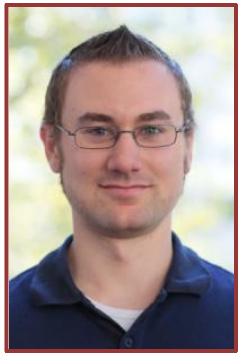
Shoal: smart allocation and replication of memory for parallel programs



Stefan Kaestle, Reto Achermann, Timothy Roscoe, Tim Harris[$]

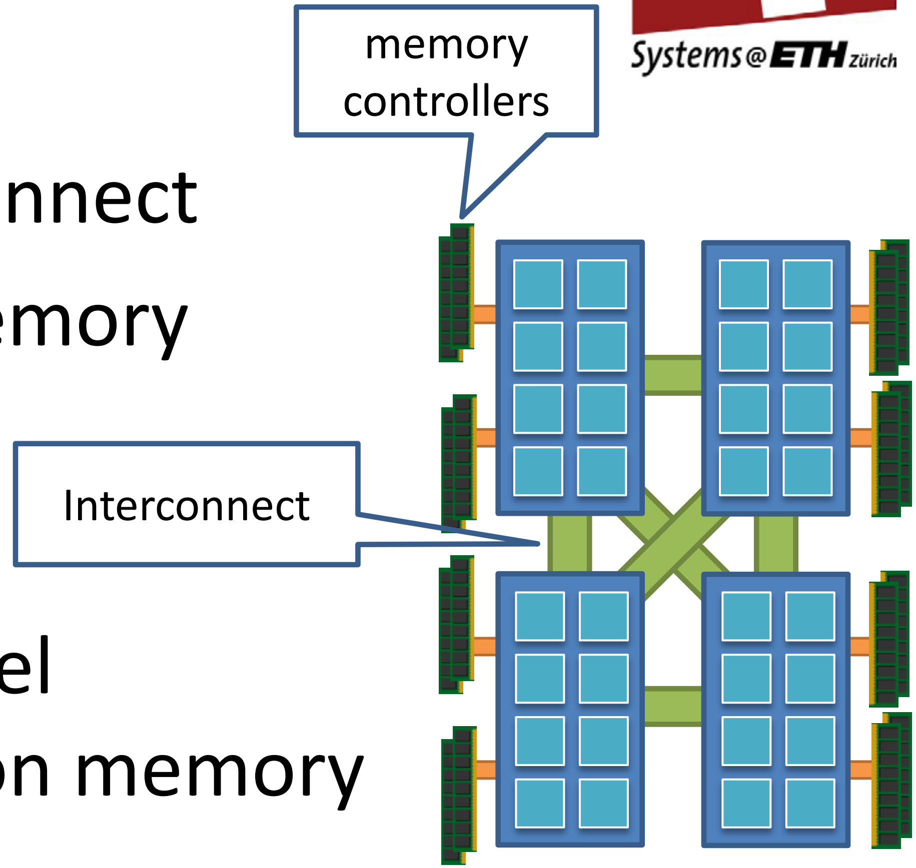
ETH Zurich [$]Oracle Labs Cambridge, UK

Systems@ETH zürich

ETH zürich

Problem

- Congestion on interconnect
- Load imbalance of memory controllers
- Performance of parallel application depends on memory allocation



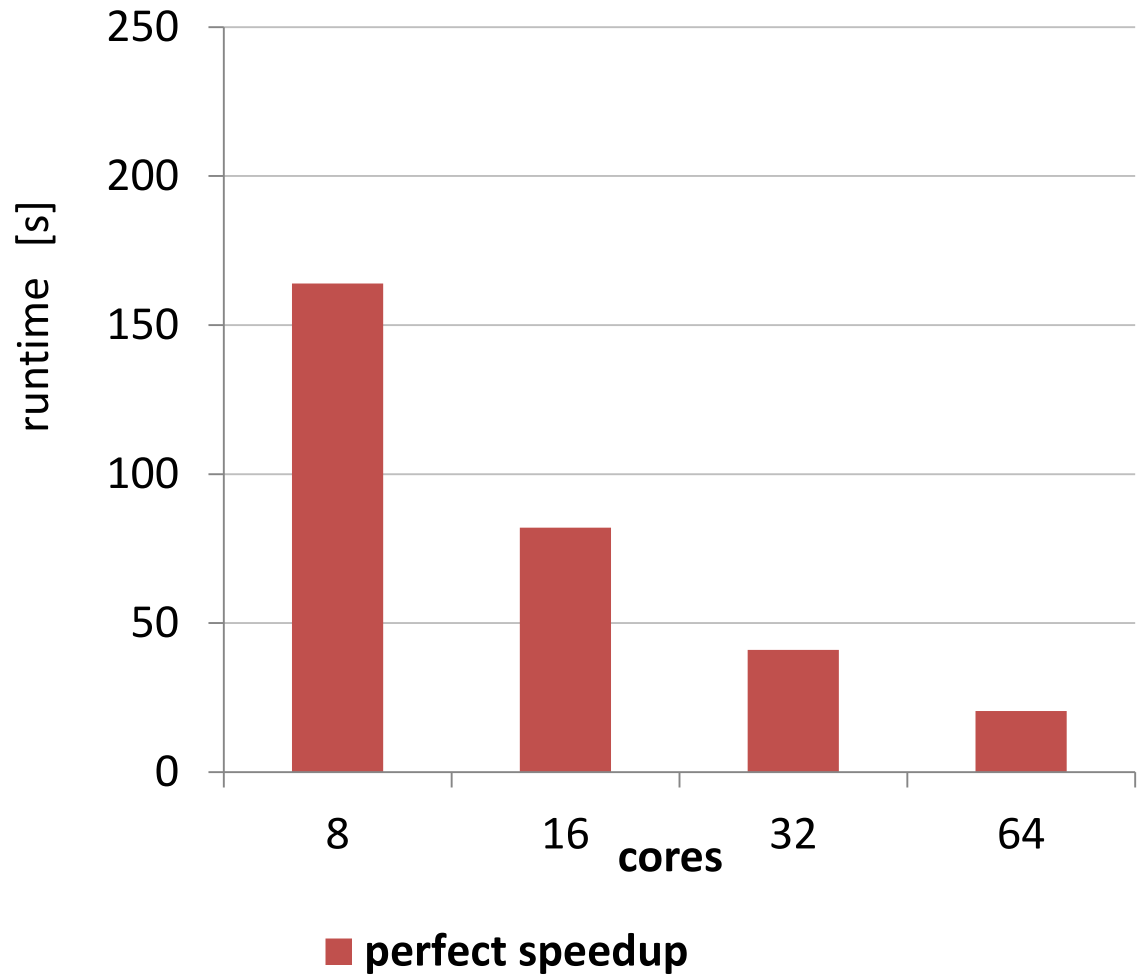
Suboptimal allocation → bad performance

Shoal

- Memory abstraction: Arrays
- Statically derive access patterns from code
- Choose array implementation at runtime

- Reduces runtime:
  - 4x over naïve memory allocation

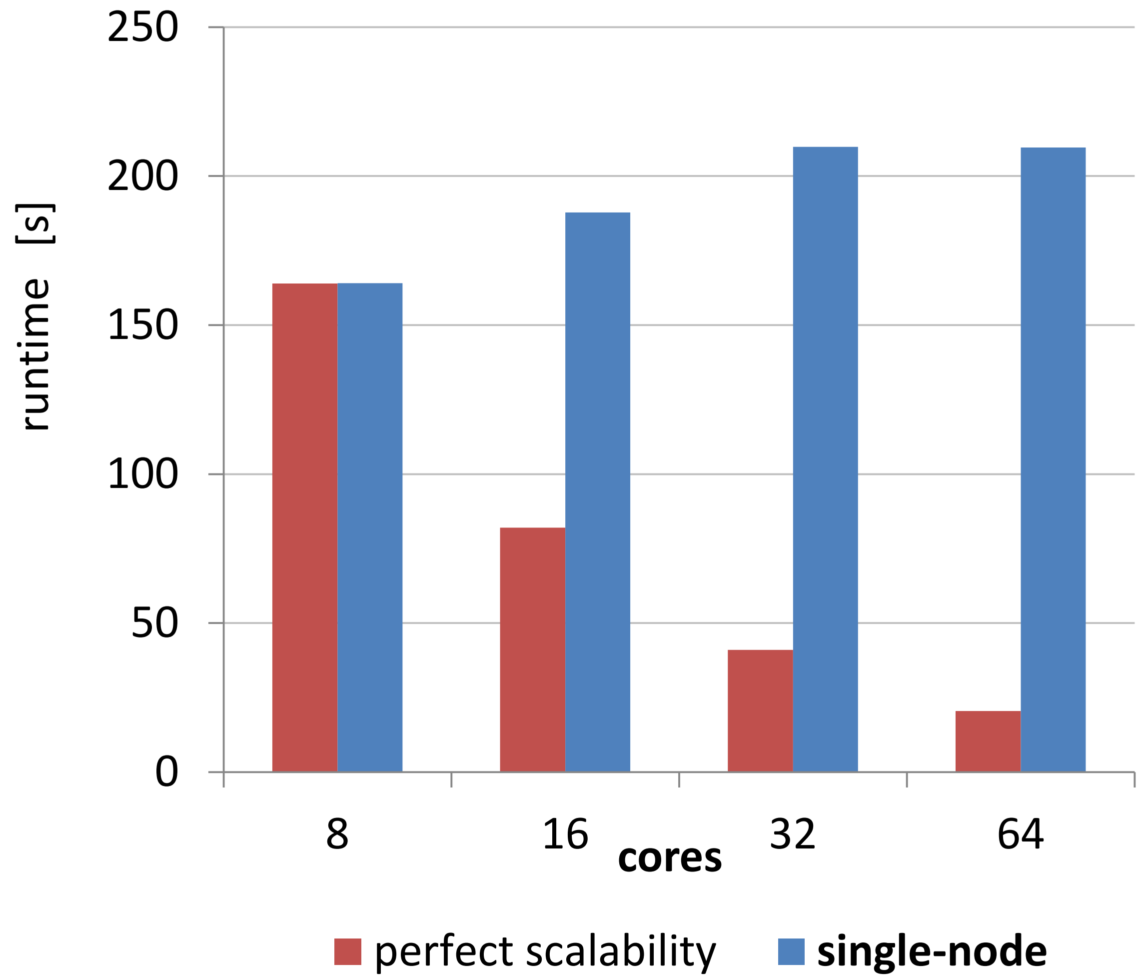
Example: PageRank



8x8 AMD Opteron 6378
Bulldozer
4 Sockets
512 GB RAM

SNAP Twitter graph
41M nodes
1468M edges
size in RAM: 2.5 GB

Example: PageRank



8x8 AMD Opteron 6378 Bulldozer
4 Sockets
512 GB RAM

SNAP Twitter graph
41M nodes
1468M edges
size in RAM: 2.5 GB

Problem: implicit allocation

```
void *data = malloc(SIZE);
memset(data, 0, SIZE);
```

- **Implicit Linux policy** on where to allocate memory
- First touch → all memory on same NUMA node

What we would like to do?

- Partitioning
  - Split working space, put on different nodes
- Replication
  - Copy array
  - Updates: consistency

→ Reduce load-imbalance
→ Localizes access → reduces interconnect traffic

- DMA
- 2M/1G pages

What we have today:

- Explicit placement of memory
  - libnuma
- Advise Kernel about use of memory region
  - madvise

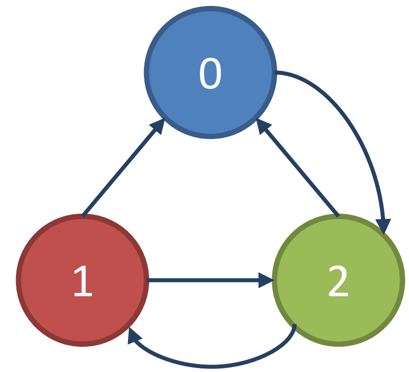
SHOAL

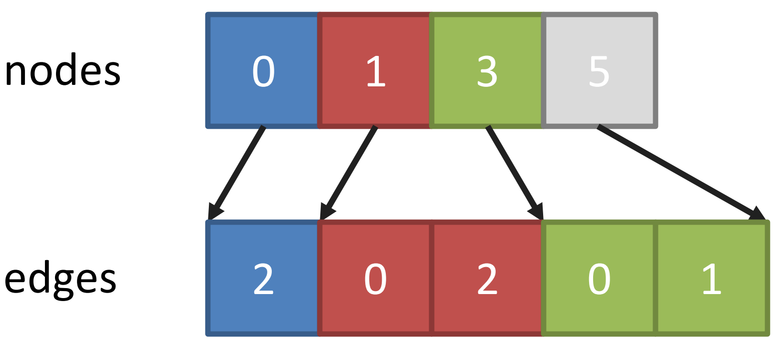
Exploiting DSLs

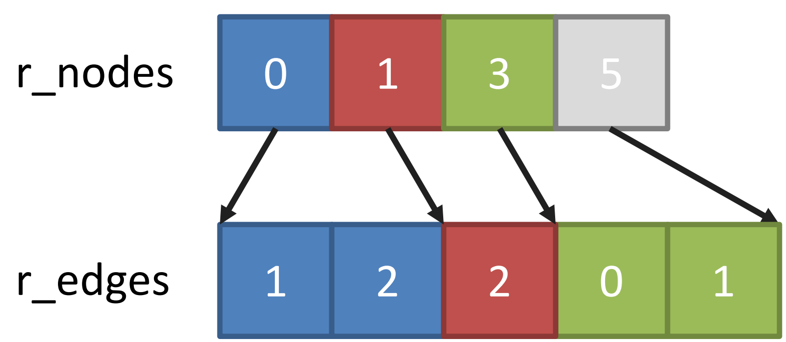
- High-level API
- Efficient parallelization
- widely used
  - Machine learning
  - Signal processing
  - Graph processing

Idea: derive access patterns

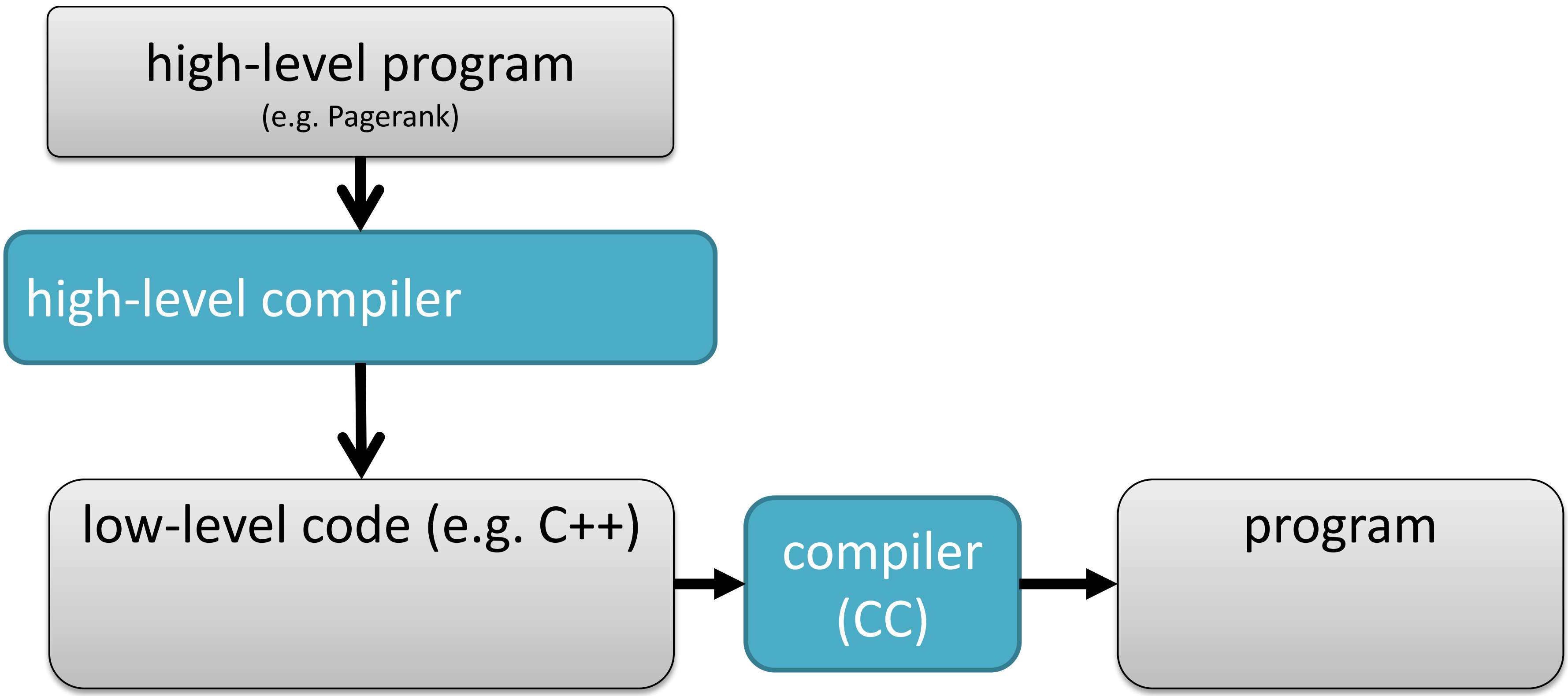
Green Marl: graph storage



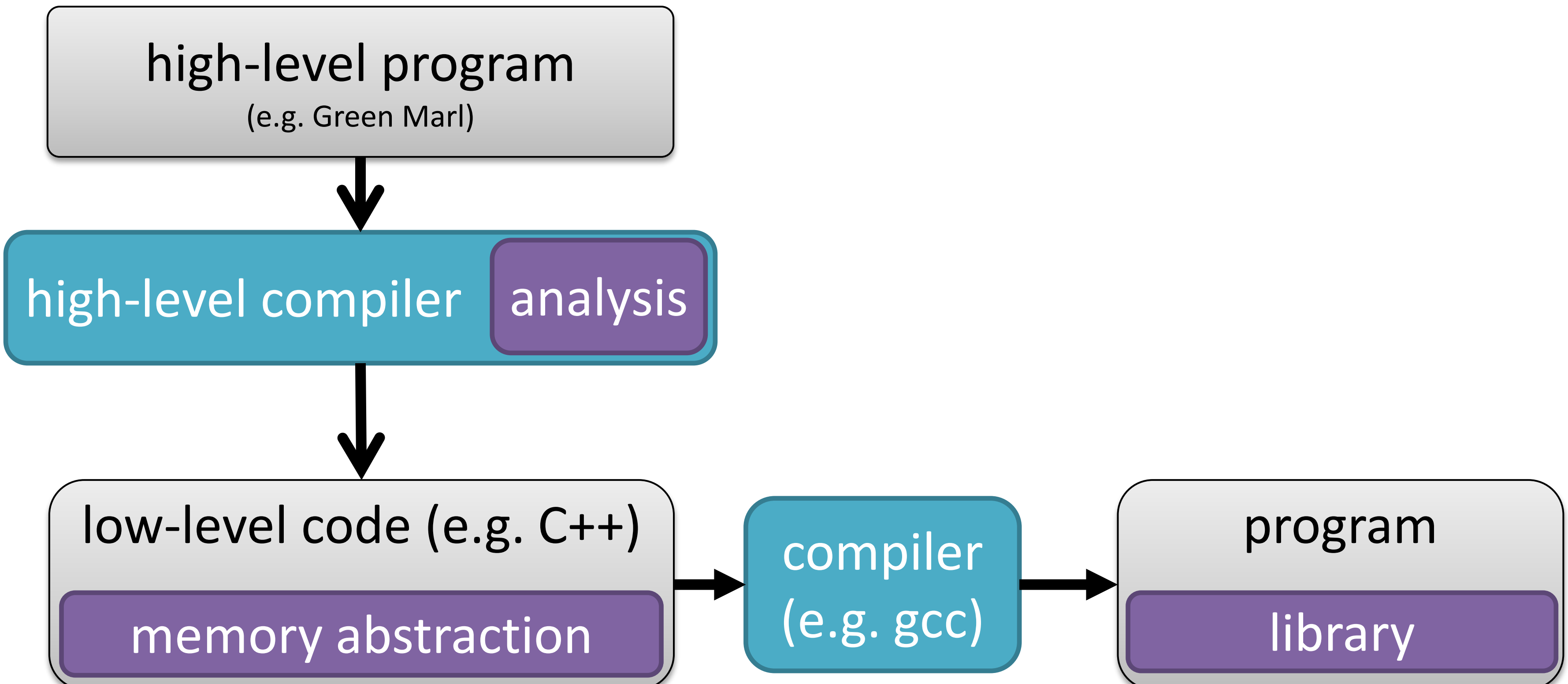




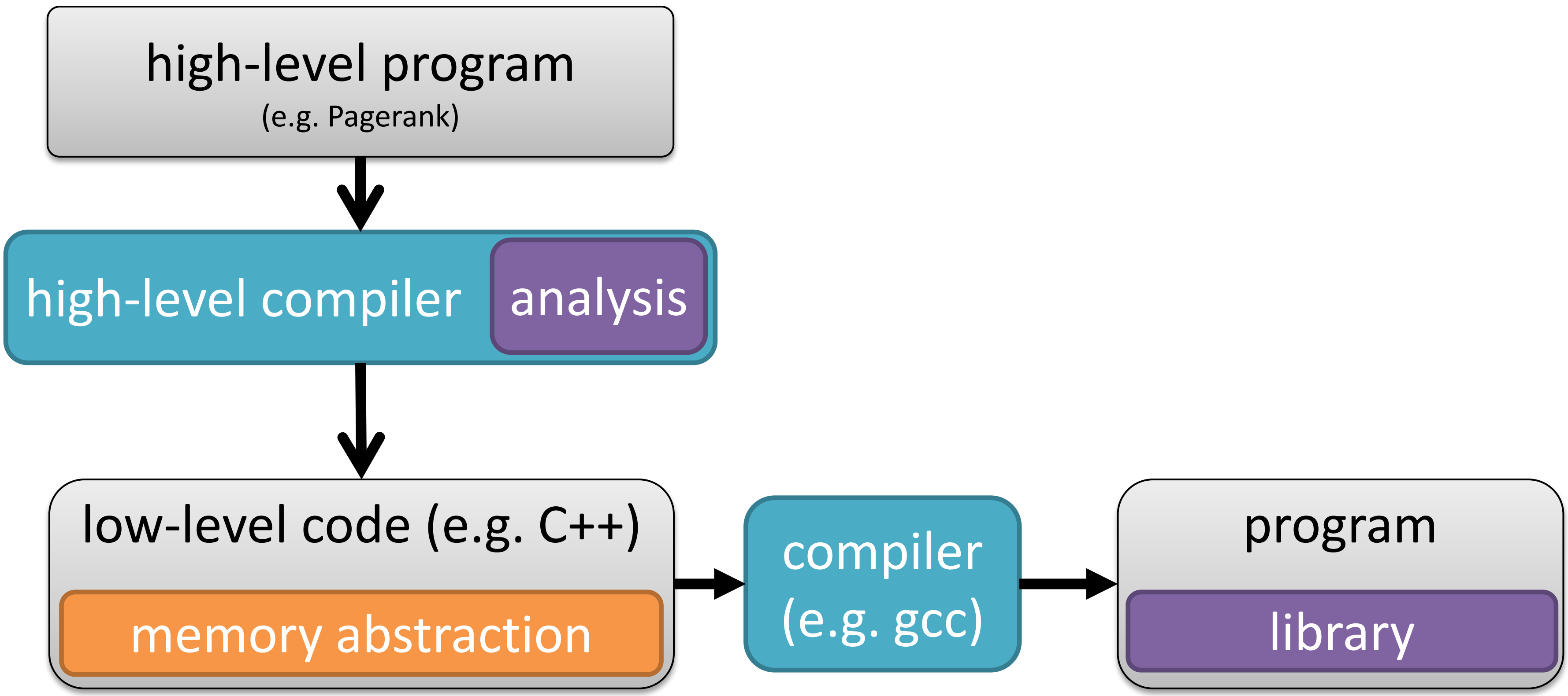
Overview: Green Marl



Modifications to Green Marl



1) Array abstraction



Array abstraction

- `get()` and `set()`
- `copy_from(arr)` and `init_with(const)`
- `array_malloc(size, access_patterns)`

Shoal’s access patterns

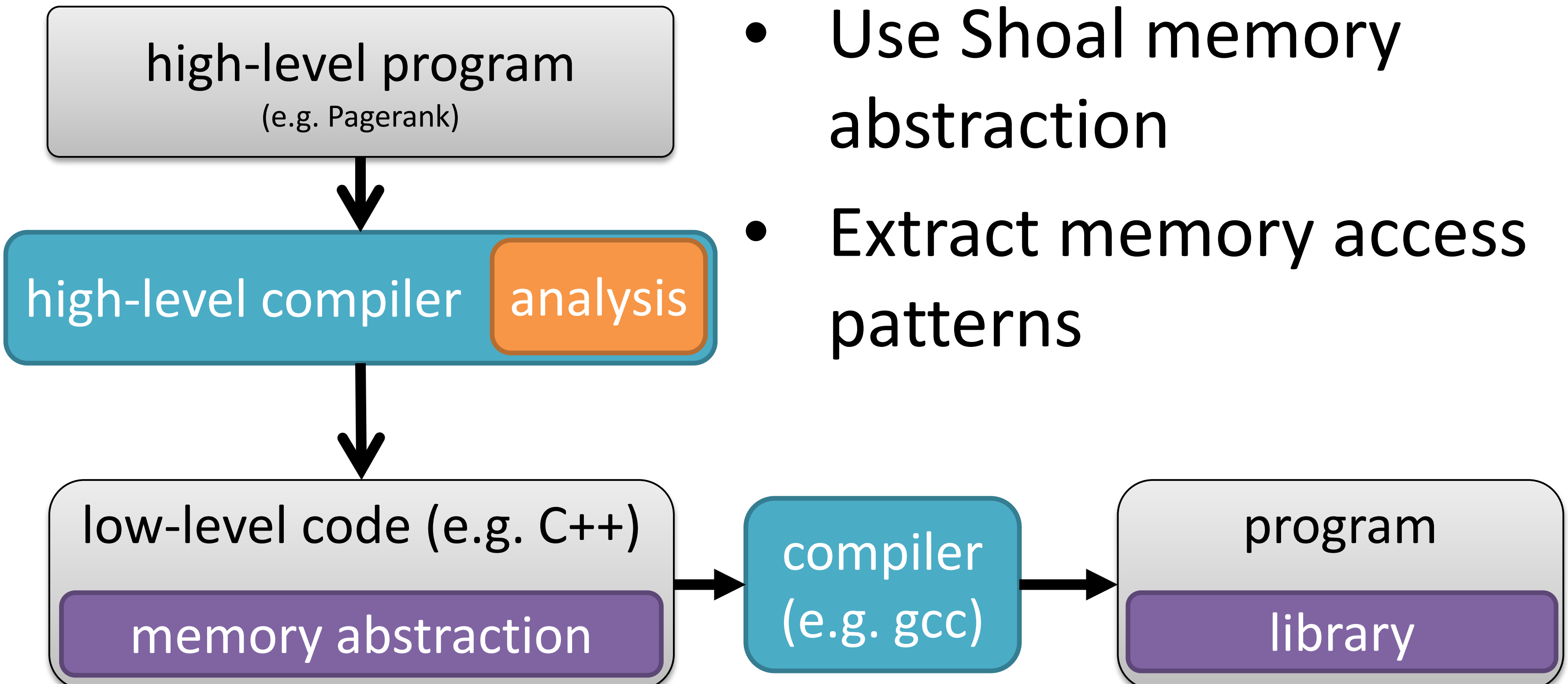
- Read-only
- Sequential
- Random
- Indexed

indexed:

```
for (i=0; i<SIZE; i++) {
  foo(arr[i]);
}
```

→ sequential + local

2) Compiler



- Use Shoal memory abstraction
- Extract memory access patterns

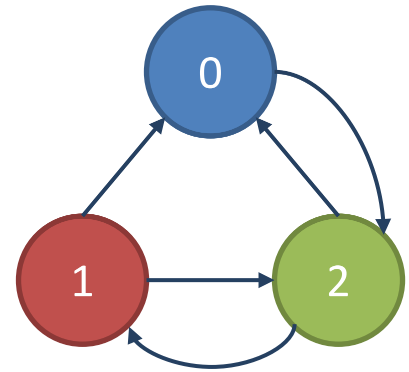
Derivation of access patterns

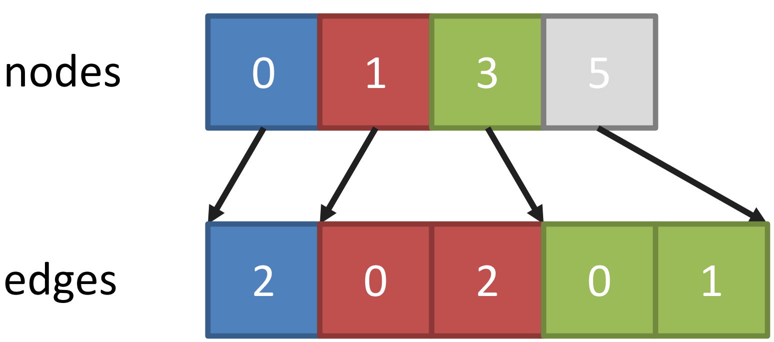
```
Foreach (t: G.Nodes) {
  Double val = k * Sum(nb: t.InNbrs){
    nb.rank / nb.OutDegree()} ;
  diff += | val - t.pg_rank |;
  t.pg_rank <= val @ t;
}
```

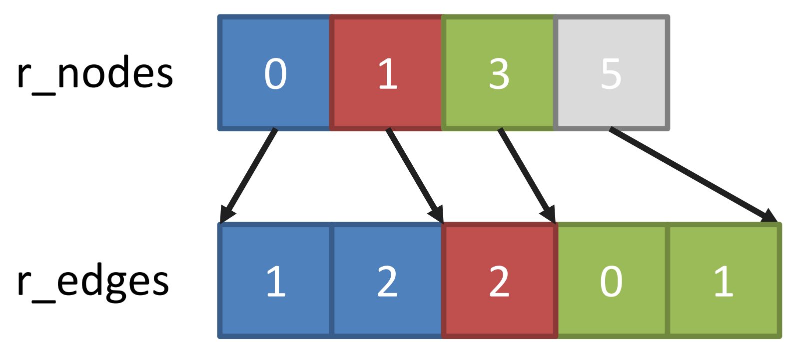
Derivation of access patterns

```
Foreach (t: G.Nodes) {
  Double val = k * Sum(nb: t.InNbrs){
    nb.rank / nb.OutDegree()} ;
  diff += | val - t.pg_rank |;
  t.pg_rank <= val @ t;
}
```

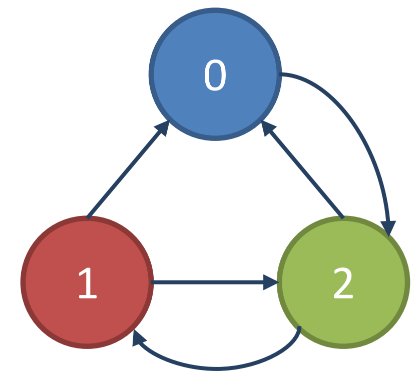
Green Marl: graph storage

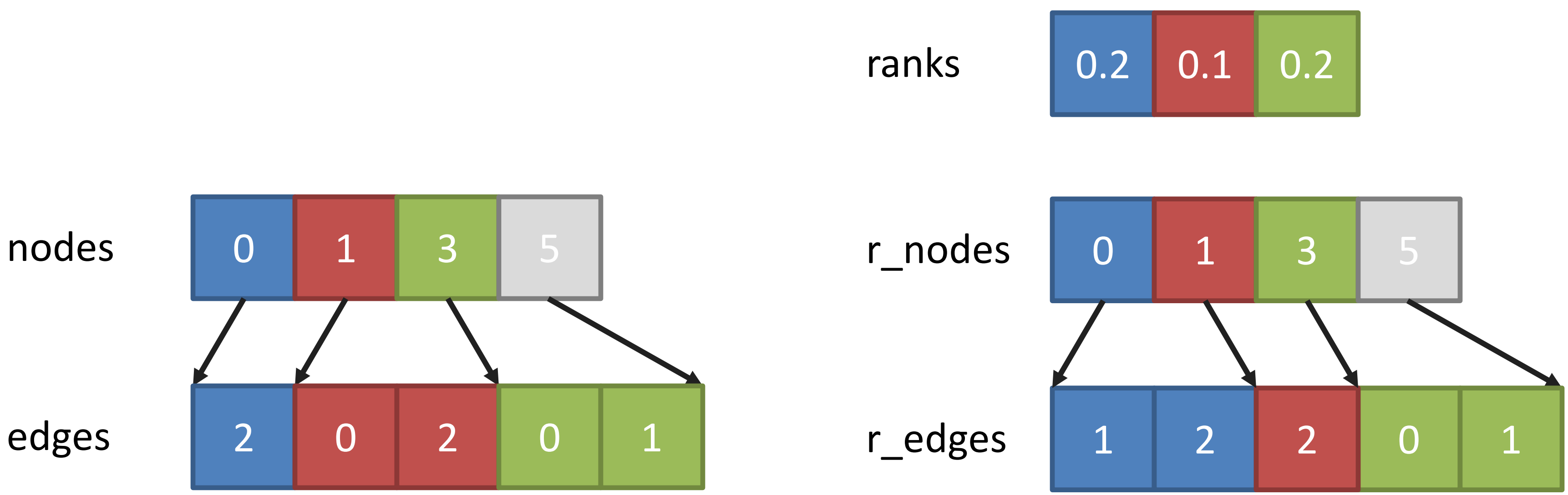






Green Marl: graph storage



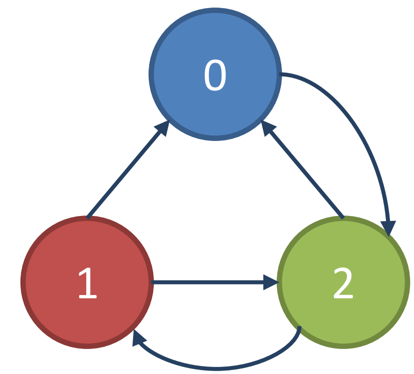


Deriving access patterns

```
Sum(nb: t.InNbrs) {
    // …
};
```

Operation: InNbrs - neighbors of node *t*:

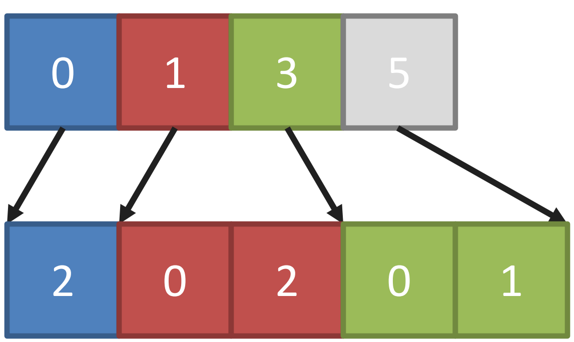
```
s  = r_nodes[t]
e  = r_nodes[t+1]-1
nb = [r_edges[x] for x in (s..e-1)]
```



ranks

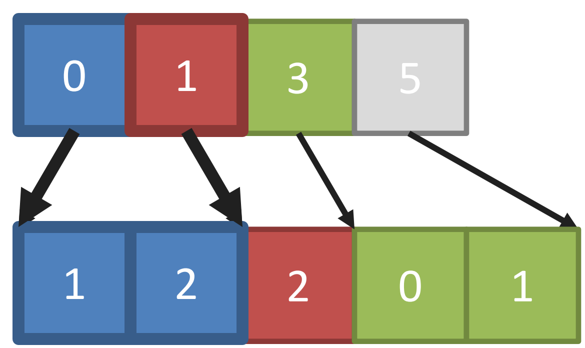
nodes

edges



r_nodes

r_edges



Deriving access patterns

```
Sum(nb: t.InNbrs) {
    // …
};
```

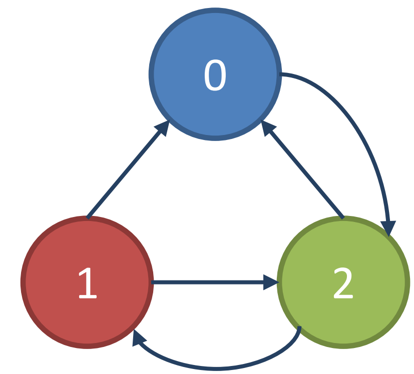
Operation: InNbrs - neighbor

*s = **r_nodes**[t]*
*e = **r_nodes**[t+1]-1*
*nb = [**r_edges**[x] for x in (s..e-1)]*

- indexed
- read-only

- sequential
- read-only

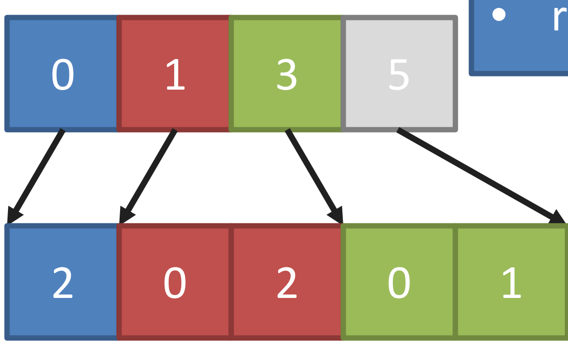
- sequential
- read-only



ranks

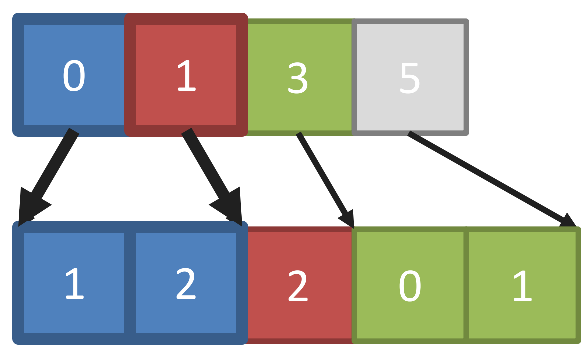
nodes

edges



r_nodes

r_edges



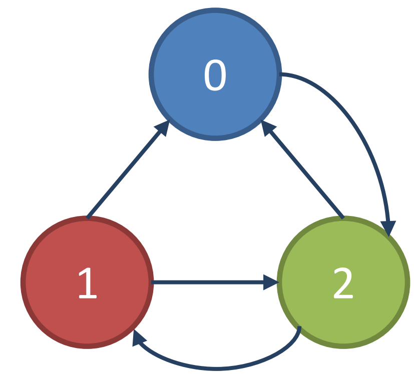
Deriving access patterns

```
Sum(nb: t.InNbrs) {
    nb.rank / nb.OutDegree()
};
```

Operation: rank - rank of neigbor *nb*:

rnk_tmp = rank[nb]

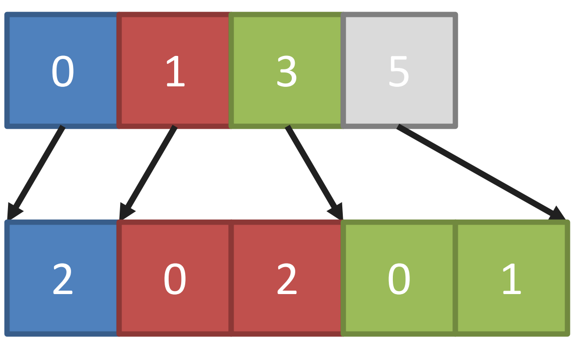
- random
- read-only



ranks

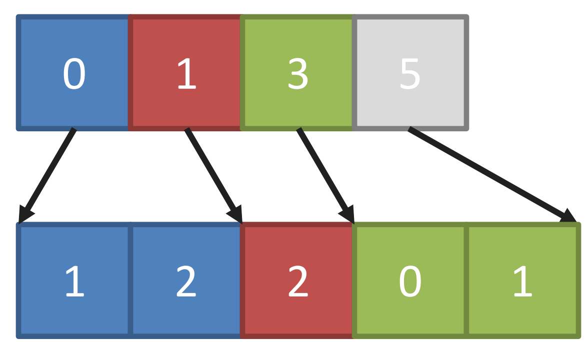
nodes

edges



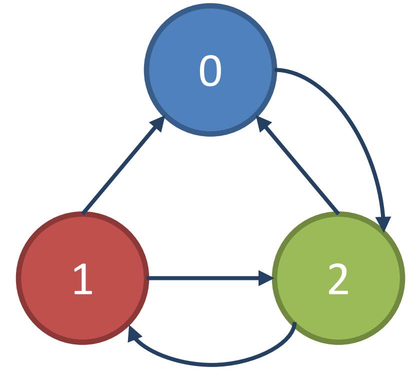
r_nodes

r_edges



Deriving access patterns

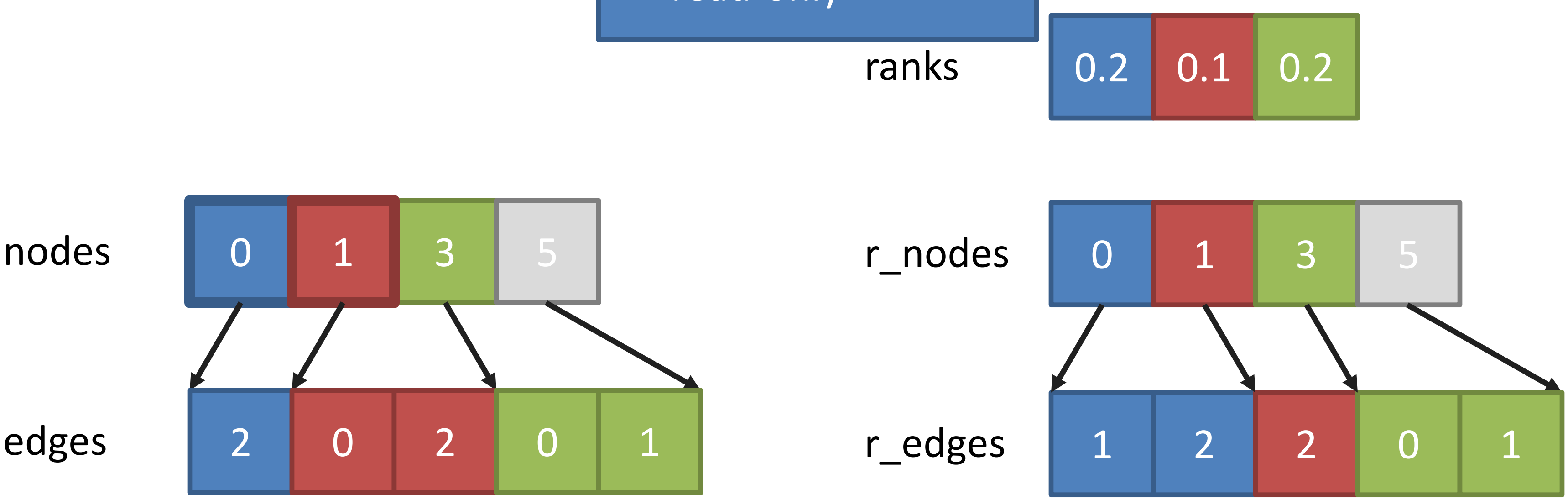
```
Sum(nb: t.InNbrs) {
    nb.rank / nb.OutDegree()
};
```



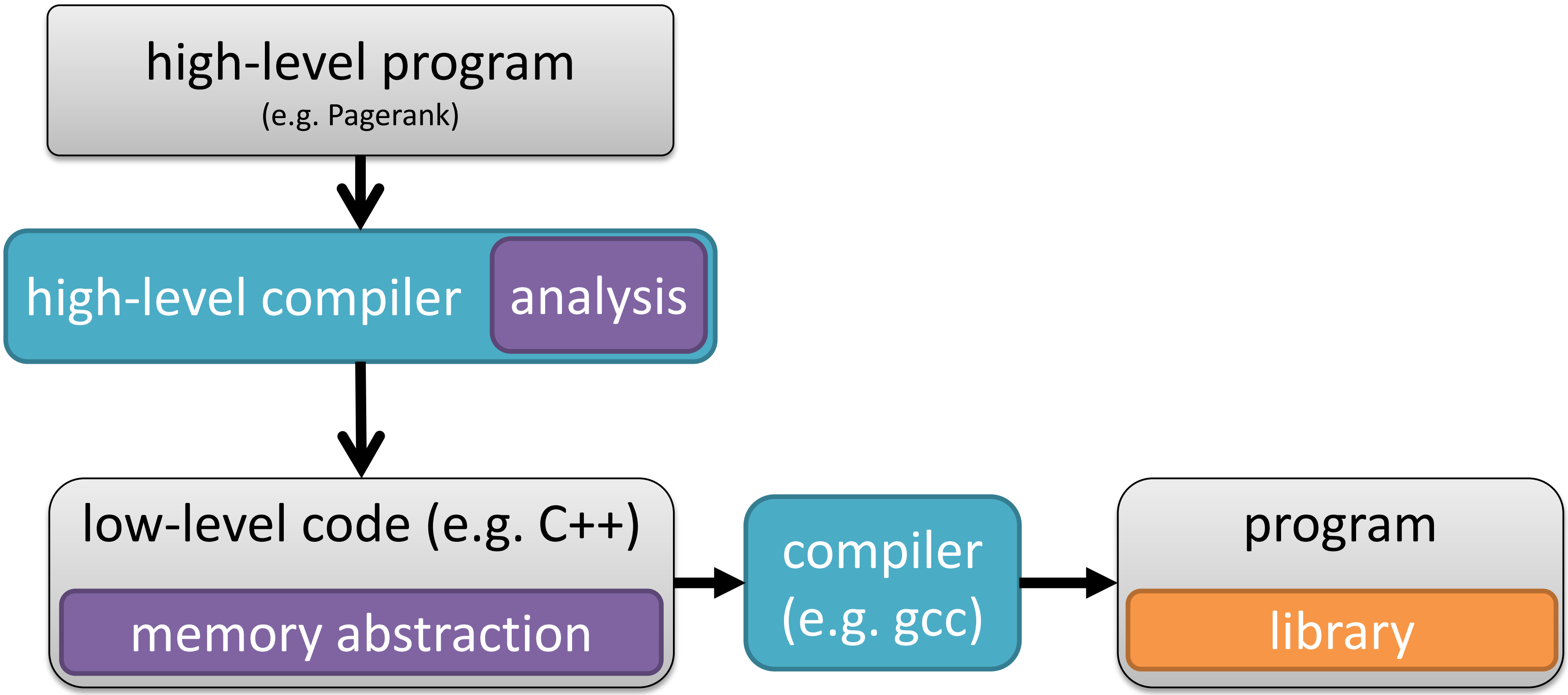
Operation: OutDegree() - of neighbor *w*:

nodes[nb+1] – nodes[nb]

- random
- read-only

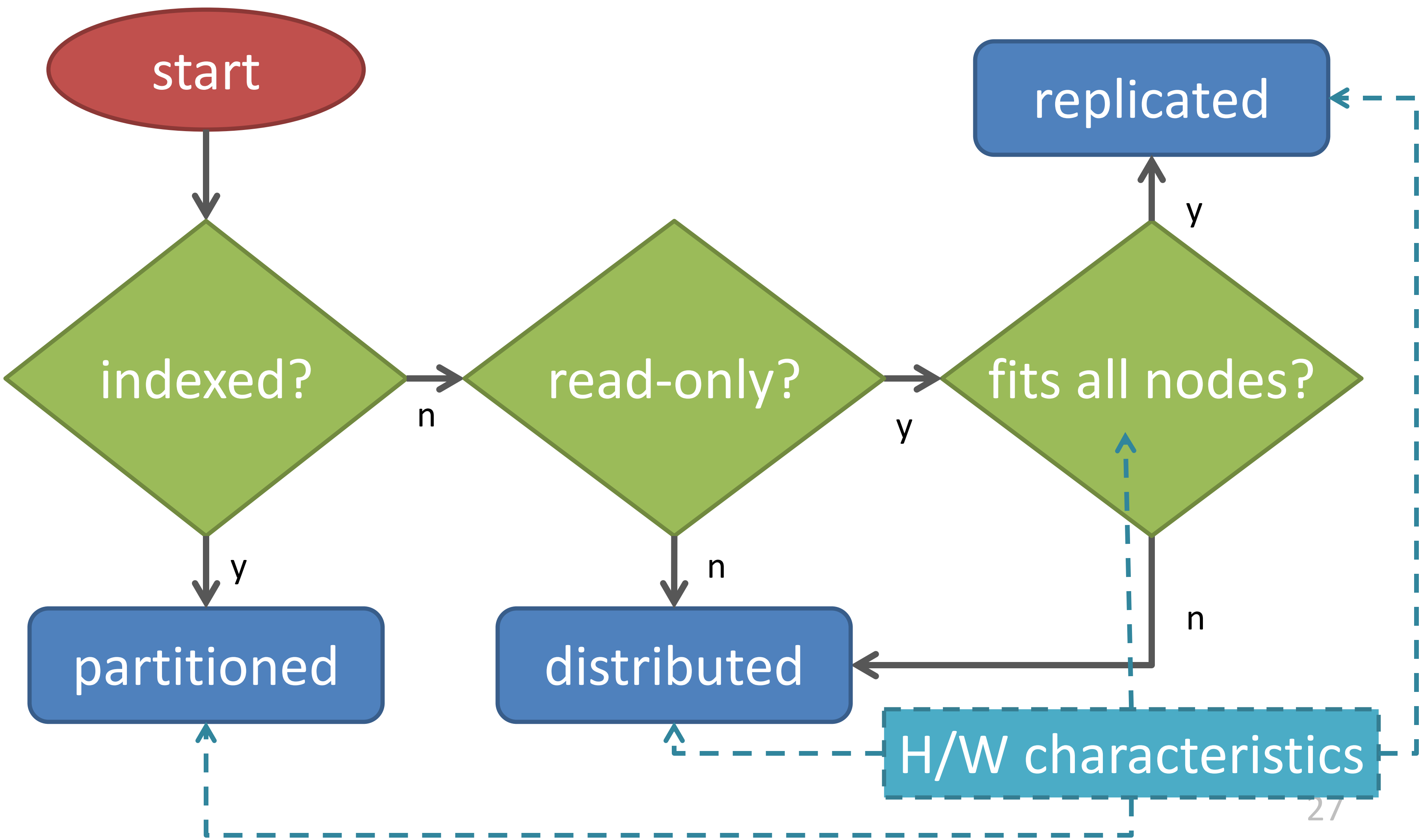


3) Runtime

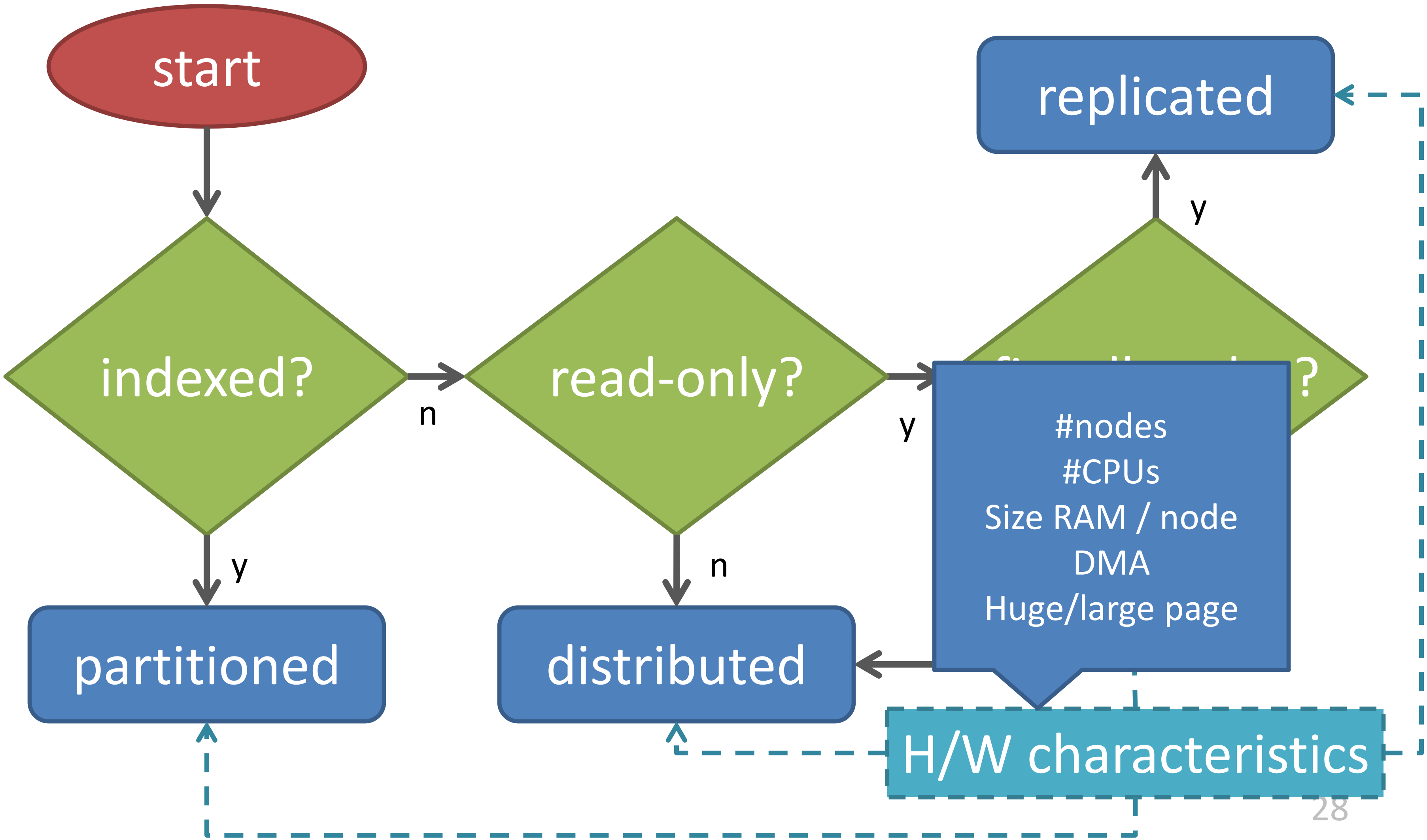


high-level program
(e.g. Pagerank)
high-level compiler
analysis
low-level code (e.g. C++)
memory abstraction
compiler
(e.g. gcc)
program
library

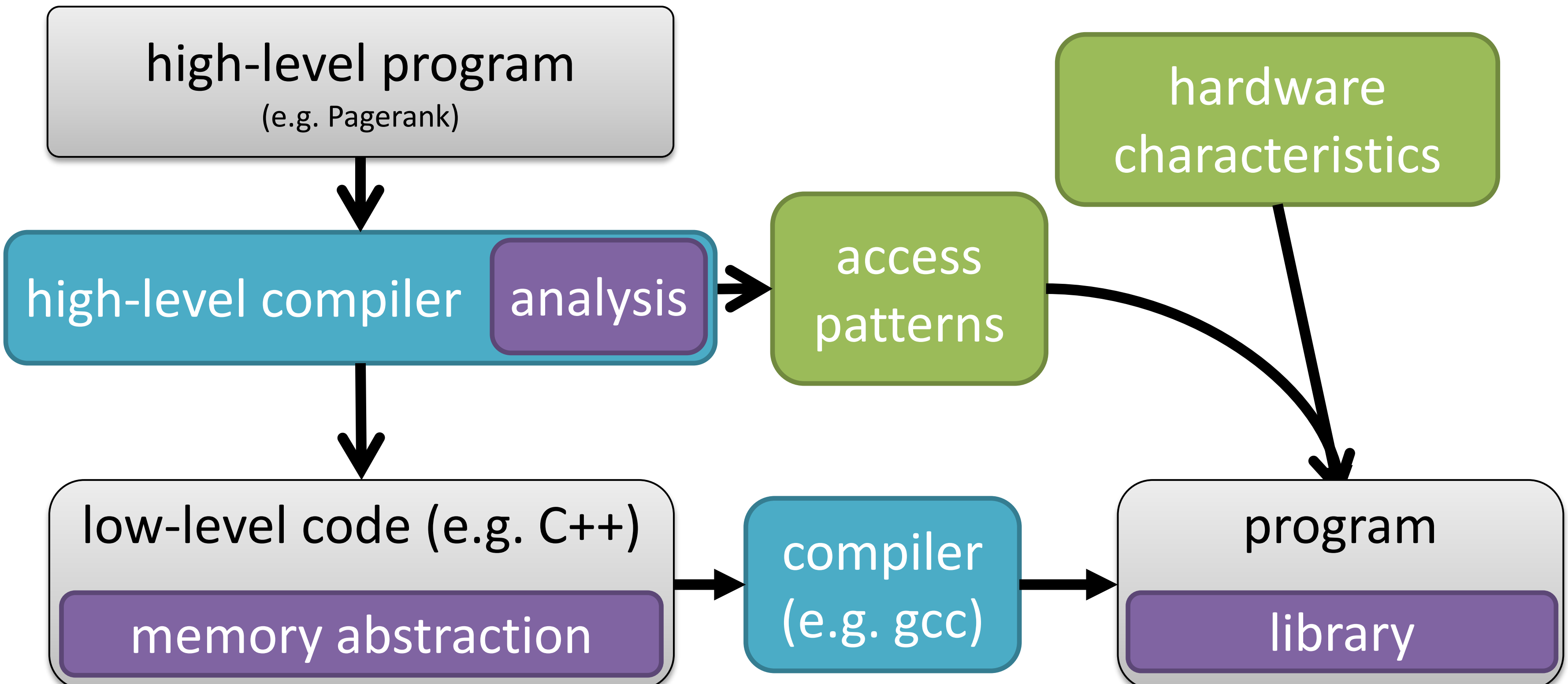
Runtime: Choice of arrays



Runtime: Choice of arrays



Shoal workflow

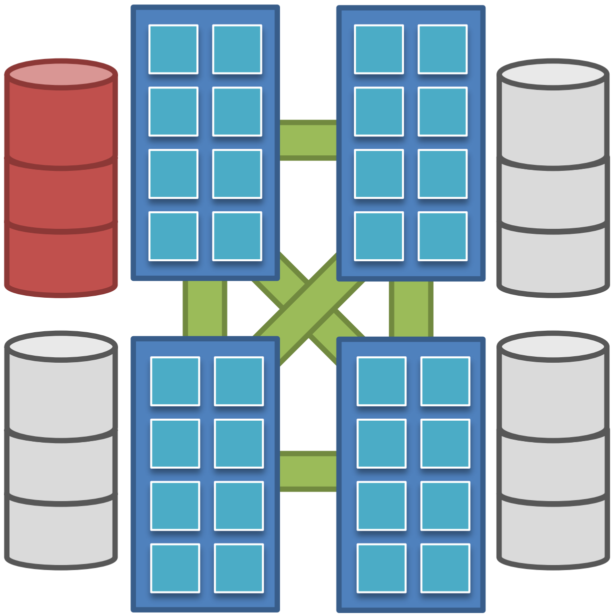


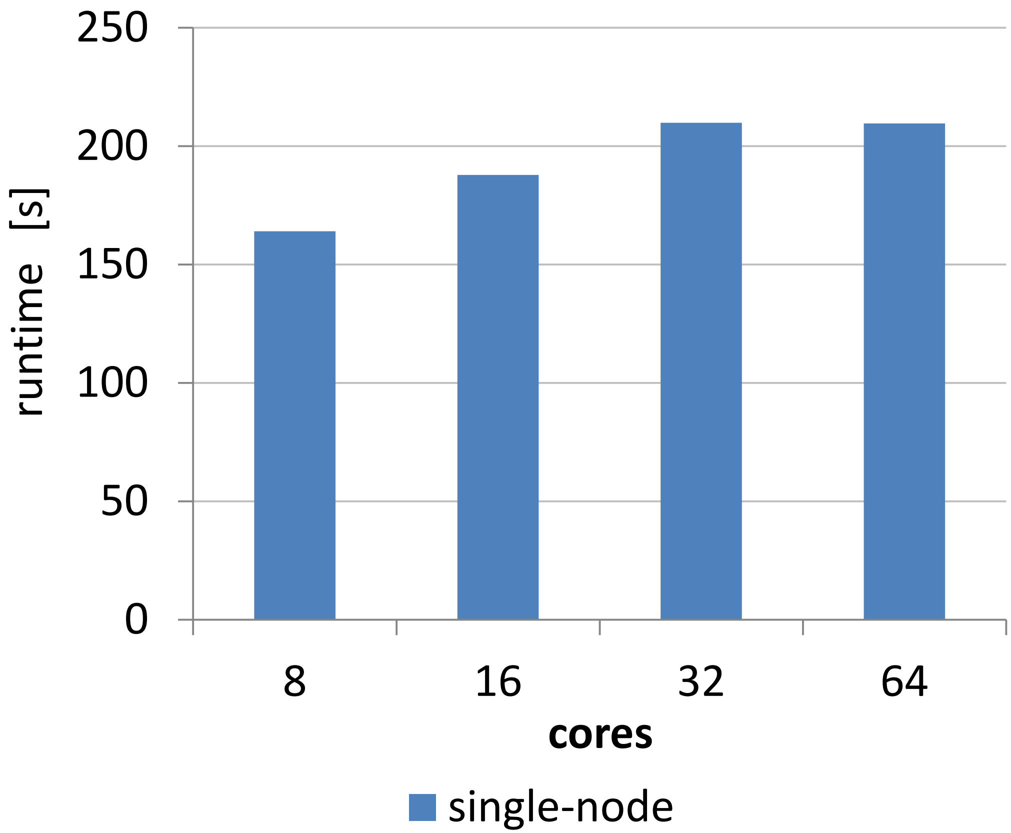
Alternative approaches

- Search-based auto-tuning
- HW page migration
- Carrefour: Online analysis of access patterns
  - Simon Fraser University, ASPLOS 2013
  - Performance counters to monitor accesses
  - Dynamically migrate and replicate pages

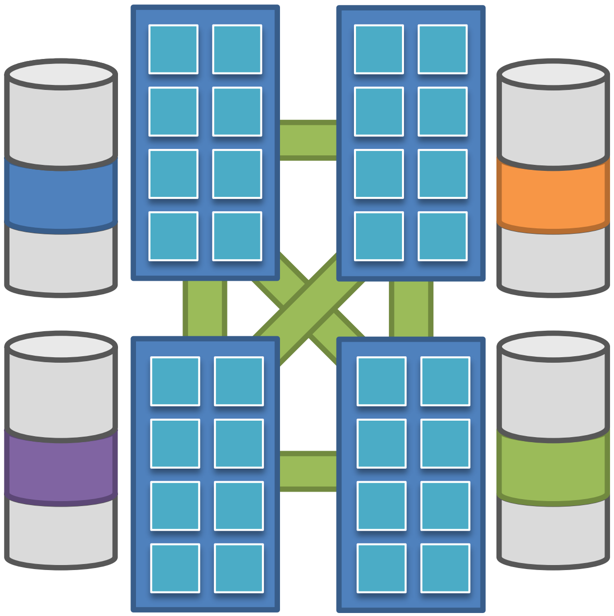
EVALUATION

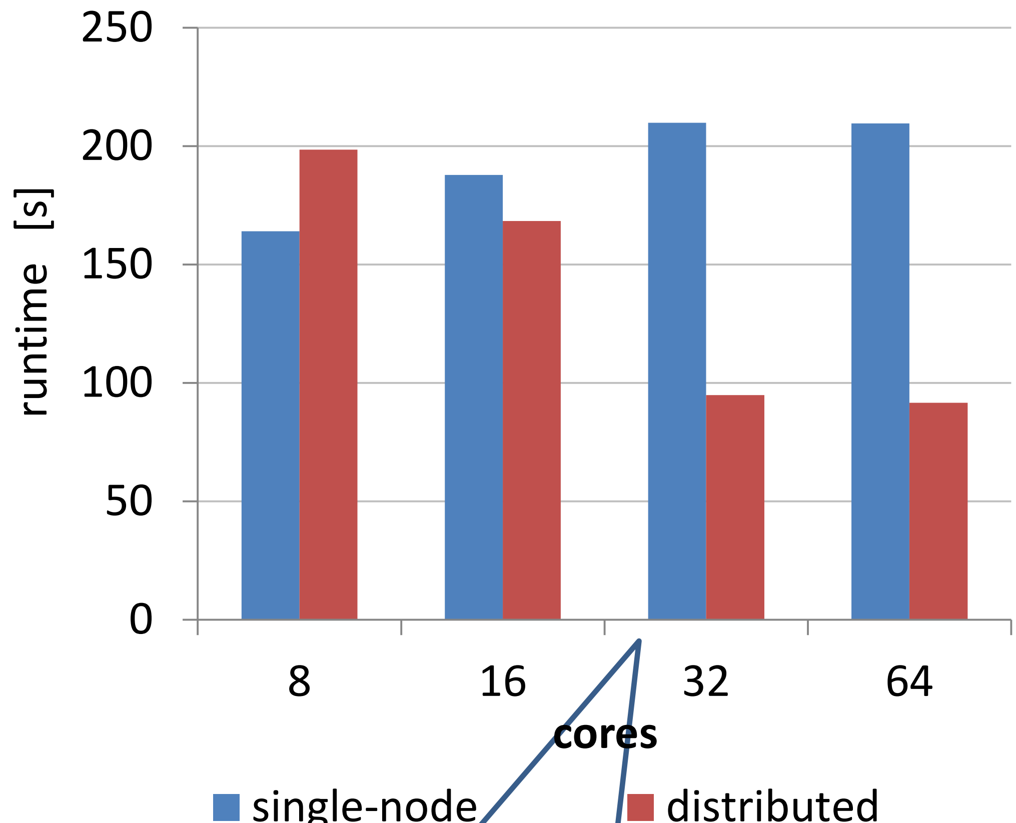
Single node allocation





Distribute memory



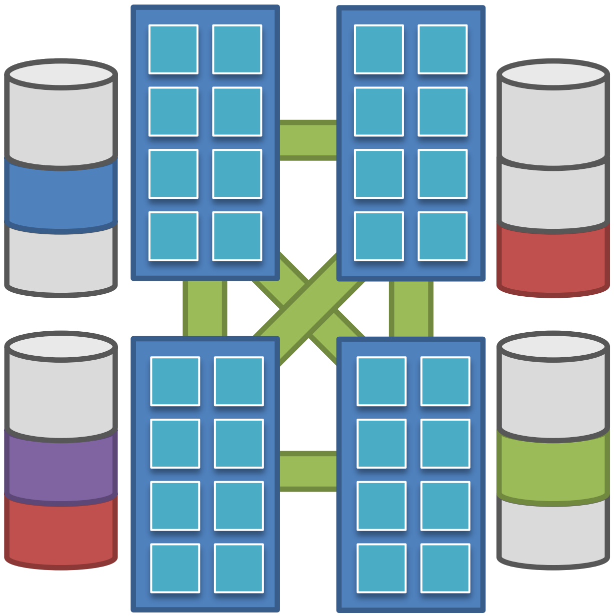


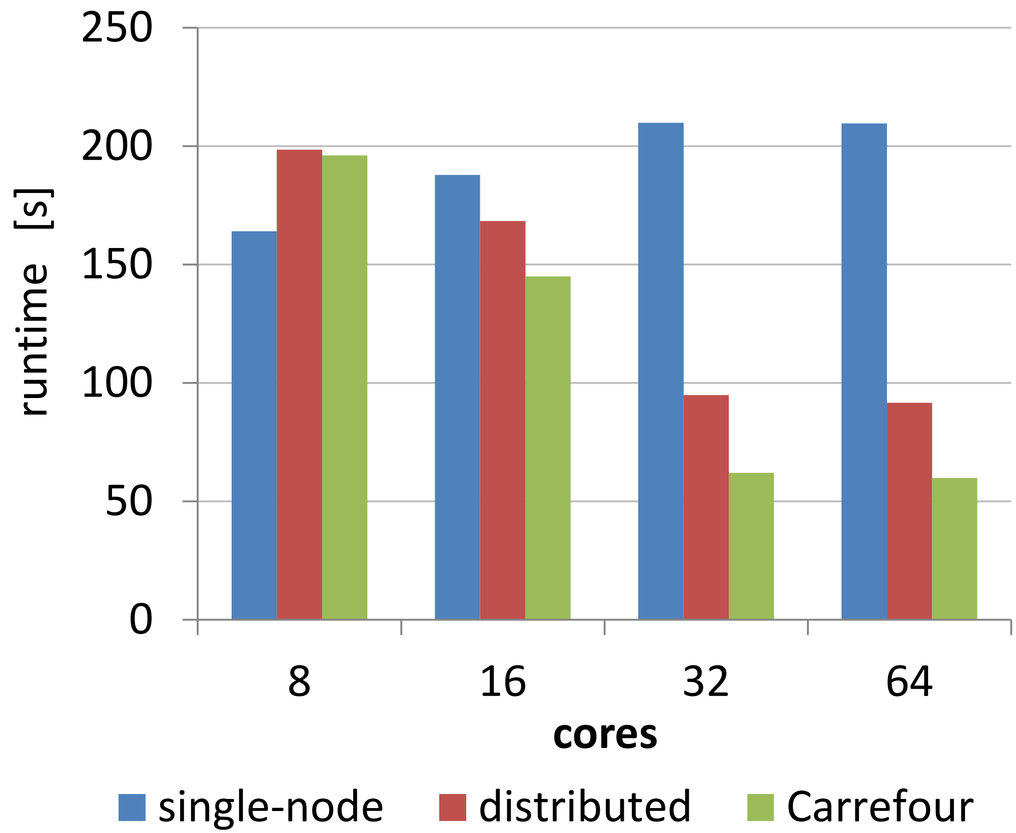

```
#pragma parallel omp for
for (int i=0; i<SIZE; i++)
        data[i] = 0;
```

Problem: this is OS / HW specific

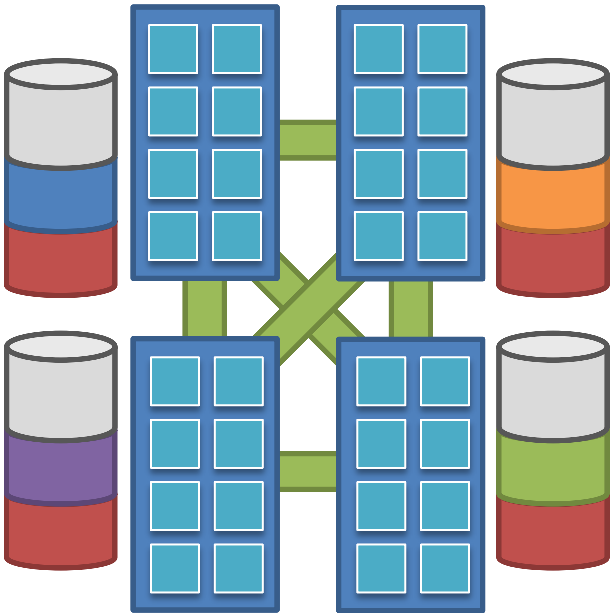
Default Green Marl behavior

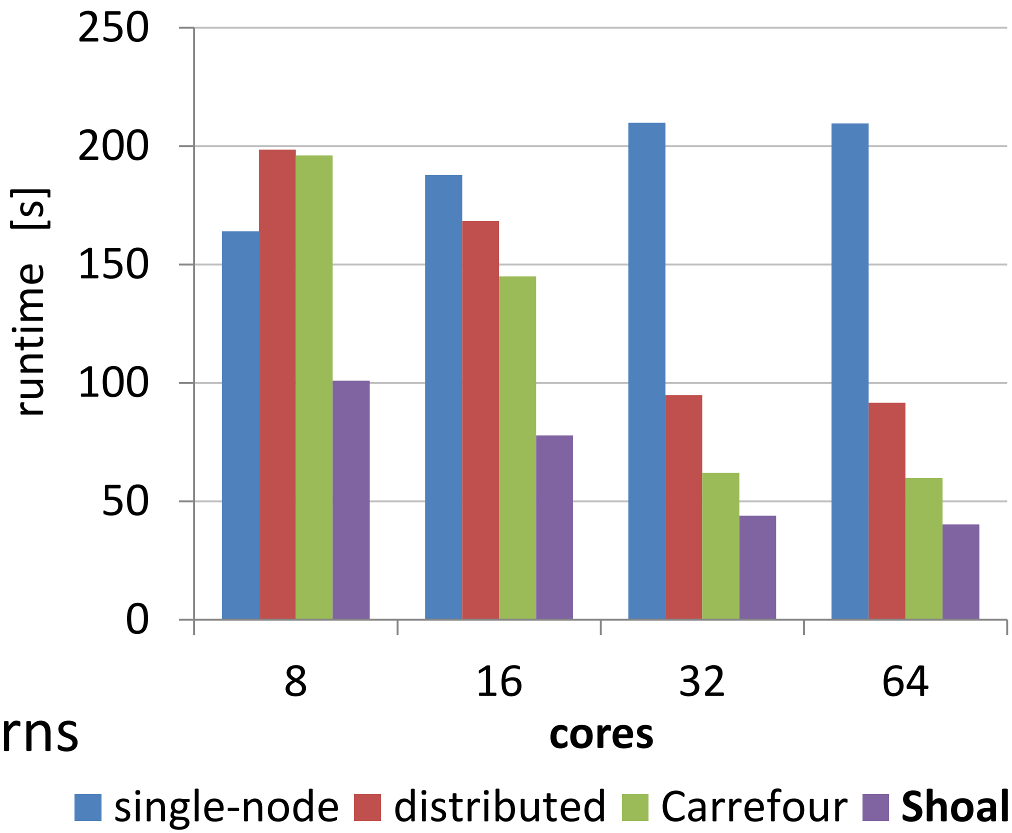
Carrefour: reactive tuning





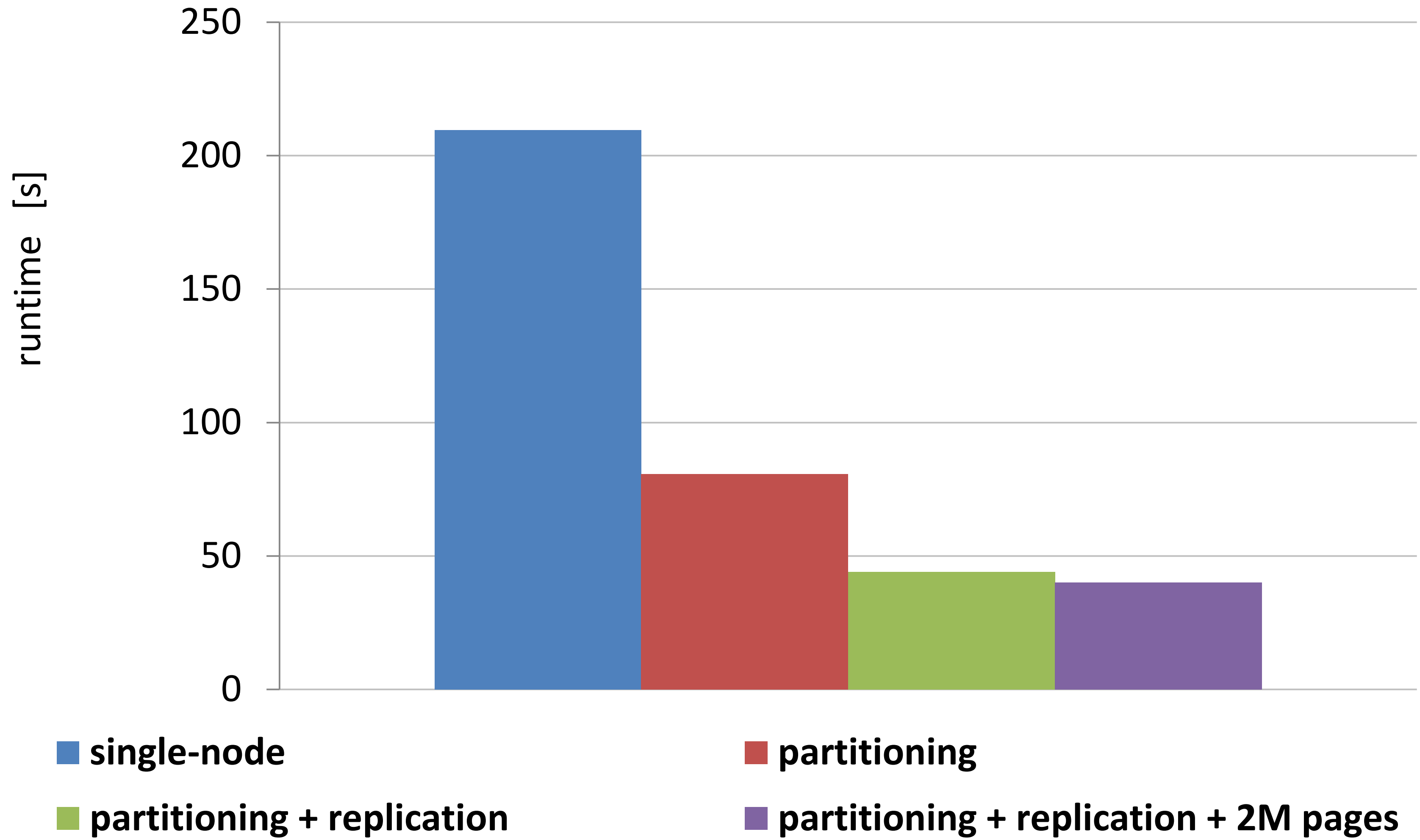
Shoal





- Knowledge of access patterns
- Replication
- Distribution
- Large pages (2M)

Performance breakdown



Conclusion

Systems@ETH zürich

- Memory abstraction, arrays
- Compiler analysis → derive access patterns
- Runtime library → selects implementation
- Works well with domain specific languages
- Also: support for manual annotation
  - Too complex, too dynamic → Online

- Public release next week