Efficient matrix-free methods in deal.II

Martin Kronbichler

Joint work with Katharina Kormann

Technische Universität München

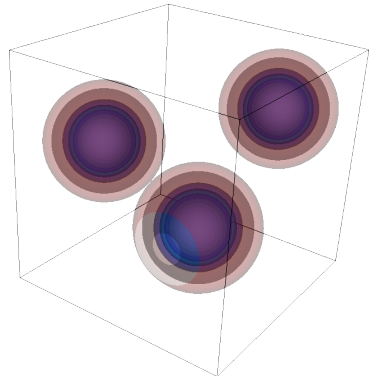
August 20, 2013

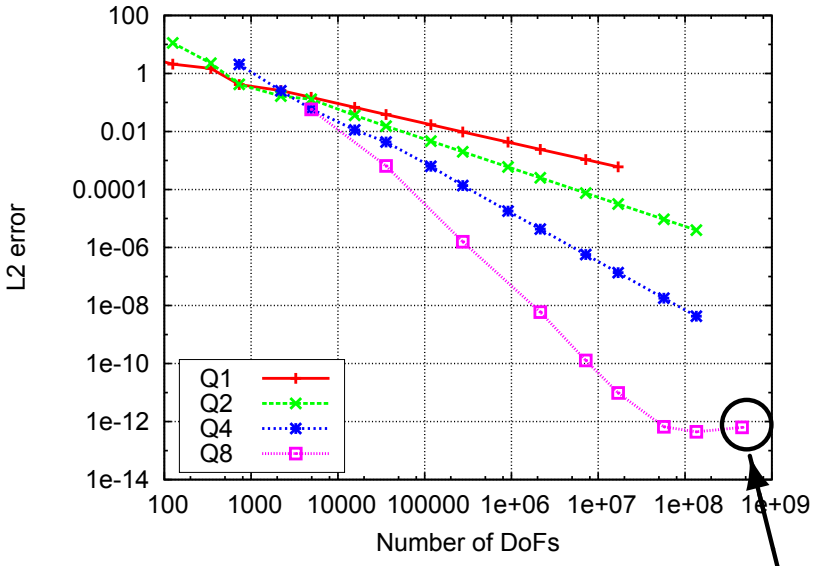
Outline

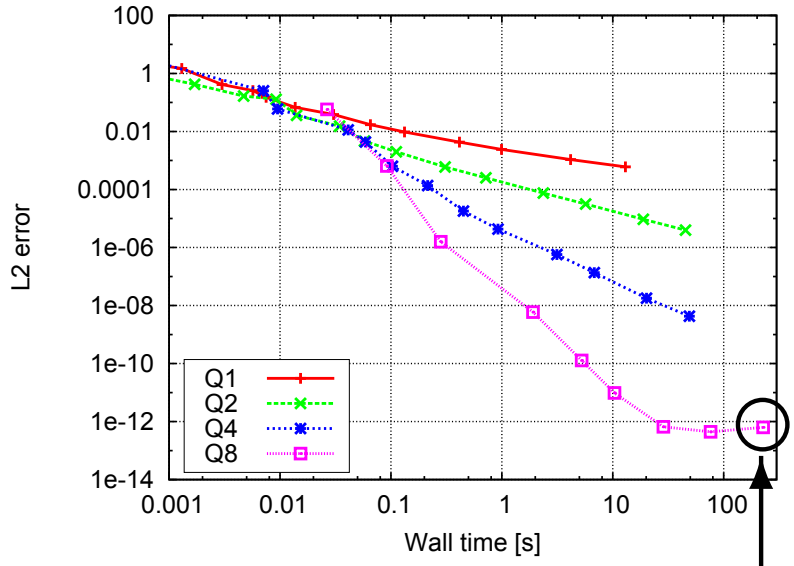
Outline

Performance of matrix-free methods

Problem solved: 3D Poisson equation $(\nabla v, \nabla u) = (v, f)$, uniform grid, CG + geometric multigrid





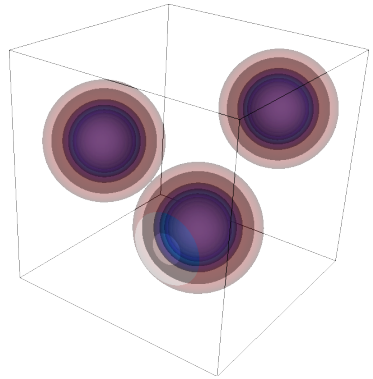


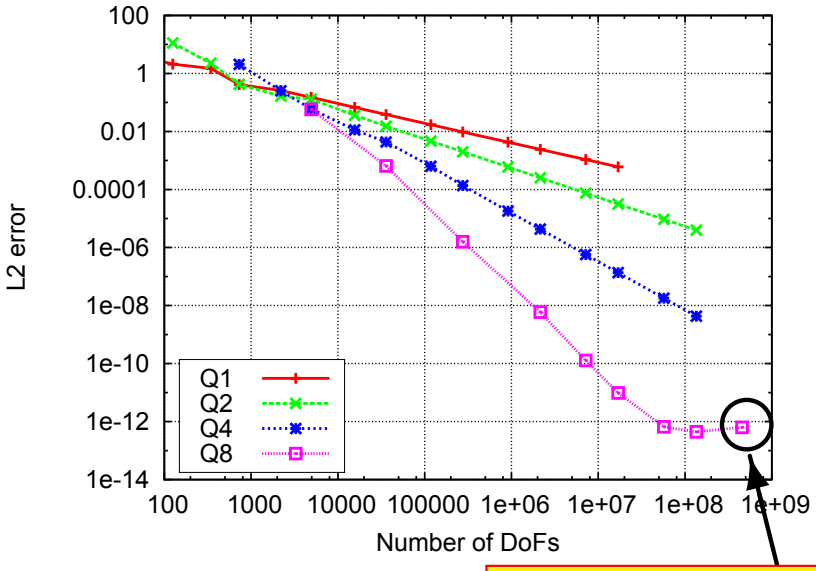
$\mathcal{Q}_8$ elements, 455 m DoFs, solver time: 226.7 s

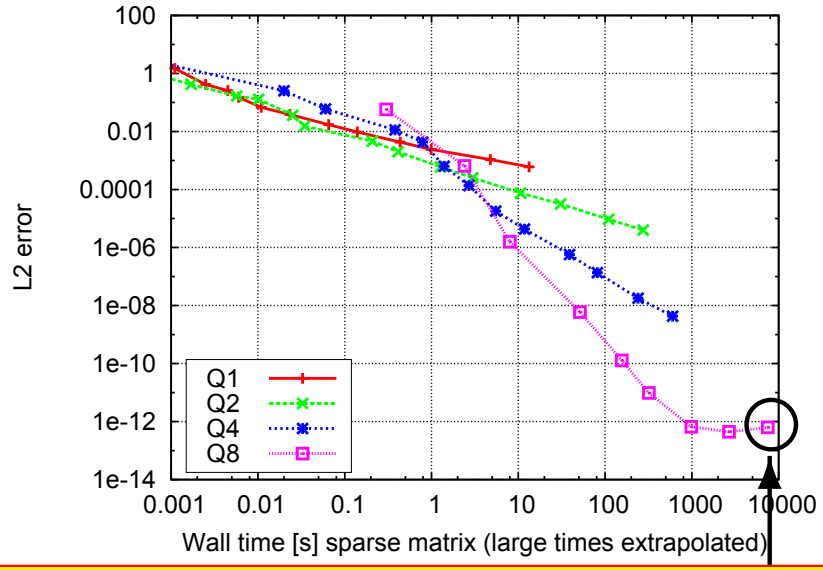
How many processors?

Performance of matrix-free methods

Problem solved: 3D Poisson equation $(\nabla v, \nabla u) = (v, f)$, uniform grid, CG + geometric multigrid







$\mathcal{Q}_8$ elements, 455 m DoFs, solver time: 226.7 s

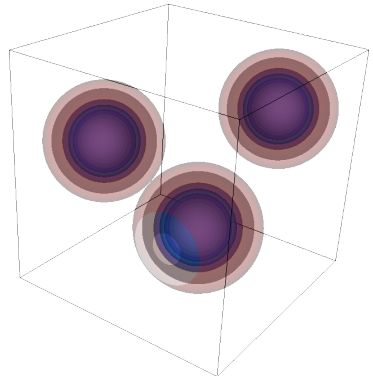
How many processors?

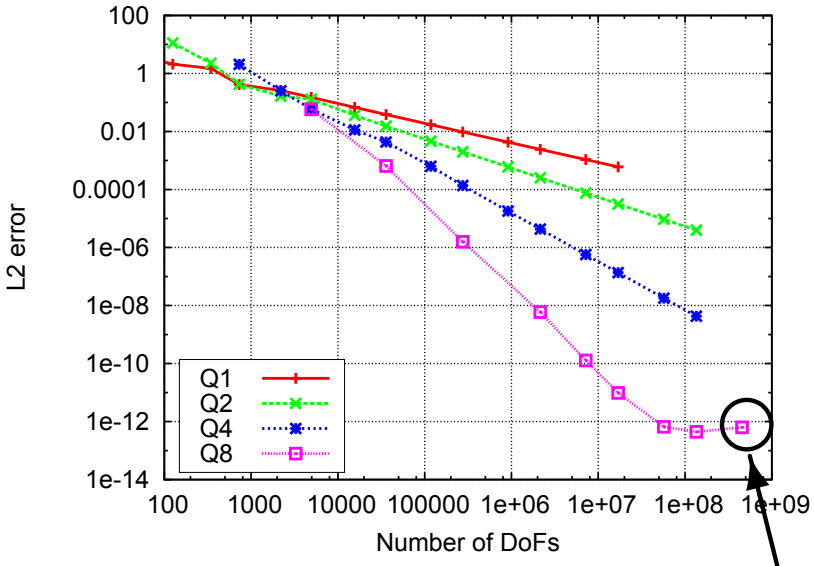
Sparse matrix (`step-16` without adaptivity):

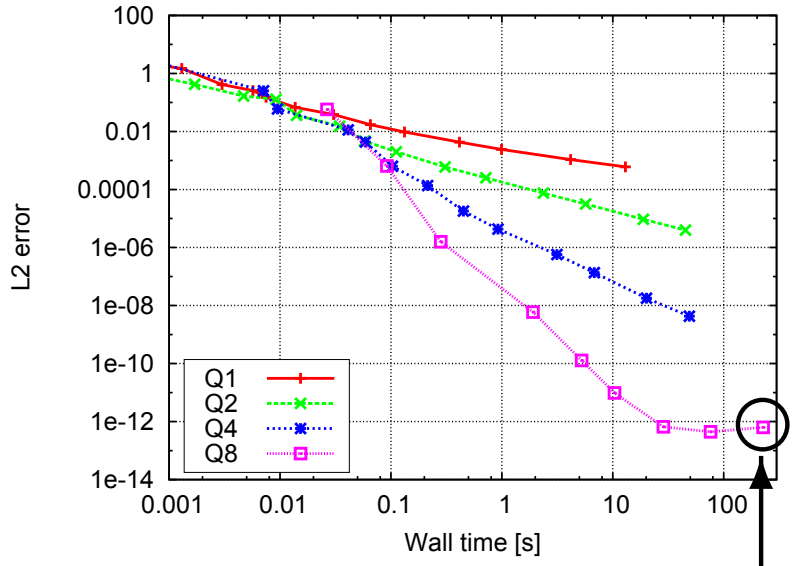
$\sim$ 240 s on 512 cores, 12 GB memory per core

Performance of matrix-free methods

Problem solved: 3D Poisson equation $(\nabla v, \nabla u) = (v, f)$, uniform grid, CG + geometric multigrid





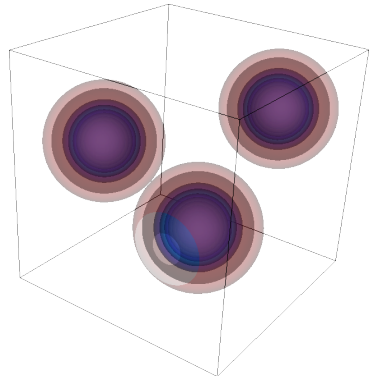


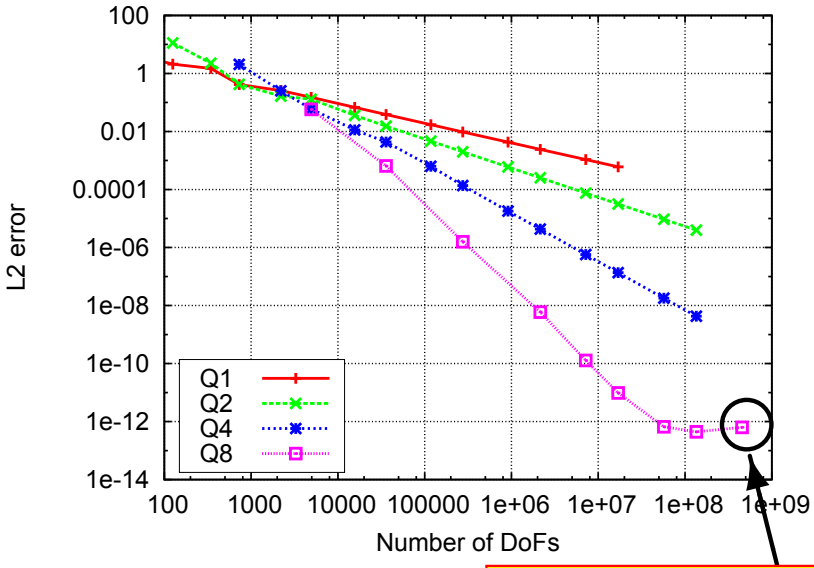
$\mathcal{Q}_8$ elements, 455 m DoFs, solver time: 226.7 s

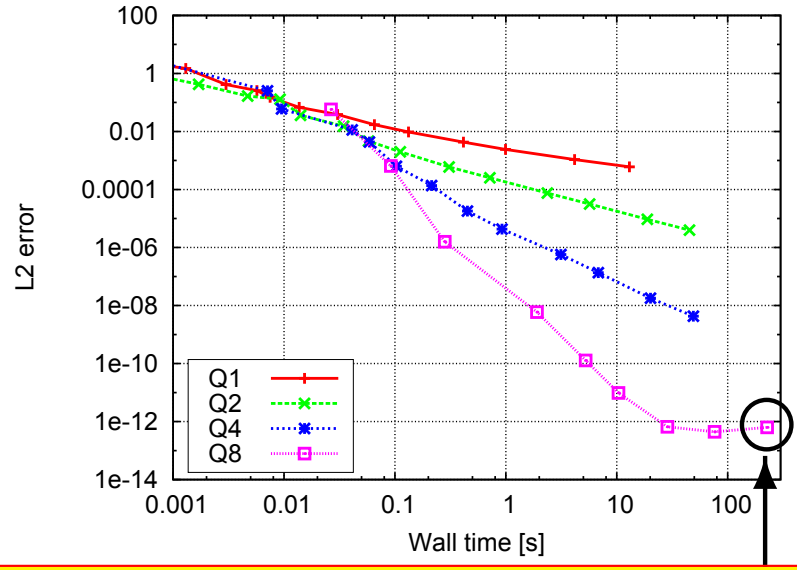
How many processors?

Performance of matrix-free methods

Problem solved: 3D Poisson equation $(\nabla v, \nabla u) = (v, f)$, uniform grid, CG + geometric multigrid







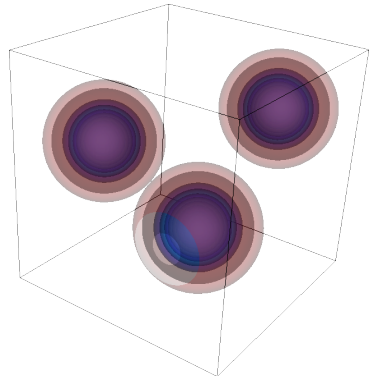
$\mathcal{Q}_8$ elements, 455 m DoFs, solver time: 226.7 s

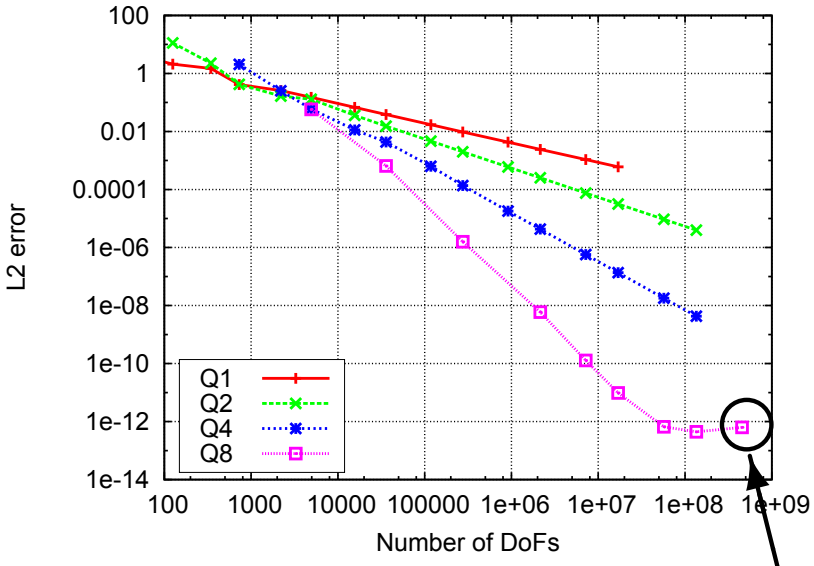
How many processors?

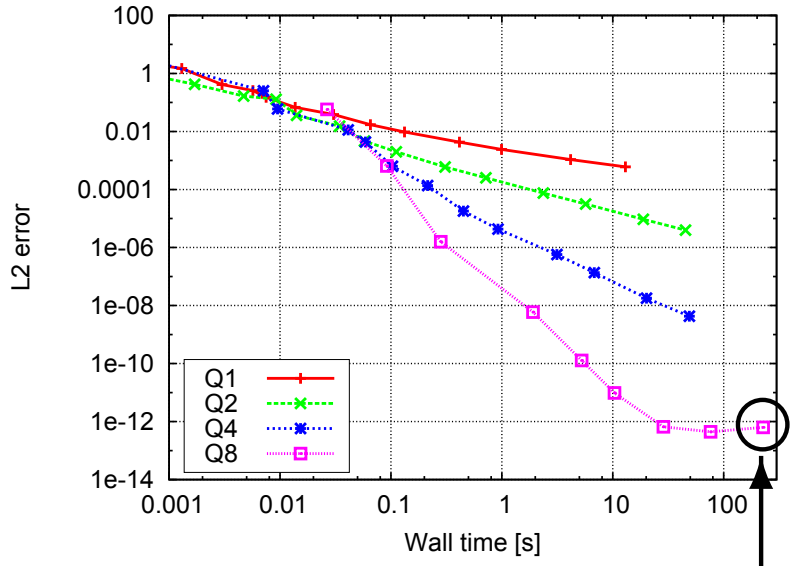
Desktop machine as of 2013, 6 cores of Sandy-Bridge EP @ 3.2 GHz, uses 61 GB RAM

Performance of matrix-free methods

Problem solved: 3D Poisson equation $(\nabla v, \nabla u) = (v, f)$, uniform grid, CG + geometric multigrid



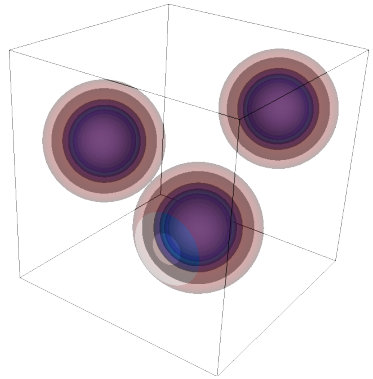


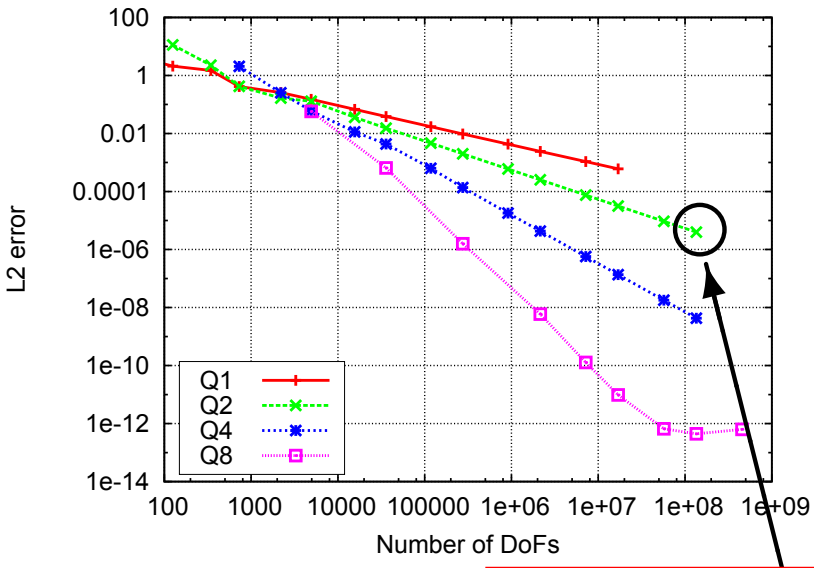


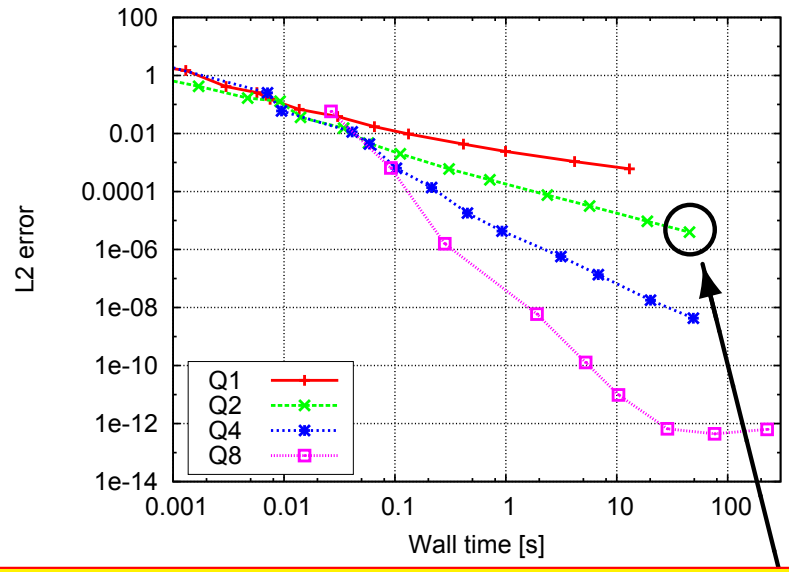
$\mathcal{Q}_8$ elements, 455 m DoFs, solver time: 226.7 s
Run-time improvement matrix-free: 63×
Memory reduction: 95×

Performance of matrix-free methods

Problem solved: 3D Poisson equation $(\nabla v, \nabla u) = (v, f)$, uniform grid, CG + geometric multigrid







$\mathcal{Q}_2$ elements, 135 m DoFs
Matrix-free time 6 cores: 35.3 s, 35 GB memory
Sparse matrix 64 cores: 27 s, 160 GB memory
Improvements: time 7×, memory 3.5×

Issues with classical algorithms based on sparse matrices

- Typical finite element programs spend between 70 and 95% of time in **iterative methods (linear solvers)**, especially for more complicated problems (Stokes, Navier–Stokes)
- Linear solvers spend ∼95% of time on sparse matrix-vector products (SpMV) (or sparse matrix substitutions)

[1] S. Williams et al. (2007). Optimization of sparse matrix-vector multiplication on emerging multicore platforms. Proc. SC2007

[2] Tuned kernels: http://bebop.cs.berkeley.edu/oski

Issues with classical algorithms based on sparse matrices

- Typical finite element programs spend between 70 and 95% of time in **iterative methods (linear solvers)**, especially for more complicated problems (Stokes, Navier–Stokes)
- Linear solvers spend ∼95% of time on sparse matrix-vector products (SpMV) (or sparse matrix substitutions)
- However, SpMV perform poorly on modern computers (memory bandwidth limited, not computation limited)
- Tuning has been tried, but does not get very far (maybe 10-30% improvement)[1] [2]
- Computer architecture: memory bandwidth has increased more slowly than pure arithmetic throughput and will continue to do so in the future

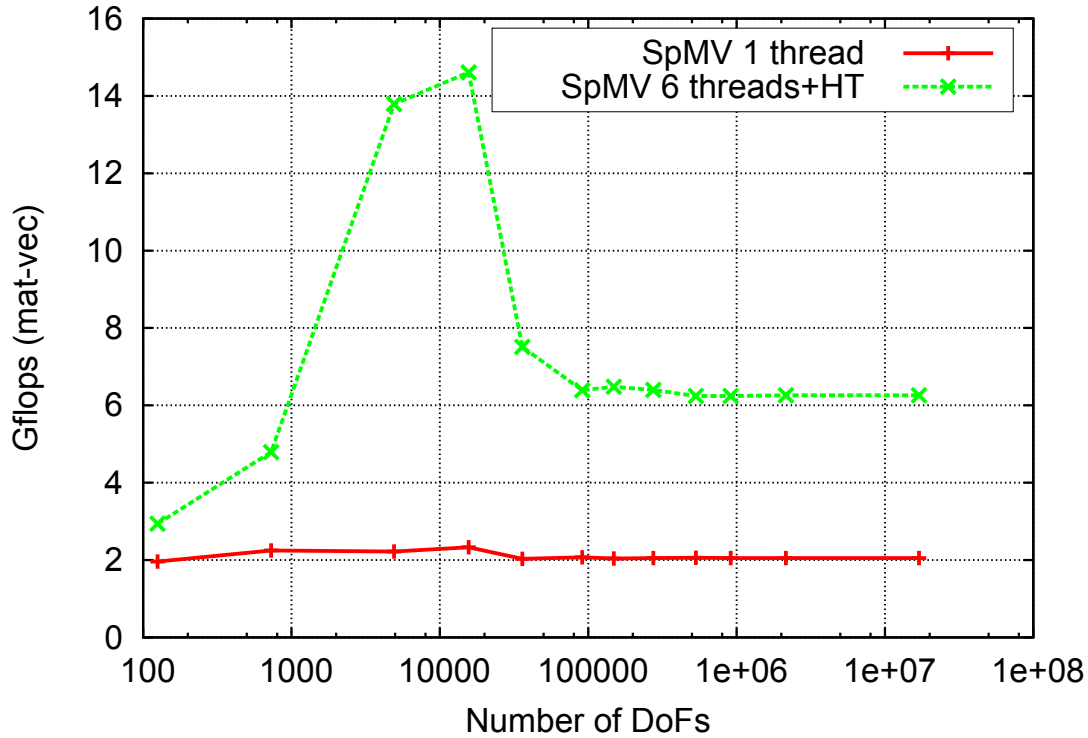
---

[1] S. Williams et al. (2007). Optimization of sparse matrix-vector multiplication on emerging multicore platforms. Proc. SC2007

[2] Tuned kernels: http://bebop.cs.berkeley.edu/oski

Performance of SpMV

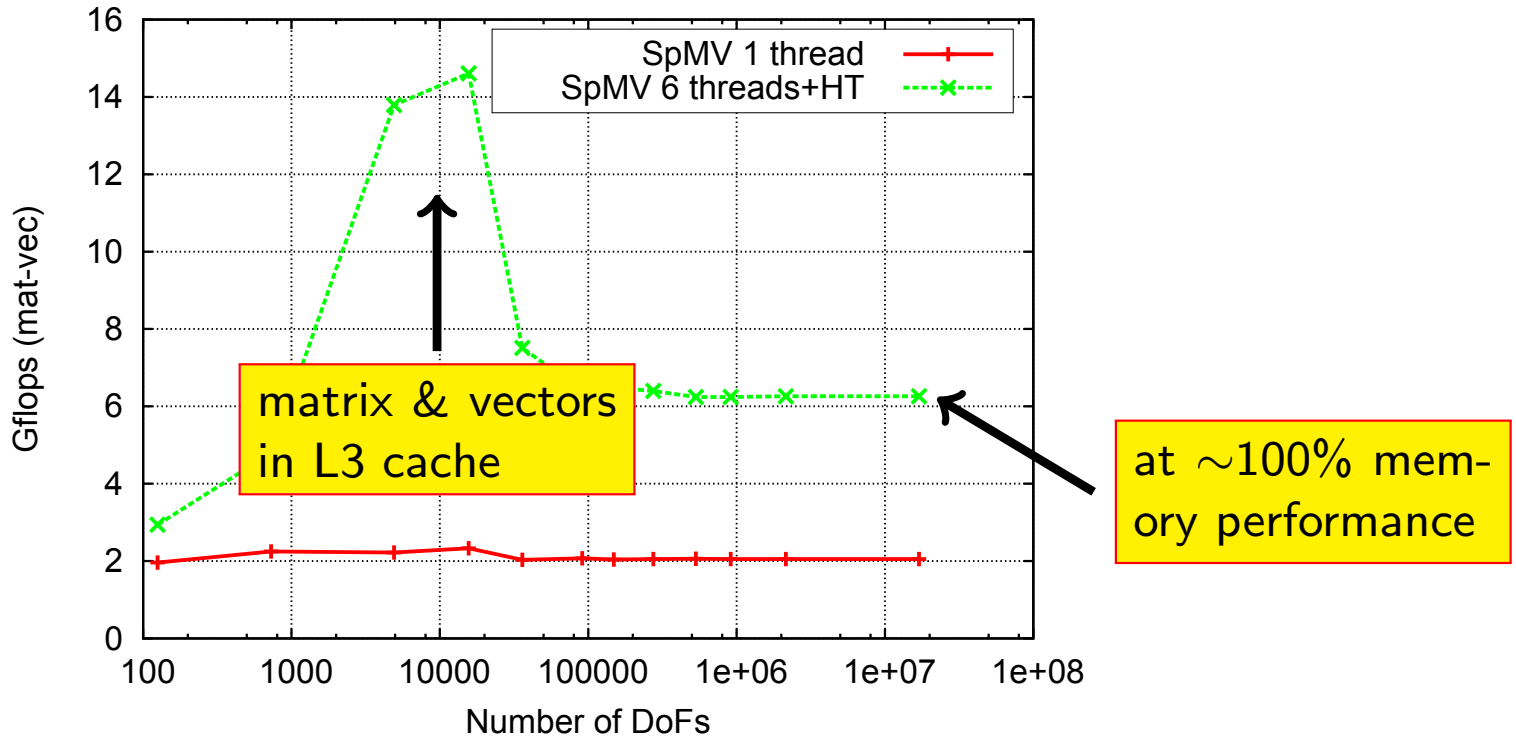
Number of billion arithmetic operations per second (Gflops) of sparse-matrix vector products depending on problem size, 3D Laplacian, $\mathcal{Q}_2$ elements, Sandy Bridge EP, 3.2 GHz



Peak 1 core: 28 Gflops
Peak 6 cores: 144 Gflops
Sustained memory throughput: 35 GB/s

Performance of SpMV

Number of billion arithmetic operations per second (Gflops) of sparse-matrix vector products depending on problem size, 3D Laplacian, $\mathcal{Q}_2$ elements, Sandy Bridge EP, 3.2 GHz



Peak 1 core: 28 Gflops
Peak 6 cores: 144 Gflops
Sustained memory throughput: 35 GB/s

Motivation for matrix-free approach

Less memory requirements for matrix representation $\rightarrow$ faster matrix-vector products (even when doing more arithmetic operations)[3]

[3] M. Kronbichler, K. Kormann: A generic interface for parallel finite operator application. *Comput. Fluids* 63:135–147 (2012)

Outline

Matrix-vector products without creating a global matrix

$$v = Au = \left(\sum_{K \in \{\text{cells}\}} C^T P_K^T A_K P_K C \right) u$$

Matrix-vector products without creating a global matrix

$$v = Au = \sum_{K \in \{\text{cells}\}} C^T P_K^T A_K \left(P_K C u\right)$$

Basic algorithm:

- $v \leftarrow 0$
- loop over cells
  - (i) Extract local vector values on cell, resolve constraints: $u_K = P_K C u$
  - (ii) Apply operation locally on cell: $v_K = A_K u_K$
  - (iii) Sum results from (ii) into the global solution vector, apply constraints: $v \leftarrow v + C^T P_K^T v_K$

Cell operation $v_K = A_K u_K$ for variable-coefficient Laplacian $(\nabla v, a \nabla u)$

Compute cell contribution to matrix-vector product:

$$(A_K u_K)_j = \int_K \nabla \phi_j a(\nabla u^h) d\mathbf{x} \approx \sum_q w_q |\det J_q| \left[\nabla \phi_j a \nabla u\right]_{\mathbf{x}=\mathbf{x}_q}$$

(a) Compute gradient on cell for all quadrature points.

(b) On each quadrature point:

- Multiply each component of the gradient $\nabla u^h(\mathbf{x}_q)$ by $a(\mathbf{x}_q) w_q |\det J(\hat{\mathbf{x}}_q)|$ (coefficient, quadrature weight, Jacobian determinant).

(c) Test by gradient of basis functions and sum over all quadrature points.

Cell operation $v_K = A_K u_K$ for variable-coefficient Laplacian $(\nabla v, a\nabla u)$: deal.II vector assembly

```
// ...
typename DoFHandler<dim>::active_cell_iterator
  cell = dof_handler.begin_active(),
  endc = dof_handler.end();
for (; cell!=endc; ++cell)
  {
    local_dst = 0;
    fe_values.reinit (cell);
    coefficient.value_list(fe_values.get_quadrature_points(),
                           coefficient_values);

    fe_values.get_function_gradients (src, src_gradients);

    for (unsigned int q=0; q<n_q_points; ++q)
      for(unsigned int i=0; i<dofs_per_cell; ++i)
        local_dst(i) += (fe_values.shape_grad(i,q) *
                         coefficient_values[q] *
                         fe_values.JxW(q) *
                         src_gradients[q]);

    cell->get_dof_indices(local_dof_indices);
    constraints.distribute_local_to_global (local_dst, local_dof_indices,
                                            dst);
  }
```

Cell operation $v_K = A_K u_K$ for variable-coefficient Laplacian $(\nabla v, a\nabla u)$: more efficient to split gradient and geometry

Compute cell contribution to matrix-vector product:

$$(A_K u_K)_j = \int_K \nabla\phi_j a(\nabla u^h) d\mathbf{x} \approx \sum_q w_q |\det J_q| \left[\nabla\phi_j a \nabla u\right]_{\mathbf{x}=\mathbf{x}_q}$$

(a) Compute gradient on unit cell for all quadrature points.

(b) On each quadrature point:

- Apply Jacobian transformation $J^{-T}(\hat{\mathbf{x}}_q)$
- Multiply each component of the gradient $\nabla u^h(\mathbf{x}_q)$ by $a(\mathbf{x}_q) w_q |\det J(\hat{\mathbf{x}}_q)|$ (coefficient, quadrature weight, Jacobian determinant).
- Apply Jacobian transformation $J^{-1}(\hat{\mathbf{x}}_q)$

(c) Test by unit cell gradient of basis functions and sum over all quadrature points.

Outline

Efficient element kernels I: Evaluation of unit cell gradient

- Form of basis functions: $\phi(x, y) = \varphi(x)\varphi(y)$
- Evaluate unit cell derivative:

$$\frac{\partial u(x_q, y_q)}{\partial \hat{x}_1} = \sum_{i \in \text{cell\_dofs}} u^{(i)} \frac{\partial \phi_i(x_q, y_q)}{\partial \hat{x}_1} = \sum_{i_x} \sum_{i_y} u^{(i_x, i_y)} \varphi_{i_y}(y_q) \frac{\partial \varphi_{i_x}(x_q)}{\partial \hat{x}_1}$$

- Set basis functions evaluated at all quadrature points in one dimension into matrices

$$A = \begin{pmatrix} \varphi_1(x_1) & \varphi_1(x_2) & \cdots \\ \varphi_2(x_1) & \varphi_2(x_2) & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}, \quad B = \begin{pmatrix} \varphi_1'(x_1) & \varphi_1'(x_2) & \cdots \\ \varphi_2'(x_1) & \varphi_2'(x_2) & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

Efficient element kernels I: Tensor product structure in gradient evaluation (sum factorization)

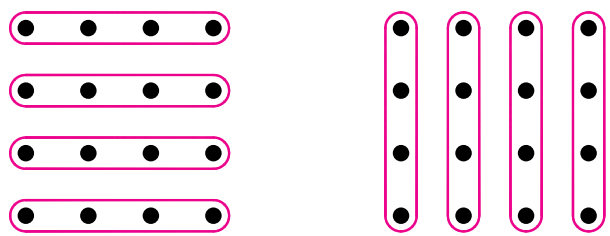
Evaluation of unit cell derivative $\frac{\partial u(x_q,y_q)}{\partial \hat{x}_1}$ on all quadrature points corresponds to the matrix-vector product

$$\frac{\partial u(x_q, y_q)}{\partial \hat{x}_1}\Big|_{\text{q\_points}} = (A \otimes B)\mathbf{u}_K,$$

where $\mathbf{u}_K$ collects the node values on cell K. Reshape this as matrix-matrix products to reduce complexity from $\mathcal{O}((p+1)^{2d})$ to $\mathcal{O}(d(p+1)^{d+1})$

$$(A \otimes B)\mathbf{u}_K = BU_K A$$

Illustration on nodes (successively apply 1D operators):



$\text{tmp} = B \cdot U_k$ $\qquad$ $\text{tmp} \cdot A$

Efficient element kernels II: Make loop bounds known to compiler

- Value and gradient evaluation involves many short loops of length $p+1$
- For low element degree p, this involves high loop overhead
- Introduce element degree (and number of quadrature points) as compile-time constant in `FEEvaluation` $\rightarrow$ speedup $3\times$ at low order and $1.5\times$ at high order

Implementation

`MatrixFree` class stores indices and mapping data (evaluated geometry is cached), `FEEvaluation` implements element kernels similar to `FEValues`.

Code example for Laplacian $(\nabla v, a\nabla u)$:

```
template <int dim, int fe_degree>
void local_operation (const MatrixFree<dim>          &matrix_free,
                      Vector<double>                 &dst,
                      const Vector<double>           &src,
                      const std::pair<unsigned int,unsigned int> &cell_range)
{
  FEEvaluation<dim,fe_degree> phi(matrix_free);
  for (unsigned int cell=cell_range.first; cell<cell_range.second; ++cell)
    {
      phi.reinit(cell);
      phi.read_dof_values (src);
      phi.evaluate (false, true);
      for (unsigned int q=0; q<phi.n_q_points; ++q)
        phi.submit_gradient (coefficient_values(cell, q) *
                             phi.get_gradient(q), q);
      phi.integrate (false, true);
      phi.distribute_local_to_global(dst);
    }
}
```

MPI parallelization

Matrix-free algorithm including MPI:

- $v \leftarrow 0$
- `Update ghost values`: Import from other MPI processes.
- loop over cells
  - (i) Extract local vector values on cell, resolve constraints: $u_K = P_K C u$
  - (ii) Apply operation locally on cell: $v_K = A_K u_K$
  - (iii) Sum results from (ii) into the global solution vector, apply constraints: $v \leftarrow v + C^T P_K^T v_K$
- `compress`: Exchange of information along processor boundaries.

Use specially adapted vector type `parallel::distributed::Vector` that allows direct array access in MPI-local index space (otherwise: performance penalty by factor $2-3$).

Thread parallelization

- Recall loop over all cells:
  - (i) Extract local vector values on cell: $u_K = P_K C u$
  - (ii) Apply operation locally on cell: $v_K = A_K u_k$
  - (iii) Sum results from (ii) into the global solution vector: $v \leftarrow v + C^T P_K^T v_K$
- Tasks (i) and (ii) independent between cells, but final write operation (iii) must be synchronized between neighbors

[4] K Kormann, M. Kronbichler: Parallel finite element operator application: Graph partitioning and coloring. *Proceedings of the 7th IEEE International Conference on e-Science*, 2011

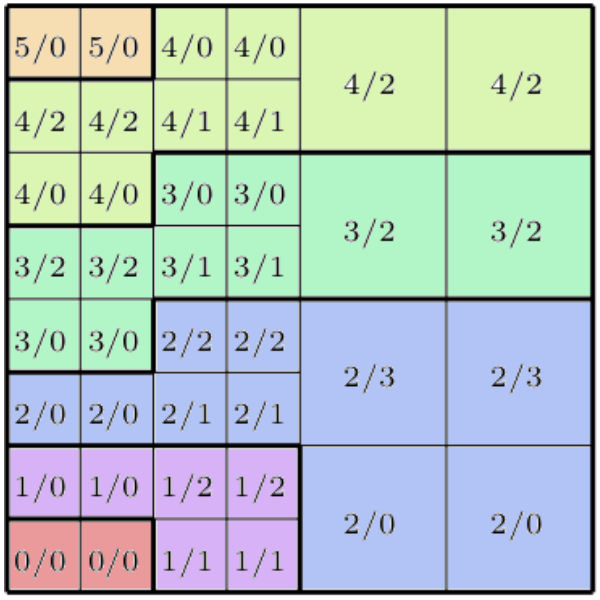
Thread parallelization

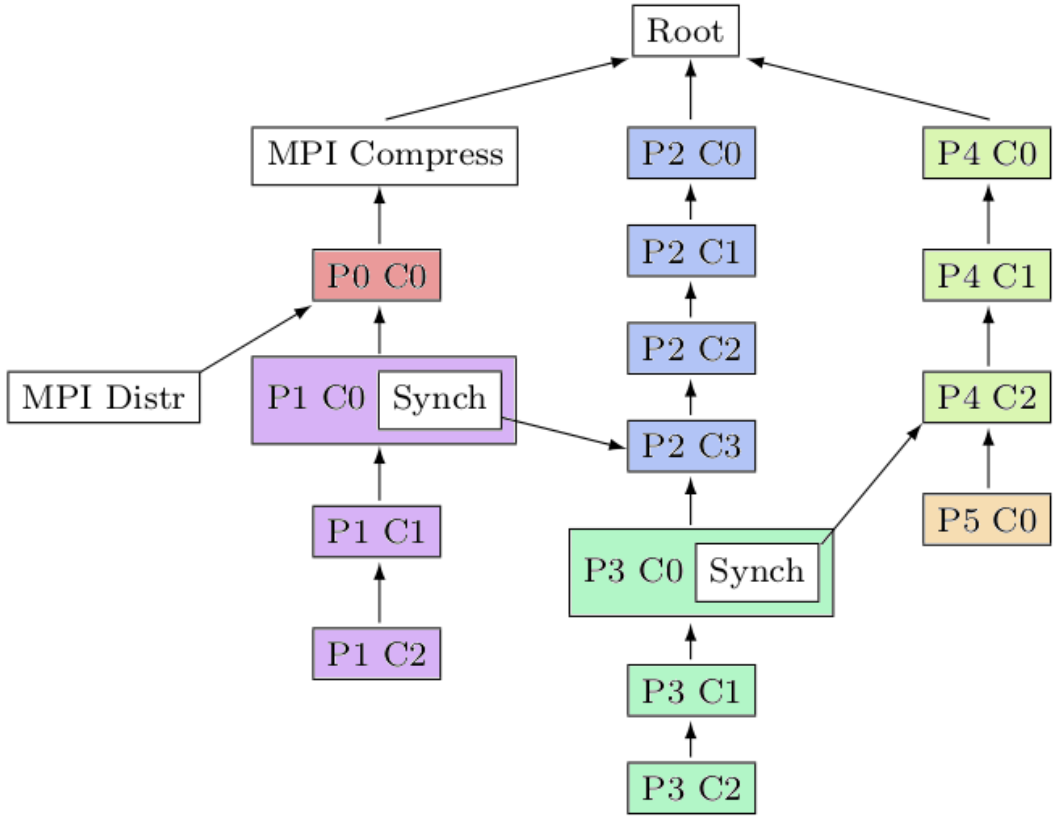
- Recall loop over all cells:
  - (i) Extract local vector values on cell: $u_K = P_K C u$
  - (ii) Apply operation locally on cell: $v_K = A_K u_k$
  - (iii) Sum results from (ii) into the global solution vector: $v \leftarrow v + C^T P_K^T v_K$
- Tasks (i) and (ii) independent between cells, but final write operation (iii) must be synchronized between neighbors
- To avoid serializing writes or private solution vectors (MPI-type approach), work on non-overlapping chunks of cells by a combination of partitioning and coloring[4]
- Coloring: Cells are assigned colors – cells with same color are not adjacent. Different colors are worked on at different times.
- Partitioning: Cells are subdivided into partitions such that P_k is only adjacent to P_{k-1} and P_{k+1} (Cuthill-McKee type of partitions).

[4] K Kormann, M. Kronbichler: Parallel finite element operator application: Graph partitioning and coloring. *Proceedings of the 7th IEEE International Conference on e-Science*, 2011

Partitioning and coloring: Illustration

Combining partitions on outer level with coloring within the partitions gives good cache performance, enough chunks of cells to keep all threads busy





Dynamic task scheduling based on this graph done by Intel's Threading Building Blocks

Vectorization

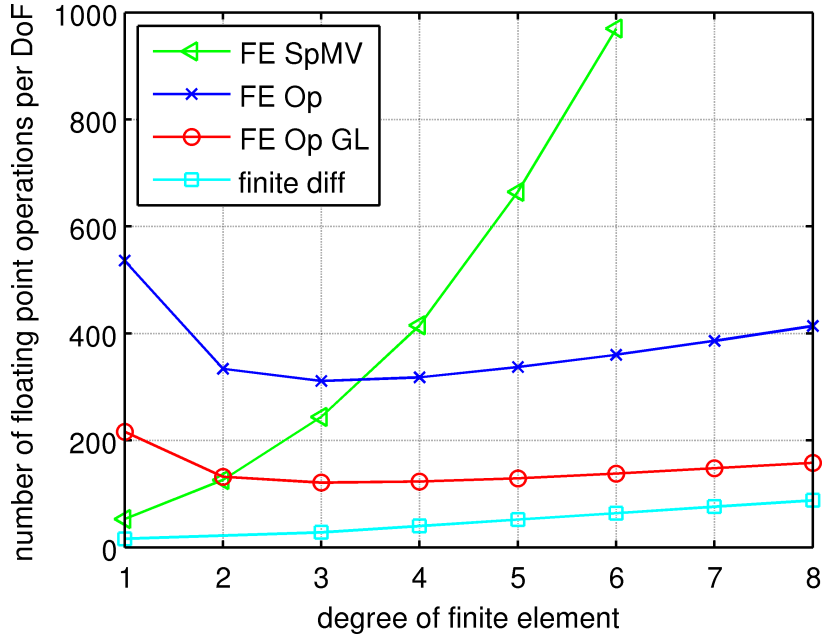
- Modern processors support arithmetic operations on several data items (e.g. SSE, AVX). AVX can do operations on 4 double precision variables with one instruction
- However, data must be laid out contiguously in memory for this to be efficient $\rightarrow$ typical array-of-structs approach that for each cell stores local struct-like data is inefficient
- For low to medium orders, it is most efficient to place data from several cells together in small arrays at the innermost level (array-of-structs-of-arrays data layout)
- Vectorization with AVX speeds up computations by a factor 3.5–3.7
- Throughput devices (GPUs, Xeon Phi) have even wider vector units, as will have future CPUs

Outline

Computational complexity and memory in 3D

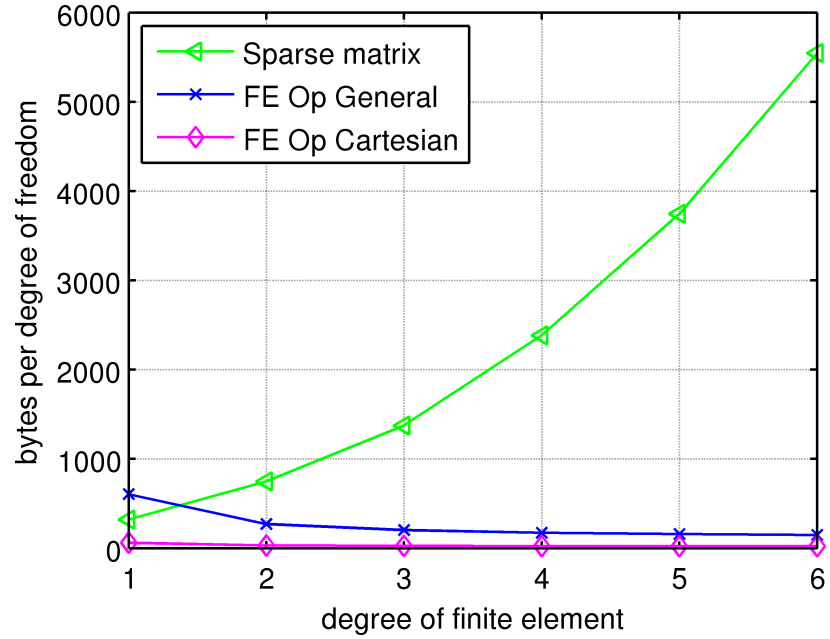
Arithmetic operations

matrix-free approach (fast assembly) vs. sparse matrices



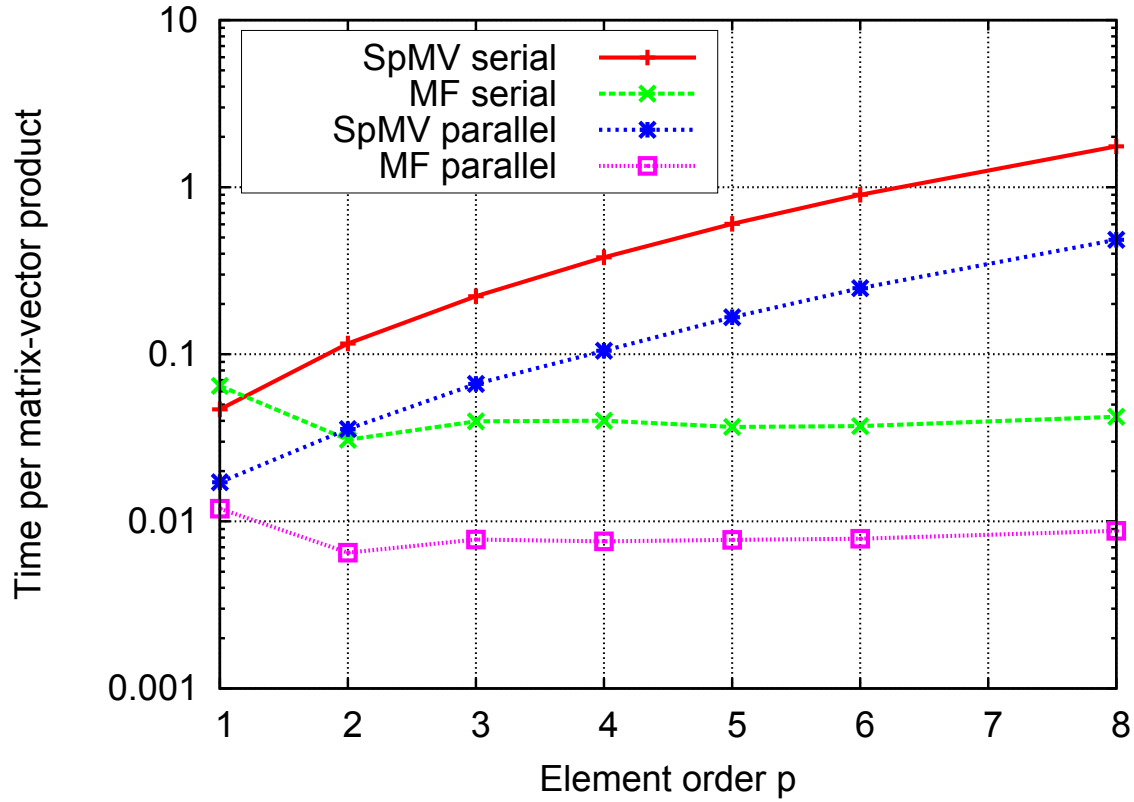
Memory consumption

matrix-free approach vs. sparse matrices



Performance in 3D

1.7 million degrees of freedom, 20 matrix-vector products, standard $\mathcal{Q}_p$ finite elements, degree 1 to 8

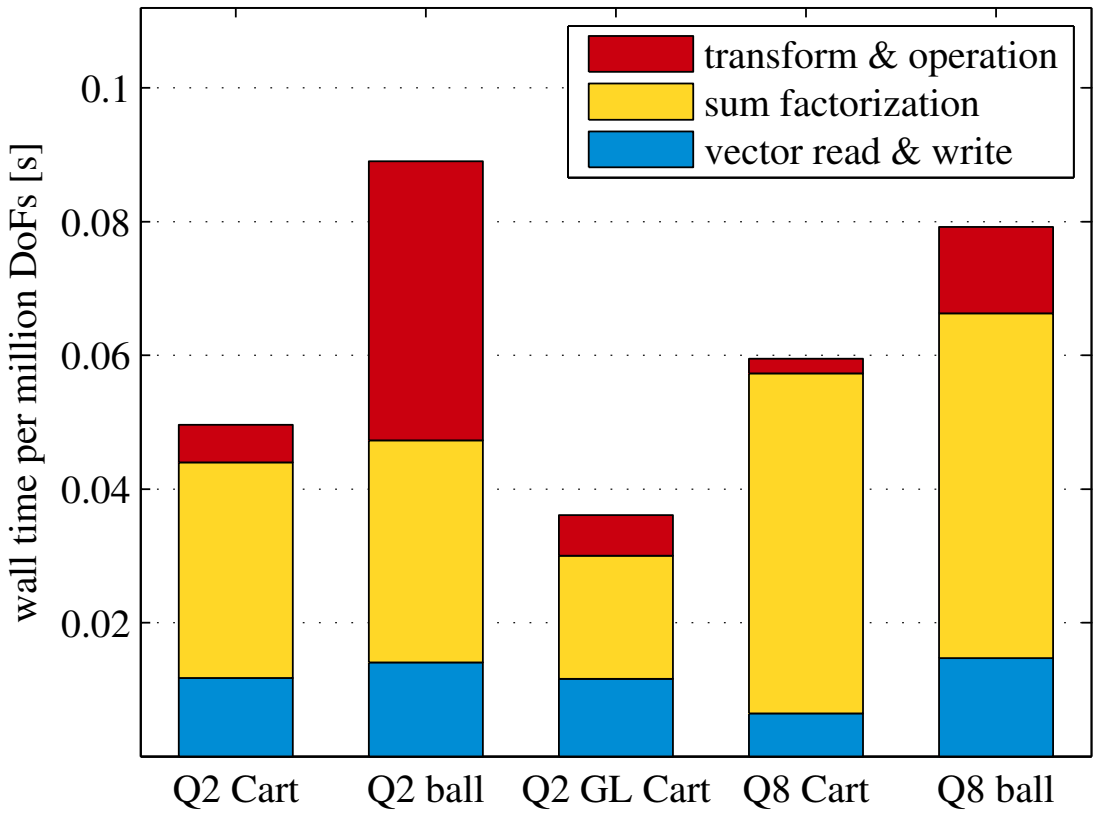


Computation on Sandy Bridge EP (with AVX)

"Interesting" result: except for $p = 1$, time for matrix-free is almost independent of element order at the same number of DoFs

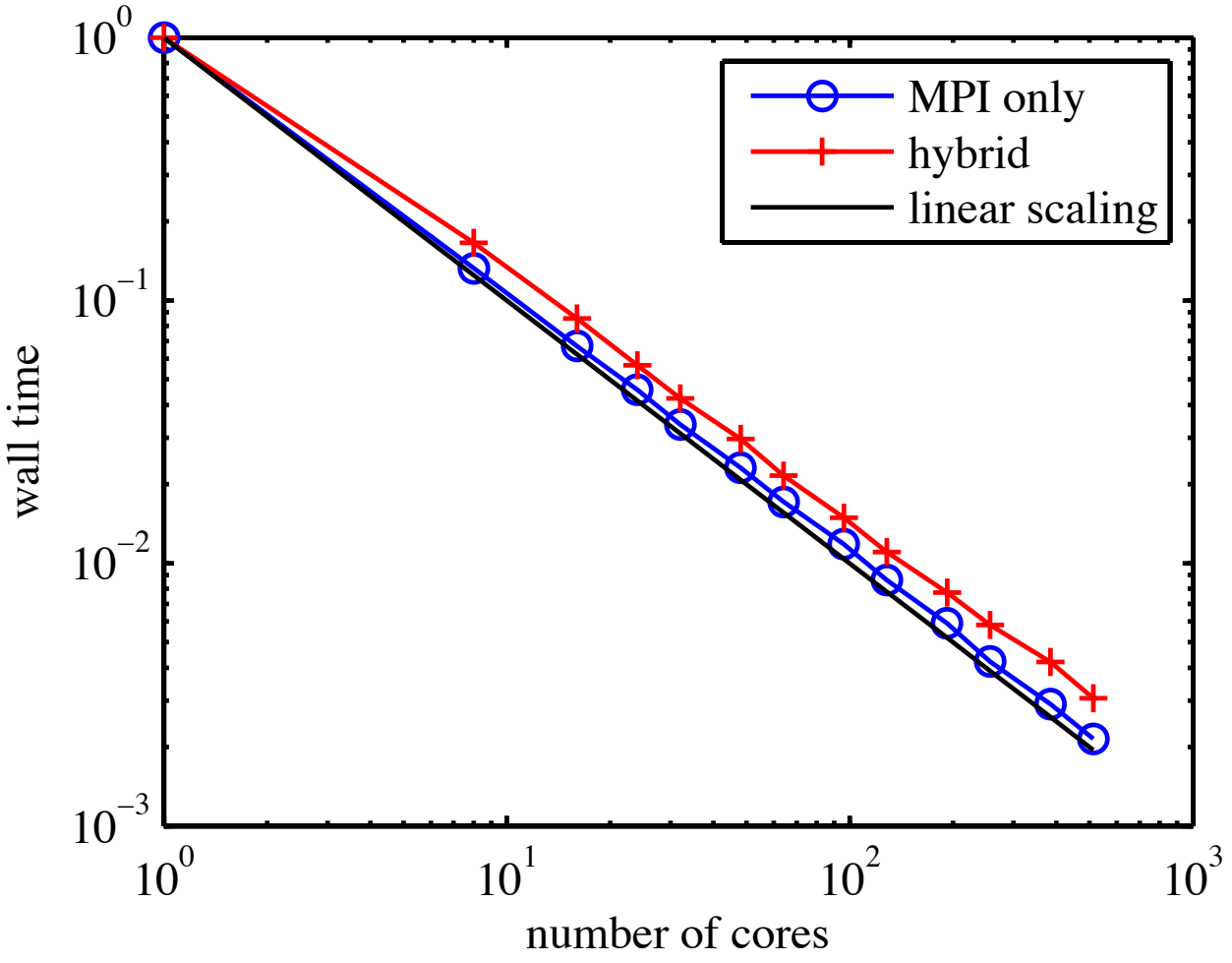
Time spent in different components

3D Laplacian $(\nabla v, \nabla u)$ on Cartesian mesh (structured) and ball mesh (unstructured, adaptive refinement) on Nehalem-EP:



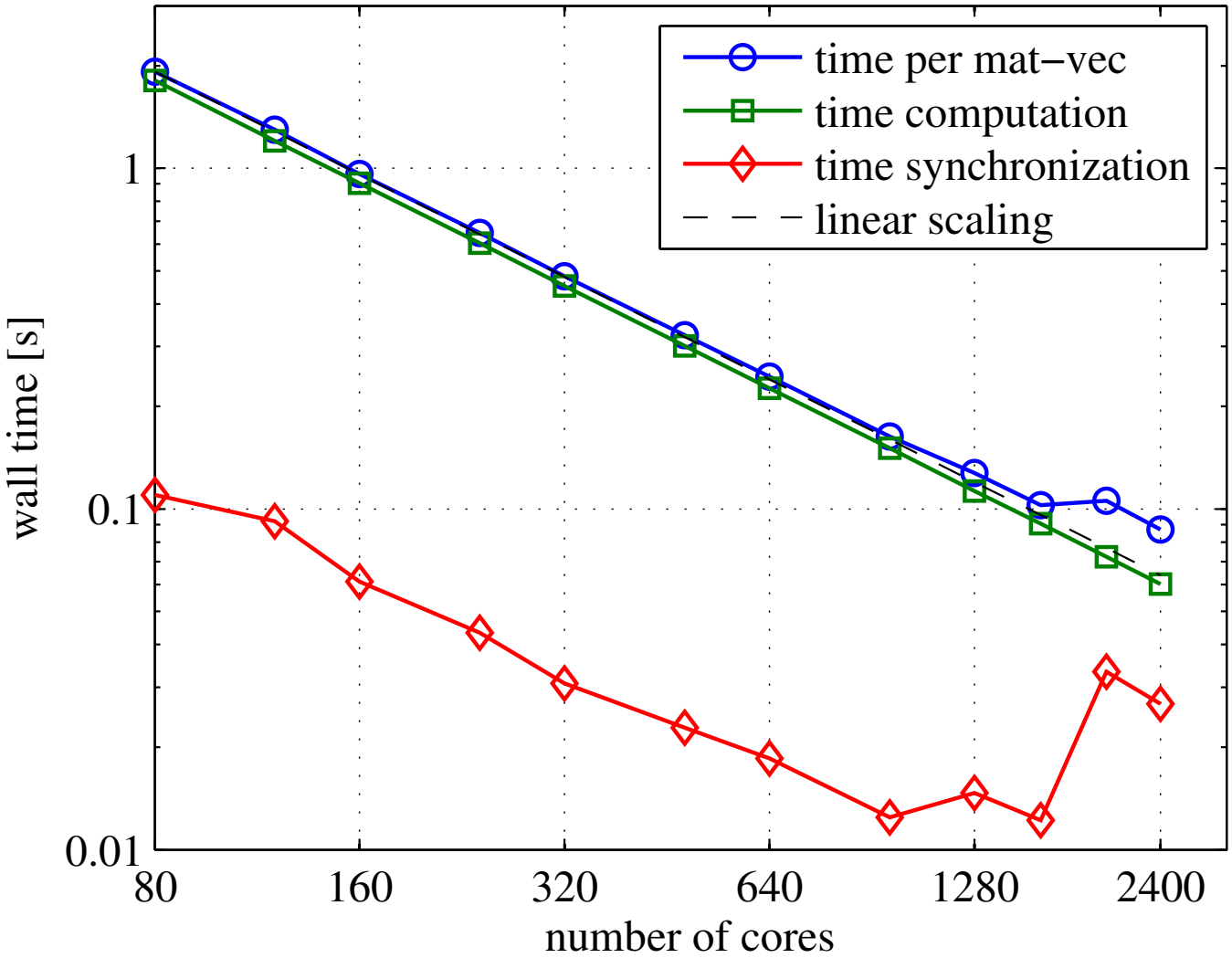
Scaling results

Time for one matrix-vector product with 17 million DoFs ($\mathcal{Q}_2$ elements), 1 to 512 cores, Nehalem-EP cluster



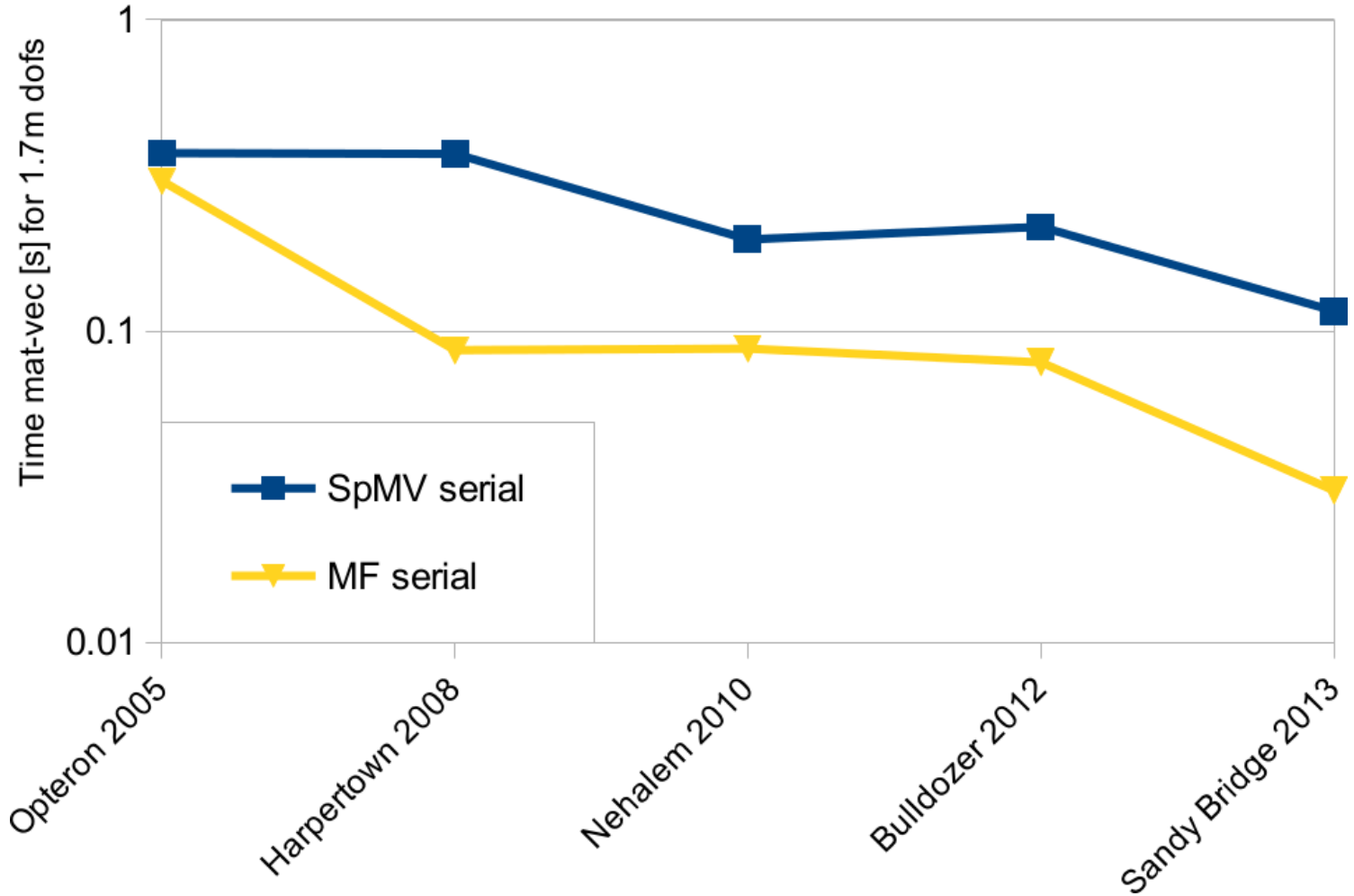
Scaling results

Time for one matrix-vector product with 2.1 billion DoFs ($\mathcal{Q}_2$ elements), 80 to 2400 cores, Nehalem-EP cluster



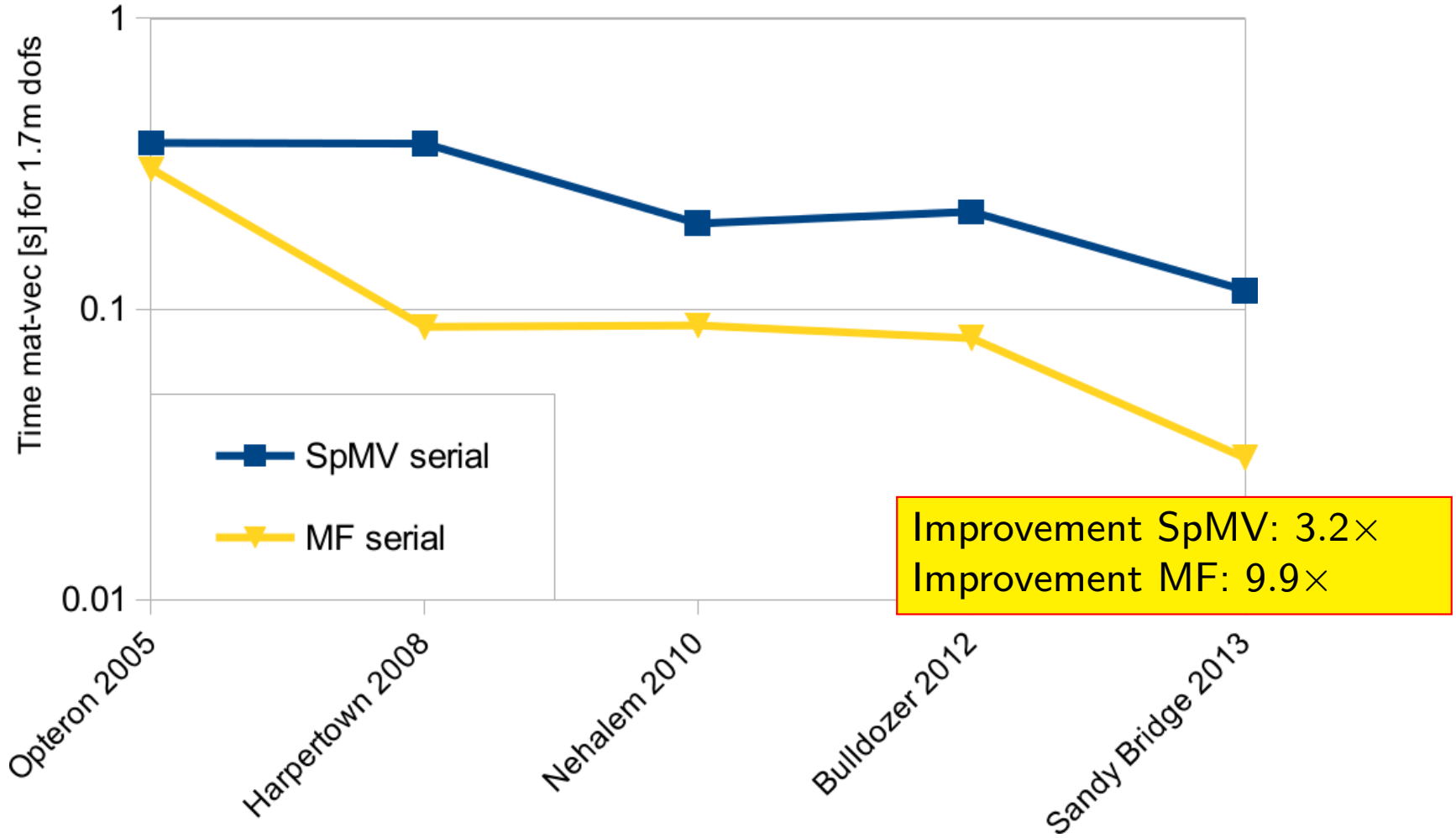
Arithmetic performance increases faster than memory performance: actual SpMV and matrix-free data

$\mathcal{Q}_2$ elements, 1.7m DoFs, Laplacian $(\nabla v, \nabla u)$



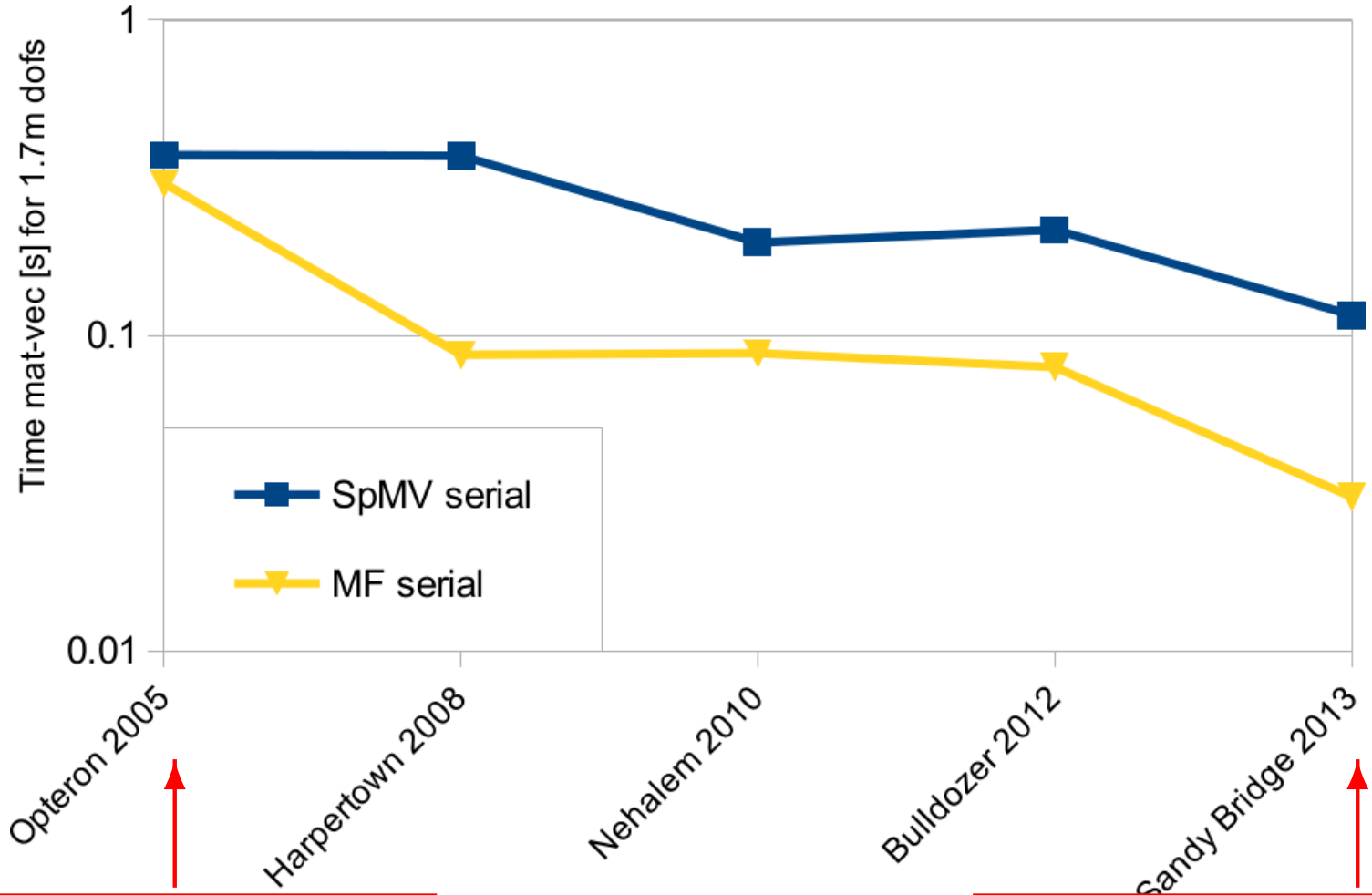
Arithmetic performance increases faster than memory performance: actual SpMV and matrix-free data

$\mathcal{Q}_2$ elements, 1.7m DoFs, Laplacian $(\nabla v, \nabla u)$



Arithmetic performance increases faster than memory performance: actual SpMV and matrix-free data

$\mathcal{Q}_2$ elements, 1.7m DoFs, Laplacian $(\nabla v, \nabla u)$



Parallel efficiency SpMV: 62%
Parallel efficiency MF: 85%

Parallel efficiency SpMV: 52%
Parallel efficiency MF: 79%

Outline

Application 1: Incompressible Navier–Stokes equations

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right) + \nabla\cdot\mu(\nabla\mathbf{u} + \nabla\mathbf{u}^T) - \nabla p = \rho\mathbf{f}$$

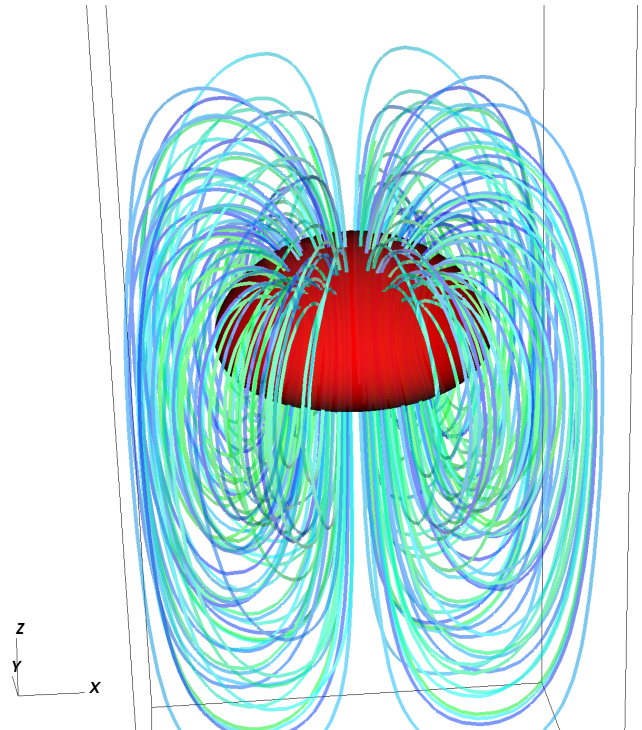
$$\nabla\cdot\mathbf{u} = 0$$

Time for evaluation of linearized Navier–Stokes operator with $\mathcal{Q}_2\mathcal{Q}_1$ elements in 3D in seconds on Nehalem-EP:

	MF	SpMV	Speedup
serial	0.293	1.30	4.4×
8 threads	0.0487	0.486	10×

Simulation of two-phase flow in 2D/3D with *hp* adaptivity, total application speedup in 2D/3D

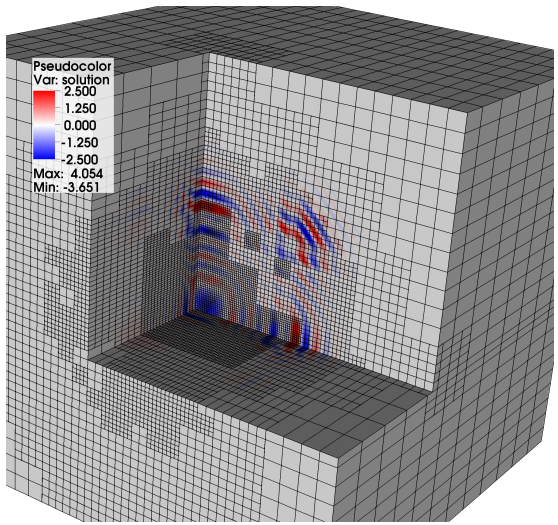
- serial: 2× / 4×
- parallel: 3× / 7×



Application 2: Time-dependent Schrödinger equation for quantum dynamics

$$\mathrm{i}\frac{\partial}{\partial t}\begin{pmatrix}\psi_a\\ \psi_b\end{pmatrix} = \begin{pmatrix}H_a & V_{a,b}\\ V_{a,b}^* & H_b\end{pmatrix}\begin{pmatrix}\psi_a\\ \psi_b\end{pmatrix}$$

$$H_* = -\sum_{i=1}^{d}\frac{\hbar^2}{2m_i}\frac{\partial^2}{\partial x_i^2} + V_*(x)$$

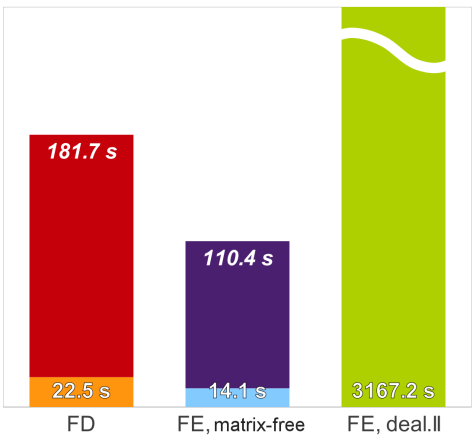


Use high order Gauss–Lobatto elements in space and exponential integrator (Lanczos) in time

Application speedup for $\mathcal{Q}_4$ over SpMV: $4.5\times$

Low memory consumption **allows for** $\mathcal{Q}_6$, $\mathcal{Q}_7$

Implementation is competitive with high order finite differences: $\mathcal{Q}_7$ elements / 8th order FD by M. Gustafsson @ Uppsala University, 11.3 million DoFs (light color) and 89.9 million DoFs (dark color)



Outline

Summary

- Matrix-free implementation is essentially a very fast vector assembly framework that fits both linear operators as well as nonlinear ones (see `step-48`)
- **Significant speedups** over state-of-the-art (sparse matrices) for element order 2 and higher due to reduced memory consumption
- Makes **higher order elements** ($p \geq 3$) much more attractive

Future challenges

The matrix-free implementation re-implements several of deal.II's algorithms in a faster but less general way

- (Parallel) loop over cells is already done in `MeshWorker` which also provides complete support for face integrals (DG) $\rightarrow$ first step would be to introduce task dependency graph concept in other parts of `deal.II`.
- `FEEvaluation` has its own syntax but essentially does a subset of what `FEValues` does. However, we need both vectorization and compile-time information on loop lengths for well-performing algorithms.
- Vectorization will likely become important in other parts of programs in the near future as well.
- Also, `FEEvaluation` can be used to assemble sparse matrices quickly, quicker than any other method I know ($2-3\times$ for $\mathcal{Q}_1$, $4-25\times$ for $\mathcal{Q}_4$). But it would be useful if we could closer collaborate with the mapping data in FEValues instead of caching it in MatrixFree.