Efficient, Optimal MPI Datatype Reconstruction for Vector and Index Types

Martin Kalany, Jesper Larsson Träff
{kalany,traff}@par.tuwien.ac.at

Vienna University of Technology
Faculty of Informatics, Institute for Information Systems
Research Group Parallel Computing

MPI Derived datatypes

…you all know about them

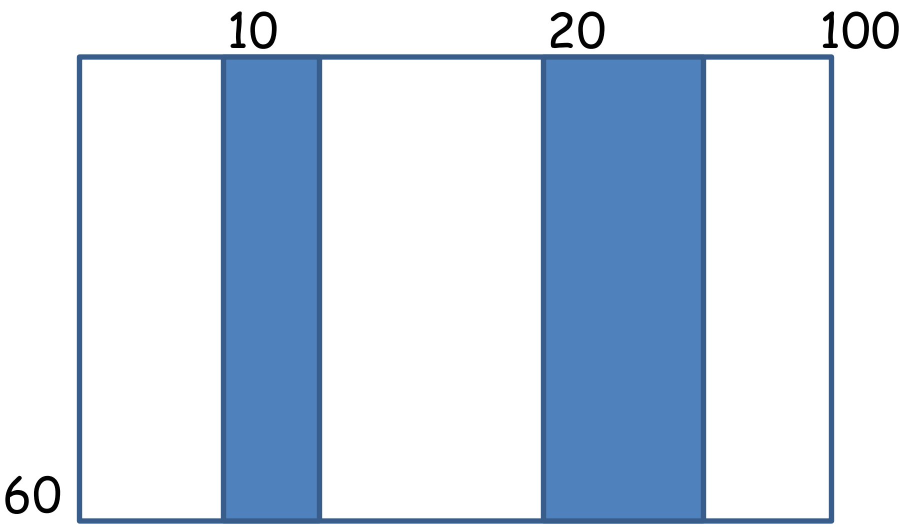
Mechanism for describing application data layouts that are

- Non-consecutive (sub-matrices, parts of buffers, …)
- Non-homogeneous: different basetypes (MPI_CHAR, MPI_DOUBLE, MPI_INT, …)

Derived (user-defined) datatypes orthogonal to communication mode: works for point-to-point, collective, one-sided, extremely important for MPI-IO specification, …

Unique feature of MPI (could be useful for other interfaces!)

Example: Receive two matrix-column blocks…



Column1 =
MPI_Vector

Column2 =
MPI_Vector

BlocksType' =
MPI_Struct(Colum1,Column2)

Rowtype =
MPI_Indexed

BlocksType =
MPI_Contig(Rowtype)

Note:
BlocksType ≠ BlocksType'

Note: BlocksType ≠ BlocksType'

Derived datatype describes the layout of data, and fixes the order in which data are accessed

Layout: Ordered sequence of <displacement,basetype> pairs

aka MPI type map

Example: The row-wise layout of matrix-columns:

<10,DOUBLE>, <11,DOUBLE>, <20,DOUBLE>, <21,DOUBLE>,
<22,DOUBLE>, <110,DOUBLE>,<111,DOUBLE>,<120,DOUBLE>,
<121,DOUBLE>, <122,DOUBLE>, …

Note: BlocksType ≠ BlocksType'

Derived datatype describes the layout of data, and fixes the order in which data are accessed

Layout: Ordered sequence of <displacement,basetype> pairs

Homogeneous: same basetype for all pairs

aka MPI type map

Example: Row-wise layout of matrix-columns, basetype DOUBLE

<10,11,20,21,22,110,111,120,121,122, …>

Displacement sequence: list of displacements (in access order)

Displacements themselves not necessarily ordered!

Note: BlocksType ≠ BlocksType'

Derived datatype describes the layout of data, and fixes the order in which data are accessed

Layout: Ordered sequence of <displacement,basetype> pairs

Homogeneous: same basetype for all pairs

aka MPI type map

Example: Column-wise layout of matrix-columns

<10,11,110,111,210,211,…,20,21,22,120,121,122,220,221,222, …>

Displacement sequence: list of displacements (in access order)

Displacements themselves not necessarily ordered!

Datatype representation

Tree (or DAG) like structure:

- Internal type nodes describe how child nodes are repeated (how often, where)
- Leaf nodes describe basetype

Semantics:
There is a function flatten(T) that generates the type map from datatype T, i.e., flatten(T) = D

MPI usage:
Derived datatype explicitly constructed by application using type constructor operations, MPI library maintains some convenient and efficient internal representation

Derived datatype representation of displacement sequence

MPI_Contig(60)

↓

MPI_Indexed(2,<2,3>)

↓

MPI_DOUBLE

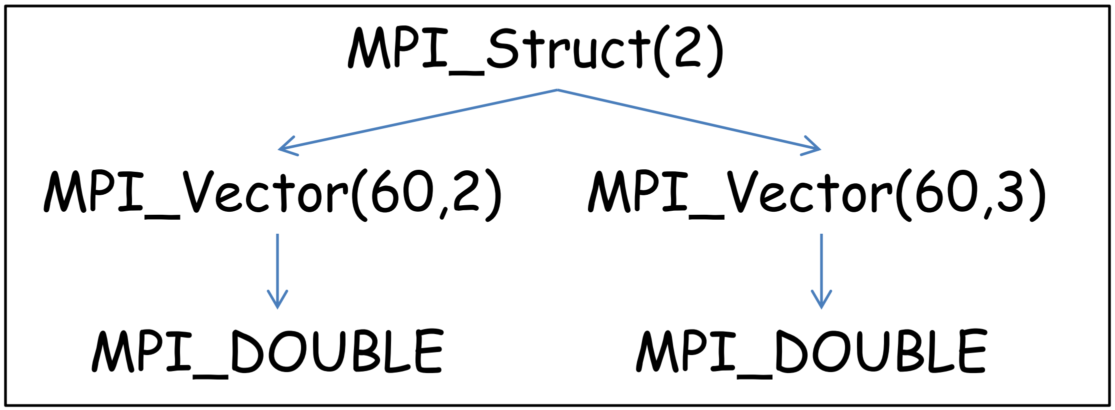
Type nodes:

- MPI_Contig
- MPI_Vector
- MPI_IndexBlock
- MPI_Indexed

MPI datatype constructors

- MPI_Type_contiguous(oldtype, …, &newtype);
- MPI_Type_vector(oldtype,…,&newtype);
- MPI_Type_create_indexed_block(oldtype,. …, &newtype);
- MPI_Type_indexed(oldtype, …, &newtype);

Derived datatype representation of heterogeneous type map



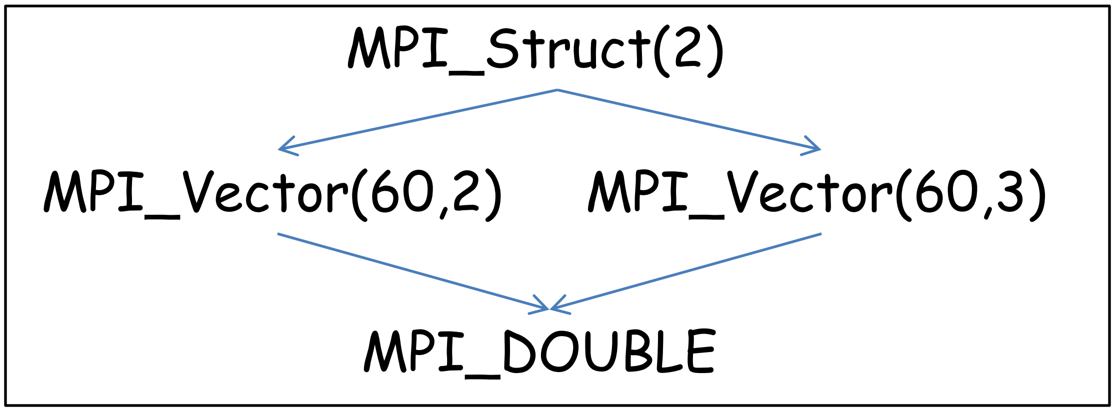
Type nodes:

- MPI_Contig
- MPI_Vector
- MPI_IndexBlock
- MPI_Indexed
- MPI_Struct

MPI datatype constructors

- MPI_Type_contiguous(oldtype, …, &newtype);
- MPI_Type_vector(oldtype,…,&newtype);
- MPI_Type_create_indexed_block(oldtype,. …, &newtype);
- MPI_Type_indexed(oldtype, …, &newtype);
- MPI_Type_create_struct(oldtypes[], …, &newtype);

Derived datatype representation of heterogeneous type map



Type nodes:

- MPI_Contig
- MPI_Vector
- MPI_IndexBlock
- MPI_Indexed
- MPI_Struct

MPI datatype constructors

- MPI_Type_contiguous(oldtype, …, &newtype);
- MPI_Type_vector(oldtype,…,&newtype);
- MPI_Type_create_indexed_block(oldtype,. …, &newtype);
- MPI_Type_indexed(oldtype, …, &newtype);
- MPI_Type_create_struct(oldtypes[], …, &newtype);

Our internal representation (not very important here)

- leaf(B): basetype B at displacement 0
- vec(c,d,C): regular, strided repetition of subtype C at displacements 0, d, 2d, 3d, … (c-1)d
- idx(c,<i0,i1,i2,…,i(c-1)>,C): repetition of subtype C at displacements i0, i1, i2, …, i(c-1)

- strc(c,<i0,i1,i2,…,i(c-1)>,<C0,C1,C2,…,C(c-1)>): subtypes C0, C1, C2, …, C(c-1) at displacements i0, i1, i2, …, i(c-1)

- Can easily represent the MPI constructors (for now, we omit MPI_Type_indexed)
- Avoids some tedious problems with extents and "true extents"

Datatype representation

Tree (or DAG) like structure:

- Internal type nodes describe how child nodes are repeated (how often, where)
- Leaf nodes describe basetype

Semantics:
There is a function flatten(T) that generates the type map from datatype T, i.e., flatten(T) = D

Note:
There are (infinitely) many ways T to describe a given D

Some may be better than others

Some natural questions

- What is the best way to describe given displacement sequence (homogeneous type map) as derived datatype?
- What is the best way to describe heterogeneous type map as derived datatype?
- How do the set of constructors affect complexity and quality?

Depends on what is meant by "best" (cost function)

Abstract cost function:
Account for space consumption and (indirectly) processing cost

Real "best" depends on MPI library and (communication) system

Cost model

•cost(leaf(B)): some constant K
•cost(vec(c,d,C)): some (other) constant K'
•cost(idx(c,<i0,i1,i2,…,i(c-1)>,C)): some constant K''+c (non-constant)

•cost(strc(c,<i0,i1,i2,…,i(c-1)>,<C0,C1,C2,…,C(c-1)>)): some constant K'''+2c (non-constant)

Justification:
leaf, vec: only constant information needed to process
idx, struc: space proportional to displacement and type lists required, must be traversed on processing

cost(T): sum of costs of all nodes in T (additive cost model)

Problems

Given:
- set of constructors (leaf, vec, idx, struc, …)
- cost function cost(T)

Type reconstruction problem: Find cost-optimal representation for given type map

Type normalization problem: Find cost-optimal representation for given, user-defined datatype

William Gropp, Torsten Hoefler, Rajeev Thakur, Jesper Larsson Träff: Performance Expectations and Guidelines for MPI Derived Datatypes. EuroMPI 2011: 150-159

Known facts

MPI libraries use heuristics (folklore) to improve internal datatype representation (MPI_Struct => MPI_Indexed => MPI_Vector => MPI_Contig): type normalization, see, e.g.,

Fredrik Kjolstad, Torsten Hoefler, Marc Snir: Automatic datatype generation and optimization. PPOPP 2012: 327-328 (and more extentsive TR!)

Homogeneous type maps of size n, leaf, vec and idx nodes: $O(n\sqrt{n})$ time reconstruction, normalization in time O(size(T))

Jesper Larsson Träff: Optimal MPI Datatype Normalization for Vector and Index-block Types. EuroMPI/ASIA 2014: 33

New results

Homogeneous type maps of size n, leaf, vec and idx nodes: Better algorithm for finding all repeated prefixes, O(n log n/log log n) time type reconstruction

- Prefix algorithm can be used for normalization
- MPI_Index (idxbuc node) can be incorporated in O(n^2 log^2 n) time

Not here:

Arbitrary type maps of size n, all type nodes leaf, vec, idx, struc, type reconstruction into trees is polynomial but O(n^4)

Robert Ganian, Martin Kalany, Stefan Szeider, Jesper Larsson Träff: Polynomial-time Construction of Optimal Tree-structured Communication Data Layout Descriptions. CoRR abs/1506.09100 (2015)

and Master's Thesis by Martin Kalany

Type reconstruction strategy

Derived datatype nodes describe structured repetition of subtype

↓

To compute datatype from *given* displacement sequence: look for all possible repetitions

↓

Use dynamic programming to compute best datatype

From now on: homogeneous type maps = displacement sequences

Terminology

Displacement sequence:
Array D[] of n integer displacements , D[i] i'th displacement in sequence (D[i] can be >0, <0, =0, repetitions allowed)

D[] = <10,11,20,21,22,110,111,120,121,122, …>

…can trivially be represented as derived datatype

idx(n,<10,11,20,21,22,110,111,120,121,122, …>)
↓
DOUBLE

Cost = n+K"+K

Terminology

Displacement sequence:
Array D[] of n integer displacements , D[i] i'th displacement in sequence (D[i] can be >0, <0, =0, repetitions allowed)

D[] = <10,11,20,21,22,110,111,120,121,122, …>

D[i,j]: Segment of displacement sequence from i to j (inclusive)

D[2,4] = <20,21,22>

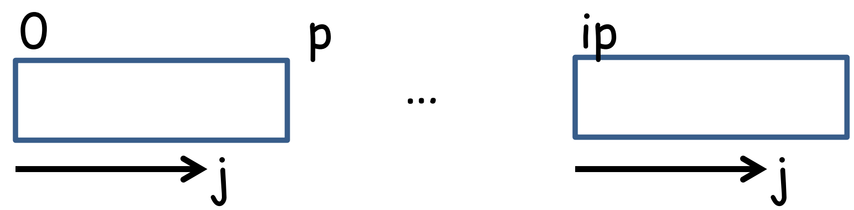
D[0,2] = <10,11,20>

D[0,p-1]: Prefix of length p

Definition:
Prefix of length p is repeated in D if

$D[j]-D[0] = D[ip+j]-D[ip]$

for $0 \le j < p$ and $1 \le i < n/p$



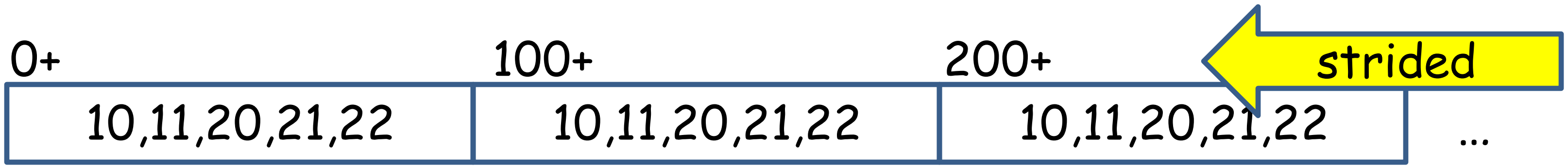
Observation: A prefix of length p can be repeated only if $p|n$, trivial prefixes $D[0,0]$ of length 1 and $D[0,n-1]$ of length 1

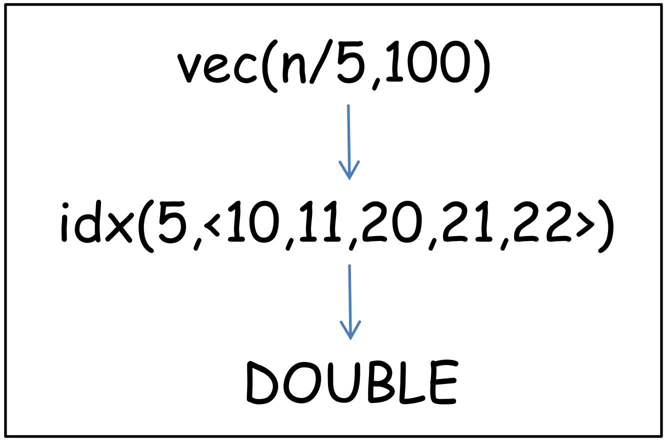
Definition: A repeated prefix of length p is strided if additionally

$D[p]-D[0] = D[(i+1)p]-D[ip]$

Prefix D[0,4] = <10,11,20,21,22> is repeated (and strided) in

D[] = <10,11,20,21,22,110,111,120,121,122, …>





Cost = K'+5+K''+K

Finding repeated prefixes

Finding strided prefixes is easy (EuroMPI 2014): longest repeated prefix in arbitrary D can be found in one scan in O(n) time

Finding repeated, non-strided prefixes; trivial approach:

Try all divisors p of n, for each check by scan of D whether prefix D[0,p-1] is repeated: total time $O(n\sqrt{n})$

At most $2\sqrt{n}$ divisors in n to check

Observation:
If D[0,p-1] is repeated prefix of D, checking whether D[0,p-1] is a strided prefix takes O(n/p) time

Claim:
Let p divisor of n. All repeated prefixes of length q where q is a divisor of p (including q=p) can be found in linear time

Idea:
To find all repeated prefixes of D, let

$n = (p1^{a1}) (p2^{a2}) (p3^{a3}) \ldots (pk^{ak})$

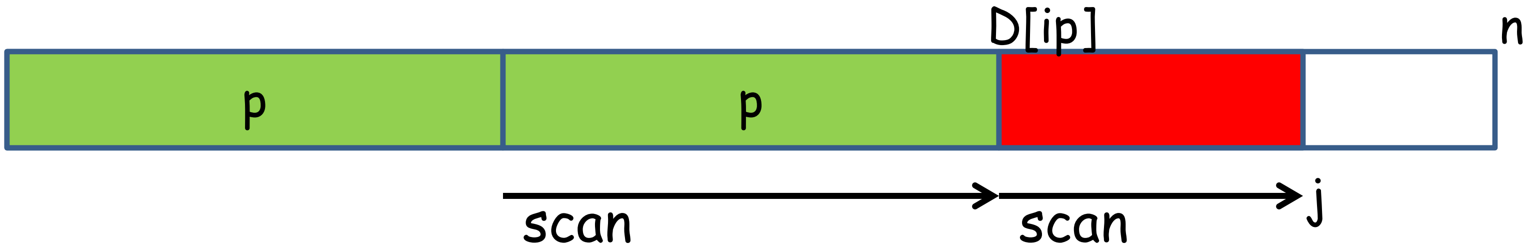
be prime factorization of n. Apply claim for all $p = (n/pi)$

Result from number theory (Robin, 1983):

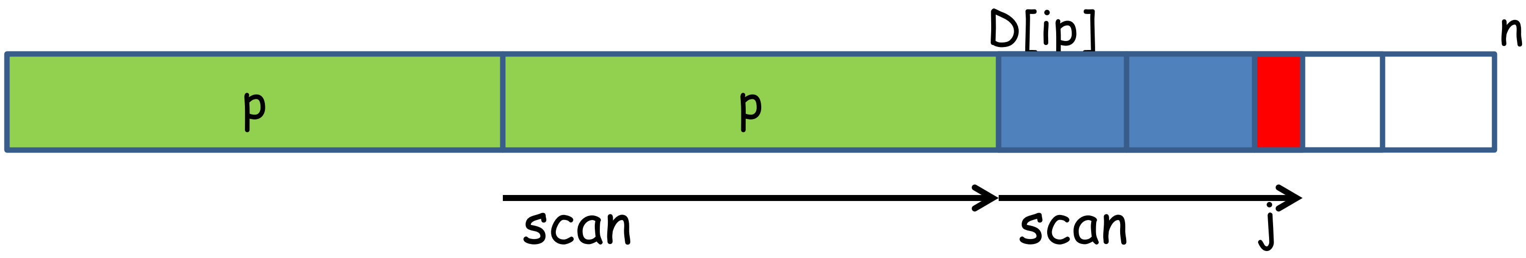
Number of distinct prime factors of n is $O(\log n/\log \log n)$

Proposition:
All repeated prefixes of given displacement sequence D of length n can be found in O(n log n/log log n) time

Proof of claim:
Pick (largest) divisor p of n, check if D[0,p-1] is repeated prefix



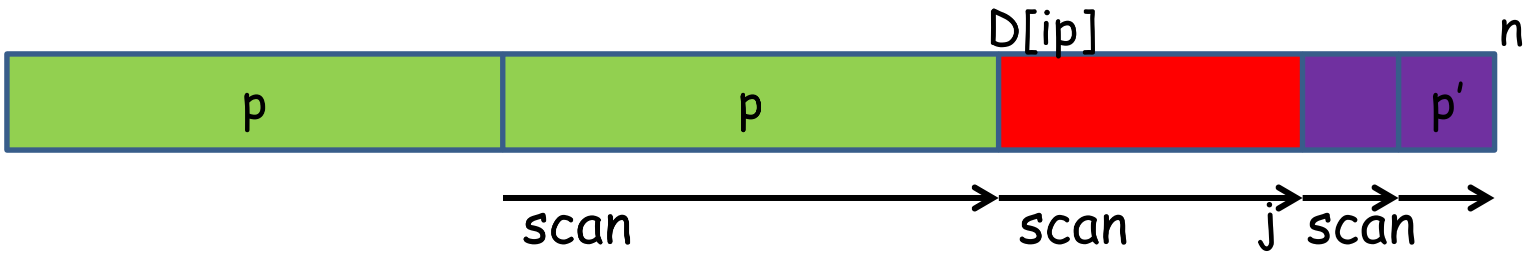
Prefix mismatch:
$D[ip+j]-D[ip] \neq D[j]-D[0]$



Prefix mismatch:
D[ip+j]-D[ip] ≠ D[j]-D[0]

1. Choose p' = gcd(p,j), continue scan for repeated prefix D[0,p'-1] from j

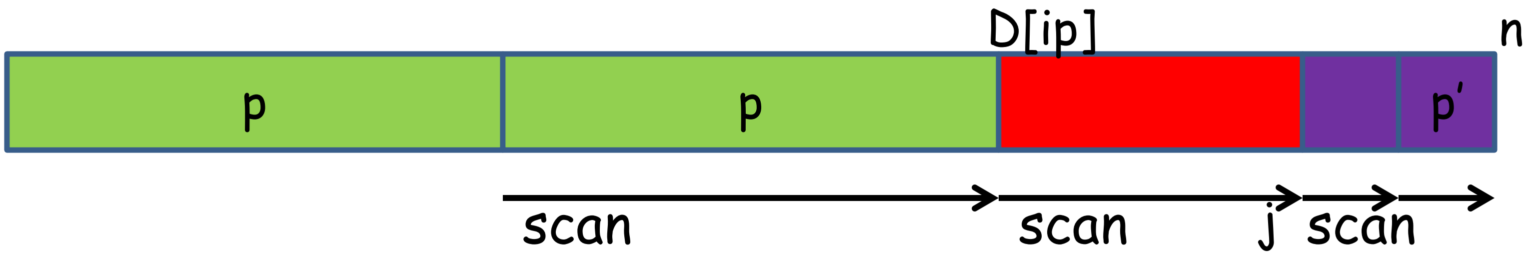
Prefix that is divisor of p but not j cannot be repeated prefix



Prefix mismatch:
D[ip+j]-D[ip] ≠ D[j]-D[0]

1. Choose p' = gcd(p,j), continue scan for repeated prefix D[0,p'-1] from j
2. Check whether prefix D[0,p'-1] is repeated in D[0,p-1]

Step 2: recurse on 

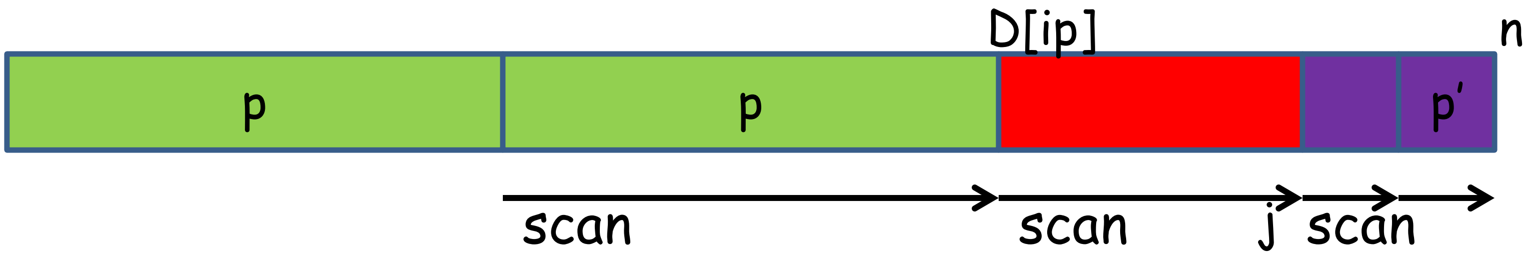


Prefix mismatch:
$D[ip+j]-D[ip] \neq D[j]-D[0]$

Step 1: linear scan, always increasing index order: $O(n)$

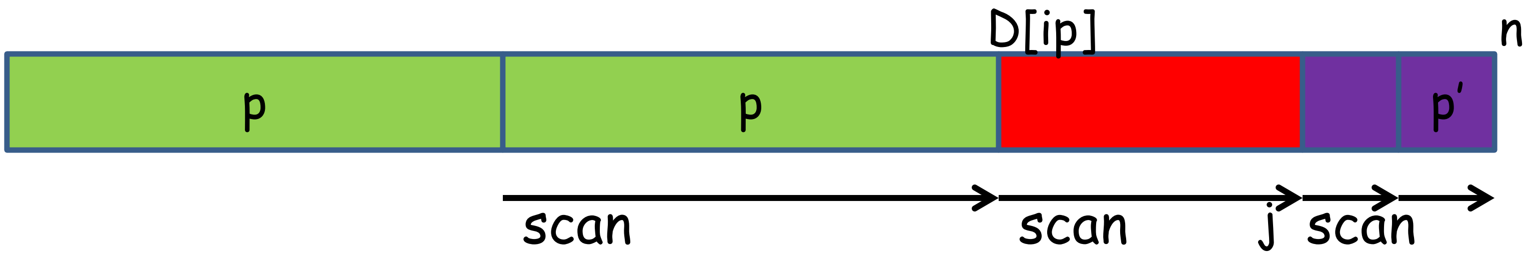
Step 2: recurse on p′ in p so $O(p)$

Total time: $O(n)$



Prefix mismatch:
D[ip+j]-D[ip] ≠ D[j]-D[0]

Algorithm determines largest p' that is a divisor of p where D[0,p'-1] is repeated prefix of D



Prefix mismatch:
D[ip+j]-D[ip] ≠ D[j]-D[0]

To find all repeated prefixes of length q where q divisor of p: recurse on q in p.

Observation:
If D[0,q-1] is repeated prefix of D[0,p-1], and D[0,p-1] is repeated prefix of D, then D[0,q-1] is repeated prefix of D

Structure of optimal path

Observations:
With leaf, vec, idx nodes (no struc), datatypes are simple paths. Each constructor has only one child

Call index node ixd(c,<i0,i1,…>) where i0≠0 a shifted node. A type tree with at most one shifted node is nice. For any datatype path T there exists nice T' (describing the same displacement sequence) with cost(T')≤cost(T)

Cost-optimal T has at most one node with count=1 (a shifted idx)

Cost optimal T has depth (log n)

Cost optimal T's have optimal substructure:
dynamic programming principle applies

idx(1,<0>)
↓
idx(3,<27,2,55>)
↓
idx(2,<0,7,13>)
↓
idx(3,<1,2,4>)
↓
DOUBLE

⇒

Nice type path

idx(1,<28>)
↓
idx(3,<0,-25,28>)
↓
idx(2,<0,7,13>)
↓
idx(3,<0,1,3>)
↓
DOUBLE

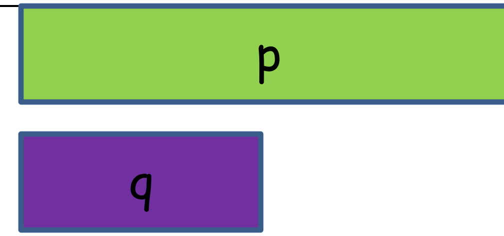
Full algorithm

Precompute: all repeated prefixes and longest strides

1. Find all repeated prefixes p
2. For each p, find largest s(p)≤n such that D[0,p-1] is strided in D[0,s(p)-1]
3. Optimal datatype representation for segment D[0,0] of length 1 is T(0) = leaf(basetype)

Technicality:
Algorithm for aligned displacement sequences with D[0] = 0

Step 4: dynamic programming




```
for all repeated prefixes D[0,p-1]:
    BestCost = ∞
    for all repeated prefixes q of D[0,p-1]:

        VecCost = K'+cost(T(q))  // cost of vec node
        if VecCost<BestCost and p≤s(q)
          T(p) = vec(p/q,stride,T(q)) where stride = D[q]-D[0]
          BestCost = VecCost

        IdxCost = K''+p/q+cost(T(q)) // cost of idx node
        if IdxCost<BestCost
          indices = <D[0],D[q],D[2q],…>
          T(p) = idx(p/q,indices,T(q))
          BestCost = IdxCost
```

Complexity

Steps 1 takes $O(n \log n/\log \log n)$ time by the proposition on repeated prefixes

Step 2 requires $O(n/p)$ time for each divisor $p|n$. By a theorem from number theory (Divisor summatory function, Gronwall, 1913)

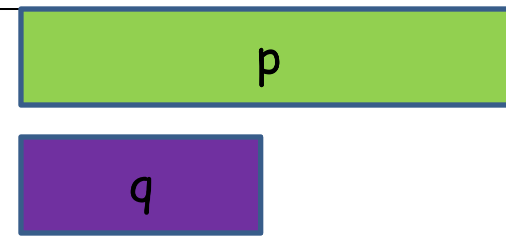
$\sum(p|n): n/p = \sum(p|n): p = O(n \log n/\log \log n)$

In step 4, both loops are over repeated prefixes. There can be at most $2\sqrt{n}$, so if body of inner loop can be implemented in constant time, total time is $O(n)$

```
for all repeated prefixes D[0,p-1]:
    BestCost = ∞
    for all repeated prefixes q of D[0,p-1]:

        VecCost = K'+cost(T(q))  // cost of vec node
        if VecCost<BestCost and p≤s(q)
          T(p) = vec(p/q,stride,T(q)) where stride = D[q]-D[0]
          BestCost = VecCost

        IdxCost = K''+p/q+cost(T(q)) // cost of idx node
        if IdxCost<BestCost
          indices = <D[0],D[q],D[2q],...>
          T(p) = idx(p/q,indices,T(q))
          BestCost = IdxCost
```



Fill in indices later

Theorem:
Cost optimal datatype path representing displacement sequence D of length n using constructors leaf, vec, idx can be computed in $O(n \log n/\log \log n)$ time

Additional constructors

idx node corresponds to MPI_Type_create_indexed_block constructor.

MPI_Type_indexed needs list of displacements and list of block sizes, represented by additional constructor

- idxbuc(c,d,<i0,i1,i2,…,i(c-1)>,<b0,b1,b2,…,b(c-1)>) with cost K'''+2c

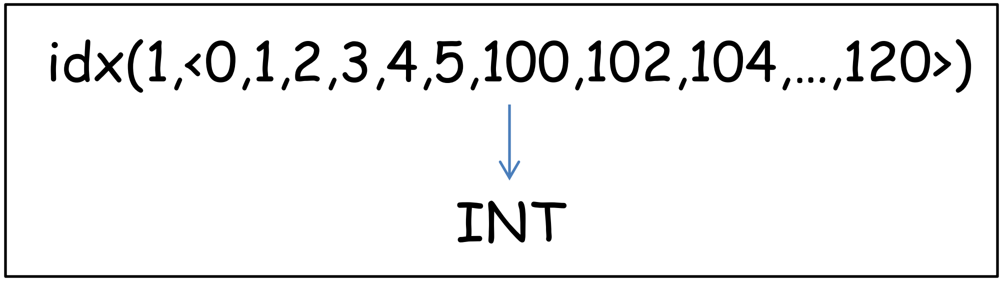
Extra check in inner loop of dynamic programming algorithm needed, requires sorting (to find best block size), total time O(√n n log n)

Outlook, summary

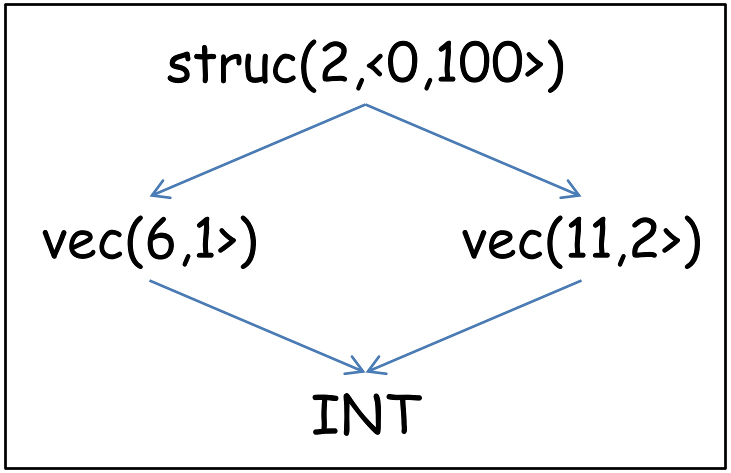
- Simple algorithm to find all repeated prefixes much faster than trivial $O(n\sqrt{n})$ approach
- Much better algorithm for type reconstruction with restricted set of constructors leaf, vec, idx, now $O(n \log n/\log \log n)$
- Can be used in algorithm for type normalization (EuroMPI 2014)
- Can incorporate additional constructors: idxbuc (MPI_Type_indexed), triangular types (see tomorrow), …, but not for type normalization

Note:

struc() node does give more power, even for (homogeneous) displacement sequences



Cost = $K''+17+K$



Cost = $K'''+2+2K'+K$

Makes sense to look for constructors inbetween idx() and struc()

This work was supported by

EPiGRAM