The FastMat2 Matrix Library. Description and Parallel Implementation

by MA Storti, HG Castro, RR Paz, LD Dalcín

Centro Internacional de Métodos Numéricos
en Ingeniería - CIMEC
INTEC, (CONICET-UNL), Santa Fe, Argentina
<mstorti@intec.unl.edu.ar>
http://www.cimec.org.ar/mstorti

((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Finite element assemble in PETSc-FEM

Most Finite Element codes have the following strategy

```
1  Vec xnod(2,nnod,ndim),
2    state(2,nnod,ndof), res(2,nnod,ndof),
3    new_state(2,nnod,ndof);
4  Mat Jac(4,nnod,ndof,nnod,ndof);
5  Array<int> icone(2,nelem,nel);
6  FastMat2 x(2,nel,ndim),u(2,nel,ndof),
7    eres(2,nel,ndof) ,eJac(4,nel,ndof,nel,ndof);
8  for (int k=0; k<nelem; k++) {
9    // Load element nodes and coords
10   localize(xnod,state,icone,k,x,u);
11   // Compute element residual and jacob
12   element_routine(x,u,eres,eJac);
13   // Assemble
14   assemble(eres,eJac,res,icone,k,Jac);
15 }
16 // Solve system
17 solve(new_state,Jac,res);
```


((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Finite element assemble in PETSc-FEM (cont.)

We focus here in a matrix library for computation of the *residual and Jacobian* contributions.

- **Usually *small matrix dimensions* at the element level computation.**

```
FastMat2 x(2,nel,ndim),u(2,nel,ndof),
    eres(2,nel,ndof) ,eJac(4,nel,ndof,nel,ndof);
```

  - ▷ `ndim` **nbr of spatial dimensions**
  - ▷ `ndof` **nbr of degrees of fredom (fields)**
  - ▷ `nel` **nbr of nodes per element**
- ***Multi-index* support**
- ***Tensorial operations* (e.g. contraction)**
- ***MatLab/Octave operations* (element-by-element operators).**
- **Implements *operation caching* for efficiency.**
- ***Thread safe* for use in a SMP environment (OpenMP). Operation caching is also thread-safe provided that independent cache contexts are used in each thread.**

((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Finite element assemble in PETSc-FEM (cont.)

Basic use

- ***Creation***: `FastMat2 a(3,n,m,p);`.
- ***Delayed creation***: `FastMat2 a; ... ; a.resize(3,n,m,p)`
- ***Resizing, reshaping***.
- ***Import/Export*** vals to plain C vectors of doubles.
- ***Masks***: if `a` is the rank-3 tensor above. `a.ir(1,k);` selects row `k`. `a` behaves like a rank-2 tensor from here on. Mask is cleared by a ***reset*** (`a.rs()`), operation.
- Most operations can be ***concatenated***

  ```
  a.reshape(2,m,p).is(1,k).rs();
  ```


((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Multi-index support

- Full *multi-index*.
- *Rank and shape* can be modified with the `reshape(rank,shape...)` operation.
- Masks can reduce the *effective* rank of the object.
- Many standard tensor operations available:
  - ▷ $a_i = b_{ijj}$ is `a.ctr(b,1,-1,-1)`
  - ▷ $a_{ik} = b_{ij}c_{jk}$ is `a.prod(b,1,-1,-1,2)`.
  - ▷ Full tensor notation like $a_{ik} = b_{ij}c_{kj}$, (equiv to $a = bc'$) is `a.prod(b,1,-1,2,-1)`, allows operations on *transposed matrix* without actually need to transpose the matrix explicitly.
  - ▷ 0-rank tensors are automatically *casted to doubles*.

((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

No operator overloading

- One first approach to implement a syntax for matrix operations in C++ is *overloading the arithmetic operators* (`+-*/`). This allows users to enter matrix products as in standard math notation (`a=b*c+d`). 
- Naive implementation of operator overloading implies *creation of temporaries,* with the resulting *loss of efficiency.* More complex implementations can avoid this in simple cases, but no complete solution to this problem is available. 
- In FastMat2 the *functional approach* is preferred, all operations are performed via function (methods, actually) calls. Notation is harder to read/write for the novice user. 
- This approach avoids creation of temporaries, encompasses full tensorial notation, allows *multiprod* implementation. 

((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Multi-prod

For the operation `a=b*c*d`, with shape (m_a, n_a), (m_b, n_b),... *we are free to choose the order* in which the products are performed, i.e. `b*(c*d)` or `(b*c)*d`. The cost of performing the first product `b*c` is $m_b.n_b.n_c$ and the second is $m_c.n_c.n_d$. They may be very diferent if the dimensions are very disimilar, for instance if `d` is a vector and the other two square matrices. It is highly convenient in this case if the second product is performed first.

There is a *simple heuristic algorithm* that exploits this in the general case. If the multi-product is `a=b*c*d*e*...`, it consists in computing the triple product of indices as before ($m_b.n_b.n_c$, $m_c.n_c.n_d$, ...) for each of the possible products, and to *perform the product that gives the smaller count*.

((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Multi-prod (cont.)

The evaluation order could be imposed by the user, but in more subtle situations *the optimal order of evalution may be non-trivial to determine* and even it may depend on the specific integer parameters of the problem (nbr of fields, nbr of nodes per element, number of dimensions...).

***Operator overloading is not friendly* with the implementation of an algorithm like this, because there is no way to capture the whole set of matrices. In the functional approach it is very simple to implement.**

((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Some common operations

- `a.set(b)` $\longrightarrow$ `a = b`
- `a.add(b)` $\longrightarrow$ `a = a+b`
- `a.rest(b)` $\longrightarrow$ `a = a-b`
- `a.axpy(b,alpha)` $\longrightarrow$ `a = a+alpha*b`
- `a.mult(b)` $\longrightarrow$ `a .* b` **(Matlab element-by-element)**
- `a.div(b)` $\longrightarrow$ `a ./ b` **(Matlab element-by-element)**
- `a.inv(b)` $\longrightarrow$ `a = inv(b)`

((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Some common operations (cont.)

- `a.scale(alpha)` $\longrightarrow$ `a = alpha*a`
- `a.fun(f)` $\longrightarrow$ apply `f` to each element `x` of `a`: `x=f(x)` (signature of f: `double f(double)`)
- `a.fun(f,data)` $\longrightarrow$ same as before, pass also a generic pointer to user data: `f=f(x,data)`
- `d.eig(a,v)` $\longrightarrow$ `[v,d] = eig(a)`
- `d.seig(a,v)` (idem, but `a` must be symmetric)
- `a.det()` $\longrightarrow$ `det(a)`

((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Reduction operations

- `a.sum(...)` $\longrightarrow$ `sum(a)` or `sum(a')`, reduces rank
- `a.sum_square(...)`
- `a.norm_p(...)`
- `a.max(...)`
- `a.min(...)`
- `a.max_abs(...)`
- `a.min_abs(...)`
- `a.reduce(g,null,indices...)` makes reduction with associative binary funcion `double g(double,double)`.
- For each of these there exists and `_all()` version that means reducing the whole matrix to a double, for instance `double sum = a.sum_all()`.

((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Copy, import/export

- `a.set(x)` $\longrightarrow$ sets the whole matrix to constant `x`
- `a.set(ptr)` $\longrightarrow$ fills with doubles starting at `double *ptr`
- `a.export_vals(ptr)` $\longrightarrow$ copy contents to `double *ptr`
- `a.set(b)` $\longrightarrow$ copy contents from `FastMat2 b`
- `ptr=a.storage_bein()` $\longrightarrow$ returns `double *` to internal storage.

((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Operation caching

Many computations need some operations to be performed for each element that could be *factored out* of the element loop.

- **Computing indices in selection of columns or row ranges.**

  `a(k,:)=b(k,:)*C(1:n,1:n)`
- **Computing matching indices in tensorial expressions**

  `a(i,j)=b(i,k)*c(k,j)`
- **Computing the order in which a multi-prod is evaluated.**

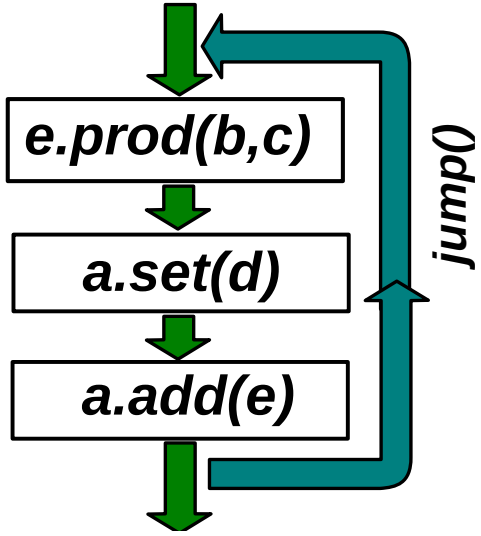
```
1 for (int j=0; j<nelem; j++) {
2   b.is(1,j);
3   c.is(2,k);
4   e.prod(b,c,d,...);
5 }
```


((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Operation caching (cont.)

As a distinctive feature of the FastMat2 library, this is done activating a *tree of operation caches* (in fact it is a *direct acyclic graph* DAG). The operation caches are stored in the `ctx`.

```
1  // Computes a=b*c+d
2  FastMat2::CacheCtx ctx;
3  FastMat2 a(ctx,...),b,c,d,e;
4  FastMat2::CacheCtx2::Branch branch;
5  for (int j=0; j<nelem; j++) {
6    ctx.jump(branch);
7    // Fill b,c,d
8    e.prod(b,c);
9    a.set(d).add(e);
10 }
```

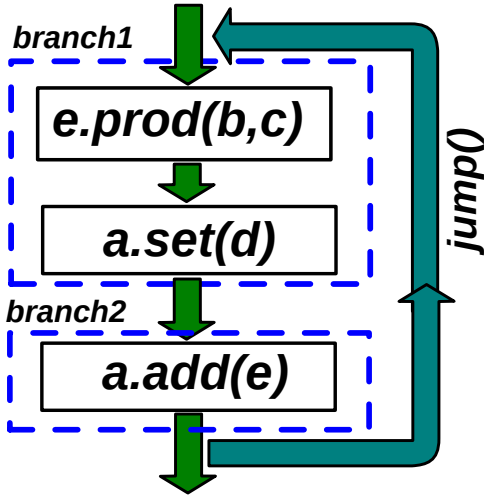


((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Operation caching (cont.)

If *branching* is used, then special directives must be used in order to keep *synchronization* between the position in the cache and the operations in the code. A *branch* is a sequence of operations that are always executed in the same order. For each branch the sequence of operation caches is stored the first time that the code is executed (normally in the first execution of the loop).

```
1  // Computes 'a=b*c+d' or 'a=b*c'
2  FastMat2::CacheCtx c;
3  FastMat2 a(ctx,...),b,c,d,e;
4  FastMat2::CacheCtx2::Branch br1,br2;
5  for (int j=0; j<nelem; j++) {
6    ctx.jump(br1);
7    // Fill b,c,d
8    e.prod(b,c);
9    a.set(d);
10   if (flag) {
11     ctx.jump(br2);
12     a.add(e);
13   }
14 }
```



((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Operation caching (cont.)

Lack of synchronization results in a *cache missmatch*. If `flag==0` for the `j==0`, then the cache for `a.add(d)` is not added to branch `br1`. Then if in a subsequent execution (`j>0`) `flag` results to be true, then the cache for `a.add(d)` is found when executing `a.add(e)`, and a *cache missmatch* is produced.

```
1  // Computes a=b*c+d or a=b*c+e
2  for (int j=0; j<nelem; j++) {
3    ctx.jump(br1);
4    a.prod(b,c,...);
5    if (flag) {
6      // missing ctx.jump(br2);
7      a.add(d);
8    } else {
9      // missing ctx.jump(br3);
10     a.add(e);
11   }
12 }
```


((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Operation caching (cont.)

There are a series of utilities to help in *debugging* cache errors.

- **Under user option a *terse/friendly string identifiyng the operation* (type, operands, and other information) is added to the cache. If a missmatch is found an error message is generated.**
- **A faster and more general error detection can be activated based in a *hash* (MD5SUM) of the operation type and all other operation info.**
- **Both mechanisms are deactivated by default (for efficiency).**

((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Efficiency

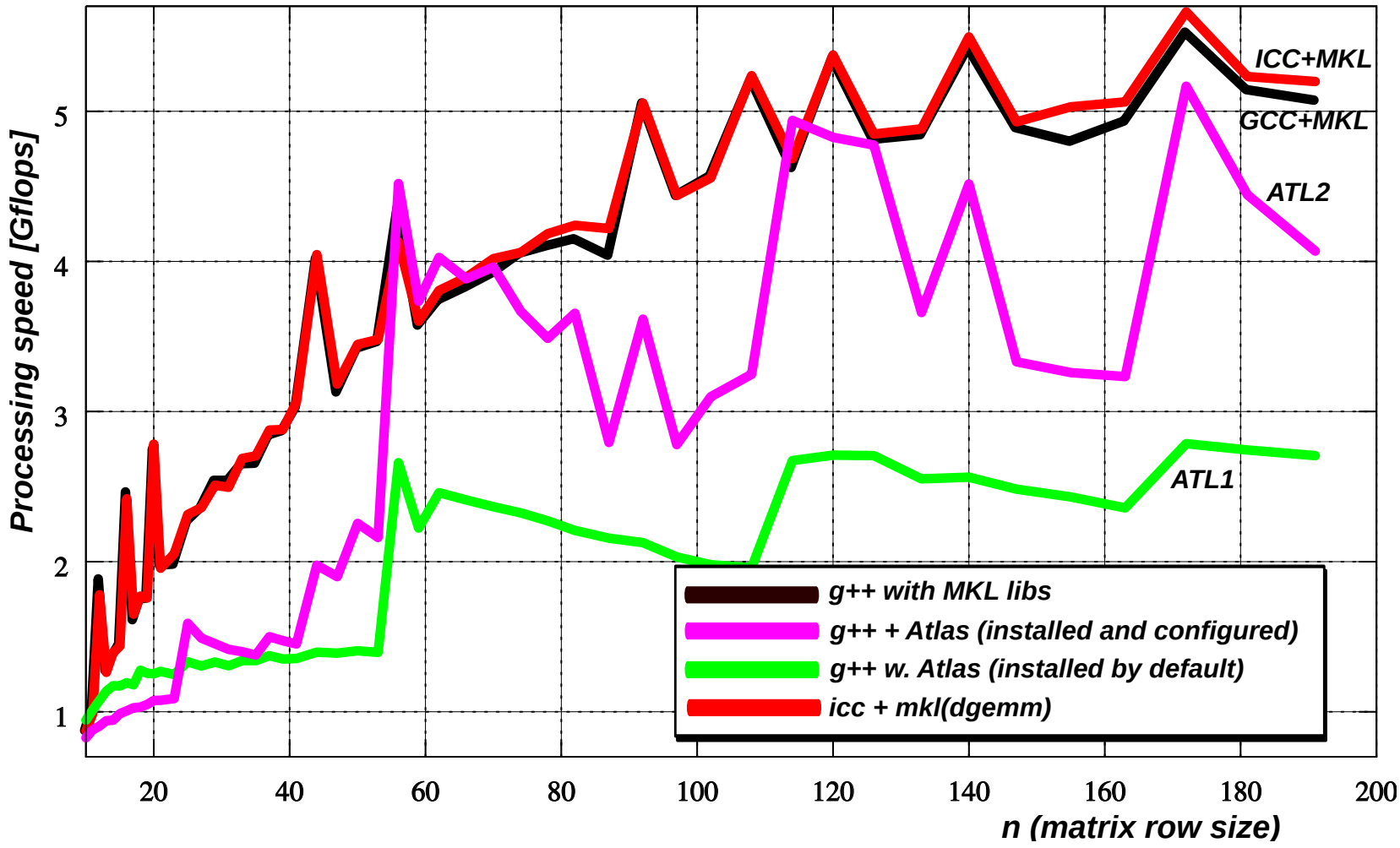
- With operation caching activated the overhead of the mask computation is avoided, and the amortized cost is similar to a plain C loop.
- This benchmark computes `a=b*c` in a loop for a large number N of distinct square matrices `a,b,c` of varying size n. The amortized cost is the same as the underlying `dgemm()` call. Gflops rate are computed in base to an operation count of $2n^3$

$$\text{rate [Gflops]} = \text{1e-9}\frac{N.2n^3}{\text{(elapsed time [secs])}} \tag{1}$$

- The number of times for reaching a 50% amortization ($n_{1/2}$) is in the order of 15 to 30.
- For the matrix product a significant improvement may be obtained if the library is linked to the Intel Math Kernel Libray (MKL), combined with either Intel icc or GCC compiler. This combinations peak at 5 to 6 Gflops for $100 < n < 200$.

((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Efficiency (cont.)



((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Use in SMP (OpenMP)

- Non-caching use is thread-safe provided distinct threads don't manipulate the same object at the same time.
- With operation caching activated the library is thread safe provided that a cache DAG (`FastMat2::CacheCtx`) is used for each thread and each thread has its own copies of matrices associated with this context.

```
1  #pragma omp parallel
2      {
3        FastMat2::CacheCtx ctx;
4        FastMat2::CacheCtx::Branch br;
5        FastMat2 a(&ctx,2,n,n),b(&ctx,2,n,n),c(&ctx,2,n,n);
6  #pragma omp for schedule(dynamic)
7        for (int j=0; j<N; j++) {
8          ctx.jump(br);
9          // Fill b and c...
10           a.prod(b,c);
11       }
12     }
```


((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))

Acknowledgment

This work has received financial support from Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET, Argentina, PIP 5271/05), Universidad Nacional del Litoral (UNL, Argentina, grants CAI+D 2005-10-64) and Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT, Argentina, grants PICT Lambda 12-14573/2003, PME 209/2003, PICT-1506/2006).

We made extensive use of *Free Software* (`http://www.gnu.org`) as GNU/Linux OS, MPI, PETSc, GCC/G++ compilers, Octave, Open-DX among many others. In addition, many ideas from these packages have been inspiring to us.

((version texstuff-1.0.29-197-gf8e0d76) (date Wed Nov 4 18:20:44 2009 -0300) (processed-date Thu Nov 5 08:50:14 2009 -0300))