High Performance Linear Algebra

Lecture 2.4: Sparse Matrix Software Implementation

Ph.D. program in High Performance Scientific Computing

Salvatore Filippone **Pasqua D'Ambra** **Fabio Durastante**

March 19 2026 — 13.00:15.00

Table of Contents

- Sparse matrices (again)
- Design Patterns
  - STATE
  - BUILDER
  - MEDIATOR
  - PROTOTYPE
- GPUs
- Performance
- Parallel SPMV
  - The Distributed Matrix-Vector Product
- Bibliography

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!

Sparse Matrices (again)

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*

- "taking advantage" of zeros essentially means avoiding their explicit storage;
- But this destroys the simple mapping between the index pair (I, J) and the position in memory;
- Therefore, sparse matrix storage formats must rebuild this map using auxiliary index information; at what cost?

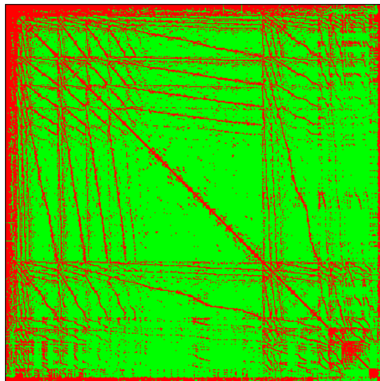
How costly is rebuilding? How does interact with memory traffic? How do you access the various vectors? Multiple factors contribute to determine the overall performance:

- the match between the data structure and the underlying computing architecture, including the possibility of exploiting special hardware instructions;
- the suitability of the data structure to decomposition into independent, load-balanced work units;
- the amount of overhead (both space and time) due to the explicit storage of indices;
- the amount of padding with explicit zeros that may be necessary;
- the interaction between the data structure and the distribution of nonzeros (sparsity pattern) within the sparse matrix;
- the relation between the sparsity pattern and the sequence of memory accesses especially into the x vector.

Many storage formats have been invented over the years; a number of attempts have also been directed at standardizing the interface to these data formats for convenient usage (see e.g., [1, 14, 15]).

Memory access patterns: we have seen that memory accesses are best if they follow the *locality principle*.

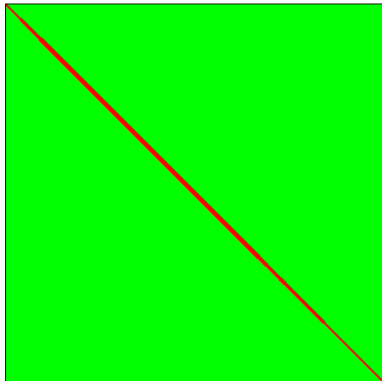
Consider the matrix depicted here:



the accesses into x will be scattered.

Sparse Matrices (again)

However if we do apply a renumbering $\hat{A} \leftarrow PAP^T$ we can obtain:



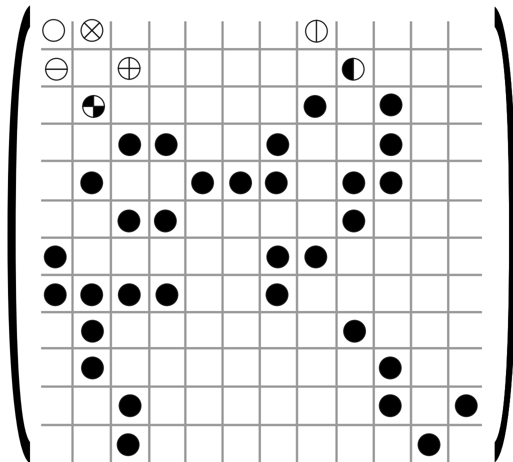
where the x accesses follow a significantly better sequence. We will not discuss renumbering schemes in detail, except to mention that they are all based on heuristics (and they are obviously useful).

Storage schemes

Name	Description
M	Number of rows in matrix
N	Number of columns in matrix
NZ	Number of nonzeros in matrix
AVGNZR	Average number of nonzeros per row
MAXNZR	Maximum number of nonzeros per row
NDIAG	Number of nonzero diagonals
AS	Coefficients array
IA	Row indices array
JA	Column indices array
IRP	Row start pointers array
JCP	Column start pointers array
NZR	Number of nonzeros per row array
OFFSET	Offset for diagonals

Notation for parameters describing a sparse matrix

A hypotetical sparse matrix



Storage schemes: COO

AS ARRAY

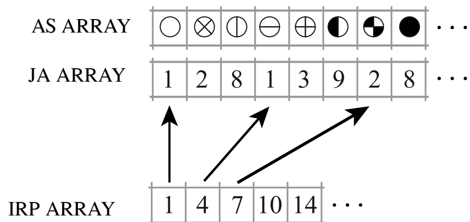
JA ARRAY

IA ARRAY

```
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

AS ARRAY

○	⊗	⦶			
⊖	⊕	◐			
◕	●	●			
●	●	●	●		

⋮

JA ARRAY

1	2	8			
1	3	9			
2	8	10			
3	4	7	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

- Diagonals;
- Jagged diagonals;
- Block CSR;
- Variable Block Rows;
- CSRG;
- Compressed Storage by Columns;
- ...

Note: we are not discussing right now block formats, i.e. storage formats in which the entries are small dense matrices.

...and MANY more...

1 Sparse matrices (again)

Base format	Issues	GPU variants
COO	Memory footprint, atomic data access	ALIGNED_COO [40] SCOO [12] BRO-COO [43] BCCOO and BCCOO+ [50]
CSR	Coalesced access, thread mapping, data reuse	CSR optimization [3, 5, 6, 17, 23, 24, 36, 39, 47, 53] SIC [19] RgCSR [38] BIN-CSR and BIN-BCSR [48] CMRS [26] PCSR [13] BCSR optimization [7, 11, 46, 49]
CSC	Coalesced access	CSC optimization [25]
ELLPACK	Zero padding	ELLPACK-R [44] Sliced ELLPACK [35] Warped ELL [31] ELLR-T [45] Sliced ELLR-T [16] BELLPACK [11] BSELLPACK [41] AdELL [30] ELLPACK-RP [8] BiELL and BiJAD [55] BRO-ELL [43] JAD optimization [28] BTJAD [2] Enhanced JDS [10] pJDS [27]
DIA	Zero padding	DDD-NAIVE and DDD-SPLIT [54] CDS [22]
Hybrid		HYB [6] Combination of CSR and ELL [32, 33] HEC [29] TILE-COMPOSITE [52] SHEC [18] Cocktail [41] HDC [51] Combination of ELLPACK and DIA [31] Combination of BCOO and BCSR [34] BRO-HYB [43]
New		BLSI [37] CRSD [42]

From [21]

Are you unsure what to use?

Are you unsure what to use?

Well, so are we.

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);

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, somewhat annoying);
- We want to be able to change in response to usage requirements within the application (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)

How many other storage schemes out there?

When I heard for the first time Prof. Salvatore Filippone's words "The issue of efficient multiplication of a sparse matrix by a dense vector is worth the effort of an entire PhD" I said to myself I would never bother with that. — Dr. M. Martone

Table of Contents

▶ Sparse matrices (again)

▶ Design Patterns
- STATE
- BUILDER
- MEDIATOR
- PROTOTYPE

▶ GPUs

▶ Performance

▶ Parallel SPMV
- The Distributed Matrix-Vector Product

▶ Bibliography

Design Patterns:

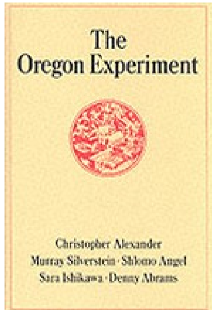
“Best practices” for solving programming problems (appearing in different application domains)

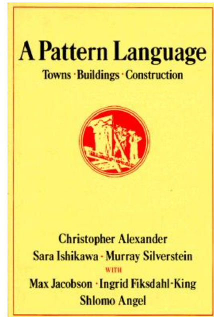
Why bother?

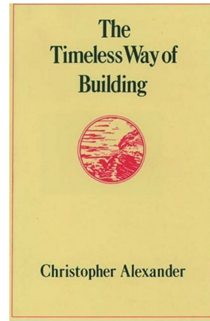
Premature optimization is the root of all evil
Donald Knuth

When someone says “I want a programming language in which I need only say what I wish done,” give him a lollipop.
Alan Perlis

Building architecture (Alexander et al, 1975–1979)







OOD patterns emerge like the independent recurrence of a solution to a common biological problem in multiple evolutionary tree branches. In this sense, they resemble organic growth in nature. Moreover, as the construction of natural beings, software bears consideration as a product of nature. Copying time honored designs that naturally arise in one environment eases the creation of designs that feel natural in another.

Design Patterns:

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

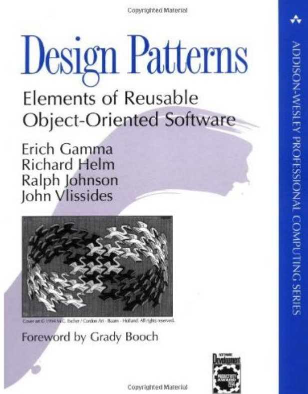
Why bother?

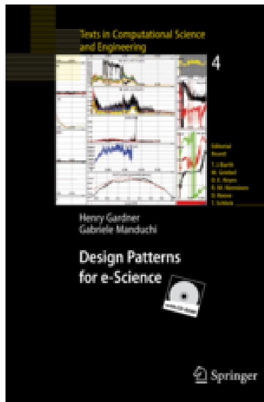
Elements of a software design pattern (Gamma et al. 1995):

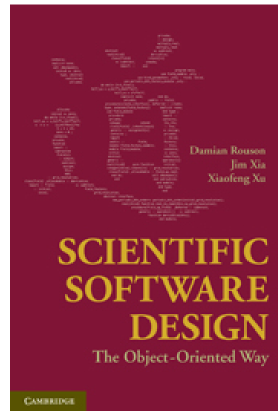
1. The pattern name: a handle the describes a design problem. Its solution, and consequences in a word or two.
2. The problem: a description of when to apply the pattern and within what context.
3. The solution: the elements that constitute the design, the relationships between these elements, their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







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
[4, 9, 20]

> *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

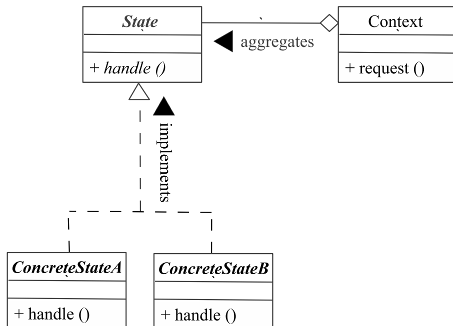
Our sparse matrices need to be adjusted at runtime, so:

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

STATE 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.

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.

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

```
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
```

All non-trivial data structures require non-trivial data management.

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;
- **PROTOTYPE:** Allows existing code to instantiate new classes.

[9] V. Cardellini, S. Filippone and D. Rouson, 2014

We have multiple choices for the data structure that will hold the matrix. Does that mean we (users) have to learn how to build from scratch each and every data structure?

BUILDER design pattern

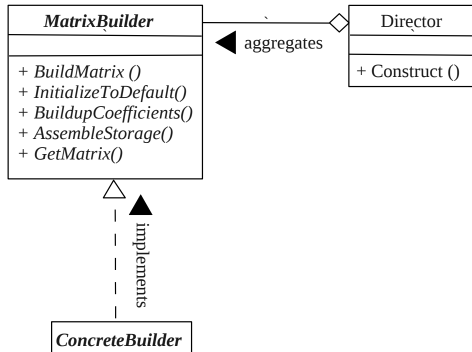
We have multiple choices for the data structure that will hold the matrix. Does that mean we (users) have to learn how to build from scratch each and every data structure?
It is impossible to have a one-step constructor for all but the most trivial cases.

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



BUILDER design pattern

The structure of the application (as seen from the user, including MPI data):

```
call psb_spall(a,desc_a,info)
do i=1, n
   nz = <number of entries in equation  i>
   ia(1:nz) = i
   ja(1:nz) = <list of neighbours of i>
   val(1:nz) = <coefficients A(ia(:),ja(:)) >
   call psb_spins(nz,ia,ja,val,a,desc_a,info)
enddo
call psb_spasb(a,desc_a,info [, afmt, upd, dupl, mold])
```

I.e.: our interface is implicitly in COO-like format. (never mind `desc_a`, that's MPI stuff).
You (application writer) know best how to generate the coefficients, we take care of the rest.
Note: we have shown a simple loop, one row at a time, but there's NO requirement to generate every row in full before passing it to the library; it is perfectly fine to proceed element by element (or even coefficient by coefficient).

Ok, we've seen how you build a sparse matrix.
Now 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.
Now 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 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 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 also that shortcuts may be added if/when warranted.

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 compilation, because your software is growing over time. Can we do it?

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 compilation, because your software is growing over time. Can we do it?

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

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 compilation, because your software is growing over 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:

```
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.

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
```

`mold=` was introduced in Fortran 2008; there is also the `source=` option which copies the data together with the dynamic type.

The STATE design pattern and its companions are also used in our *index maps* which will be discussed later.

It is well known that sorting methods can exhibit wildly varying performance depending on usage conditions. In particular, sorting based on comparison is done often with:

QuickSort
: average complexity $O(n\log(n))$ but worst case complexity $O(n^2)$;

MergeSort
: average and worst case complexity $O(n\log(n))$, but average slower than QuickSort; also, preserves relative order of ties;

If we are sorting matrix entries based on their row/column indices there are some issues:

1. Sorting coefficients within rows is done on lists that are often very short;
2. Sorting coefficients by rows implies there will be many repeated indices, which implies that most QuickSort implementations will get closer to their worst case;
3. Ties need to be handled;
4. Lists of coefficients will often exhibit partial sublists which are already ordered (and QuickSort does not exploit this).

For all of these reasons we prefer MergeSort in the context of sparse matrix data preprocessing, because:

1. The worst case is $O(n \log(n))$ like the average;
2. Repeated indices do not degrade performance;
3. Ties in indices are handled by preserving the relative ordering in the original list;
4. It is possibile to write the code in order to take advantage of existing ordered sublists.

Note that in the handling of Approximate Inverses we also need the Heap data structure, but we do not need a full HeapSort implementation.

Table of Contents

- Sparse matrices (again)
- Design Patterns
  - STATE
  - BUILDER
  - MEDIATOR
  - PROTOTYPE
- GPUs
- Performance
- Parallel SPMV
  - The Distributed Matrix-Vector Product
- Bibliography

Case Study: Techniques for GPUs

Motivations

3 GPUs

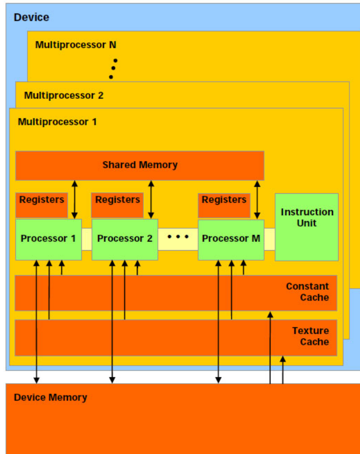
Theoretical GFLOP/s at base clock

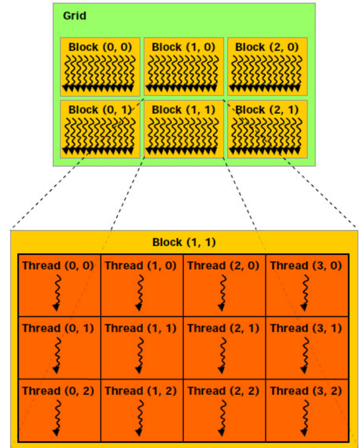
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision

11000
10500
10000
9500
9000
8500
8000
7500
7000
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0

2003 2005 2007 2009 2011 2013 2015

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

Table of Contents

- Sparse matrices (again)
- Design Patterns
  - STATE
  - BUILDER
  - MEDIATOR
  - PROTOTYPE
- GPUs
- Performance
- Parallel SPMV
  - The Distributed Matrix-Vector Product
- Bibliography

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 interfacing 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	—
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
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	—	—	—	—	—
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
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
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

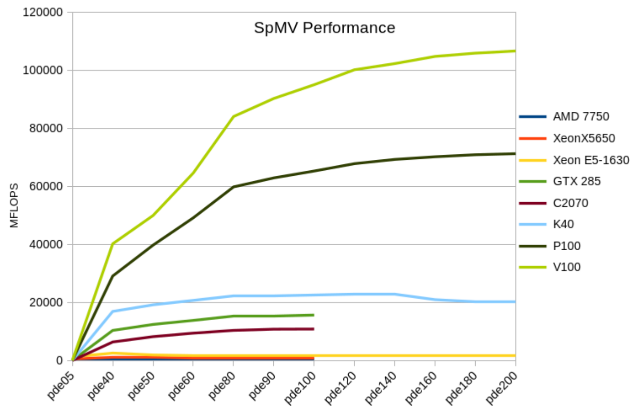
Matrix name	DIA	HDIA	ELL	HLL	HYB
cant	17.1	16.9	15.34	14.73	14.7
mac_econ_fwd500	0.3	1.3	4.9	4.42	5.6
olafu	1.6	11.3	14.9	13.9	11.96
raefsky2	2.7	4.6	12.6	12.3	4.5
af23560	13.5	15.2	15.2	14.47	14.4
mhd4800a	5.8	5.8	8.0	7	3.7
bcsstk17	0.9	6.4	9.45	8.19	7.0
lung2	0.1	3.8	8.0	7.46	5.1
af_1_k101	—	15.9	18.3	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
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_dto	—	—	—	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	—
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 of SpMV



For a detailed analysis see [21]

How good are those numbers?

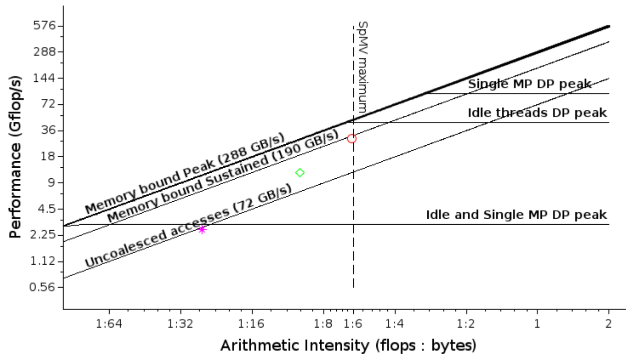


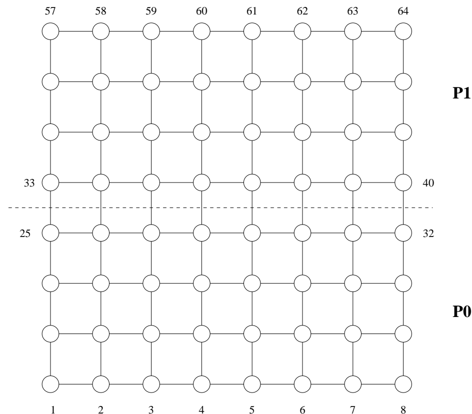
Table of Contents

- Sparse matrices (again)
- Design Patterns
  - STATE
  - BUILDER
  - MEDIATOR
  - PROTOTYPE
- GPUs
- Performance
- Parallel SPMV
  - The Distributed Matrix-Vector Product
- Bibliography

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 on any single process we need to:

- Gather into a buffer y coefficients to be sent:

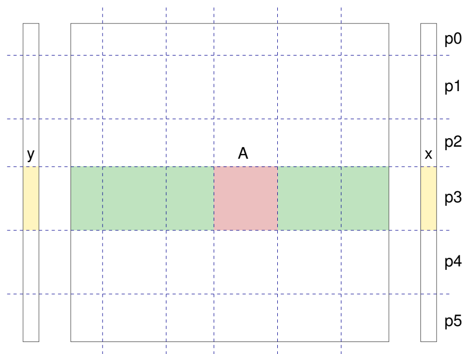
```
for(i=0;i<n;i++)
   y[i] = x[index[i]];
```

- 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]];
```

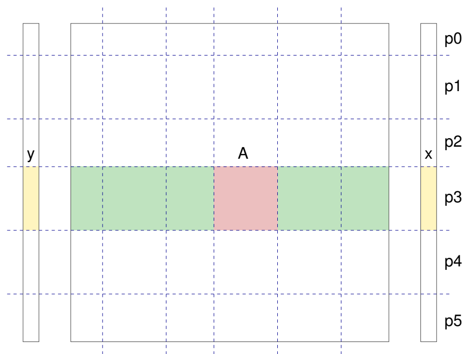
Why?

Distributed Matrix-Vector Product



Pink area corresponds to the local indices, green area corresponds to the halo. *Surface to Volume effect*: for “sensible” data distributions, most of the nonzeros are in the pink area, green area is almost empty.

Distributed Matrix-Vector Product



To compute the SpMV, we have to retrieve values of X corresponding to entries in the green area with a *halo data exchange*, going first through a *gather* and (upon arrival) through a *scatter*. This is a *sparse persistent all-to-all-V collective communication*

The halo data exchange

Persistent variable all-to-all neighborhood collective communication

- **All-to-all:** Each process may send and receive to/from any other;
- **Variable:** The amount of data is specific to any given send/receive pair;
- **Persistent:** The operation is repeated involving the values associated with the same set of vector indices, multiple times;
- **Neighborhood:** Many pairs of processes would exchange an empty set of data, and we can obviously drop the send/receive pair; thus, each process will only actually communicate with a subset of the other processes (its neighborhood).

In our library this is called a *Halo data exchange*.

The halo data exchange

We can implement the halo data exchange in multiple ways:

- Use `MPI_Alltoallv`;
- Build and use a list of messages, then use `MPI_Send` and `MPI_Recv`;
- same as above, but with `MPI_Isend` and `MPI_Irecv`;
- same as above, but with `MPI_Put` and `MPI_Get`;
- Use the new MPI 4.0 Neighborhood Collective Communications on Process Topologies

The halo data exchange

Use `MPI_Alltoallv`

The main issue is that the MPI collective is used in a situation in which most pairs of processes do NOT actually exchange data

Use `MPI_Send` and `MPI_Recv`

Beware of scheduling: should avoid unsafe communications

Use `MPI_Isend` and `MPI_Irecv`

Currently, default (also, uses scheduling)

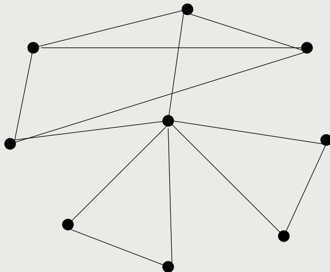
Use Neighborhood Collectives

Should actually be *persistent* collectives (reuse multiple times)

Work under way to test new MPI releases.

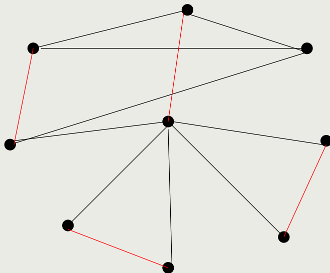
The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



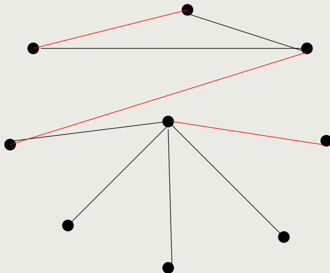
The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



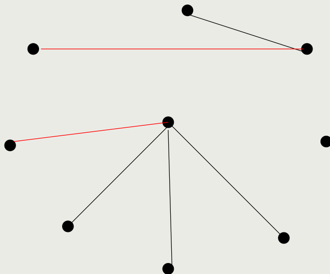
The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



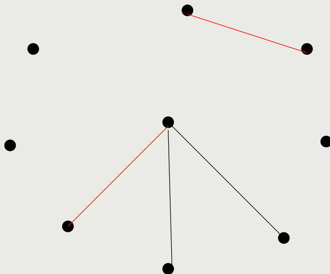
The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



The problem

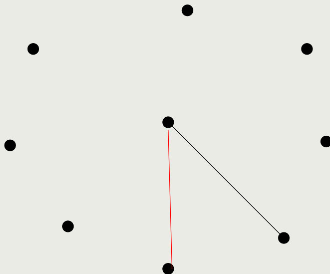
If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



Scheduling communications

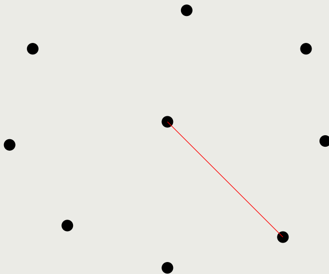
The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



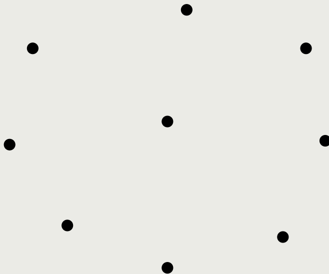
The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



Scheduling communications

The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?

Recipe:

Apply a series of maximal matchings.
Heuristics: scan the nodes in order of descending degree

Matrix Computations for Sparse Matrices implicitly reference an

Index Space

$$\mathcal{I} = \{i, i = 1 \ldots n\}$$

which can be *distributed* over a set of processes so that

$$\forall i \in \mathcal{I} \ : \ \exists (p, j) \leftrightarrow i,$$

where p is a process index and j is a (local) index

Index Maps

There exists an index set spanning a problem space, and this index set is partitioned among multiple processors.

This partition can be realized in many different ways, for example:

- serial/replicated distribution, where each process owns a full copy and no communication is needed (this also cover the case of a serial run);
- Block distribution, where each process owns a subrange of the indices;
- Local list assignment, where we have for each process a list specifying the indices it owns;
- Global list assignment, where we have a list specifying for each index its owner process, replicated on all processes.

Beware of memory footprint

Given a data distribution, we need to answer questions that hinge upon relating the "global" indices to their "local" counterparts:

1. Which global index is the image of a certain local one?
2. Which processor owns a certain global index, and to what local index does it correspond?

- First question is much easier to answer: when we assign a set of indices to a process, can keep track of local/global index values (memory proportional to the local number of points)

Index Maps

- Second question is instead much more complex:
  1. If a global index is owned by a certain process, that process can answer the query *provided* it has a mapping from global to local indices; however, finding any one of the global indices may require searching through a set that is potentially disordered, therefore proper data structures should be designed to facilitate this task;
  2. If the query originated from a process such that the global index corresponds to one of the halo indices, then the process will likely also know which other process owns it, but it will not necessarily know what local index is in use on that other process;
  3. If the query is about an arbitrary global index that may be owned by any process, it may be necessary to have an expensive search phase in which all processes cooperate.

Very useful auxiliary data: a subdomain adjacency graph, one vertex for each subdomain, an edge between two subdomains if they exchange data

Index Maps—base

There are multiple possible implementations of the index map object All of them share some base components:

```
type       :: psb_indx_map
  !> State of the map
  integer(psb_ipk_) :: state = psb_desc_null_
  !> Communication context
  type(psb_ctxt_type) :: ctxt
  !> Number of global rows
  integer(psb_lpk_) :: global_rows = -1
  !> Number of global columns
  integer(psb_lpk_) :: global_cols = -1
  !> Number of local rows
  integer(psb_ipk_) :: local_rows = -1
  !> Number of local columns
  integer(psb_ipk_) :: local_cols = -1
  !> A pointer to the user-defined parts subroutine
  procedure(psb_parts), nopass, pointer :: parts => null(
  !> The global vector assigning indices to
  !>    processes, temp copy
  integer(psb_ipk_), allocatable :: tempvg(:)
  !> Reserved for future use.
  integer(psb_ipk_), allocatable :: oracle(:,:)
  !> Halo owners
  integer(psb_ipk_), allocatable :: halo_owner(:)
  !> Adjacency list for processes
  integer(psb_ipk_), allocatable :: p_adjcncy(:)
end type psb_indx_map
```

Index Maps—List

Lists contain the full mapping

```
type, extends(psb_indx_map) :: psb_list_map
  integer(psb_ipk_) :: pnt_h               = -1
  integer(psb_lpk_), allocatable :: loc_to_glob(:)
  integer(psb_ipk_), allocatable :: glob_to_loc(:)
```

Almost identical, with the addition of a vector identifying the owner of any global index

```
type, extends(psb_list_map) :: psb_glist_map
  integer(psb_ipk_), allocatable :: vgp(:)
```

Index Maps—Block

Block format: indices are assigned to processes in contiguous blocks (not necessarily of the same size)

```
type, extends(psb_indx_map) :: psb_gen_block_map
  integer(psb_lpk_) :: min_glob_row   = -1
  integer(psb_lpk_) :: max_glob_row   = -1
  integer(psb_lpk_), allocatable :: loc_to_glob(:), srt_g2l(:,:), vnl(:)
```

We keep global indices sorted, but maintain a hash table to speed up their retrieval; another auxiliary hash table is used during construction

```
type, extends(psb_indx_map) :: psb_hash_map

  integer(psb_lpk_) :: hashvsize, hashvmask
  integer(psb_ipk_), allocatable :: hashv(:)
  integer(psb_lpk_), allocatable :: glb_lc(:,:), loc_to_glob(:)
  type(psb_hash_type)  :: hash
```

Index Maps

Now we can restate our algorithm to find who owns a certain global index:

```
subroutine psi_graph_fnd_owner(idx,iprc,ladj,idxmap,info)
 integer(psb_lpk_), intent(in)        :: idx(:)
 integer(psb_ipk_), allocatable, intent(out) ::  iprc(:), ladj(:)
 class(psb_indx_map), intent(in) :: idxmap
 integer(psb_ipk_), intent(out)  :: info
```

The argument `idx` contains on each process the (global) indices for which we need to find the owner, which is returned in `iprc`. In principle we could do the following:

1. Do an allgatherv of `idx`
2. For each of the collected indices figure out if current proces owns it
3. Scatter the results
4. Loop through the answers

This method is guaranteed to find the owner, unless an input index has an invalid value, however it could easily require too much additional space because each block of indices is replicated to all processes.

Hence the real algorithm goes like this:

- -1 Figure out a maximum size for a buffer to collect `idx` (min: NP); also check if we have an adjacency list of processes;
- 0 If the initial adjacency list is not empty, use `psi_adj_fnd_sweep` to go through all indices with `psi_adjcncy_fnd_owner` (possibly multiple calls);
- 1 While there are unanswered queries:
- 2 Extract a sample from `idx`, and call `psi_a2a_fnd_owner`;
- 3 Build the list of processes that own the sample indices and store it in `idxmap`;
- 4 Using `psi_adj_fnd_sweep` go through remaining indices and use with `psi_adjcncy_fnd_owner` (possibly multiple calls);
- 5 Go back to 1.

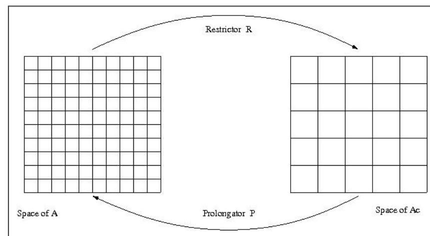
We are alternating between

- Asking all processes for a subset of indices;
- Asking a subset of processes for all the indices;

limiting memory footprint to a specified maximum amount (`psb_cd_set_maxspace()`).

Multilevel Corrections (a preview)

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 (a preview)

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!

```
type amg_d_base_solver_type
end type

type amg_d_base_smoother_type
  class(amg_d_base_solver_type),allocatable :: sv
end type

type amg_d_onelev_type
  type(psb_dspmat_type) :: ac
  type(psb_dlinmap_type) :: map
  class(amg_d_base_smoother_type),allocatable :: sm
end type

type :: amg_dprec_type
  type(amg_d_onelev_type), allocatable :: precv(:)
end type
```

Table of Contents

- Sparse matrices (again)
- Design Patterns
  - STATE
  - BUILDER
  - MEDIATOR
  - PROTOTYPE
- GPUs
- Performance
- Parallel SPMV
  - The Distributed Matrix-Vector Product
- Bibliography

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.
- D'Ambra, Pasqua and Durastante, Fabio and Filippone, Salvatore: AMG Preconditioners for Linear Solvers towards Extreme Scale, *SIAM Journal on Scientific Computing*, 43(5):S679-S703, 2021

[1] A. Abdelfattah et al. *Interface for Sparse Linear Algebra Operations*. 2024. arXiv: 2411.13259 [cs.MS]. URL: https://arxiv.org/abs/2411.13259.

[2] W. Abu-Sufah and A. Abdel-Karim. “An Effective Approach for Implementing Sparse Matrix-Vector Multiplication on Graphics Processing Units”. In: *Proc. of 14th IEEE Int’l Conf. on High Performance Computing and Communication*. HPCC ’12. June 2012, pp. 453–460.

[3] A.-K. C. Ahamed and F. Magoules. “Fast Sparse Matrix-Vector Multiplication on Graphics Processing Unit for Finite Element Analysis”. In: *Proc. of IEEE 14th Int’l Conference on High Performance Computing and Communications*. HPCC ’12. 2012, pp. 1307–1314.

[4] D. Barbieri et al. “Design Patterns for Scientific Computations on Sparse Matrices”. In: *Euro-Par 2011: Parallel Processing Workshops*. Vol. 7155. LNCS. Springer, 2012, pp. 367–376.

[5] M. M. Baskaran and R. Bordawekar. *Optimizing Sparse Matrix-Vector Multiplication on GPUs*. Tech. rep. RC24704. IBM Research, 2009.

[6] N. Bell and M. Garland. *Efficient sparse matrix-vector multiplication on CUDA*. Tech. rep. NVR-2008-004. NVIDIA Corp., 2008.

[7] L. Buatois, G. Caumon, and B. Levy. "Concurrent Number Cruncher: A GPU Implementation of a General Sparse Linear Solver". In: *Int. J. Parallel Emerg. Distrib. Syst.* 24.3 (2009), pp. 205–223. ISSN: 1744-5760.

[8] W. Cao et al. "Implementing Sparse Matrix-Vector multiplication using CUDA based on a hybrid sparse matrix format". In: *Proc. of 2010 In'tl Conf. on Computer Application and System Modeling*. Vol. 11. ICCASM '10. IEEE, Oct. 2010, pp. 161–165.

[9] V. Cardellini, S. Filippone, and D. Rouson. "Design patterns for sparse-matrix computations on hybrid CPU/GPU platforms". In: *Sci. Program.* 22.1 (2014), pp. 1–19.

[10] A. Cevahir, A. Nukada, and S. Matsuoka. "Fast Conjugate Gradients with Multiple GPUs". In: *Computational Science - ICCS 2009*. Vol. 5544. LNCS. Springer, 2009, pp. 893–903. ISBN: 978-3-642-01969-2.

[11] J. W. Choi, A. Singh, and R. W. Vuduc. "Model-driven autotuning of sparse matrix-vector multiply on GPUs". In: *SIGPLAN Not.* 45 (5 Jan. 2010), pp. 115–126.

[12] H.-V. Dang and B. Schmidt. "CUDA-enabled Sparse Matrix-Vector Multiplication on GPUs using atomic operations". In: *Parallel Comput.* 39.11 (2013), pp. 737–750. ISSN: 0167-8191.

[13] M. Dehnavi, D. Fernandez, and D. Giannacopoulos. "Finite-Element Sparse Matrix Vector Multiplication on Graphic Processing Units". In: *IEEE Trans. Magnetics* 46.8 (2010), pp. 2982–2985. ISSN: 0018-9464.

[14] I. Duff et al. "Level 3 Basic Linear Algebra Subprograms for Sparse Matrices: a User Level Interface". In: *ACM Trans. Math. Softw.* 23.3 (1997), pp. 379–401.

[15] I. S. Duff, M. A. Heroux, and R. Pozo. “An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum”. In: *ACM Trans. Math. Softw.* 28.2 (June 2002), pp. 239–267. ISSN: 0098-3500. DOI: 10.1145/567806.567810. URL: https://doi.org/10.1145/567806.567810.

[16] A. Dziekonski, A. Lamecki, and M. Mrozowski. “A Memory Efficient and Fast Sparse Matrix Vector Product on a GPU”. In: *Progress in Electromagnetics Research* 116 (2011), pp. 49–63.

[17] A. H. El Zein and A. P. Rendell. “Generating optimal CUDA sparse matrix vector product implementations for evolving GPU hardware”. In: *Concurr. Comput.: Pract. Exper.* 24.1 (2012), pp. 3–13. ISSN: 1532-0634.

[18] X. Feng et al. “A segment-based sparse matrix vector multiplication on CUDA”. In: *Concurr. Comput.: Pract. Exper.* 26.1 (2014), pp. 271–286. ISSN: 1532-0634.

[19] X. Feng et al. “Optimization of sparse matrix-vector multiplication with variant CSR on GPUs”. In: *Proc. of 17th Int’l Conf. on Parallel and Distributed Systems*. ICPADS ’11. IEEE Computer Society, 2011, pp. 165–172.

[20] S. Filippone and A. Buttari. “Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003”. In: *ACM Trans. Math. Softw.* 38.4 (2012), 23:1–23:20.

[21] S. Filippone et al. “Sparse matrix-vector multiplication on GPGPUs”. In: *ACM Trans. Math. Software* 43.4 (2017), Art. 30, 49. ISSN: 0098-3500.

[22] J. Godwin, J. Holewinski, and P. Sadayappan. “High-performance Sparse Matrix-vector Multiplication on GPUs for Structured Grid Computations”. In: *Proc. of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. GPGPU-5. London, United Kingdom: ACM, 2012, pp. 47–56. ISBN: 978-1-4503-1233-2.

[23] D. Guo and W. Gropp. “Adaptive thread distributions for SpMV on a GPU”. In: *Proc. of Extreme Scaling Workshop*. BW-XSEDE ’12. Chicago, Illinois: University of Illinois at Urbana-Champaign, 2012, 2:1–2:5.

[24] P. Guo and L. Wang. “Auto-tuning CUDA parameters for sparse matrix-vector multiplication on GPUs”. In: *Proc. of 2010 Int’l. Conf. on Computational and Information Sciences*. ICCIS ’10. IEEE Computer Society, Dec. 2010, pp. 1154–1157.

[25] M. R. Hugues and S. G. Petiton. “Sparse Matrix Formats Evaluation and Optimization on a GPU”. In: *Proc. of 12th IEEE Int’l Conf. on High Performance Computing and Communications*. HPCC ’10. IEEE Computer Society, Sept. 2010, pp. 122–129.

[26] Z. Koza et al. “Compressed Multiple-Row Storage Format”. In: *ArXiv e-prints* (Mar. 2012). eprint: `http://arxiv.org/abs/1203.2946v1`.

[27] M. Kreutzer et al. “Sparse Matrix-vector Multiplication on GPGPU Clusters: A New Storage Format and a Scalable Implementation”. In: *Proc. of 26th IEEE Int’l Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IPDPSW ’12. 2012, pp. 1696–1702. ISBN: 978-0-7695-4676-6.

[28] R. Li and Y. Saad. “GPU-accelerated preconditioned iterative linear solvers”. In: *J. Supercomput.* 63.2 (2013), pp. 443–466. ISSN: 0920-8542.

[29] H. Liu et al. “Sparse matrix-vector multiplication on NVIDIA GPU”. In: *Int. J. Numerical Analysis and Modeling, Series B* 3.2 (2012), pp. 185–191.

[30] M. Maggioni and T. Berger-Wolf. “AdELL: An Adaptive Warp-Balancing ELL Format for Efficient Sparse Matrix-Vector Multiplication on GPUs”. In: *Proc. of 42nd Int’l Conf. on Parallel Processing*. ICPP ’13. IEEE Computer Society, Oct. 2013, pp. 11–20.

[31] M. Maggioni, T. Berger-Wolf, and J. Liang. “GPU-Based Steady-State Solution of the Chemical Master Equation”. In: *Proc. of IEEE 27th Int’l Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IPDPSW ’13. May 2013, pp. 579–588. DOI: 10.1109/IPDPSW.2013.271.

[32] A. Maringanti, V. Athavale, and S. B. Patkar. “Acceleration of conjugate gradient method for circuit simulation using CUDA”. In: *Proc. of 2009 Int’l Conf. on High Performance Computing*. HiPC ’09. IEEE, Dec. 2009, pp. 438–444.

[33] K. Matam and K. Kothapalli. “Accelerating Sparse Matrix Vector Multiplication in Iterative Methods Using GPU”. In: *Proc. of 40th Int’l Conf. on Parallel Processing*. ICPP ’11. IEEE Computer Society, Sept. 2011, pp. 612–621.

[34] A. Monakov and A. Avetisyan. "Implementing Blocked Sparse Matrix-Vector Multiplication on NVIDIA GPUs". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Vol. 5657. LNCS. Springer-Verlag, 2009, pp. 289–297. ISBN: 978-3-642-03137-3.

[35] A. Monakov, A. Lokhmotov, and A. Avetisyan. "Automatically tuning sparse matrix-vector multiplication for GPU architectures". In: *High Performance Embedded Architectures and Compilers*. Vol. 5952. LNCS. Springer-Verlag, 2010, pp. 111–125.

[36] D. Mukunoki and D. Takahashi. "Optimization of sparse matrix-vector multiplication for CRS format on NVIDIA Kepler architecture GPUs". In: *Computational Science and Its Applications*. Vol. 7975. LNCS. Springer, 2013, pp. 211–223. ISBN: 978-3-642-39639-7.

[37] B. Neelima, S. R. Prakash, and R. M. Reddy. "New Sparse Matrix Storage Format to Improve The Performance of Total SpMV Time". In: *Scalable Comput.: Pract. Exper.* 13.2 (2012), pp. 159–171.

[38] T. Oberhuber, A. Suzuki, and J. Vacata. "New row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA". In: *Acta Technica* 56 (2011), pp. 447–466.

[39] I. Reguly and M. Giles. "Efficient sparse matrix-vector multiplication on cache-based GPUs". In: *Proc. of Innovative Parallel Computing*. InPar '12. IEEE, May 2012, pp. 1–12.

[40] M. Shah and V. Patel. "An Efficient Sparse Matrix Multiplication for Skewed Matrix on GPU". In: *Proc. of 14th IEEE Int'l Conf. on High Performance Comput. and Comm.* June 2012, pp. 1301–1306.

[41] B.-Y. Su and K. Keutzer. “clSpMV: a cross-platform OpenCL SpMV framework on GPUs”. In: *Proc. of 26th ACM Int’l Conf. on Supercomputing*. ICS ’12. Venice, Italy, 2012, pp. 353–364.

[42] X. Sun et al. “Optimizing SpMV for diagonal sparse matrices on GPU”. In: *Proc. of 40th Int’l Conf. on Parallel Processing*. ICPP ’11. IEEE Computer Society, Sept. 2011, pp. 492–501. DOI: 10.1109/ICPP.2011.53.

[43] W. T. Tang et al. “Accelerating Sparse Matrix-vector Multiplication on GPUs Using Bit-representation-optimized Schemes”. In: *Proc. of Int’l Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’13. Denver, Colorado: ACM, 2013, 26:1–26:12. ISBN: 978-1-4503-2378-9.

[44] F. Vázquez, J. J. Fernández, and E. M. Garzón. “A new approach for sparse matrix vector product on NVIDIA GPUs”. In: *Concurr. Comput.: Pract. Exper.* 23.8 (2011), pp. 815–826. ISSN: 1532-0634.

[45] F. Vázquez, J. Fernández, and E. M. Garzón. "Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach". In: *Parallel Comput.* 38.8 (2012), pp. 408–420. ISSN: 0167-8191.

[46] M. Verschoor and A. C. Jalba. "Analysis and performance estimation of the Conjugate Gradient method on multiple GPUs". In: *Parallel Comput.* 38.10-11 (2012), pp. 552–575. ISSN: 0167-8191.

[47] Z. Wang et al. "Optimizing sparse matrix-vector multiplication on CUDA". In: *Proc. of 2nd Int'l Conf. on Education Technology and Computer*. Vol. 4. ICETC '10. IEEE, June 2010, pp. 109–113.

[48] D. Weber et al. "Efficient GPU Data Structures and Methods to Solve Sparse Linear Systems in Dynamics Applications". In: *Computer Graphics Forum* 32.1 (2013), pp. 16–26. ISSN: 1467-8659.

[49] W. Xu et al. “Optimizing Sparse Matrix Vector Multiplication Using Cache Blocking Method on Fermi GPU”. In: *Proc. of 13th ACIS Int’l Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing*. SNPD ’12. IEEE Computer Society, Aug. 2012, pp. 231–235. DOI: 10.1109/SNPD.2012.20.

[50] S. Yan et al. “yaSpMV: yet another SpMV framework on GPUs”. In: *Proc. of 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’14. Feb. 2014, pp. 107–118.

[51] W. Yang et al. “Optimization of quasi-diagonal matrix-vector multiplication on GPU”. In: *Int. J. High Perform. Comput. Appl.* 28.2 (2014), pp. 183–195.

[52] X. Yang, S. Parthasarathy, and P. Sadayappan. “Fast Sparse Matrix-vector Multiplication on GPUs: Implications for Graph Mining”. In: *Proc. VLDB Endow.* 4.4 (2011), pp. 231–242. ISSN: 2150-8097.

[53] H. Yoshizawa and D. Takahashi. “Automatic tuning of sparse matrix-vector multiplication for CRS format on GPUs”. In: *Proc. of IEEE 15th Int’l Conf. on Computational Science and Engineering*. CSE ’12. Dec. 2012, pp. 130–136.

[54] L. Yuan et al. “Optimizing Sparse Matrix Vector Multiplication Using Diagonal Storage Matrix Format”. In: *Proc. of 12th IEEE Int’l Conf. on High Performance Computing and Communications*. HPCC ’10. Sept. 2010, pp. 585–590. DOI: 10.1109/HPCC.2010.67.

[55] C. Zheng et al. “BiELL: A bisection ELLPACK based storage format for optimizing SpMV on GPUs”. In: *J. Parallel Distrib. Comput.* 74.7 (2014), pp. 2639–2647. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.03.002.