BOAST ULtimately LeaDing to an Optimized Gysela

Collaborators of CEA:
Julien Bigot, Ksander Ejjaaouani, Virginie Grandgirard, Guillaume Latu

Collaborators of INRIA and University of Grenoble:
Steven Quinito Masnada, Luis Felipe Garlet-Millani,
Jean-Francois Mehaut, Brice Videau

Collaborators of INRIA Lyon:
Jerome Richard

25/08/2016

Outline

1. Context: Kernels from the Gysela code
2. Dynamic task-based tuning with StarPU
3. Static auto-tuning with BOAST
4. Final words

Gysela: Gyrokinetic Semi-Lagrangian

Tokamak plasma simulation for fusion (ITER)

A self-consistent coupling

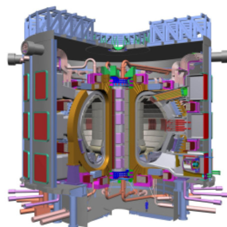
- Fields computation + particle motion
- Vlasov equation to solve at each time step t

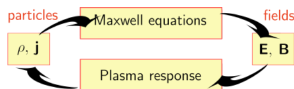
Tokamak kinetic model: 6D phase space

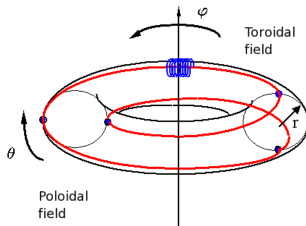
- 3D in space: (r, θ, φ)
- 3D in velocity: $(v_\perp, \alpha, v_\parallel)$

Time scale of particle gyration
$\ll$ ITG turbulence

- **Gyrokinetic**: 5D reduced problem
- $\mu = \frac{mv_\perp^2}{2B}$ replaces $(v_\perp, \alpha)$

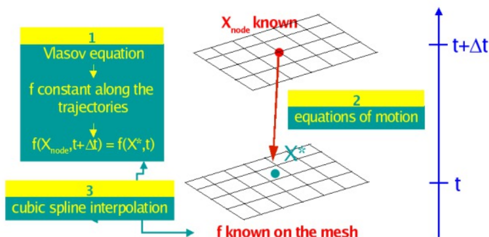






The Semi-Lagrangian method (backward version)

2D (r, $theta$) advection



- f conserved along characteristics
- Find the origin of the characteristics ending at the grid points
- Interpolate value at origin $X^\star$ of characteristics from known grid values $\rightarrow$ Interpolation method needed

- Typical interpolation schemes:
  - Cubic spline
  - Lagrange polynomial (high order)

Parallel algorithm for a 2D advection (in r, θ directions)

Input : $\bar{f}^{\star}(r, \theta, \varphi, v_{\|}, \mu)$
Output : $\bar{f}^{\diamond}(r, \theta, \varphi, v_{\|}, \mu)$

```
for μ // MPI do in parallel
    for v∥ // MPI do in parallel
        for φ // MPI+OpenMP do in parallel
            for θ and r do
                [Δr_(r,θ), Δθ_(r,θ)] ← Compute_displacement(Electric field, Δt)
            Prepare_interpolation (if spline coeff needed)
            for θ and r do
                f̄◇(r,θ,φ,v∥,μ) = interpolate(f̄⋆(r−Δr_(r,θ), θ−Δθ_(r,θ), φ, v∥, μ))
```

- for μ // MPI do in parallel
  - for $v_{\|}$ // MPI do in parallel
    - for φ // MPI+OpenMP do in parallel
      - for θ *and* r do
        - $[\Delta r_{(r,\theta)}, \Delta\theta_{(r,\theta)}] \leftarrow \mathit{Compute_displacement}(\mathit{Electric\ field}, \Delta t)$
      - *Prepare_interpolation* (if spline coeff needed)
      - for θ *and* r do
        - $\bar{f}^{\diamond}(r, \theta, \varphi, v_{\|}, \mu) = \mathit{interpolate}(\bar{f}^{\star}(r - \Delta r_{(r,\theta)}, \theta - \Delta\theta_{(r,\theta)}, \varphi, v_{\|}, \mu))$

Algorithm 1: Advection in variables along r, θ dimensions on $\bar{f}^{\star}$

Task-based programming models

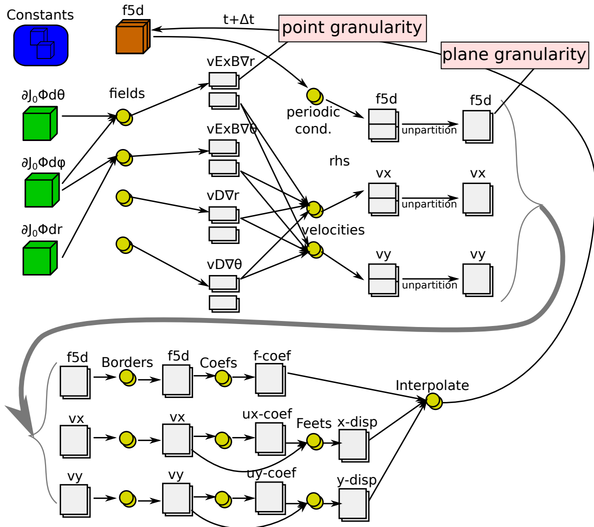
- **Goal**: improve performance portability
- **Approach**: choose execution order at run-time
- Done by a *scheduler*: **StarPU** in our case
- program is expressed as a **DAG**: a set of *tasks* each reading & writing into buffers

Scheduling

- can improve performance & **performance portability** by handling load imbalance due to code or from execution resources
- introduces an overhead

Cf. all the talks of this morning

2D Advection, now with **tasks**! Analysis

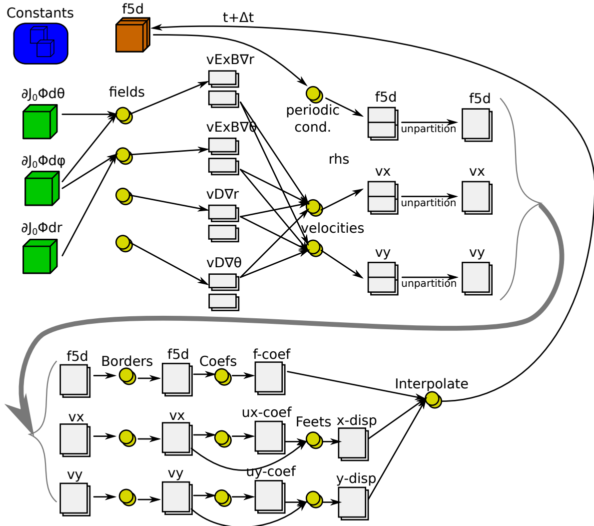


As preliminary work, let's identify:

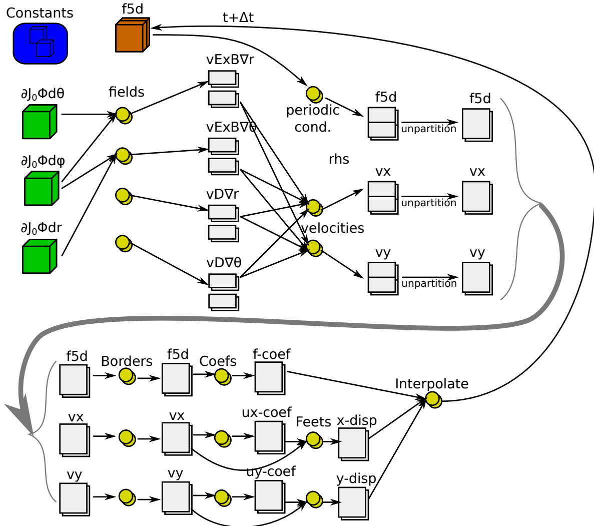
- the inputs,
- in/outs,
- constants used,
- potential independent tasks,
- their minimal granularity,
- their dependencies →

⇒ that's our **DAG**

Tasks: Strategy & implementation choices

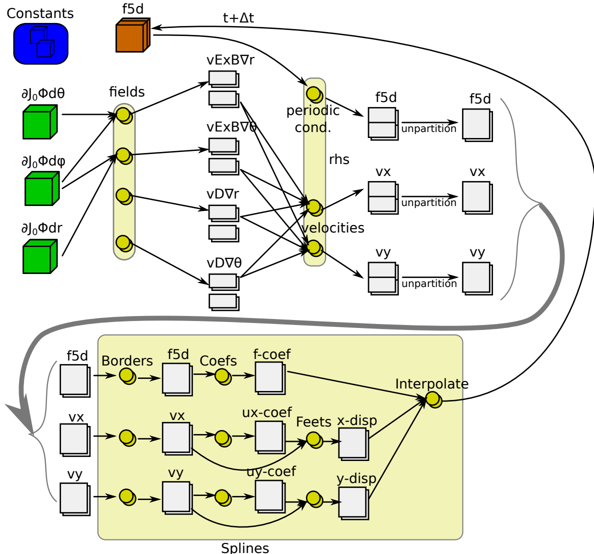


Tasks: Strategy & implementation choices



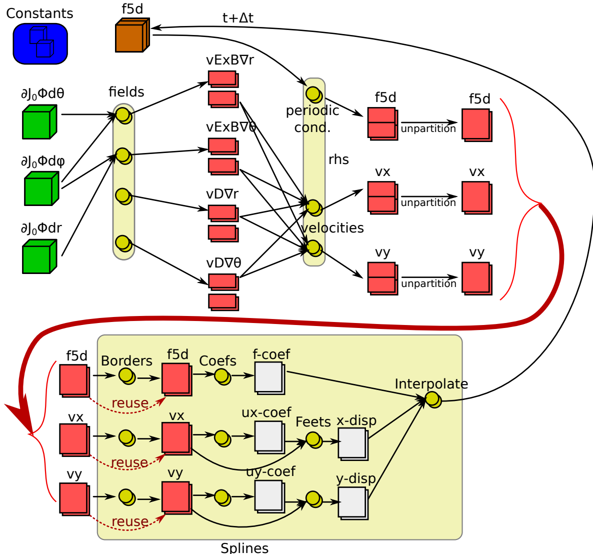
- **data-parallelism granularity**: *block of lines* where possible, *whole plane* otherwise
  - repartitioning

Tasks: Strategy & implementation choices



- **data-parallelism granularity**: *block of lines* where possible, *whole plane* otherwise
  - repartitioning
- **task-parallelism granularity**: regroup *full plane* tasks, regroup tasks *accessing same data*

Tasks: Strategy & implementation choices



Expectation: improve cache behavior

- **data-parallelism granularity**: *block of lines* where possible, *whole plane* otherwise
  - repartitioning
- **task-parallelism granularity**: regroup *full plane* tasks, regroup tasks *accessing same data*
- **data management**: use *scratch* if possible *temp* otherwise

Tasks: Some preliminary results

Tests on *Poincare*, 2×Sandy E5-2670 (2.60GHz, 8×2 cores), 32 GB RAM
StarPU SVN-18399 (2016-08-4), Scheduler: WS with data locality
$(r, \theta, \varphi, v_{||}, \mu)$ small:$(64, 128, 32, 16, 1)$, big:$(512, 512, 32, 16, 1)$

- **ref**: Reference version from Gysela
- **tasks**: Initial task version
- **fuse**: Fuse tasks as described
- **scratch**: Use scratch data instead of temporary where possible
- **pause**: Pause the scheduler when submitting to prevent conflicts
- **lines**: Use block of line granularity on the first tasks ⇒ repartitioning

	duration	
version	small	big
ref	**0.11s**	**3.18s**
tasks	0.52s	3.69s
fuse	0.22s	3.70s
scratch	0.19s	2.73s
pause	0.10s	2.69s
lines	crash	crash

But... executing this new version with static scheduling:

small	big
0.12s	2.67s

Tasks: Lessons learned

- Choosing the right granularity
  - is di cult (scheduler liberty vs. overhead)
  - can be done at the task & data parallelism
  - depends on the problem size
  - has a HUGE impact
- Optimizing data management is a **requirement**
- Tasks should be submitted in an good order to ease the scheduler work
- Re-partitioning the right way is complex
  - might incur a full barrier if synchronous
  - can be done asynchronously...
  - but we didn't manage to get it to work correctly ☹
- Static scheduling remains as efficient as tasks

But thinking with tasks ⇒ 15% gain on big (realistic) case

BOAST intro - Sample code

- BOAST is a programming framework dedicated to **code generation** and autotuning

  e.g. A BOAST program (**ruby**) generates a Fortran/C program
- Possible goal: design an **auto-tuning** process for computation kernels

Eg: The following declaration

```
unroll = 5
tt = (0...unroll).collect { |t| Real("tt#{t}") }
pr For(j, 1, ndat-(unroll-1), :step => unroll){
  pr For( i, 0, n-1) {
    tt.each { |t| pr t === 0.0 }
    pr For(l, -8, 7) {
      pr k === modulo(i + l, n)
      (0...unroll).each { |u|
        pr tt[u] === tt[u] + x[k,j+u]*fil[l]
      }
    }.unroll
    (0...unroll).each { |u| pr y[j+u,i] === tt[u] }
  }
}
```

Outputs Fortran:

```
do j=1, ndat-4, 5
  do i=0, n-1
    !......
    k = modulo(i+2, n)
    tt0=tt0+x(k,j+0)*fil(2)
    tt1=tt1+x(k,j+1)*fil(2)
    !......
  enddo
enddo
```

Or C:

```
1    for(j=1; j<=ndat-4; j+=5){
2      for(i=0; i<=n-1; i+=1){
3    /*...........*/
4        k = modulo(i+2, n)
5        tt0=tt0+x[k-0+(j+0-1)*(n-1-0+1)]*fil[2-lowfil];
6        tt1=tt1+x[k-0+(j+1-1)*(n-1-0+1)]*fil[2-lowfil];
7    /*...........*/
8      }
9    }
```

Gysela 2d avection - BOAST implementation

- Preparation steps
  - Extract 4 targeted routines from Gysela (subpart of 2d advection)
  - Change API of the 2d advection kernel
    only arrays of integers and floats for inputs/outputs
    (transmitting data structures is ***possible*** but more ***complex***)
  - Define valid ***fake*** inputs for the kernel to design a regression test
  - Integrate the reference/original version into BOAST
- Install ruby & BOAST on 4 parallel machines
  - Easiest step
  - Get a working compilation/execution of the kernel: a bit more di cult
- Write a meta-program that *prints* a program
  1. Need to learn a little bit of ruby & BOAST
  2. Incremental approach: begin with internal routines then external
  3. Identify what are the ***parameters*** of the auto-tuning
  4. Integrate the best kernel version to the Gysela compilation process

Gysela 2d avection - BOAST implementation (cont'd)

- Auto-tuning parameters that we chose
  - directive based inlining / BOAST driven inlining
  - BOAST driven loop unrolling
  - C or Fortran code generated
  - scan versions of gfortran/gcc/icc/ifort (`module load`)
  - loop blocking parameter (one of the most internal loop)
  - explicit vectorization: BOAST generates INTEL intrinsincs, e.g.

```
ftmp1 = _mm256_setzero_pd( );
ftmp2 = _mm256_setzero_pd( );
ftmp1 = _mm256_fmadd_pd( base1[0], _mm256_load_pd( &ftransp[(0) * (4)]);
ftmp2 = _mm256_fmadd_pd( base1[0 + 1], _mm256_load_pd( &ftransp[(0 + 1) * (4)] ), ftmp2 );
```

- Final result
  - ruby code of 200 lines for the 2d advection kernel
    compared to original fortran code of 300 lines

- Auto-tuning runs
  - configure the list of modules/compilers for the *parameter scan*
  - between 1 min and 20 min for the parameter scan on 1 machine

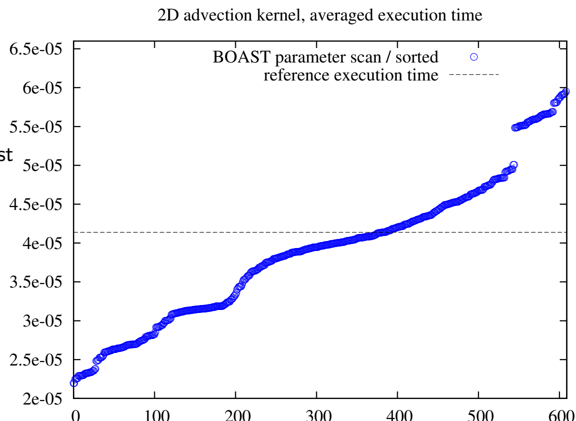
Auto-tuning on INTEL Westmere (2011)

Auto-tuning for 2D advection
Computing center at Marseille
12-cores node -
Intel X5675, 3.07GHz

Nb of runs in this scan: 609
Runs sorted from quickest to slowest
Result of the scan
(best parameters):

```
:lang: FORTRAN
:unroll: true
:force.inline: true
:intrinsic: false
:blocking.size: 4
:module: intel/16.0.2
```

Speedup: 1.9



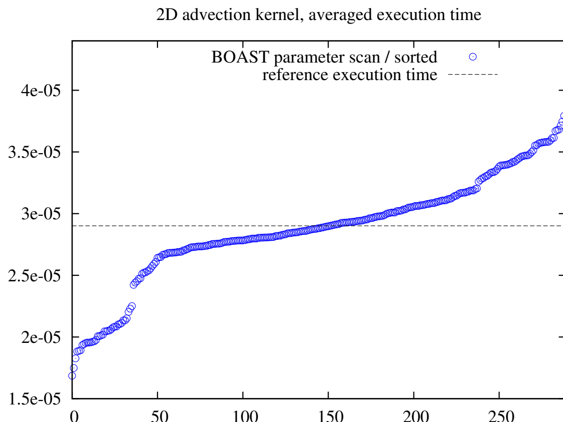
Auto-tuning on INTEL Sandy-Bridge (2012)

Auto-tuning for 2D advection
Computing center at Orsay
16-cores node -
 Intel E5-2670 v1, 2.60GHz

Result of the scan
 (best parameters):

```
:lang: FORTRAN
:unroll: false
:force.inline: false
:intrinsic: false
:blocking.size: 2
:module: intel/15.0.0
```

Speedup: 1.7



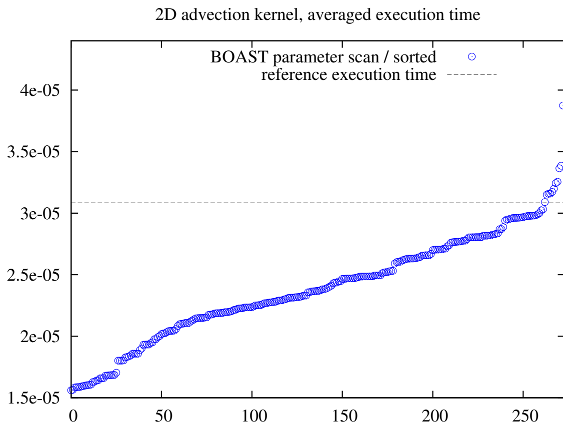
Auto-tuning on INTEL Haswell (2015)

Auto-tuning for 2D advection
Computing center at Montpellier
24-cores node -
Intel E5-2690 v3, 2.60GHz

Result of the scan
(best parameters):

```
:lang: FORTRAN
:unroll: true
:force.inline: true
:intrinsic: false
:blocking.size: 4
:module: intel/14.0.4.211
```

Speedup: 2.0



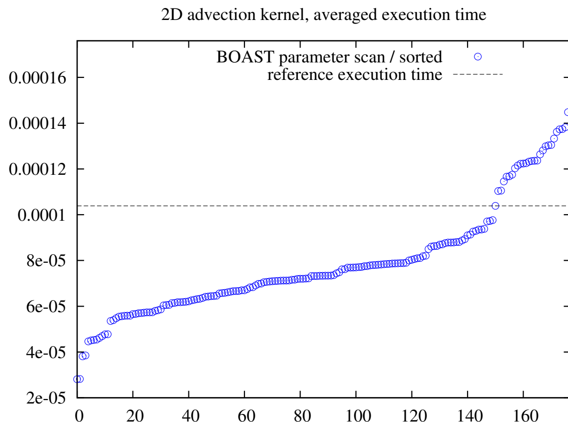
Auto-tuning on INTEL KNL (Xeon Phi 2016)

Auto-tuning for 2D advection
Computing center at Montpellier
64-cores node -
Intel 7210 1.30GHz

Result of the scan
(best parameters):

```
:lang: FORTRAN
:unroll: true
:force.inline: true
:intrinsic: false
:blocking.size: 1
:module: intel/17.0
```

Speedup: 3.6



Conclusion, future works

- StarPU
  - Managed to reach Gysela perf. on small case, −15% on big (realistic) case
    - Requires medium to large grain (each task $\gtrsim 1ms$)
    - Improvement related to code reorganization rather than scheduling
  - Will be di cult to integrate in full code with other parallelism models
  - Code complexity increased, native StarPU API too heavy (OpenMP 4.5 better ?)
  - *Might* become more useful with finer grain, heterogeneous architectures, ...
- BOAST
  - DONE: Auto-tuning of 2d kernel improves perf (speedup 1.7 up to 3.6)
  - DONE: On Sandy-Bridge, 1.3 speedup of ***advec2d*** step on Gysela runs
  - DONE: Several improvements of BOAST during CEMRACS
    AVX512 support, rules to restrict parameter space,
    energy consumption analysis, integration of 'module load',
    improved cross-langage support (fortran+C)
  - TODO: integrating BOAST for production use in Gysela
  - TODO: better vectorization is needed on KNL
  - Not yet determined: Add other Gysela kernels in BOAST ?

https://github.com/Nanosim-LIG/boast