SAND2017-1444R

Weak scaling study: Semi-coarsening and line-smoothing with MueLu

Tobias Wiesner

August 12, 2015

Chapter 1

Problem setup

1.1 Problem size

Antarctica problem

	8km	4km	2km
Number of Dofs	2,522,040	18,509,612	141,467,550
Number of nnz	119,647,360	931,492,216	7,358,086,204
Number of layers	5	10	20
Number of procs	16	128	1024

Chapter 2

Scaling study 1

2.1 Multigrid setup for MueLu

The number of levels is set to 5. ML uses different level smoothers (Gauss-Seidel sweeps). MueLu uses Jacobi and ILU as coarse solver.

2.1.1 8km resolution

```
--------------------------------------------------------------------------------
---                           Multigrid Summary                              ---
--------------------------------------------------------------------------------
Number of levels    = 5
Operator complexity = 1.07

matrix    rows        nnz  nnz/row procs
A 0    2522040  119647360    47.44  16
A 1     420340    7477960    17.79  16
A 2      47352     656296    13.86  16
A 3       4814      65236    13.55   4
A 4        420       5448    12.97   1

Smoother (level 0) both : "Ifpack2::BlockRelaxation", "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 1) both : "Ifpack2::BlockRelaxation", "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 2) both : "Ifpack2::Relaxation", Type: Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 3) both : "Ifpack2::Relaxation", Type: Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 4) pre  : "Ifpack2::ILUT", Level-of-fill: 1
Smoother (level 4) post : no smoother
```

2.1.2 4km resolution

```
--------------------------------------------------------------------------------
---                           Multigrid Summary                              ---
--------------------------------------------------------------------------------
Number of levels    = 5
Operator complexity = 1.04

matrix     rows        nnz  nnz/row procs
A 0    18509612  931492216    50.32  128
A 1     1682692   30048136    17.86  128
A 2      188086    2625396    13.96  128
A 3       18030     246124    13.65   18
A 4        1678      22164    13.21    1
```

```
Smoother (level 0) both : "Ifpack2::BlockRelaxation", "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 1) both : "Ifpack2::BlockRelaxation", "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 2) both : "Ifpack2::Relaxation", Type: Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 3) both : "Ifpack2::Relaxation", Type: Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 4) pre  : "Ifpack2::ILUT", Level-of-fill: 1
Smoother (level 4) post : no smoother
```

2.1.3 2km resolution

```
------------------------------------------------------------------------------
---                           Multigrid Summary                            ---
------------------------------------------------------------------------------
Number of levels    = 5
Operator complexity = 1.08

matrix      rows         nnz  nnz/row procs
A 0    141467550  7358086204    52.01  1024
A 1     13473100   482497456    35.81  1024
A 2      6736550   120624364    17.91  1024
A 3       744676    10458336    14.04   744
A 4        70672      973136    13.77    70

Smoother (level 0) both : "Ifpack2::BlockRelaxation", "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 1) both : "Ifpack2::BlockRelaxation", "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 2) both : "Ifpack2::BlockRelaxation", "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 3) both : "Ifpack2::Relaxation", Type: Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 4) pre  : "Ifpack2::ILUT", Level-of-fill: 1
Smoother (level 4) post : no smoother
```

2.2 General timings

Comparison of the timings between MueLu run and the results reported by Ray.

	MueLu			ML		
	8km	4km	2km	8km	4km	2km
Albany: ***Total Time***	2334	2695	10780	622.6	669.6	1100
Albany: Setup Time	138.6	779.3	5631	39.76	50.36	83.82
AztecOO: GMRES total solve time	1665	1415	4617	317.2	364.9	753.9
MueLu: Hierarchy: Setup (total)	87.09	85.93	125.5	42.62	40.57	41.97
MueLu: Hierarchy: Solve (total)	1449	1165	3444	274.6	324.3	711.9

Relative fraction of timings related to Albany total time. Note that the Solve time is contained in the GMRES solve time.

	MueLu			ML		
	8km	4km	2km	8km	4km	2km
Albany: ***Total Time***	100.0	100.0	100.0	100.0	100.0	100.0
Albany: Setup Time	5.9	28.9	52.2	6.4	7.5	7.6
AztecOO: GMRES total solve time	71.3	52.5	42.8	50.9	54.5	68.5
MueLu: Hierarchy: Setup (total)	3.7	3.2	1.2	6.8	6.1	3.8
MueLu: Hierarchy: Solve (total)	62.1	43.2	31.9	44.1	48.4	64.7

Filtered statistics with MueLu (only linear solves with less than 71 iterations are taken into account)

Mesh	# num solves	Average iters	Time per solve	Time per iter
8km	33	16.3	23.789	1.459
4km	34	23.9	24.884	1.041
2km	30	35.5	36.456	1.027

Results provided by Ray using ML

Mesh	# num solves	Average iters	Time per solve	Time per iter
8km	34	14.4	8.077	0.56
4km	33	17.9	9.829	0.548
2km	31	35.3	22.97	0.651

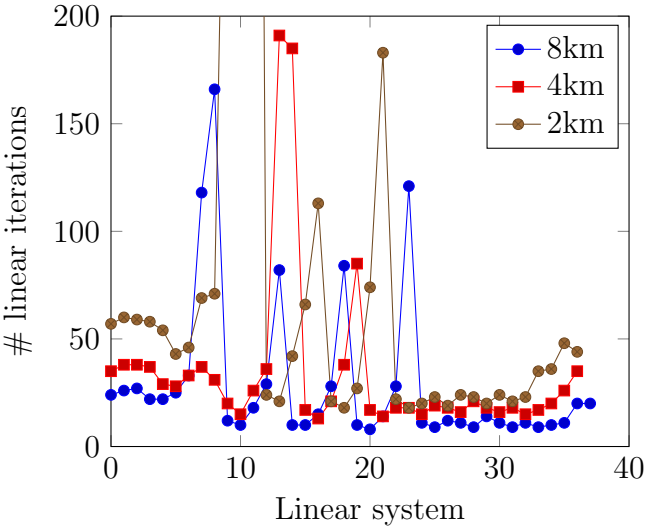
Findings:

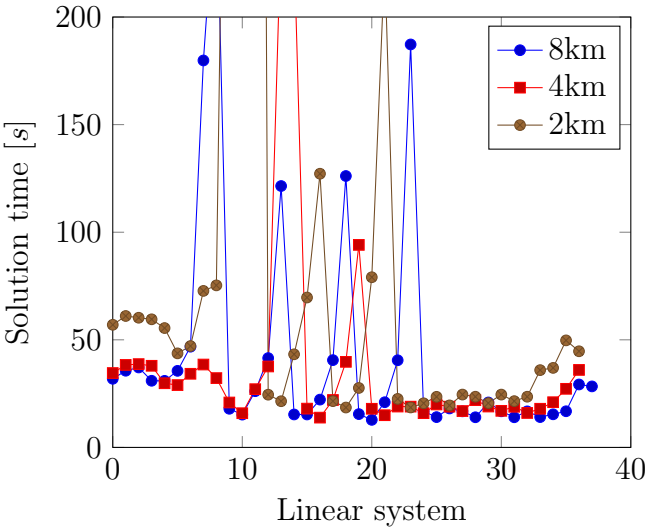
- Albany setup time drastically increasing with problem size for Tpetra stack. No increase of setup time in Epetra stack?
- MueLu setup about twice as expensive as ML setup. However: MueLu uses ILU as coarse solver versus GS
- MueLu/Ifpack2 solve much more expensive than ML! (factor 3 to 7)
- Filtered statistics: average number of iterations comparably small. Time per iteration by a factor of 2-3 more expensive in MueLu.

2.3 Number of iterations and timings

Findings:

- Number of iterations consistent with timings.
- Number of linear iterations increasing within nonlinear solver (use adaptive tolerance for linear solver?)
- Number of linear iterations slightly increasing with problem size
- Timings less affected by increasing problem size


200
150
100
50
0
linear iterations
0
10
20
30
40
Linear system
8km
4km
2km


200
150
100
50
0
Solution time [s]
0
10
20
30
40
Linear system
8km
4km
2km

Chapter 3

Scaling study 2

3.1 Multigrid setup for MueLu

The coarse level size is chosen to be 15000. We use at maximum 2 levels of semi-coarsening and smoothed aggregation on the coarser levels.

3.1.1 8km resolution

```
--------------------------------------------------------------------------------
---                           Multigrid Summary                              ---
--------------------------------------------------------------------------------
Number of levels    = 4
Operator complexity = 1.08

matrix    rows        nnz  nnz/row procs
A 0    2522040  119647360    47.44  16
A 1     420340    7477960    17.79  16
A 2      47352    1323720    27.95  16
A 3       4642     204732    44.10   4

Smoother (level 0) both : "Ifpack2::BlockRelaxation": "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3

Smoother (level 1) both : "Ifpack2::BlockRelaxation": "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3

Smoother (level 2) both : "Ifpack2::Relaxation": Type: Jacobi, sweeps: 2, damping factor: 0.3

Smoother (level 3) pre  : "Ifpack2::Relaxation": Type: Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 3) post : no smoother
```

3.1.2 4km resolution

```
--------------------------------------------------------------------------------
---                           Multigrid Summary                              ---
--------------------------------------------------------------------------------
Number of levels    = 5
Operator complexity = 1.04

matrix     rows        nnz  nnz/row procs
A 0    18509612  931492216    50.32  128
A 1     1682692   30048136    17.86  128
A 2      188086    5325780    28.32  128
A 3       17078     738044    43.22   17
```

```
A 4         900      35992    39.99    1

Smoother (level 0) both : "Ifpack2::BlockRelaxation": "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3

Smoother (level 1) both : "Ifpack2::BlockRelaxation": "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3

Smoother (level 2) both : "Ifpack2::Relaxation": Type: Jacobi, sweeps: 2, damping factor: 0.3

Smoother (level 3) both : "Ifpack2::Relaxation": Type: Jacobi, sweeps: 2, damping factor: 0.3

Smoother (level 4) pre  : "Ifpack2::Relaxation": Type: Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 4) post : no smoother
```

3.1.3 2km resolution

```
--------------------------------------------------------------------------------
---                            Multigrid Summary                             ---
--------------------------------------------------------------------------------
Number of levels    = 6
Operator complexity = 1.09

matrix      rows         nnz  nnz/row procs
A 0    141467550  7358086204    52.01  1024
A 1     13473100   482497456    35.81  1024
A 2      6736550   120624364    17.91  1024
A 3       744676    21269816    28.56   744
A 4        57400     2302976    40.12    57
A 5         3586      159100    44.37     3

Smoother (level 0) both : "Ifpack2::BlockRelaxation": "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3

Smoother (level 1) both : "Ifpack2::BlockRelaxation": "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3

Smoother (level 2) both : "Ifpack2::BlockRelaxation": "relaxation: type": Block Jacobi, sweeps: 2, damping factor: 0.3

Smoother (level 3) both : "Ifpack2::Relaxation": Type: Jacobi, sweeps: 2, damping factor: 0.3

Smoother (level 4) both : "Ifpack2::Relaxation": Type: Jacobi, sweeps: 2, damping factor: 0.3

Smoother (level 5) pre  : "Ifpack2::Relaxation": Type: Jacobi, sweeps: 2, damping factor: 0.3
Smoother (level 5) post : no smoother
```

3.2 General timings

Results for MueLu

	MueLu		
	8km	4km	2km
Albany: ***Total Time***	1792	2065	8760
Albany: Setup Time	109.4	744.1	5644
NOX Total Linear Solve	1117	785.1	2546
NOX Total Preconditioner Construction	111.3	99.71	141.1
AztecOO: GMRES total solve time	1136	803	2567
AztecOO: Operation Prec x	1024	690.8	1964
AztecOO: Ortho (Inner Product)	73.63	79.52	442.8
AztecOO: Ortho (Norm)	8.072	14.01	68.34
AztecOO: Ortho (Update)	37.27	41.01	299.4
MueLu: Hierarchy: Setup (total)	99.06	94.46	138.2
MueLu: TentativePFactory: Build (total)	4.103	3.134	2.646
MueLu: TentativePFactory: Build (total, level=3)	0.4572	0.5414	1.567
MueLu: TentativePFactory: Build (total, level=4)	–	–	0.5999
MueLu: SemiCoarsenPFactory: Build (total)	16.51	18.78	30.23
MueLu: SemiCoarsenPFactory: Build (total, level=1)	16.51	18.78	28.62
MueLu: SemiCoarsenPFactory: Build (total, level=2)	–	–	1.61
MueLu: Hierarchy: Solve (total)	1021	687.9	1955
MueLu: Hierarchy: Solve : residual calculation (total)	72.6	68.58	261.4
MueLu: Hierarchy: Solve : residual calculation (level=0)	65.48	62.13	187.6
MueLu: Hierarchy: Solve : residual calculation (level=1)	19.83	12.57	52.93
MueLu: Hierarchy: Solve : residual calculation (level=2)	1.282	7.665	48.26
MueLu: Hierarchy: Solve : residual calculation (level=3)	–	–	84.83
MueLu: Hierarchy: Solve : smoothing (total)	969	642.2	1814
MueLu: Hierarchy: Solve : smoothing (level=0)	637.5	465.3	1197
MueLu: Hierarchy: Solve : smoothing (level=1)	347.1	176.2	330
MueLu: Hierarchy: Solve : smoothing (level=2)	4.431	8.211	286.8
MueLu: Hierarchy: Solve : smoothing (level=3)	–	–	86.22
MueLu: Ifpack2Smoother: Setup Smoother (total)	37.28	37.31	56.91
MueLu: Ifpack2Smoother: Setup Smoother (total, level=0)	32.81	34.88	53.27
MueLu: Ifpack2Smoother: Setup Smoother (total, level=1)	4.421	2.403	2.261
MueLu: Ifpack2Smoother: Setup Smoother (total, level=2)	0.04704	0.02916	1.374
MueLu: Ifpack2Smoother: Setup Smoother (total, level=3)	–	–	0.02508

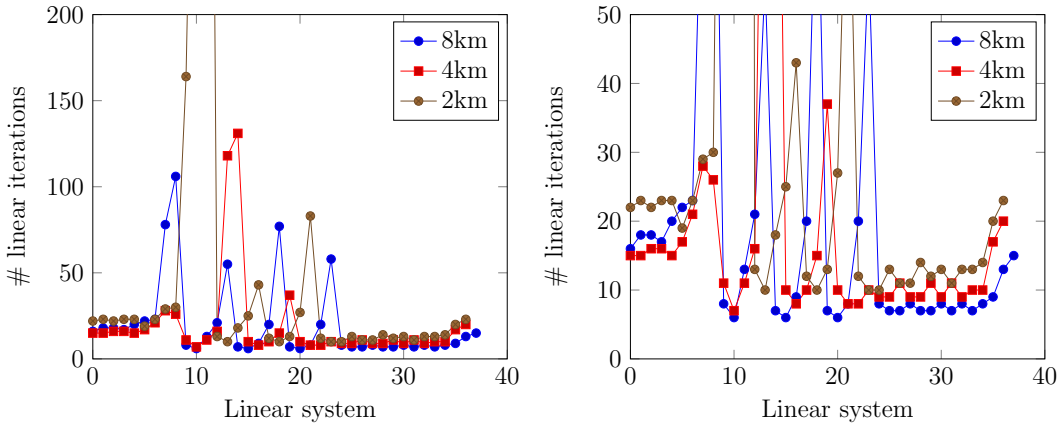
Filtered statistics with MueLu (only linear solves with less than 71 iterations are taken into account)

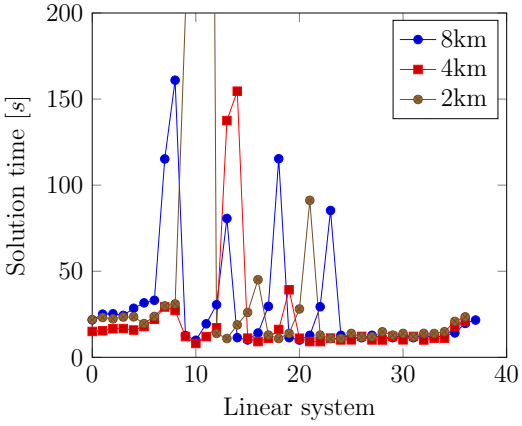
Mesh	# num solves	Average iters	Time per solve	Time per iter
8km	35	14.2	21.278	1.498
4km	35	13.5	14.598	1.078
2km	33	17.7	18.54	1.046

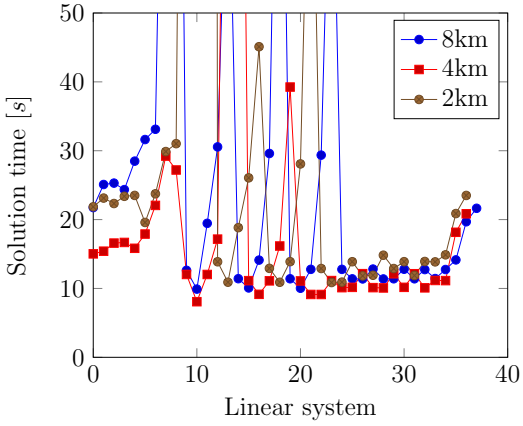
3.3 Number of iterations and timings

Findings:

- Number of iterations consistent with timings.
- Number of linear iterations increasing within nonlinear solver (use adaptive tolerance for linear solver?)
- Number of linear iterations slightly increasing with problem size
- Timings less affected by increasing problem size
- The increasing absolute timings in the overall table comes mainly from the failing linear solves
- Ifpack2 is about a factor 2-3 slower than Ifpack?




200
150
100
50
0
Solution time [s]
0
10
20
30
40
Linear system
8km
4km
2km


50
40
30
20
10
0
Solution time [s]
0
10
20
30
40
Linear system
8km
4km
2km

Chapter 4

Conclusion

- With an increasing problem size of factor 8 the Albany setup time is increasing with a factor 7.6 (using 8 times more processors). Optimal would be a constant setup time.
- The overall absolute timings are biased for the 2km example due to failing linear solves. Timings per iteration are rather constant but by a factor of 2-3 higher than for Ifpack based smoothers.
- Compared to the optimized TentativePFactory the SemiCoarsenPFactory seems to be slow (setup costs)

Chapter 5

Identified issues

5.1 Incredibly high setup times for Albany

Tested different versions of Albany and Trilinos on hopper. It seems that the problem is related with Albany.

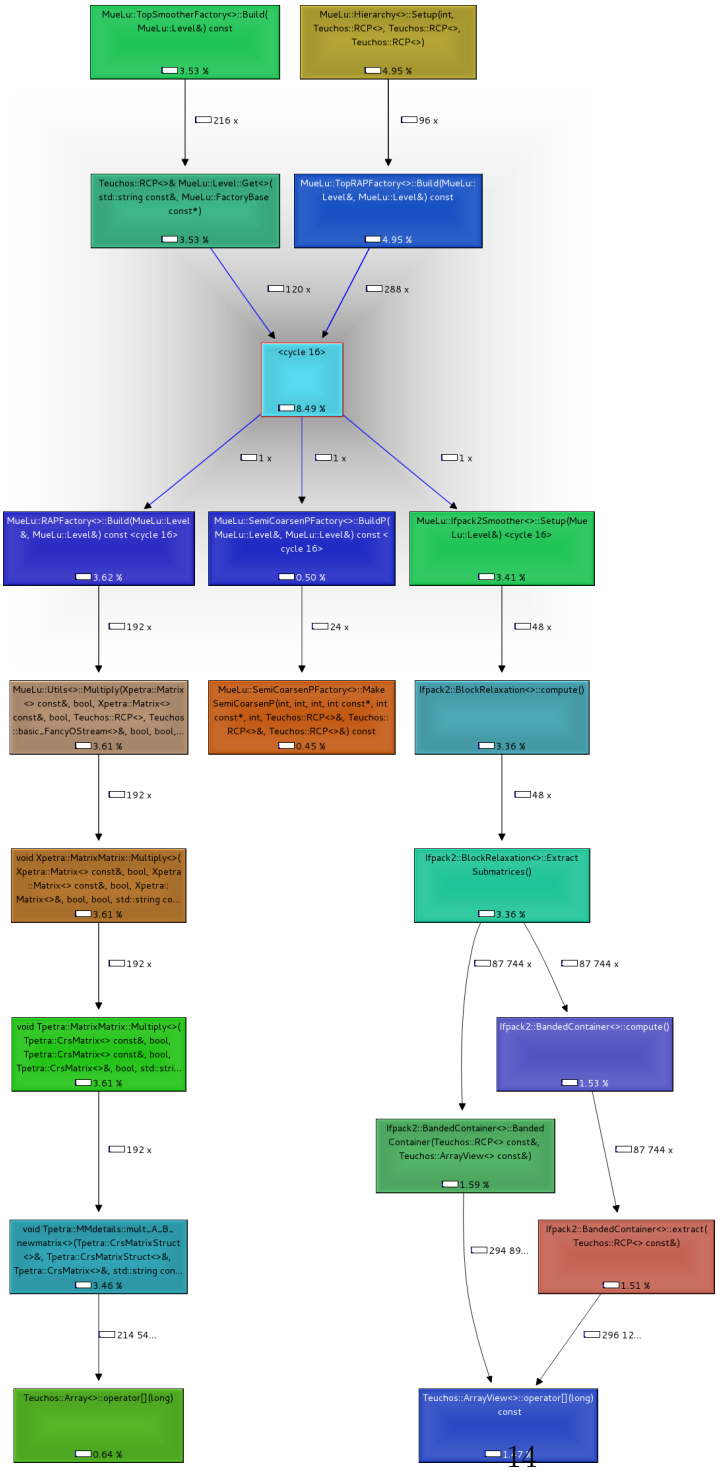
The *slow* version of Albany is:

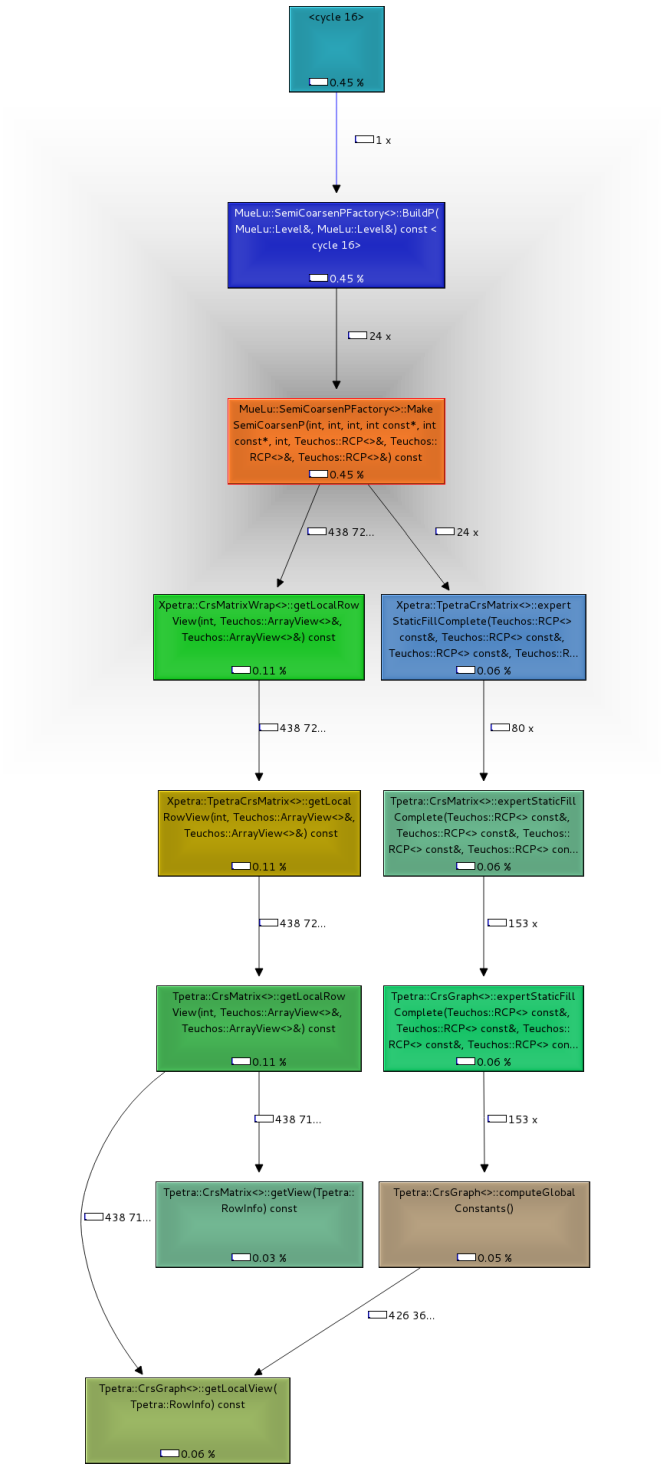
```
commit 78820bf24e09f2dfaae764a21fde508d4a10b10c
Author: Eric Phipps <etphipp@sandia.gov>
Date:   Wed Jul 22 15:36:13 2015 -0600
```

No problems have been observed with the version:

```
commit 4eba4c18d0d943afc775c3cb853854d65213f1d1
Merge: d727d9a c1f2938
Author: Irina K. Tezaur <ikalash@sandia.gov>
Date:   Thu Jul 30 15:27:55 2015 -0700
```

5.2 Setup costs for multigrid

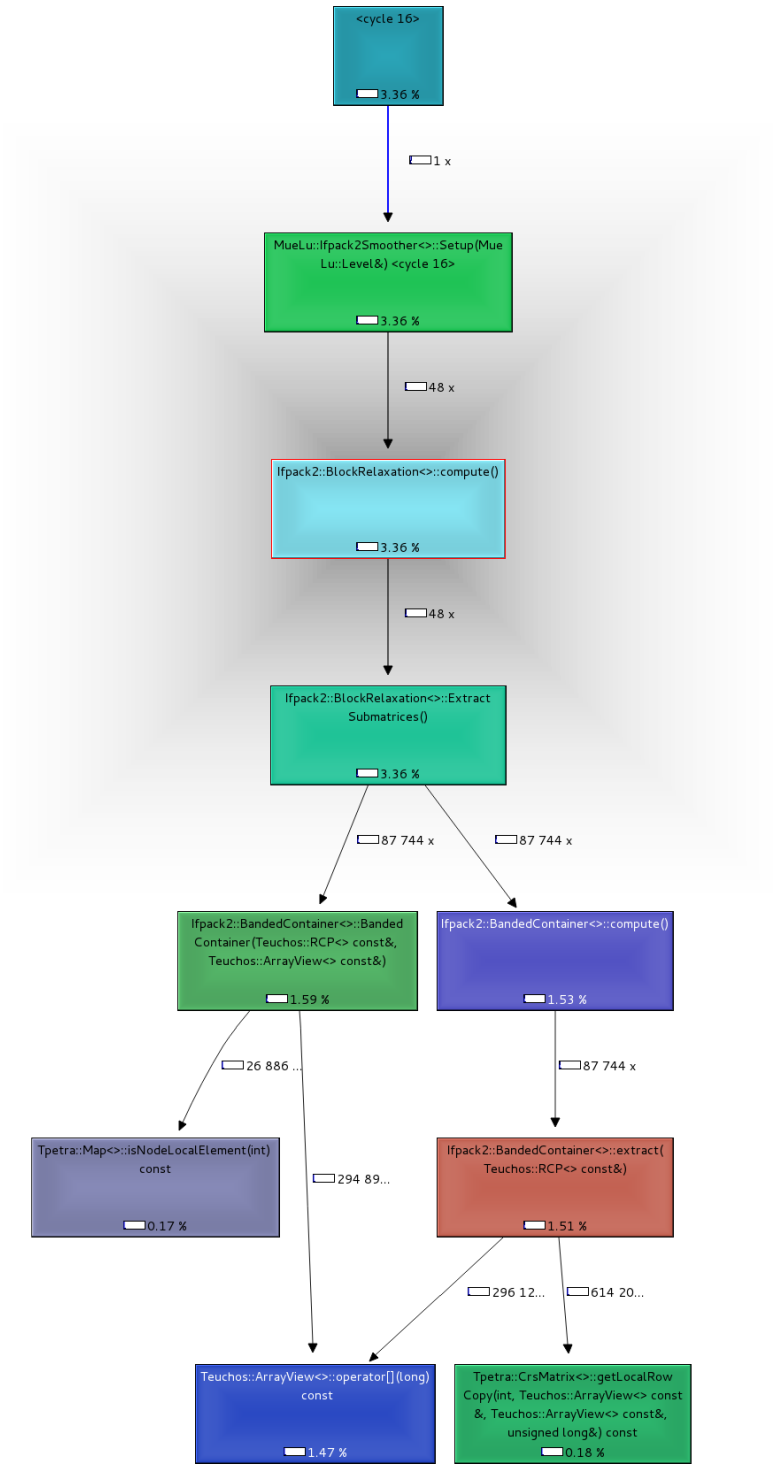


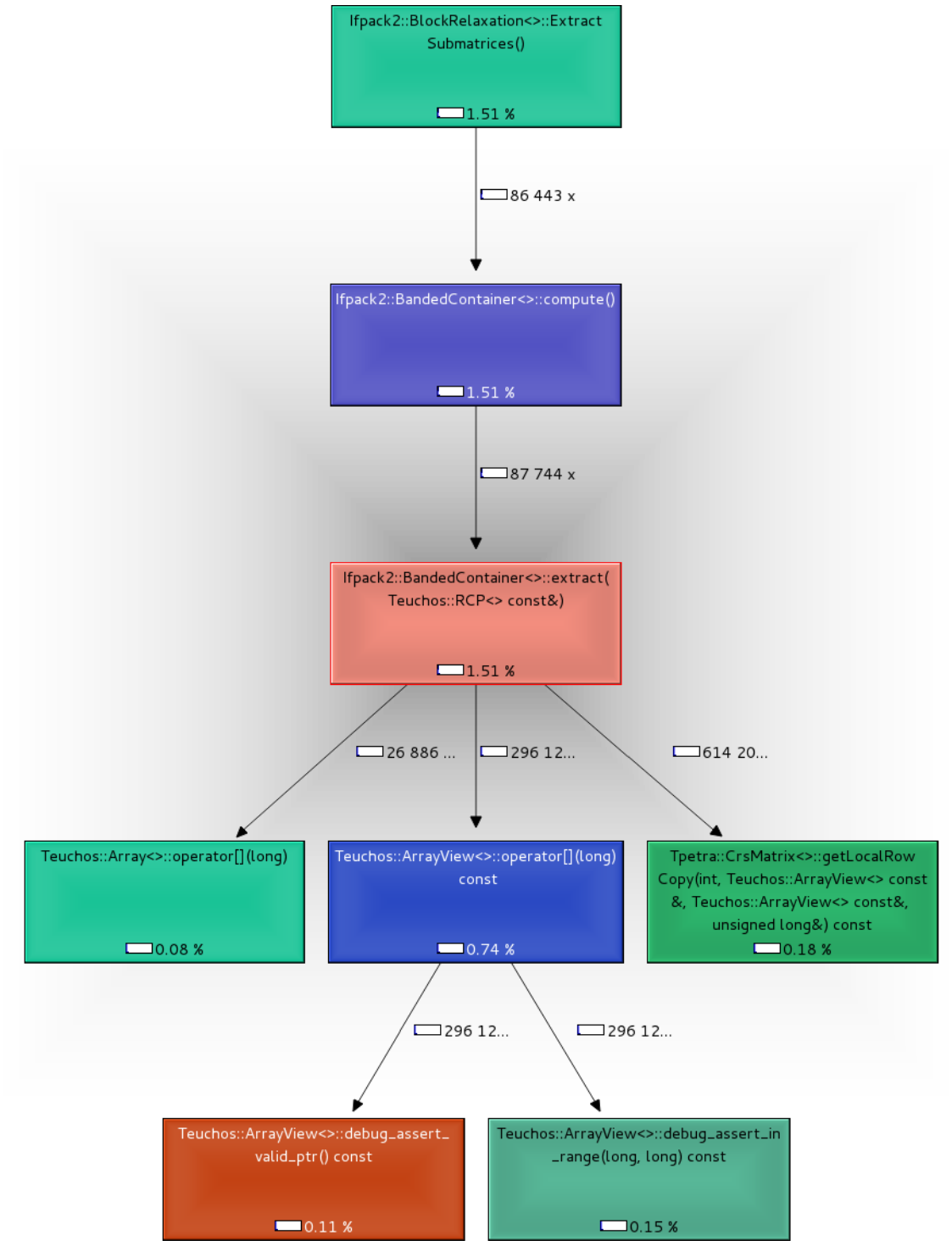

<cycle 16>
0.45 %
1 x
MueLu::SemiCoarsenPFactory<>::BuildP(MueLu::Level&, MueLu::Level&) const < cycle 16>
0.45 %
24 x
MueLu::SemiCoarsenPFactory<>::Make SemiCoarsenP(int, int, int, int const*, int const*, int, Teuchos::RCP<>&, Teuchos:: RCP<>&, Teuchos::RCP<>&) const
0.45 %
438 72...
24 x
Xpetra::CrsMatrixWrap<>::getLocalRow View(int, Teuchos::ArrayView<>&, Teuchos::ArrayView<>&) const
0.11 %
Xpetra::TpetraCrsMatrix<>::expert StaticFillComplete(Teuchos::RCP<> const&, Teuchos::RCP<> const&, Teuchos::RCP<> const&, Teuchos::R...
0.06 %
438 72...
80 x
Xpetra::TpetraCrsMatrix<>::getLocal RowView(int, Teuchos::ArrayView<>&, Teuchos::ArrayView<>&) const
0.11 %
Tpetra::CrsMatrix<>::expertStaticFill Complete(Teuchos::RCP<> const&, Teuchos::RCP<> const&, Teuchos:: RCP<> const&, Teuchos::RCP<> con...
0.06 %
438 72...
153 x
Tpetra::CrsMatrix<>::getLocalRow View(int, Teuchos::ArrayView<>&, Teuchos::ArrayView<>&) const
0.11 %
Tpetra::CrsGraph<>::expertStaticFill Complete(Teuchos::RCP<> const&, Teuchos::RCP<> const&, Teuchos:: RCP<> const&, Teuchos::RCP<> con...
0.06 %
438 71...
153 x
438 71...
Tpetra::CrsMatrix<>::getView(Tpetra:: RowInfo) const
0.03 %
Tpetra::CrsGraph<>::computeGlobal Constants()
0.05 %
426 36...
Tpetra::CrsGraph<>::getLocalView(Tpetra::RowInfo) const
0.06 %

- LineSmoother setup expensive
- Check SemiCoarsenP setup

5.3 Setup costs for LineSmoothing

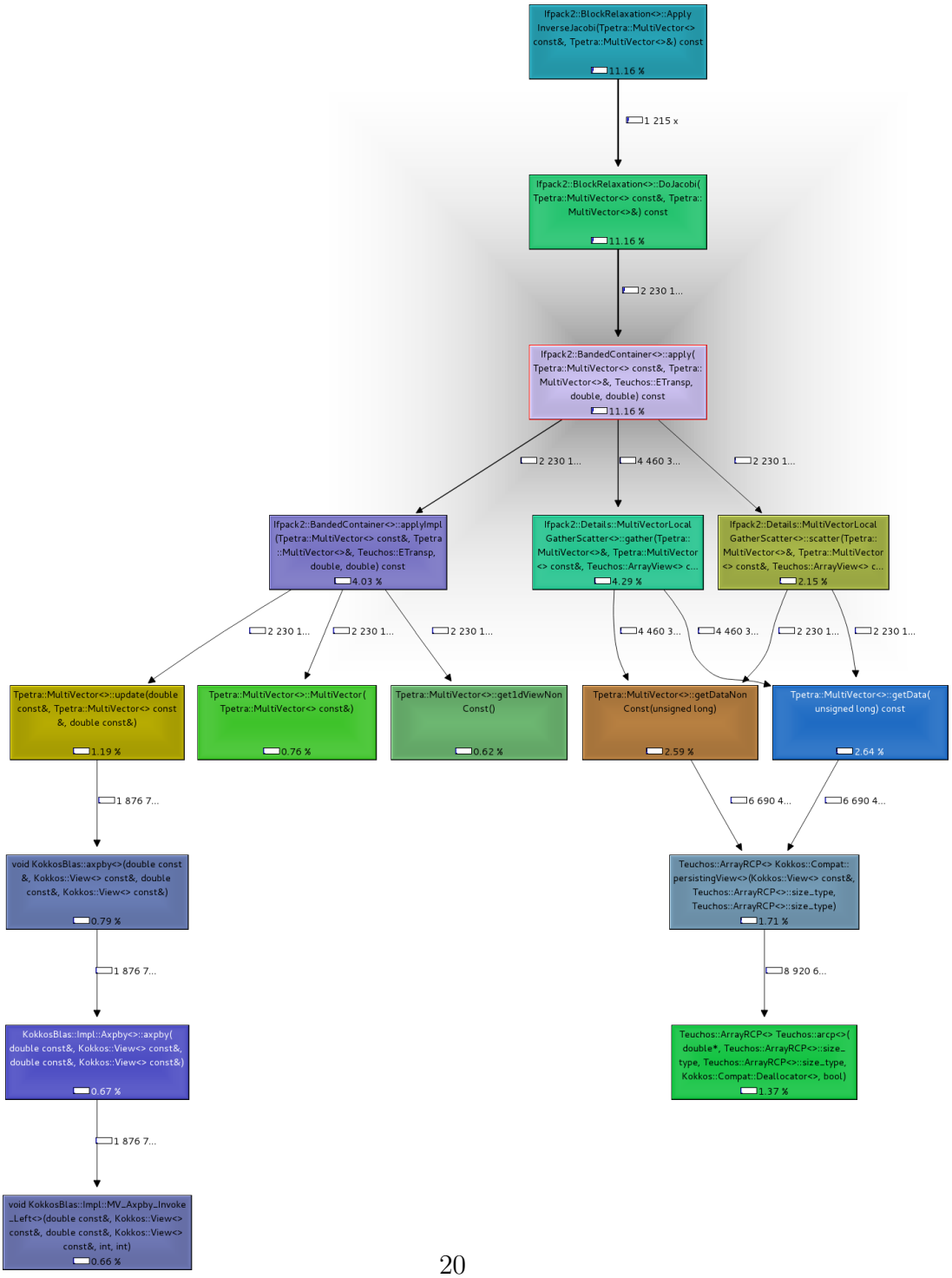
The Setup for the line smoother is expensive since it has to extract the diagonal blocks of the matrix. Note that the following graphs have been produced with a debug version of Trilinos and Albany for a 16km greenland example on 4 processors.

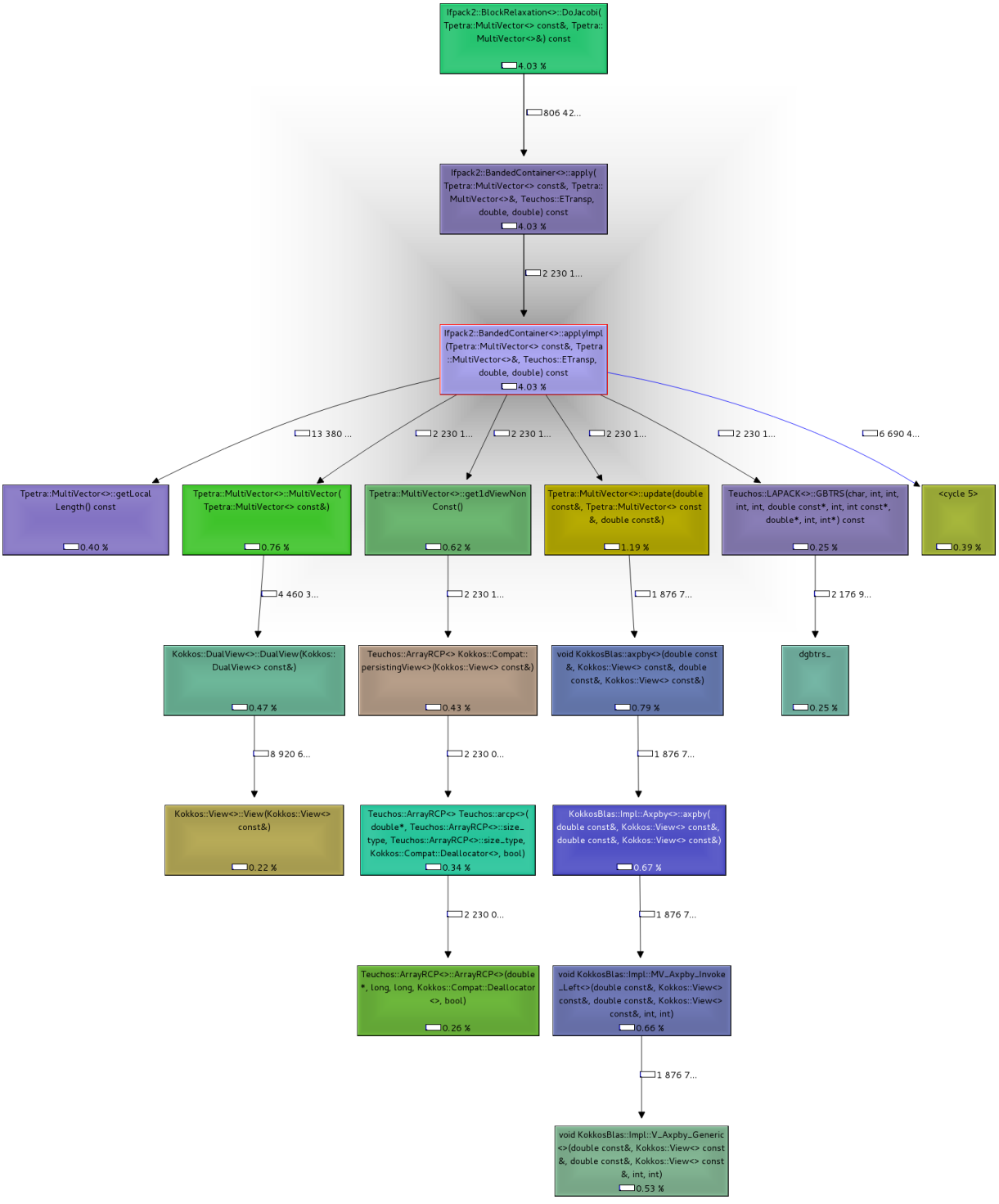

<cycle 16>
3.36 %
1 x
MueLu::Ifpack2Smoother<>::Setup(MueLu::Level&) <cycle 16>
3.36 %
48 x
Ifpack2::BlockRelaxation<>::compute()
3.36 %
48 x
Ifpack2::BlockRelaxation<>::ExtractSubmatrices()
3.36 %
87 744 x
87 744 x
Ifpack2::BandedContainer<>::BandedContainer(Teuchos::RCP<> const&, Teuchos::ArrayView<> const&)
1.59 %
Ifpack2::BandedContainer<>::compute()
1.53 %
26 886 ...
87 744 x
Tpetra::Map<>::isNodeLocalElement(int) const
0.17 %
294 89...
Ifpack2::BandedContainer<>::extract(Teuchos::RCP<> const&)
1.51 %
296 12...
614 20...
Teuchos::ArrayView<>::operator[](long) const
1.47 %
Tpetra::CrsMatrix<>::getLocalRowCopy(int, Teuchos::ArrayView<> const&, Teuchos::ArrayView<> const&, unsigned long&) const
0.18 %


Ifpack2::BlockRelaxation<>::Extract Submatrices()
1.51 %
86 443 x
Ifpack2::BandedContainer<>::compute()
1.51 %
87 744 x
Ifpack2::BandedContainer<>::extract(Teuchos::RCP<> const&)
1.51 %
26 886 ...
296 12...
614 20...
Teuchos::Array<>::operator[](long)
0.08 %
Teuchos::ArrayView<>::operator[](long) const
0.74 %
Tpetra::CrsMatrix<>::getLocalRow Copy(int, Teuchos::ArrayView<> const &, Teuchos::ArrayView<> const&, unsigned long&) const
0.18 %
296 12...
296 12...
Teuchos::ArrayView<>::debug_assert_ valid_ptr() const
0.11 %
Teuchos::ArrayView<>::debug_assert_in _range(long, long) const
0.15 %

- Refactored code to determine optimal bandwidth for `Ifpack2::BandedContainer` to save computational time. Bandwidth is still calculated separately for each diagonal block! Avoid `isNodeLocalElement` etc...
- TODO: Calculate optimal bandwidth once in MueLu's `MueLu::Ifpack2Smoother` and use it for all diagonal blocks. This solution is less general but for usual FEM matrices ok, as the pattern of the diagonal blocks is always the same. Make it an option in the parameter lists such that the user has the choice between the general but more expensive routine (default) and the fast routine (based on some assumptions).

5.4 MueLu slower than ML – iteration phase




Ifpack2::BlockRelaxation<>::DoJacobi(Tpetra::MultiVector<> const&, Tpetra::MultiVector<>&) const
4.03 %
806 42...
Ifpack2::BandedContainer<>::apply(Tpetra::MultiVector<> const&, Tpetra::MultiVector<>&, Teuchos::ETransp, double, double) const
4.03 %
2 230 1...
Ifpack2::BandedContainer<>::applyImpl(Tpetra::MultiVector<> const&, Tpetra::MultiVector<>&, Teuchos::ETransp, double, double) const
4.03 %
13 380 ...
2 230 1...
2 230 1...
2 230 1...
2 230 1...
6 690 4...
Tpetra::MultiVector<>::getLocalLength() const
0.40 %
Tpetra::MultiVector<>::MultiVector(Tpetra::MultiVector<> const&)
0.76 %
Tpetra::MultiVector<>::get1dViewNonConst()
0.62 %
Tpetra::MultiVector<>::update(double const&, Tpetra::MultiVector<> const&, double const&)
1.19 %
Teuchos::LAPACK<>::GBTRS(char, int, int, int, double const*, int, int const*, double*, int, int*) const
0.25 %
<cycle 5>
0.39 %
4 460 3...
2 230 1...
1 876 7...
2 176 9...
Kokkos::DualView<>::DualView(Kokkos::DualView<> const&)
0.47 %
Teuchos::ArrayRCP<> Kokkos::Compat::persistingView<>(Kokkos::View<> const&)
0.43 %
void KokkosBlas::axpby<>(double const&, Kokkos::View<> const&, double const&, Kokkos::View<> const&)
0.79 %
dgbtrs_
0.25 %
8 920 6...
2 230 0...
1 876 7...
Kokkos::View<>::View(Kokkos::View<> const&)
0.22 %
Teuchos::ArrayRCP<> Teuchos::arcp<>(double*, Teuchos::ArrayRCP<>::size_type, Teuchos::ArrayRCP<>::size_type, Kokkos::Compat::Deallocator<>, bool)
0.34 %
KokkosBlas::Impl::Axpby<>::axpby(double const&, Kokkos::View<> const&, double const&, Kokkos::View<> const&)
0.67 %
2 230 0...
1 876 7...
Teuchos::ArrayRCP<>::ArrayRCP<>(double*, long, long, Kokkos::Compat::Deallocator<>, bool)
0.26 %
void KokkosBlas::Impl::MV_Axpby_Invoke_Left<>(double const&, Kokkos::View<> const&, double const&, Kokkos::View<> const&, int, int)
0.66 %
1 876 7...
void KokkosBlas::Impl::V_Axpby_Generic<>(double const&, Kokkos::View<> const&, double const&, Kokkos::View<> const&, int, int)
0.53 %

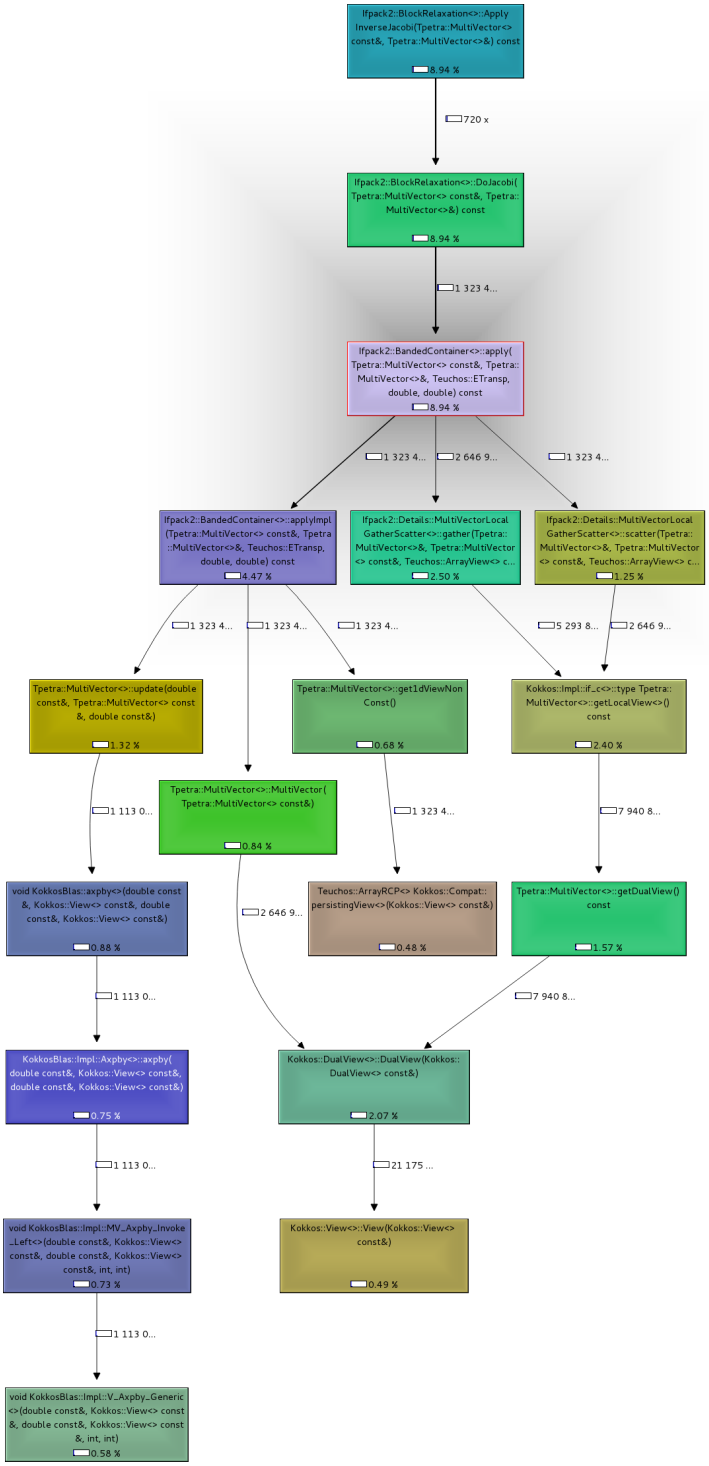
- Probably less a problem of MueLu but of Ifpack2
- It seems that we loose quite a bit time in the `gather` and `scatter` calls which are necessary to copy (?) the corresponding vector entries between the big local vector (the part that the processor owns) and the small local vector of size of the current block (allowing for permutations).
- In `Tpetra` we have to access the data through the `getData` routines and handle `ArrayRCP`s?
- The corresponding code in ML for a block Jacobi looks like this:

```
if (blkOffset == NULL) BlkPtr    = (int    *) ML_allocate((NBlks+1)*sizeof(int) );
RowsInBlk = (int    *) ML_allocate((Nrows+1)*sizeof(int) );
dtemp     = (double *) ML_allocate((Bsize+1)*sizeof(double));
res       = (double *) ML_allocate((inlen+1)*sizeof(double));
if (res == NULL)
   pr_error("Error(ML_LineJacobi): Not enough space\n");
if (blkOffset == NULL) {
   for (i = 0; i < NBlks; i++) BlkPtr[i] = i*Bsize;
   for (i = 0; i < Nrows; i++) RowsInBlk[BlkPtr[block_indices[i]]++]= i;
   ML_free( BlkPtr );
}
else
   for (i = 0; i < Nrows; i++)
      RowsInBlk[Bsize*block_indices[i] + blkOffset[i]] = i;

strcpy(N,"N");
for (iter = 0; iter < smooth_ptr->ntimes; iter++) {

   if ( (iter != 0) || (smooth_ptr->init_guess == ML_NONZERO)) {
      ML_Operator_Apply(smooth_ptr->my_level->Amat, inlen, x, inlen, res);
      for (i = 0; i < inlen; i++) res[i] = rhs[i] - res[i];
   }
   else for (i = 0; i < inlen; i++) res[i] = rhs[i];

   for (i = 0; i < NBlks; i++) {

      for (k = 0; k < Bsize; k++) dtemp[k] = res[RowsInBlk[i*Bsize+k]];
```

```
      DGTTS2_F77(N,&Bsize, &one, trid_dl[i], trid_d[i],  trid_du[i],
                   trid_du2[i],  trid_ipiv[i],dtemp,&Bsize);

      for (k = 0; k < Bsize; k++) res[RowsInBlk[i*Bsize+k]] = dtemp[k];
   }
   for (i = 0; i < inlen; i++)  x[i] += (omega * res[i]);
}
```

That is, we directly access the memory using the `RowsInBlk` helper array.

- Is something similar possible in `Tpetra`? Maybe as a special high-speed expert variant?
- According to the callgrind data, the two `gather` calls and the `scatter` call costs about 60% of the apply call. The costs for the `applyImpl` seem to be fair though.
- Rewrote `gather` and `scatter` routines using the `Kokkos::View` mechanism instead of `ArrayRCP`. The resulting call graph then looks like


Ifpack2::BlockRelaxation<>::Apply InverseJacobi(Tpetra::MultiVector<> const&, Tpetra::MultiVector<>&) const
8.94 %
720 x
Ifpack2::BlockRelaxation<>::DoJacobi(Tpetra::MultiVector<> const&, Tpetra:: MultiVector<>&) const
8.94 %
1 323 4...
Ifpack2::BandedContainer<>::apply(Tpetra::MultiVector<> const&, Tpetra:: MultiVector<>&, Teuchos::ETransp, double, double) const
8.94 %
1 323 4...
2 646 9...
1 323 4...
Ifpack2::BandedContainer<>::applyImpl (Tpetra::MultiVector<> const&, Tpetra ::MultiVector<>&, Teuchos::ETransp, double, double) const
4.47 %
Ifpack2::Details::MultiVectorLocal GatherScatter<>::gather(Tpetra:: MultiVector<>&, Tpetra::MultiVector <> const&, Teuchos::ArrayView<> c...
2.50 %
Ifpack2::Details::MultiVectorLocal GatherScatter<>::scatter(Tpetra:: MultiVector<>&, Tpetra::MultiVector <> const&, Teuchos::ArrayView<> c...
1.25 %
1 323 4...
1 323 4...
1 323 4...
5 293 8...
2 646 9...
Tpetra::MultiVector<>::update(double const&, Tpetra::MultiVector<> const &, double const&)
1.32 %
Tpetra::MultiVector<>::get1dViewNon Const()
0.68 %
Kokkos::Impl::if_c<>::type Tpetra:: MultiVector<>::getLocalView<>() const
2.40 %
Tpetra::MultiVector<>::MultiVector(Tpetra::MultiVector<> const&)
0.84 %
1 113 0...
1 323 4...
7 940 8...
void KokkosBlas::axpby<>(double const &, Kokkos::View<> const&, double const&, Kokkos::View<> const&)
0.88 %
2 646 9...
Teuchos::ArrayRCP<> Kokkos::Compat:: persistingView<>(Kokkos::View<> const&)
0.48 %
Tpetra::MultiVector<>::getDualView() const
1.57 %
1 113 0...
7 940 8...
KokkosBlas::Impl::Axpby<>::axpby(double const&, Kokkos::View<> const&, double const&, Kokkos::View<> const&)
0.75 %
Kokkos::DualView<>::DualView(Kokkos:: DualView<> const&)
2.07 %
1 113 0...
21 175 ...
void KokkosBlas::Impl::MV_Axpby_Invoke _Left<>(double const&, Kokkos::View<> const&, double const&, Kokkos::View<> const&, int, int)
0.73 %
Kokkos::View<>::View(Kokkos::View<> const&)
0.49 %
1 113 0...
void KokkosBlas::Impl::V_Axpby_Generic <>(double const&, Kokkos::View<> const &, double const&, Kokkos::View<> const &, int, int)
0.58 %

- The overall timings are somewhat reduced. More important: the relative time fraction of the `gather` and `scatter` calls compared to the `applyImpl` call is now 44% instead of 61%

Chapter 6

Scaling study 3

We use exactly the same setup as for the previous scaling study. The Trilinos version is

```
commit 89b3865d6aeb4a28cf76bf2863cb8f13eb18302d
Author: Tobias Wiesner <tawiesn@sandia.gov>
Date:   Tue Aug 11 15:58:26 2015 -0600
```

The Albany version is

```
commit 90981b6a0ab44e6384cf6ea97992ea39b56cd7ed
Author: David Littlewood <djlittl@sandia.gov>
Date:   Mon Aug 3 16:38:28 2015 -0600
```

6.1 Timings

The following table shows the maximum timings over all processors for the specific tasks. The number in brackets gives the maximum timings from the previous run in comparison.

	8km		4km		2km	
Albany: ***Total Time***	1672	(1792)	1443	(2065)	3216	(8760)
Albany: Setup Time	12	(109)	21.27	(744.1)	72	(5644)
NOX Total Linear Solve	1090	(1117)	823	(785)	2572	(2546)
NOX Total Preconditioner Construction	108	(111)	105	(99.7)	129	(141)
AztecOO: GMRES total solve time	1111	(1136)	840	(803)	2592	(2567)
AztecOO: Operation Prec x	997	(1024)	712	(691)	1946	(1964)
AztecOO: Ortho (Inner Product)	72	(73)	90.4	(79.5)	461	(443)
AztecOO: Ortho (Norm)	7.8	(8.07)	15.1	(14.0)	70.5	(68.3)
AztecOO: Ortho (Update)	37.23	(37.27)	41.0	(41.0)	294	(299)
MueLu: Hierarchy: Setup (total)	96.2	(99.06)	99.73	(94.46)	125.9	(138.2)
MueLu: TentativePFactory: Build (total)	4.08	(4.10)	3.74	(3.13)	2.651	(2.646)
MueLu: TentativePFactory: Build (total, level=3)	0.561	(0.457)	0.58	(0.54)	1.593	(1.567)
MueLu: TentativePFactory: Build (total, level=4)	–	–	–	–	0.5751	(0.5999)
MueLu: SemiCoarsenPFactory: Build (total)	16.61	(16.51)	23.92	(18.78)	29.49	(30.23)
MueLu: SemiCoarsenPFactory: Build (total, level=1)	16.61	(16.51)	23.92	(18.78)	27.83	(28.62)
MueLu: SemiCoarsenPFactory: Build (total, level=2)	–	–	–	–	1.67	(1.61)
MueLu: Hierarchy: Solve (total)	994.6	(1021)	709	(688)	1936	(1955)
MueLu: Hierarchy: Solve : residual calculation (total)	72.6	(72.6)	83.74	(68.58)	272.9	(261.4)
MueLu: Hierarchy: Solve : residual calculation (level=0)	65.6	(65.5)	69.28	(62.13)	193.9	(187.6)
MueLu: Hierarchy: Solve : residual calculation (level=1)	19.8	(19.8)	15.22	(12.57)	57.77	(52.93)
MueLu: Hierarchy: Solve : residual calculation (level=2)	1.21	(1.28)	9.39	(7.665)	50.77	(48.26)
MueLu: Hierarchy: Solve : residual calculation (level=3)	–	–	–	–	81.13	(84.83)
MueLu: Hierarchy: Solve : smoothing (total)	943	(969)	646.3	(642.2)	1760	(1814)
MueLu: Hierarchy: Solve : smoothing (level=0)	624.4	(637.5)	474	(465)	1183	(1197)
MueLu: Hierarchy: Solve : smoothing (level=1)	333.5	(347.1)	175	(176)	319.8	(330)
MueLu: Hierarchy: Solve : smoothing (level=2)	3.51	(4.43)	11.6	(8.2)	270	(287)
MueLu: Hierarchy: Solve : smoothing (level=3)	–	–	–	–	84.71	(86.22)
MueLu: Ifpack2Smoother: Setup Smoother (total)	34.6	(37.28)	35.2	(37.3)	44.81	(56.91)
MueLu: Ifpack2Smoother: Setup Smoother (total, level=0)	30.23	(32.81)	30.9	(34.9)	41.32	(53.27)
MueLu: Ifpack2Smoother: Setup Smoother (total, level=1)	4.34	(4.42)	2.3	(2.4)	2.15	(2.26)
MueLu: Ifpack2Smoother: Setup Smoother (total, level=2)	0.044	(0.047)	0.027	(0.029)	1.346	(1.374)
MueLu: Ifpack2Smoother: Setup Smoother (total, level=3)	–	–	–	–	0.02502	(0.02508)

Filtered statistics with MueLu (only linear solves with less than 71 iterations are taken into account)

Mesh	# num solves	Average iters	Time per solve	Time per iter
8km	35	14.2	20.797	1.465
8km	35	14.2	20.822	1.466
8km	35	14.2	20.871	1.47
4km	35	13.5	15.747	1.163
2km	33	17.7	18.128	1.023

6.2 Conclusions

- The problem with the expensive Setup in Albany is fixed.
- Running the same simulation with the same executable using the same amount of resources on hopper gives different timings. Variation in timings are higher than expected savings. Some of the nodes seem much slower than others. This affects the maximum timings as well as mean timings over all processors. Giving the minimum timings does not make sense due to rebalancing (min time often is just zero)
- Especially the timings of the residual calculation vary drastically:

```
                                                    MinOverProcs  MeanOverProcs  MaxOverProcs
MueLu: Hierarchy: Solve : residual calculation (level=0)   87.34         124.1          193.9
```

  Problem seems to be imbalanced on the finest level.