

Some Ideas on Hermes Performance Improvement

Jakub Cervený

hp-FEM group, University of Nevada, Reno
Academy of Sciences, Prague, Czech Republic

<http://hpfem.math.unr.edu>

February 2009

1 Introduction

We are currently facing the following performance or usability-related problems in Hermes:

1. Assembling in Hermes takes about 50% of the CPU time in most single-mesh 2D computations (the other 50% is spent on UMFPACK factorization). This is not bad, and in fact it is the least serious problem described here, but we could always do better, especially to compare more favorably with other FEM solvers. Our experiments show that 2D assembling could run about 3 times faster using the improvements described below.
2. The way weak forms are defined now in Hermes is very inflexible. It is very difficult for the users to create their own weak forms. This was not necessary for problems like the Poisson equation or linear elasticity, but, e.g., for nonlinear problems using the Newton's method this is inevitable. Even if the user is capable of writing his/her own integrals, the definition of complicated problems (e.g., Taylor-Galerkin) gets extremely long. Also, it is problematic to define C-style callbacks in Python.
3. Multi-mesh assembling is extremely slow, which defeats its original purpose of saving time by using different meshes for different components of the solution. For small differences between meshes the performance is OK, which makes it a good tool for dynamical meshes. Still, it would be nice to have multi-mesh perform well and be really able to use it for its original purpose, which is nearly impossible now.

In the following we will list some ideas on how to address these problems.

Note: this document does not attempt to use mathematical optimizations, only programming ones. We will probably not be able to avoid standard integration since in many cases we need to integrate forms of the type

$$\int_{\Omega} wuv \, dx, \int_{\Omega} w \frac{\partial u}{\partial x} v \, dx, \dots,$$

where $w(x, y)$ is a constant function (e.g., previous solution). Techniques based on precalculating products of shape functions do not work on these forms unless w is projected to a basis, see [1].

2 Pre-transformed derivatives

Currently, weak forms are passed untransformed shape function derivatives, along with a pointer to the `RefMap` class which offers inverted transformation matrices and the jacobian. These are used to transform the shape function derivatives inside the integrated expression. For example, the weak form of the ubiquitous Laplace operator looks like this inside the function `int_grad_u_grad_v`:

```
(m[0][0]*dudx[i] + m[0][1]*dudy[i]) * (m[0][0]*dvdx[i] + m[0][1]*dvdy[i]) +
(m[1][0]*dudx[i] + m[1][1]*dudy[i]) * (m[1][0]*dvdx[i] + m[1][1]*dvdy[i])
```

This expression contains 10 multiplication operations. This number could be reduced to just two if the values `dudx`, `dudy`, `dvdx`, `dvdy` were transformed already:

```
dudx[i]*dvdx[i] + dudy[i]*dvdy[i]
```

The pre-transformation would only take $O(p^4)$ operations, since there are $O(p^2)$ shape functions to be evaluated in $O(p^2)$ integration points. On the contrary, there are $O(p^6)$ transformations taking place if they are done inside the integrand. In theory, this could speed up the integration of the local stiffness matrices up to 5 times. In practice this will be less but significant savings can still be expected.

The memory requirements are also $O(p^4)$ bytes for the temporary buffers where the transformed derivatives are stored. The transformations would be performed in a setup phase before an element is assembled and the values forgotten when moving to the next element.

TODO: check how values need to be transformed in H(curl)

TODO: check if this would also work for second derivatives

The simplified integrands working with pre-transformed derivatives are also better suited to automatic vectorization by a compiler supporting the SSE3 instruction set. The compilers we have tried (`gcc`, `icc`) refuse to vectorize the current expressions due to their complexity (use of the matrix `m`).

3 New weak form definitions

The pre-transformed derivatives should also simplify the definition of the weak forms for the user. The bilinear forms could have an easy to understand format, for example,

```
scalar biform(int n, double* wt, double* u[3], double* v[3], double** ext[])
{
```

```

double *dudx = u[1], *dudy = u[2];
double *dvdx = v[1], *dvdy = v[2];
scalar result = 0.0;
for (int i = 0; i < n; i++)
    result += wt[i] * (dudx[i]*dvdx[i] + dudy[i]*dvdy[i]);
return result;
}

```

Here, `wt` is a weight vector which combines the weights of the integration points with the value of the Jacobian in the points. This removes the need to distinguish between elements with constant and nonconstant Jacobian inside the form. The distinction is simply done elsewhere and is cheaper.

The shape functions are passed as an array of three pointers, the first of which points to a buffer of precalculated function values and the other two point to buffers with transformed x and y derivatives. The last parameter, “`ext`”, optionally contains precalculated external functions to be used inside the form, such as previous solutions or their combinations.

Note: physical coordinates contained in integrands (e.g., `x*dudx*dvdx`) should probably be treated as “external functions” too.

4 Precalculated constant expressions

Some problems, such as the Taylor-Galerkin formulation of the Euler equations, contain relatively simple forms with complex constant expressions, such as

$$\left[\frac{u_2}{u_1}(\gamma - 1) \frac{u_2^2 + u_3^2}{u_1^2} - \gamma \frac{u_4}{u_1} \right] \frac{\partial u}{\partial x} \frac{\partial v}{\partial x}.$$

Here, $u_1 \dots u_4$ are previous solutions and the expression in the square brackets is constant in each time step. Its value should only be calculated once for each element (and integration point) in a setup phase, otherwise it is very expensive to evaluate it $O(p^6)$ -times in the bilinear form. It should be possible to define an optional precalculation function for each bilinear form such as

```

void precalc(int n, scalar* ext[], int oext[])
{
    scalar *u1 = ext[0], *u2 = ext[1], *u3 = ext[2], *u4 = ext[3];
    for (int i = 0; i < n; i++)
        ext[4][i] = u2[i]/u1[i] * (gamma-1) *
            (sqr(u2[i]) + sqr(u3[i]))/sqr(u1[i]) - gamma*u4[i]/u1[i];
}

```

This function would evaluate the constant expression on the current element and produce an additional “external function”, `ext[4]`. The polynomial degree of the new function would be written to `oext[4]`. Something similar can currently only be done in a very cumbersome way in Hermes using Filters.

5 Problem with integration orders

A major difficulty in *hp*-FEM is the fact that the weak forms have to be integrated with many different Gauss rules, as opposed to standard FEM, where it is sufficient to have, say, a 9-point quadrature rule for everything. To integrate everything with order 20 rule (121 points for quads) in *hp*-FEM would be a huge waste of CPU time and it is therefore necessary to always determine the smallest possible degree of integration. This is currently done inside the predefined forms, such as `int_grad_u_grad_v`. In order to implement the simplified style of weak forms described above, Hermes would have to know the integration orders ahead of time for all combinations of shape functions for a given form to be able to prepare the precalculated and pretransformed tables. Unfortunately, the user would probably have to provide an additional callback function for each form which would return the required integration order. Such function could be similar to the following:

```
int int_order(int n, int ou, int ov, int oext[])
{
    return ou + ov + oext[0];
}
```

A possible solution for this would be to provide several most common callbacks since most forms just need to add the orders of the shape functions up or take their maximum. The user would then not need to define his own callback most of the time.

TODO: Is there a way to determine the required integration order automatically without knowing the expression for the integrand, only the result of the bilinear form?

6 Remove Judy arrays

We have implemented the above ideas in the form of an experimental assembling procedure with limited capabilities (hard-coded, single equation, no hanging nodes, no multi-mesh) in order to evaluate the possible performance improvements. We have found that the integrands with pre-transformed derivatives are indeed several times ($2.5\times - 3.3\times$) faster to evaluate. However, we observed that 20–60% of the CPU time of the current implementation is lost in calls to `libJudy` (depending on the order of the elements), the library providing sparse associative arrays, which are used to overcome some of the difficulties of *hp*-FEM (the “everything is variable size” effect). The Judy arrays were designed to make good use of the CPU cache and to scale well to millions of items stored in the associative arrays (good asymptotic complexity). Unfortunately, the library is very complicated and the sheer amount of instructions executed on each call to `JudyLIns` makes it unsuitable for our purposes, since we need to access the database of precalculated shape functions very often.

We have come to the conclusion that `libJudy` should be abandoned altogether in future Hermes implementations, if possible. Standard, fixed-size C arrays should be used instead.

This brings us to the next section:

7 Reduce the number of precalculated tables

Hermes stores the values and derivatives of shape functions in precalculated tables. When multi-mesh was introduced, these tables were simply extended to also store all the required sub-element cut-outs of the shape functions. Unfortunately, this turned out to be a bad decision. The size of the table database can grow quickly from several hundred kilobytes in a single-mesh computation to several hundred megabytes (easily turning to gigabytes – in 3D this will be a disaster) in a multi-mesh computation. Not only is this a waste of precious memory that could be used for the solution of the linear system, this also slows down the assembling significantly, since it is vital that most data the program is working with at a given moment fit in the L2 cache.

We believe it would be best not to store the sub-element precalculated tables at all, as it is probably faster to recalculate them in the element setup phase than to fetch them from the RAM. As the following section shows, a full set of shape functions for a degree p sub-element can be calculated with a number of operations that is somewhere between $O(p^4)$ and $O(p^6)$. In the worst case this is comparable with the $O(p^6)$ operations needed to integrate the local stiffness matrix. Given the savings described earlier, in total this might still be comparable or faster than the current performance of single-mesh assembling. This could actually make many multi-mesh computations practical if used carefully (the problem with one high-degree element in one mesh and many low-degree elements in another would still remain, though).

The standard precalculated tables along with probably one or two sub-element levels would of course remain in the code. We still want to be fast when comparing the coarse and fine solution, which is a 1-level multi-mesh problem. But the number of stored levels would be fixed and this would be easy to implement without Judy arrays. Temporary tables deeper than the fixed level would immediately be thrown away after assembling a sub-element.

8 Faster shapesets

So far not much attention has been paid in the code to how quickly shape functions are evaluated, as this is only needed in the beginning of the computation when populating the precalculated tables. If the tables are to be recalculated in the course of the multi-mesh assembling, the shapesets need to be optimized.

We suggest that the shapesets are modified so that their functions return shapes evaluated at all integration points at once (currently there is one function call per shape and integration point). This should be faster, allow vectorization and also it should enable additional

optimizations. For example, quadrilateral shapes are now expressed as

$$\varphi_{ij}(x, y) = l_i(x) \cdot l_j(y),$$

where l_i, l_j are Lobatto polynomials. It costs $O(p)$ operations to evaluate one shape in one integration point and therefore $O(p^3)$ operations are required to evaluate the values in all integration points. On the other hand, if all values are requested at once, one can precalculate $l_i(x)$ for all x and $l_j(y)$ for all y in $O(p^2)$ and then combine these with one multiplication per integration point for $O(p^2)$ points. The total number of operations is $O(p^2)$ for one shape and $O(p^4)$ for all shapes.

For non-product shapes the worst possible complexity is $O(p^2)$ for one integration point, since any polynomial in two variables can be expressed as a linear combination of $(p+1)^2$ monomials $x^i y^j$. The number of operations is therefore $O(p^2)$ using the Horner scheme. The complete set of shapes for all integration points can then be obtained in $O(p^6)$ operations. Typically, however, the actual absolute number can be lower depending on the structure of the shapes: some coefficients in the expansion can be zeros or the coefficients themselves can have some structure, as in the case of product shapes.

9 Combine vertex and edge constrained shape fns

Constrained edge shape functions currently do not contain the two vertex parts. These are added separately to the assembly lists bearing the same DOF number. This means that the assembly lists can be longer than necessary (more shapes with the same DOF). This may cost us something since the number of elements of the local stiffness matrix grows quadratically with the assembly list length.

In $H(\text{curl})$ this is of course not an issue.

10 Faster insertion of local stiffness matrices

After removing the Judy array accesses in the experimental code mentioned above, we found that 50% of the CPU time is spent by integrating the local stiffness matrices, 17% by pretransforming the derivatives and as much as 33% by inserting the local stiffness matrices into the global one, especially if the global matrix has several hundred megabytes. This is most probably caused by poor caching. We should probably use some of the reordering packages when assigning degrees of freedom (e.g., AMD from UMFPACK), so that the elements of the resulting global matrix are as close to the diagonal as possible and that all DOFs of each finite element are as close as possible. This way all accesses into the matrix would cache better. We have done one step in this direction in the past by assigning DOFs in the order they will likely be used, instead of assigning vertex DOFs first, then edge

DOFs and finally the bubble DOFs. This speeded up the assembly somewhat as well as the solve (back-substitution) stage of UMFPACK.

Additionally, one could tell the CPU to start fetching the relevant parts of the global matrix from the RAM before the program begins integrating the local matrix. This should hide some of the RAM latency. A special instruction (PREFETCH) is provided for this purpose on modern CPUs.

11 OpenMP parallelization for SMP machines

The trend of recent years is multi-core CPUs. It is unlikely that CPU frequencies will grow again and this means that (at least) SMP parallelism is a must for packages like Hermes. If Hermes stays serial it means we have hit a performance ceiling and our computations will not get faster with newer CPUs. However, it is hard to introduce efficient parallelism into an existing serial program even with OpenMP. This is the case with Hermes as well. While some speed-up could be gained by adding several OpenMP directives, truly efficient scaling to more cores would currently be very difficult. We tried integrating different rows of the local stiffness matrix in parallel on different cores. This was not enough, since other parts like element setup and matrix insertion need to be parallel too and these operations, if not parallelized, quickly overshadow the parallel part as the number of CPUs grows (Amdahl's law). A reasonable approach seems to be allocating different elements to different cores in the main assembling loop. This, however, would require significant changes in PrecalcShapeset in order for it to become thread-safe.

12 Expression parser

For complex problems even the simplified weak forms of Sections 3 and 4 would still be quite difficult and error-prone to define by the user. We believe a built-in expression parser could save a lot of work and make Hermes much more usable. For example, the following line of code could replace the two callbacks and their registration in our Poisson examples:

```
wf.set_eqn("(u_x*v_x + u_y*v_y) = (2*v)");
```

The expression parser would be an optional layer automatically producing the callbacks and could be used most of the time. The manually-defined callbacks could still be registered and used in case something cannot be done through the parser.

13 Other design changes

The following simplifications and design clean-ups should be considered:

- Merge `Shapeset` and `PrecalcShapeset`, remove `PrecalcShapeset` and calls like `set_pss()`.
- Merge `Solution` and `Filter`, remove `Filter`.
- Rename `Solution` to `Function`, remove old `Function`.
- Maybe split `LinSystem` into more classes, this is of marginal importance though.

References

- [1] R. C. Kirby, M. Knepley, A. Logg, L. R. Scott: Optimizing the Evaluation of Finite Element Matrices, *SIAM J. Sci. Comput.* Vol. 27, No. 3, pp. 741–758, 2005