Parallel Skeletons for Variable-Length Lists in SkeTo Skeleton Library

Aug. 27, 2009

Haruto Tanno
Hideya Iwasaki

The University of Electro-Communications (Japan)

Outline

- Introduction
- Problems of Exiting Fixed-Length Lists
- Proposed Variable-Length Lists
  - Skeletons and Operations
  - Data Structure
- Experiments
- Related Work
- Conclusion

Introduction

- Writing efficient parallel programs is difficult
  - synchronization, inter-process communications, data distributions among processes
- Solution: Skeletal parallel programming
  - implements generic patterns within parallel programs
    - C++ parallel skeleton library SkeTo [06] (intended for distributed environments such as PC cluster)
  - enables users to write parallel programs as if they were sequential

SkeTo

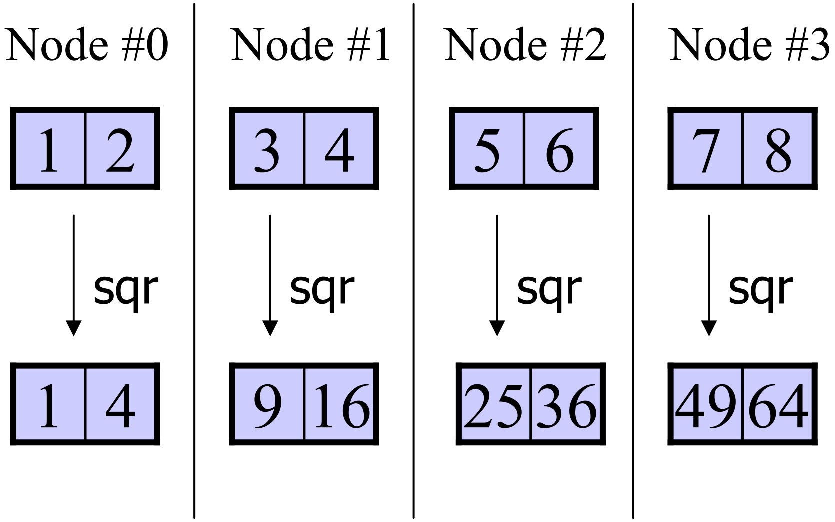
- SkeTo (Skeletons in Tokyo)
  - Constructive parallel skeleton library (C++ and MPI)
  - Joint project of The University of Tokyo, National Institute of Informatics, and The University of Electro-communications
  - Started from 2003, Version 1.0 coming soon
- Distinguishing features of SkeTo
  - It is based on the theory of Constructive Algorithmics
  - It provides skeletons for lists, matrices, and trees
  - It introduces no special extension to the base C++

Examples of List Parallel Skeletons

- map skeleton on list
  - applies a function to all elements

map sqr [1,2,3,4,5,6,7,8]

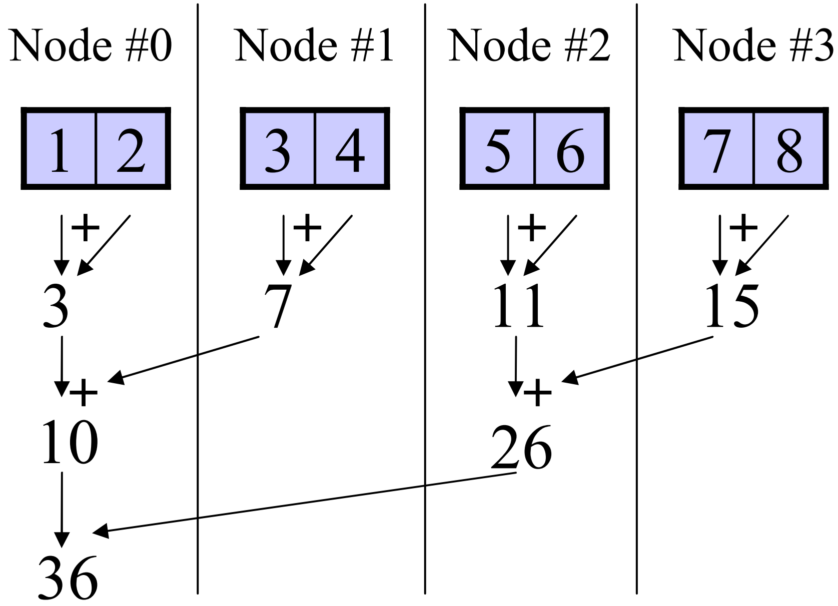
= [1,4,9,16,25,36,49,64]



- reduce skeleton on list
  - collapses a list with an associative binary operator

reduce (+) [1,2,3,4,5,6,7,8]

= 1+2+3+4+5+6+7+8 = 36



Problem in Existing List

- Computing twin primes

  (Pairs of prime numbers that differ by two)

```
nums = [2,3,...,13,14,15], ps = [ ];
do { // the Sieve of Eratosthenes
  p = take the front element from nums
  remove every element that is divisible by p from nums
  add p to ps
} while( p <= sqrt_size );
concatenate ps and nums
//ps is [2,3,5,7,11,13]
twin_ps = make pairs of adjacent prime numbers
remove every pair of prime numbers whose difference isn't two
//twin_ps = [(3,5), (5,7), (11,13)]
```

List shrinks

List stretches

We can't shrink or stretch lists because their size is fixed

Problems that Need Variable-Length Lists

Type1: Problems that leave such elements in a given list that satisfy various conditions

- e.g., twin-primes problem, problem of the convex hull

Type2: Searching problems in which the number of candidates for solutions may dynamically change

- e.g., knight's tour

Type3: Iterative calculations in which computational loads for all elements in a list lack uniformity

- e.g., calculations of Mandelbrot and Julia sets

If we can remove elements that have already finished their calculations, we can solve these problems efficiently

Our Proposal

Proposal

- We propose parallel skeletons for lists of variable lengths that enable us to solve a wide range of problems

Approach

- We implement a new library of variable-length lists in SkeTo (They are compatible with existing lists)
  - Provide new skeletons and operations that dynamically and destructively change a list's length
  - Change data structure that expresses lists
  - Add automatic data relocation mechanism for adequate load balancing

Proposed Skeletons and Operations

- concatmap applies a function to every element and concatenates the resulting lists

  e.g., concatmap dup [1,2] => [1,1,2,2]
- filter leaves elements that satisfy a Boolean function

  e.g., filter odd [1,2,3,4,5,6,7,8] => [1,3,5,7]
- append concatenates two lists

  e.g., append [1,2] [3,4,5,6] => [1,2,3,4,5,6]
- popfront and popback remove an element from the front and the back in a list
- pushfront and pushback add an element to the front or the back in a list

C++ Program for Twin Prime Problem

```
…
do{ //the Sieve of Eratosthenes
    p = nums->pop_front();
    list_skeletons::filter_ow(IsNotMultipleOf(p), nums);
    ps->push_back(p);
} while(p <= sqrt_size);
ps->append(nums);
//ps is list of prime numbers
dist_list<int>* dup_ps = ps->clone<int>();
dup_ps->pop_front();
twin_ps = list_skeletons::zip(ps, dup_ps);
list_skeletons::filter_ow(twin_ps, IsTwin());
//twin_ps is solution list
```

C++ Program for Knight's Tour

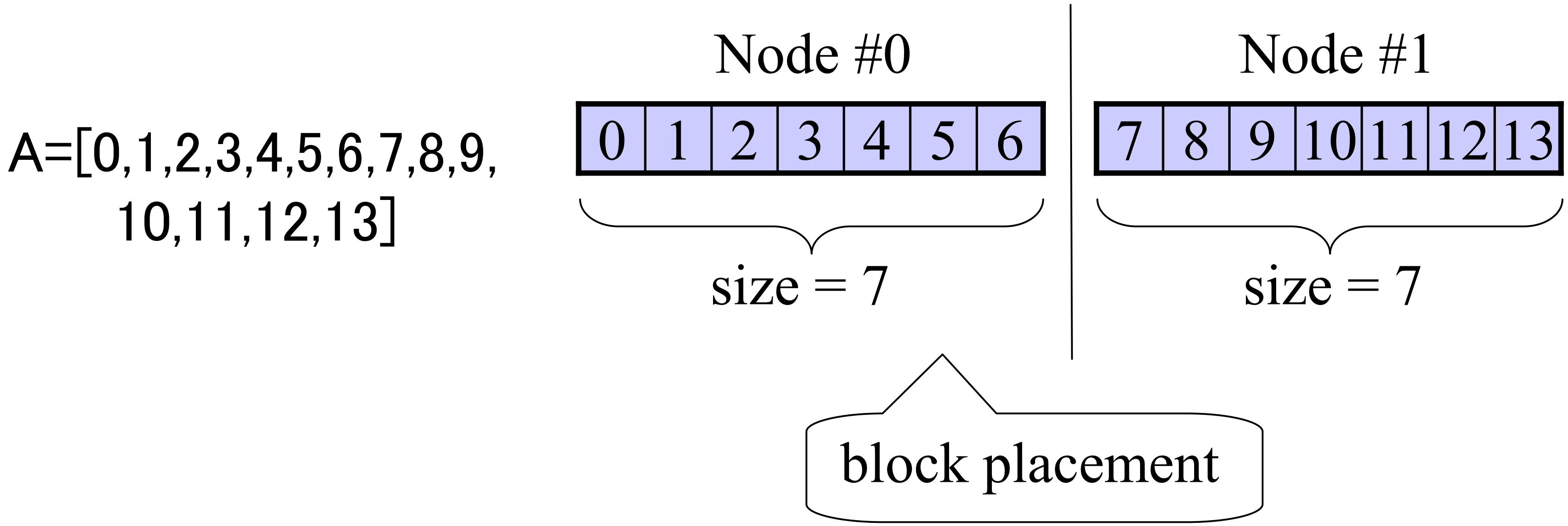
```
dist_list<Board>* bs; //list of solution boards
…
// add initial board state to bs
bs->push_back(initBoard);
// increase bs dynamically up to MAXSIZE
while( bs->get_size() < MAXSIZE ){
  // generate next moves from each board states
  concatmap_ow(nextBoard, bs);
}
// search all solutions with depth first order
concatmap_ow(solveBoard, bs);
// bs is a list of solution boards
```

C++ program for Mandelbrot set

```
dist_list<point>* ps;        // list of points
dist_list<point>* rs;        // list of calculation results
…
for ( int i=0; i<maxForCount; i++ ){
  // progress calculations in small amounts
  map_ow(calc, ps);
  // remove elements that have already finished calculation
  dist_list<point>* es = filtersplit_ow(isEnd, ps);
  // add them to rs
  rs->append(es);
  delete es;
}
rs->append(ps);
// rs is a list of calculation results
```

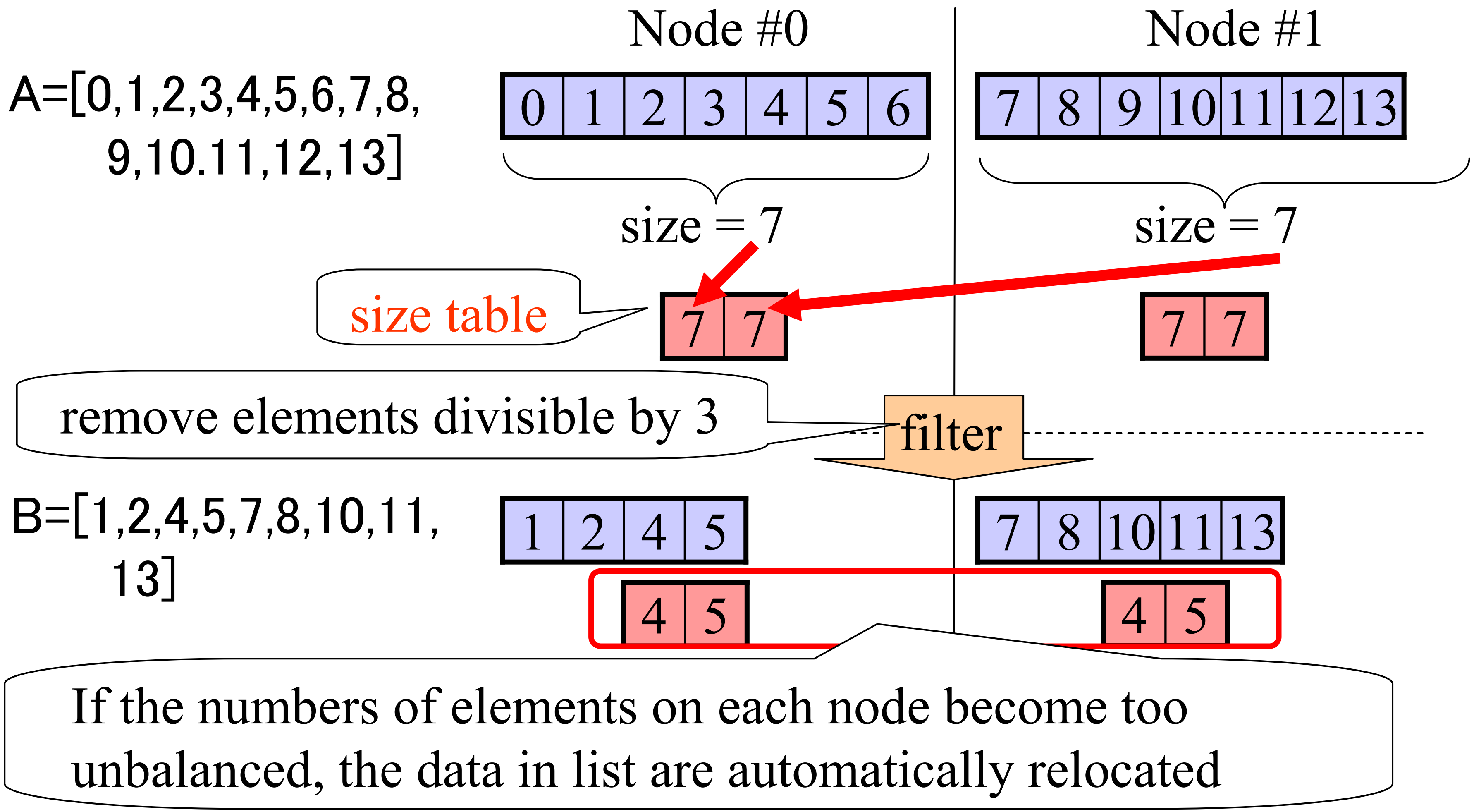
Data Structure of Existing Fixed-Length List

- Elements in lists are equally distributed to each node using block placement
- Data placement does not change during computation



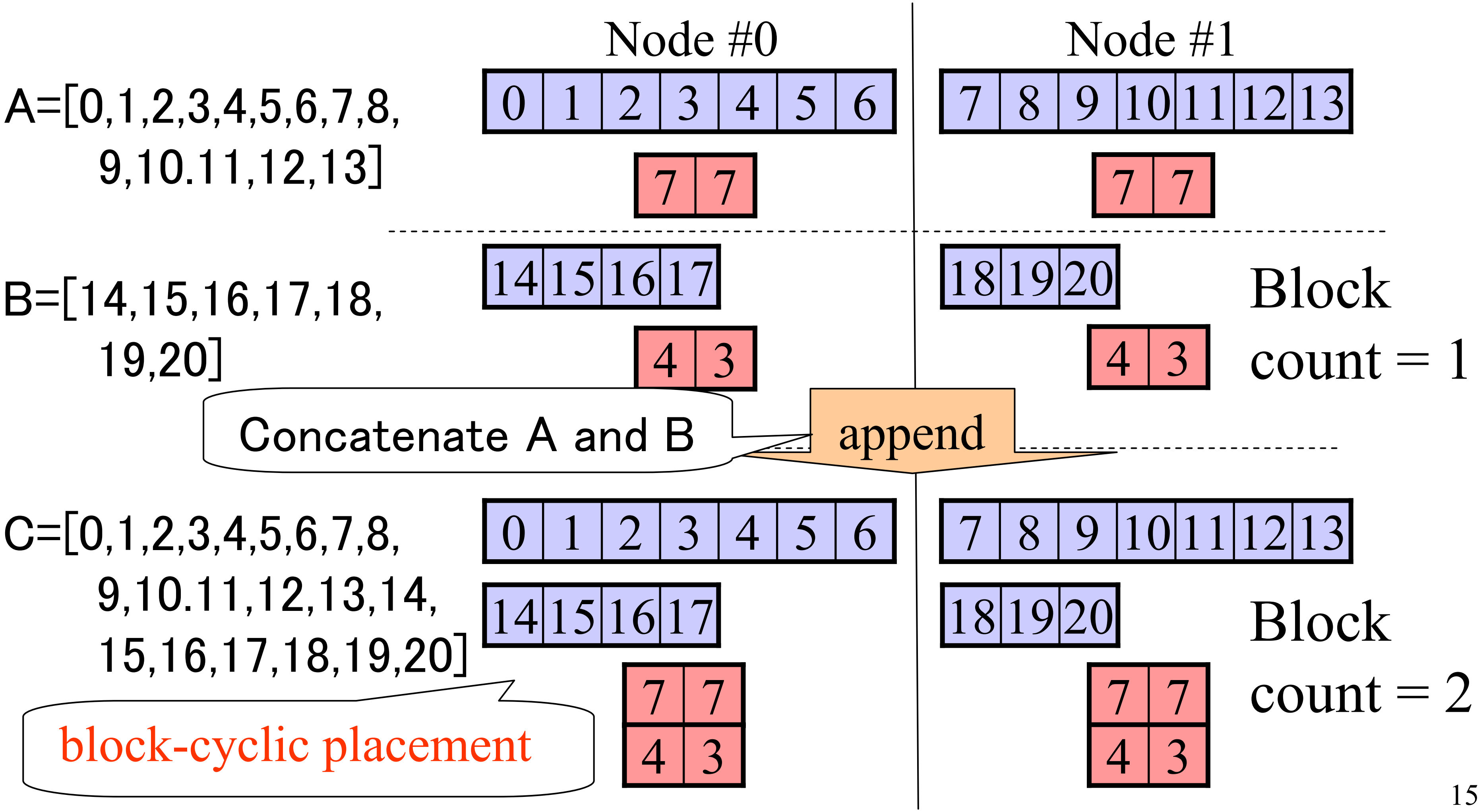
Data Structure of Variable-Length List (1)

- Each node has to know the latest information on the numbers of elements in other nodes



Data Structure of Variable-Length List (2)

- When we concatenate two lists, we adopt block-cyclic placement without relocating the entire amount of data



Micro-benchmark (1)

- Experimental environment
  - Pentium4 3.0GHz, Mem 1GB, Linux 2.6.8, 1000BaseT Ethernet

- We measured the execution times for applying `map`, `reduce`, and `scan`, and the data relocation in a list
  - Input list: 80 million elements
  - block count: from 1 to 4,000
  - two functions: short/long execution time

Micro-benchmark (2)

Execution time (s)

block count	1	10	100	1000	2000	3000	4000
Map (short)	0.0128	0.0129	0.0128	103%	0.0129	0.013[illegible]	0.0132
Reduce (short)	0.0183	0.0182	0.0183	109%	0.0194	0.019[illegible]	0.0200
Scan (short)	0.0407	0.0408	0.0411	142%	0.0484	0.053[illegible]	0.0580
Map (long)	16.8	16.8	16.8	100%	16.8	16.[illegible]	16.8
Reduce (long)	16.9	16.9	16.9	101%	16.9	16.[illegible]	17.0
Scan (long)	33.8	33.8	33.8	101%	33.9	34.[illegible]	34.1
Data Reloaction	-	3.74	4.64	4.67	4.66	4.62	4.72

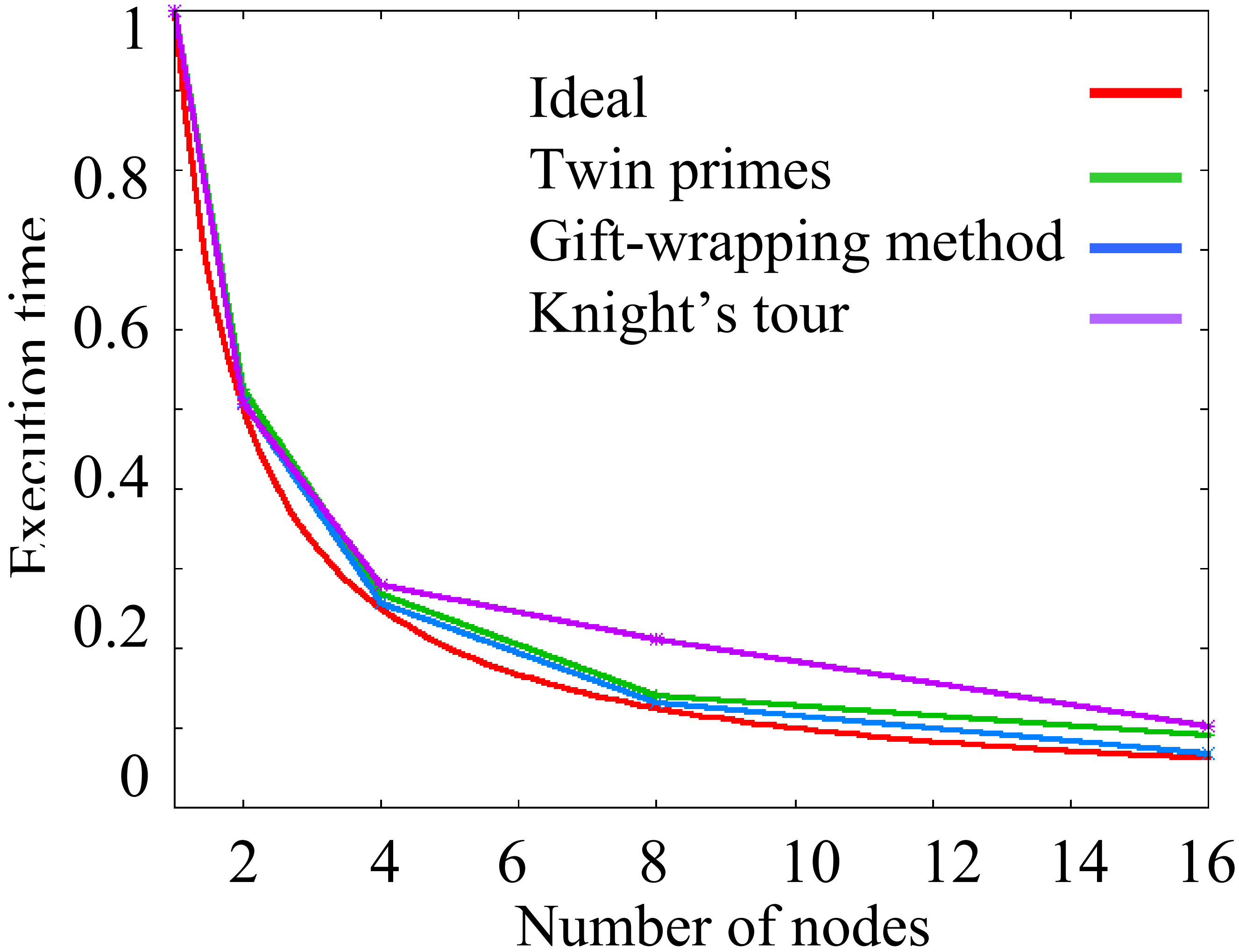
The overheads of data relocation are large

➔ It is effective to delay the relocation of data

Macro-benchmark (1)

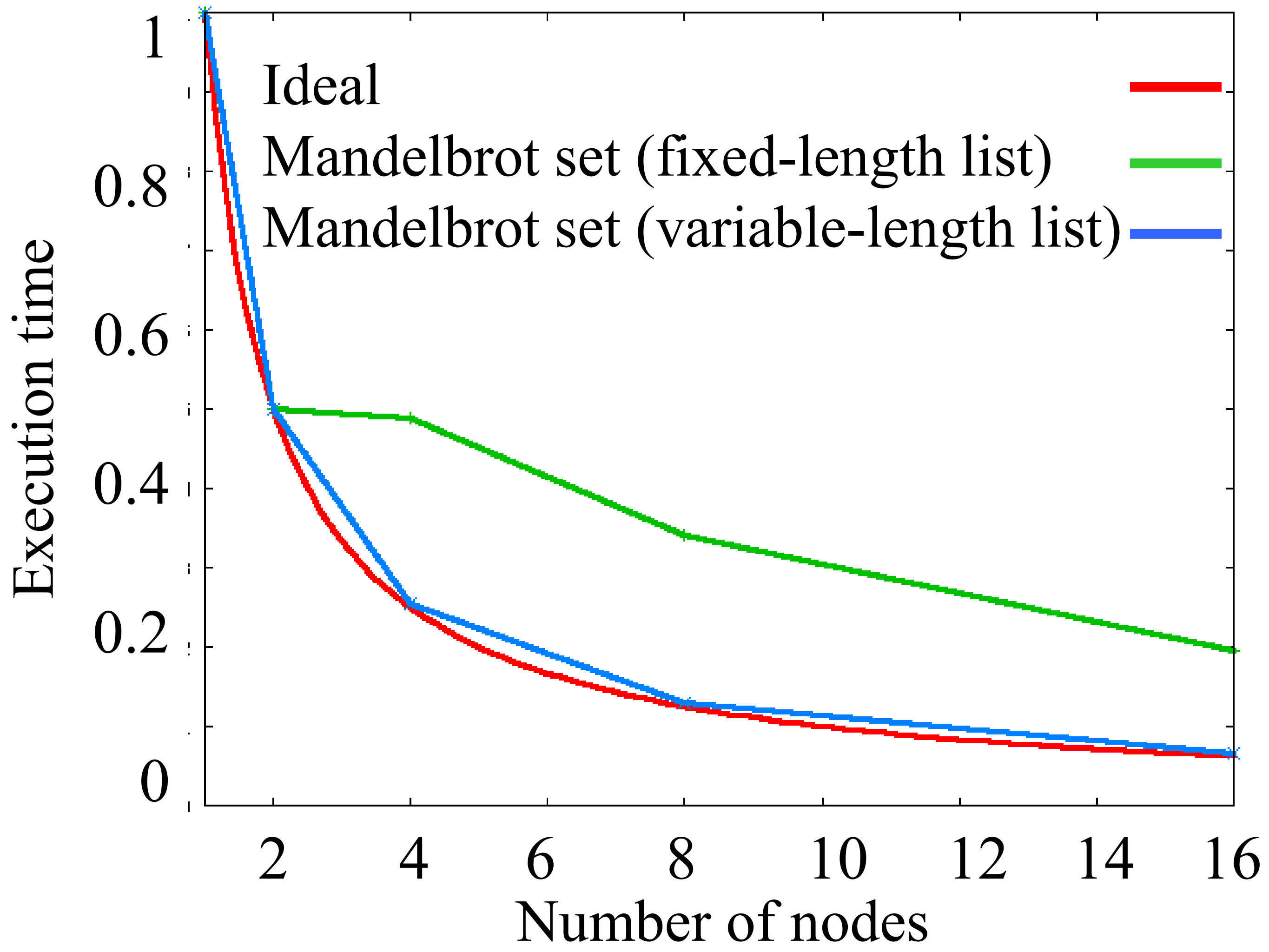
- Type1
  - Twin primes（list of 10 million integers）
  - Gift-wrapping method（1 million points)
- Type2
  - Knight’s tour (5 × 6 board)
- Type3
  - Mandelbrot set (1,000 ×1,000 coordinates)
    - Using variable-length lists with 100 iterative calculations ×100 times
    - Using fixed-length lists with 10,000 iterations

Macro-benchmark (2)



These results indicate excellent performance in all problems

Macro-benchmark (3)



Programs with variable-length lists show good speedups

Parallel Skeleton Libraries

- P3L [95], Muesli [02], Quaff [06]
  - support data parallel skeletons
  - offer lists (distributed one-dimensional arrays)
- eSkel [05]
  - supports task parallel skeletons but does not support data parallel skeletons for list like `map` and `reduce`
- Muskel [07], Calcium [07]
  - a Java skeleton library on a grid environment

These libraries do not support variable-length lists

Another Group of Libraries

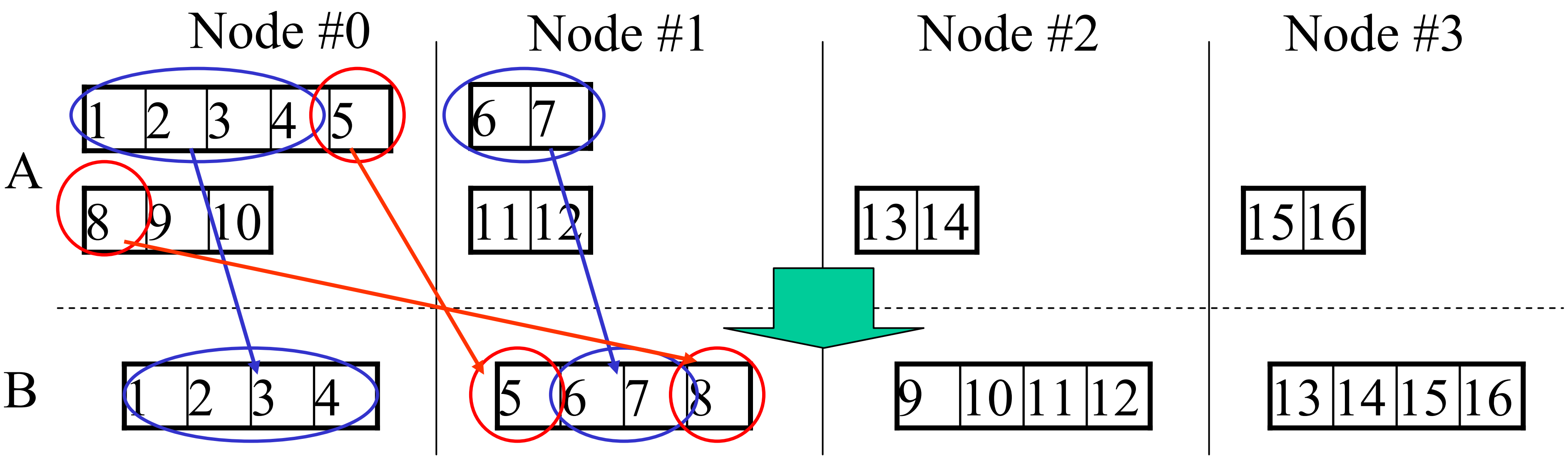
- STAPL [07]
  - Offers variable-length arrays and lists (pVector, pList)
  - Provides the same operations as C++ STL
  - Does not have operations such as concatmap

- Data Parallel Haskell [08]
  - Offers distributed nested lists
  - Provides filter, concatmap, and append
  - Only targets shared-memory environments

Conclusion

- We proposed parallel skeletons for variable-length list and their implementation
  - We proposed skeletons and operations for variable-length list, e.g., `concatmap`, `filter`, and `append`
  - We adopted a block-cyclic representation of lists with size tables
  - We confirmed the efficiency of our implementation through tests in various experiments

Data relocation (1)

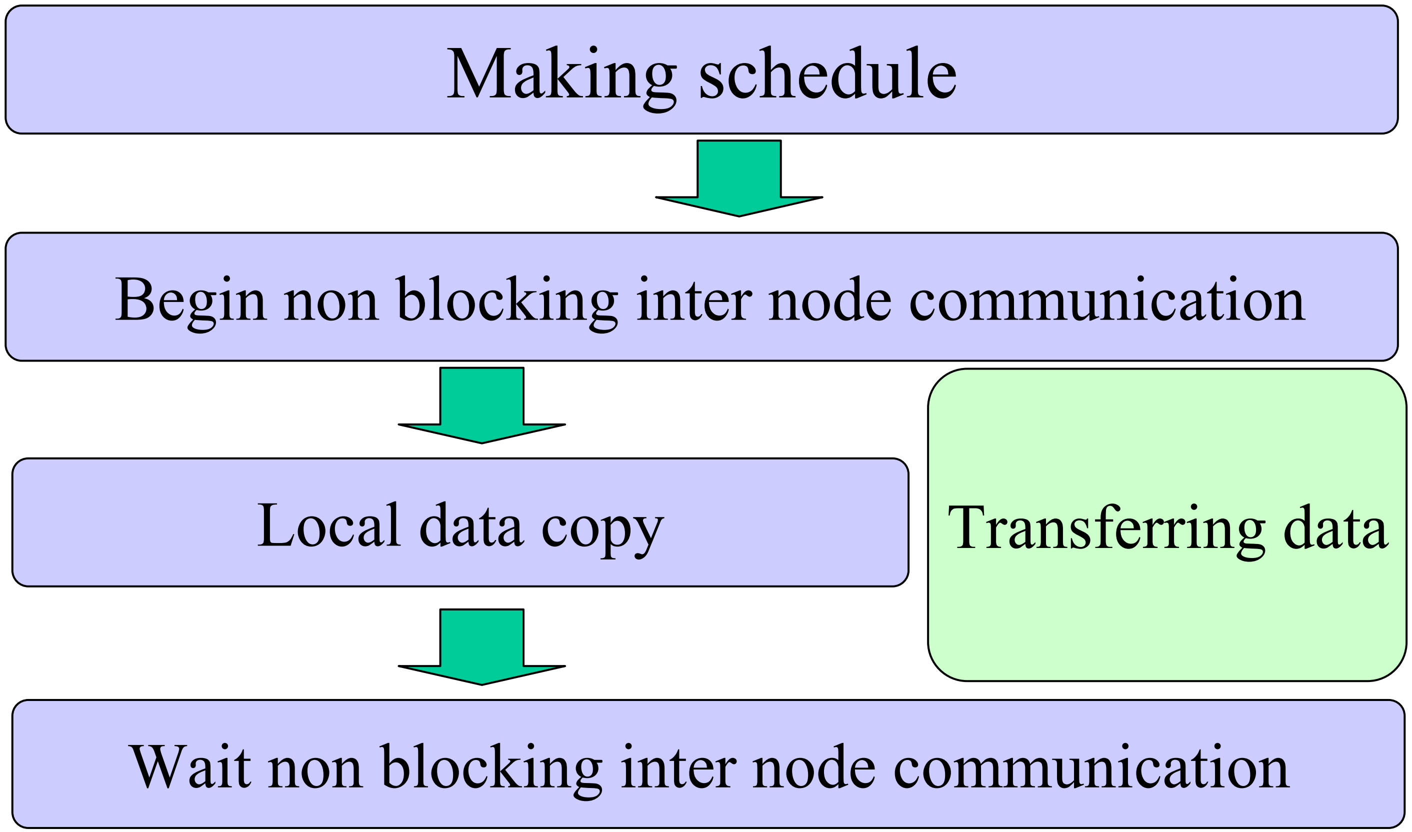


source	dest	
#0	#0	(1,2,3,4)
#0	#1	(5)
#1	#1	(6,7)
#0	#1	(8)
...	...	...

local data copy

inter node communication

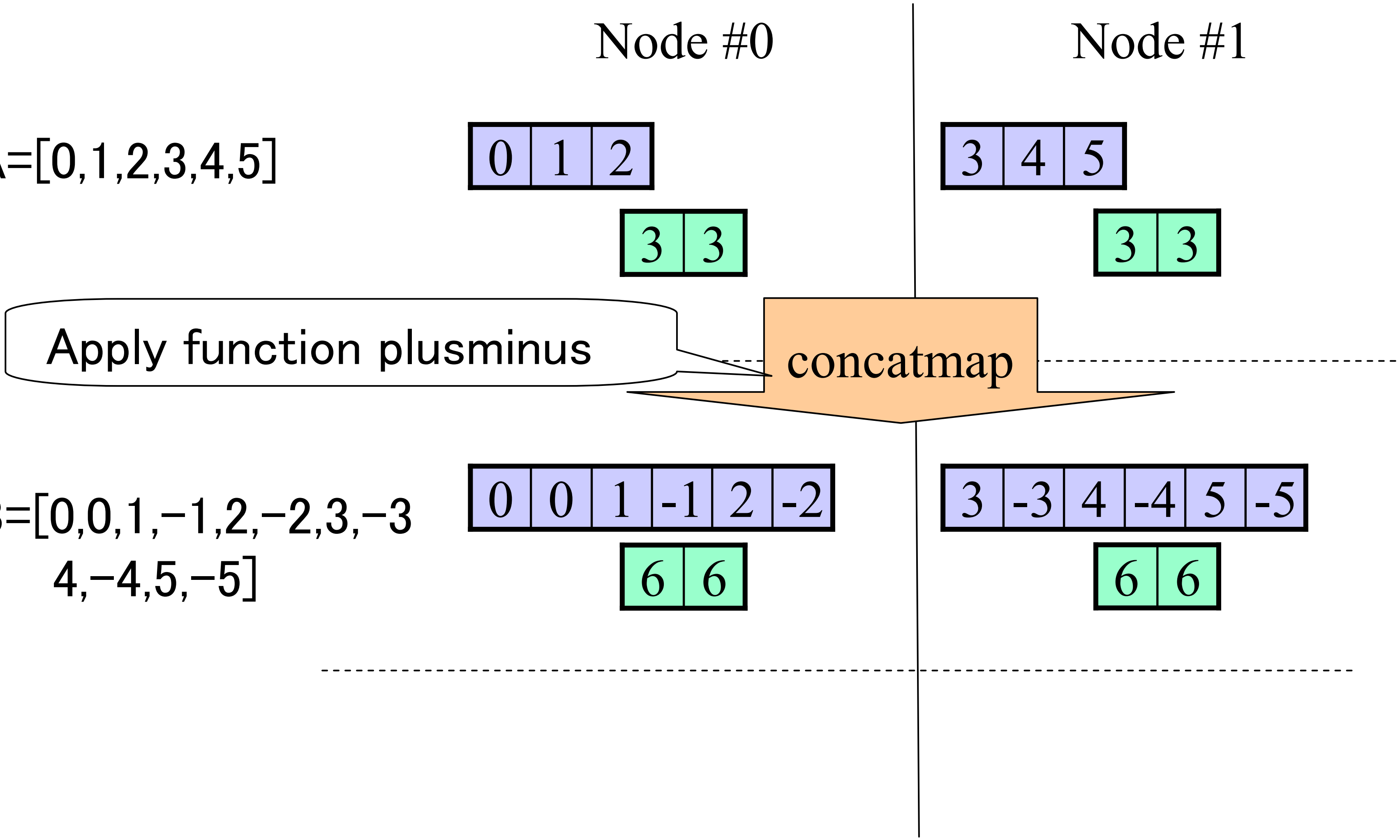
Data relocation (2)



Cocatmap

A=[0,1,2,3,4,5]

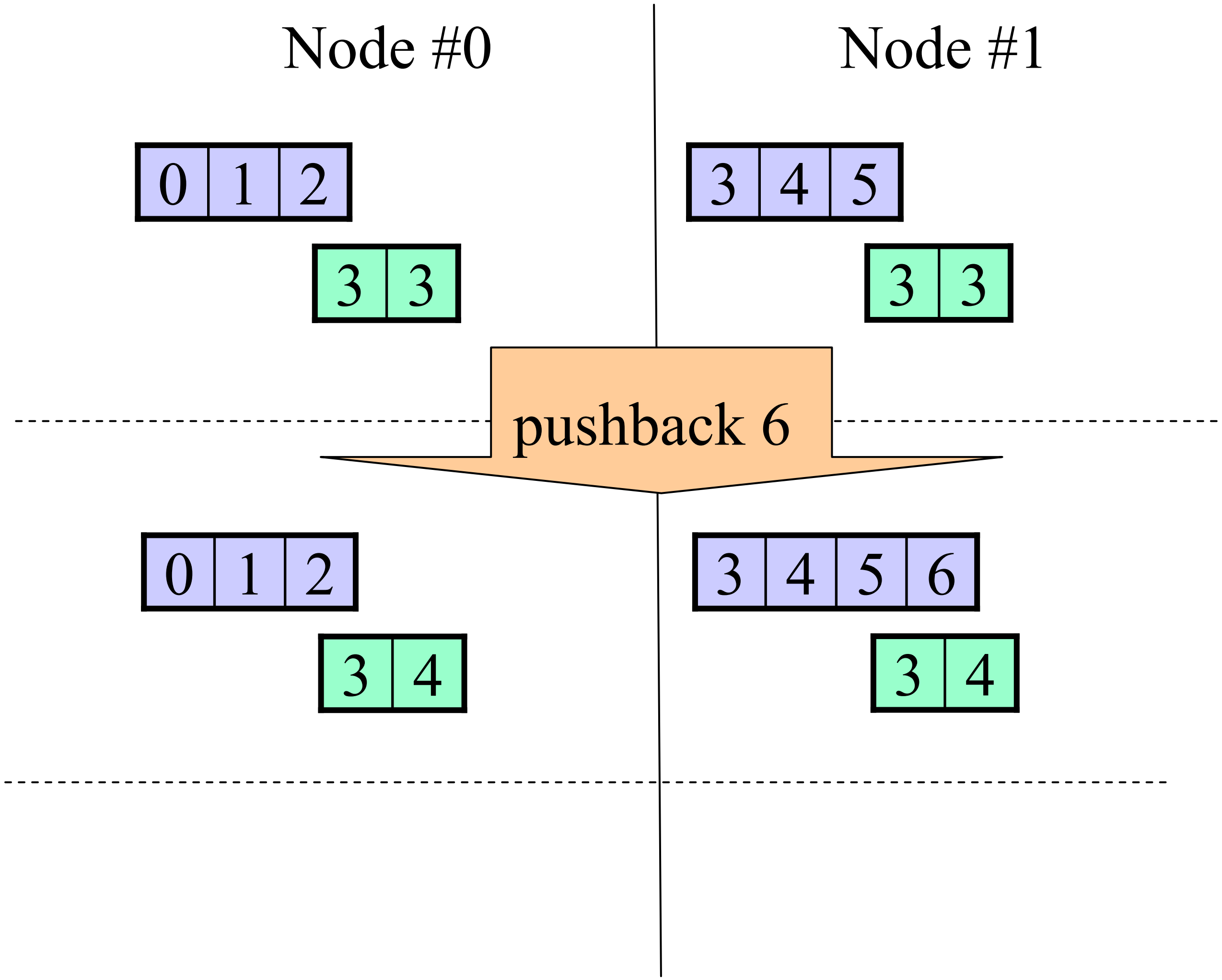
B=[0,0,1,−1,2,−2,3,−3
4,−4,5,−5]



Pushback

A=[0,1,2,3,4,5]

A=[0,1,2,3,4,5,6]



Condition of Data Relocation

$$imbalance\,(A) = n \times max\,(P_1, \ldots, P_n)/(P_1 + \cdots + P_n)$$

where $P_i = \sum_{j=1}^{m} p_{ij}$, $n =$ the number of nodes, $m =$ the block count of A,

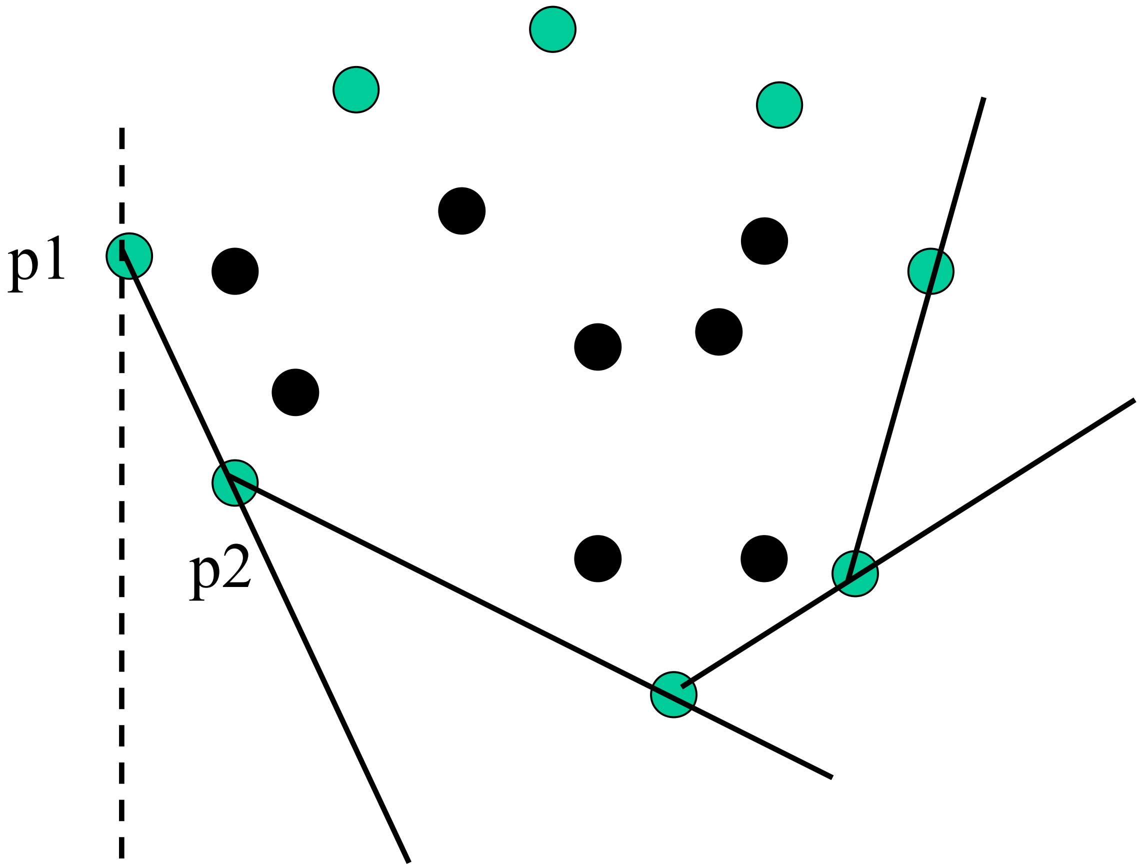
$p_{ij} =$ the number of elements of the j-th block at the i-th node.

imbalance(A) > 1.5

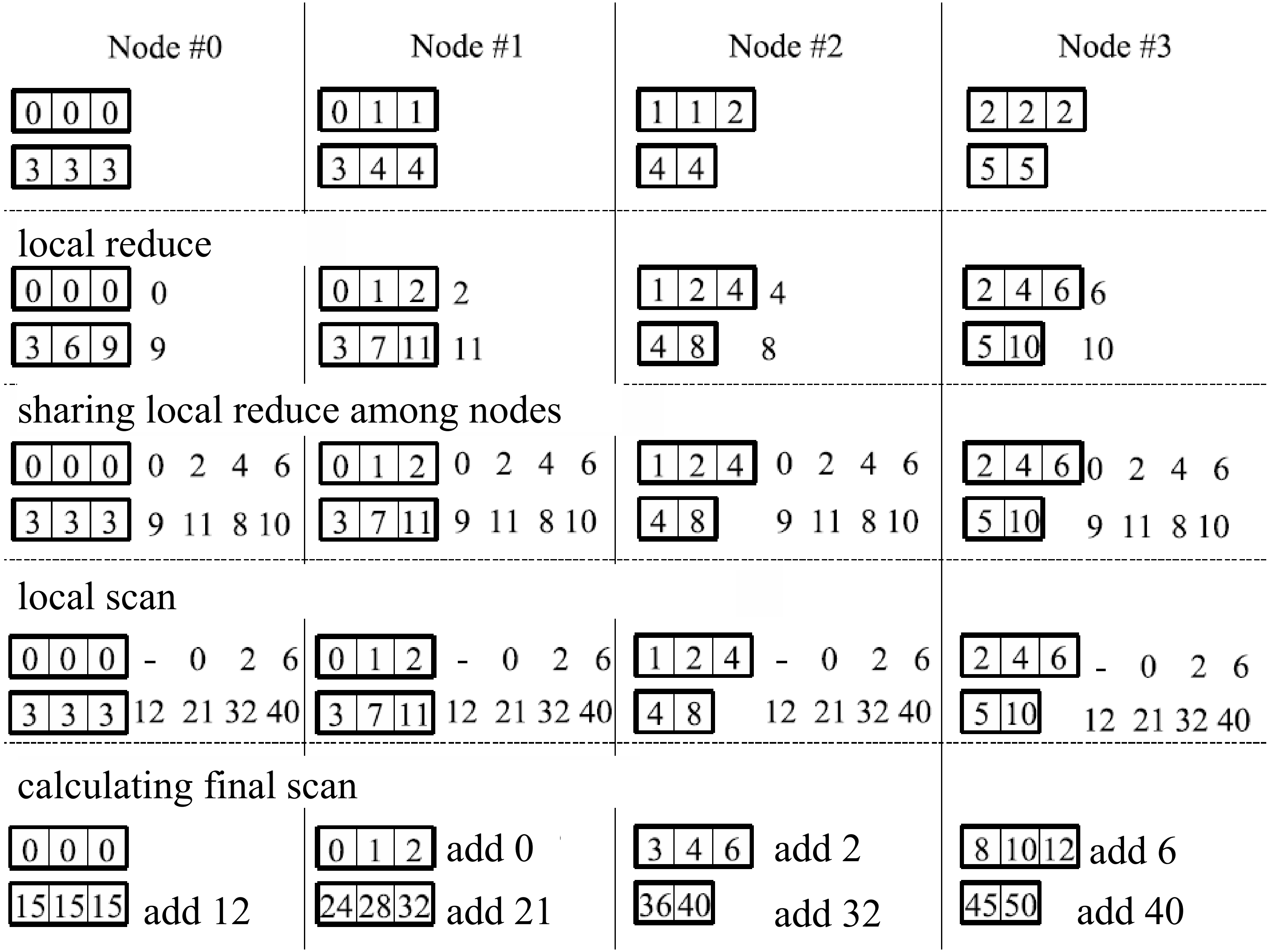
➔ data relocation

Convex hull

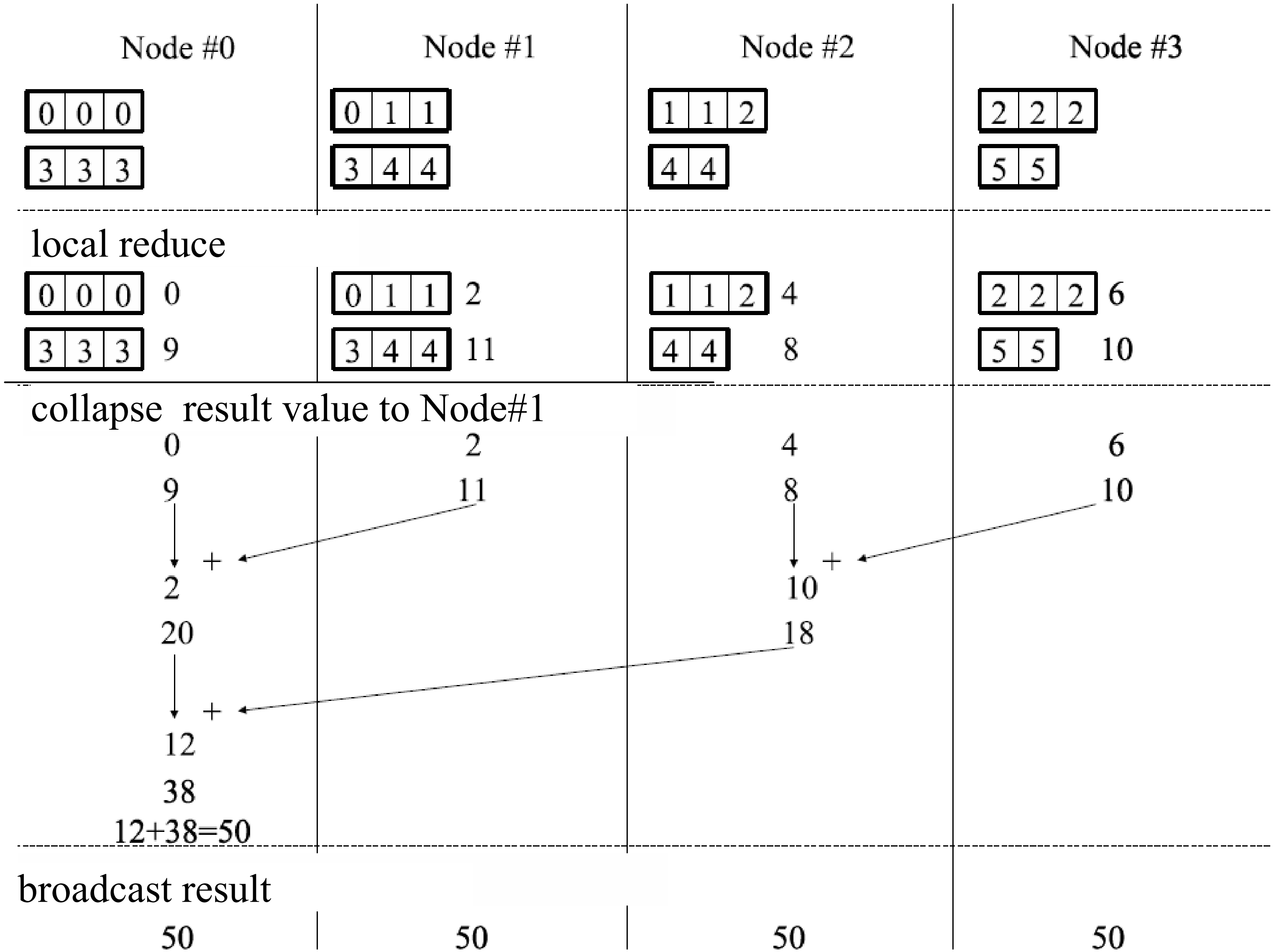
- Gift wrapping method



Scan in Variable-Length List



Reduce in Variable-Length Lists



Zip in Variable-Length Lists