Sparse Matrix Computations on Hybrid Computers

From Design Patterns to
Preconditioners on Heterogeneous Platforms

Salvatore Filippone

Università di Roma Tor Vergata
salvatore.filippone@uniroma2.it
https://psctoolkit.github.io

Outline

- Introduction: the PSBLAS library
- Serial kernels;
- Design patterns and object-oriented techniques;
- Using GPUs;
- Heterogeneous platforms;
- Preconditioners: approximate inverses;

Credits

- Alfredo Buttari, CNRS-IRIT, Toulouse (F);
- Davide Barbieri, Daniele Bertaccini, Valeria Cardellini, Un. Roma “Tor Vergata”, (I);
- Alessandro Fanfarillo, ARM Inc. (USA)
- Michele Martone, Max Planck, Garching (D);
- Pasqua D’Ambra, CNR, Naples (I);
- Fabio Durastante, Un. Pisa (I);
- Karla Morris, Sandia National Labs, Livermore (CA);
- Damian Rouson, Lawrence Berkeley Lab, (CA)

PSBLAS and AMG4PSBLAS: Parallel Sparse Solvers

PSBLAS and AMG4PSBLAS: Parallel Sparse Krylov solvers

Main features:

- Designed for Krylov iterative linear system solvers;
- Main application: PDEs;
- Support for graph partitioning with graph partitioning;
- Preconditioners: Simple preconditioners, plud AMG4PSBLAS: Agebraic Multigrid/Domain Decomposition framework;
- OOP design, easy to use (and extend);
- Transparent MPI/CUDA via PSBLAS (with some caveats);

Freely available from `https://psctoolkit.github.io`

S. Filippone and M. Colajanni, ACM TOMS, Dec. 2000,
P. D'Ambra, S. Filippone, D. di Serafino, Appl. Num. Math. 2007
P. D'Ambra, S. Filippone, D. di Serafino, ACM TOMS 2010.
S. Filippone and A. Buttari, ACM TOMS, 38(4), 2012
P. D'Ambra, F. Durastante and S. Filippone, ArXiV 2021 (SCICOMP)

The Goal: write CG

or any other Krylov method, like this:

Template CG	PSBLAS Implementation
Compute $r^{(0)} = b - Ax^{(0)}$	`call psb_geaxpby(one,b,zero,r,desc_a,info)`
	`call psb_spmm(-one,A,x,one,r,desc_a,info)`
	`rho = zero`
for $i = 1, 2, \ldots$	`iterate: do  it = 1, itmax`
solve $Mz^{(i-1)} = r^{(i-1)}$	`call prec%apply(r,z,desc_a,info,work=aux)`
$\rho_{i-1} = {r^{(i-1)}}^T z^{(i-1)}$	`rho_old = rho`
	`rho = psb_gedot(r,z,desc_a,info)`
if $i = 1$	`if (it == 1) then`
$p^{(1)} = z^{(0)}$	`call psb_geaxpby(one,z,zero,p,desc_a,info)`
else	`else`
$\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$	`beta = rho/rho_old`
$p^{(i)} = z^{(i-1)} + \beta_{i-1}p^{(i-1)}$	`call psb_geaxpby(one,z,beta,p,desc_a,info)`
endif	`endif`
$q^{(i)} = Ap^{(i)}$	`call psb_spmm(one,A,p,zero,q,desc_a,info)`
$\alpha_i = \rho_{i-1}/{p^{(i)}}^T q^{(i)}$	`sigma = psb_gedot(p,q,desc_a,info)`
	`alpha = rho/sigma`
$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$	`call psb_geaxpby(alpha,p,one,x,desc_a,info)`
$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$	`call psb_geaxpby(-alpha,q,one,r,desc_a,info)`
Check convergence:	
$\Vert r^{(i)}\Vert_2 \leq \epsilon \Vert b\Vert_2$	`rn2 = psb_genrm2(r,desc_a,info)`
	`bn2 = psb_genrm2(b,desc_a,info)`
	`err = rn2/bn2`
	`if (err.lt.eps) exit iterate`
end	`enddo iterate`

The Goal:

The code in the previous slide is:

- Object-oriented (even though methods don't always appear as such, but that's syntactic sugar);
- Evolvable to new machines;
- Parallel: uses MPI (even if you don't see it);
- Parallel: uses OpenMP and CUDA (even if you don't see it);
- Handles heterogeneous and hybrid nodes transparently.

To achieve this, we had to develop a (substantial) support infrastructure.

Sparse Matrices

Sparse matrices

A matrix is sparse when there are so many zeros that it pays off to take advantage of them in the computer representation. J. Wilkinson

We need to implement the product of a sparse matrix by a dense vector.

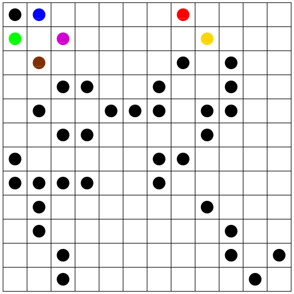
$$y \leftarrow \alpha A x + \beta y$$

Sparse matrix-vector product performs poorly . . . and that's a fact!

- Low ratio between floating point operations and memory accesses;
- High consumption of memory bandwidth
- Indirect addressing;
- Low spatial or temporal locality:

Data storage formats are essential! And many of them were invented over the years (more than you care to know).

Storage schemes: COO

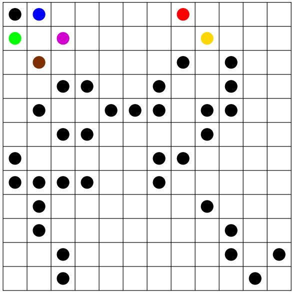


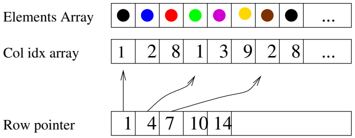
Elements Array	●	●	●	●	●	●	●	●	...
Col idx array	1	2	8	1	3	9	2	8	...
Row idx array	1	1	1	2	2	2	3	3	...

```
for i=1:nz
  ir = ia(i);
  jc = ja(i);
  y(ir) = y(ir) + as(i)*x(jc);
end
```

Cost: 5 memory reads, 1 write and 2 flops per iteration.

Storage schemes: CSR

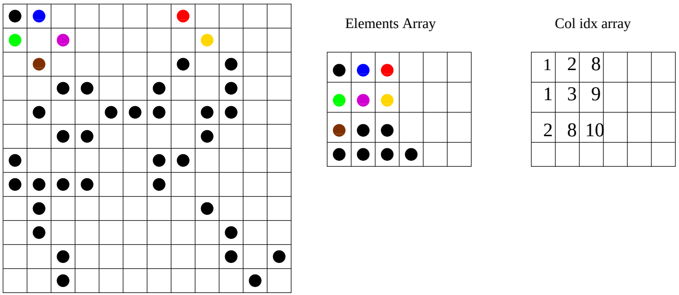




```
for i=1:m
   t=0;
   for j=irp(i):irp(i+1)-1
      t = t + as(j)*x(ja(j));
   end
   y(i) = t;
end
```

Cost: 3 memory reads and 1 write per outer iteration, 3 memory reads and 2 flops per inner iteration.

The ELLPACK (ELL) storage format



1	2	8			
1	3	9			
2	8	10			

```
for  i=1:m
   t=0;
   for  j=1:maxnzr
      t = t +  as(i,j)*x(ja(i,j));
   end
   y(i) = t;
end
```

Cost: 1 memory read and 1 write per outer iteration, 3 memory reads and 2 flops per inner iteration (also, regular access pattern).

DIA storage

Store dense diagonals:

```
for j=1:ndiag
  if (offset(j) > 0)
    ir1 = 1; ir2 = m - offset(j);
  else
    ir1 = 1 - offset(j); ir2 = m;
  end
  for i=ir1:ir2
    y(i) = y(i) + alpha*as(i,j)*x(i+offset(j));
  end
end
```

3 memory reads, 1 write, 2 or 3 flops per iteration, but no indirect addressing.

. . . and MANY more. . .

Base format	Issues	GPU variants
COO	Memory footprint, atomic data access	ALIGNED_COO [36] SCOO [25] BRO-COO [37] BCCOO and BCCOO+ [38]
CSR	Coalesced access, thread mapping, data reuse	CSR optimization [35, 22, 39, 40, 41, 42, 43, 44, 45, 46] SIC [47] RgCSR [48] BIN-CSR and BIN-BCSR [49] CMRS [50] PCSR [51] BCSR optimization [52, 24, 53, 54]
CSC	Coalesced access	CSC optimization [55]
ELLPACK	Zero padding	ELLPACK-R [29] Sliced ELLPACK [27] Warped ELL [56] ELLR-T [57] Sliced ELLR-T [58] BELLPACK [24] BSELLPACK [59] AdELL [60] ELLPACK-RP [61] BiELL and BiJAD [62] BRO-ELL [37] JAD optimization [63] BTJAD [64] Enhanced JDS [65] pJDS [26]
DIA	Zero padding	DDD-NAIVE and DDD-SPLIT [66] CDS [67]
Hybrid		HYB [35] Combination of CSR and ELL [68, 69] HEC [70] TILE-COMPOSITE [71] SHEC [72] Cocktail [59] HDC [73] Combination of ELLPACK and DIA [56] Combination of BCOO and BCSR [74] BRO-HYB [37]
New		BLSI [75] CRSD [76]

Are you unsure what to use?

Well, so are we.

Facts:

- Different computer architectures are best exploited by different formats;
- Different formats are suited to different operations (and we need them all);

Requirements (put your library developer's hat on):

- We want to be able to change in response to machine changes (might possibly be done at compile time, but annoying);
- We want to be able to change in response to usage requirements (need to change at run time)
- We need to switch among formats, some of them unkonwn at compile time;

We want maximum freedom, flexibility, maintainability and performance (i.e. we like to have our cake and eat it too)

Design Patterns

Design Patterns Basics

Design Patterns:

> *"Best practices" for solving programming problems (appearing in different application domains)*

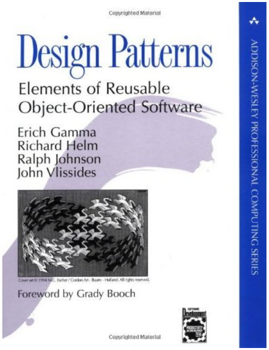
Why bother?

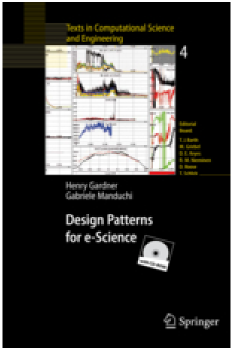
Elements of a software design pattern (Gamma et al. 1995):

1. The pattern name: a handle the describes a design problem. Its solution, and consequences in a word or two.
2. The problem: a description of when to apply the pattern and within what context.
3. The solution: the elements that constitute the design, the relationships between these elements, their responsibilities, and their collaborations.
4. The consequences: the results and trade-offs of applying the pattern.

Origins of Design Patterns

Software: "Gang of Four"; Scientific software: Gardner and Manduchi; Scientific Computing: Rouson, Xia and Xu.




Texts in Computational Science and Engineering
4
Henry Gardner
Gabriele Manduchi
Design Patterns for e-Science
Springer

Design Patterns to the Rescue

Meet our four knights:

- STATE;
- BUILDER;
- MEDIATOR;
- PROTOTYPE.

D. Barbieri, V. Cardellini, S. Filippone and D. Rouson, Springer LNCS 7155
S. Filippone and A. Buttari, ACM TOMS, 38(4), 2012
V. Cardellini, S. Filippone and D. Rouson, Scientific Programming, 2014

STATE design pattern

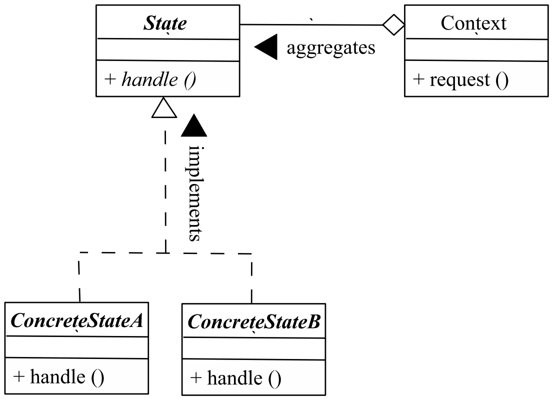
Often a seemingly simple representation problem for a set or mapping presents a difficult problem of data structure choice. Picking one data structure for the set makes certain operations easy, but others take too much time and it seems that there is no one data structure that makes all the operations easy. In that case the solution often turns out to be the use of two or more different structures for the same set or mapping.

Aho, Hopcroft & Ullmann, 1983.

STATE design pattern

Our sparse matrices need to be adjusted at runtime, so:

> *We want a polymorphic object to actually morph "before our very eyes"!*



STATE allows an object to (appear to) change its data type at runtime; this is not supported natively by any common OOP language.

STATE design pattern

Advantages:

- The internal storage is not directly visibile to the code using the object; thus, it is free to evolve without impact;
- The internal storage is dynamic: if your application goes through phase A and then phase B, the storage may change to whatever works best in each phase;
- You can easily add new variations;
- You can easily have different types in different processes (aka: heterogeneous computing).

Disadvantages:

- Every invocation involves one more level of indirection; if done right it's practically unnoticeable.

BUILDER, MEDIATOR, PROTOTYPE

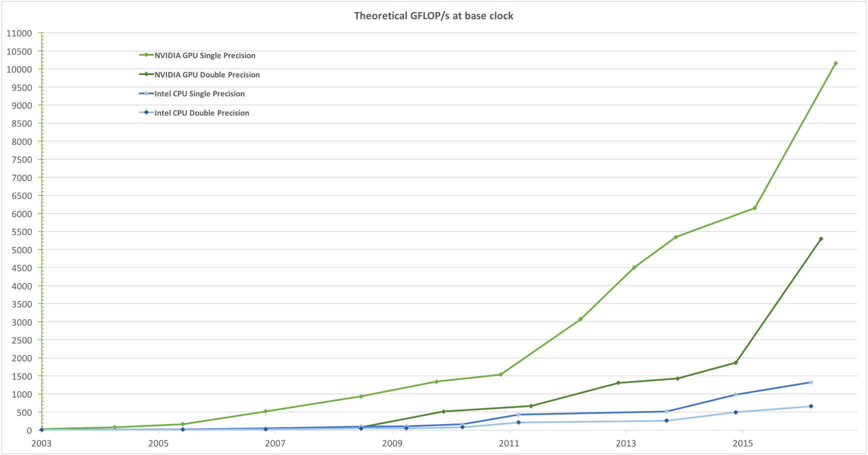
All non-trivial data structures are impossible with one-step constructors. Hence we need

BUILDER:	Unifies the *process* of instantiating a complex object (by defining a buildup strategy);
MEDIATOR:	Allows any format to convert to any other with a linear amount of code by routing through a common format (from fully-connected to star topology);
PROTOTYPE:	Allows existing code to instantiate new classes that were unknown at the time of writing.

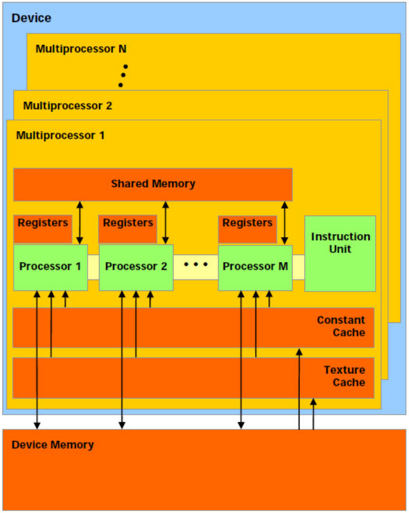
Further details in: V. Cardellini, S. Filippone and D. Rouson, Scientific Programming, 2014

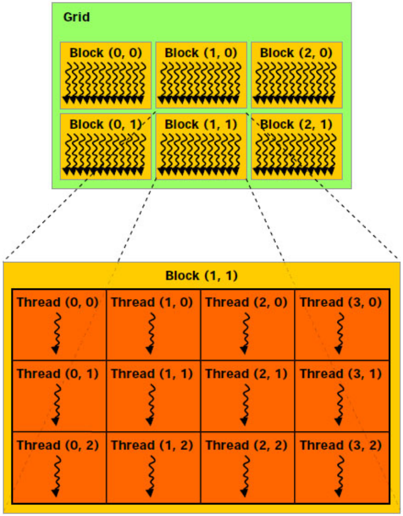
Case Study: Techniques for GPUs

Motivations



NVIDIA GPU architecture





Sparse matrix format: ELL-G

ELLPACK amenable to use on GPUs, provided you handle memory accesses in the proper way

Thus: use a variation on ELL with:

- Align and pad sizes to 16/32, so that accesses are coalesced
- Extra row-size array (mix with CSR)

Resulting code kernel with 1 thread per matrix row.

GPU Storage Format: ELL-G

```
Dspmvm_gpu_krn (double *y, double alpha, double* cM, int* rP,int* rS,
                int n, int pitch, double *x, double beta, int firstIndex)
{
  int i = threadIdx.x + blockIdx.x * (THREAD_BLOCK);
  if (i >= n)
    return;
  double y_prod = 0.0;
  int row_size = rS[i];

  rP += i;
  cM += i;
  for (int j = 0; j < row_size; j++)    {
    int pointer = rP[0] - firstIndex;
    double value = cM[0];
    rP += pitch;
    cM += pitch;
    y_prod += __dmul_rn (value, x[pointer]);
  }
  if (beta == 0.0)
    y[i] = (alpha * y_prod);
  else
    y[i] = __dmul_rn (beta, y[i]) + __dmul_rn (alpha, y_prod);
}
```

Variation: HLL and HDIA

Problem with ELL:

If one row is much longer than the others, padding will require a large amount of additional memory

Problem with DIA:

A lone coefficient will require storing one diagonal full of zeros

Memory is a precious resource!

Solution: HLL and HDIA

similar to ELL and DIA, but:

Store coefficients in the desired formats for stripes of the matrix, limit each stripe to HK rows; therefore, padding is limited.

Matrices/vectors on GPU

Need to handle data structure on “device” side for both matrices and vectors

Using the framework based on the STATE, BUILDER and PROTOTYPE patterns we can hide necessary data movement behind format conversion methods.

Vector design:

- Vectors have dual memory: on host and device side;
- Vector data change much more frequently than matrix data;
- Whenever a vector is touched, make device side up-to-date and execute on the GPU (“attractor”);
- Get out of the GPU only upon implicit/explicit request

Performance data

Performance data

Well, we first begin with *human* performance data, i.e. development time for ELL-G:

- Develop the CUDA kernels (easy: grad students are cheap. On further reflection, *good* grad student may be cheap but not easy to find . . .)
- Needed to write the wrappers around CUDA code, adapting the existing ELL code: about a day;
- Glueing the CUDA code and running the initial tests took about half a day;
- Repeating the wrapping process to interface the NVIDIA CUSPARSE library took all of an additional day.

Similar considerations apply to the other formats: each one has a marginal cost order of 1 day.

Performance data — Memory footprint MB

Matrix name	DIA	HDIA	ELL	HLL	HYB
cant	49.5	49.9	58.7	54.9	48
mac_econ_fwd500	844	206	109	56	16.1
olafu	153.4	17.3	17.3	14.3	12.2
raefsky2	18.8	5.4	4.2	3.7	3.5
af23560	6.2	5.54	6	6	5.9
mhd4800a	2.2	2.2	1.9	1.9	1.2
bcsstk17	91.6	12.2	19.8	6.9	5.1
lung2	1318	25	11	10.2	6.3
af_1_k101	3614	233.2	213.5	213.5	212.62
af_2_k101	3614	233.2	213.5	213.5	212.62
af_3_k101	3614	233.2	213.5	213.5	212.62
af_4_k101	3614	233.2	213.5	213.5	212.62
af_5_k101	3614	233.2	213.5	213.5	212.62

Performance data — Memory footprint MB

Matrix name	DIA	HDIA	ELL	HLL	HYB
FEM_3D_thermal1	163	4.2	5.9	5.9	5.2
FEM_3D_thermal2	10744	35.6	48.51	48.53	42.5
Cube_Coup_dt0	1098802	3306	1775	901	—
ML_Laplace	14161	334	336	336	333.8
StocF-1465	858380	1606	3329	311	258
thermal1	21499	75.1	11.2	8.9	7.2
thermal2	6543314	1210	167	132.1	107.9
thermomech_dK	455211	472	49.8	40	35
thermomech_dM	212576	160	25.3	20.6	17.9
thermomech_TC	106305	157.6	12.7	10.5	8.9
thermomech_TK	106305	157.6	12.7	10.5	8.9
DK01R	0.245	0.21	0.192	0.191	0.144
GT01R	12.7	5.1	8.6	6.2	5.2
PR02R	4973	142.6	178	128	98.8
RM07R	874967	1206	1352	818	451.1
nlpkkt80	652900	272	361	347	349
nlpkkt120	4897552	888.3	1204	1160	—
pde50	7	7	11	10	10.8
pde60	12.1	12.2	19	18.9	18.7
pde80	28.7	29	45	44.8	44.6
pde90	40.8	41.3	64.1	63.9	63.6
pde100	56	56.7	88	87.6	87.3

Performance data — Matrix-Vector Product GFLOPS

Platform 1: AMD 7750 dual-core, NVIDIA GTX 285

Matrix name	DIA	HDIA	ELL	HLL	HYB
cant	11.67	12.30	13.24	12.93	12.36
mac_econ_fwd500	—	1.90	5.14	4.55	4.99
olafu	0.92	7.78	14.86	14.50	8.97
raefsky2	1.96	2.80	7.18	6.98	2.43
af23560	10.95	11.23	14.97	14.02	13.29
mhd4800a	6.64	5.47	6.46	6.06	2.33
bcsstk17	0.68	4.48	6.72	6.63	5.90
lung2	—	2.57	8.80	7.98	5.19
af_1_k101	—	—	19.51	18.41	18.95
af_2_k101	—	—	19.51	18.41	18.95
af_3_k101	—	—	19.51	18.41	18.95
af_4_k101	—	—	19.51	18.41	18.95
af_5_k101	—	—	19.51	18.41	18.99
FEM_3D_thermal1	0.61	10.85	12.45	11.84	11.61
FEM_3D_thermal2	—	—	13.24	12.81	12.98
Cube_Coup_dt0	—	—	—	—	—
ML_Laplace	—	—	16.07	15.76	15.91
StocF-1465	—	—	—	8.04	10.86

Performance data — Matrix-Vector Product GFLOPS

Platform 1: AMD 7750 dual-core, NVIDIA GTX 285

Matrix name	DIA	HDIA	ELL	HLL	HYB
thermal1	—	—	9.28	8.58	6.72
thermal2	—	—	8.78	8.83	7.71
thermomech_dK	—	—	6.14	5.73	5.00
thermomech_dM	—	—	6.14	5.73	5.00
thermomech_TC	—	—	8.82	8.34	8.00
thermomech_TK	—	—	8.47	8.04	7.45
DK01R	0.83	0.82	1.27	1.14	0.35
GT01R	6.43	9.48	9.13	8.96	5.37
PR02R	—	—	12.67	12.70	12.72
RM07R	—	—	—	—	—
nlpkkt80	—	—	17.25	17.20	17.14
nlpkkt120	—	—	—	—	—
pde50	16.58	13.74	15.14	14.12	14.45
pde60	17.23	14.63	16.14	15.34	15.37
pde80	17.94	15.67	17.41	17.61	16.05
pde90	17.67	15.69	16.93	16.59	16.29
pde100	17.85	15.76	17.19	16.95	16.39

Performance data — Matrix-Vector Product GFLOPS

Platform 2: Intel Xeon X5650, NVIDIA Tesla C2050

Matrix	DIA	HDIA	ELL	HLL	HYB
cant	13.63	13.04	11.83	11.59	12.35
mac_econ_fwd500	0.27	1.03	3.53	3.53	4.74
olafu	1.17	7.40	11.04	10.49	10.65
raefsky2	2.00	2.92	10.21	10.06	4.89
af23560	11.84	10.73	12.28	12.20	11.94
mhd4800a	5.30	5.11	6.42	5.81	3.65
bcsstk17	0.80	4.12	7.29	7.16	7.57
lung2	0.09	3.03	6.58	6.54	5.50
af_1_k101	0.90	12.44	14.15	14.14	15.16
af_2_k101	0.90	12.44	14.15	14.14	15.16
af_3_k101	0.90	12.44	14.15	14.14	15.16
af_4_k101	0.90	12.44	14.15	14.14	15.16
af_5_k101	0.90	12.45	14.15	14.14	15.16
FEM_3D_thermal1	0.61	12.82	11.37	14.14	13.49
FEM_3D_thermal2	—	15.25	11.56	10.75	13.07
Cube_Coup_dt0	—	6.60	10.56	10.69	—
ML_Laplace	—	14.21	14.31	14.24	14.83
StocF-1465	—	2.26	9.40	9.19	10.66

Performance data — Matrix-Vector Product GFLOPS

Platform 2: Intel Xeon X5650, NVIDIA Tesla C2050

Matrix	DIA	HDIA	ELL	HLL	HYB
thermal1	—	1.30	8.17	7.95	6.90
thermal2	—	1.26	7.74	7.64	7.00
thermomech_dK	—	0.87	7.88	7.78	7.75
thermomech_dM	—	1.31	5.77	5.70	6.08
thermomech_TC	—	0.77	5.38	5.35	5.09
thermomech_TK	—	0.77	5.38	5.35	5.09
DK01R	0.61	0.61	2.58	2.47	0.68
GT01R	7.74	9.91	10.11	9.99	7.70
PR02R	—	9.63	10.48	10.43	10.50
RM07R	—	5.18	8.34	8.44	9.08
nlpkkt80	—	15.62	13.14	13.10	15.05
nlpkkt120	—	15.00	12.97	12.94	14.01
pde50	15.84	14.79	11.83	11.96	12.43
pde60	16.37	15.43	12.02	12.17	12.93
pde80	16.76	16.02	12.05	12.37	13.30
pde90	16.58	16.04	11.94	12.27	13.16
pde100	16.29	15.95	11.78	12.18	13.37

Performance data — Matrix-Vector Product GFLOPS

Platform 3 AMD FX 8120, GTX 660

Matrix name	DIA	HDIA	ELL	HLL	HYB
cant	17.1	16.9	15.34	14.73	14.7
mac_econ_fwd500	0.3	1.3	4.9	4.42	5.6
olafu	1.6	11.3	14.9	13.9	11.96
raefsky2	2.7	4.6	12.6	12.3	4.5
af23560	13.5	15.2	15.2	14.47	14.4
mhd4800a	5.8	5.8	8.0	7	3.7
bcsstk17	0.9	6.4	9.45	8.19	7.0
lung2	0.1	3.8	8.0	7.46	5.1
af_1_k101	—	15.9	18.3	18.1	17.8
af_2_k101	—	15.9	18.3	18.1	17.8
af_3_k101	—	15.9	18.3	18.1	17.8
af_4_k101	—	15.9	18.3	18.1	17.8
af_5_k101	—	15.9	18.3	18.1	17.8
FEM_3D_thermal1	0.7	15.62	14.0	13.65	14.8
FEM_3D_thermal2	—	19.3	14.6	14.34	15.4
Cube_Coup_dt0	—	—	—	10.7	—
ML_Laplace	—	18.7	18.3	18.44	18.0
StocF-1465	—	2.2	—	12.32	13.7

Performance data — Matrix-Vector Product GFLOPS

Platform 3 AMD FX 8120, GTX 660

Matrix name	DIA	HDIA	ELL	HLL	HYB
thermal1	—	1.7	10.2	9.9	8.5
thermal2	—	1.6	9.27	9.54	8.3
thermomech_dK	—	1.1	9.0	10.17	8.4
thermomech_dM	—	1.56	6.2	6.6	6.5
thermomech_TC	—	0.92	6.36	6.39	6.5
thermomech_TK	—	0.92	6.36	6.39	6.5
DK01R	0.7	0.644	2.1	2.07	0.58
GT01R	9	13.1	12.0	12.64	8.7
PR02R	—	12.8	13.4	13.74	13.6
RM07R	—	6.9	6.5	12.3	11.0
nlpkkt80	—	21.28	16.7	16.8	16.7
nlpkkt120	—	21.5	16.6	16.7	—
pde50	17.6	17.4	15.0	14.5	14.3
pde60	19.5	18.6	15.5	15	14.7
pde80	19.3	18.43	15.0	14.7	14.9
pde90	18.3	17.6	14.4	14	14.6
pde100	17.4	17.16	13.7	13.7	13.3

Performance data — Matrix-Vector Product GFLOPS

Multiple threads per row, platform 1: AMD Athlon(tm) 7750 – GTX 285

Matrix	ELL-G		HLL-G	
	1	2	1	2
cant	13.24	14.57	12.93	13.92
mac_econ_fwd500	5.14	4.74	4.55	5.15
olafu	14.86	14.29	14.50	13.18
raefsky2	7.18	11.73	6.98	9.82
af23560	14.97	14.81	14.02	12.10
mhd4800a	6.46	8.07	6.06	6.79
bcsstk17	6.72	9.64	6.63	8.69
lung2	8.80	8.53	7.98	6.80
af_1_k101	19.51	20.38	18.41	20.22
af_2_k101	19.51	20.48	18.41	20.22
af_3_k101	19.51	20.48	18.41	20.22
af_4_k101	19.51	20.49	18.41	20.22
af_5_k101	19.51	20.49	18.41	20.22
FEM_3D_thermal1	12.45	11.76	11.84	10.35
FEM_3D_thermal2	13.24	13.78	12.81	13.55
ML_Laplace	16.07	17.54	15.76	17.50
StocF-1465	0.00	0.00	8.04	9.06

Performance data — Matrix-Vector Product GFLOPS

Multiple threads per row, platform 3, AMD FX 8120, GTX 660

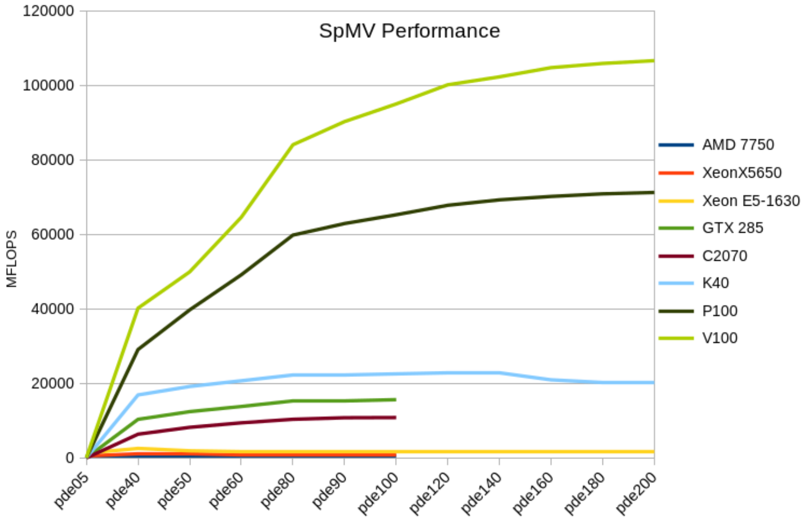
Matrix	ELL-G		HLL-G		ELLRT		
	1	2	1	2	1	2	4
thermal1	10.2	8.4	9.9	8.6	10	7.6	7.4
thermal2	9.27	7.8	9.54	7.87	9.27	7.2	7.5
thermomech_dK	8.96	8.76	10.17	9.8	8.7	9.16	7.6
thermomech_dM	6.25	5.8	6.6	6.3	6.25	6	6.2
thermomech_TC	6.36	6.5	6.39	6.8	6.7	5.7	6.5
thermomech_TK	6.36	6.5	6.39	6.8	6.7	5.7	6.5
DK01R	2.1	2.07	2.07	2.07	2.36	2.39	2.34
GT01R	12.66	12	12.64	12	12	10.47	11.63
PR02R	13.45	13.71	13.74	13.72	13.86	13.63	13.16
RM07R	6.46	12	12.3	12.25	6.66	6.78	8.2
nlpkkt80	16.65	17.4	16.78	16.9	16.47	15.1	12.17
nlpkkt120	16.63	16.1	16.66	15.59	17	16.6	15.22
pde50	15	11.73	14.5	11	14.53	9.6	8.98
pde60	15.5	11.78	15	10.7	15.02	9.6	8.85
pde80	15	11.58	14.7	10.14	15.18	9.1	8.81
pde90	14.4	11.24	14	10.3	14.78	9.5	8.82
pde100	13.7	11.2	13.7	10.45	13.7	9.8	8.8

Performance data — Matrix-Vector Product GFLOPS

Multiple threads per row, platform 3, AMD FX 8120, GTX 660

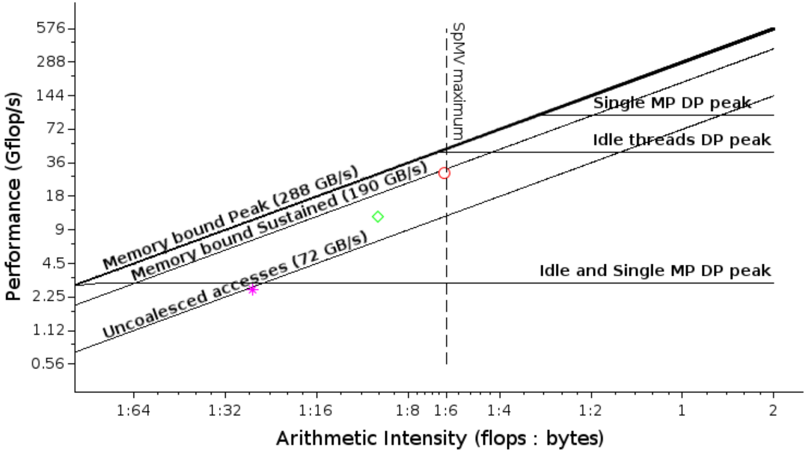
Matrix	ELL-G		HLL-G		ELLRT		
	1	2	1	2	1	2	4
cant	15.34	16.14	14.73	16.32	15	16.2	16.4
mac_econ_fwd500	4.9 2	4.61	4.44	5	4.5	4.42	4.65
olafu	14.9	14.37	13.9	14.7	14.2	14.4	14.5
raefsky2	12.64	15.63	12.3	15.9	9.85	15.12	14.38
af23560	15.23	15.29	14.47	14.87	14.77	14.31	12.56
mhd4800a	8	8.17	7	7.76	7.48	6.7	7.9
bcsstk17	9.45	10.6	8.19	11	8.5	10.5	11.2
lung2	7.97	7.8	7.46	7	7.5	6.32	5.95
af_1_k101	18.3	18.8	18.1	18.84	18.3	17.6	16.8
af_2_k101	18.3	18.8	18.1	18.84	18.3	17.6	16.8
af_3_k101	18.3	18.8	18.1	18.84	18.3	17.6	16.8
af_4_k101	18.3	18.8	18.1	18.84	18.3	17.6	16.8
af_5_k101	18.3	18.8	18.1	18.84	18.3	17.6	16.8
FEM_3D_thermal1	14	13.7	13.65	13.36	14.33	13.7	12.63
FEM_3D_thermal2	14.6	15.23	14.34	15.2	14.76	14.72	13.8
Cube_Coup_dt0	—	—	10.7	11.2	10.7	11.51	11.7
ML_Laplace	18.3	19	18.44	19.2	18.6	18.78	17.87
StocF-1465	-	-	12.32	12.50			

Performance of SpMV



For a detailed analysis see: S. Filippone, V. Cardellini, A. Fanfarillo, D. Barbieri, ACM TOMS, 2017

How good are those numbers?



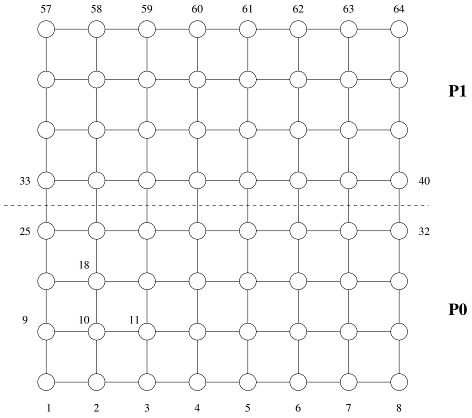
S. Filippone, V. Cardellini, A. Fanfarillo, D. Barbieri, ACM TOMS, 2017

Parallel Matrix-Vector Product

From single-node to parallel matrix-vector product

For each sparse matrix-vector product we need an essential communication kernel:

A (sparse) variable all-to-all collective, also known as "halo exchange"



From single-node to parallel matrix-vector product

From a logical point of view we need to:

- Gather into a buffer y coefficients to be sent:

```
for(i=0;i<n;i++)
    y[i] = x[index[i]];
```

- Scatter from a buffer y coefficients that have been received:

```
for(i=0;i<n;i++)
    x[index[i]] = y[i] +
         beta*x[index[i]];
```

How do you go about sending your data?

Possible solutions:

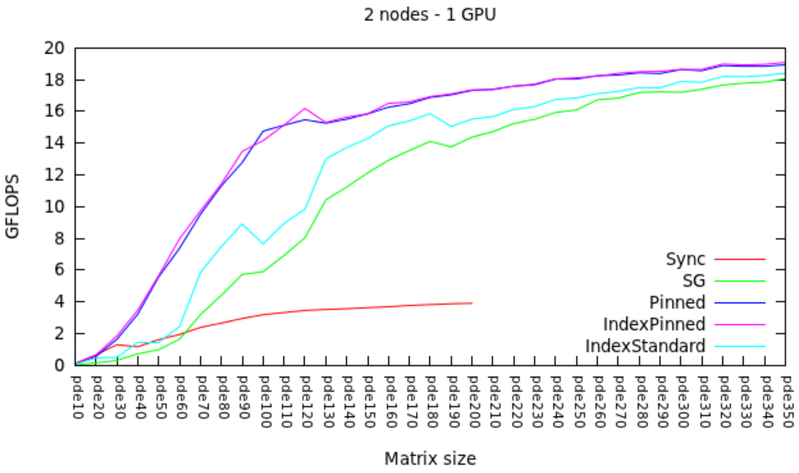
1. Brute force: sync vector x between host and device, then use CPU send/receive;
2. GPU-enabled gather/scatter kernels;
3. Pinned memory gather/scatter;
4. MPI with direct CUDA support and MPI_Type_Indexed

Last option very attractive given the interfacing of GPUDirect-CUDA/6.0 with latest OpenMPI versions Platforms:

1. CINECA: dual-socket exa-core Intel Xeon X5650 48 GB RAM; two NVIDIA Fermi S2050, QDR InfiniBand (40 Gbit/s).
2. AWS: dual socket Intel Xeon X5570, quad-core with hyperthread, two NVIDIA Tesla M2050 GPUs, 10 Gbit/s network.

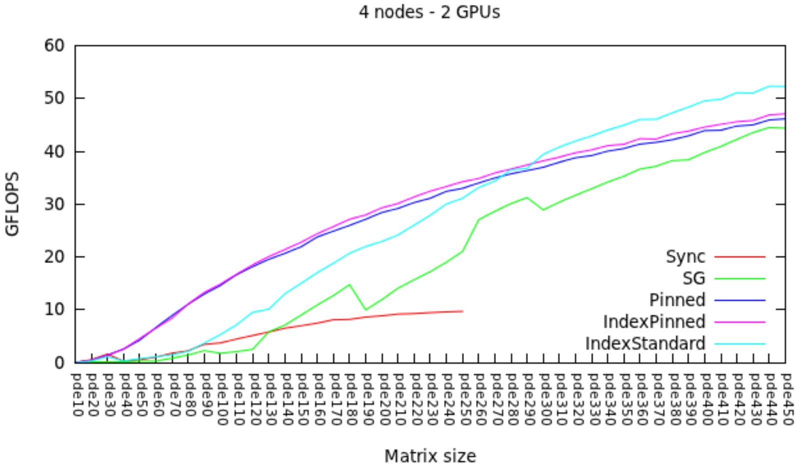
Performance results 2-1

CINECA platform:



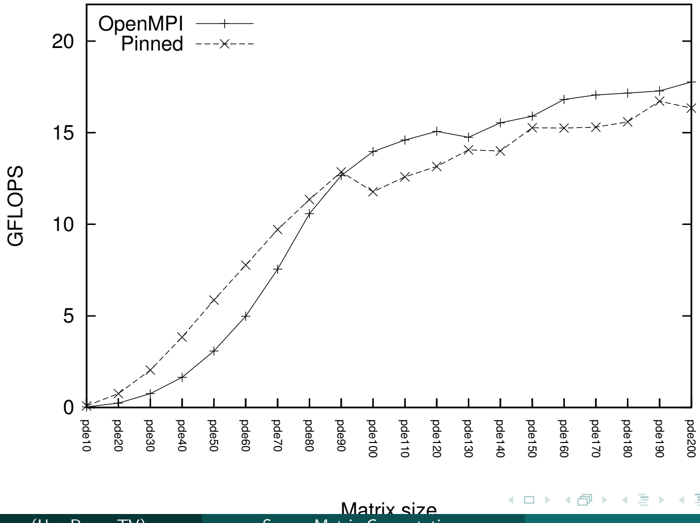
Performance results 2-1

CINECA platform:



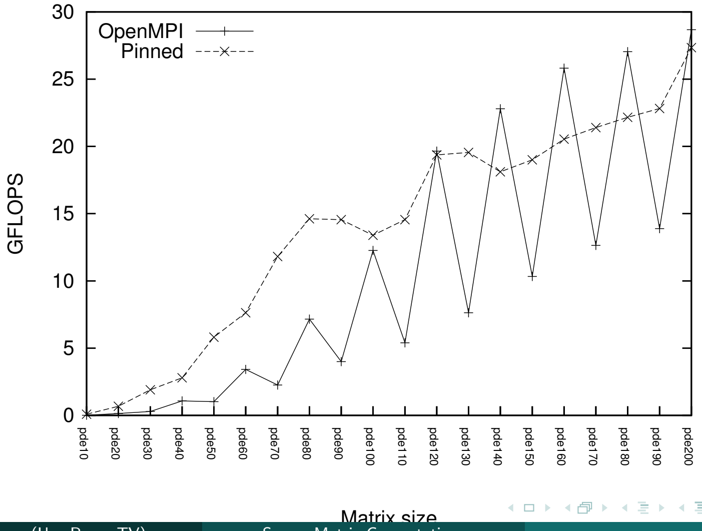
Performance results: Pinned memory vs. OpenMPI/CUDA

AWS platform (2 nodes / 1 GPU):



Performance results: Pinned memory vs. OpenMPI/CUDA

AWS platform (2 nodes - 2 GPU):



Preconditioned iterations

Preconditioners

A preconditioner is a transformation M

$$Ax = b \Leftrightarrow M^{-1}Ax = M^{-1}b,$$

such that:

1. M can be computed easily;
2. M^{-1} can be applied easily;
3. $M^{-1}A \approx I$.

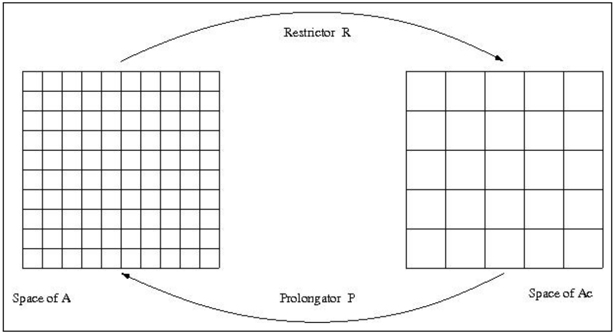
These requirements are contradicting each other! But if you can find a good compromise, you'll get convergence in $N_{it} \ll n$ iterations at an affordable cost T_{it}, where time to solution is:

$$T_{tot} = T_{prec} + N_{it} \times T_{it}$$

Often: no convergence without preconditioner.

Domain decomposition: multilevel corrections

Basic idea for much more sophisticated preconditioners:



Project the problem onto a new space with a coarser discretization, solve approximately, then project back onto the original fine space to correct the (fine level) approximate solution. Algebraic Multigrid: extension to a purely "topological" framework, i.e. no assumptions on the underlying geometry.
Works very well for elliptic PDEs.

Multilevel preconditioners

Multilevel preconditioner: an array of 1-level entities with

- The current level aggregate matrix;
- The (linear) map between levels;
- The smoother, which in turn contains:
- The subdomain solver;

Note: the nested object hierarchy means smoothers & solvers dynamic while at the same time the outer preconditioner type is stable and well-defined!

Approximate inverses

Many preconditioners based on incomplete factorizations:

$$A \approx LU;$$

these require sparse triangular system solve kernels. What if we seek a matrix $G = M^{-1}$ directly approximating the inverse of A ?
Possible strategies:

- Approx Biconjugation
- Inversion of incomplete factors

$$A^{-1} \approx ZD^{-1}W^T$$

Require sparse matrix-vector multiplication for application, which is good because *Solving sparse triangular systems on the GPU is very hard!*
Implemented in AMG4PSBLAS: hierarchy of nested object, i.e. we naturally have STATE and PROTOTYPE patterns.

Experimental setup

- Platform: AMD Athlon 7750, GTX 285, or, PLX cluster, Intel Westmere and NVIDIA Tesla M2070;
- Data distribution: graph partitioning via METIS;
- For hybrid cases:
  - non-uniform weight of partitions;
  - inner solver AINV on GPU, ILU on CPU;
- Compilers: GNU 4.9 (development)/cuda 5.0 on first platform, GNU 4.7.3/cuda 4.2.9 on second;
- Test case: a 2D/3D convection-diffusion PDE, centered finite differences

$$\begin{aligned} -\nu\nabla\cdot(a(x)\nabla u) + q(x)\cdot\nabla u &= 0, \qquad & x &\in \Omega, \\ u &= g, \qquad & x &\in \partial\Omega. \end{aligned}$$

ILU on the GPU? Bad idea!

Platform: AMD Athlon 7750, GTX 285;

Case		$ILU(0)$ CPU		$ILU(0)$ GPU CSR		$ILU(0)$ GPU *	
	tpr	it	tslv	it	tslv	it	tslv
pde0300	0.08	145	1.80	144	5.03	145	1.61
pde0400	0.09	201	4.62	201	10.11	202	3.82
pde0500	0.16	255	9.63	255	17.32	255	7.71
pde0600	0.20	312	16.43	310	26.34	313	13.50
pde0700	0.30	367	26.17	366	38.00	362	20.47
pde0800	0.36	416	38.61	416	52.10	415	30.05
pde0900	0.51	468	54.98	469	68.95	471	42.26
pde1000	0.58	524	76.15	539	92.91	530	58.67
pde1100	0.76	586	102.99	570	113.79	594	78.92

Using NVIDIA CuSPARSE 4.1: on 2D we actually get a slowdown, best speedup by (Naumov, 2011) around 2.

Simple preconditioner tests, 2D

Platform: AMD Athlon 7750, GTX 285;

Case	CPU (ILU)				GPU (AINV)			
	NP=1		NP=2				+1 CPU	
	it	tslv	it	tslv	it	tslv	it	tslv
pde0100	44	0.04	50	0.03	69	0.03	68	0.04
pde0200	100	0.55	102	0.50	151	0.10	152	0.12
pde0300	150	1.95	151	1.95	267	0.45	254	0.46
pde0400	198	4.69	205	5.88	377	0.84	352	0.81
pde0500	243	9.21	254	7.69	569	1.62	459	1.31
pde0600	287	15.33	312	15.74	600	2.19	453	1.70
pde0700	346	25.93	358	20.88	794	3.67	566	2.73
pde0800	392	37.12	414	30.79	607	3.69	711	4.52
pde0900	424	51.45	459	42.47	578	3.91	623	4.67
pde1000	482	73.77	514	60.06	658	5.33	742	6.43
pde1100	538	100.06	578	85.30	704	6.79	751	7.75

Multilevel preconditioner tests, 2D

Platform: AMD Athlon 7750, GTX 285; exponential force term.

Case	CPU				GPU			
	NP=1		NP=2				+1 CPU	
	it	tslv	it	tslv	it	tslv	it	tslv
pde0100	3	0.01	4	0.01	9	0.04	8	0.07
pde0200	4	0.08	4	0.06	6	0.05	7	0.11
pde0300	4	0.20	5	0.20	7	0.13	9	0.19
pde0400	4	0.36	5	0.41	8	0.20	9	0.22
pde0500	4	0.57	5	0.68	10	0.40	10	0.29
pde0600	6	1.20	5	0.99	12	0.66	14	0.54
pde0700	5	1.36	5	1.25	9	0.68	11	0.59
pde0800	5	1.77	6	1.89	11	1.07	11	0.68
pde0900	5	2.26	6	2.38	13	1.53	13	0.89
pde1000	6	3.30	6	2.94	14	1.99	14	1.07
pde1100	6	4.31	6	3.57	12	2.08	16	1.38

Multilevel preconditioner tests, 3D

Platform: AMD Athlon 7750, GTX 285;

Case	CPU				GPU			
	NP=1		NP=2				+1 CPU	
	it	tslv	it	tslv	it	tslv	it	tslv
pde020	3	0.01	3	0.01	6	0.02	6	0.06
pde030	3	0.06	4	0.08	8	0.06	8	0.09
pde040	4	0.20	4	0.23	10	0.12	10	0.14
pde050	5	0.51	5	0.60	11	0.23	12	0.24
pde060	5	0.89	5	1.06	13	0.47	14	0.45
pde070	5	1.44	5	1.67	14	0.73	13	0.53
pde080	6	2.64	6	2.85	15	1.33	15	0.84
pde090	6	3.70	6	4.09	16	1.68	18	1.36
pde100	7	5.87	7	6.45	18	2.49	19	1.89

Multilevel preconditioner tests, 2D

Platform: PLX (Intel Westmere, Tesla M2070)

Case	CPU				GPU			
	NP=1		NP=2				+1 CPU	
	it	tslv	it	tslv	it	tslv	it	tslv
pde0100	4	0.01	5	0.00	8	0.04	8	0.12
pde0200	4	0.04	5	0.02	8	0.08	8	0.18
pde0300	5	0.12	6	0.07	9	0.16	10	0.27
pde0400	5	0.22	6	0.13	11	0.28	12	0.36
pde0500	7	0.48	7	0.24	15	0.51	13	0.46
pde0600	6	0.61	6	0.30	12	0.56	12	0.55
pde0700	5	0.70	6	0.41	9	0.56	10	0.56
pde0800	7	1.29	6	0.54	10	0.79	12	0.70
pde0900	7	1.66	6	0.69	10	0.98	17	1.08
pde1000	7	2.08	6	0.86	12	1.42	19	1.34
pde1100	6	2.13	6	1.05	12	1.70	18	1.35

Issues and directions

- ✓ We are able to handle new storage formats transparently;
- ✓ Easy to experiment with multiple combinations of solvers;
- ✗ Finding the right load balance is *very* tricky and problem-dependent;
- ✗ GPU performance on sparse kernels is much less than we would like;
- ✗ Moving data is horribly expensive, and seems to be *worsening* with time;
- ? Robustness of preconditioning strategies over various problem classes?
- ? Shift from traditional Krylov methods to new variants? Lots of work under way by e.g. Demmel, Grigori, Meerbergen,. . .

We currently have many more questions than answers: good, means we're not out of work!

Bibliography — research

- Valeria Cardellini, Salvatore Filippone and Damian Rouson Design Patterns for sparse-matrix computations on hybrid CPU/GPU platforms, Scientific Computing, 2014.
- D. Barbieri, V. Cardellini, S. Filippone and D. Rouson: Design Patterns for Scientific Computations on Sparse Matrices, Springer LNCS 7155
- A. Buttari, D. di Serafino, P. D'Ambra, S. Filippone: 2LEV-D2P4: a package of high-performance preconditioners, Applicable Algebra in Engineering, Communications and Computing, 18(3), 2007
- A. Buttari, V. Eijkhout, J. Langou and S. Filippone: Performance optimization and modeling of sparse kernels, International Journal of High Performance Computing Applications, 21(4), 2007
- P. D'Ambra, S. Filippone, D. di Serafino: On the Development of PSBLAS-based Parallel Two-level Schwarz Preconditioners, Applied Numerical Mathematics, 57(11-12), 2007
- P. D'Ambra, S. Filippone, D. di Serafino: MLD2P4: A Package of Parallel Algebraic Multilevel Domain Decomposition Preconditioners in Fortran 95, *ACM Trans. Math. Softw.*, 37(3), 2010.
- S. Filippone and A. Buttari:Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003, *ACM Trans. Math. Softw.*, 38(4), 2012
- S. Filippone and M. Colajanni: PSBLAS: A Library for Parallel Linear Algebra Computation on Sparse Matrices, *ACM Trans. Math. Softw.*, 26(4):527–550, 2000.

Bibliography — general

- R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst.
  Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.
  SIAM, Philadelphia, PA, 1994.
- Jack Dongarra, Iain Duff, Danny Sorensen, and Henk van der Vorst.
  Numerical Linear Algebra for High-Performance Computers.
  SIAM, Philadelphia, PA, 1998.
- Gene Golub and Charles F. Van Loan.
  Matrix Computations.
  Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.
- Yousef Saad.
  Iterative Methods for Sparse Linear Systems.
  SIAM, Philadelphia, PA, 2nd edition, 2003.

Thank you.

STATE design pattern

State pattern in Fortran 2003: the outer type holds an allocatable polymorphic scalar component. Every outer method is mapped onto an inner method:

```fortran
type :: psb_d_base_sparse_mat
   ! data components here
contains
  procedure, pass(a) :: foo => inner_foo
end type psb_d_base_sparse_mat

subroutine inner_foo(a)
  class(psb_d_base_sparse_mat) :: a
   ! foobar
end subroutine inner_foo

type :: psb_dspmat_type
  class(psb_d_base_sparse_mat), allocatable :: a
contains
  procedure, pass(a) :: foo => outer_foo
end type psb_dspmat_type

subroutine outer_foo(a)
  class(psb_dspmat_type) :: a
  call a%a%foo()
end subroutine outer_foo
```

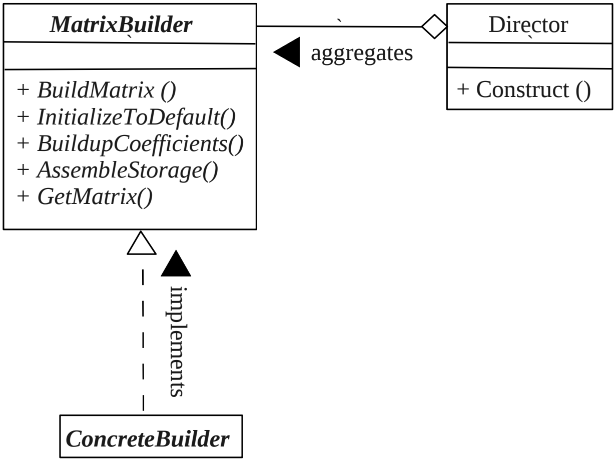
BUILDER design pattern

Does that mean we (users) have to learn how to build from scratch each and every data structure? It is impossible to have a one-step constructor for all but the most trivial cases.
Enters the BUILDER pattern:

1. Initialize to some default;
2. Add sets of coefficients by calling buildup methods in a loop;
3. When finished, invoke a post-processing method to bring to the desired STATE.

In other words, provide a unified process to instantiate the object.

BUILDER design pattern



MEDIATOR design pattern

Ok, we've seen how you build a sparse matrix. We want to switch to a new format (which, by the way, also happens internally at assembly time)

```
call a%cscnv(info,afmt='XYZ')
```

So for evey conceivable pair of formats `XYZ` and `ABC` you provide a converter. Or do you?

MEDIATOR design pattern

Ok, we've seen how you build a sparse matrix. We want to switch to a new format (which, by the way, also happens internally at assembly time)

```
call a%cscnv(info,afmt='XYZ')
```

So for evey conceivable pair of formats `XYZ` and `ABC` you provide a converter. Or do you?
Problems:

- The set of formats may grow over time, you'd have to go back and fix the old ones;
- With N formats you end up with $N(N-1) = O(N^2)$ converters.

Clearly impractical. Enters MEDIATOR.

MEDIATOR design pattern

Basic idea very simple: move from a fully connected graph to a star graph (or: from $O(N^2)$ to $2 \times N$), i.e. route all conversions through a common object.
In our case the COO storage format takes the role of MEDIATOR;

MEDIATOR design pattern

Basic idea very simple: move from a fully connected graph to a star graph (or: from $O(N^2)$ to $2 \times N$), i.e. route all conversions through a common object.

In our case the COO storage format takes the role of MEDIATOR;

```
subroutine d_cp_fmt_to_fmt(a,b,info)
  class(psb_d_base_sparse_mat), intent(in) :: a
  class(psb_d_base_sparse_mat), intent(out)  :: b
  integer, intent(out)                         :: info
  type(psb_d_coo_sparse_mat) :: tmp
  call a%cp_to_coo(tmp,info)
  if (info == 0) call b%cp_from_coo(tmp,info)
  call tmp%free()
end subroutine d_cp_fmt_to_fmt
```

Note that this bit of code does not need to know the exact dynamic type of `A` and `B`, as long as they provide `cp_to_coo` and `cp_from_coo`.

Note: shortcuts may be added if/when warranted.

MEDIATOR design pattern

It also makes sense to use our mediator COO as the initial state for any sparse matrix, because:

- It is easy to add coefficients to a COO matrix during BUILDer (just queue them);
- It is easy to apply a renumbering (just apply renumbering to the individual entries in `IA` and `JA`).

So, COO is not just our mediator, it is also the default for interacting with the “external” world (i.e. the rest of the application).
Example: in finite elements you often proceed element by element, producing a “cloud” of coefficients in a rather unordered fashion, and jumping back and forth through matrix rows. Easy as pie with COO.

PROTOTYPE design pattern

Problem: at assembly time we can convert to any format the inner object with the MEDIATOR scheme, but first we need to instantiate the target object (the `b` we will pass to `cscnv`), and its dynamic type is unknown at compile time. Can we do it?

PROTOTYPE design pattern

Problem: at assembly time we can convert to any format the inner object with the MEDIATOR scheme, but first we need to instantiate the target object (the `b` we will pass to `cscnv`), and its dynamic type is unknown at compile time. Can we do it?

Solution 1 Ask the user for a sample of the new class, then let it clone itself (see Stroustrup).

PROTOTYPE design pattern

Problem: at assembly time we can convert to any format the inner object with the MEDIATOR scheme, but first we need to instantiate the target object (the `b` we will pass to `cscnv`), and its dynamic type is unknown at compile time. Can we do it?

Solution 1 Ask the user for a sample of the new class, then let it clone itself (see Stroustrup). On the library side you have the following:

```
subroutine spmat_allocate(a,mold)
  class(psb_dspmat_type), intent(out)      :: a
  class(psb_d_base_sparse_mat), intent(in) :: mold
  call mold%mold(a%a)
end subroutine
```

with the actual argument for `mold` coming from the user application,

PROTOTYPE design pattern

On the user/application side, while devising the new YYY storage format, the following code is needed:

```fortran
module YYY_module

  type, extends(psb_d_base_sparse_mat) :: psb_d_YYY_sparse_mat
   ! attributes here
  contains
   procedure, pass(a) :: mold => YYY_mold
  end type

contains

  subroutine YYY_mold(a,b)
    implicit none
    class(psb_d_YYY_sparse_mat), intent(in)  :: a
    class(psb_d_base_sparse_mat), intent(out), allocatable :: b
    allocate(psb_d_YYY_sparse_mat :: b)
  end subroutine YYY_mold
```

The library allocate method can handle `d_YYY_sparse_mat` without having seen it before; the set of formats can grow over time.

PROTOTYPE

Solution 2 Embed it in the language itself: the compiler will fill in the necessary bits for the equivalent of the `YYY_mold` method, which does not need to exist any longer:

```
subroutine spmat_allocate(a,mold,info)
  implicit none
  class(psb_dspmat_type), intent(out)       :: a
  class(psb_d_base_sparse_mat), intent(in)  :: mold
  integer, intent(out)                      :: info

  allocate(a%a, mold=mold)
end subroutine
```

Strictly speaking, `mold=` is Fortran 2008; in Fortran 2003 you have `source=` but if the variable is empty they are essentially equivalent.