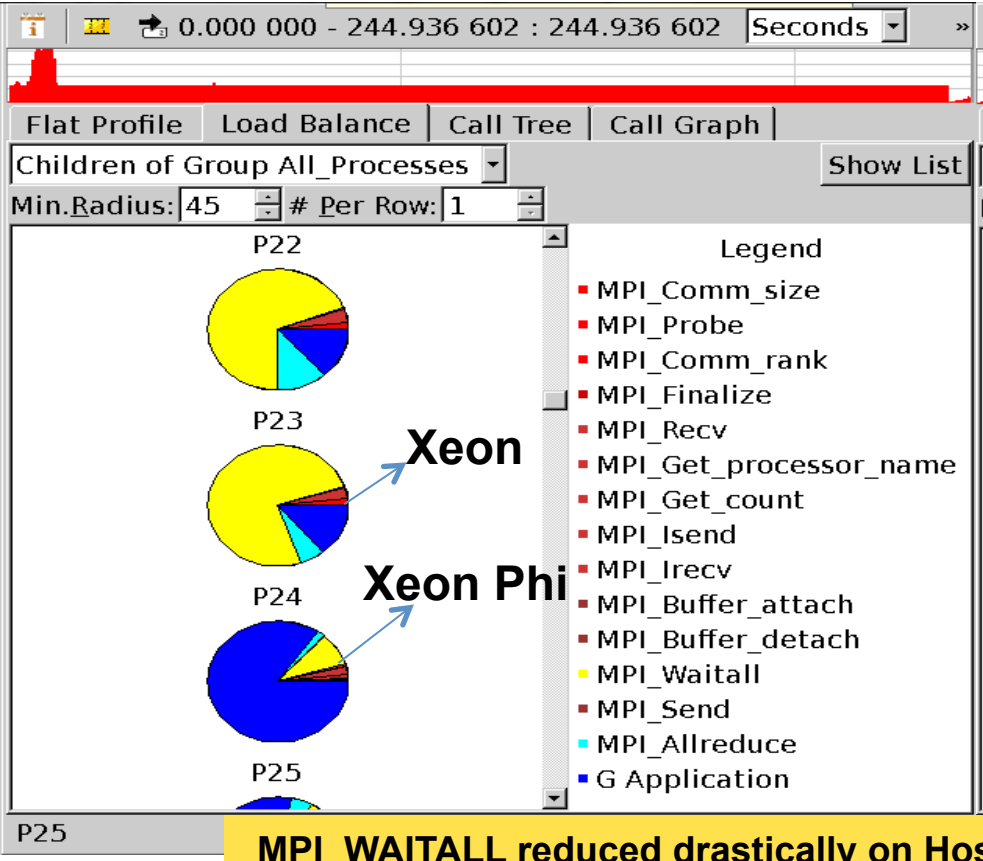
Xeon Phi Optimization Break-up in % of Total Improvements (341 secs)



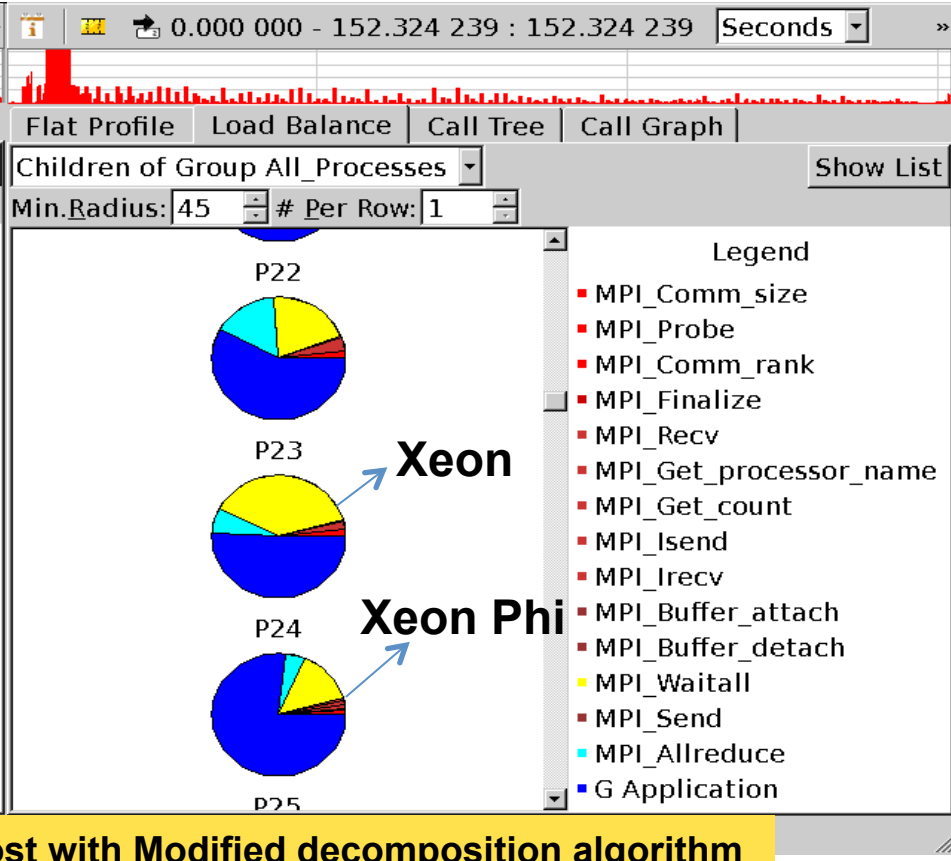
Optimization gains depend on the order in which they are applied

Modifying decomposition algorithm for weigths

Original



Modified



MPI_WAITALL reduced drastically on Host with Modified decomposition algorithm