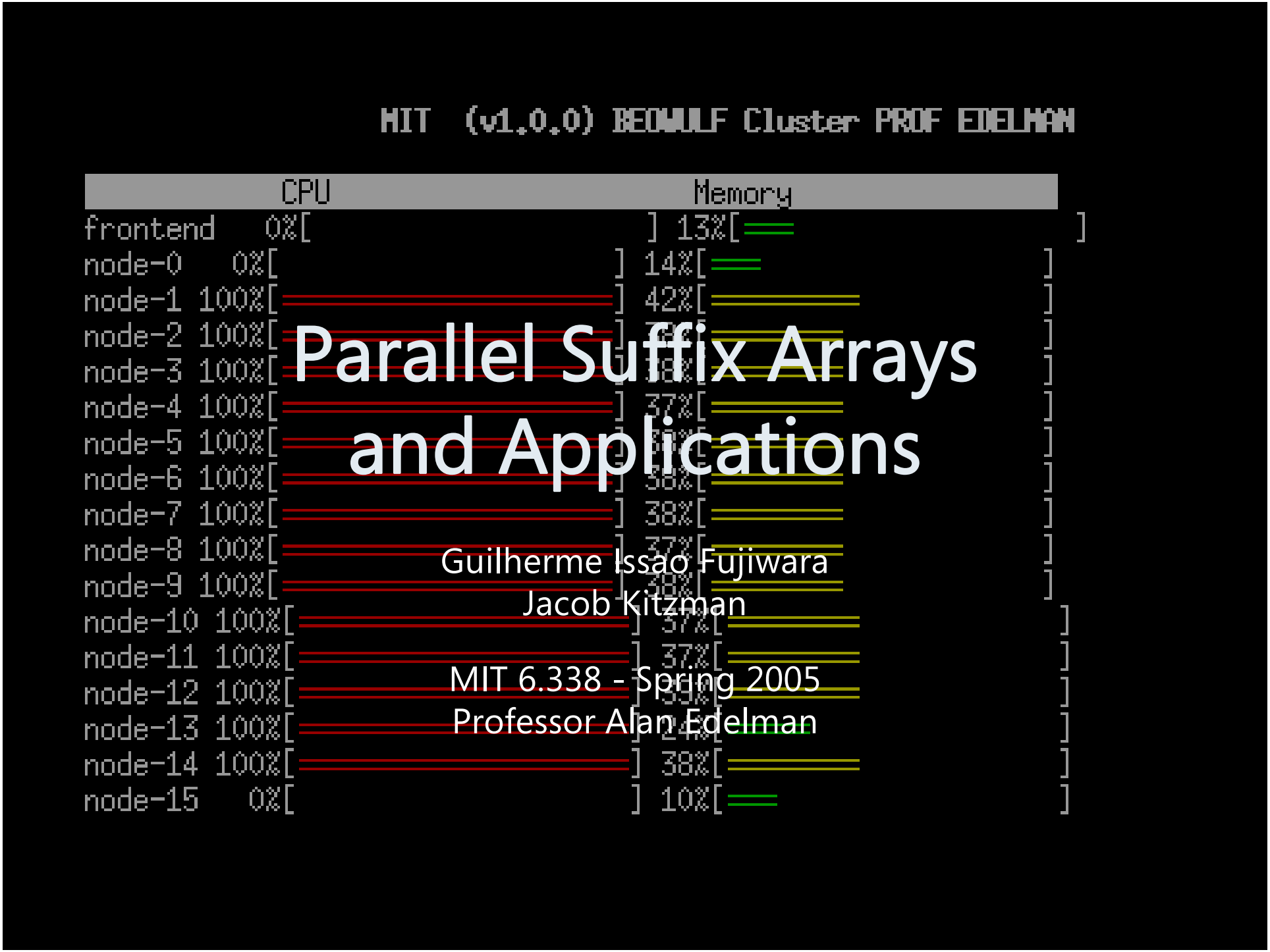


Parallel Suffix Arrays and Applications

Guilherme Issao Fujiwara
Jacob Kitzman

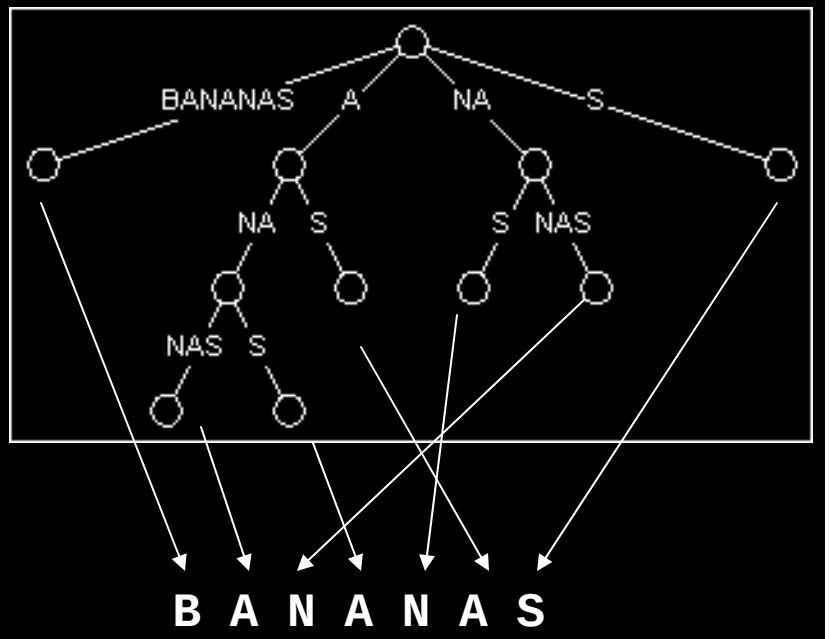
MIT 6.338 - Spring 2005
Professor Alan Edelman

Order of Business

- Suffix Arrays 101
- Applications
- Real World Parallel Suffix Sorting
- Our project
- Results

Suffix Arrays 101

- Suffix Trees
  - Internal structure of str.
  - Build O(n) time ☺
    O(n|Σ|) space ☹
  - Many O(n) time apps…



1 **ANANAS**
3 **ANAS**
5 **AS**
0 **BANANAS**
2 **NANAS**
4 **NAS**
6 **S**

- Suffix Arrays = Suffix Sorting
  - Myers EW, Manber U 1993
  - Obtain sorted order of suffices
  - Space-efficient: O(n)

Nelson M 1996 Dr. Dobbs J

Selected Applications

- O(n) pattern matching, online unlike KMP
- Genome-scale (10^6 – 10^9 chars) alignment
  - MUMMER (Delcher *et al*, TIGR)
    - build suffix tree for query and reference sequences,
    - find set of maximal unique matches,
    - extend and add gaps with DP.
- EST Clustering
  - High redundancy, low quality, short, … and lots!
  - Avoid $O(n^2)$ all-pairs comparisons.
- Data Compression

Suffix Sorting

- Naively sort?
  - Comparison-based: $\Omega(n^2 \log n)$ time,
  - Radix: $\Omega(n^2)$ time
- Trees
  - Construct, walk in O(n) space and time.
    - Large overhead => impractical.
    - Even highly optimzed (Giegerich, 2003) ~9 bytes/char
- Suffix Arrays
  - Various worse-cased linear time approaches, often slower for practical n than O(n lg n) ones.

Larsen NJ, 1998, Franek F, 2004, Larsson 2004

Parallel Suffix Sorting

- Tree-based algo's parallelizable…
  - So far only theoretically, no implementations.
- We implemented approach of Futamura et al (2001).

Futamura N, Aluru S, Kurtz S. "Parallel Suffix Sorting", Proc 9th Intl Conf Adv Comp and Comm (*2001*)

Parallel Suffix Sorting

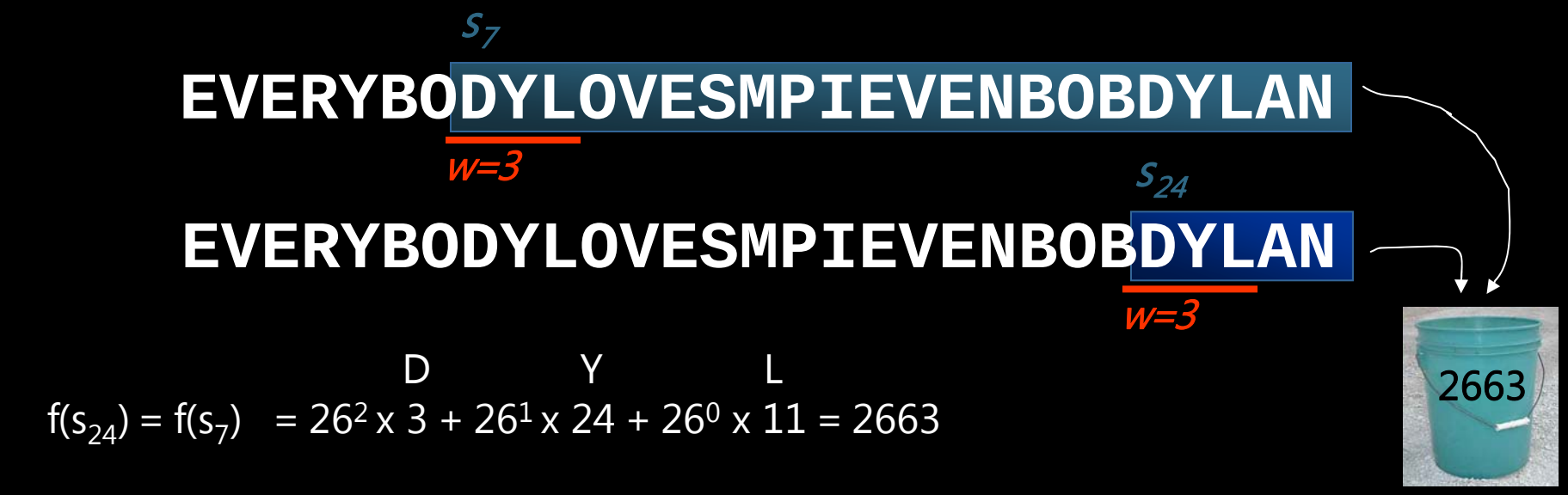
- Approach proposed by Futamura et al
  - Divide suffices into buckets
  - Sort within buckets, using general sorting algorithm
    - Bentley & Sedgewick, fast tripartioned quicksort.
  - Read out buckets in order

- Must to ensure buckets ordered… how?

Futamura N, Aluru S, Kurtz S. "Parallel Suffix Sorting", Proc 9th Intl Conf Adv Comp and Comm (*2001*)

Sorted Buckets

- Consider input suffix a base |∑| number
  - This maps suffix to an integer (bucket #)
- Actually just consider the first *w* 'digits'
  - *w* = 'window size'
  - Then suffices map to {0,1,…, $|\Sigma|^w$-1}



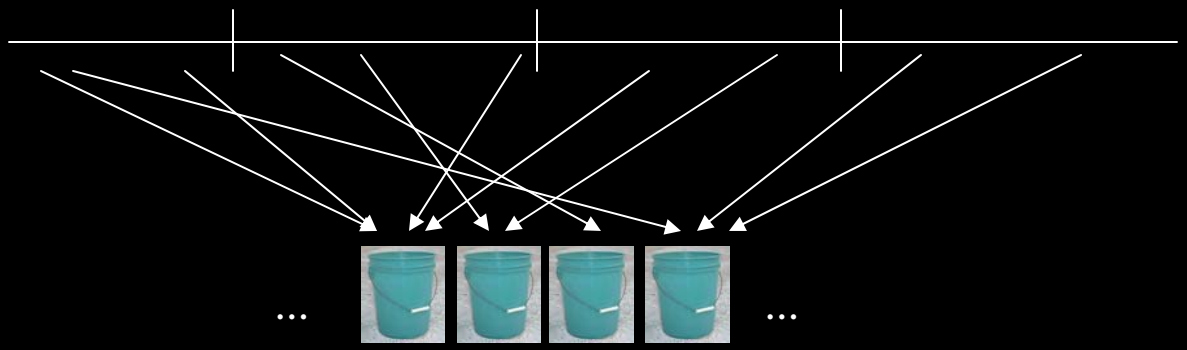
D Y L

$f(s_{24}) = f(s_7) = 26^2 \times 3 + 26^1 \times 24 + 26^0 \times 11 = 2663$

Now in parallel

- Overall goal: Distribute the buckets and sort individually.

1. Split string evenly across cpu's, find bucket mappings.



Now each bucket has suffices from each processor, but the bucket's contents are distributed…

s0 from cpu2, s4 from cpu2, s9 from cpu5, …

Load Balancing

2. Collect buckets and redistribute evenly for within-bucket sorting.

   Allreduce so everybody knows collective size of each (distributed) bucket.

   Agree upon load balancing:

   s.t. each cpu has a collection of whole buckets with roughly equal # of suffs.

Load Balancing

2. Collect buckets and redistribute evenly for within-bucket sorting.

   Allreduce so everybody knows collective size of each (distributed) bucket.

   Agree upon load balancing:

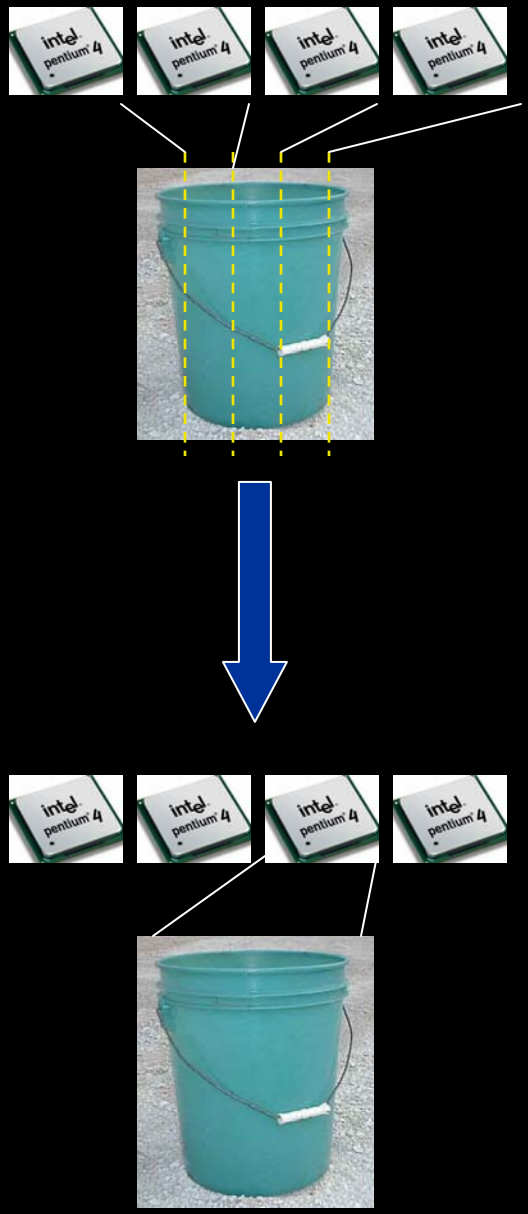
   s.t. each cpu has a collection of whole buckets with roughly equal # of suffs.

Collect buckets

2. Collect buckets and redistribute evenly for within-bucket sorting.

   Bucket-to-CPU assignment known but each CPU has suffs from many buckets.

   Send partial buckets to their owner nodes, and receive all suffices in the buckets that belong to me.



Sort!

3. Each node now has the suffices in its own bucket. Sort them locally, send sorted order back to head node.

Done!

Parallel Buckets: Running time

- Worst case: AAAAAAAAAAAAAAAAAAAA……
  - Everything in one bucket.
  - $\Omega(n^2 \log n)$ and same as sequential algorithm.
- For realistic inputs, not so bad
  - Total time O(n log n), and efficient so it beats many linear algo's.
  - Except for pathological inputs, we achieve linear speedup through np=30.

Application: Fast Compression

- Newer compression algorithms based on **Burrows-Wheeler Transform** (BWT)
- BWT is the most computationally-intensive step in compression. (bzip2, ACE, RAR)
- Efficiently compute BWT via suffix sorting.
- We integrated our parallel suffix sort with a compressor that currently relies on BWT
  - BWTZIP: A portable research-grade data compressor. (Lavarej ST, unpublished, 2005)

Burrows-Wheeler Transform

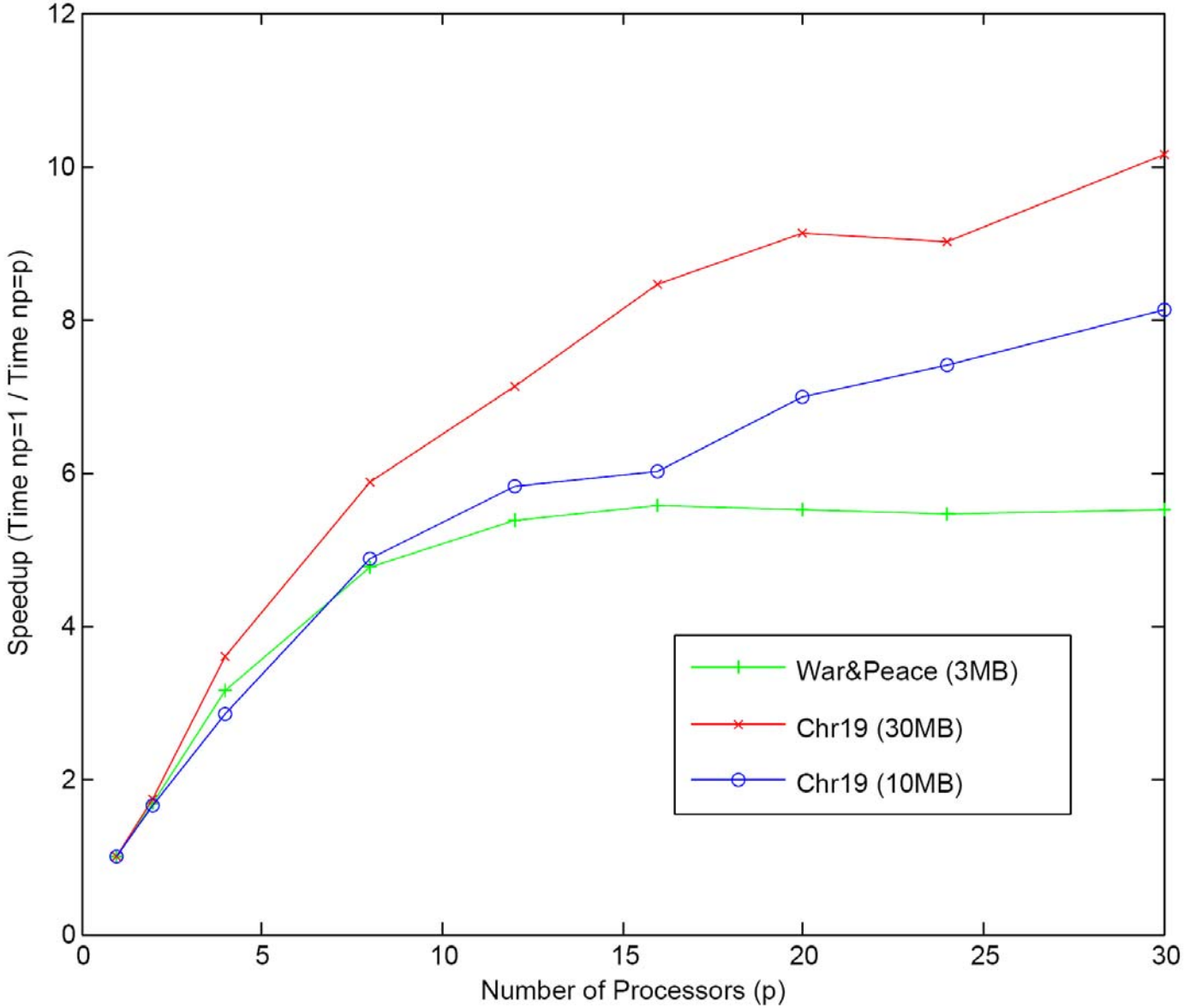
c_1	parallel
c_2	arallelp
c_3	rallelpa
c_4	allelpar
c_5	llelpara
c_6	lelparal
c_7	elparall
c_8	lparalle

Burrows-Wheeler Transform

c_1	parallel
c_2	arallelp
c_3	rallelpa
c_4	allelpar
c_5	llelpara
c_6	lelparal
c_7	elparall
c_8	lparalle

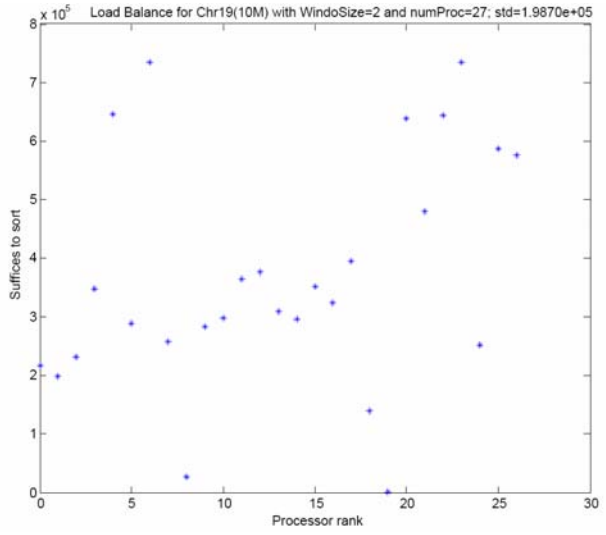
c_4	allelpar
c_2	arallelp
c_7	elparall
c_6	lelparal
c_5	llelpara
c_8	lparalle
c_1	parallel
c_3	rallelpa

Results: Performance

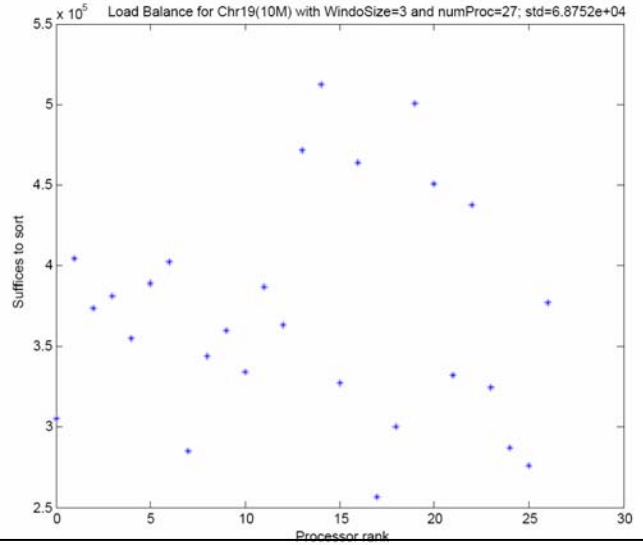


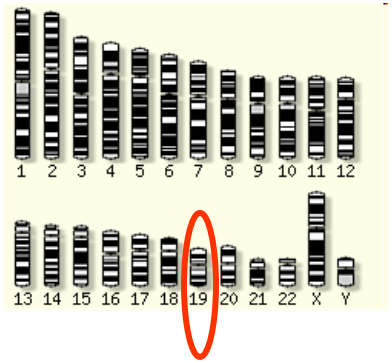
Results: Load Balancing

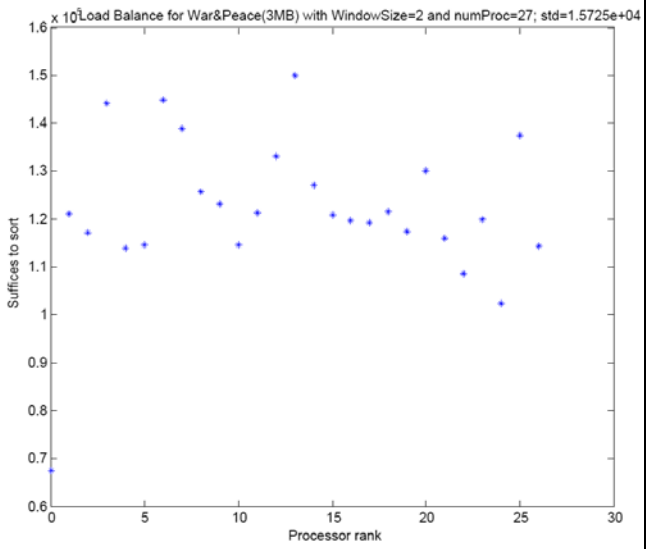
w=2

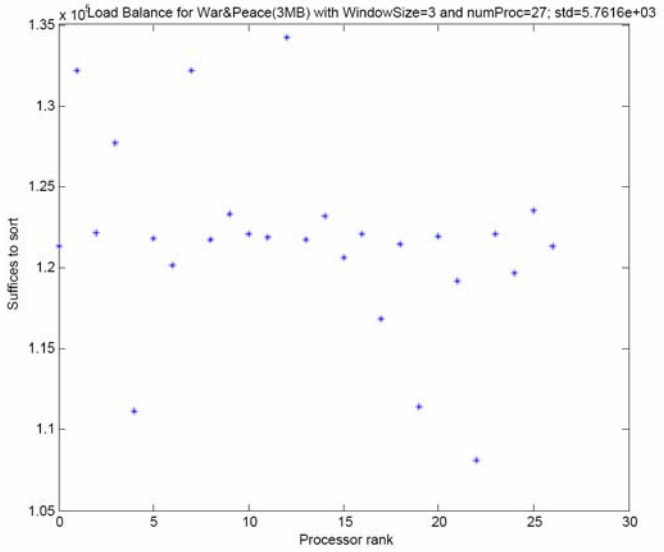


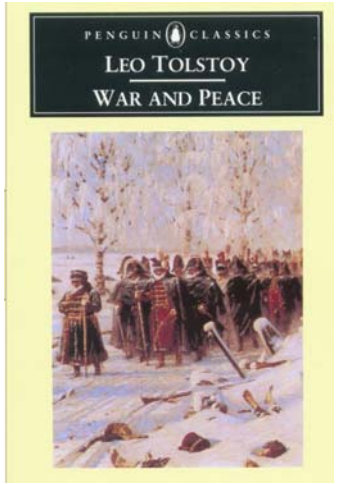
w=3











Results: Load Balancing

- Better balancing through heuristics in suffix distribution.

- Improve through pipelining
  - Distribute jobs as soon as ready;
  - Bigger jobs to earlier processors.

Versus the Competition

	Running Time (sec)	*Compression Rate*
bwtparallel		76%
np=27	5.95 (0.87+overhead)	
np=1	9.71 (4.65+overhead)	
bzip2	1.8	73%
gzip	0.82	63%
zip	0.88	63%