Westfälische
Wilhelms-Universität
Münster

Hybrid Parallelization of Assembly in DUNE

living.knowledge
WWU Münster

Jö Fahlke, Christian Engwer

July 16, 2014

ExaDune

Dune+PDELab

- Framework for solving PDEs
- Already good at MPI-Parallelization

Exa-Scale Computers

- Arriving around 2018
- Many processing units
- Little memory per processing unit
- Accelerator hardware (e.g. GPU, MIC, etc.)

Outline

Intro

Threading

Vectorization

Conclusions

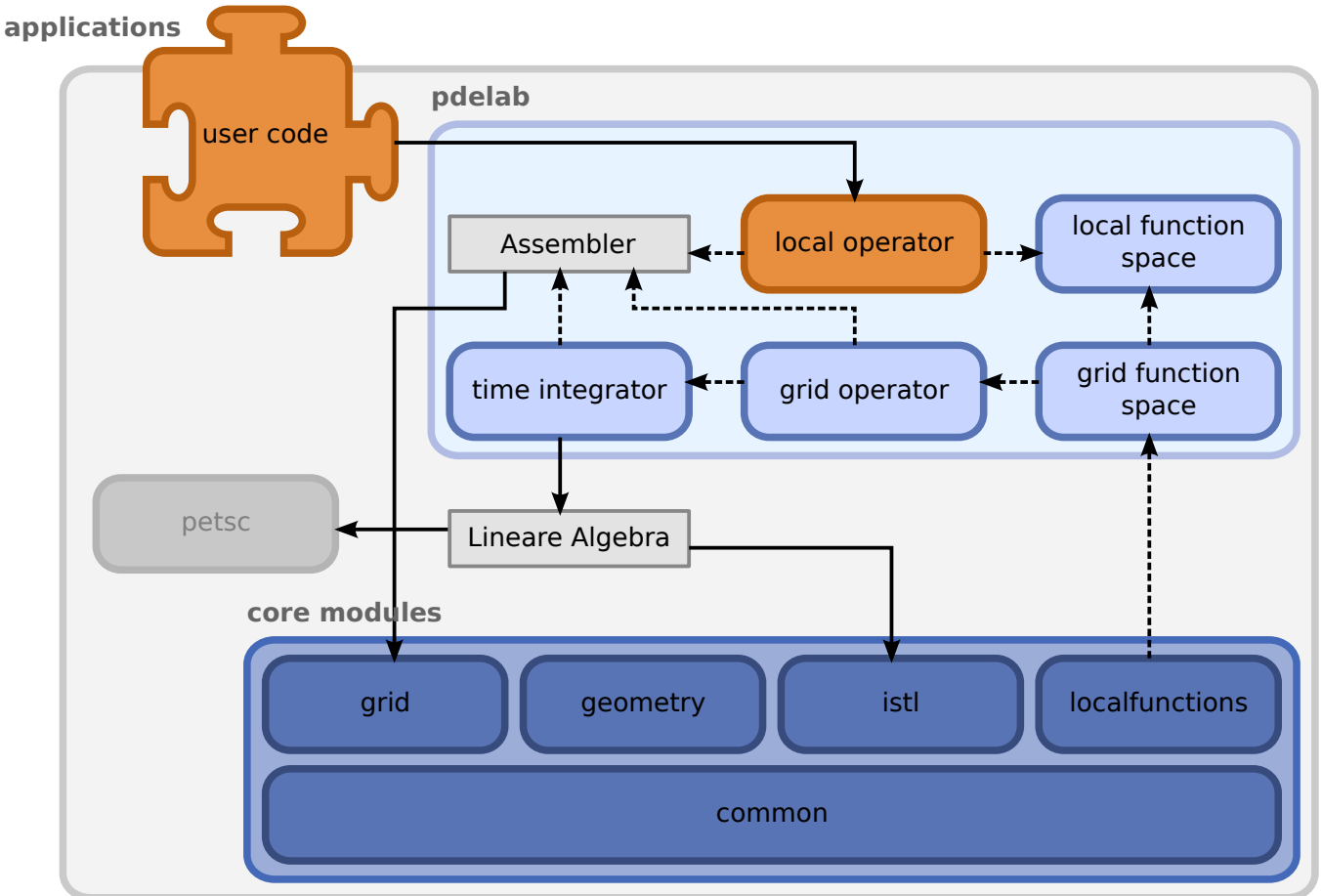
Outlook

Anatomy of a PDE solver

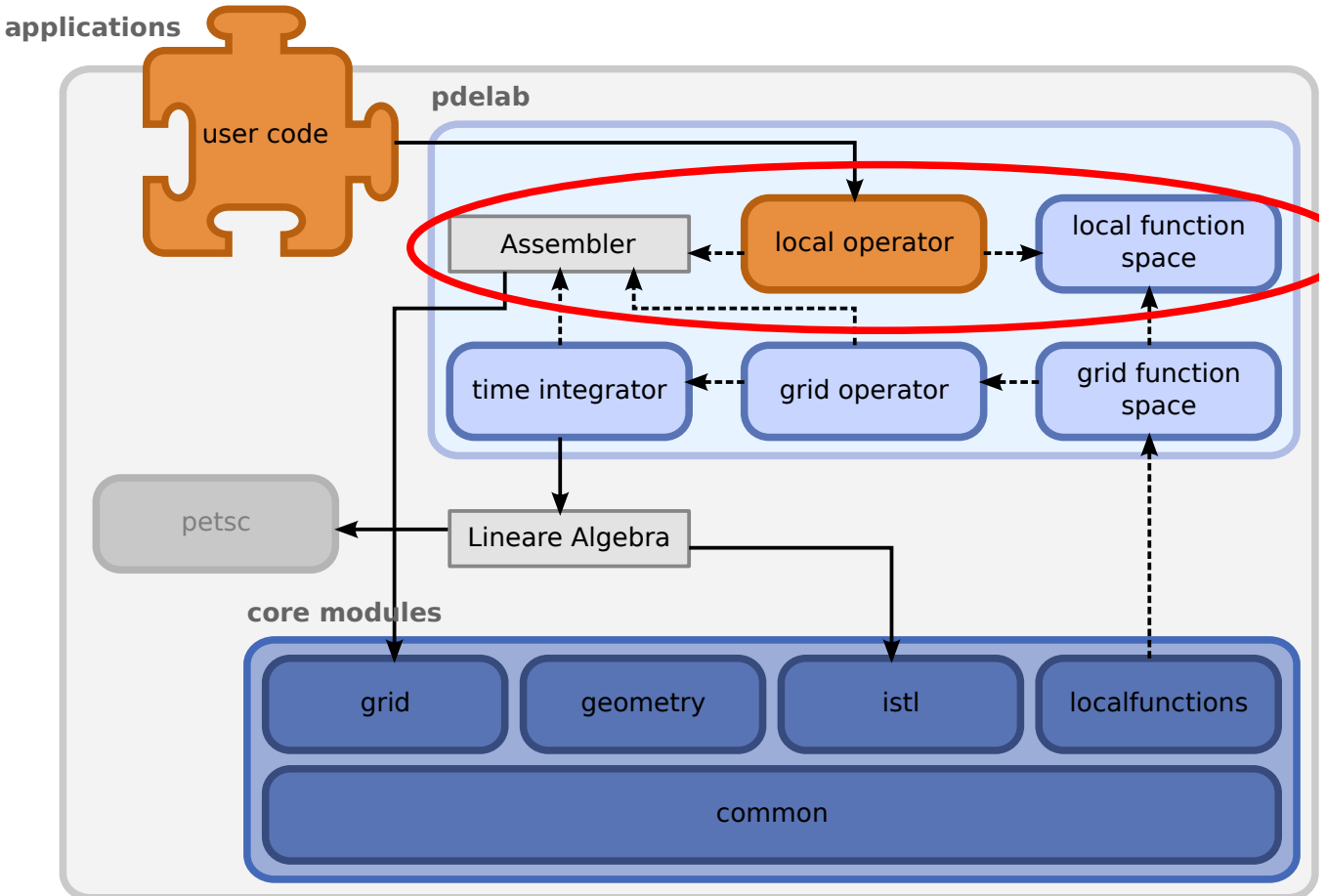
Two components eat most of the CPU cycles and benefit most from parallelization:

- Linear algebra
  (Steffen Müthing)
- Assembling the linear system
  (this talk)

DUNE-PDELab design



DUNE-PDELab design



Challenges for PDE frameworks

- Assembly kernels often written by the framework user
- User is not an expert in programming

  $\Longrightarrow$ We can't rely on obscure languages
- Avoid multiple versions of a kernel

  $\Longrightarrow$ Kernels must be portable.
- Keep it open

  $\Longrightarrow$ Avoid relying on proprietary languages and libraries.

Threading

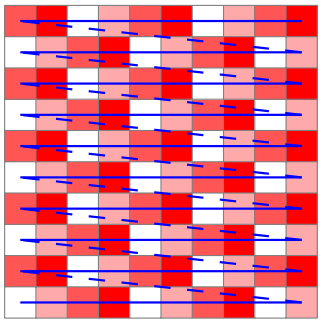
Why Threading?

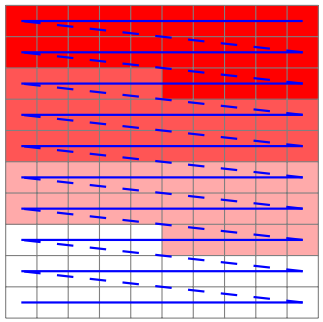
- Reduced memory overhead compared to message passing
- Reduced communication overhead.

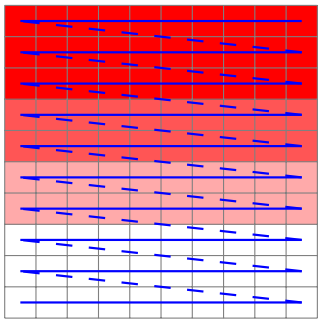
Challenges in threading

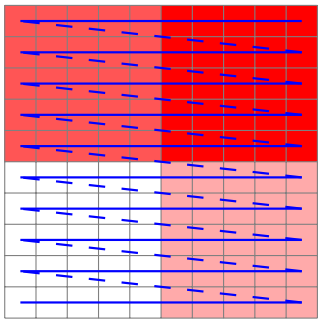
- Choosing a partitioning scheme.
- Choosing a race avoidance strategy.

Partitioning Strategies


strided


ranged


sliced


tensor

- Strided: calculated on the fly, but only efficient with random access iterators.
- Ranged: memory efficient, needs preprocessing or random access iterators.
- General (sliced, tensor): not memory efficient, but enables coloring, or tuning for small surface area.

Data Access Strategies

Races can occur when accumulating into global data structures.

Strategies to avoid races:

- batched: Batched writeback with global lock.
- elock: One lock per mesh entity
- coloring: partitions of the same color do not “touch”.

Other strategies not considered here:

- global locking
- race-free schemes
  - not considered since it is often not possible.
  - but tried by R. Klöfkorn (Proceedings of ALGORITMY 2012).

Test Setup

- Stationary advection problem

$$\nabla \cdot (-A(x)\nabla u + b(x)u) + c(x)u = f \qquad \text{in } \Omega,$$

  with appropriate Dirichlet and outflow boundary conditions.
- DG (weighted SIPG).
- Orthonormalized P^k basis.

- Wall time for assembly of residual and jacobian.
- As many threads as possible.

Results

	batched	elock	colored
 strided	1048ns (7%)	350ns (22%)	
 ranged	712ns (10%)	209ns (37%)	
 sliced	712ns (10%)	212ns (37%)	209ns (36%)
 tensor	715ns (10%)	208ns (37%)	211ns (36%)

Table: PHI, degree=1, jacobian, threads=240.

- Runtime per DoF and (efficiency).

CPU vs. PHI

k	t_1^{CPU}	t_{10}^{CPU}	t_{20}^{CPU}	t_1^{PHI}	t_{60}^{PHI}	t_{120}^{PHI}	t_{240}^{PHI}
0	4.59µs	0.74µs	0.54µs	59.57µs	1.33µs	1.17µs	1.20µs
1	1.38µs	0.22µs	0.17µs	18.92µs	0.37µs	0.27µs	0.26µs
2	1.10µs	0.15µs	0.12µs	17.12µs	0.32µs	0.21µs	0.19µs
3	1.29µs	0.16µs	0.13µs	19.84µs	0.36µs	0.23µs	0.20µs
4	1.52µs	0.18µs	0.15µs				
5	1.81µs	0.21µs	0.18µs				

Runtimes per dof, degree k, jacobian, sliced partitioning, entity-wise locking

- CPU still better than PHI.

CPU vs. PHI

k	t_1^{CPU}	t_{10}^{CPU}	t_{20}^{CPU}	t_1^{PHI}	t_{60}^{PHI}	t_{120}^{PHI}	t_{240}^{PHI}
0	4.59µs	0.74µs	0.54µs	59.57µs	1.33µs	1.17µs	1.20µs
1	1.38µs	0.22µs	0.17µs	18.92µs	0.37µs	0.27µs	0.26µs
2	1.10µs	0.15µs	0.12µs	17.12µs	0.32µs	0.21µs	0.19µs
3	1.29µs	0.16µs	0.13µs	19.84µs	0.36µs	0.23µs	0.20µs
4	1.52µs	0.18µs	0.15µs				
5	1.81µs	0.21µs	0.18µs				

Runtimes per dof, degree k, jacobian, sliced partitioning, entity-wise locking

- CPU still better than PHI.
- Unfair comparison: SIMD units are 128bit for CPU and 512bit for PHI.

$\Longrightarrow$ Requires vectorization.

Conclusions

- Useful partitionings:
  - ranged: memory efficient
  - general: as a backup, allows coloring
- Useful data access strategies:
  - entity-wise locking: general, good performance
  - coloring: good performance, needs particular partitioning
- Need vectorization.

Vectorization

Zoology of Devices

- CPU
  - Lots of smart heuristics
  - Good single-thread performance

  Typical stats (1 UMA node):
  - 10 cores, 20 threads
  - 2 SIMD lanes (`double` precision)
  - 48GiB memory
- Phi
- GPU

Zoology of Devices

- CPU
- Phi
  - Simplified CPU
  - Needs a host system for housing
  - Main program can run on host (with offloading) or native on device

  Typical stats:
  - 60 cores, 240 threads
  - 8 SIMD lanes (`double` precision)
  - 8GiB memory
- GPU

Zoology of Devices

- CPU
- Phi
- GPU
  - Very basic processor
  - Needs host system for housing and scheduling
  - Main programs runs on host, offloading to device

  Typical stats:
  - 2000–3000 cores currently*
  - 32 SIMT lanes
  - 5–12GiB memory

  *Meaning of *core* differs from CPU/PHI

Zoology of Devices

- CPU
- Phi
- GPU

Programming Approaches I

Unsuitable approaches:

- Intrinsics (non-portable)
- Special language (needs special compiler)
- Autovectorizer (difficult to drive portably)

Programming Approaches II

Better approach: vectorization library

- Hide instrinsics beneath a portable interface
- Possible drawback: are compilers still capable of reordering optimizations?
- Implementations: Vc, Boost.SIMD[1], NGSolve
- Could also be implemented on top of some special languages (Cilk, OpenMP).

[1] Not an official Boost library

SIMD Example: SISD

Compute one scalar product

```
double scalar_product(unsigned size,
                      double *a, double *b)
{
  double sum = 0;
  for(unsigned i = 0; i < size; ++i)
    sum += a[i] * b[i];
  return sum;
}
```

SIMD Example: SIMD I

Compute one scalar product

```
double scalar_product(unsigned size,
                      double *a, double *b) {
  // assume *a and *b are properly aligned
  // assume size % lanes == 0
  Vector<double> sum = 0;
  for(unsigned i = 0; i < size; i += lanes)
    sum += (Vector<double>&)(a[i])
                 * (Vector<double>&)(b[i]);
  for(unsigned i = 1; i < lanes; ++i)
    sum[0] += sum[i];
  return sum[0];
}
```

SIMD Example: SIMD I

Compute one scalar product

```
double scalar_product(unsigned size,
                      double *a, double *b) {
  // assume *a and *b are properly aligned
  // assume size % lanes == 0
  Vector<double> sum = 0;
  for(unsigned i = 0; i < size; i += lanes)
    sum += (Vector<double>&)(a[i])
         * (Vector<double>&)(b[i]);
  for(unsigned i = 1; i < lanes; ++i)
    sum[0] += sum[i];
  return sum[0];
}
```

Bad example!

SIMD Example: SISD

Compute one scalar product

```
double scalar_product(unsigned size,
                      double *a, double *b)
{
  double sum = 0;
  for(unsigned i = 0; i < size; ++i)
    sum += a[i] * b[i];
  return sum;
}
```

SIMD Example: SIMD II

Compute many scalar products at once

```
Vector<double> scalar_product(unsigned size,
                              Vector<double> *a, Vector<double> *b)
{
  Vector<double> sum = 0;
  for(unsigned i = 0; i < size; ++i)
    sum += a[i] * b[i];
  return sum;
}
```

SIMD Example: SIMD II

Compute many scalar products at once

```
template<class T>
T scalar_product(unsigned size,
                 T *a, T *b)
{
  T sum = 0;
  for(unsigned i = 0; i < size; ++i)
    sum += a[i] * b[i];
  return sum;
}
```

Works both in scalar and vectorized modes

Vectorization Results

- Continuous CG, Q^k
- Simplified problem: scattering is missing.
- Best result: assembly of residual, Q^4 on a regular grid of $15 \times 16 \times 16$ elements, 240 threads: 282GFlop/s (28% peak).

Residual assembly

	#elems	#dofs	GFlop/s	%peak	speedup
Q^1	1 966 080	2 013 561	21	2	1.2
Q^2	245 760	2 013 561	76	8	1.5
Q^3	30 720	856 219	166	16	1.7
Q^4	3 840	257 725	282	28	2.0

Vectorization Results

- Continuous CG, Q^k
- Simplified problem: scattering is missing.
- Best result: assembly of residual, Q^4 on a regular grid of $15 \times 16 \times 16$ elements, 240 threads: 282GFlop/s (28% peak).

Jacobian assembly

	#elems	NNZ	GFlop/s	%peak	speedup
Q^1	1 966 080	4 027 123	21	2	1.8
Q^2	245 760	23 876 718	62	6	3.5
Q^3	30 720	30 178 929	65	6	3.4
Q^4	3 840	19 897 272	60	6	3.6

Conclusions

Conclusions

Vectorization libraries

+ Easy to understand interface.
+ Simple enough for non-expert.
– Preliminary performance results are not very satisfying.

- Finish evaluation of vectorization approaches.
- Grid data structures for GPU-type devices
  - Current ideas inspired by OP2
    `http://www.oerc.ox.ac.uk/projects/op2`
  - Should also help with vectorization on CPU/PHI.
- GPU

Pointers

- DUNE: `http://www.dune-project.org/`
- PDELab: `http://www.dune-project.org/pdelab/`
- Vc by M. Kretz and V. Lindenstruth: `http://compeng.uni-frankfurt.de/?vc`
- Thanks to Steffen Müthing who provided code for counting floating point ops.
- Funding was provided by the DFG through the SPPEXA priority programme: `http://www.sppexa.de/`