Parallel programming models (cont'd)

Prof. Richard Vuduc

Georgia Institute of Technology

CSE/CS 8803 PNA, Spring 2008

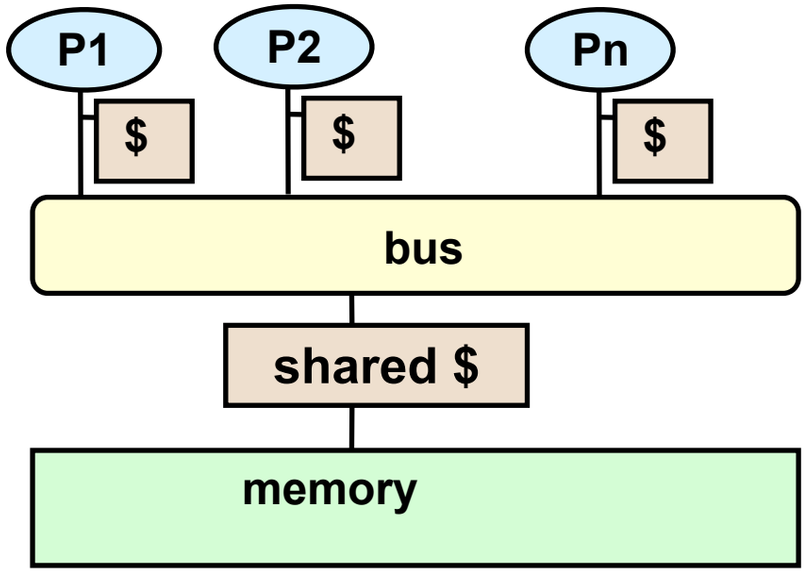
[L.06] Thursday, January 24, 2008

Sources for today's material

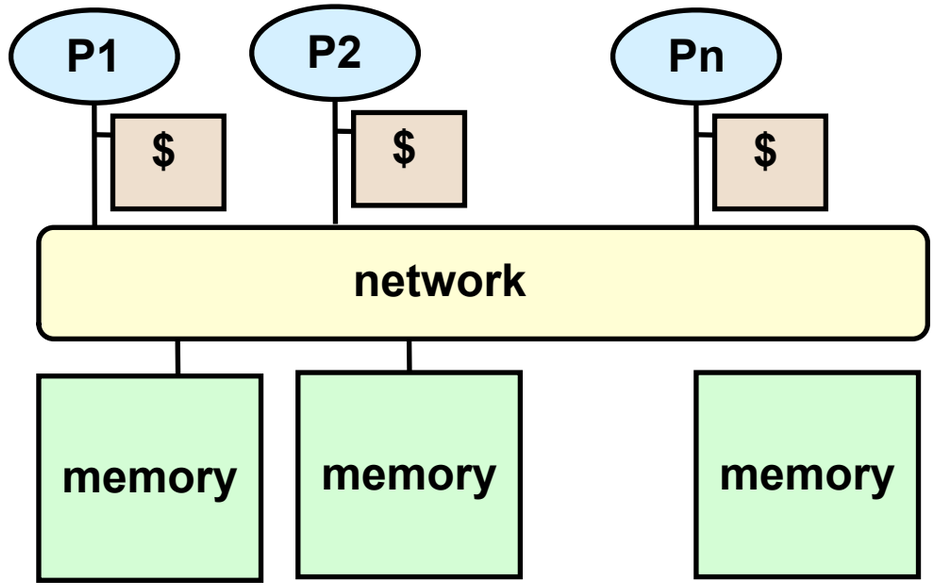
- Mike Heath at UIUC
- CS 267 (Yelick & Demmel, UCB)
- Leiserson (MIT)
- Bill Gropp (UIUC)
- Alan Edelman (MIT/Interactive Supercomputing)

Review: Shared memory

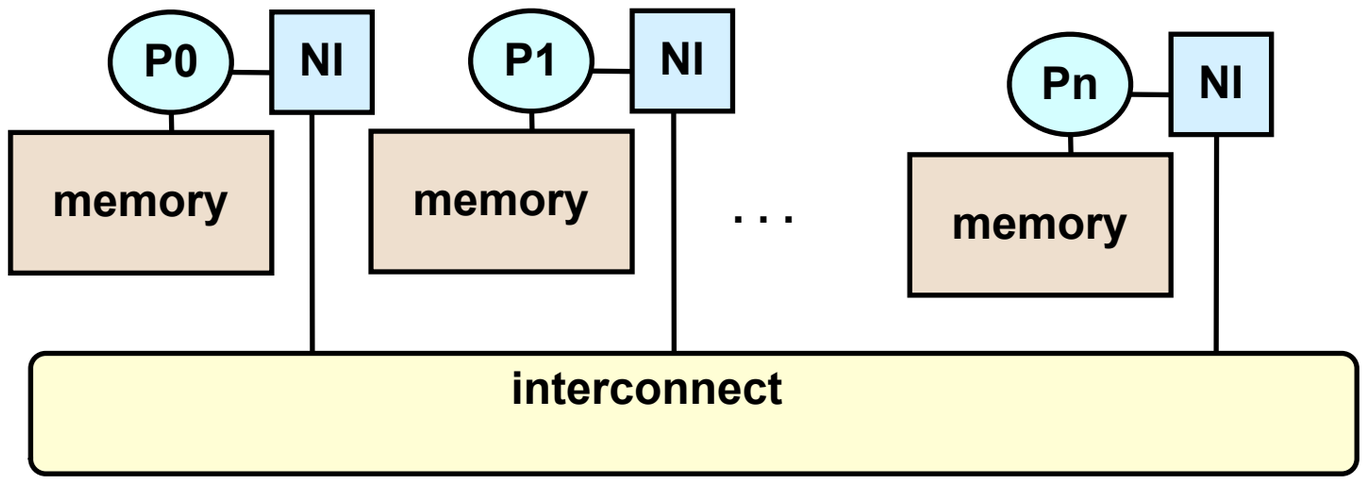
- Shared memory multiprocessors
  - Shared vs. distributed caches: performance vs. complexity (coherence)
  - NUMA
  - Current practical scalability in O(10) to O(100) processors
- Shared memory programming models
  - Program = **threads** of control + shared address space
  - **Communicate implicitly** via shared variables
  - Synchronize using **locks** and **barriers**



Machine model 1a:
Symmetric Multiprocessor (SMP)



Machine model 1c: "Dancehall"
Distributed shared memory



Machine model 2a:
Distributed memory

Cilk: C extensions to support dynamic multithreading

Cilk (Leiserson, *et al.*, 1996+)

- **Fork/join-style C extensions** for dynamic multithreaded apps on SMPs

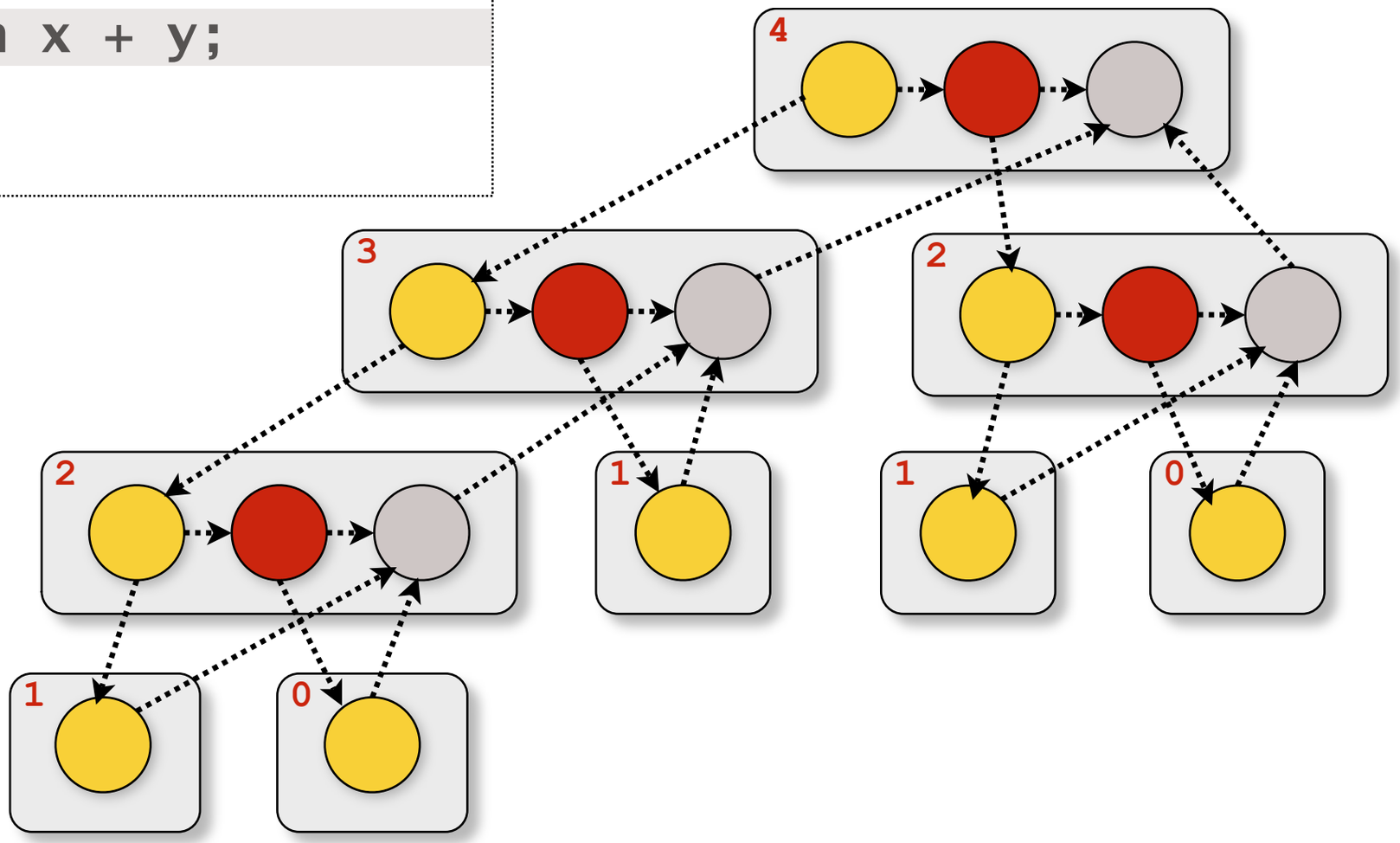
```
int fib (int n) {
  if (n < 2) return 1;
  else {
    int x, y;
    x = fib (n-1);
    y = fib (n-2);

    return x + y;
  }
}
```

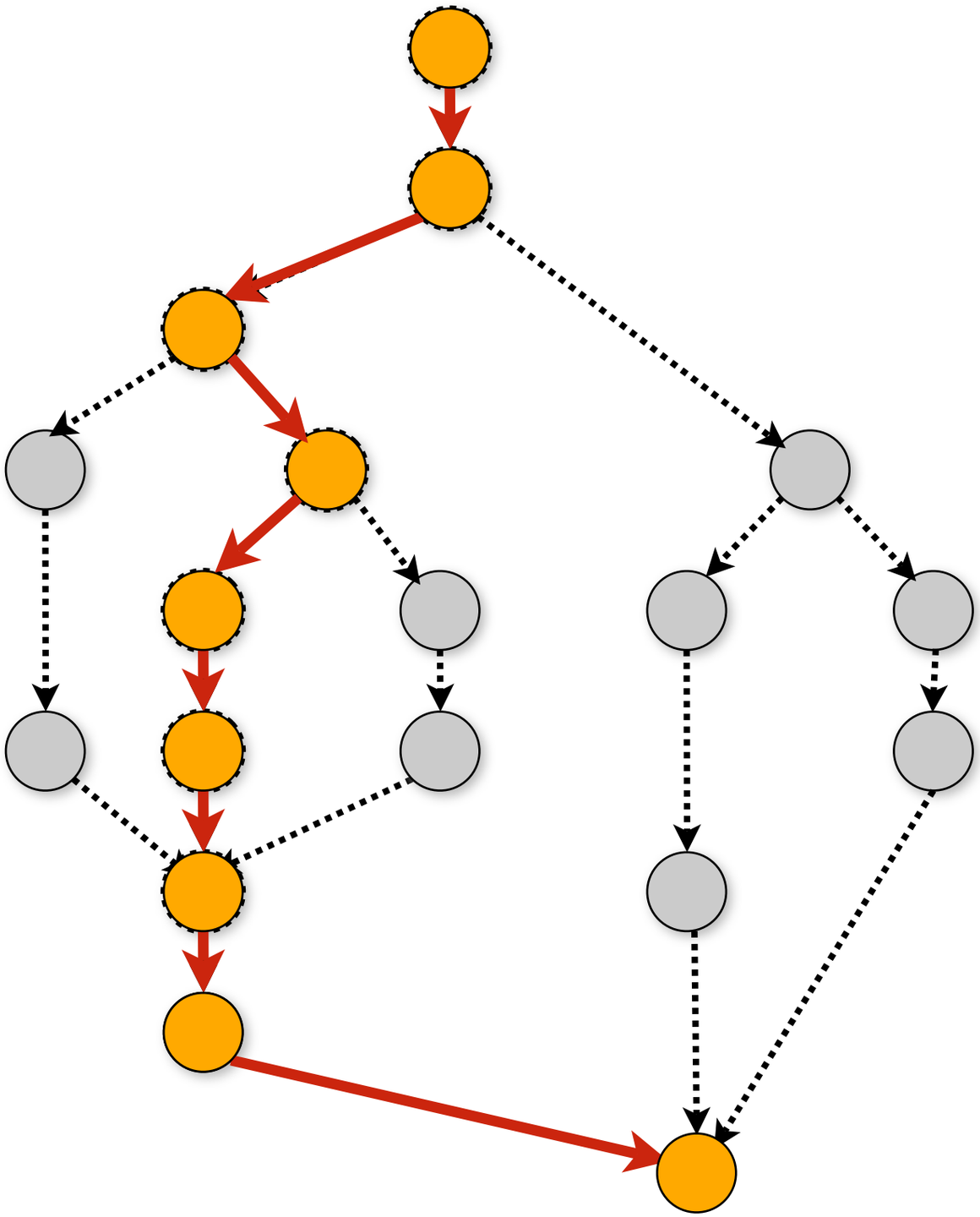
```
cilk int fib (int n) {
  if (n < 2) return 1;
  else {
    int x, y;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;
    return x + y;
  }
}
```

- **"Faithful" extension**: Omitting parallel keywords = valid serial C program

```
cilk int fib (int n) {
  if (n < 2) return 1;
  else {
    int x, y;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;
    return x + y;
  }
}
```

Dynamic computation DAG:
fib(4)





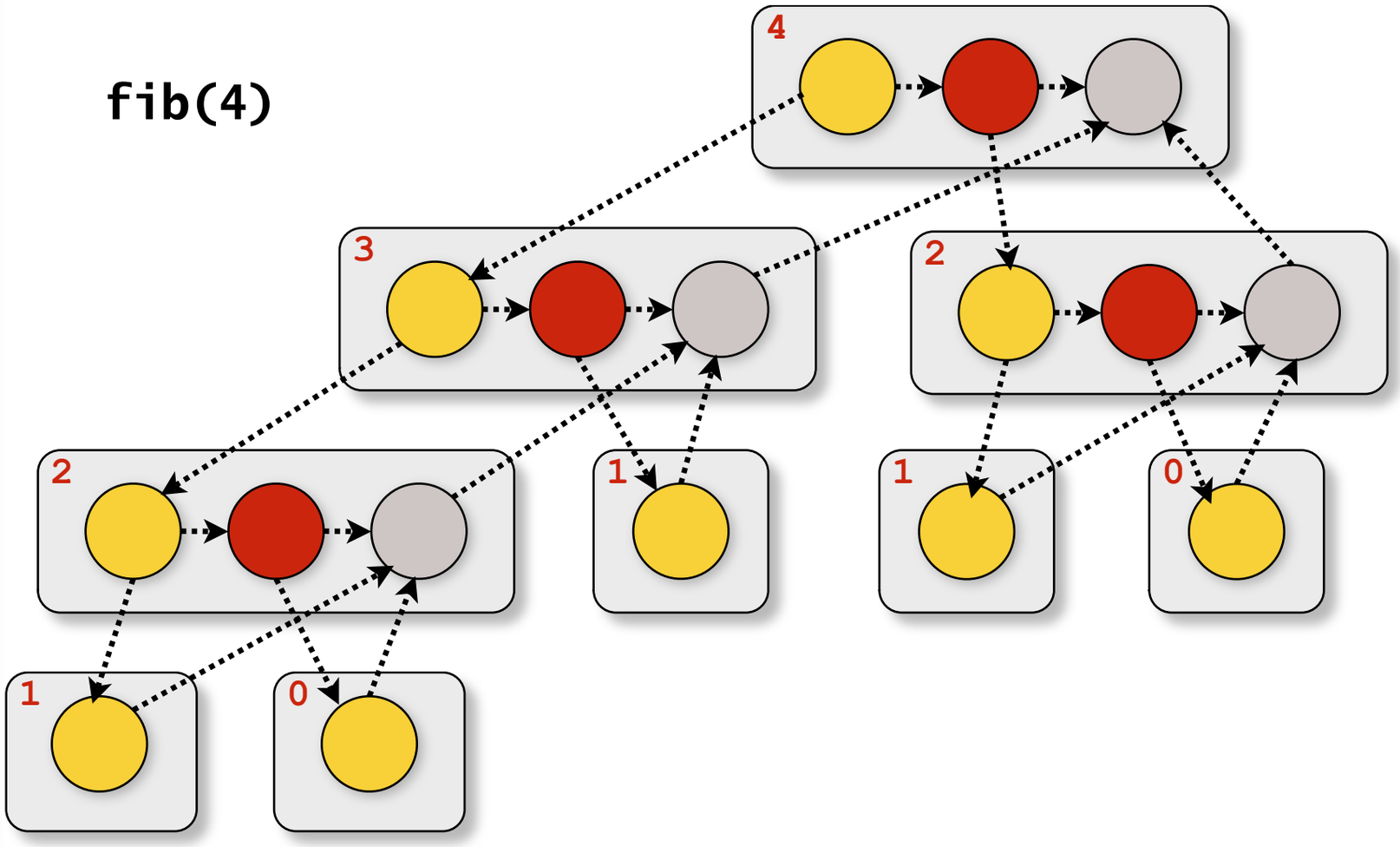
Critical path:

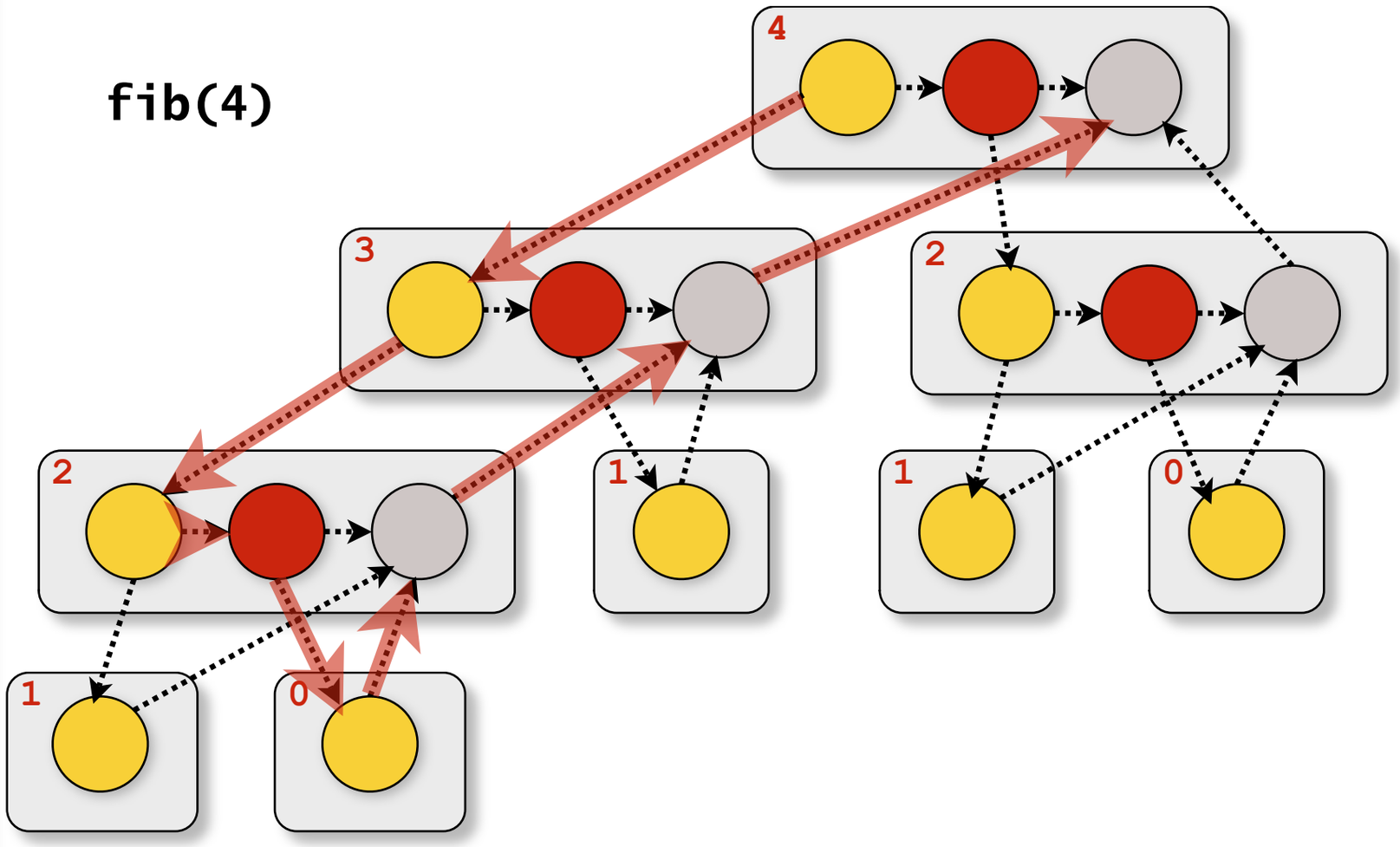
$$T_\infty = \text{“span”}$$

Degree of parallelism:

$$\frac{T_1}{T_\infty}$$

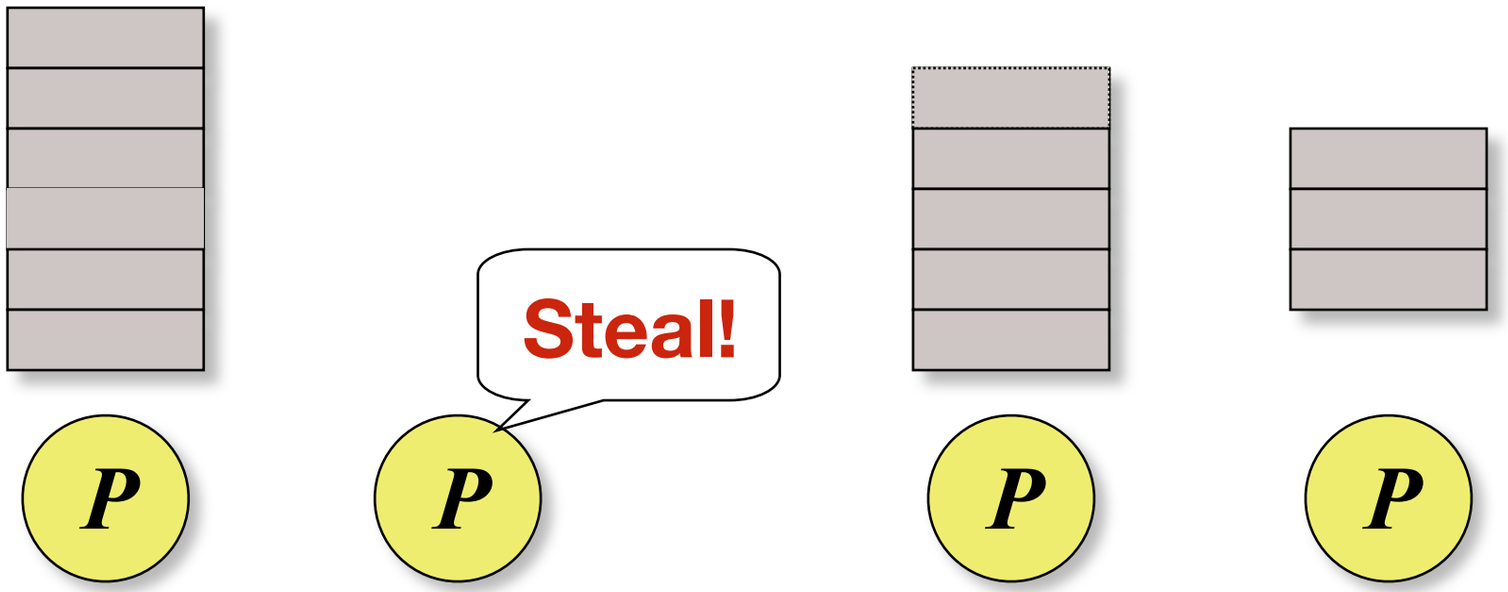
fib(4)





$$T_1 = 17$$
$$T_\infty = 8$$
$$\Longrightarrow \quad \frac{T_1}{T_\infty} \approx 2$$

Cilk's work-stealing scheduler



- Processors maintain **work queues**
- When out of work, processor selects another processor uniformly at **random** and takes work

Performance of Cilk's work-stealing scheduler

- **Theorem**: Expected running time is

$$T_p \leq \frac{T_1}{p} + O(T_\infty)$$

- *Proof sketch*: See Blumofe & Leiserson (FOCS '94)
  - A processor is either **working** or **stealing**.
  - Total time working is $\mathbf{T_1}$.
  - Each steal has **1/p** chance of reducing span by 1, so cost of all steals is $\mathbf{O(p*T_\infty)}$.
  - Expected time: $(T_1 + O(p^* T_\infty)) / p = \mathbf{T_1/p + O(T_\infty)}$

Cilk vs. PThreads

```
for (i = 0; i < N; ++i)
  spawn-or-fork foo (i);
sync-or-join;
```

- What happens if one instance of **foo** waits on another?
- Liveness property
  - Cilk: Lazy (“**may**”) parallelism
  - PThreads: Eager (“**must**”) parallelism

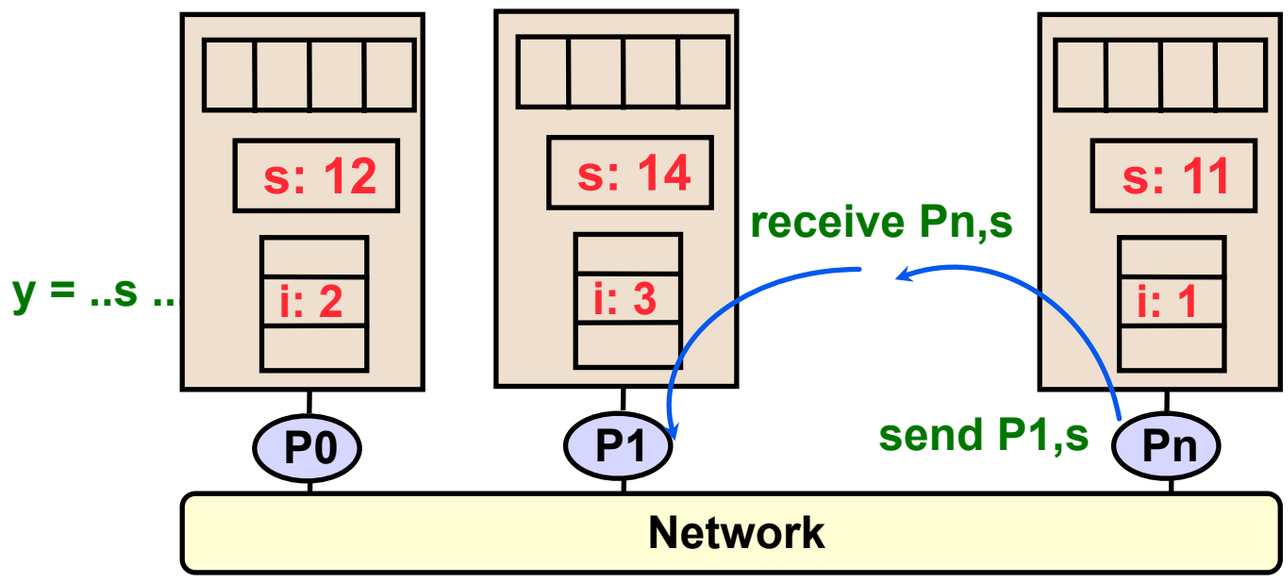
Additional features and caveats

- Provides
  - Fence (Cilk_fence)
  - Locking (Cilk_lock)
  - Automatic clean-up for local dynamic allocation (Cilk_alloca)
  - Aborts
- “Inlets” support use of spawned results in arbitrary expressions
- Run-time scheduler uses work-stealing
- Beware sharing through pointer-passing, deadlocks

Message Passing Interface (MPI)

Recall the message passing model

- Program = **named** processes
- **No shared** address space
- Processes communicate *via* **explicit send/receive** operations



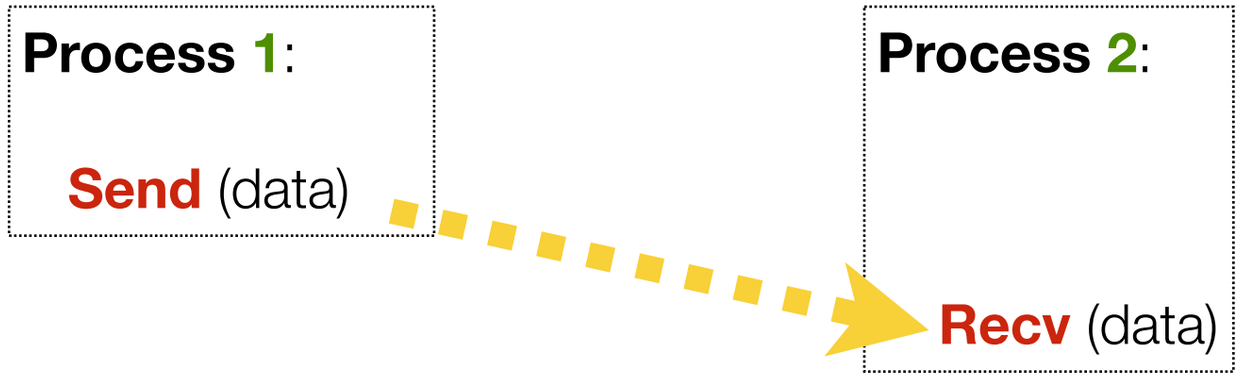
Message Passing Interface (MPI)

- Logical processes/tasks with distinct address spaces
- Communication primitives
  - Pairwise, or "**point-to-point**," send & recieve
  - **Collectives** on subsets of processes: broadcast, scatter/gather, reduce
- **Barrier** synchronization
- Advanced interface: **topology**, **one-sided**, **I/O**
- **Profiling** interface
- See: http://www.mpi-forum.org ; https://computing.llnl.gov/tutorials/mpi/

Hello, world in MPI

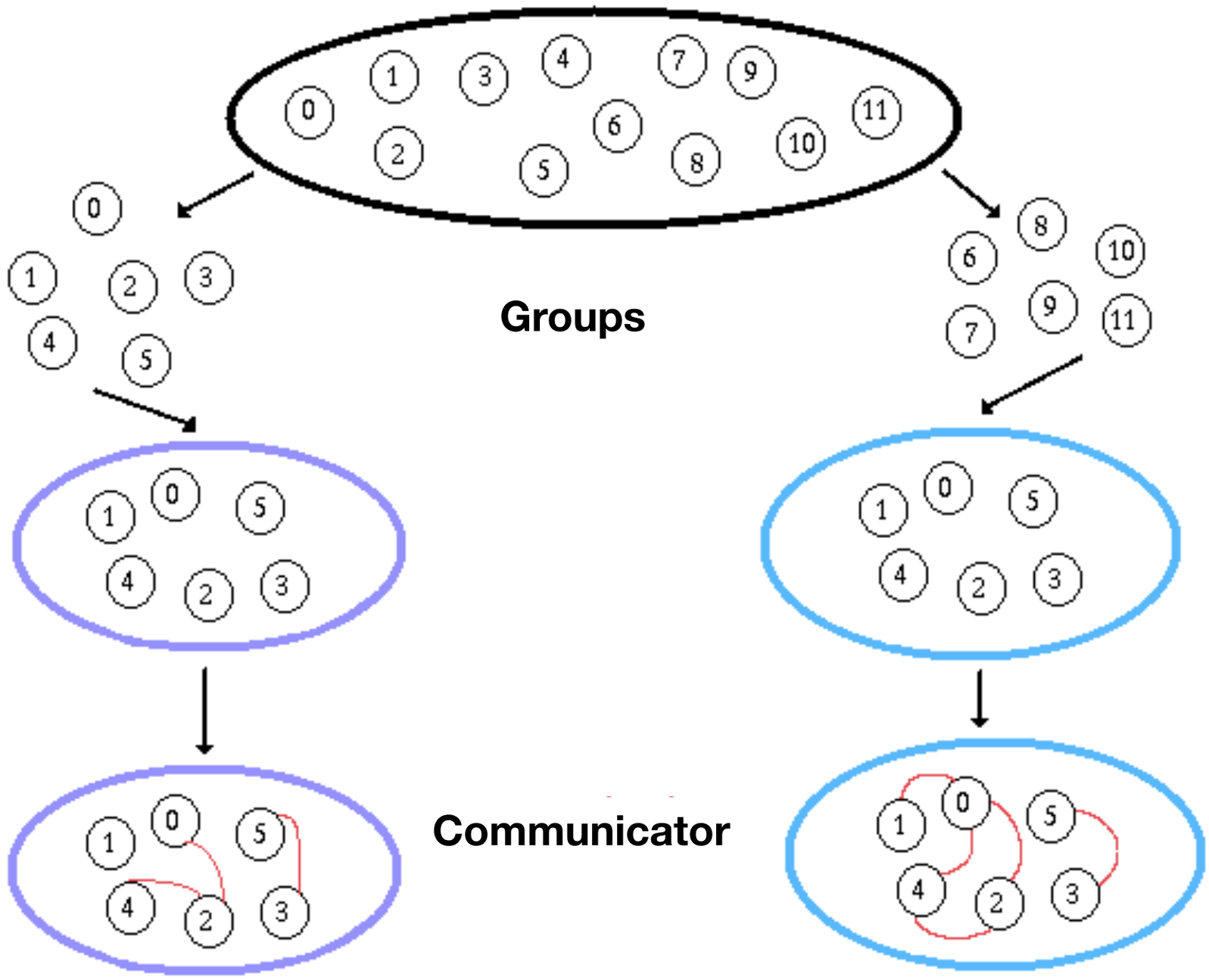
```
#include "mpi.h"
#include <stdio.h>

int main (int argc, char *argv[] )
{
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf ("I am %d of %d\n", rank, size);
    MPI_Finalize ();
    return 0;
}
```

Basic concepts: Send and receive



- How to **describe "data?"**
- How to **identify** processes?
- How will **receiver** recognize and screen messages?
- What does it mean for operations to **complete**?

MPI_COMM_WORLD



Basic concepts: Communicators

- **Group** = subset of processes
- **Communicator** = Group + attributes (*e.g.*, topology)
- **Rank** = process ID in its communicator
- **MPI_COMM_WORLD** = Group consisting of all processes
- **MPI_ANY_RANK** = Wildcard rank

Basic concepts: Data types

- In MPI call, “data” = (**address**, **count**, **type**)
- Data type = (recursively defined)
  - “Standard” scalar types: **MPI_INT**, **MPI_DOUBLE**, **MPI_CHAR**, …
  - An array of data types
  - A strided block of data types
  - An indexed array of blocks of data types
  - An arbitrary structure of data types

Basic concepts: Message tags and status objects

- **Message tags**
  - Every message has a user-defined integer ID
  - Wildcard: **MPI_ANY_TAG**

- **Status objects**: Opaque structures for querying error and other conditions

MPI blocking send:

MPI_Send (buffer-start, count, **type**, **dest-rank**, **tag**, **communicator**)

Process 1:

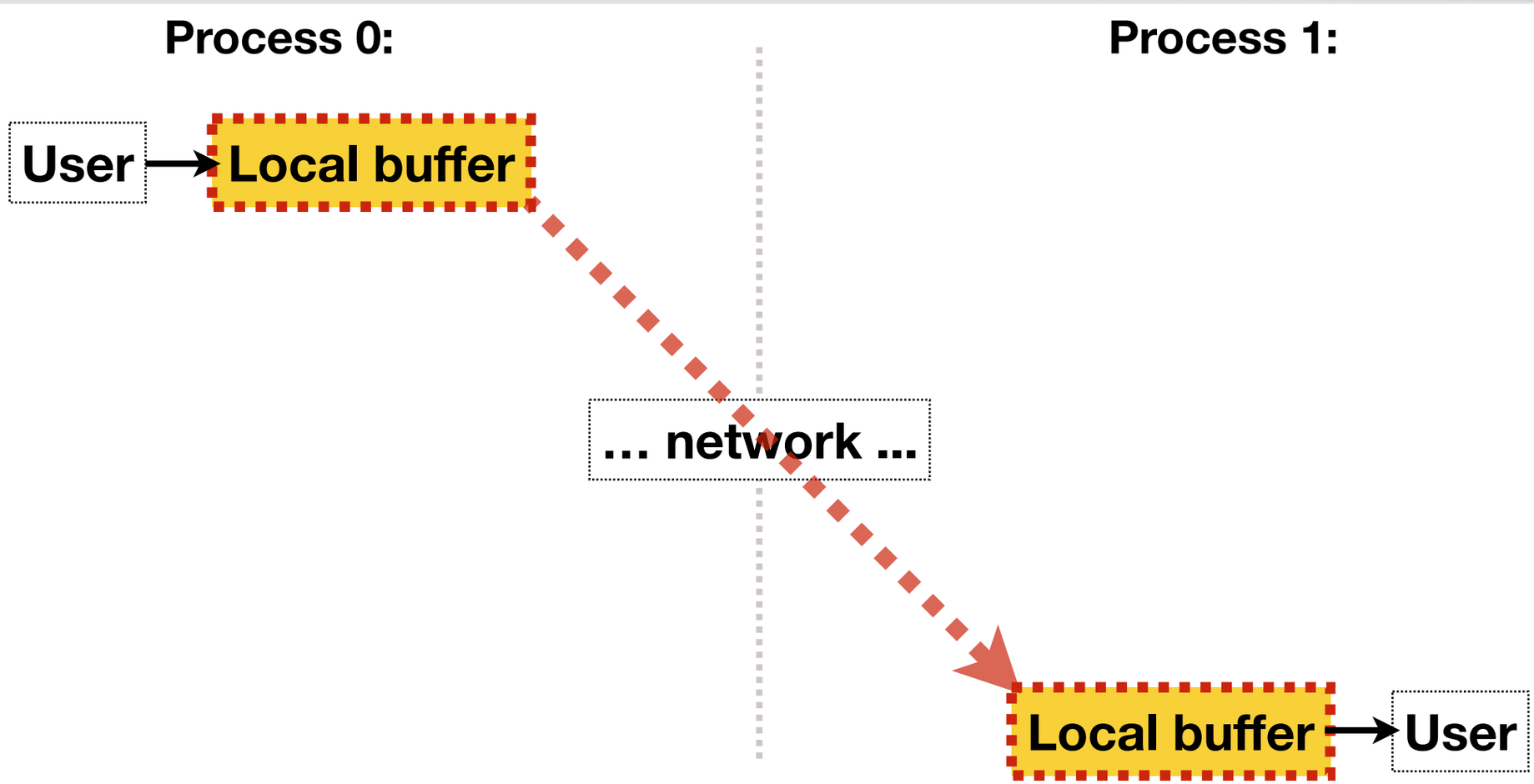
```
MPI_Send (data, n, MPI_INT, 2, tag, comm);
```

Process 2:

```
MPI_Recv (data, n, MPI_INT, 1, tag, comm, &stat);
```

- **Buffer** = (buffer-start, count, type) ; **Target** = (dest-rank, tag, comm)
- On return:
  - Data delivered to “the system”
  - May **reuse** buffer
  - Semantic note: Target may **not yet** have received message

What happens to data on “send”?



MPI blocking receive:

MPI_Recv (buffer-start, count, **type**, **source-rank**, **tag**, **comm, status**)

Process 1:

```
MPI_Send (data, n, MPI_INT, 2, tag, comm);
```

Process 2:

```
MPI_Recv (data, n, MPI_INT, 1, tag, comm, &stat);
```

- **Buffer** = (buffer-start, count, type) ; **Source** = (source-rank, tag, comm)
- Returns when **matching message** received
  - Match on source triplet; wildcards OK
  - Receiving fewer than *n* items is OK, but more is an error
  - May query “status” for more information (*e.g.*, size of message)

(Potentially) Avoid copies with non-blocking communication

- Non-blocking operations **return immediately** with handles
- **Wait** on handles

```
MPI_Request req;
MPI_Status stat;

MPI_Isend (buf, n, MPI_INT, dest, tag, comm, &req);

// … do not use “buf” …

MPI_Wait (&req, &stat);
```

- May poll instead of wait (“test”), or poll or wait on multiple requests

Other communication modes

- **Synchronous** sends (**MPI_Ssend**): Send completes when receive begins
- **Buffered** mode (**MPI_Bsend**): Use user-supplied buffer
- **Ready** mode (**MPI_Rsend**): User guarantees matching receive has posted
- Non-blocking versions of above
- **MPI_Recv** accepts messages sent in any mode
- **MPI_Sendrecv** initiates simultaneous send and receive

Beware of deadlock

- **"Unsafe" orderings** of send/receive

(a)

Process 1:
Recv (data → **2**)
Send (data ← **2**)

Process 2:
Recv (data → **1**)
Send (data ← **1**)

(b)

Process 1:
Send (data → **2**)
Recv (data ← **2**)

Process 2:
Send (data → **1**)
Recv (data ← **1**)

- How to avoid?

Ways to avoid deadlock

- Use **safe orderings**

Process 1:
Send (data → **2**)
Recv (data ← **2**)

Process 2:
Recv (data → **1**)
Send (data ← **1**)

- Use **simultaneous** send/receive

Process 1:
Sendrecv (data → **2**)

Process 2:
Sendrecv (data → **1**)

More ways to avoid deadlock

- **Supply** buffer space

Process 1:
Bsend (data → **2**)
Recv (data ← **2**)

Process 2:
Bsend (data → **1**)
Recv (data ← **1**)

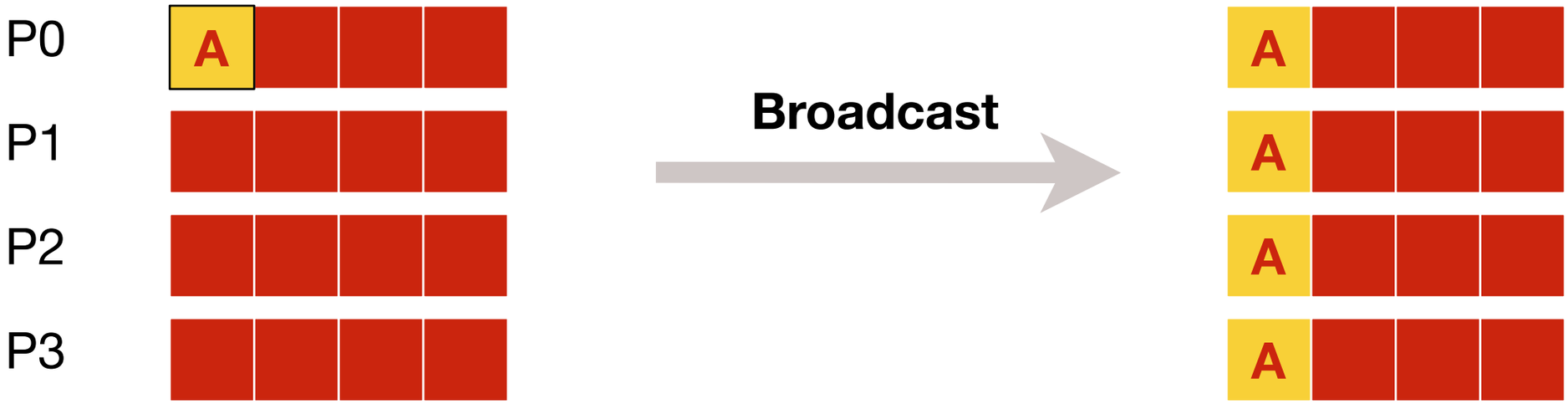
- Use **non-blocking** send/receive

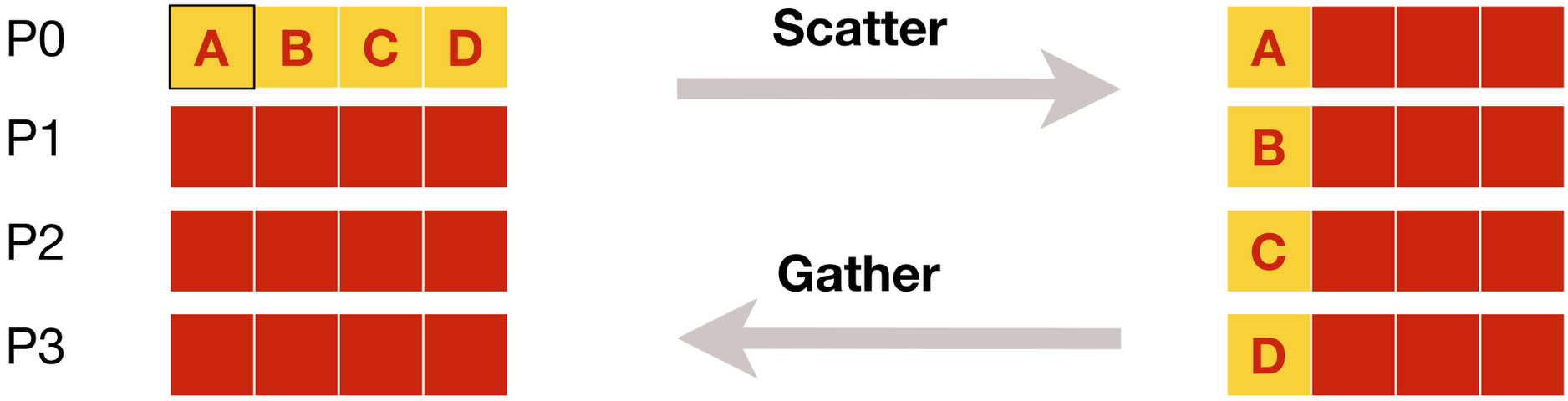
Process 1:
Isend (data1 → **2**)
Irecv (data2 ← **2**)
Waitall

Process 2:
Isend (data1 → **1**)
Irecv (data2 ← **1**)
Waitall

Collective communication

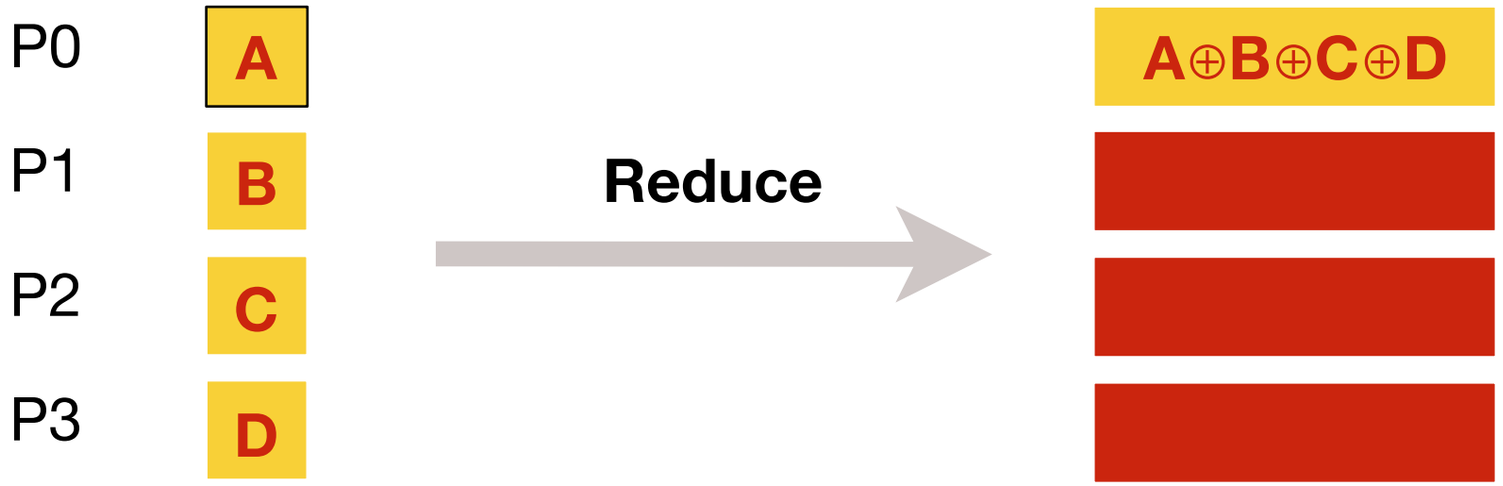
- Higher-level communication primitives
  - **MPI_Bcast**: Broadcast data to all processes
  - **MPI_Reduce**: Combine data from all processes to one process
  - **MPI_Barrier**
- Each process executes same operation
- Presumably optimized/tuned for hardware, but …

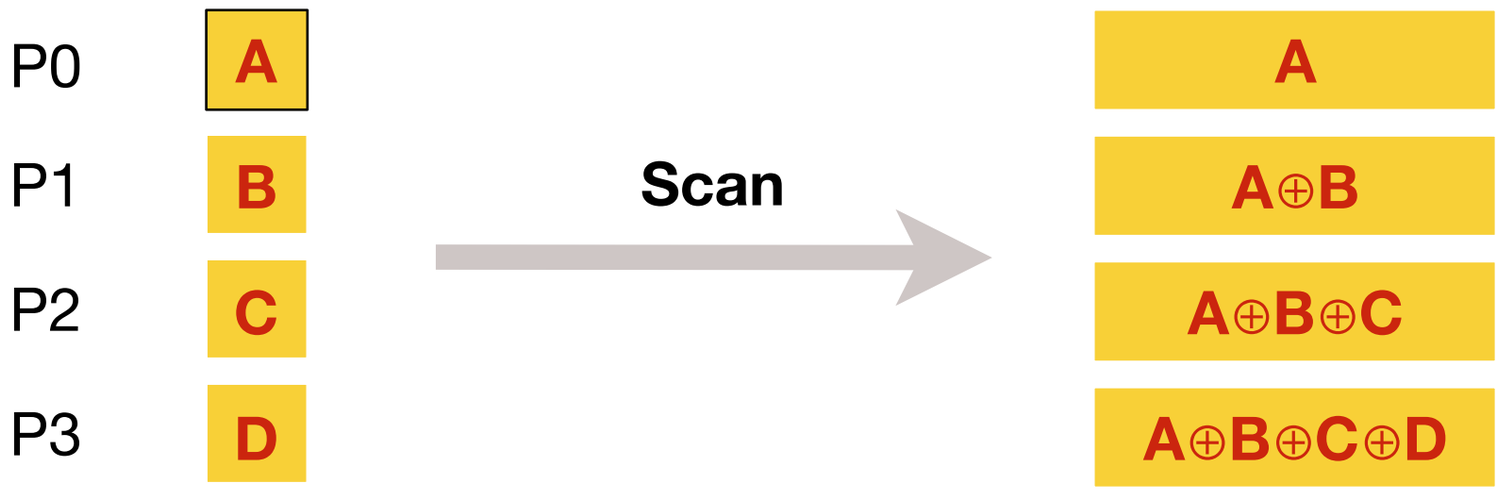

P0
P1
P2
P3
A
Broadcast
A
A
A
A


P0
P1
P2
P3
A
B
C
D
Scatter
Gather
A
B
C
D


P0
P1
P2
P3
A
B
C
D
All gather
A B C D
A B C D
A B C D
A B C D


P0
P1
P2
P3
A0 B0 C0 D0
A1 B1 C1 D1
A2 B2 C2 D2
A3 B3 C3 D3
All-to-all
A0 A1 A2 A3
B0 B1 B2 B3
C0 C1 C2 C3
D0 D1 D2 D3


P0
A
P1
B
P2
C
P3
D
Reduce
A⊕B⊕C⊕D


P0
A
P1
B
P2
C
P3
D
Scan
A
A⊕B
A⊕B⊕C
A⊕B⊕C⊕D

Summary: MPI

- Most commonly used MPI primitives
  - Init, Comm_size, Comm_rank, Send, Recv, Finalize
  - Non-blocking primitives for correctness and performance
- “Advanced” MPI features
  - Custom communicators
  - I/O
  - one-sided communication

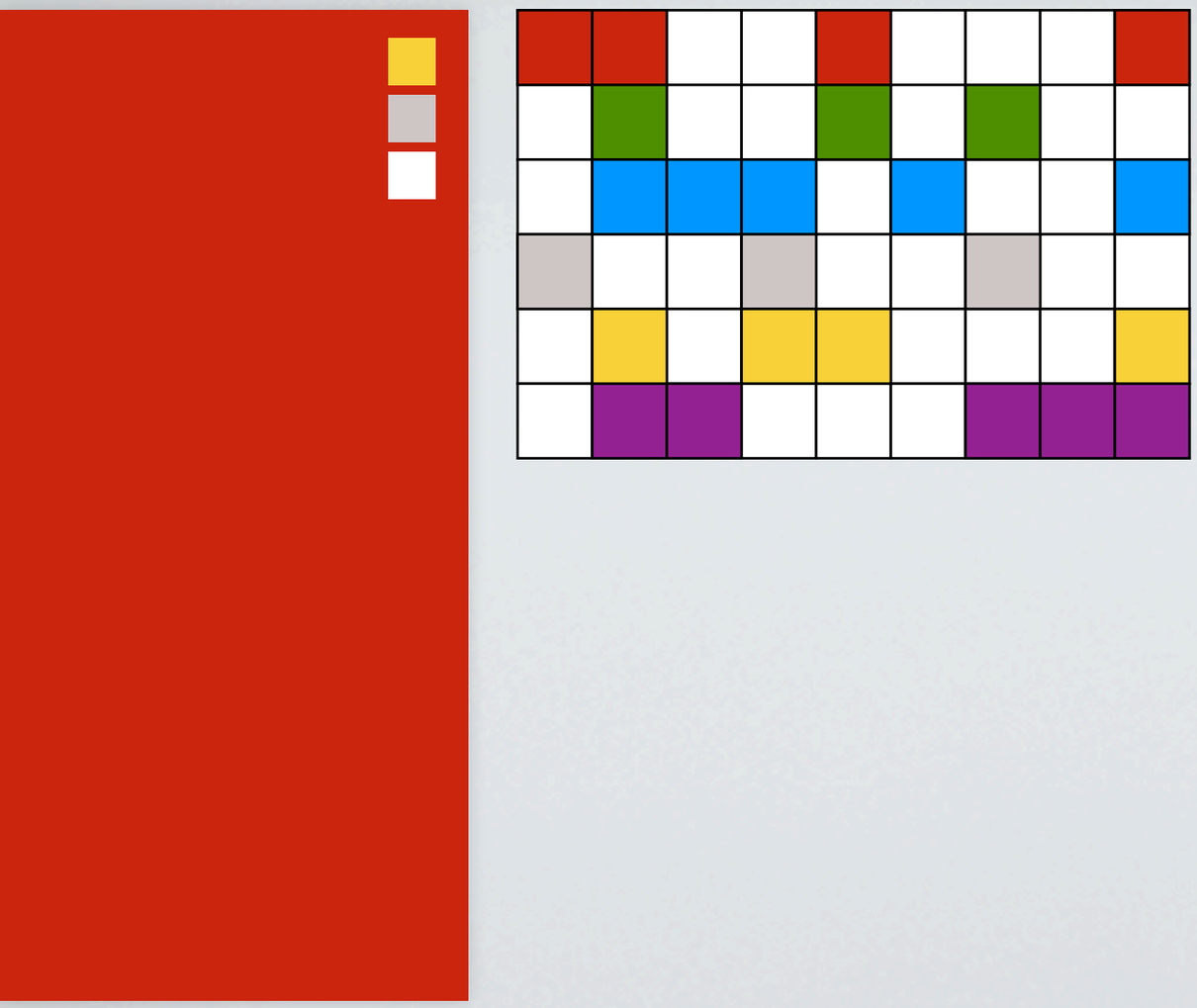
Administrivia

Administrative stuff

- Accounts: Apparently, you already have them or will soon (!)
  - **Try logging into ‘warp1’** with your UNIX account password
  - If it doesn’t work, go see TSO Help Desk (and good luck!)
    - CCB 148 / M-F 7a-5p / 404.894.7065 / AIM:tsohlpdsk
  - **IHPCL mailing list:**
    - https://mailman.cc.gatech.edu/mailman/listinfo/ihpc-lab

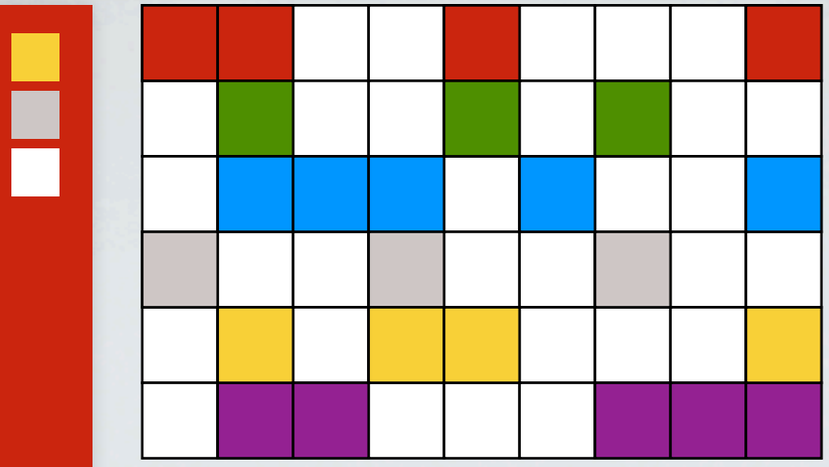
Homework 1:
Parallel conjugate gradients

- Implement a parallel solver for Ax = b (serial C version provided)
  - Evaluate on three matrices: 27-pt stencil, and two application matrices
  - “Simplified:” No preconditioning
  - **Bonus**: Reorder, precondition
- Performance models to understand scalability of your implementation
  - Make measurements
  - Build predictive models
- Collaboration encouraged: Compare programming models or platforms



Compressed sparse row (CSR)

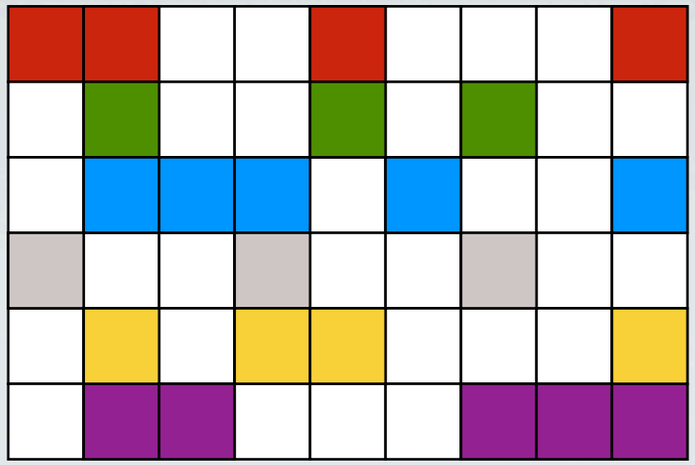
One method for implementing an “adjacency list” for a sparse graph.



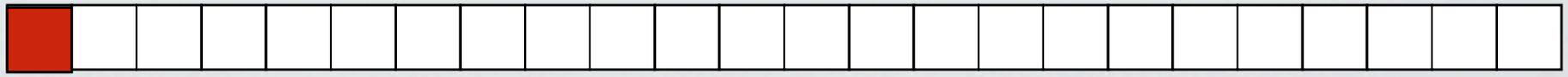
value

Compressed sparse row (CSR)

Pack by rows.

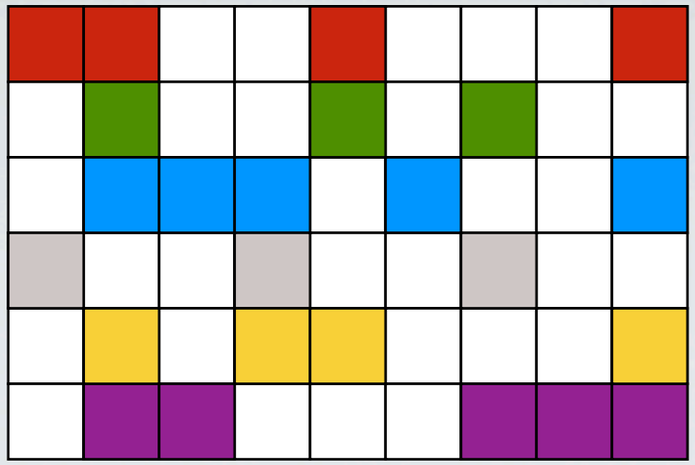


value

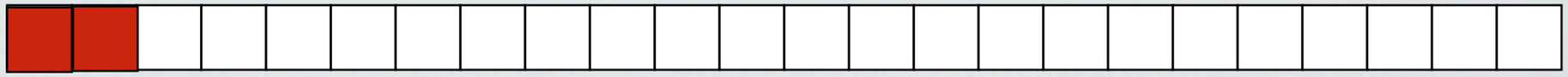


Compressed sparse row (CSR)

Pack by rows.

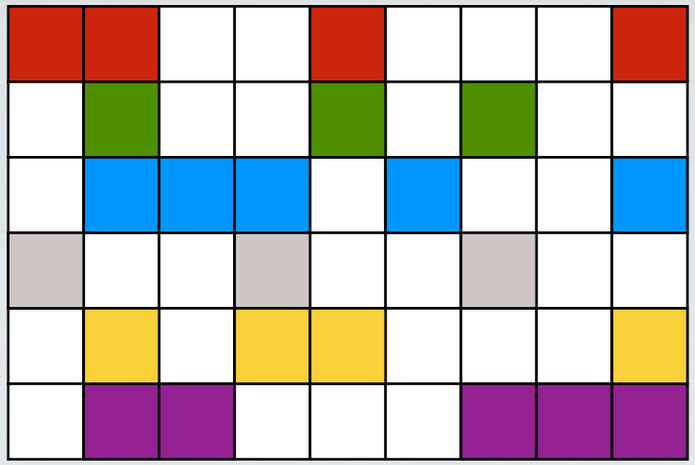


value

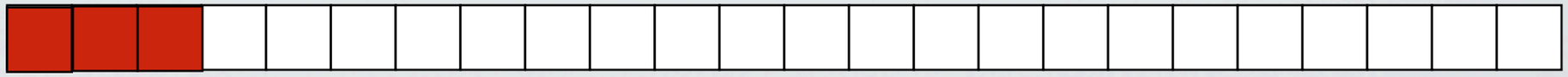


Compressed sparse row (CSR)

Pack by rows.

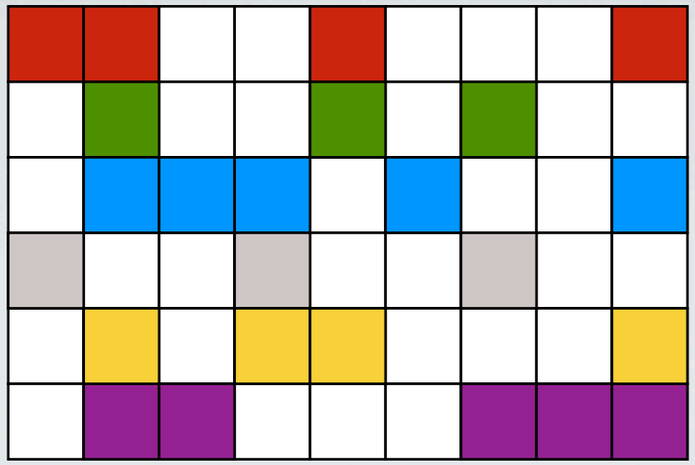


value

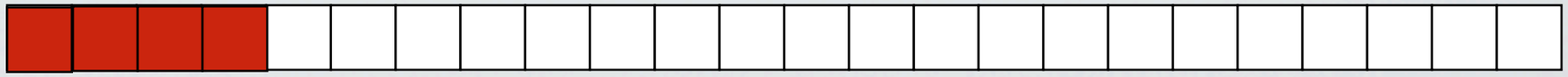


Compressed sparse row (CSR)

Pack by rows.

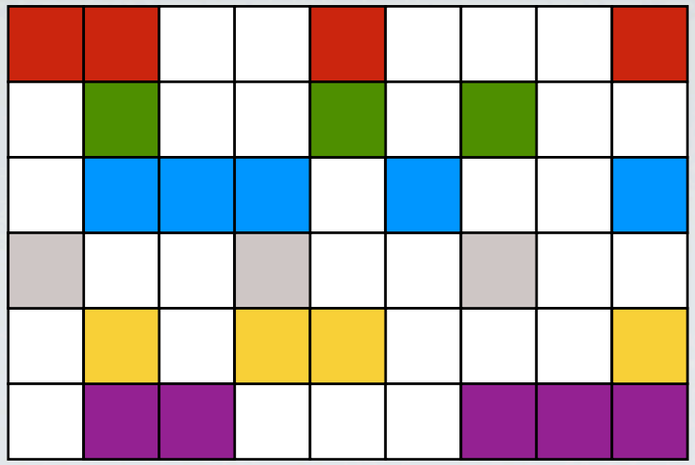


value

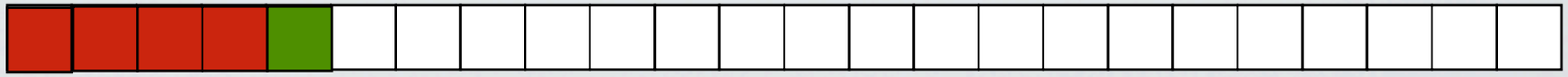


Compressed sparse row (CSR)

Pack by rows.

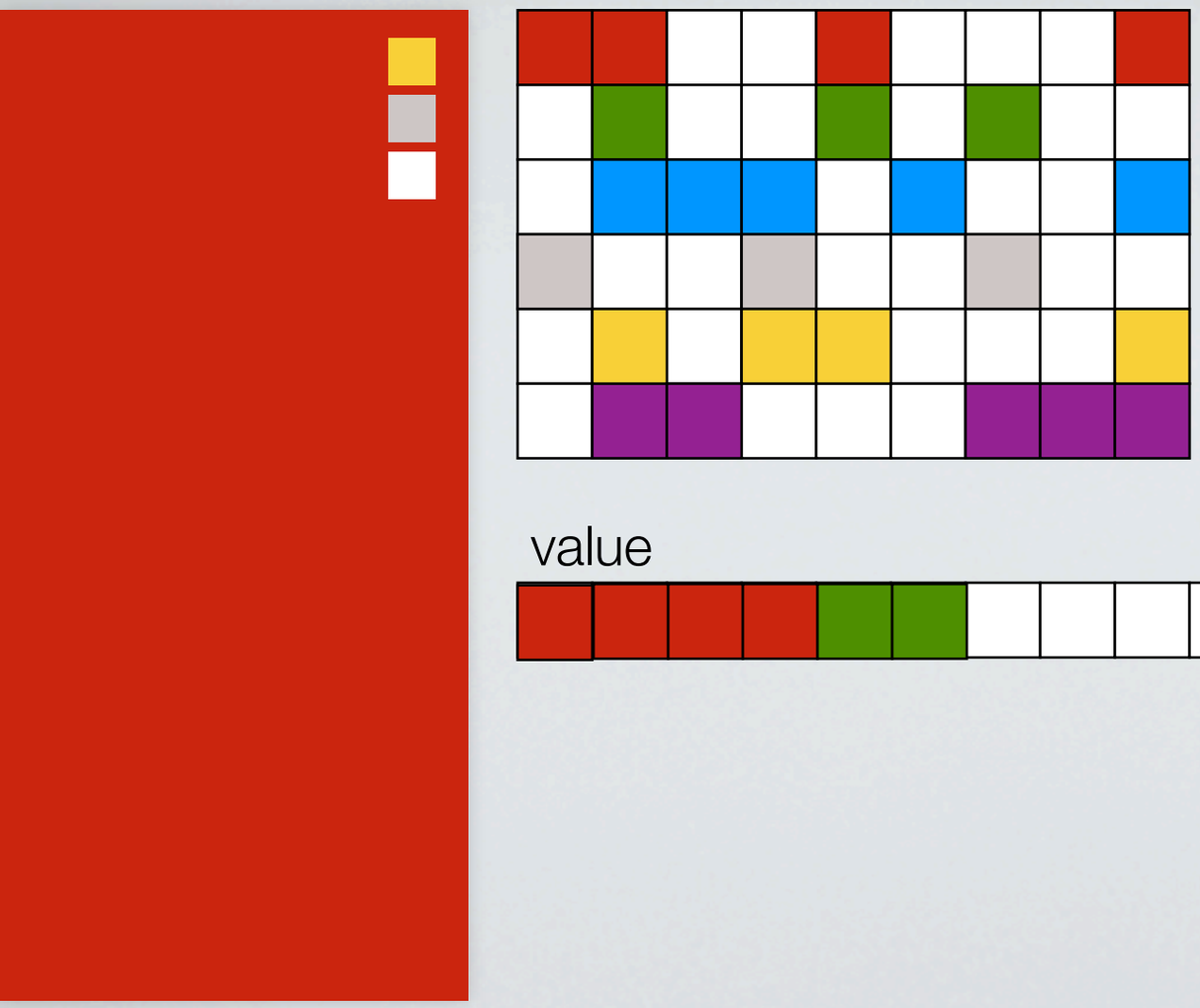


value

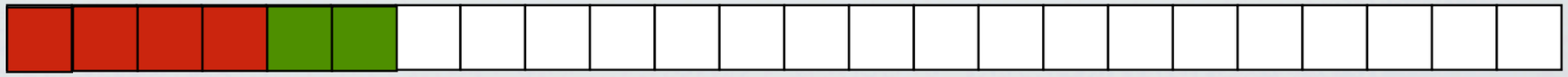


Compressed sparse row (CSR)

Pack by rows.

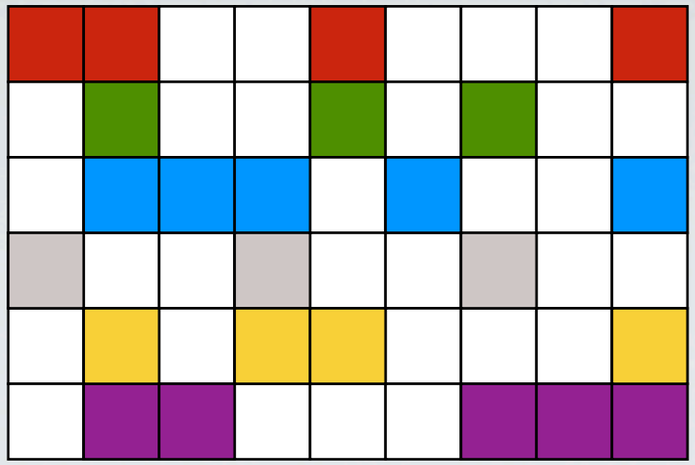


value

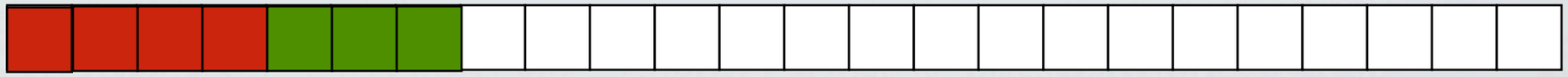


Compressed sparse row (CSR)

Pack by rows.

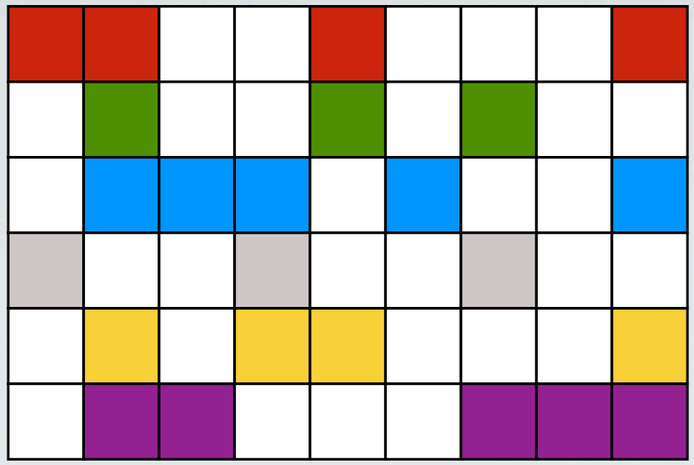


value



Compressed sparse row (CSR)

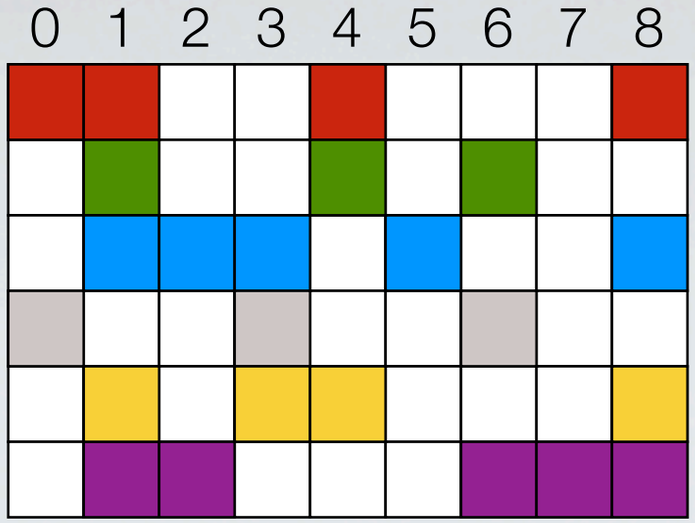
Pack by rows.



value

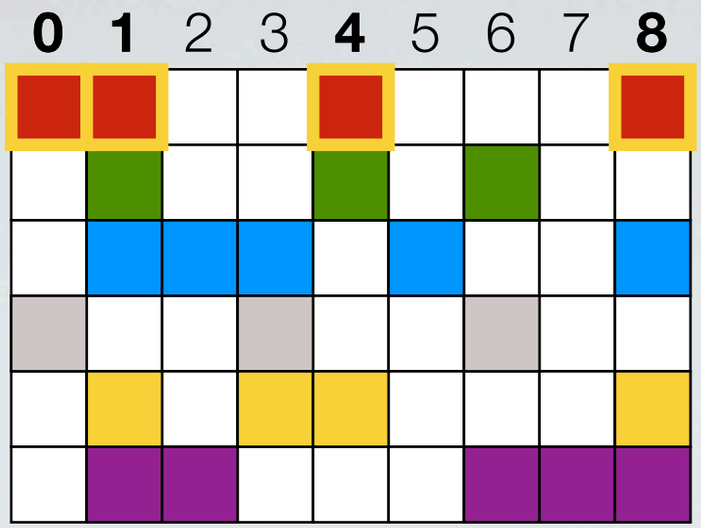
Compressed sparse row (CSR)

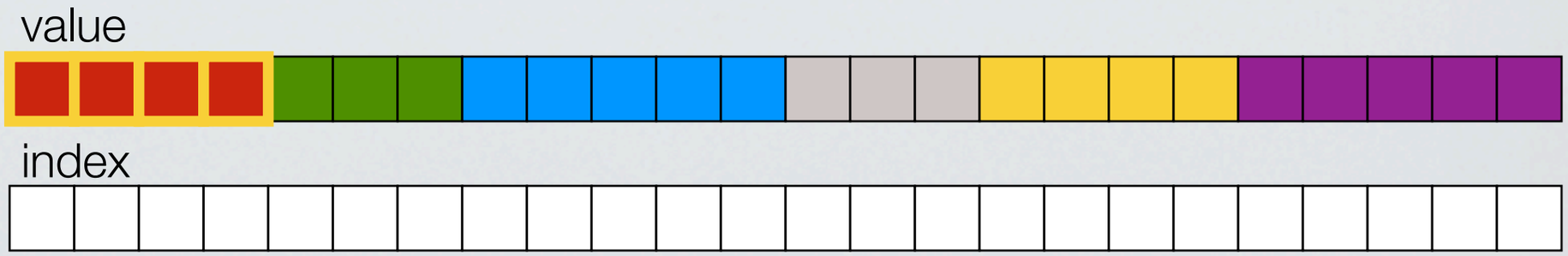
Pack by rows.



Compressed sparse row (CSR)

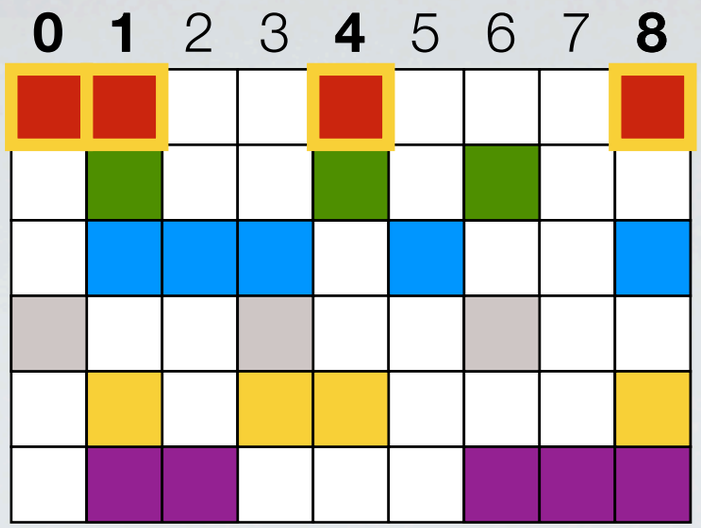
Record column indices.

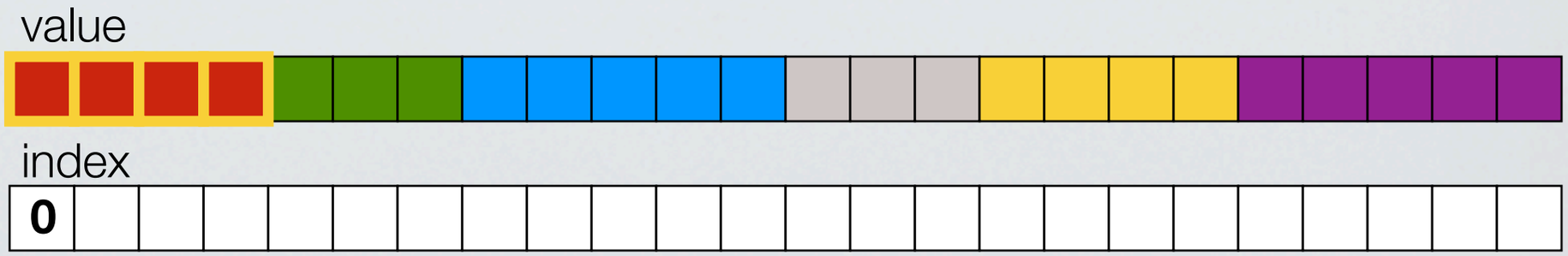




Compressed sparse row (CSR)

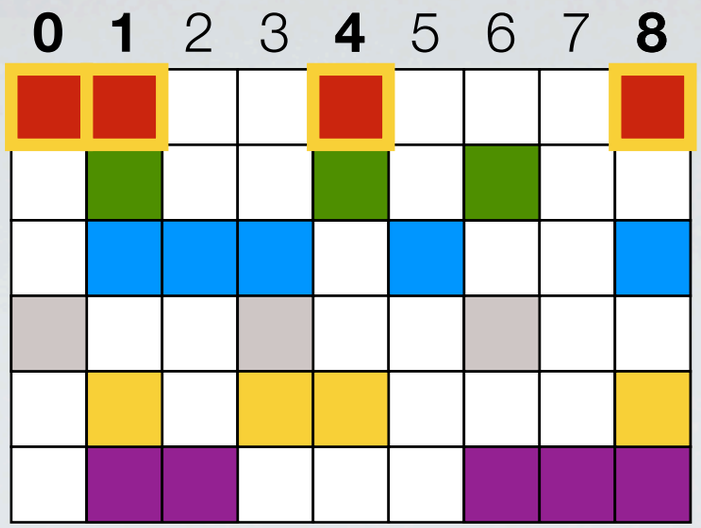
Record column indices.

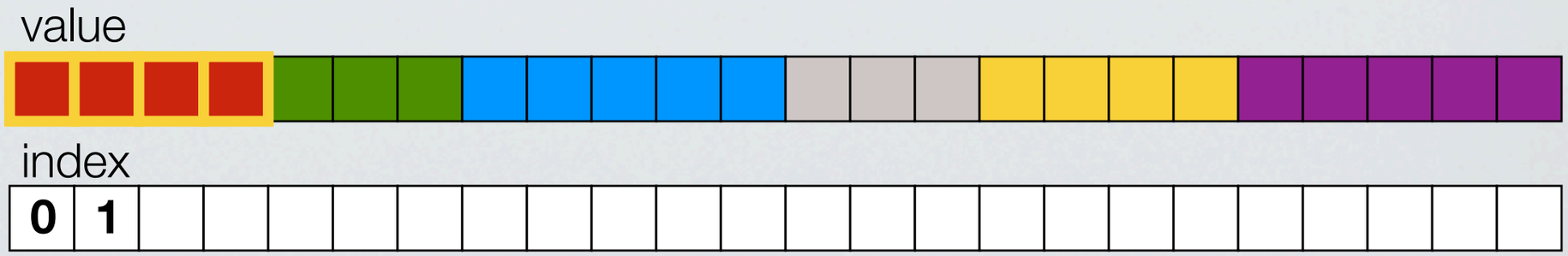




Compressed sparse row (CSR)

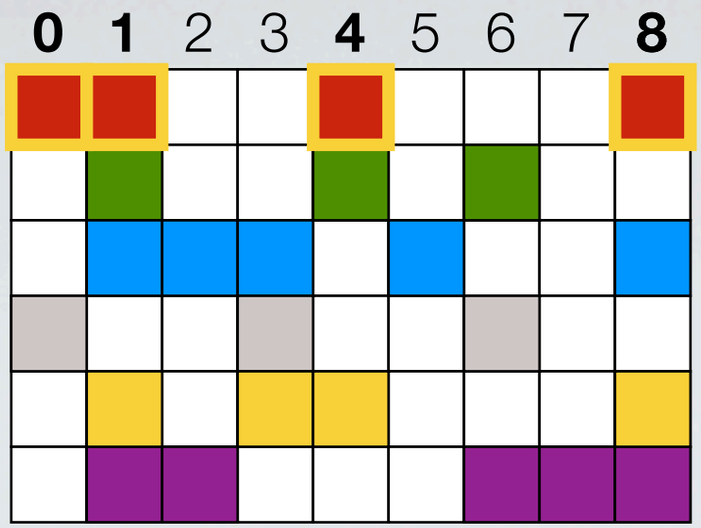
Record column indices.

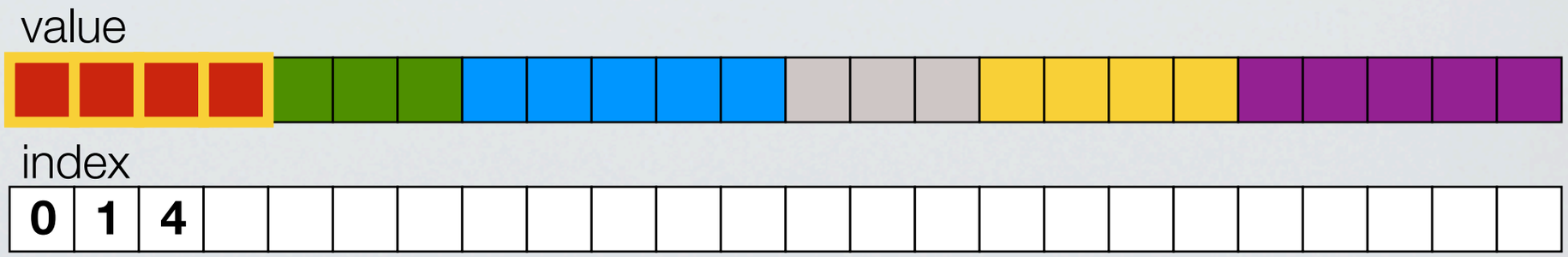




Compressed sparse row (CSR)

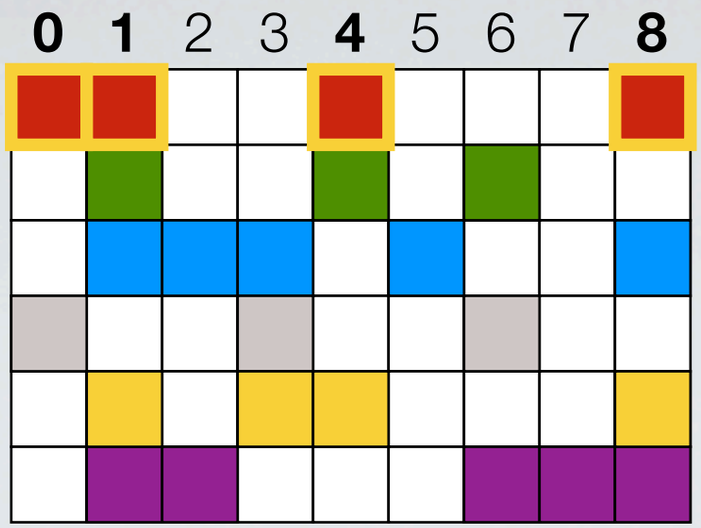
Record column indices.

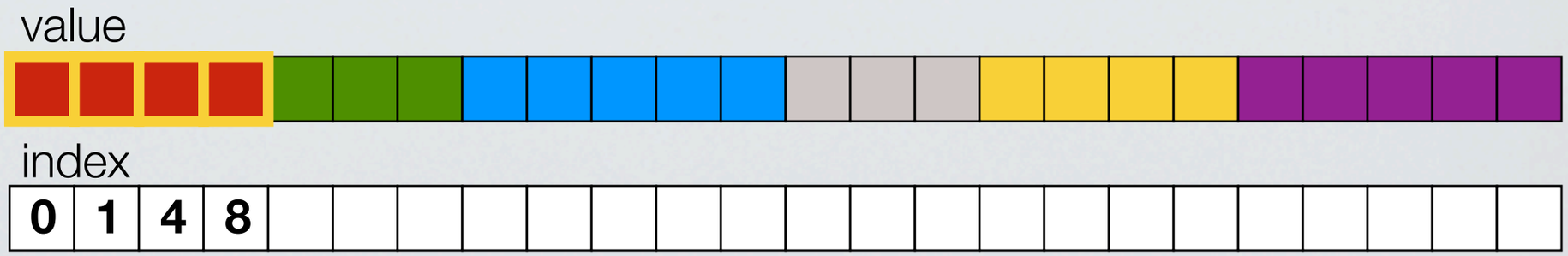




Compressed sparse row (CSR)

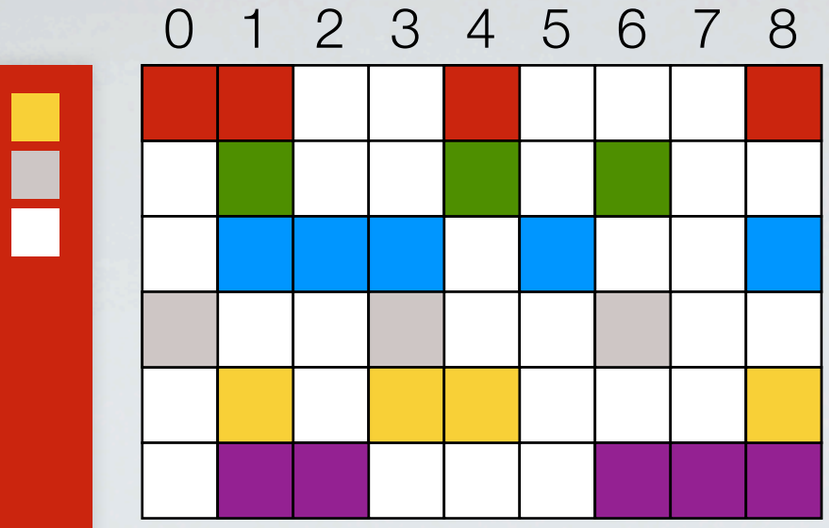
Record column indices.

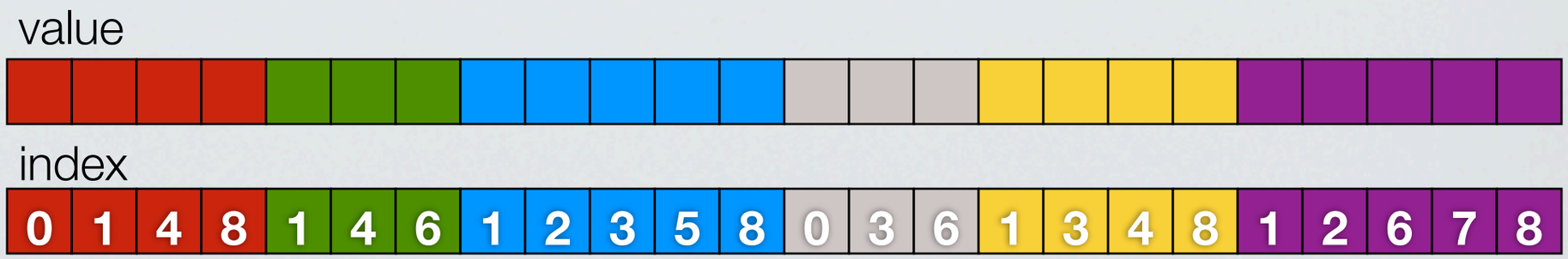




Compressed sparse row (CSR)

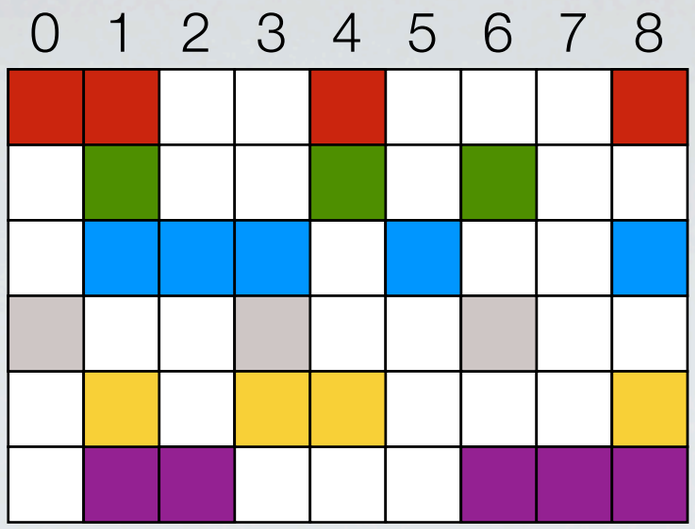
Record column indices.





Compressed sparse row (CSR)

Record column indices.



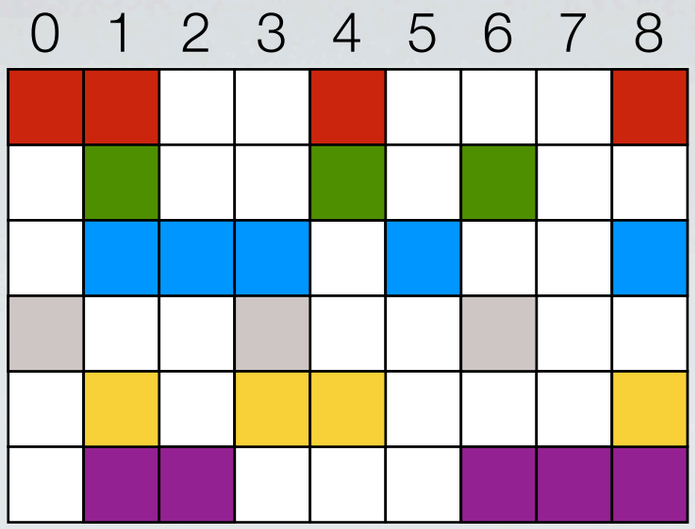
value

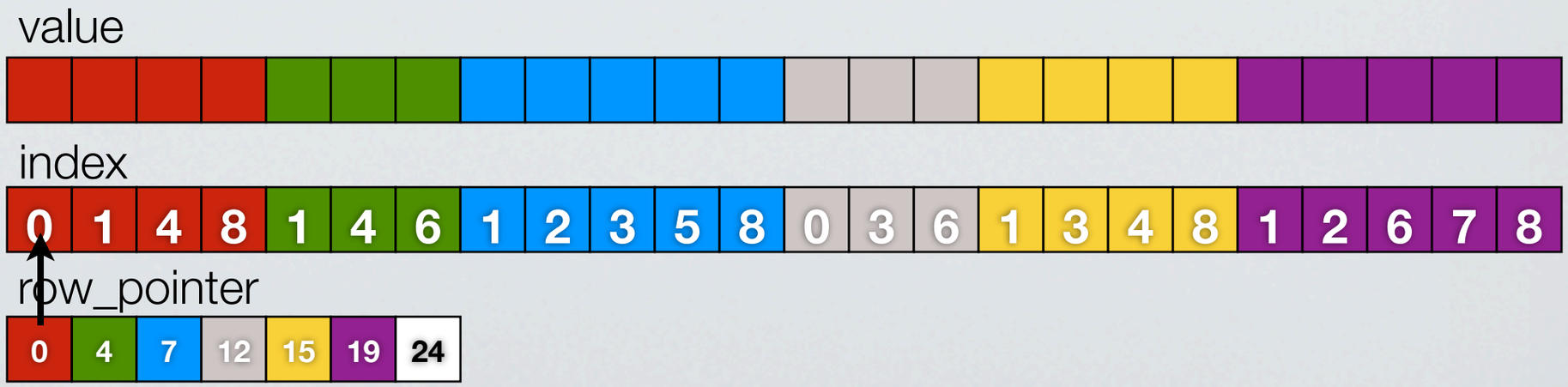
index

row_pointer

Compressed sparse row (CSR)

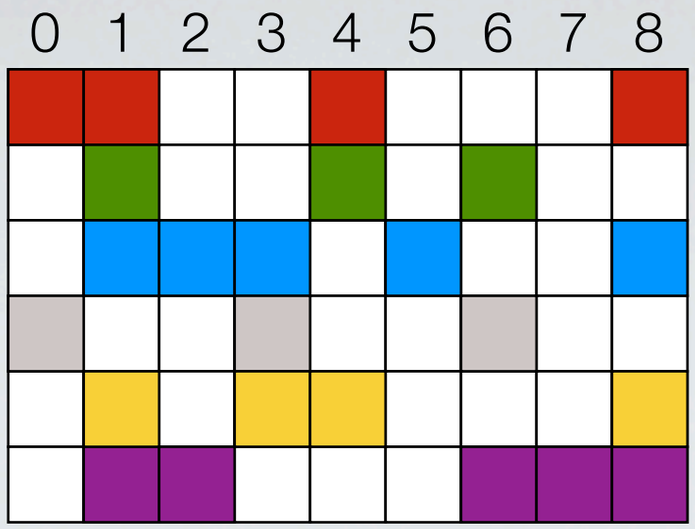
Mark start of each row in the packed format.

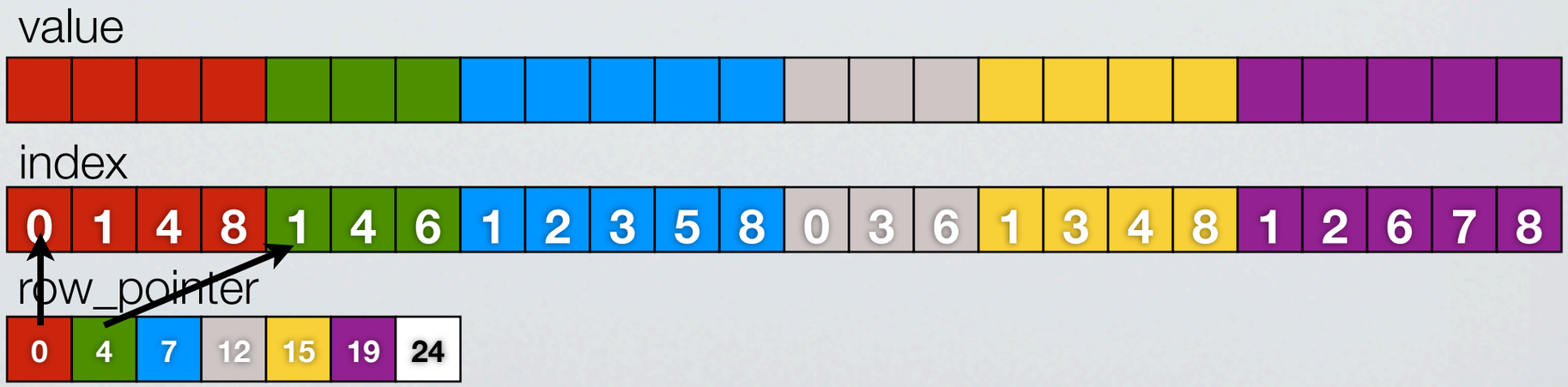




Compressed sparse row (CSR)

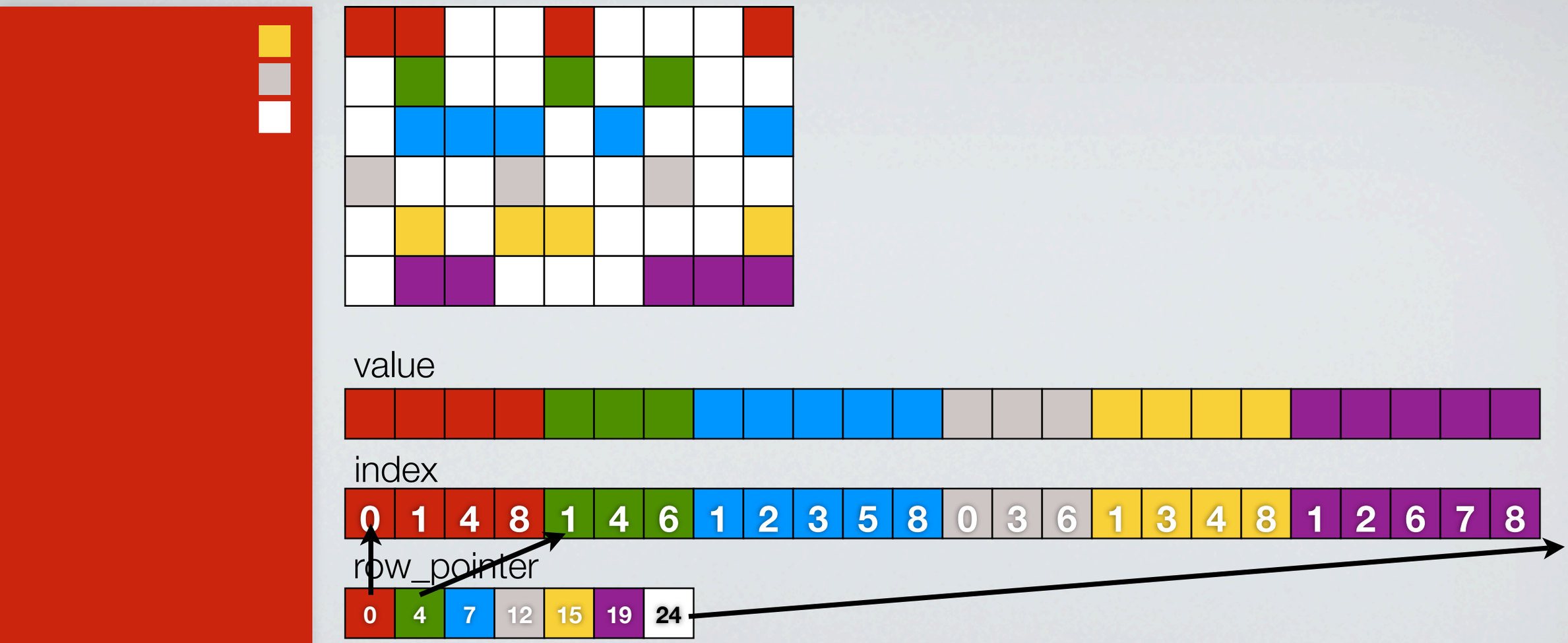
Mark start of each row in the packed format.





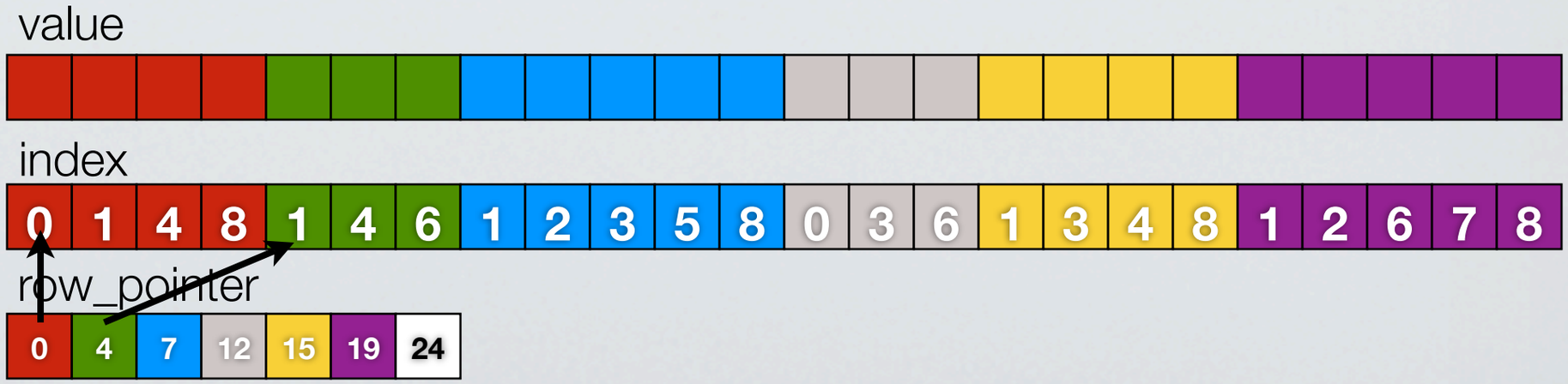
Compressed sparse row (CSR)

Mark start of each row in the packed format.



Compressed sparse row (CSR)

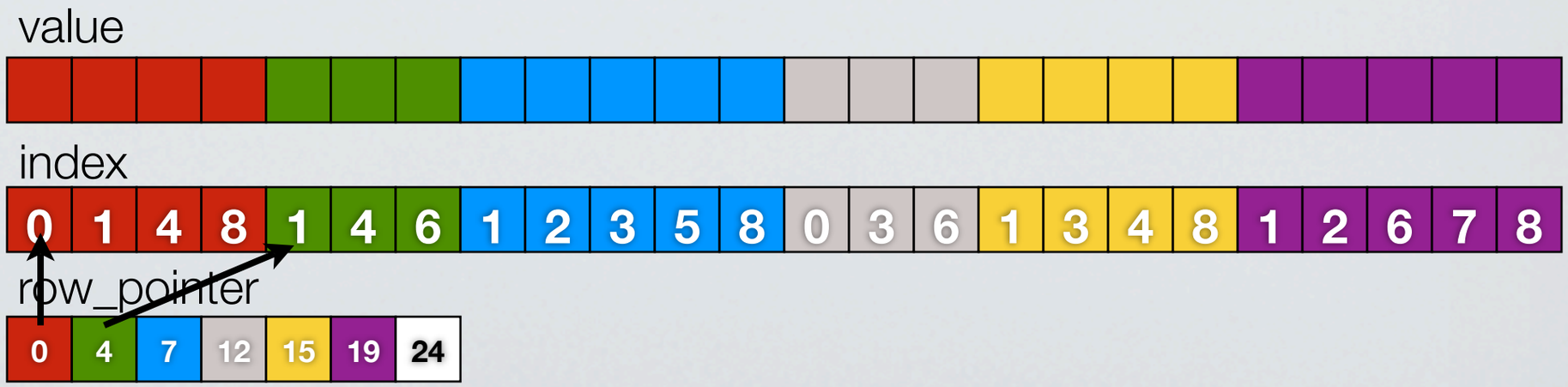
Mark start of each row in the packed format.



$$\forall j : y(j) = \sum_{i=1}^{n} W(j,i) \cdot x(i)$$

Matrix-vector multiply for a matrix in CSR format.

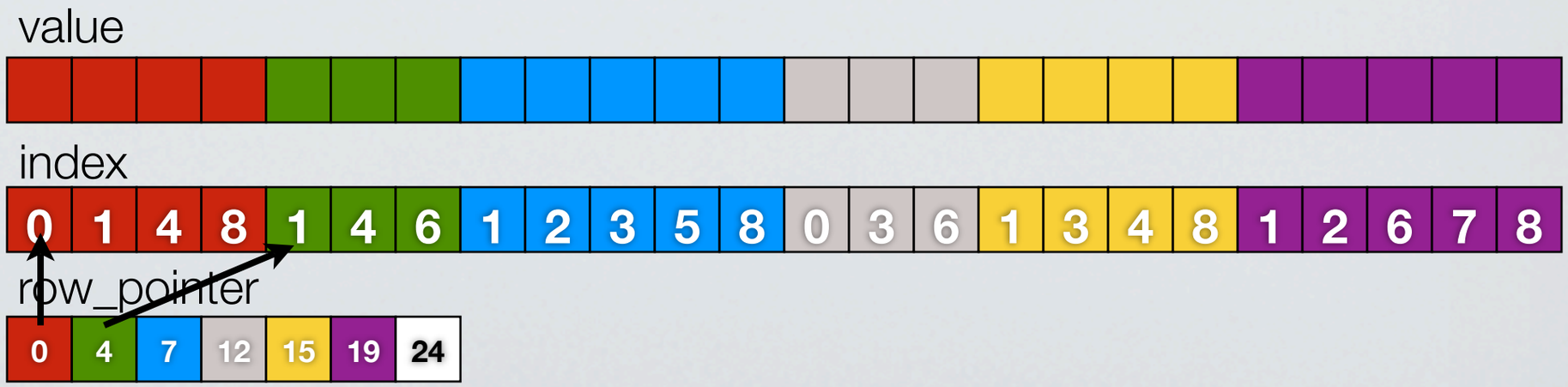
for **j** = 1 to **n** do



$$\forall j : y(j) = \sum_{i=1}^{n} W(j,i) \cdot x(i)$$

Matrix-vector multiply for a matrix in CSR format.

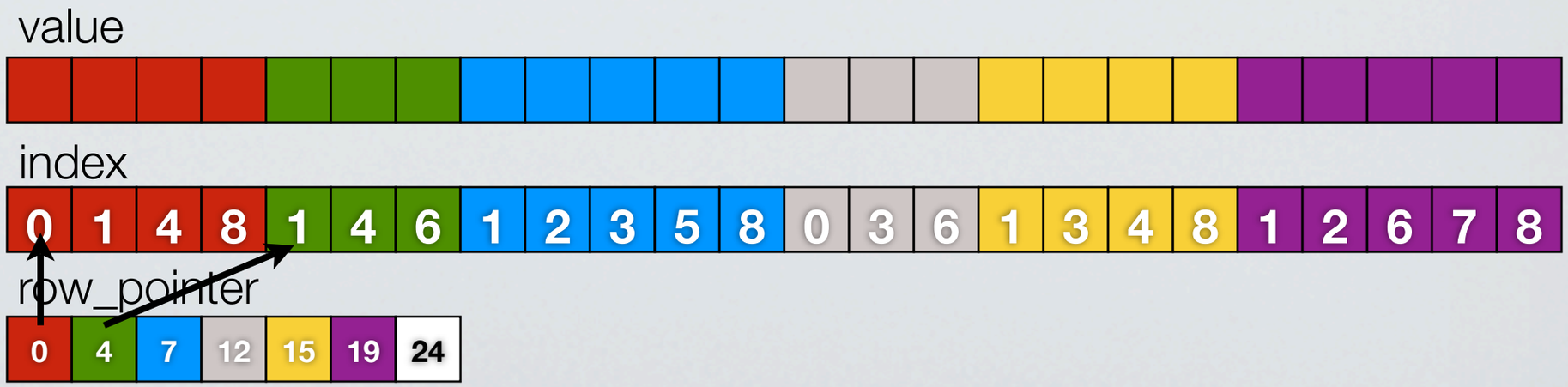
```
for j = 1 to n do
   for k = row_pointer[j] to row_pointer[j+1]-1 do
```



$$\forall j : y(j) = \sum_{i=1}^{n} W(j,i) \cdot x(i)$$

Matrix-vector multiply for a matrix in CSR format.

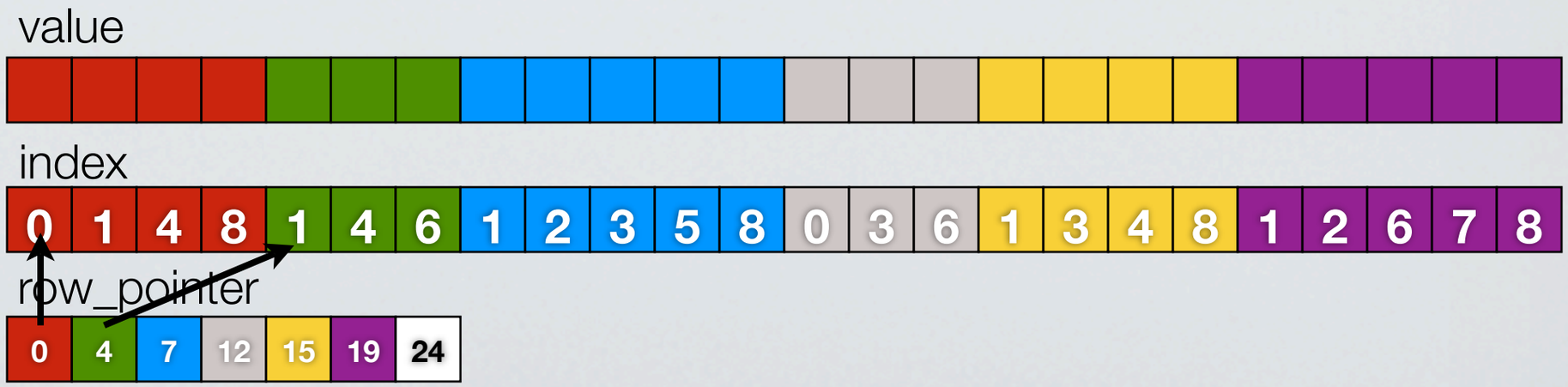
```
for j = 1 to n do
   for k = row_pointer[j] to row_pointer[j+1]-1 do
      i = index[k]
```



$$\forall j : y(j) = \sum_{i=1}^{n} W(j,i) \cdot x(i)$$

Matrix-vector multiply for a matrix in CSR format.

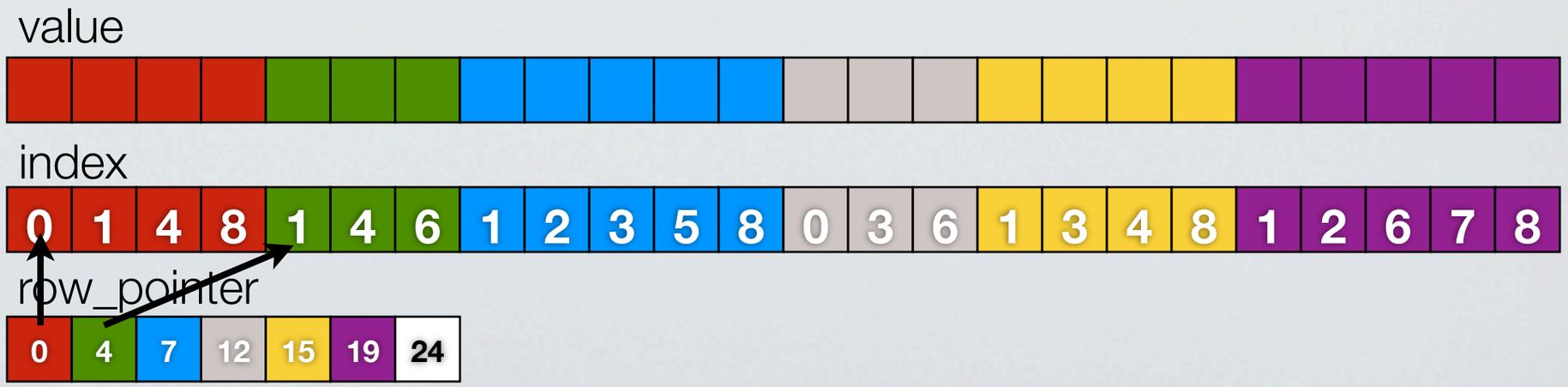
```
for j = 1 to n do
   for k = row_pointer[j] to row_pointer[j+1]-1 do
      i = index[k]
      w_ij = value[k]
```



$$\forall j : y(j) = \sum_{i=1}^{n} W(j,i) \cdot x(i)$$

Matrix-vector multiply for a matrix in CSR format.

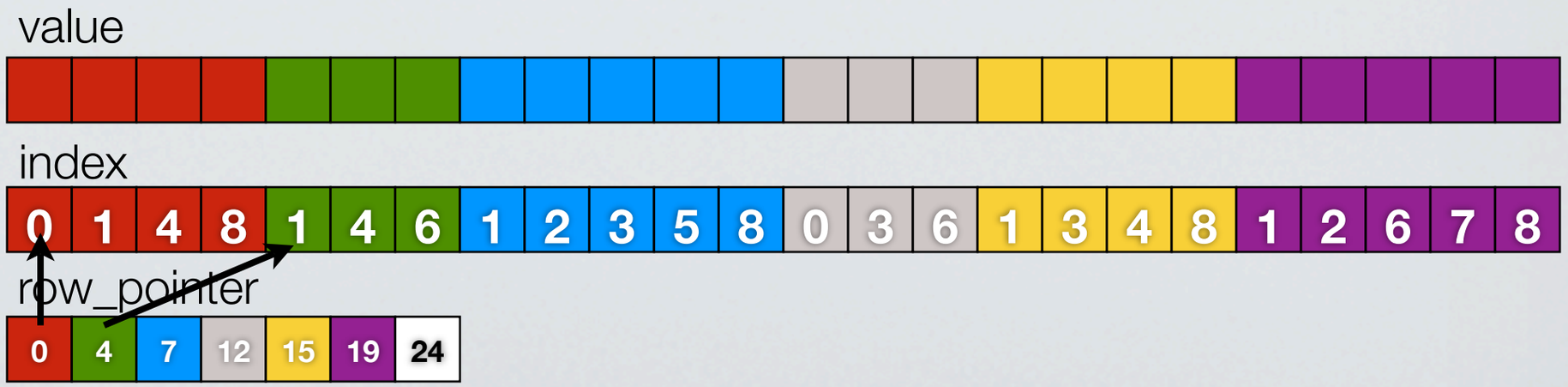
```
for j = 1 to n do
   for k = row_pointer[j] to row_pointer[j+1]-1 do
      i = index[k]
      w_ij = value[k]
      y[j] = y[j] + w_ij*x[i]
```



$$\forall j : y(j) = \sum_{i=1}^{n} W(j,i) \cdot x(i)$$

Matrix-vector multiply for a matrix in CSR format.

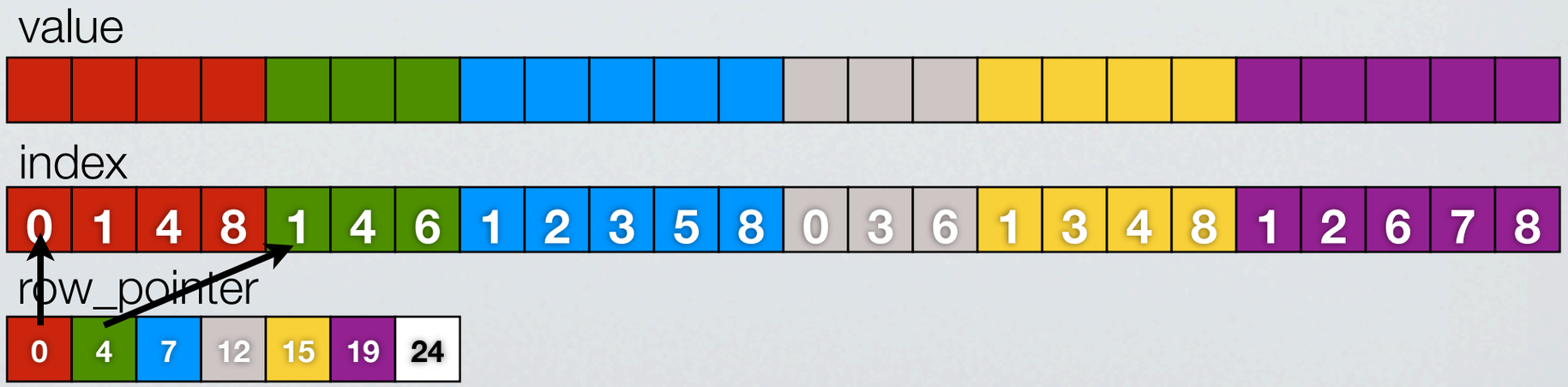
```
for j = 1 to n do
   for k = row_pointer[j] to row_pointer[j+1]-1 do
      i = index[k]
      w_ij = value[k]
      y[j] = y[j] + w_ij*x[i]
```



$$\forall j : y(j) = \sum_{i=1}^{n} W(j,i) \cdot x(i)$$

Matrix-vector multiply for a matrix in CSR format.

```
i = index[k]
w_ij = value[k]
y[j] = y[j] + w_ij*x[i]
```



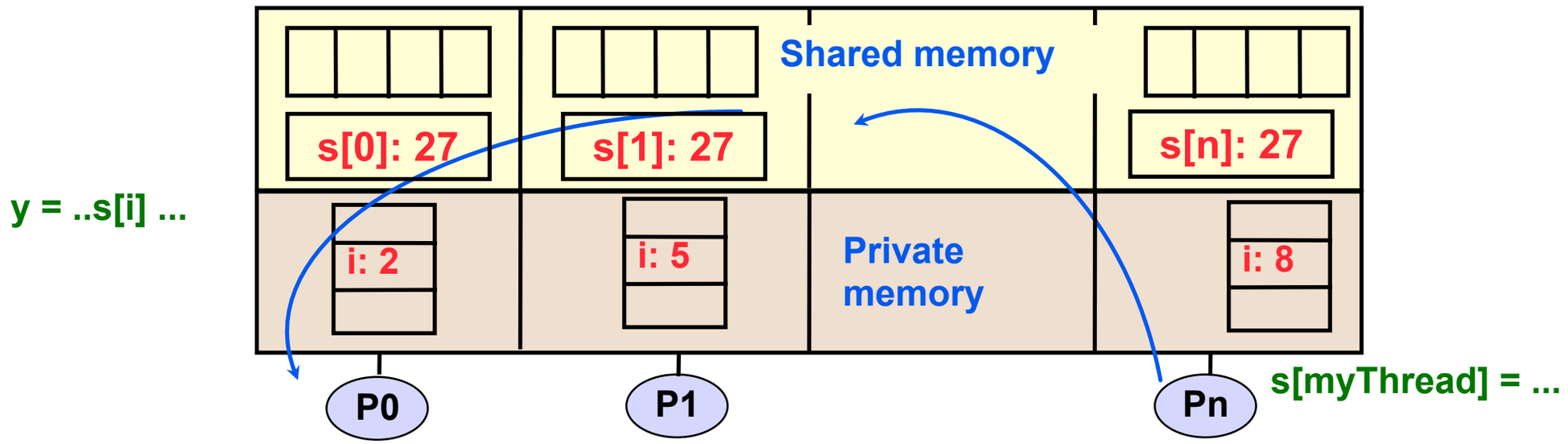
Bottleneck: Time to read *W*.

Matrix-vector multiply for a matrix in CSR format.

Unified Parallel C (UPC)

Recall the partitioned global address space (PGAS) model

- Program = **named** threads
- Shared data, but **partitioned** over local processes
- Implied cost model: **remote accesses cost more**



Unified Parallel C (UPC)

- Implements PGAS model using concise, explicit parallel extensions to C
- Aimed at “low-level” performance programmers
- Precursors: Split-C, AC, PCP
- Other PGAS languages: Co-Array Fortran, Titanium (Java)

UPC execution model: Threads running in SPMD fashion

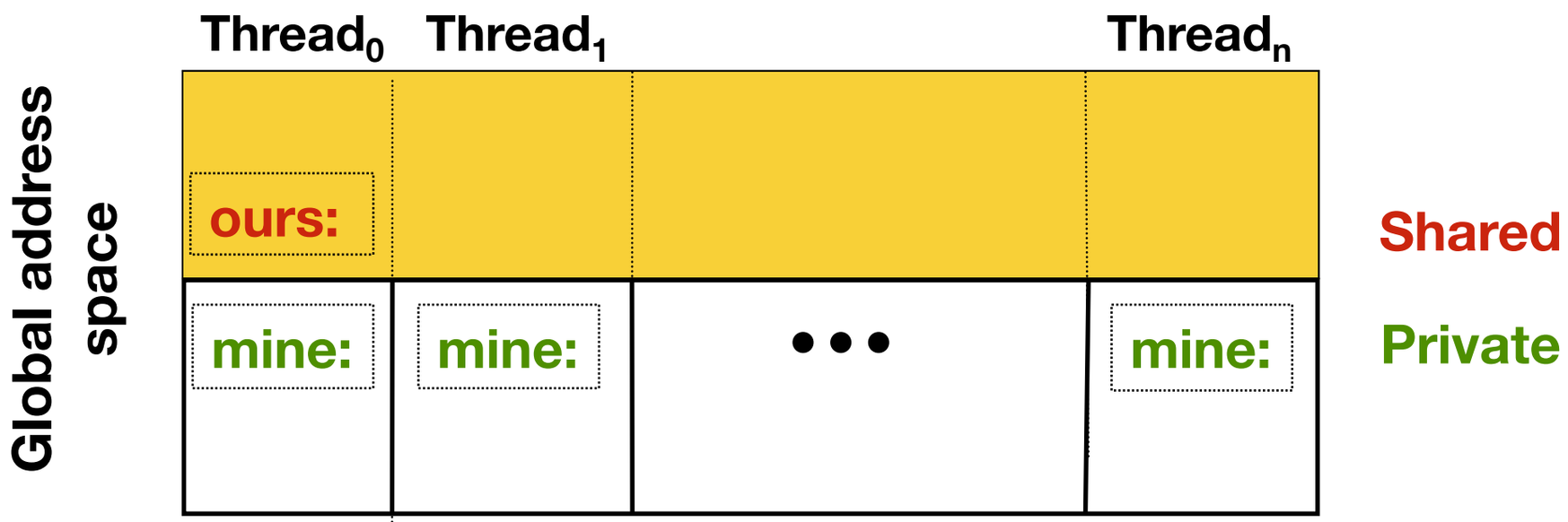
- **THREADS** = no. of threads, specified at compile- or run-time
- **MYTHREAD** = current thread's index (0 … **THREADS**-1)
- **upc_barrier**: All wait (global sync)

“Hello, world” in UPC

```
#include <upc.h>
#include <stdio.h>

int main ()
{
    printf (“[Thread %d of %d] Hello, world!\n”,
             MYTHREAD, THREADS);
    return 0;
}
```

Private vs. shared variables in UPC

```
int mine;          /* thread-private */
shared int ours; /* thread 0 */
```



Shared arrays distributed cyclically by default

```
shared int x[THREADS];     /* 1 elt per thread */
shared int y[3][THREADS]; /* 3 elt per thread */
shared int z[3][3];        /* 2 or 3 per thread */
```

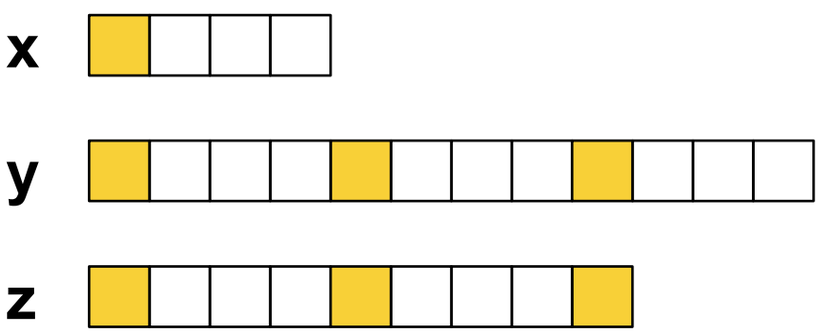
Example:

THREADS = 4

■ = "lives" on thread 0

Distribution rule:

1. Linearize
2. Distribute round-robin



Synchronization

- "Traditional" all-threads block barrier: **upc_barrier** [LABEL];
- Split-phase barrier

```
upc_notify;    /* Ready */
… do computation …
upc_wait;      /* Wait */
```

- Locks

```
upc_lock_t* l = upc_all_lock_alloc ();
   …
upc_lock (l);
   … critical code …
upc_unlock (l);
```

Collectives

- Usual suspects: broadcast, scatter, gather, reduce, prefix, …
- Interface has synchronization modes
  - Avoid over-synchronizing (barrier before/after is simplest, but may be unnecessary)
  - Data collected may be read/written by any thread
- Simple interface for collecting scalar values
  - Berkeley UPC value-based collectives
  - Reference: http://upc.lbl.gov/docs/user/README-collectivev.txt

Example: Compute sum of an array

```
shared double data[N][THREADS];

…

{
   double s_local = 0; /* local sum */
   double s;  /* global sum */

   for (i = 0; i < N; ++i)
     s_local += data[i][MYTHREAD];

   /* Reduce to sum on thread 0 */
   s = bupc_allv_reduce (double, s_local, 0, UPC_ADD);
       /* Implicit barrier */

   if (MYTHREAD == 0)
     printf (“Sum = %g\n”, s);
   …
}
```

Common idiom: Owner computes

Example: Vector addition

```
shared double A[N], B[N], Sum[N];  /* laid out cyclically */
…
{
    int i;
    for (i = 0; i < N; ++i)
      if (i % THREADS == MYTHREAD)  /* owner computes */
          Sum[i] = A[i] + B[i];
    …
}
```

Work sharing with **upc_forall**

- Special type of loop for preceding idiom

```
upc_forall (init; test; inc; affinity)
  statement;
```

- Programmer asserts iterations are independent
- "Affinity" field
  - Integer: affinity % **THREADS** == **MYTHREAD**
  - Pointer: **upc_threadof** (affinity) == **MYTHREAD**
- Compiler may do better than iterate N times

Common idiom: Owner computes

Example: Vector addition using **upc_forall**

```
int i;
upc_forall (i = 0; i < N; ++i; i)  /* Note affinity */
    Sum[i] = A[i] + B[i];
```

“In conclusion…”

Backup slides