Leibniz-Rechenzentrum

der Bayerischen Akademie der Wissenschaften

Technical Report

Optimization of a Novel Seismological Solver Code

Update of LRZ-Bericht 2006-04

Dr. Martin Felder und Orlando River
Leibniz Supercomputing Centre,
Garching/Munich

Dr. Martin Käser und Dr Michael Dumbser
Department of Earth and Environmental Sciences,
University Munich

April 2008 **LRZ-Bericht 2008-2**

Leibniz-Rechenzentrum
Boltzmannstraße 1
85748 Garching

UST-ID-Nr. DE811305931

Telefon: (089) 35831-8784
Telefax: (089) 35831-9700
E-Mail: lrzpost@lrz.de
Internet: http://www.lrz.de

1 Project Information

1.1 Project description

Research on the interior structure of the earth and its geophysical properties are mainly based on results of seismology. Today, computer simulations of the propagation of seismic waves represent an invaluable tool for the understanding of the wave phenomena, their generation and their consequences. However, the simulation of a complete, highly accurate wave field in realistic media with complex geometry an geological rheologies is still a great challenge. Therefore, the aim of the proposed project is the intensive application of the highly accurate and powerful simulation code SEISSOL in order to provide realistic simulations of earthquake scenarios. The code is able to incorporate complex geological models and accounts for a variety of geophysical processes affecting seismic wave propagation, such as strong material heterogeneities, viscoelastic attenuation and anisotropy.

Kinematic models of real earthquake rupture processes, geometrically difficult internal and external material boundaries as well as free surface topography will be included. The code is based on the so-called ADER-Discontinuous Galerkin method that has the unique property of a numerical scheme that achieves arbitrarily high approximation order for the solution of the governing partial differential equations in space and time using three-dimensional tetrahedral meshes. The flexibility of the tetrahedral meshes allows for the discretization of geometrically extremely complex three-dimensional computational domains that might be prescribed by the geological structure and the distribution of geophysical parameters.

The method is conceptually designed in a way such that the order of approximation can arbitrarily be increased in order to reach machine precision as the capacity and performance of computing facilities improve in the future. Differences between simulation data, i.e. synthetic seismograms, and reference solutions or worldwide registered real seismograms can then be interpreted as effects of the used model geometry or the geophysical model parameterization. The result will be a better understanding and knowledge of the earth's interior.

However, a comparison with other well-established simulation algorithms is indispensable as far as accuracy, memory requirements and CPU-time requirements are concerned. For this reason the European Marie Curie Research and Training Network SPICE (Seismic wave Propagation and Imaging in Complex media: a European network) was launched recently, where one of the authors (Martin Käser) is involved (www.spice-rtn.org). One of the main targets of this project is the construction of a database, where three-dimensional test models and verified seismograms are stored that can be accessed by researchers worldwide via internet. The proposed Discontinuous Galerkin method with the ADER time integration approach will contribute to this database. First results have already been presented on international conferences and published in geophysical journals [1],[2],[3],[4].

Within the Emmy Noether Programme (KA 2281/1-1) of the Deutsche Forschungsgemeinschaft Martin Käser is currently spending two years as a visiting researcher at the University of Trento in Italy. In the following phase of this programme a new research group should be established at the Department of Earth and Environmental Sciences, Geophysics section, of the Ludwig-Maximilians-Universität München. This group will mainly work on large scale numerical simulations of earthquake scenarios. Due to the close collaboration with Prof. H. Igel from the Department of Earth and Environmental Sciences in Munich within the HLRB project (h019z), the joint supervision of the PhD student J. de la Puente and the future built-up of the new research group at the same department a number of long term perspectives are opened in order to use and further develop the optimized version of SEISSOL for computationally intensive applications.

Furthermore, a strong link between research in numerical geophysics and the Leibniz-Rechenzentrum (LRZ) will be established through various seminars on different topics of numerical and computational seismology. The indispensable collaboration with high performance computing centres will be a main issue and the upcoming installation of the SGI Tornado System at the LRZ with represent one of the most important aspects.

The „Intensive Application, Optimization and Porting Initiative“ should lead to a jointly developed production code, which efficiently and accurately computes a complete three-dimensional wave field in complex geometrical and geological structures. The application of highly accurate algorithms by the use of massively parallel high performance computer technologies will contribute to the solution of actual problems in numerical seismology in order to improve ground motion prediction caused by strong earthquake events. The long term goal should be to compute global wave propagation in a frequency range (>2Hz) that is of particular interest for civil and earthquake engineers. This way, extremely precise estimations of local seismic hazard will be possible. By synthesizing highly accurate accelerograms decisions of earthquake engineers designing earthquake resistant structures can greatly be supported and will help to optimize the trade-off between safety and cost.

Additionally, the simulation of realistic earthquake scenarios together with a coupling with shallow water simulations can lead to remarkable improvements in the early-time monitoring or forecasting of tsunamis.

Therefore, the „Intensive Application, Optimization and Porting Initiative“ of the LRZ represents an excellent and promising opportunity to solve most recent, challenging and especially computationally intensive problems. We want to stress once more, that for future research in the field of computational geophysics and seismology a solid basis of the collaboration between geoscientists and experts from high performance computing centers is an essential requirement. The establishment of such a joint approach might have a pioneering character and demonstrate the importance of combining the expertise of these two field when approaching complex large scale computational problems in numerical geophysics.

References:

[1] Dumbser, M. (2005). Arbitrary High Order Schemes for the Solution of Hyperbolic Conservation Laws in Complex Domains, Shaker Verlag, Aachen.

[2] Dumbser, M. and M. Käser (2006). An Arbitrary High Order Discontinuous Galerkin Method for Elastic Waves on Unstructured Meshes II: The Three-Dimensional Isotropic Case, to appear in *Geophs. J. Int.*

[3] Käser, M., M. Dumbser (2006). An Arbitrary High Order Discontinuous Galerkin Method for Elastic Waves on Unstructured Meshes I: The Two-Dimension Isotropic Case with External Source Terms, to appear in *Geophys. J. Int.*

[4] Käser, M., M. Dumbser, J. de la Puente and H. Igel (2006). An Arbitrary High Order Discontinuous Galerkin Method for Elastic Waves on Unstructured Meshes III: Viscoelastic Attenuation, submitted to *Geophys. J. Int.*

1.2 Description of the FORTRAN Code

The existing code is based on the Discontinuous Galerkin Method using Arbitrary high order DERivatives (ADER-DG) and is programmed in FORTRAN 90. It runs in its parallel MPI version on the SGI Altix at the Leibniz-Rechenzentrum in Munich.

The data structure is arranged in a way, that the data is organized in objects (TYPE) that finally represent dynamic, multi-dimensional array. The code itself is built up by dividing the source code into modules in order to assure a logical and intuitively understandable structured.

As the numerical method (ADER-DG) represents an explicit one-step time integration scheme in order to solve the governing partial differential equations, a large amount of calculations is carried out on a rather small amount of data, i.e. the numerical algorithm is programmed in a cache-oriented manner.

The computationally most expensive operations are matrix-vector and matrix-matrix-multiplications, in particular, which are embedded within nested DO-loops.

The input is strictly based on a small number of ASCII-files, specifying the simulation parameters, the material description, the mesh geometry and auxiliary files including the coefficients of the basis functions as well as the entries of stiffness, flux and mass matrices of the DG approach. In addition, the parallel version of SEISSOL requires to read in a mesh partitioning file precomputed by a mesh partitioner (e.g. METIS) that specifies the distribution of the unstructured tetrahedral mesh on the desired number of processors.

The output can consists of quite large and numerous files consisting of huge datasets, that e.g. contain all coefficients of the approximation basis polynomials for each tetrahedral mesh element. These coefficients are used in a post-processing step to visualize the complete three-dimensional wave-field. Furthermore, time histories of single components of the wave field are outputted as seismograms in order to analyse the numerical earthquake simulations in detail. The position of the seismogram registrations can be chosen arbitrarily. The output of these seismograms enables the user to apply standard signal processing tools for the interpretation of the computed synthetic seismograms. All data is output in ASCII-format.

2 Optimizing the Solver

2.1 Code structure analysis

SeisSol is a solver for seismological wave-propagation problems. It is written in well-commented and somewhat object-oriented Fortran90. All major data arrays and parameters are thematically clustered and hidden in extensive hierarchical structures via `TYPE` definitions. As a simple example consider

```
TYPE(tStructMesh)
 |_[some arrays and scalars]
 |_TYPE(tBoundaryGeometryStruct)
 |_TYPE(tSolidBodies)
      |_[some arrays and scalars]
      |_TYPE(tCube)
      |_TYPE(tPrism)
      |_TYPE(tEllipsoid)
```

Due to the complexity of the problem these structures, and the associated `MODULE`s, tend to become somewhat lengthy. For instance, `TYPE(tGalerkin)`, responsible for the Galerkin time propagation algorithm, comprises 159 entities on its first level alone, 19 of which are other user-defined `TYPE`s. The corresponding module `galerkin3d.f90` grew to over 4300 lines.

To facilitate cooperation between the persons responsible for the project and LRZ, the entire source tree has been placed under version control using a Subversion repository. Currently there is only one branch of code because commits were fairly infrequent and did create conflicts. Since coding activities by the users are expected to rise in frequency, it is likely that a separate optimization branch will be opened for the next round of optimization, if applicable.

The code behaves quite typically for a scientific application, in that it roughly performs the following actions in turn:

1. Read parameter file (location of files, approximation parameters, log options, etc.)
2. Read static input data (problem description, pre-calculated matrices, etc.)
3. Setup initial field and boundary conditions
4. Enter main loops and iterate until max. time or # of iterations is reached

A simplified callgraph is shown in Figure 1. It was created on `altix2` using VTune in the following command sequence (in the following, `$EXE`, `$PAR`, and `$LOG` are assumed to contain names of the corresponding SeisSol files):

```
module load vtune/3.0
vtl activity -c callgraph -app "$EXE","$PAR" -moi "$EXE"
vtl run
vtl view -gui
```

The time needed for the above steps 1. to 3. is small compared to the time spent within the main loops, therefore optimization was restricted to the routine `QuadFreeADERGalerkin3D_us` containing these loops, and its subroutines. The performance-relevant part is summarized in Table 1, with the second column giving the approximate percentage of total time (for Benchmark 1, see below) spend in the corresponding line. Pseudo-code is printed in italics.

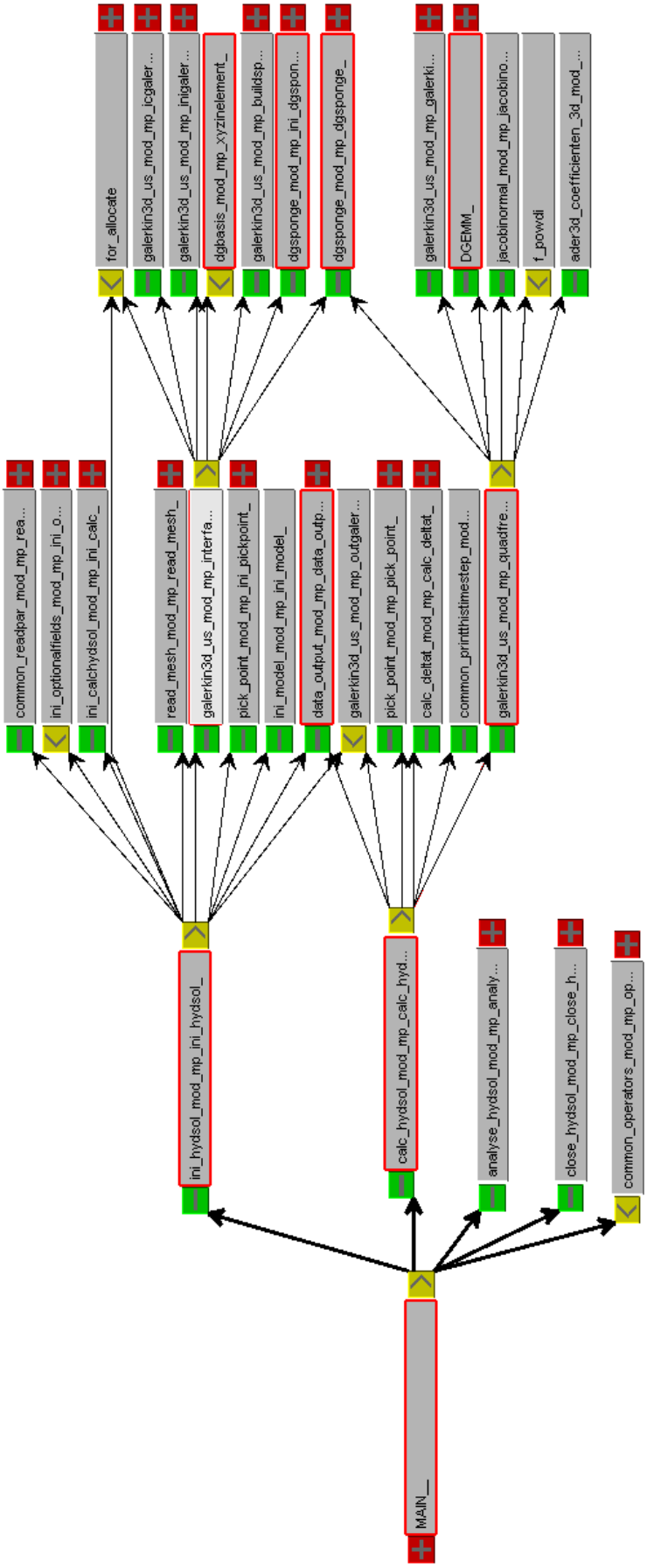


Figure 1: Incomplete callgraph for the SeisSol program. Top 10 total time routines (i.e. including subroutines) are highlighted. Since this is only a 5 iteration run, initialization carries somewhat too much weight.

Table 1: Performance-critical part of the code. %t gives the total CPU time spent, for Benchmark 1.

label	*%t*	
		`! calculate coefficients for all grid elements`
		`DO iElem = 1, MESH%nElem`
		`  [load required matrices and temporary variables]`
	3.6	`  CALL turbo_sum_permut_mat3D( ... )`
		`  DO i = 1, DISC%Galerkin%NonZeroCoeff`
	4.8	`    [get the non-zero indices r,r1,r2,l,m from a lookup-table]`
|		`    ! Do the interior of the tensor multiplication`
|	*8.0*	`    Coeff_level1(:,:) = JacobianProductSum(:,:,r1,r2,r)*  &`
(a)		`       DISC%Galerkin%Coeff_level03D(r,r1,r2,l,m)`
|	*13.0*	`    Coeff_level2(:,:) = Coeff_level2(:,:) +               &`
|		`       mdtr(r)/DISC%Galerkin%Faculty(r+1)*Coeff_level1(:,:)`
		`    ! If update marker is set (inner loop terminated), update DGwork`
		`    IF(DISC%Galerkin%NonZeroCoeffIndex(6,i).EQ.1) THEN`
	4.9	`     DISC%Galerkin%DGwork(l,:,iElem) = DISC%Galerkin%DGwork(l,:,iElem) &`
		`         + MATMUL( Coeff_level2(:,:), DISC%Galerkin%dgvar(m,:,iElem,1) )`
		`     Coeff_level2(:,:) = 0.`
		`    ENDIF`
		`  ENDDO`
		`ENDDO`
		`! now calculate fluxes for all element sides`
		`DO iElem = 1, MESH%nElem`
		`  ! Multiplication with Kxi stiffness matrix`
		`  auxvar(:,:) = 0.`
(b)		`  DO iNz = 1, DISC%Galerkin%NonZero_Kxi`
		`    l = DISC%Galerkin%IndexNonZero_Kxi(1,iNz)`
		`    m = DISC%Galerkin%IndexNonZero_Kxi(2,iNz)`
	0.9	`    auxvar(l,:) = auxvar(l,:) + &`
		`        DISC%Galerkin%CNonZero_Kxi(iNz)*DISC%Galerkin%DGwork(m,:,iElem)`
		`  ENDDO`
		`  auxMatrix(:,:) = - ( A(:,:)*JacobiT(1,1) + B(:,:)*JacobiT(1,2) + &`
		`                       C(:,:)*JacobiT(1,3) )`
	0.9	`  [MATMUL by hand]`
	3.6	`  [repeat the above procedure twice, namely for Keta and Kzeta matrices]`
		`  DO iSide = 1, MESH%GlobalElemType`
		`    [find neighboring element at side iSide from lookup table]`
		`    [load neighboring element's parameters]`
		`    ! Flux contribution of the element itself`
		`    CALL JacobiNormal3D( ... )`
	4.7	`    auxMatrix(:,:) = 0.5*MATMUL( T,MATMUL(locA+locabsA,iT) )*  &`
		`                     2.*DISC%Galerkin%geoSurfaces(iElem,iSide)`
|		`    ! Multiplication with the flux matrix of the element itself`
|		`    DO iVar = 1, EQN%nVar`
(c)	*5.5*	`      auxvar(1:LocDegFr,iVar) = MATMUL(   &`
|		`          DISC%Galerkin%FMatrix3D(1:LocDegFr,1:LocDegFr,0,1,iSide),  &`
|		`          DISC%Galerkin%DGwork(1:LocDegFr,iVar,iElem)     )`
|		`    ENDDO`
	4.0	`    [MATMUL by hand]`
		`    [case selection by neighbor's type: Inflow, Outflow, Wall,`
		`     MPI boundary, and standard case]`
	12.0	`      [--> for standard case, calculation similar to the above block`
		`       is repeated with the neighbor]`
		`   ENDDO`
		`   [update field with computed delta]`
		`ENDDO`
	sum 65.9	

As can be seen, the block *[MATMUL by hand]* occurs quite frequently. It usually looks like this:

```
DO kk = 1, EQN%nVar
   DO iDegFr = 1, LocDegFr
      dudt(iDegFr,:)=dudt(iDegFr,:)+ auxMatrix(:,kk)*auxvar(iDegFr,kk)
   ENDDO
ENDDO
```

The size of the colon dimension here is also `EQN%nVar`. These code snippets will be referred to later, when discussing optimization.

2.2 Optimization tasks

First of all, note that code for certain classes of problems was not optimized, as determined by the parameters in Table 2.

Table 2: The SeisSol software was optimized for only a subset of its possible applications.

Parameter	*Possible values*	*Notes*
problem dimensionality	2D, 3D	2D not optimized: used only for test cases
spatial order of approximation	static, p-adaptive	static not optimized: obsolete, most likely used only for reference purposes
type of algorithm used	CPU-intensive, memory-intensive	memory-intensive not optimized; will probably not be applicable to full-scale problems
parallelism	single thread, MPI	so far the MPI part of the code was not touched, since it does seem to work reasonably well

The following description lists specific major optimization work performed on the code by Subversion repository revision (rXX), and other important changes.

Revision 07: Last revision without p-adaptation.

Revision 18: Last revision before optimization.

- Martin Kaeser implemented p-adaptation: Here the order of the approximation basis functions is proportional to the size of the corresponding grid element, within a user defined range. This is reflected by the variable `LocDegFr` in the above code excerpt, which therefore varies from grid element to grid element.

Revision 20:

- Moved coefficient calculation for Label (a) out of the main loop, whereby everything is packed into a new local variable `Coeff_level0`. The new code for Label (a) merely reads

```
Coeff_level2(:,:) = Coeff_level2(:,:) +    &
                    JacobianProductSum(:,:,r1,r2,r)*Coeff_level0(i)
```

- Switched dimensions for `dndt` and `auxvar`, with corresponding loop reordering. For instance,

the block *[MATMUL by hand]* contains colons in the second dimension of `dndt`, which are in effect constituting an additional, inner loop. This is suboptimal and was changed to

```
DO iDegFr = 1, LocDegFr
   DO kk = 1, EQN%nVar
      dudt(:,iDegFr)=dudt(:,iDegFr)+ auxMatrix(:,kk)*auxvar(kk,iDegFr)
   ENDDO
ENDDO
```

Unfortunately, rearrangement of indices is counter-productive at Label (b) and similar lines in the case selection statement, but the overall effect is a slight speedup.

- Introduced a new routine `flat_sum_permut_mat3d` to replace `turbo_sum_permut_mat3d`, which contains several `IF`-clauses in the innermost of five nested loops, calculating the elements of 5-dimensional array `JacobianProductSum`. The new routine treats approximation order 1 and 2 directly, i.e. without loops, and uses a buffer (array starts at index -1) for order 3 and up, thereby avoiding the `IF`-clauses for the price of some superfluous calculations.

Revision 22:

- Replaced all performance-relevant matrix multiplications in `QuadFreeADERGalerkin3D_us` by calls to the BLAS routine `DGEMM` from the Intel Math Kernel Library (MKL). This highly optimized library routine can obtain 95% of peak performance on the Itanium2 processor. However, it is designed for large matrices, whereas in SeisSol most matrix operations involve dimensions in the order of 10. For this reason it was decided to use the F77 interface to the library instead of the F95 wrapper, which would have led to additional overhead.

With these modifications, for instance Label (c) above becomes

```
! Multiplication with the flux matrix of the element itself
!    auxvar = (FMatrix3D . DGwork)^T = DGwork^T . FMatrix3D^T
call DGEMM('T','T',nV,LocDegFr,LocDegFr,1., &
   DISC%Galerkin%DGwork(1,1,iElem),DISC%Galerkin%nDegFr,  &
   DISC%Galerkin%FMatrix3D(1,1,0,1,iSide),DISC%Galerkin%nDegFr, &
                  0.,auxvar,nV)
```

The code has been garnished with additional comments describing the matrix algebra performed, to increase readability. Furthermore, the outer product calculation in routine `DGSponge` was formulated in terms of a BLAS call to `DGER`.

Note: It is important not to accidentally use F90 array notation when passing arguments to the function, because the compiler may decide to make copies of the arrays, which renders the stride specification invalid.

Note: This in turn seems to cause strange runtime error messages („wrong dimensions for DGEMM“) when compiled with `-O3`, although the output has so far been found unaffected

2.3 Benchmark results

2.3.1 Benchmark configurations

The following configurations for the sample problem LOH4 with reduced grid size and METIS-partitioning for 1, 8 and 16 processes were selected as benchmarks (Table 3).

Table 3: Benchmark configurations.

Name	***BF order min/max***	***# of iterations***
Benchmark 1	1 / 3	200
Benchmark 2	2 / 4	50
Benchmark 3	3 / 6	30
Benchmark 4	n / n, n=0,...,5	20

The second column of the table refers to the parameters „Minimum basis functions Order“ and „Basis functions Order“, respectively.

When interpreting the results below in view of future production runs, it has to be considered that

a) the grid size is very small, and scaling is expected to be better for larger grids, due to the increasing ratio of bulk computation to MPI communication, plus
b) the runs are very short, which means the initialization overhead is higher than in practice. For Benchmark 2 on one CPU this overhead is about 5%.

Ideally, these simulation runs would be analyzed in all levels of detail by Intel's VTune suite, but due to stability and configuration problems this approach has been unsuccessful. Thus three different tools were used instead, namely, from low to high detail:

- `/usr/bin/time`: For measuring Wall Clock time for the complete run. The amount of System Time consumed is almost negligible, unless output is required from very frequent iterations. The SeisSol code also prints the execution time spent in the main calculation routines. For verification, the different time measures for one Benchmark 2 run of code Rev. 22 are given in Table 4.

Table 4: Different ways to measure computation time.

Method	***runtime / sec***	***MFlop/s***	***Comment***
time command (real)	2011	497.7	wall clock
time command (user)	1968	508.6	w/o system, e.g. I/O, swap and such
`CPU_CYCLES`	1962	510.3	w/o system etc.; should be ~=user time
`call CPU_TIME`	1897	527.7	excludes some data init routines

- `pfmon`: Used for obtaining program-wide counts of processor events and CPU cycles. A typical call would be:

```
pfmon --aggregate-results --follow-all --events=FP_OPS_RETIRED,   \
   CPU_CYCLES,IA64_INST_RETIRED_THIS,BACK_END_BUBBLE_ALL \
   mpirun -np $NP $EXE $PAR >& $LOG
```

The quotient of `FP_OPS_RETIRED` and `CPU_CYCLES`, or the results from `time` yields performance in MFlop/s, like in the Table above. For comparison, maximum performance of the Itanium2 at 1.6 GHz is assumed to be 6.4 GFlop/s, since it has four FPUs. In practice, reaching 5% peak performance is considered acceptable, 8-10% is already very good.

- `histx`: Used for drilling down to function and source code level. This utility can sample the instruction counter with its own timer interval (~ 0.977 ms), or respond to CPU events like `pfmon` does. To assign timer ticks to source lines like in the above code excerpt,

  ```
  histx -l $EXE $PAR > $LOG
  ```

was submitted as a serial job. For interactive use, execution must be bound to a specific CPU, and the runs were found to be ~5-10% slower. Each histx run produces an ASCII file in the execution directory, which can be parsed for further processing. By convention, these were renamed to `${EXE}_<cpu-event-name>.histx`., whereby TIMER was substituted for the event name for the above case. The simplest form of postprocessing the files is to parse them with the `iprep` tool, which results in a table showing the counts per routine and line, e.g.:

```
iprep seissolxx_TIMER.histx
```

When a timer run is combined with a FP counter run at a fixed sampling rate, e.g.

```
histx -e pm:FP_OPS_RETIRED@100000 -l $EXE $PAR > $LOG
```

Flop/s can be theoretically calculated for single source lines using the formula

$$\text{Flop/s} = \text{FP}*100000/(\text{ticks}*0.977[\text{ms}]) = 10^8 * \text{FP}/(0.977*\text{ticks}) \; [1/\text{s}],$$

whereby the tick length 0.977 ms is predefined by the histx TIMER event.

A perlscript was written for this purpose, yet the results need to be treated with care because sampling can be off by one or two source lines, and it is not clear whether there is a systematic shift between timer and CPU sampling. In any case, call

```
flopcalc.pl seissolxx_ FP_OPS_RETIRED.histx seissolxx_TIMER.histx
```

to obtain an `iprep`-like report containing MFlop/s and peak percentages. Another perlscript creates reports similar to Table 1, optionally using several .histx files at the same time. Use

```
histx2xls.pl galerkin3d.f90 <name1>.histx [<name2>.histx …] | \
   less -RA
```

to browse through annotated source code at the terminal, or

```
histx2xls.pl -o bench.xls galerkin3d.f90 <name1>.histx \
   [<name2>.histx …]
```

to write the output in the form of a spreadsheet. LaTeX table output will be supported in the near future.

2.3.2 Serial performance

The effect of various optimization attempts described in the previous Section was monitored using Benchmark 1.

Benchmark 3 was introduced to get a first estimate on the effect of approximation order on the results. As can be seen from the Table 5 below, the BLAS optimized version of the code reduces the runtime by over 50% with respect to the reference, whereas for the lower order Benchmark 1, the reduction is a mere 18%, worse than the manually optimized code. This is an effect of matrix dimensions increasing with the approximation order, which lets BLAS run more efficiently.

Table 5: Effect of different optimization measures.

	CPU time / s	
Code version	***Benchmark 1***	***Benchmark 3***
Revision 18: Baseline	7572	23954
partially pulled `Coeff_level0` calculation out of the loop	7114	
completed `Coeff_level0` pull out	6453	
introduced `flat_sum_permut_3d`	6800	14800
Revision 20: Changed order of indices for `dndt` and `auxvar`	5536	
Revision 22: Introduced BLAS routine calls	6232	11700

Since Benchmark 3 took too much computation time, it was replaced by Benchmark 4 for a more systematic investigation in the next Subsection. However, the fairly extensive run showed in which direction the time spent per code line will probably shift for operational runs: Label (a) alone takes up 28.7% of total CPU time, as seen in the following code excerpt from r20 (cf. Table 1):

Note the significant percentage of runtime taken up by the index lookups alone, but also the slight uncertainty of line assignments by `histx`. In this run, the innermost loop of routine `DGSponge` took another 16.2% of total CPU time. This was an incentive for reformulating it in terms of BLAS, too. In contrast, `flat_sum_permut_mat3d` lost significance, taking a mere 1.2%.

In cases where there was no apparent shift between timer and FP counter sampling, between roughly 2.5% and 13% of the peak performance was measured for the time-critical loops. This hints at the fact that access to high-dimensional, large data arrays like `JacobianProductSum(:,:,r1,r2,r)` in the code above causes frequent cache misses. Additionally, indirect indexing (`r1,r2,r` are taken from index array each loop iteration) prevents the compiler from pipelining of loops even at the highest optimization level, as compilations with the `--opt_report` switch showed. Unfortunately measuring the performance of the BLAS calls is not possible in the same manner.

Table 6: Same as Table 1, but for Benchmark 3.

%t	
2.43	`DO i = 1, DISC%Galerkin%NonZeroCoeff`
	`   ! Get the non-zero indices from the lookup-table`
2.86	`   l  = DISC%Galerkin%NonZeroCoeffIndex(1,i)`
3.03	`   m  = DISC%Galerkin%NonZeroCoeffIndex(2,i)`
4.19	`   r  = DISC%Galerkin%NonZeroCoeffIndex(3,i)`
1.55	`   IF(l.GT.LocDegFr .or. m.GT.LocDegFr .or. r.GT.LocPoly)  THEN`
	`     CYCLE`
	`   ENDIF`
1.64	`   r1 = DISC%Galerkin%NonZeroCoeffIndex(4,i)`
2.11	`   r2 = DISC%Galerkin%NonZeroCoeffIndex(5,i)`
	`   ! Do the interior of the tensor multiplication`
28.73	`   Coeff_level2(:,:) = Coeff_level2(:,:) +     &`
	`          JacobianProductSum(:,:,r1,r2,r)*Coeff_level0(i)`
	`   ! If update marker is set (inner loop terminated), update DGwork`
1.53	`   IF(DISC%Galerkin%NonZeroCoeffIndex(6,i).EQ.1) THEN`
0.73	`      DISC%Galerkin%DGwork(l,:,iElem) = DISC%Galerkin%DGwork(l,:,iElem) &`
3.90	`          + MATMUL( Coeff_level2(:,:), DISC%Galerkin%dgvar(m,:,iElem,1) )`
1.20	`      Coeff_level2(:,:) = 0.`
	`   ENDIF`
	`ENDDO`

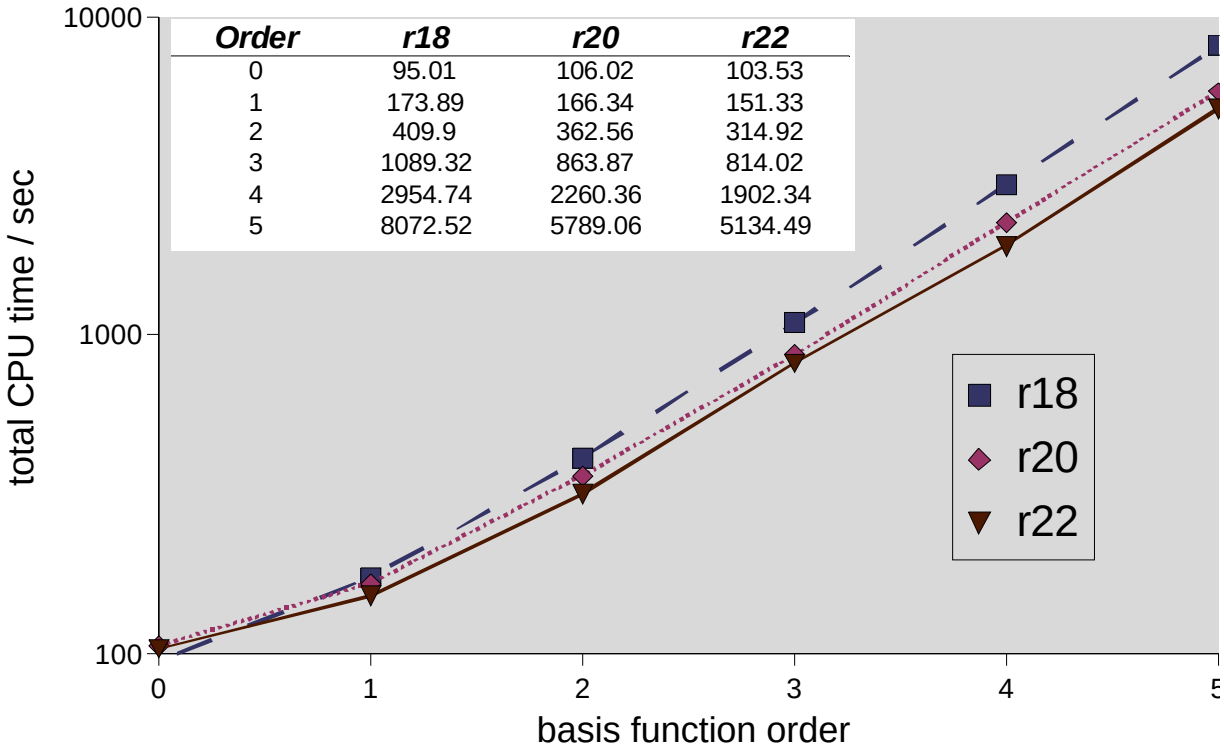
2.3.3 Dependence on basis function order

The most widely used arrays in performance critical inner loops possess dimensions of which at least one depends on the approximation order. In contrast, the number of grid elements constitutes the outermost loop and is therefore mostly optimization-neutral. Particularly the buffer arrays `dndt` and `auxvar` are dimensioned *(# variables)* x *(degrees of freedom)*, whereby the former is more or less fixed to nine in the current configuration, and the latter (DOF) depends polynomially on the basis function order:

order	0	1	2	3	4	5	6	7	8	9
DOF	1	4	10	20	35	56	84	120	165	220

As mentioned above, DGEMM calls should become more efficient at larger matrix dimensions. Figure 2 shows a comparison of three code versions by basis function order. Unfortunately, r22 does not take advantage of the larger matrices, so there must be other bottlenecks involved.

Figure 2: Dependence of run time on approximation order for Benchmark 4, on 8 CPUs. Results from three different source code versions are shown.



2.3.4 Scaling properties

Most of the benchmarks were run with serial code or at a fixed number of CPUs, because MPI call timing may change considerably as long as the serial performance is being optimized. Domain decomposition alone is used for parallelization. Since mesh partitions for 1, 8 and 16 processes were provided for the LRZ test case, the following speedups could be measured, based on CPU cycles from Benchmark 2:

Table 8: Scaling properties of the application, for Benchmark 2.

	speedup factor		
Source version	**1 CPU**	**8**	**16**
reference: r18	1	4.86	7.03
manually optimized: r20	1	5.23	3.79
BLAS: r22	1	4.92	7.34

Other scaling tests revealed that the scaling does not seem to vary much with benchmark type and code version. This is not surprising because of the low serial overhead and the very basic way in which MPI is used - each process reads its own grid chunk in the beginning, then initializes and time-propagates it, then synchronizes with other processes.

There is a reproducible anomaly for r20 at 16 CPUs, which will be looked at again. The scaling is not satisfactory yet, but this may in part be due to the relatively low number of elements in the small test grid, which makes MPI boundaries larger in comparison to the number of cells completely within one partition.

However, the main issue is that the METIS partitioning tool so far does not take into account p-adaptation. Thus while it achieves near optimal load balance for elements with equal computational effort, in cases where a significantly different number of elements with high local DOF end up in one partition, the other MPI processes stall at the end-of-iteration synchronization. This behavior is visualized in Figure 3, which was obtained using

```
module load mpi_tracing
make -j ADD_OPT=-vtrace            # clean build required
export VT_CONFIG= vt_seissol.ini
mpirun -np 8 $EXE $PAR             # submit via PBS, actually!
traceanalyzer seissol_bench5_np8.stf
```

whereby `vt_seissol.ini` contains:

```
LOGFILE-NAME seissol_bench5_np8.stf
LOGFILE-FORMAT STF
# disable all MPI activity
ACTIVITY MPI OFF
# enable all bcasts, recvs and sends
SYMBOL MPI_WAITALL ON
SYMBOL MPI_IRECV ON
SYMBOL MPI_ISEND ON
SYMBOL MPI_BARRIER ON
SYMBOL MPI_ALLREDUCE ON
SYMBOL MPI_REDUCE ON
# enable all activities in the Application class
ACTIVITY Application ON
```

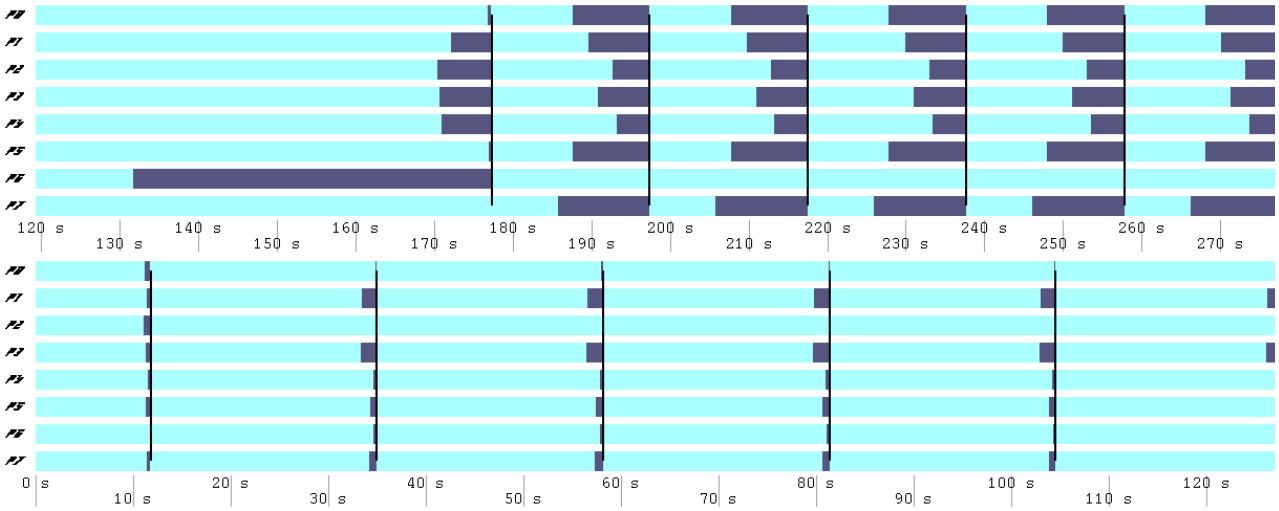


Figure 3: *Top:* Timseries of an 8-CPU MPI run with r23, using the same configuration as in Benchmark 2, i.e. basis function order 2 to 4. Part of the initialization and five iterations are shown for all eight processes. Light color designates user code execution, dark color MPI calls. The vertical bars show collective MPI operations. *Bottom:* Same but using basis function order 3 only. The load imbalance is dramatically reduced.

It is obvious from the Figure that many cycles are wasted in MPI-waits for the p-adaptive configuration. Note however that this is but a semi-quantitative comparison, since initialization and iteration times differ significantly between both runs due to the nonlinear dependence on p. Strangely enough, for instance a

benchmark run with the non-adaptive code version r07, using 200 iterations and order 2 throughout, yielded speedup factors of 5.24 and 7.68, for 8 and 16 CPUs, respectively, not much different from what the table above shows. This will be looked at again.

Another issue is that the routine in charge for synchronization, `MPIExchangeValues`, is

a) calculating the necessary message length based on the maximum DOF per element
b) written in a very general fashion, which allows for a domain size change at runtime and therefore allocates and deallocates the necessary buffer arrays dynamically at each iteration.

2.3.5 Superficial stall cycle analysis

The Itanium processor offers a wide range of performance counters (>400) suitable for further analysis of the observed hotspots in the code. They can be sampled either with VTune or by performing multiple `histx` runs as described above, using different counters. To optimize performance, it is often better not to look at the computational throughput, but rather at the delays caused by inefficient use of the CPU's functional units. The corresponding events are called "bubbles" since they effectively reduce the flow though the processors 8-stage processing pipe. For further information on the Itanium2 architecture the reader is referred to the Intel website or the latest HPC Programming and Optimization Course notes (password available upon request).

We can but scratch the complex field of stall cycle analysis here. Consider again the performance-critical inner loop segment in Table 9 below. By comparing the fraction of CPU time spent in each line with the fraction of total bubbles (counter `BACK_END_BUBBLE.ALL`), one can estimate the efficiency of that particular line compared to the average efficiency of the code sampled. In our case, it is interesting to note that loading the indices `l,m,r*` at the beginning of the loop is quite inefficient, while the calculation of `Coeff_level2` is relatively efficient, and the efficiency of the matrix multiplication at the end is slightly less than average. This is plausible because for the load operations, the Floating Point Units (FPUs) cannot be utilized, whereas in matrix and vector operations a healthier mix of computation and data transfer is possible. The analysis also suggests it may be easier to optimize the calculation of `DISC%Galerkin%Dgwork` than of `Coeff_level2`.

Further hints can be gained by considering the contributions from different pipeline stages. **Stages 1 to 3** belong to the pipeline front end (`BACK_END_BUBBLE.FE`) which deals with fetching and decoding instructions. Like in the given case, they are usually performance neutral, except in cases where the source code is voluminous and strongly non-local, i.e. contains frequent and unpredictable long branches. **Stages 4** and **5** are responsible for register management (`BE_RSE_BUBBLE.ALL`) and do not even appear here. They may come into play for very involved computations, (manually or automatically) unrolled loops, and/or nested or recursive subroutine calls, where many intermediate results have to be stored and the CPU runs out of registers. **Stage 6** (`BE_EXE_BUBBLE.ALL`) is the most frequent cause for delay in real-world applications, because its task is to feed data to the execution units (2 integer units and 2 FPUs). Since a double precision multiply-add is already 50% faster than a load from Level 2 cache, and Level 1 cache is not used for FP data, it is easy to see why these units tend to run out of data unless the code can be software-pipelined very well. In our case the compiler reports that the `Coeff_level2` line has been software-pipelined, but so far (`ifort` v9.1) this feature is only implemented for innermost loops – here matrix columns – and becomes efficient only for high loop counts. Still, it is probably responsible for the relatively few bubbles in this line. A score of sub-counters is available for fine-grained analysis of Stage 6, but their interpretation is complex and beyond the scope of this report.

With Stage 6a (`BE_L1D_FPU_BUBBLE.ALL`) we denote the two micropipelines between Stages 6 and 7, which run within the FPUs and Level 1 cache management, respectively. Stalls from the FPU micropipeline would be caused by data transfer outrunning FP computations – a rather unlikely case. Level 1 cache management may cause stalls if integer (e.g. address) data has to be swapped in and out of cache fre-

quently, but the causes may hard to get at. **Stages 7** and **8** (`BE_FLUSH_BUBBLE.ALL`) finally clean up the computations by performing exception handling and data write-back, respectively. Write-back is asynchronous and therefore mostly performance-neutral. Stage 7 also flushes the pipeline in case of branch misprediction: This may explain its relatively high influence (15-20%) in front of the first `IF`-clause and in the `Coeff_level2` loop, where software-pipelining in conjunction with a relatively low trip count in both matrix dimensions causes many instructions to be flushed at the end of the loop.

Table 9: Same as Table 1, but for r23 and Benchmark 2, with data in columns 2 to 6 showing the CPU pipeline "bubbles" in % of all bubbles which occurred during one run of the program, and this fraction further broken down into approximate pipeline stages (1 to 8). The corresponding Itanium2 Performance Counter events are given I the first row. See text for details.

	BACK_END_BUBBLE.ALL	BACK_END_BUBBLE.FE	BE_EXE_BUBBLE.ALL	BE_L1D_FPU_BUBBLE.ALL	BE_FLUSH_BUBBLE.ALL	
%t	***total***	***1-3***	***6***	***6a***	***7-8***	
2.06	*3.68*		*3.36*	*0.02*	*0.30*	`DO i = 1, DISC%Galerkin%NonZeroCoeff`
						`  ! Get the non-zero indices from the lookup-table`
0.28	*0.23*		*0.20*	*0.02*	*0.01*	`  l  = DISC%Galerkin%NonZeroCoeffIndex(1,i)`
0.27						`  m  = DISC%Galerkin%NonZeroCoeffIndex(2,i)`
3.00	*4.26*		*3.47*		*0.79*	`  r  = DISC%Galerkin%NonZeroCoeffIndex(3,i)`
0.27	*0.20*		*0.17*	*0.01*	*0.02*	`  r1 = DISC%Galerkin%NonZeroCoeffIndex(4,i)`
2.64	*5.33*		*4.12*	*0.01*	*1.20*	`  r2 = DISC%Galerkin%NonZeroCoeffIndex(5,i)`
1.03	*1.36*		*1.21*	*0.05*	*0.10*	`  IF(l.GT.LocDegFr .or. m.GT.LocDegFr .or. r.GT.LocPoly)  THEN`
						`    CYCLE`
						`  ENDIF`
						`  ! Do the interior of the tensor multiplication`
24.4	*13.8*	*0.38*	*10.42*	*0.68*	*2.31*	`  Coeff_level2(:,:) = Coeff_level2(:,:) +  &`
						`         JacobianProductSum(:,:,r1,r2,r)*Coeff_level0(i)`
						`  ! If update marker is set (inner loop terminated),`
						`  !  update DGwork`
1.29	*3.32*		*3.13*		*0.19*	`  IF(DISC%Galerkin%NonZeroCoeffIndex(6,i).EQ.1) THEN`
1.42	*1.22*	*0.26*	*0.64*	*0.02*	*0.31*	`    DISC%Galerkin%DGwork(l,:,iElem)=`
						`      DISC%Galerkin%DGwork(l,:,iElem) &`
6.39	*8.07*		*7.52*	*0.02*	*0.53*	`    +MATMUL(Coeff_level2(:,:),DISC%Galerkin%dgvar(m,:,iElem,1) )`
1.64	*0.73*		*0.72*		*0.01*	`    Coeff_level2(:,:) = 0.`
						`  ENDIF`
						`ENDDO`

To summarize, the method of stall cycle analysis is very powerful but requires some experience to yield more than trivial and/or inconclusive results. It is not recommended for users to dive deeper than to the described first level of refinement. Still, useful hints for critical code sections may be already obtained in this manner.

2.4 Conclusion and recommendations

The SeisSol software package has been put under revision control and serially optimized to a certain degree. Further optimization of the main loops in the 3D/computation intensive version of the code may be possible but may not lead to significant speed gain, except under special circumstances. To achieve further performance improvement, the following actions are recommended for the next round of optimization and/or in the frame of the current code restructuring:

1. Implement p-adaptation aware partitioning. This is very important to achieve good load balancing.
2. If possible, restructure the ADER-DG algorithm such that optimized linear algebra library routines can act on few large matrices instead of thousands of small ones.
3. Along the same lines, investigate if the use of optimized standard sparse matrix algorithms is possible without too much overhead, instead of using the current indirect indexing scheme.
4. Improve scaling by optimizing the MPI synchronization routine. It is too general for the current, static implementation of the unstructured grid.
5. There are indications that replacing EQN%nVar with a constant (i.e. `PARAMETER`) can speed up the code considerably. Maybe try creating different code versions for all possible cases.
6. Define a more comprehensive benchmark suite using a set of different problems. This would make performance assessments more robust. Artificial cases like the cubical domains used for convergences analysis could be included.
7. Investigate if a hybrid MPI/auto-parallelization or MPI/OpenMP approach leads to improvement. This is however unlikely as long as inner loop counts are very small.

4 Sparse Matrix Multiplication Benchmark

4.1 Background

A new routine, SPL_MATMUL, has been introduced into the SeisSol code to unify and speed up sparse matrix multiplication. This routine is being compared against the Math Kernel Library standard routine for this task, mkl_dcoomm. At the same time, there is the question whether sparse matrix algebra makes sense in all cases, since it requires a more complex algorithm which may slow things down if the matrix is not sparse enough. Therefore, full matrix multiplication by the Fortran intrinsic MATMUL as well as by the MKL routine DGEMM were included in the comparison. Similar investigations could be carried out for other sparse matrix manipulation routines, in case they turn out to be time-critical.

4.2 Benchmark setup

The benchmark was conducted interactively on altix2 in double precision arithmetic. SPL_MATMUL stems from SVN repository r31. To compile and run, the following command sequence was issued:

```
module switch fortran fortran/9.1 # or 8.1, see below
module load mkl/8.1 # or 8.0
module load liblrz
cd /projects/seissol/benchmarks/sparse_matrix
ifort -O2 -o sparse_matrix_test -r8 sparsemat.f90 F77Utils.f  \
  statistics.f90 ~/SeisSol/common/SP_MATMUL.f                 \
  sparse_matrix_test.f90 $LIBLRZ_LIB $MKL_LIB # or -O3
./sparse_matrix_test
```

The code itself uses loops of the form

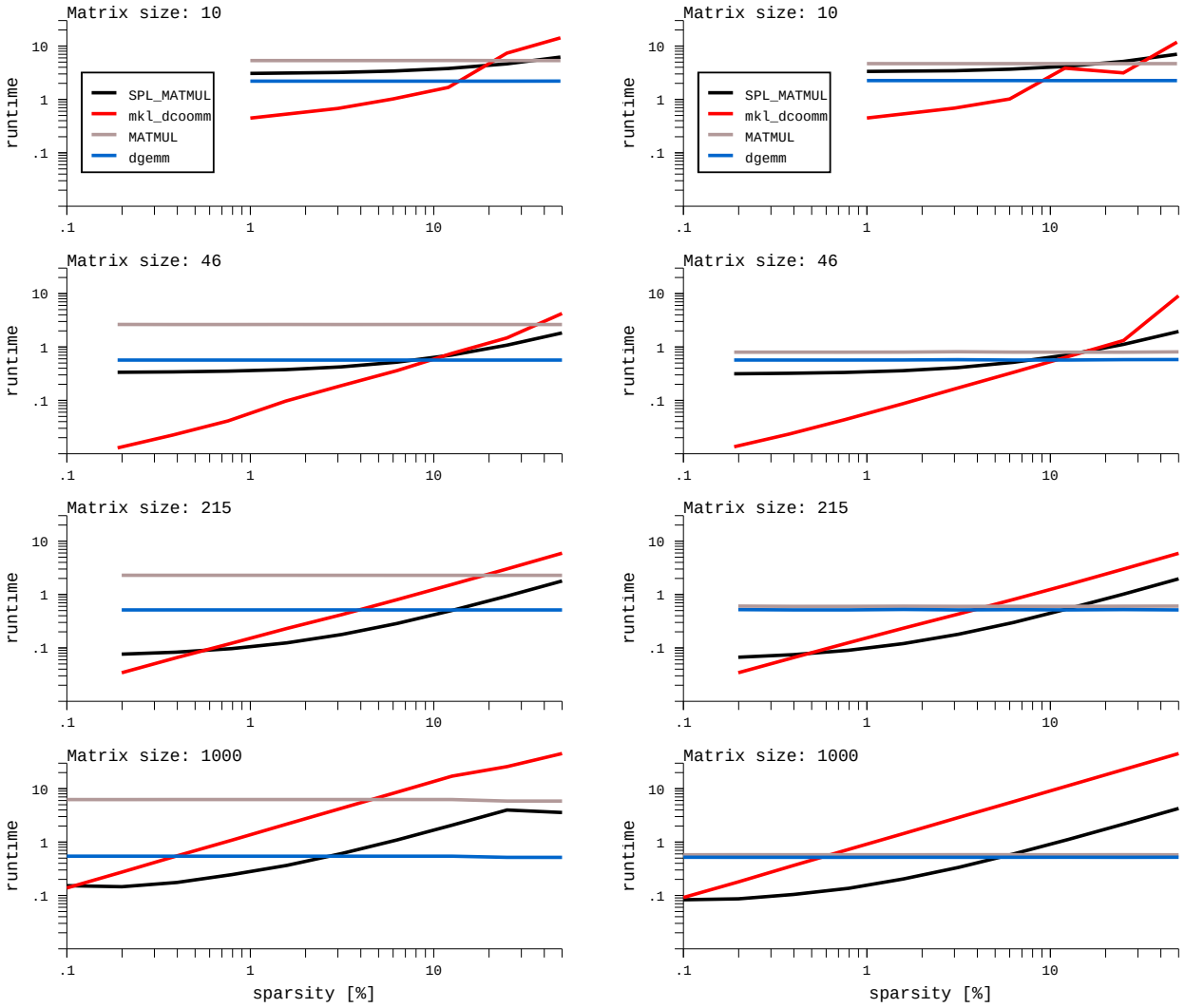
```
kDummy = get_cycles()
do i=1, nRepetitions
   call dummy_routine(A,B,C)
   call SPL_MATMUL(C(1:N,1:N), A, B(1:N,1:N), N, N)
enddo
kTime = get_cycles() - kDummy - kTimeOffs
```

whereby dummy_routine does not do anything except preventing the compiler to pass parts of the matrices in registers, merely by being defined in a separate module. Its overhead through all repetitions is measured and stored in kTimeOffset. Arrays B and C have native dimensions (N+c)×(N+c), where c is a constant with value 100 by default. nRepetitions is dynamically adjusted such that the runtime for different N is approximately comparable. get_cycles() is a routine from the LRZ library. As sparsity decreases, sparse matrix A is subsequently filled with random (non-zero) elements. The measured cycles for each combination of routine, size, and sparsity, are normalized by nRepetitions and N^3.

In addition, it was suspected that SPL_MATMUL could possibly be further improved for small matrices by passing full arrays and their leading dimensions as arguments, as done in mkl_dcoomm. This prevents the compiler from making copies of the specified subarrays prior to the routine call. The modified routine is given in the appendix.

4.3 Results

Figure 4 shows an excerpt of the benchmark results. The benchmark was run on altix2 on a single CPU, using Intel compiler version 9.1 and MKL 8.1. Additional runs were made with the modified SPL_MATMUL routine and with the old compiler version 8.1, and MKL 8.0 (Figure 5). The following observations can be made:

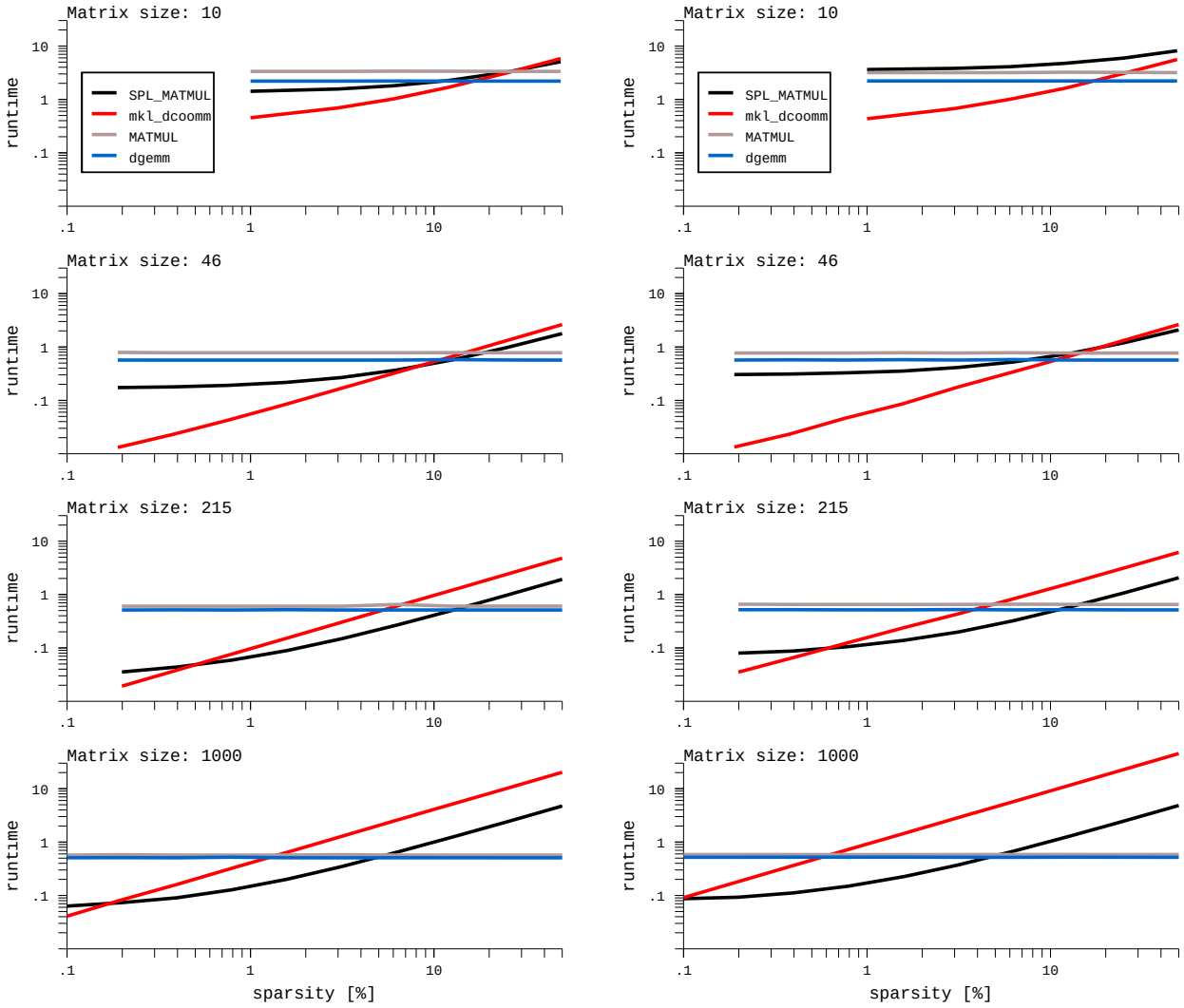


(a) Optimization option -O2.

(b) Optimization option -O3.

Figure 4: *Sparse matrix benchmark results for varying matrix size N and sparsity. Runtime is normalized to N^3 and given in arbitrary units.*

- `mkl_dcoomm` performance is strongly dependent on sparsity, such that the routine is in general only useful for very sparse matrices. Interestingly, it becomes ineffective earlier for larger matrices, i.e. the absolute number of valid elements in the matrix seem to be the performance-driving factor.
- `SPL_MATMUL` has an offset indirectly proportional to matrix size, which may be due the creation of temporary arrays (from the array slices passed as parameters) prior to entering the routine. However this offset diminishes as matrix size increases, and the procedure itself is less dependent on sparsity.
- `MATMUL` is slower than `DGEMM` in all cases, but seems to be differently implemented at -O3 and -O2 optimization. At -O2, the gap to `DGEMM` grows with the matrix size, while at -O3 it is quite small from the beginning and shrinks with the size. At N=1000 the -O3 gap to `DGEMM` is 11% (17% with the old compiler and MKL version).
- `DGEMM` has the most stable performance, above N$\simeq$100 its speed settles to about 0.5 of our arbitrary units. It does not seem to have changed between compiler and MKL versions.
- Decreasing c to 10 makes exploiting cache lines (128 bytes) at small matrix sizes easier for the MKL



(a) -O3 and modified version of `SPL_MATMUL`.

(b) -O3 but using old compiler version 8.1 and MKL version 8.0.

Figure 5: *Same as Figure 4, but using different parameters.*

routines, but did not have a notable effect on performance (not shown). It stands to reason that the data is being rearranged inside the routines prior to computations.

- The modified version of `SPL_MATMUL` (Figure 5a) features a smaller offset at low sparsity and is in general somewhat faster than the standard version. This can be seen by comparing against `DGEMM`, because as a side effect `mkl_dcoomm` was also performing faster in this run. The behavior is reproducible, but so far not completely understood.

- The main change between old and new compiler and MKL versions concerns `MATMUL`, which generally seems to have been improved, especially at small matrix sizes (Figure 5b). There may also be a slight (few %) performance increase for `SPL_MATMUL`, which is not even visible in the graphs.

- There is some (shot-)noise in the data, probably from interference by other processes, page misses, and/or issues with the timing routine. This is difficult to model in such a simple benchmark and must be watched out for in the real application. Also, it should be mentioned that the -O3 optimized code regularly produced segmentation faults after successfully executing.

4.4 Recommendations

1. For small but performance-relevant sparse matrix multiplications consider using mkl_dcoomm instead of `SPL_MATMUL`. If sparsity is >10%, use `DGEMM` instead.

2. `SPL_MATMUL` is strongest at moderate matrix size and sparsity, and at large matrices with low sparsity. The modified version of the routine should be used.

3. The level of sparsity beyond which `DGEMM` becomes very hard to beat is around 10%-20% for small matrices, and drops to 5% at N=1000.

4. The benchmark has only been run for randomly filled square matrices using the coordinate-type sparse matrix indexing convention. If the sparsity has a known structure – symmetry, blocks, multi-diagonal, etc. – there may be faster, more specialized indexing conventions and corresponding library routines available. Have a look at the MKL manual.

3.5 Original Version of SPL_MATMUL (SVN r31), by Michael Dumbser

```
C    !
C    ! The subroutine SPL_MATMUL multiplies a sparse matrix A from the left
C    ! to a full matrix B and adds the result to the matrix C:
C    !
C    ! C_ik = C_ik + A_ij * B_jk
C    !
      PURE SUBROUTINE SPL_MATMUL(C, A, B, n, o)
C    !--------------------------------------------------------------------------
      IMPLICIT NONE
C    !--------------------------------------------------------------------------
C    ! Type declaration
      TYPE tSparseMatrix
         INTEGER               :: m,n     ! Dimension of the original matrix
         INTEGER, ALLOCATABLE  :: nNonZero(:)    ! Number of non-zero entries
         INTEGER, ALLOCATABLE  :: NonZeroIndex1(:) ! Index 1 into non-zero elements
         INTEGER, ALLOCATABLE  :: NonZeroIndex2(:) ! Index 2 into non-zero elements
         REAL, ALLOCATABLE     :: NonZero(:)       ! Values
      END TYPE tSparseMatrix
C    ! Argument list declaration
      INTEGER                    :: n,o
      TYPE(tSparseMatrix)        :: A              ! (n,n)
      REAL                       :: C(n,o)         ! (n,o)
      REAL                       :: B(n,o)         ! (n,o)
C    ! Local variable declaration
      INTEGER                    :: iNonZero,i,j
      REAL                       :: BT(o,n)
      REAL                       :: CT(o,n)
C    !--------------------------------------------------------------------------
      INTENT(IN)                 :: n,o,A,B
      INTENT(INOUT)              :: C
C    !--------------------------------------------------------------------------
C    !
      CT = 0.
      BT = TRANSPOSE(B)
      DO iNonZero = 1, A%nNonZero(n)
        i = A%NonZeroIndex1(iNonZero)
        j = A%NonZeroIndex2(iNonZero)
        CT(:,i) = CT(:,i) + BT(:,j)*A%NonZero(iNonZero)
      ENDDO
      C = C + TRANSPOSE(CT)
C    !
      END SUBROUTINE SPL_MATMUL
```

3.6 Modified version of SPL_MATMUL.

The routine is listed here in free-form, since there are no separate F77-compilers anymore and thus compiling in fixed-form does not gain anything.

```
!
! The subroutine SPL_MATMUL multiplies a sparse matrix A from the left
! to a full matrix B and adds the result to the matrix C:
!
! C_ik = C_ik + A_ij * B_jk
!
PURE SUBROUTINE SPL_MATMUL(C, leadc, A, B, leadb, n, o)
!-------------------------------------------------------------------------
IMPLICIT NONE
!-------------------------------------------------------------------------
! Type declaration
TYPE tSparseMatrix
  INTEGER :: n,n ! Dimension of the original matrix
  INTEGER, ALLOCATABLE :: nNonZero(:) ! Number of non-zero entries
  INTEGER, ALLOCATABLE :: NonZeroIndex1(:) ! Index 1 into non-zero elements
  INTEGER, ALLOCATABLE :: NonZeroIndex2(:) ! Index 2 into non-zero elements
  REAL, ALLOCATABLE :: NonZero(:) ! Values
END TYPE tSparseMatrix
! Argument list declaration
INTEGER :: n,o, leadb, leadc
TYPE(tSparseMatrix) :: A  ! (n,n)
REAL :: C(leadc,o)        ! (n,o)
REAL :: B(leadb,o)        ! (n,o)
! Local variable declaration
INTEGER :: iNonZero,i,j
REAL :: BT(o,n)
REAL :: CT(o,n)
!-------------------------------------------------------------------------
INTENT(IN) :: n,o,A,B, leadb, leadc
INTENT(INOUT) :: C
!-------------------------------------------------------------------------
!
CT = 0.
BT = TRANSPOSE(B(1:n,:))
DO iNonZero = 1, A%nNonZero(n)
  i = A%NonZeroIndex1(iNonZero)
  j = A%NonZeroIndex2(iNonZero)
  CT(:,i) = CT(:,i) + BT(:,j)*A%NonZero(iNonZero)
ENDDO
C(1:n,:) = C(1:n,:) + TRANSPOSE(CT)
!
END SUBROUTINE SPL_MATMUL
```

4 Optimal Load Balance in Parallel Programs for Geophysical Simulations

Orlando Rivera, Leibniz Supercomputing Centre, Garching/Munich
Martin Käser, Department of Earth and Environmental Sciences, Univ. Munich

4.1 Space Filling Curves

Understanding and simulating seismic phenomena is of vital importance today. Seissol, the tool for the simulation of the generation and propagation of seismic waves, used for such purpose has been modified to achieve better speed up by introducing the concept of local time stepping. In the original implementation of Seissol, the time step was calculated for all elements being the smallest chosen and communicated to all elements. Thus, regardless of the shape and size of any element, all of them were doing the same calculations in one time iteration. With local time stepping each cell uses its own time measurement. In one time iteration some cells will be updated while some will not. Some subdomains have more cells to update than others, introducing load imbalance.

Standard methods like METIS [1], used to distribute data among processors are no longer capable of decomposing the domains in an efficient manner. We are now looking toward new algorithms that preserve the good qualities of the local time stepping and reduce the load imbalance.

Space Filling Curves (SFC) is, in few words, a line that passes for every point in a discrete domain, Fig. 1. We use the Hilbert space filling curve that has two nice properties. One is its shape that maintains communication at a minimum level. The second one is that we can calculate the order in the curve of any point knowing only its coordinates.

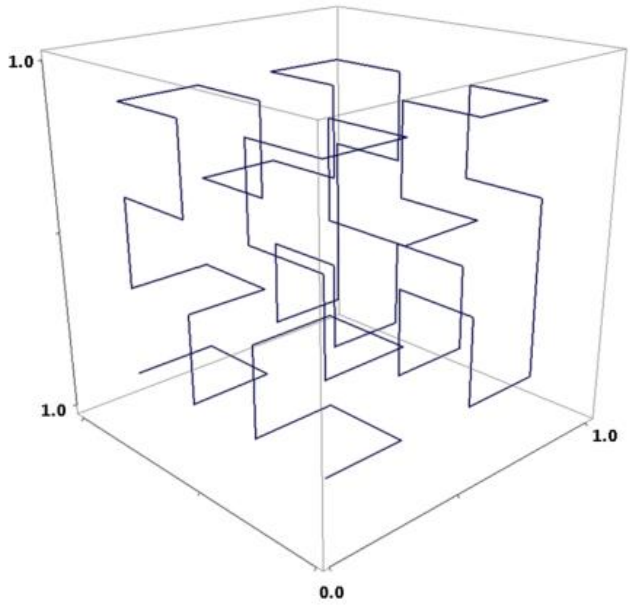


Fig. 1. Hilbert SFC of level 2 and 64 subcubes.

Generating a space filling curve is a demanding task. Besides we have also to performe a lot of calculations over millions of cells, so brute-force methods are not going to help. A SFC of level 7, normally required, has over 2.0×10^6 points whose information has to be stored somehow (each level increases the number of points by a factor of 8). Instead we use modern programming techniques and efficient algorithms that together with binary representation and Gray code reduce the ratio of calculations from $O(N^2)$ to $O(N)$. In Fig. 2 we see the load balance obtained by using SFC compared to METIS.

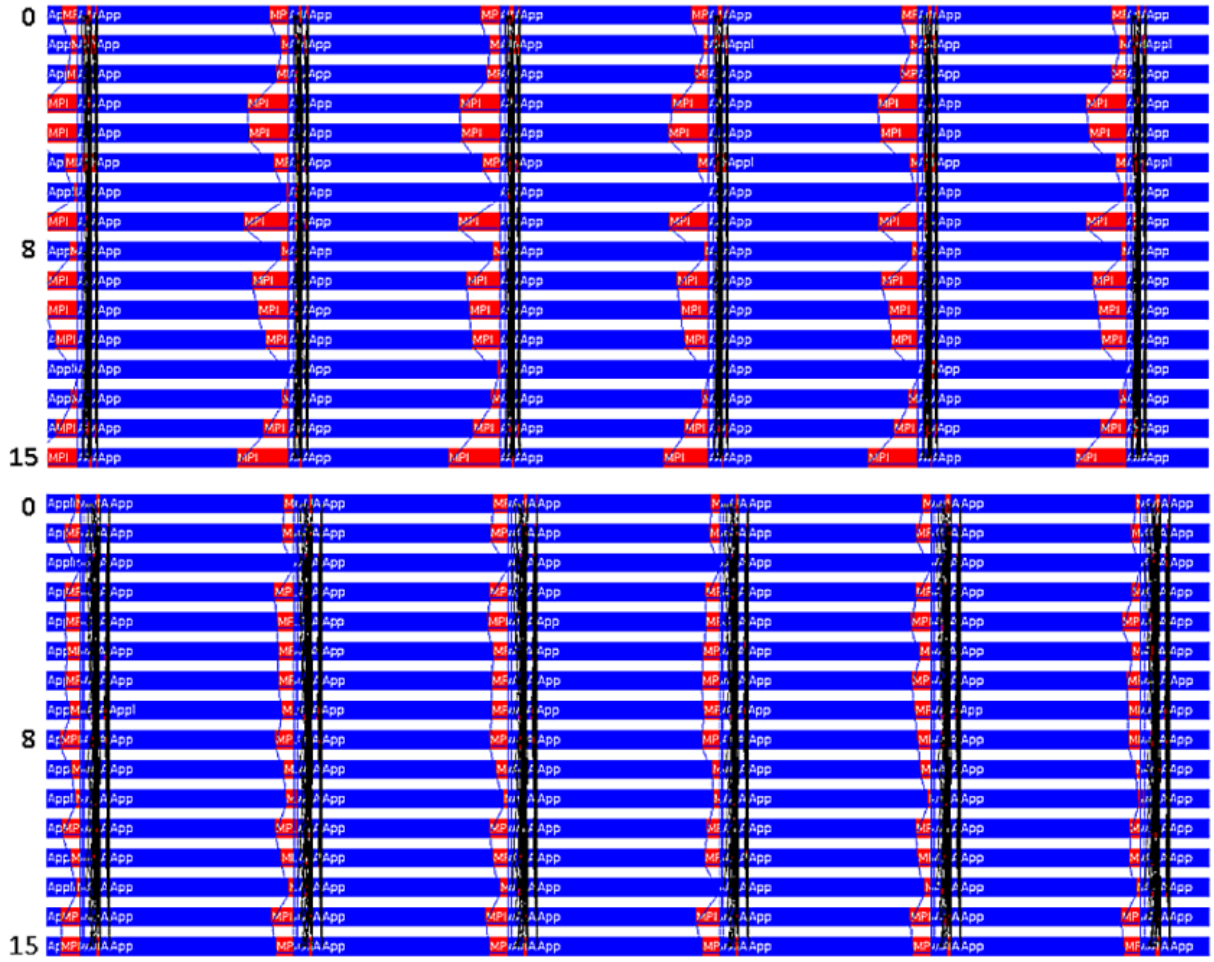


Fig. 2. Event time line. Up: Load balance using METIS, down: load balance using SFC. In blue user code and in red MPI calls, 16 processors.

4.2 How does it work?

A SFC begins at one corner of our domain and starts collecting elements until they sum up to the load that corresponds to one subdomain. This process is repeated for the next subdomian. Obviously, subdomains will no longer have the same number of cells. Instead they have a comparable load. In Fig. 4 we see a typical domain decomposition produced by SFC, while in Fig. 3 the same domain is decomposed by METIS.

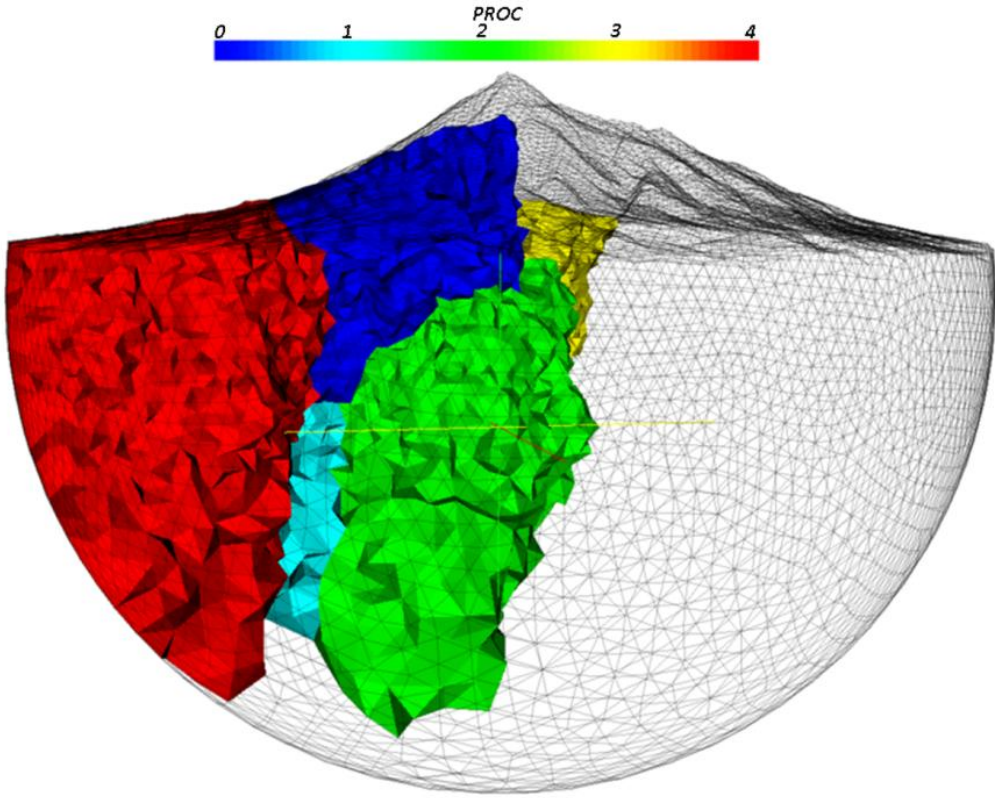


Fig. 3. Domain decompoition by METIS, 16 subdomains, visible subdomains: 0 to 4

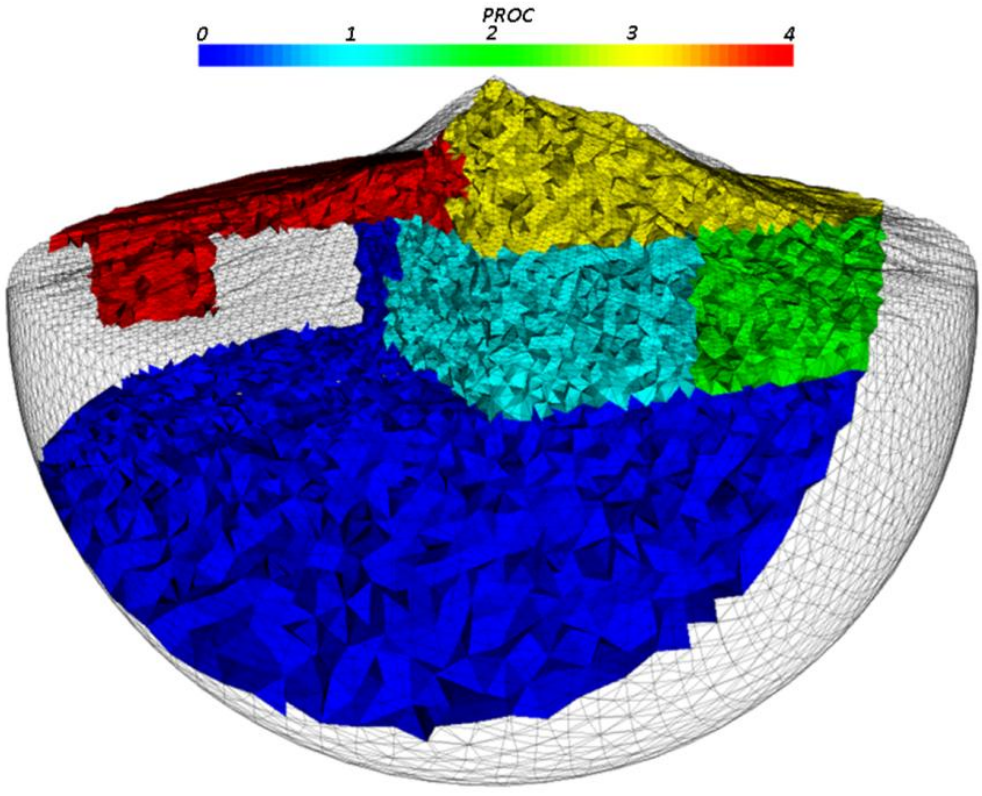


Fig. 4. Domain decomposition by SFC, 16 subdomains, visible subdomains: 0 to 4

4.3 Results and Conclusions

In general we have seen an average improvement between 15% and 20%. We are still investigating the how is time evolution of the load balance. The time tracing per iteration of all processors during one single simulation generates a spectrum-like graph. The narrower the band, the better the load balance is, as shown in Fig. 5 and 6. The more processors are used for the simulation, the more crucial the load balancing problems gets. The domain decomposition made by SFC presents a better load balance for 16 and 64 subdomains. In the later case the load imbalance is dramatically increased by the relatively small number of cells pro processor. However, the partition done by SFC presents a better performance.

One interesting property of these pictures is, if we trace one single processor during the simulation we will see it moving from top to bottom and vice versa in the band. The same effect was observed either using METIS or SFC decomposition (in Fig. 5 and 6 processor 0 is drawn in black as example). That means that a processor might be sometimes the fastest or sometimes might be the slowest. Predicting the load balance statically is a very difficult business. But most importantly, it tells us that if we start considering dynamic load balance we know as a fact that this has to be done much more often during the simulation than we thought. We will see cells jumping from subdomains to subdomains. How to migrate them without creating an excessive overhead is still an open question.

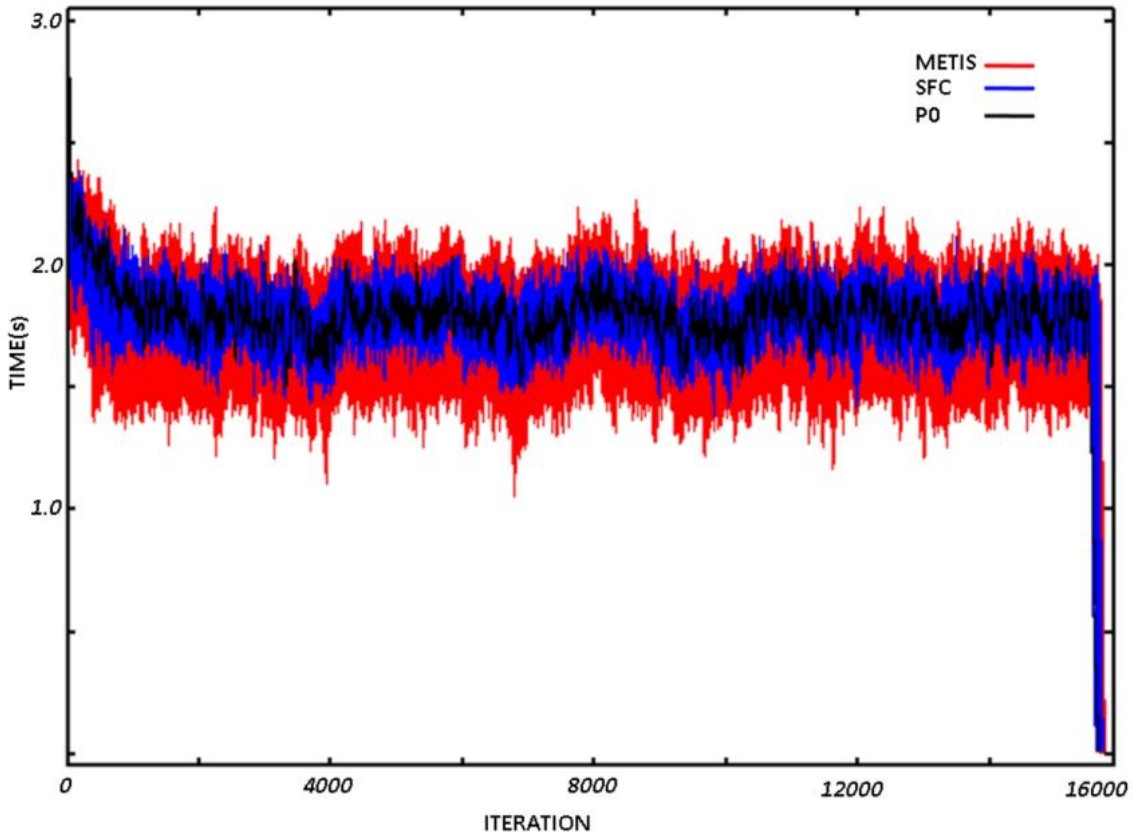


Fig. 5. Time per processor vs. Iteration. 16 processors and 15000 iterations

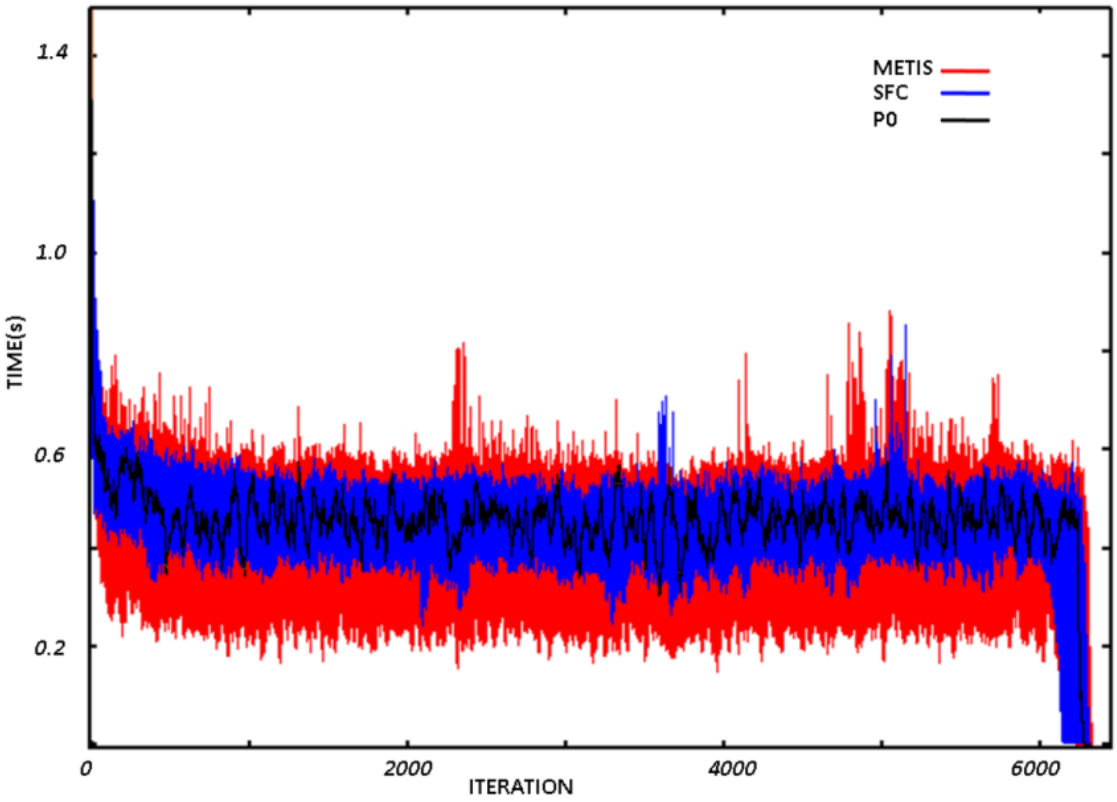


Fig. 6. Time per processor vs. Iteration, 64 processors and 6500 iterations.

4.4 References

[1] George Karypis, *METIS - Family of Multilevel Partitioning Algorithms*, http://glaros.dtc.umn.edu/gkhome/ 2006-2007,

[2] M. Dumbser, M. Käser, E. Toro, *An Arbitrary High Order Discontinuous Galerkin Method for Elastic Waves on Unstructured Meshes V: Local Time Stepping and p-Adaptivity*, 2000

[3] C. Hamilton. *Compact Hilbert Indices*. Dalhousie University Technical Report CS-2006-07, July 2006.