Recent advances in HPC with FreeFem++

Pierre Jolivet

Laboratoire Jacques-Louis Lions
Laboratoire Jean Kuntzmann

Fourth workshop on FreeFem++

December 6, 2012

With F. Hecht, F. Nataf, C. Prud'homme.

Outline

Where were we one year ago ?

What we had:

- FreeFem++ version 3.17,
- a “simple” toolbox for domain decomposition methods,
- 3 supercomputers (SGI, Bull, IBM).

What we achieved:

- linear problems up to 120 million unkowns in few minutes,
- good scaling up to ∼2048 processes on a BlueGene/P.

FreeFem++ is working on the following parallel architectures:

	N° of cores	Memory	Peak performance
hpc1@LJLL	160@2.00 GHz	640 Go	$\sim$ 10 TFLOP/s
titane@CEA	12192@2.93 GHz	37 To	140 TFLOP/s
babel@IDRIS	40960@850 MHz	20 To	139 TFLOP/s

http://www-hpc.cea.fr, Bruyères-le-Châtel, France.
http://www.idris.fr, Orsay, France (grant PETALh).

FreeFem++ is working on the following parallel architectures:

	N° of cores	Memory	Peak performance
hpc1@LJLL	160@2.00 GHz	640 Go	$\sim$ 10 TFLOP/s
titane@CEA	12192@2.93 GHz	37 To	140 TFLOP/s
babel@IDRIS	40960@850 MHz	20 To	139 TFLOP/s
curie@CEA	80640@2.7 GHz	320 To	1.6 PFLOP/s

`http://www-hpc.cea.fr`, Bruyères-le-Châtel, France.
`http://www.idris.fr`, Orsay, France (grant PETALh).
`http://www.prace-project.eu` (grant HPC-PDE).

FreeFem++ and the linear solvers

`set(A, solver = sparsesolver)` in FreeFem++

$$\Downarrow$$

instance of `VirtualSolver<T>` from which inherits a solver.

FreeFem++ and the linear solvers

`set(A, solver = sparsesolver)` in FreeFem++

$\Downarrow$

instance of `VirtualSolver<T>` from which inherits a solver.

Interfacing a new solver basically consists in implementing:

1. the constructor `MySolver(const MatriceMorse<T>&)`

FreeFem++ and the linear solvers

`set(A, solver = sparsesolver)` in FreeFem++

$\Downarrow$

instance of `VirtualSolver<T>` from which inherits a solver.

Interfacing a new solver basically consists in implementing:

1. the constructor `MySolver(const MatriceMorse<T>&)`
2. the pure virtual method

```
void Solver(const MatriceMorse<T>&, KN_<T>&,
            const KN_<T>&) const
```

FreeFem++ and the linear solvers

`set(A, solver = sparsesolver)` in FreeFem++

$\Downarrow$

instance of `VirtualSolver<T>` from which inherits a solver.

Interfacing a new solver basically consists in implementing:

1. the constructor `MySolver(const MatriceMorse<T>&)`
2. the pure virtual method

```
void Solver(const MatriceMorse<T>&, KN_<T>&,
            const KN_<T>&) const
```

3. the destructor `MySolver(const MatriceMorse<T>&)`

FreeFem++ and the linear solvers

`set(A, solver = sparsesolver)` in FreeFem++

$\Downarrow$

instance of `VirtualSolver<T>` from which inherits a solver.

Interfacing a new solver basically consists in implementing:

1. the constructor `MySolver(const MatriceMorse<T>&)`
2. the pure virtual method

   ```
   void Solver(const MatriceMorse<T>&, KN_<T>&,
               const KN_<T>&) const
   ```
3. the destructor `MySolver(const MatriceMorse<T>&)`

`real[int] x = A^-1 * b` will call `MySolver::Solver`.

MUltifrontal Massively Parallel Sparse direct Solver

`http://graal.ens-lyon.fr/MUMPS`

Distributed memory direct solver.
Compiled by `FreeFem++` with `--enable-download`.
Renumbering via AMD, QAMD, AMF, PORD, (Par)METIS, (PT-)SCOTCH.

Solves unsymmetric and *symmetric* linear systems !

MUMPS and FreeFem++

```
load "MUMPS"
int[int] l = [1, 1, 2, 2];
mesh Th;
if(mpirank != 0) // no need to store the matrix on ranks other than 0
    Th = square(1, 1, label = l);
else
    Th = square(150, 150, label = l);
fespace Vh(Th, P2);
varf lap(u, v) = int2d(Th)(dx(u)*dx(v) + dy(u)*dy(v)) + int2d(Th)(v)
    + on(1, u = 1);
real[int] b = lap(0, Vh);
matrix A = lap(Vh, Vh);
set(A, solver = sparsesolver);
Vh x; x[] = A^-1 * b;
plot(Th, x, wait = 1, dim = 3, fill = 1, cmm = "sparsesolver", value =
    1);
```

Intel MKL PARDISO

`http://software.intel.com/en-us/intel-mkl`

Shared memory direct solver.
Part of the Intel Math Kernel Library.
Renumbering via Metis or threaded nested dissection.

Solves unsymmetric and *symmetric* linear systems !

PARDISO and FreeFem++

```
load "PARDISO"
include "cube.idp"
int n = 35; int[int] NN = [n, n, n]; real[int, int] BB = [[0,1], [0,1],
    [0,1]]; int[int, int] L = [[1,1], [3,4], [5,6]];
mesh3 Th = Cube(NN, BB, L);
fespace Vh(Th, P2);
varf lap(u, v) = int3d(Th)(dx(u)*dx(v) + dz(u)*dz(v) + dy(u)*dy(v))
    + int3d(Th)(v) + on(1, u = 1);
real[int] b = lap(0, Vh);
matrix A = lap(Vh, Vh);
real timer = mpiWtime();
set(A, solver = sparsesolver);
cout << "Factorization: " << mpiWtime() - timer << endl;
Vh x; timer = mpiWtime();
x[] = A^-1 * b;
cout << "Solve: " << mpiWtime() - timer << endl;
```

Is it really necessary ?

If you are using direct methods within `FreeFem++`: yes !

Why ? Sooner or later, `UMFPACK` will blow up:

```
UMFPACK V5.5.1 (Jan 25, 2011): ERROR: out of memory

umfpack_di_numeric failed
```

Motivation

One of the most straightforward way to solve BVP in parallel.

Motivation

One of the most straightforward way to solve BVP in parallel.

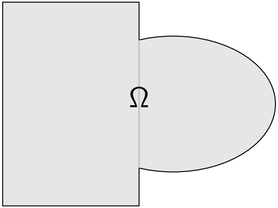
Based on the "divide and conquer" paradigm:

1. assemble,
2. factorize and
3. solve smaller problems.

A short introduction I

Consider the following BVP in $\mathbb{R}^2$:

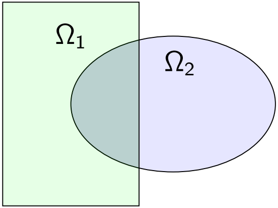
$$\begin{aligned} \nabla \cdot (\kappa \nabla u) &= F \quad \text{in } \Omega \\ B(u) &= 0 \quad \text{on } \partial\Omega \end{aligned}$$



A short introduction I

Consider the following BVP in $\mathbb{R}^2$:

$$\begin{aligned} \nabla \cdot (\kappa \nabla u) &= F \quad \text{in } \Omega \\ B(u) &= 0 \quad \text{on } \partial\Omega \end{aligned}$$



Then, solve in parallel:

$$\begin{aligned} \nabla \cdot (\kappa \nabla u_1^{m+1}) &= F_1 \;\text{ in } \Omega_1 \\ B(u_1^{m+1}) &= 0 \quad \text{on } \partial\Omega_1 \cap \partial\Omega \\ u_1^{m+1} &= u_2^m \;\text{ on } \partial\Omega_1 \cap \Omega_2 \end{aligned}$$

$$\begin{aligned} \nabla \cdot (\kappa \nabla u_2^{m+1}) &= F_2 \;\text{ in } \Omega_2 \\ B(u_2^{m+1}) &= 0 \quad \text{on } \partial\Omega_2 \cap \partial\Omega \\ u_2^{m+1} &= u_1^m \;\text{ on } \partial\Omega_2 \cap \Omega_1 \end{aligned}$$

A short introduction II

We have effectively divided, but we have yet to conquer.

Duplicated unknowns coupled via a *partition of unity*:

$$I = \sum_{i=1}^{N} R_i^T D_i R_i,$$

where R_i^T is the prolongation from V_i to V.

A short introduction II

We have effectively divided, but we have yet to conquer.

Duplicated unknowns coupled via a *partition of unity*:

$$I = \sum_{i=1}^{N} R_i^T D_i R_i,$$

where R_i^T is the prolongation from V_i to V.
Then,

$$u^{m+1} = \sum_{i=1}^{N} R_i^T D_i u_i^{m+1}.$$

One-level preconditioners

A common preconditioner is:

$$M_{\text{RAS}}^{-1} := \sum_{i=1}^{N} R_i^T D_i (R_i A R_i^T)^{-1} R_i \,.$$

For future references, let $\widetilde{A_{ij}} := R_i A R_j^T$.

One-level preconditioners

A common preconditioner is:

$$M_{\text{RAS}}^{-1} := \sum_{i=1}^{N} R_i^T D_i (R_i A R_i^T)^{-1} R_i\,.$$

For future references, let $\widetilde{A_{ij}} := R_i A R_j^T$.

These preconditioners don't *scale* as N increases:

$$\kappa(M^{-1}A) \leqslant C\frac{1}{H^2}\left(1 + \frac{H}{\delta h}\right)$$

They act as low-pass filters (think about mulgrid methods).

Two-level preconditioners

A common technique in the field of DDM, MG, deflation:

introduce an auxiliary “coarse” problem.

Two-level preconditioners

A common technique in the field of DDM, MG, deflation:

introduce an auxiliary "coarse" problem.

Let Z be a rectangular matrix so that the "bad eigenvectors" of $M^{-1}A$ belong to the space spanned by its columns. Define

$$E := Z^T A Z \qquad\qquad Q := Z E^{-1} Z^T \,.$$

Z has $\mathcal{O}(N)$ columns, hence E is *relatively* smaller than A.

Two-level preconditioners

A common technique in the field of DDM, MG, deflation:

introduce an auxiliary “coarse” problem.

Let Z be a rectangular matrix so that the “bad eigenvectors” of $M^{-1}A$ belong to the space spanned by its columns. Define

$$E := Z^T A Z \qquad\qquad Q := Z E^{-1} Z^T \,.$$

Z has $\mathcal{O}(N)$ columns, hence E is *relatively* smaller than A.

The following preconditioner can *scale* theoretically:

$$P^{-1}_{\text{A-DEF1}} := M^{-1}(I - AQ) + Q \,.$$

Preconditioners in action

Back to our Krylov method of choice: the GMRES.

procedure GMRES(input vector x_0, right-hand side b)
  $r_0 \leftarrow P^{-1}_{\text{A-DEF1}}(b - Ax_0)$
  $v_0 \leftarrow r_0 / ||r_0||_2$
  for $i = 0, \ldots, m-1$ **do**
    $w \leftarrow P^{-1}_{\text{A-DEF1}} A v_i$
    for $j = 0, \ldots, i$ **do**
      $h_{j,i} \leftarrow \langle w, v_j \rangle$
    end for
    $\tilde{v}_{i+1} \leftarrow w - \sum_{j=1}^{i} h_{j,i} v_j$
    $h_{i+1,i} \leftarrow ||\tilde{v}_{i+1}||_2$
    $v_{i+1} \leftarrow \tilde{v}_{i+1} / h_{i+1,i}$
    apply Givens rotations to $h_{:,i}$
  end for
  $y_m \leftarrow \arg\min ||\overline{H}_m y_m - \beta e_1||_2$ with $\beta = ||r_0||_2$
  return $x_0 + V_m y_m$
end procedure

Preconditioners in action

Back to our Krylov method of choice: the GMRES.

procedure GMRES(input vector x_0, right-hand side b)
  $r_0 \leftarrow P^{-1}_{\text{A-DEF1}}(b - Ax_0)$
  $v_0 \leftarrow r_0/||r_0||_2$
  for $i = 0, \ldots, m-1$ **do**
    $w \leftarrow P^{-1}_{\text{A-DEF1}} A v_i$ global sparse matrix-vector multiplication
    for $j = 0, \ldots, i$ **do**
      $h_{j,i} \leftarrow \langle w, v_j \rangle$
    end for
    $\tilde{v}_{i+1} \leftarrow w - \sum_{j=1}^{i} h_{j,i} v_j$
    $h_{i+1,i} \leftarrow ||\tilde{v}_{i+1}||_2$
    $v_{i+1} \leftarrow \tilde{v}_{i+1}/h_{i+1,i}$
    apply Givens rotations to $h_{:,i}$
  end for
  $y_m \leftarrow \arg\min ||\overline{H}_m y_m - \beta e_1||_2$ with $\beta = ||r_0||_2$
  return $x_0 + V_m y_m$
end procedure

Preconditioners in action

Back to our Krylov method of choice: the GMRES.

procedure GMRES(input vector x_0, right-hand side b)
  $r_0 \leftarrow P_{\text{A-DEF1}}^{-1}(b - Ax_0)$
  $v_0 \leftarrow r_0/||r_0||_2$
  for $i = 0, \ldots, m-1$ **do**
    $w \leftarrow P_{\text{A-DEF1}}^{-1} A v_i$
    for $j = 0, \ldots, i$ **do**
      $h_{j,i} \leftarrow \langle w, v_j \rangle$
    end for
    $\tilde{v}_{i+1} \leftarrow w - \sum_{j=1}^{i} h_{j,i} v_j$
    $h_{i+1,i} \leftarrow ||\tilde{v}_{i+1}||_2$
    $v_{i+1} \leftarrow \tilde{v}_{i+1}/h_{i+1,i}$
    apply Givens rotations to $h_{:,i}$
  end for
  $y_m \leftarrow \arg\min ||\overline{H}_m y_m - \beta e_1||_2$ with $\beta = ||r_0||_2$
  return $x_0 + V_m y_m$
end procedure

global preconditioner-vector computation

global sparse matrix-vector multiplication

Preconditioners in action

Back to our Krylov method of choice: the GMRES.

procedure GMRES(input vector x_0, right-hand side b)
  $r_0 \leftarrow P^{-1}_{\text{A-DEF1}}(b - Ax_0)$
  $v_0 \leftarrow r_0/||r_0||_2$
  for $i = 0, \ldots, m-1$ **do**
    $w \leftarrow P^{-1}_{\text{A-DEF1}} A v_i$
    for $j = 0, \ldots, i$ **do**
      $h_{j,i} \leftarrow \langle w, v_j \rangle$
    end for
    $\tilde{v}_{i+1} \leftarrow w - \sum_{j=1}^{i} h_{j,i} v_j$
    $h_{i+1,i} \leftarrow ||\tilde{v}_{i+1}||_2$
    $v_{i+1} \leftarrow \tilde{v}_{i+1}/h_{i+1,i}$
    apply Givens rotations to $h_{:,i}$
  end for
  $y_m \leftarrow \arg\min ||\overline{H}_m y_m - \beta e_1||_2$ with $\beta = ||r_0||_2$
  return $x_0 + V_m y_m$
end procedure

global preconditioner-vector computation

global sparse matrix-vector multiplication

global dot products

In practice

A scalable DDM framework must include routines for:

- *p2p* communications to compute `spmv`,
- *p2p* communications to apply a one-level preconditioner,
- global transfer between *master* and *slaves processes* to apply “coarse” corrections,
- direct solvers for the “coarse” problem and local problems.

In FreeFem++

```
subdomain A(S, D, restrictionIntersection, arrayIntersection,
    communicator = mpiCommWorld); // build buffers for spmv
preconditioner E(A); // build M_{RAS}^{-1}
if(CoarseOperator) {
  ...
  attachCoarseOperator(E, A, A = GEVPA, B = GEVPB, parameters
      = parm); // build P_{A-DEF1}^{-1}
}
GMRESDDM(A, E, u[], rhs, dim = 100, iter = 100, eps = eps);
```

Nonlinear elasticity

We solve an evolution problem of hyperelasticity on a beam, find u such that: $\forall t \in [0; T], \forall v \in [H^1(\Omega)]^3$,

$$\int_\Omega \rho \ddot{u} \cdot v + \int_\Omega \underbrace{(I + \nabla u)\,\Sigma}_{=:\mathcal{F}(u)} : \nabla v = \int_\Omega f \cdot v + \int_{\partial\Omega_S} g \cdot v$$

$$u = 0 \text{ on } \partial\Omega_D$$

$$\sigma(u) \cdot n = 0 \text{ on } \partial\Omega_N$$

where

$$\Sigma = \lambda \mathrm{tr}(E) I + 2\mu E$$

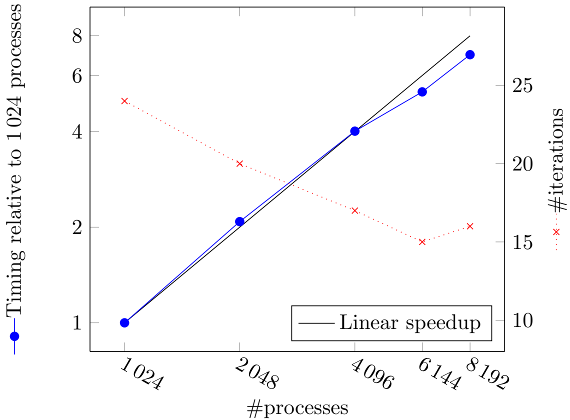
In FreeFem++

Add another operator `rebuild` (currently only rebuilds $\widetilde{A_{ii}}^{-1}$).
The rest is (almost) the same.

```
for(int k = 0; k < 200; ++k) {
    SchemeInit(u)
    if(k > 0) {
        S = vPbNonlinear(Wh, Wh, solver = CG);
        rhs = vPbLinear(0, Wh);
        rebuild(S, A, E);
    }
    GMRESDDM(A, E, h[], rhs, dim = 100, iter = 100, eps = eps);
    SchemeAdvance(u, h)
}
```

Outline

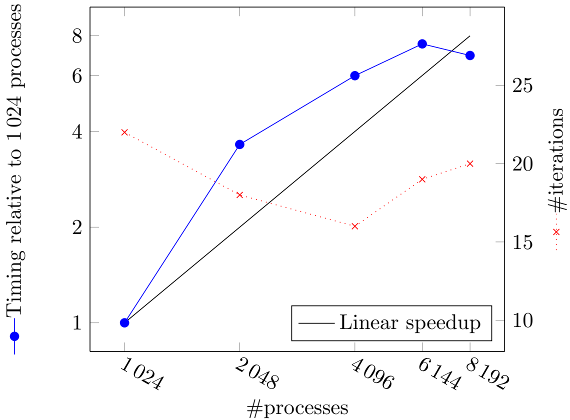
Amuses bouche before lunch I



Linear elasticity in 2D with $\mathbb{P}_3$ FE.

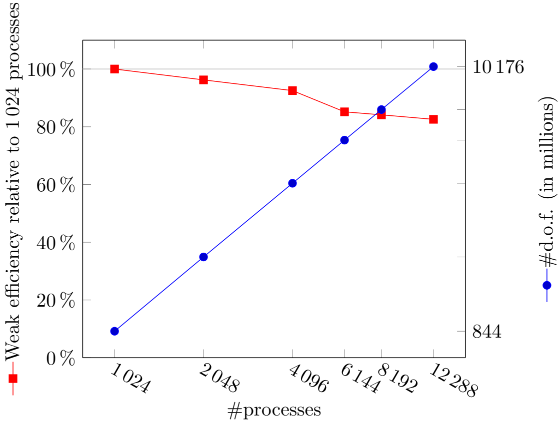
1 billion unknowns solved in 31 seconds at peak performance.

Amuses bouche before lunch II



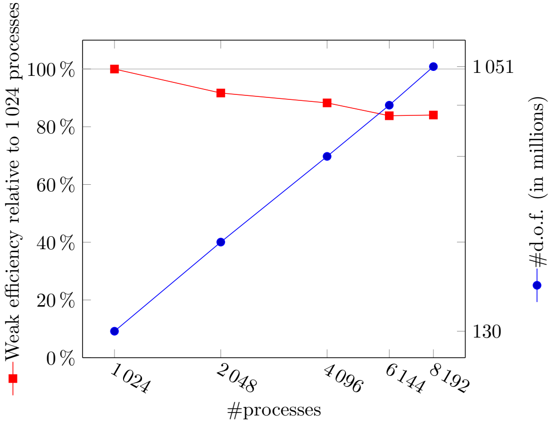
Linear elasticity in 3D with $\mathbb{P}_2$ FE.
80 million unknowns solved in 35 seconds at peak performance.

Amuses bouche before lunch III



Scalar diffusivity in 2D with $\mathbb{P}_3$ FE.
10 billion unknowns solved in 160 seconds on 12 228 processes.

Amuses bouche before lunch IV



Scalar diffusivity in 3D with $\mathbb{P}_2$ FE.
1 billion unknowns solved in 150 seconds on 8 192 processes.

To sum up

1. $$\begin{array}{c}\texttt{FreeFem++} \\ + \\ \text{Two-level DDM}\end{array} = \text{easy framework to solve large systems.}$$
2. New solvers give performance boost on smaller systems.

To sum up

1. `FreeFem++` + Two-level DDM = easy framework to solve large systems.
2. New solvers give performance boost on smaller systems.

New problems being tackled:

- more complex nonlinearities,
- reuse of Krylov and deflation subspaces,
- BNN and FETI methods.

To sum up

1. `FreeFem++` + Two-level DDM = easy framework to solve large systems.
2. New solvers give performance boost on smaller systems.

New problems being tackled:

- more complex nonlinearities,
- reuse of Krylov and deflation subspaces,
- BNN and FETI methods.

Thank you for your attention.