Load Balanced Parallel Radix Sort

Andrew Sohn

Computer Science Dept.

New Jersey Institute of Technology

Yuetsu Kodama

Parallel Computer Architecture Laboratory

Electrotechnical Laboratory, Japan

Presented at the 12th ACM International Conference on Supercomputing
Melbourne, Australia, July 11-14, 1998

Overview

- ✔ Serial and parallel radix sorting
- ✔ The load balancing problem in radix sorting
- ✔ Balanced radix sorting
- ✔ Experiment settings and implementation details
- ✔ Experimental results and discussion
- ✔ Conclusions

Parallel Sorting

		Radix	Bitonic	Sample
Comp	amount	low	moderate	high
	pattern	regular	irregular	irregular
	load imbalance	high	moderate	high
Comm	amount	moderate	high	high
	pattern	irregular	regular	irregular
	load imbalance	high	low	high

Radix Sort

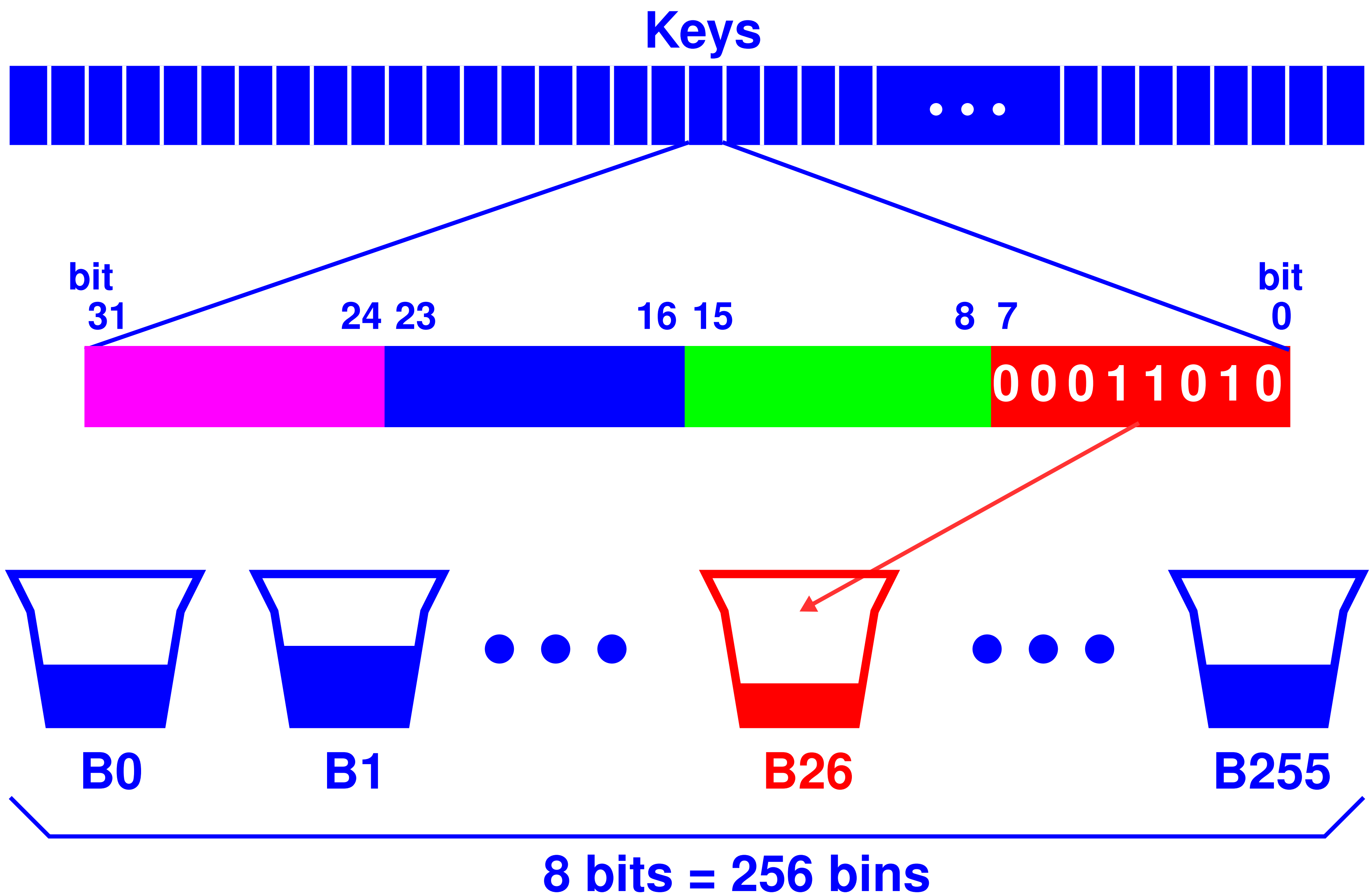
Count

Count of the number of keys for each bin (bucket).

Move

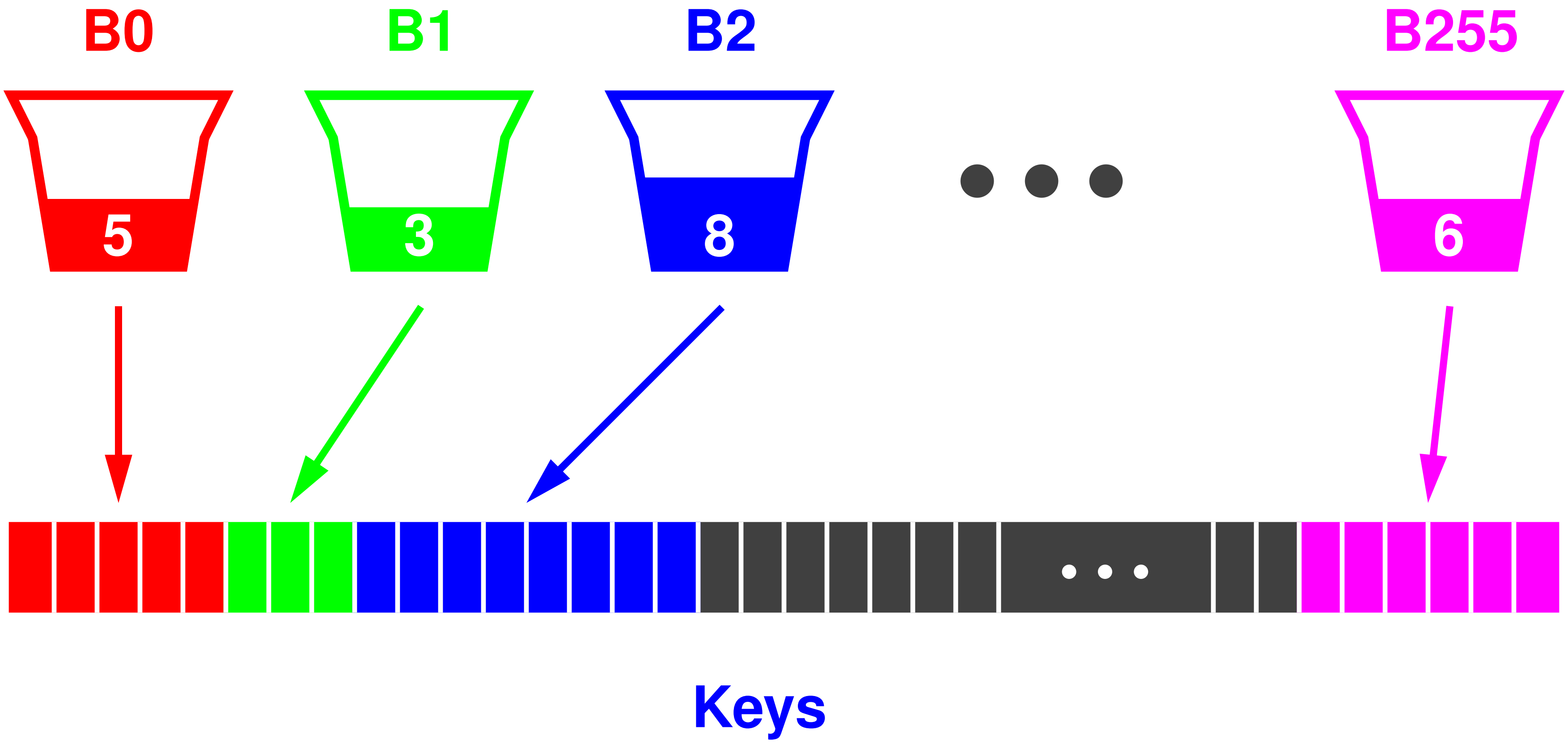
Rearrange the keys according to the bin count.

Bin Count



Keys
bit
31
24 23
16 15
8 7
bit
0
0 0 0 1 1 0 1 0
B0
B1
B26
B255
8 bits = 256 bins

Move



Parallel Radix Sort

Local count

Count locally the number of keys in each bucket.

Local move

Move locally the keys to appropriate buckets.

Interprocessor transpose

All-to-all transpose the bin count information across processors to find the prefix sum of each bucket.

No key movement, only bin count information

Interprocessor move

Move keys to appropriate processors.

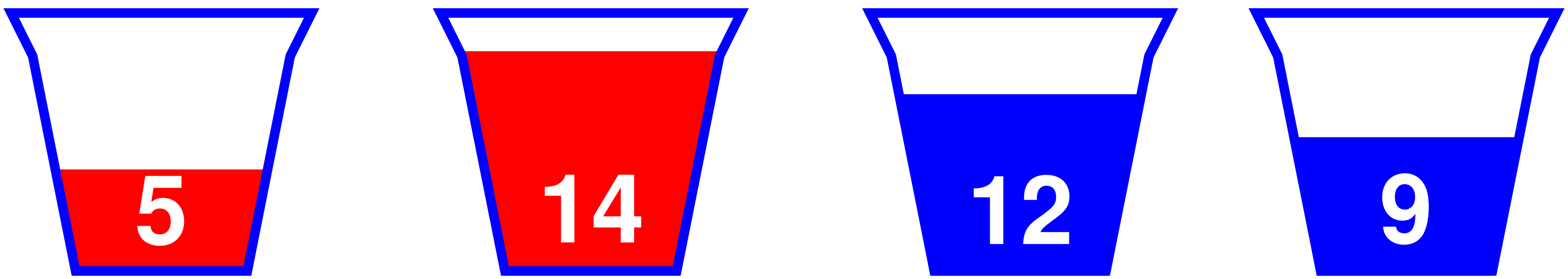
Interprocessor Transpose

	B0	B1	B2	B3
P0	1	3	4	2
P1	3	6	1	0
P2	0	3	5	2
P3	1	2	2	5

Assume
- 4 processors
- 4 buckets

P0	1	3	0	1	B0
P1	3	6	3	2	B1
P2	4	1	5	2	B2
P3	2	0	2	5	B3

Load Imbalance!



B0 on P0 | **B1** on P1 | **B2** on P2 | **B3** on P3

Different number of keys in each processor

⬇

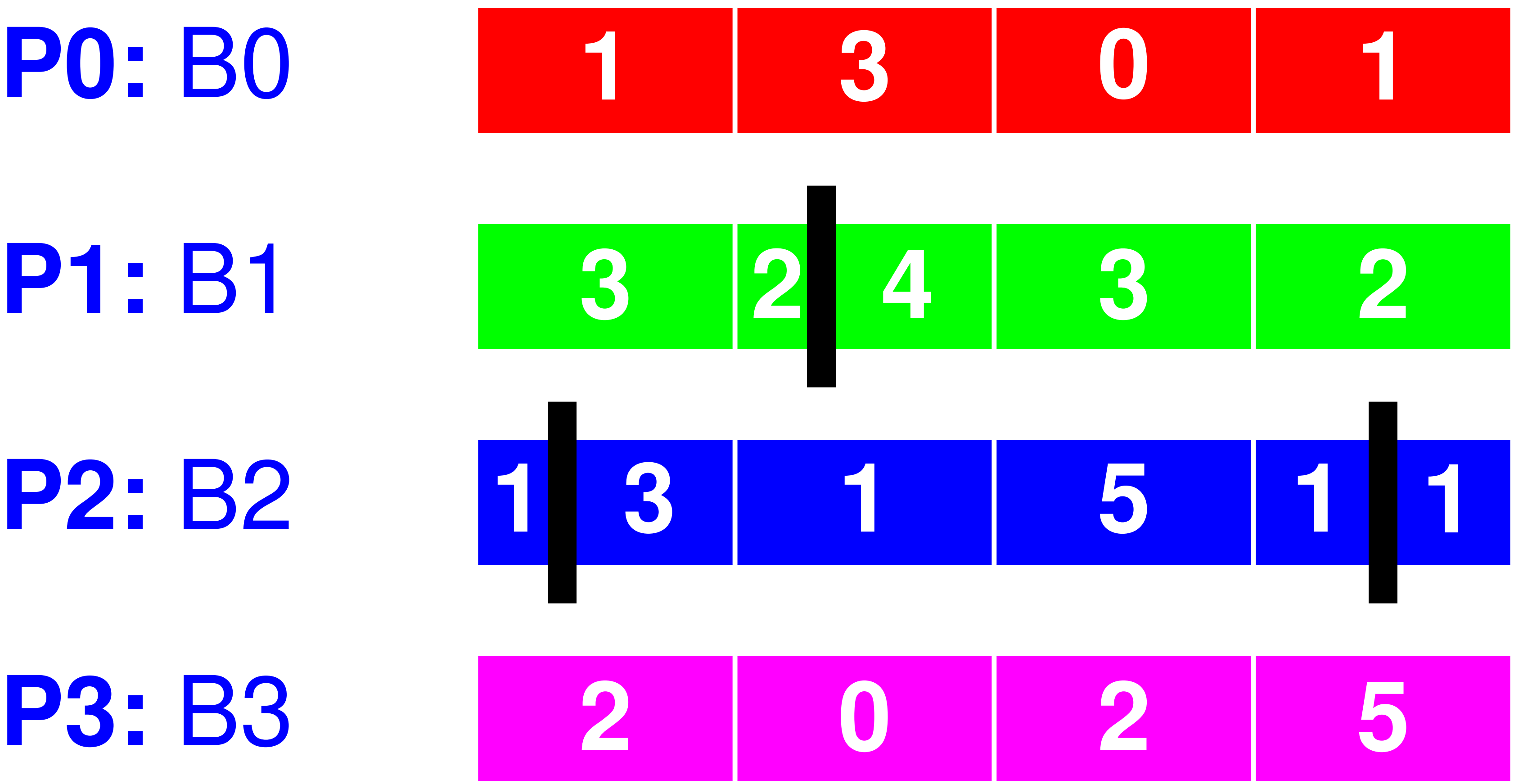
Varying computation and communication loads

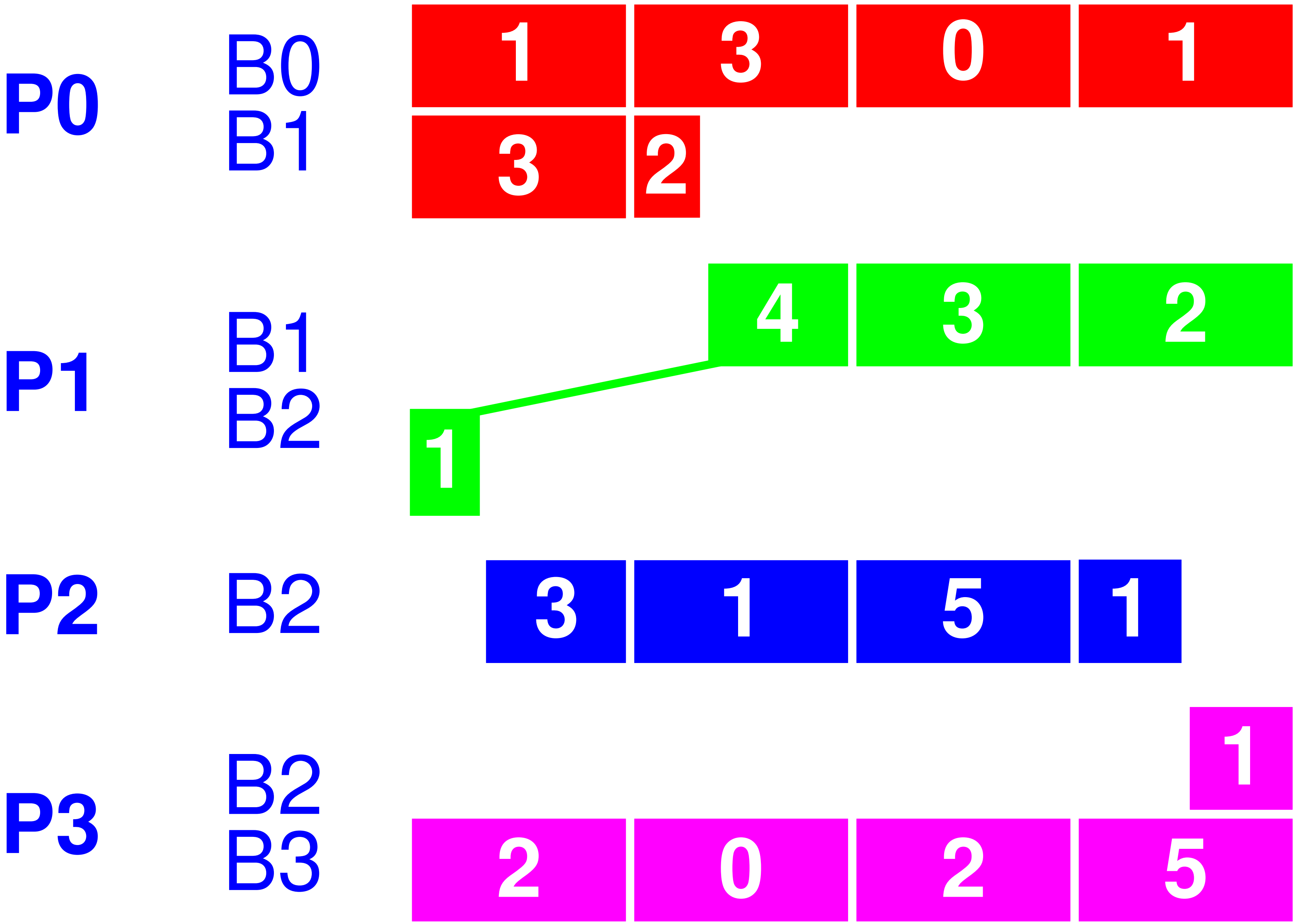
⬇

Performance to be determined by the critical path

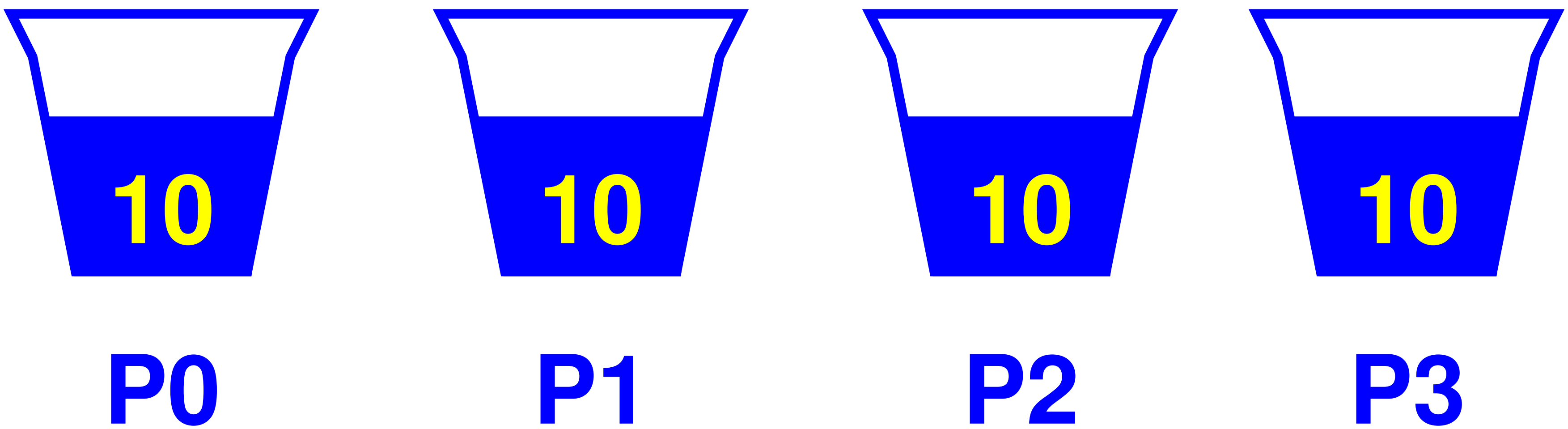
Solution?

Spill




P0
B0
1 3 0 1
B1
3 2
P1
B1
4 3 2
B2
1
P2
B2
3 1 5 1
P3
B2
1
B3
2 0 2 5

Balanced!



Experiment Setting

- Two distributed-memory multiprocessors: IBM SP-2 and Cray T3E, up to 64 processors.
- Five different key initialization: Gauss, Random, Zero, Bucket, and Stagger.
- Implemented in Message Passing Interface.
- Serial version about 50 lines of C code.
  Parallel version about 150 lines of C code,
  including parallel constructs but the initialization code.
- Data size: 512M 32-bit integers, 128M 64-bit doubles.
- Compared with other radix sorting implementations.
- Compared with the best performing sample sorting.

Key Initialization

- ✔ **Random**: calling C random() function.
- ✔ **Gauss**: four random() calls and divided by four.
- ✔ **Zero**: the same as random except 10% set to zero.
- ✔ **Bucket**:
  Keys in each processor are divided into P buckets.
  Bucket i has random keys between $i*n/P$ and $(i+1)*n/P$.
- ✔ **Stagger**: Processor P_i, has keys between
  $(2*i+1)*n$ and $(2*i+2)*n$ for $i < P/2$,
  $(i-P/2)*n$ and $(i-P/2+1)*n$ for $i \geq P/2$.

For the 32-bit machine IBM SP-2:

- ✔ 64-bit doubles are split into two 32-bits.
- ✔ Each 32-bit is initialized separately.
- ✔ All the 64 bits are therefore fully initialized.

Execution Time

On a 64 processor SP-2

# of keys	32-bit integers					64-bit doubles
	random	gauss	zero	bucket	stagger	gauss
1 M	0.137	0.135	0.136	0.116	0.136	0.313
2 M	0.181	0.182	0.179	0.170	0.172	0.456
4 M	0.264	0.254	0.252	0.212	0.210	0.684
8 M	0.399	0.395	0.414	0.330	0.336	1.164
16 M	0.684	0.678	0.750	0.560	0.606	2.120
32 M	1.142	1.167	1.308	1.148	1.298	4.064
64 M	2.178	2.225	2.477	2.286	2.665	7.869
128 M	4.272	4.465	4.905	4.598	5.524	15.611
256 M	9.250	9.485	9.601	10.000	12.072	-
512 M	18.673	19.427	18.900	19.656	24.581	-

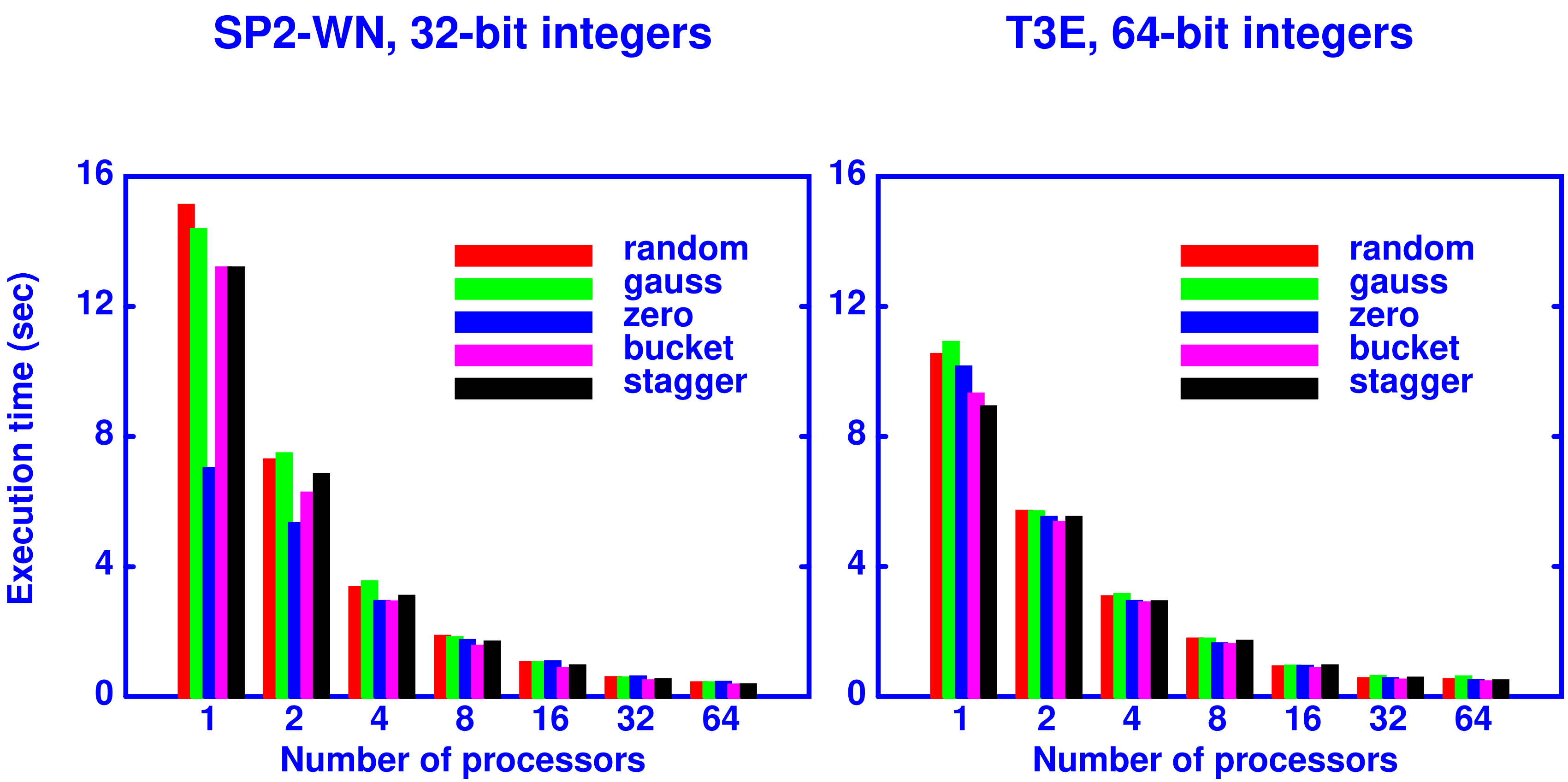
Execution Time

On a 64-processor T3E

# of keys	64-bit integers					64-bit doubles
	random	gauss	zero	bucket	stagger	gauss
1 M	0.221	0.228	0.191	0.180	0.223	0.220
2 M	0.301	0.317	0.261	0.236	0.266	0.307
4 M	0.413	0.447	0.364	0.326	0.337	0.450
8 M	0.497	0.575	0.464	0.425	0.455	0.576
16 M	0.632	0.758	0.612	0.581	0.647	0.752
32 M	1.087	1.222	1.057	1.082	1.212	1.314
64 M	2.055	2.217	1.944	2.042	2.255	2.518
128 M	3.975	4.084	3.736	3.919	4.458	4.904
256 M	7.598	7.693	7.312	7.616	9.052	-
512 M	18.474	19.787	17.715	18.270	20.445	-

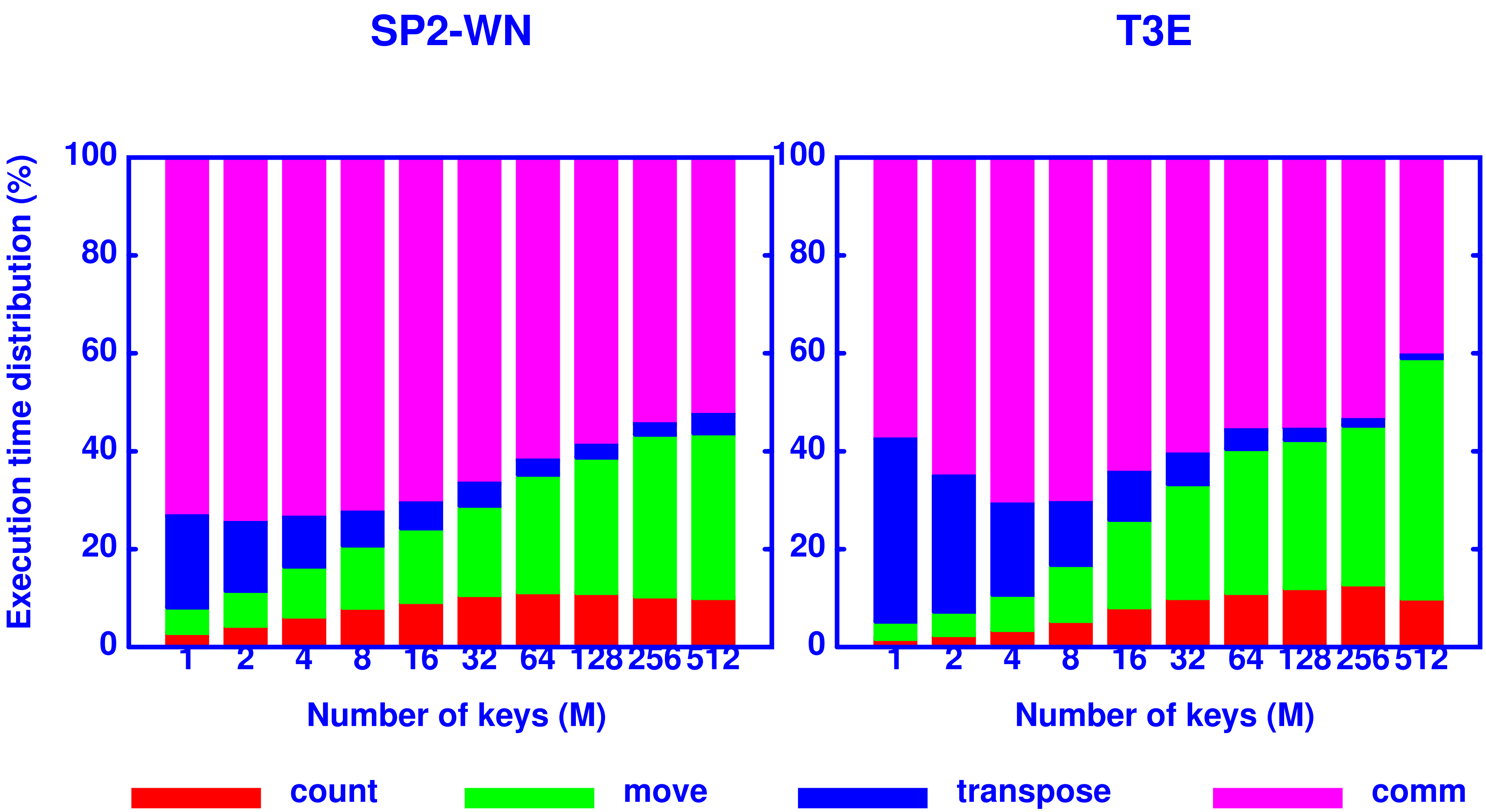
Execution Time

8M integers, Gauss initialization



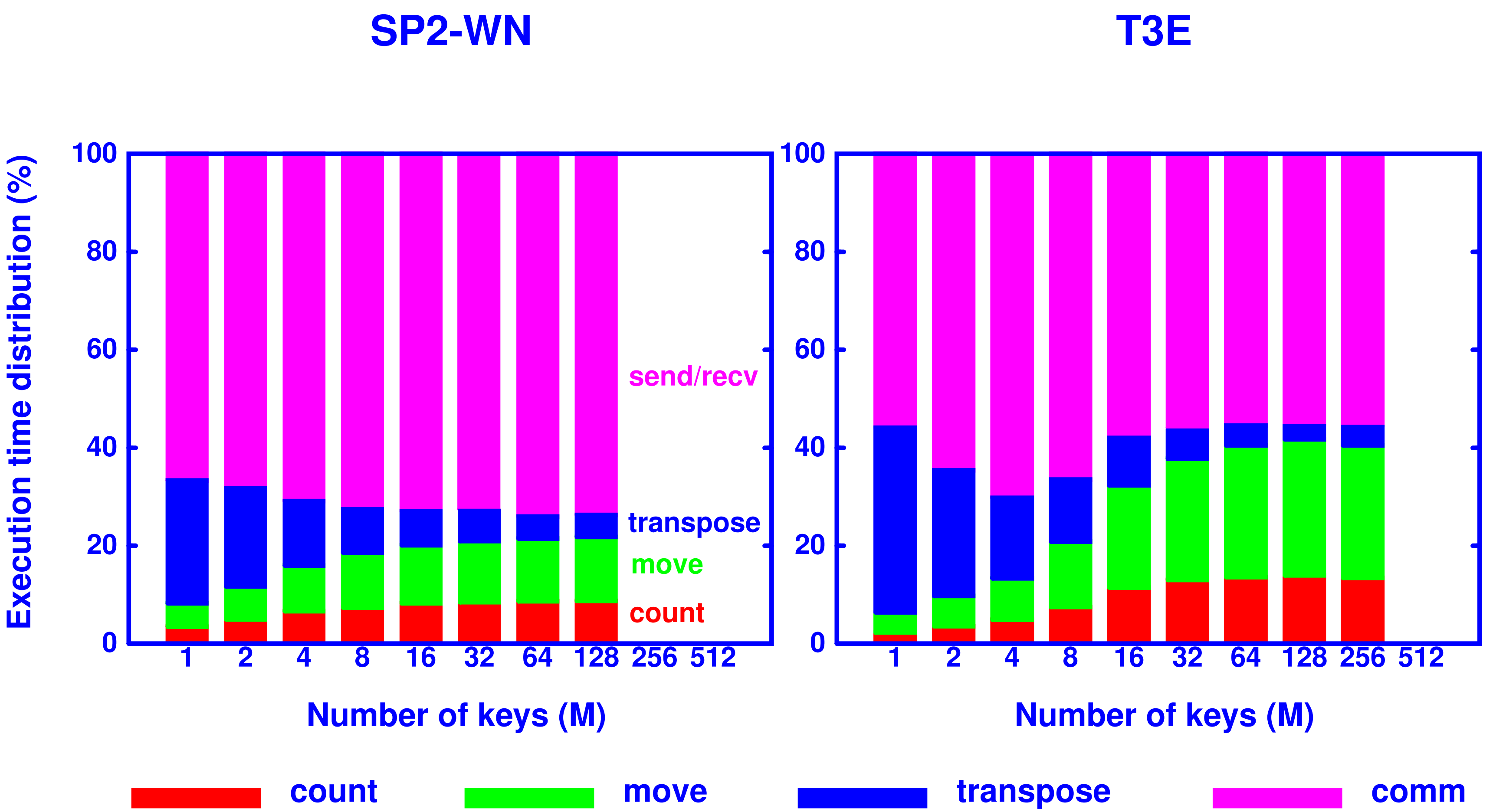
Time Distribution

64 processors, 32-bit integers, Gauss initialization



Time Distribution

64 processors, 64-bit doubles, Gauss initialization



Comparison with Other Radix

SP2-WN, 16 processors, 32-bit integer

Init method	Problem size	Execution time (seconds)			Improvement	
		Balanced	BHJ [9]	AIS [1]	BHJ/ Balanced	AIS/ Balanced
Random	64K	0.076	0.107	0.474	1.4	6.2
	1M	0.185	0.592	0.938	3.2	5.1
	8M	1.016	4.030	4.130	4.0	4.1
	64M	10.198	n/a	n/a	-	-
Gauss	64K	0.073	0.109	0.475	1.5	6.5
	1M	0.184	0.613	0.907	3.3	4.9
	8M	1.019	4.120	4.220	4.0	4.1
	64M	10.358	n/a	n/a	-	-

n/a = not available

Comparison with Sample Sort

SP2-WN, 32-bit 8M integers

	P=1	P=2	P=4	P=8	P=16	P=32	P=64	init
Balanced radix sorting	13.1	6.23	2.88	1.52	0.83	0.455	0.330	Gauss*
Random sample sorting	-	-	-	2.41	1.24	0.696	0.421	WR*
Regular sample sorting	-	-	-	3.12	1.57	0.864	-	WR*

Random/Balanced	-	-	-	1.6	1.5	1.5	1.3	
Regular/Balanced	-	-	-	2.1	1.9	1.9	-	

* Gauss performs essentially the same as WR in sample sorting.

Why Balanced Radix Sort Is Fast?

Balanced Radix Sort	Sample Sort
for (i=0; i<4; i++) {	local sort
local count and move	send/recv keys
all-to-all transpose	local sample selection
(256 integers/prcr)	
send/recv keys	1-to-all send/recv samples
}	(n/P^2 samples/processor)
	splitter selection
	on a single processor
	broadcasting splitters
	send/recv keys
	local merge

Conclusions

- ✔ The load imbalance problem of parallel radix sorting has been solved.
- ✔ The key idea is to spill those processors with more than n/p keys to neighboring processors.
- ✔ Implemented in C with MPI on SP-2 and T3E.
- ✔ Five key initialization methods were used.
- ✔ Balanced radix sort is up to six times faster than two other radix sort implementations based on the same machine and key initialization.
- ✔ Balanced radix sort is up to twice faster than sample sorting.

Balanced Radix Sort

- ✔ 2 to 6 times faster than other radix sort
- ✔ up to twice faster than sample sort