Cache-Oblivious Computations: Algorithms and Experimental Evaluation

Vijaya Ramachandran

Department of Computer Sciences

University of Texas at Austin

Dissertation work of former PhD student Dr. Rezaul Alam Chowdhury

Includes Honors Thesis results of

Mo Chen, Hai-Son , David Lan Roche, Lingling Tong

Massive Data Sets

Massive data sets often arise in practice:

- GIS data
- web graph
- in computational biology, etc.

To process these huge data sets we want:

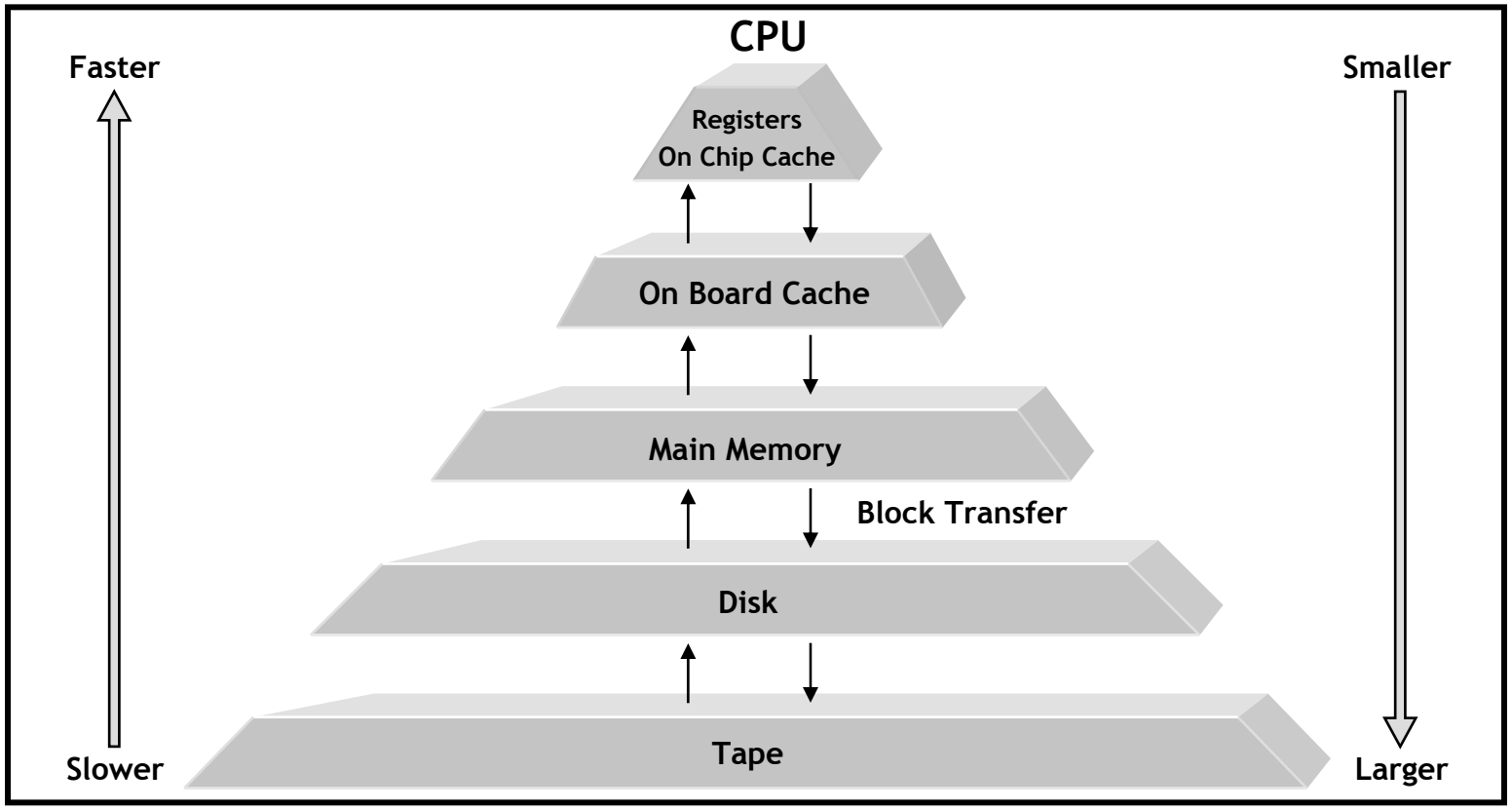
- cheap and large storage
- fast access to the data

But memory cannot be cheap, large and fast at the same time, because of

- finite signal speed
- lack of space to place connecting wires

A reasonable solution used in most current processors is a *memory hierarchy*.

The Memory Hierarchy



Large access latency at deeper levels

⇒ cost of data transfer often dominates the cost of computation

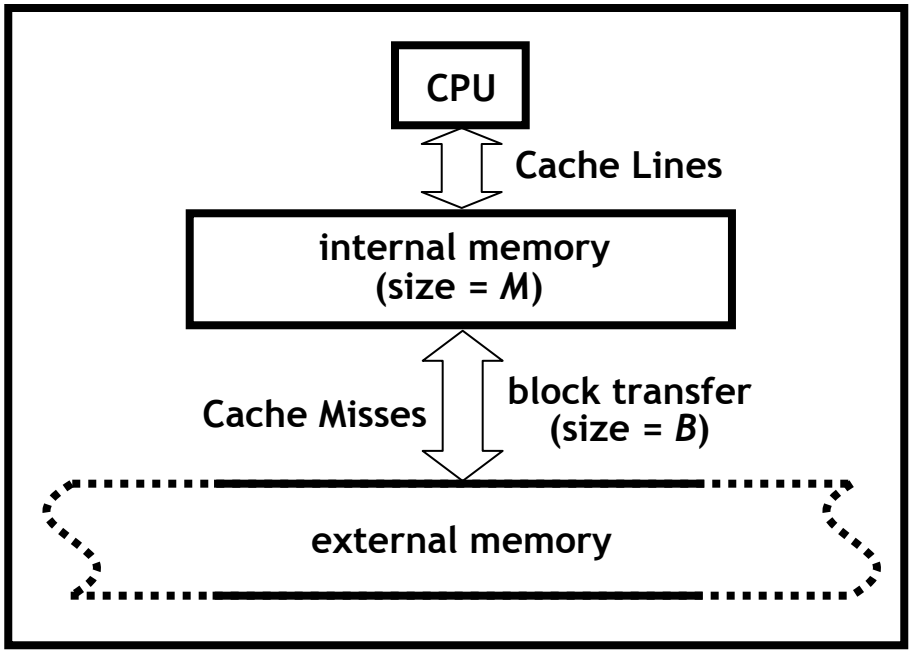
To achieve small amortized cost, data must be transferred in large blocks:
algorithms must have high locality in memory access patterns.

Analysis of Algorithms

Cost measures for algorithm analysis:

- **Traditional cost measure: *number of operations* (= running time)**
- **This talk: another cost measure motivated by the *memory hierarchy* of current computers: *I/O complexity***

The Two-level I/O (Cache-Aware) Model



The *two-level I/O model* [Aggarwal & Vitter, CACM'88] consists of:

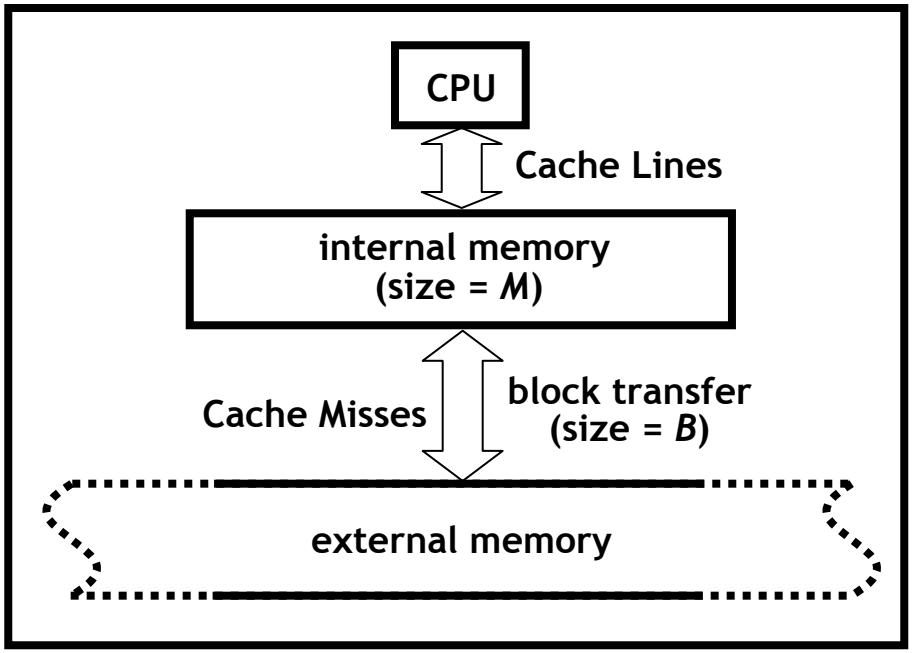
- an *internal memory* of size *M*
- an arbitrarily large *external memory* partitioned into blocks of size *B*.

I/O complexity of an algorithm = # blocks transferred between the two levels

Algorithms often crucially depend on the knowledge of *M* and *B*

⇒ algorithms do not adapt well when *M* or *B* changes

The Ideal-cache (Cache-oblivious) Model



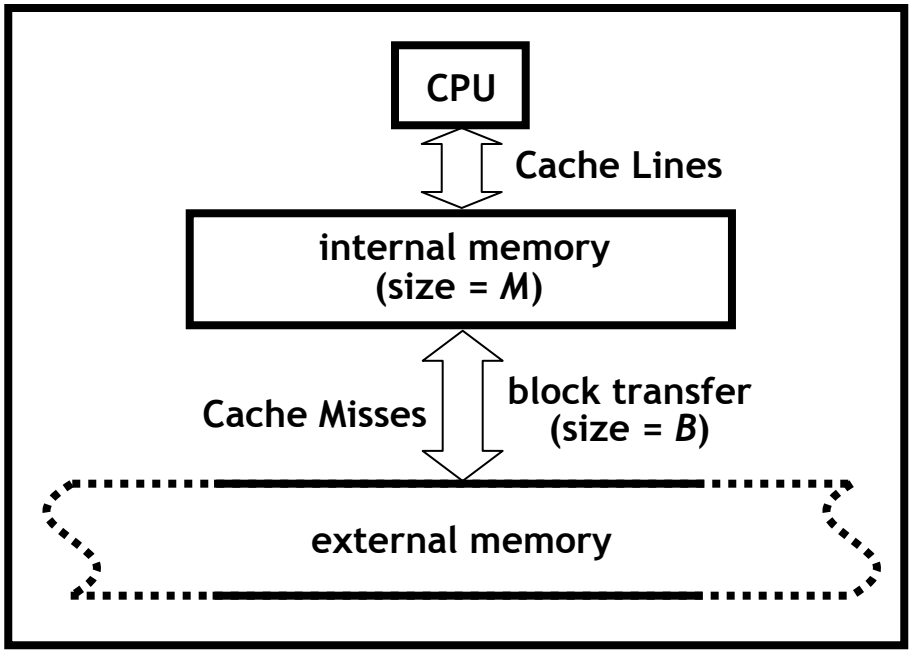
The *ideal-cache model* [Frigo et al., FOCS'99] is an extension of the I/O model with the following additional feature:

> algorithms must remain oblivious of the cache parameters *M* and *B*.

Consequences of this extension:

- algorithms can simultaneously adapt to all levels of a multi-level hierarchy
- algorithms become more flexible and portable

Basic I/O Complexities (Cache-Aware & Cache-Oblivious)



Consider N data items stored in N contiguous locations in external memory.

I/O complexity of *scanning* the data items: $scan(N) = \Theta\left(\frac{N}{B}\right)$

I/O complexity of *sorting* the data items: $sort(N) = \Theta\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$

Roadmap

- Preliminaries: Cache-oblivious model
- Cache-oblivious dynamic programming for string problems in bioinformatics
- Cache-oblivious priority queue (Buffer Heap) and Dijkstra's shortest path computation
- Cache-oblivious Gaussian Elimination Paradigm (GEP)
- Conclusion

Cache-oblivious Dynamic Programming for String Problems in Bioinformatics

(with Rezaul Chowdhury, Hai-Son Le)

The LCS Problem

A *subsequence* of a sequence X is obtained by deleting zero or more symbols from X.

Example: $X = abcba$

$Z = bca$ ← obtained by deleting the 1st '*a*' and the 2nd '*b*' from X

A *Longest Common Subsequence (LCS)* of two sequence X and Y is a sequence Z that is a subsequence of both X and Y, and is the longest among all such subsequences.

Given X and Y, the *LCS problem* asks for finding such a Z.

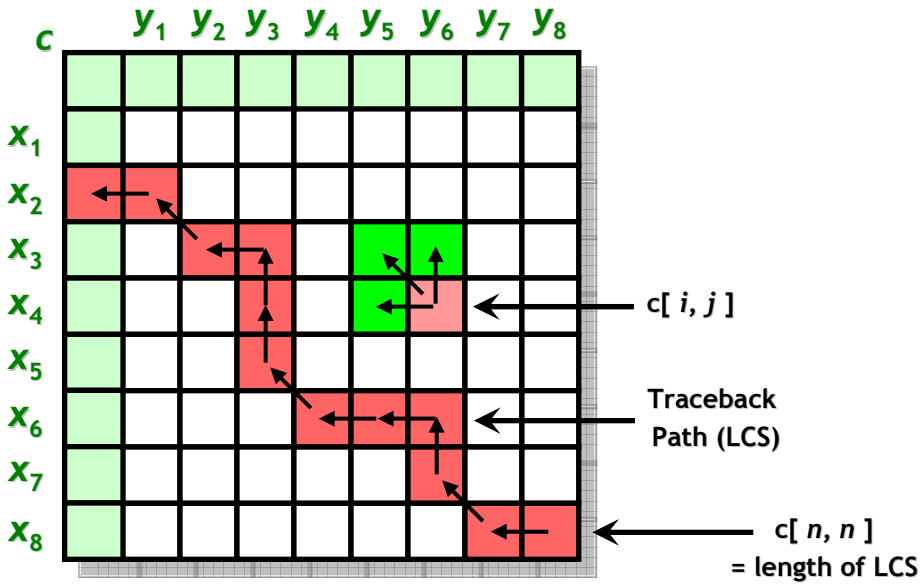
We will assume $|X| = |Y| = n = 2^q$, for some integer $q \geq 0$.

DP with Local Dependencies
The LCS Recurrence (Review)

Given: $X = x_1 x_2 \ldots x_n$ and $Y = y_1 y_2 \ldots y_n$

Fills up an array $c[0 \ldots n, 0 \ldots n]$ using the following recurrence.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \vee j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \wedge x_i = y_j, \\ \max\{c[i, j-1], c[i-1, j]\} & \text{otherwise.} \end{cases}$$



Local Dependency:
value of each cell depends only on values of adjacent cells.

I/O Complexities of Known Algorithms

The classic LCS DP runs in $\Theta(n^2)$ time, uses $\Theta(n^2)$ space, and incurs $\Theta\left(\frac{n^2}{B}\right)$ I/Os.

No sub-quadratic time algorithm is known for the general LCS problem.

Sub-quadratic space LCS algorithms are known.
(eg., Hirschberg's linear space LCS algorithm) } I/O-complexity remains $\Omega\left(\frac{n^2}{B}\right)$

Our Results

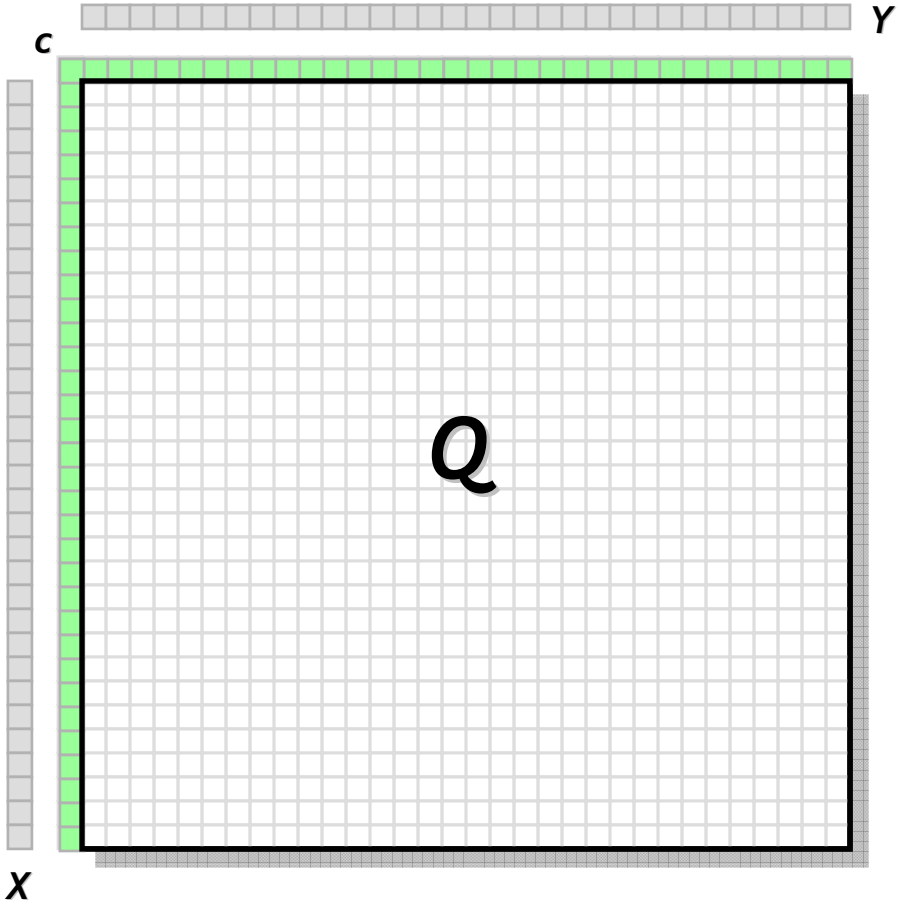
We present a new LCS algorithm which

- runs in $\Theta(n^2)$ time,
- uses *linear space*,
- is *cache-oblivious* incurring only $O\left(\frac{n^2}{BM}\right)$ cache misses,
- *computes an actual LCS*

Our Algorithm: Recursive LCS

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$



Our Algorithm: Recursive LCS

$Q \equiv c[1 \ldots n, 1 \ldots n]$

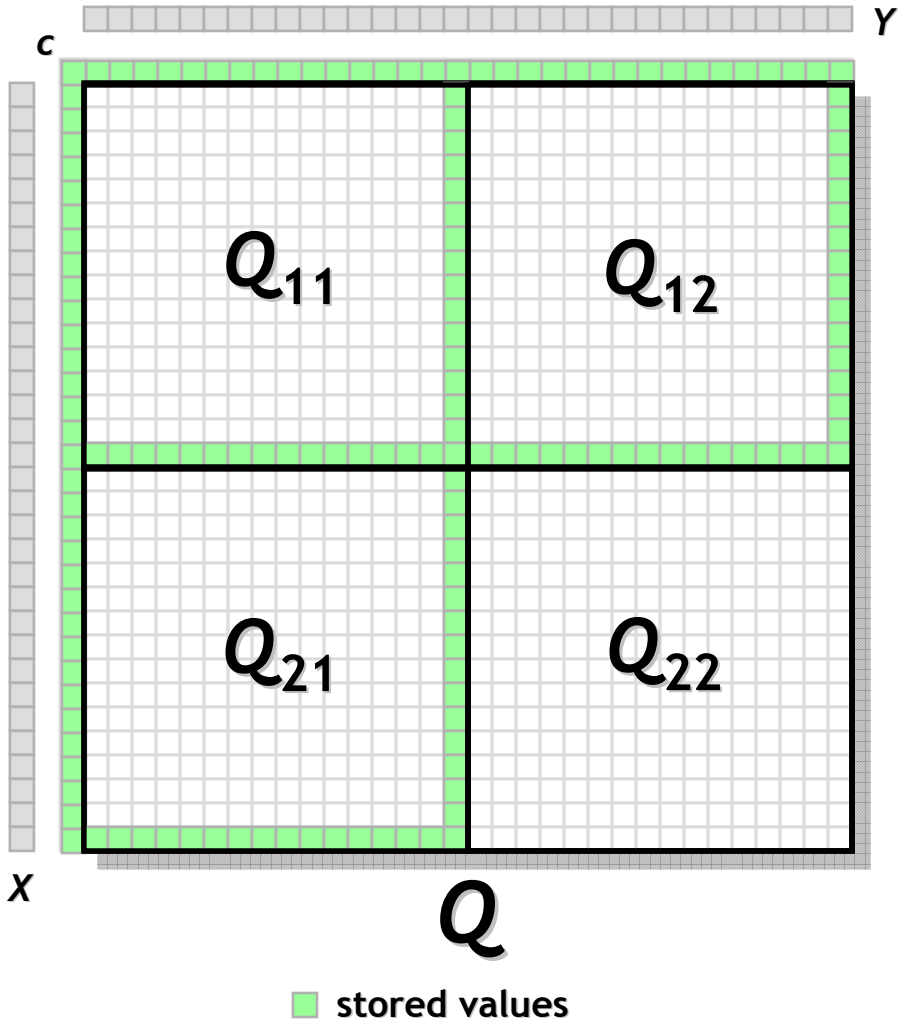
$n = 2^q$

1. Decompose Q:

   Split *Q* into four quadrants.

2. Forward Pass (Generate Boundaries):

   Generate the right and the bottom boundaries of the quadrants recursively.
   (of at most 3 quadrants)



Our Algorithm: Recursive LCS

$Q \equiv c[1 \ldots n, 1 \ldots n]$

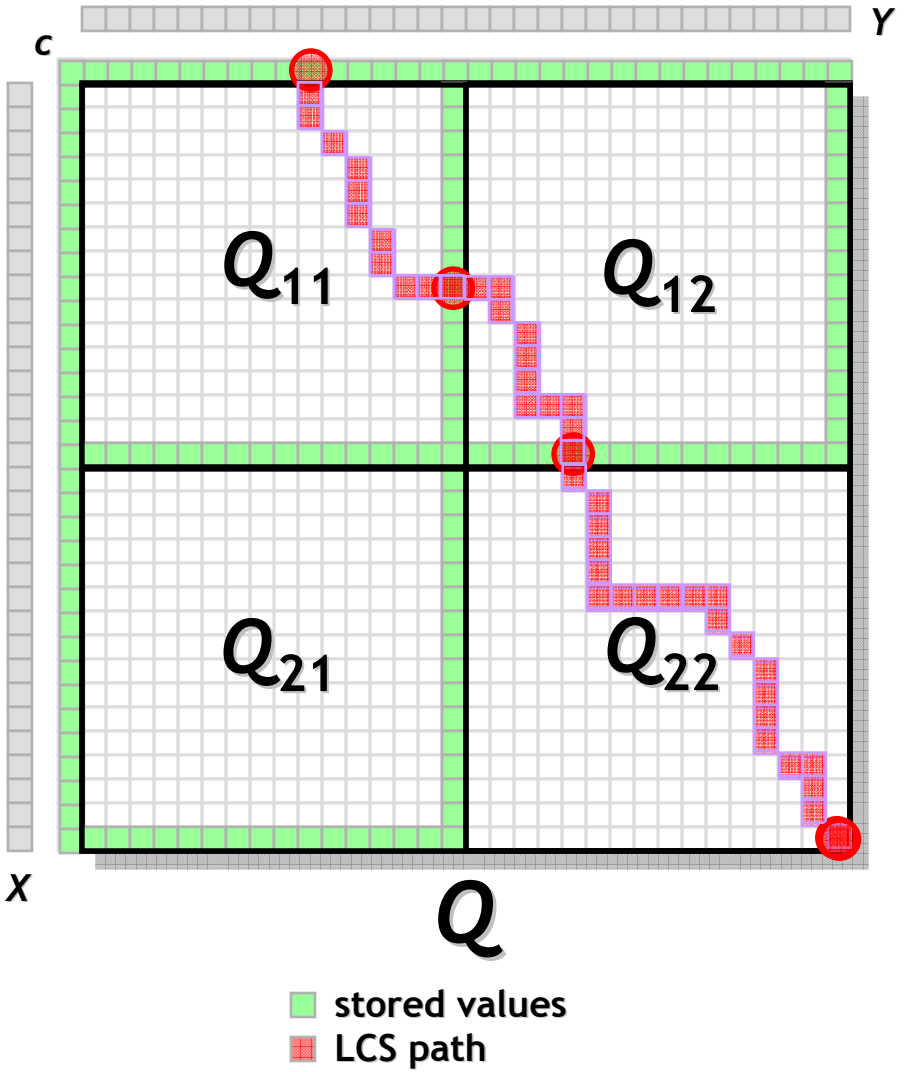
$n = 2^q$

1. Decompose Q:
   Split *Q* into four quadrants.

2. Forward Pass (Generate Boundaries):
   Generate the right and the bottom boundaries of the quadrants recursively.
   (of at most 3 quadrants)

3. Backward Pass (Extract LCS-Path Fragments):
   Extract LCS-Path fragments from the quadrants recursively.
   (from at most 3 quadrants)

4. Compose LCS-Path:
   Combine the LCS-Path fragments.



I/O Complexity

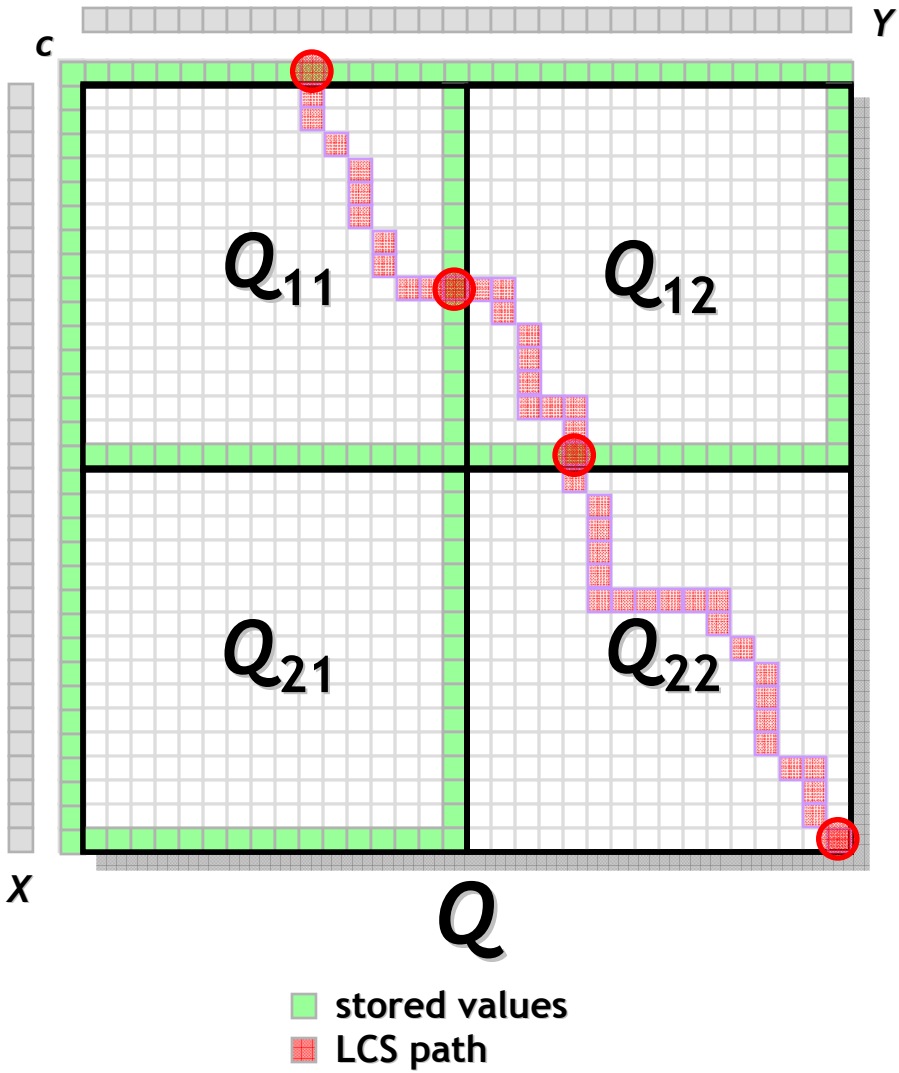
I/O-complexity of *Recursive LCS* :

$$I(n) \le \begin{cases} O\left(1+\frac{n}{B}\right), & \text{if } n \le \alpha M \\ 3I_f\left(\frac{n}{2}\right)+3I\left(\frac{n}{2}\right)+O(1), & \text{otherwise;} \end{cases}$$

where $I_f(n)$ is the I/O-complexity of *recursive boundary generation* (in the forward pass):

$$I_f(n) = O\left(\frac{n^2}{BM}\right)$$

Solving, $I(n) = O\left(\frac{n^2}{BM}\right)$ ← *optimal*



DP for String Problems in Bioinformatics

We consider DP problems for sequence alignment and RNA structure prediction:

- Pair-wise sequence alignment with affine gap costs
- Median: 3-way sequence alignment with affine gap costs
- RNA secondary structure prediction with simple pseudo-knots

We generalize the cache-oblivious algorithm for LCS to obtain good cache-oblivious algorithms for these problems.

Our Cache-Oblivious DP Results for Bioinformatics

(Chowdhury, Hai-Son Le, Ramachandran)

Problem	Time	Space	I/O Complexity	
Pairwise sequence Alignment	$\Theta(n^2)$	$\Theta(n)$	$O\left(\frac{n^2}{BM}\right)$	Gotoh, 1982 $O\left(\frac{n^2}{B}\right)$
Median of Three Sequences	$\Theta(n^3)$	$\Theta(n^2)$	$O\left(\frac{n^3}{B\sqrt{M}}\right)$	Knudsen, 2003 $O\left(\frac{n^3}{B}\right)$
RNA Secondary Structure Prediction with Pseudoknots	$\Theta(n^4)$	$\Theta(n^2)$	$O\left(\frac{n^4}{B\sqrt{M}}\right)$	Akutsu, 2000 $O\left(\frac{n^4}{B}\right)$

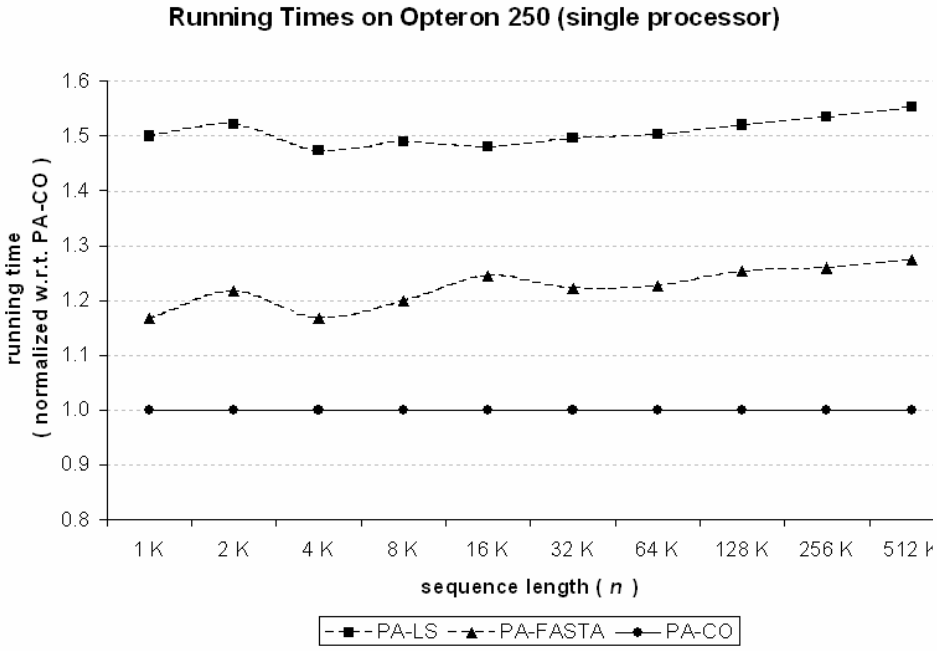
Experimental Results

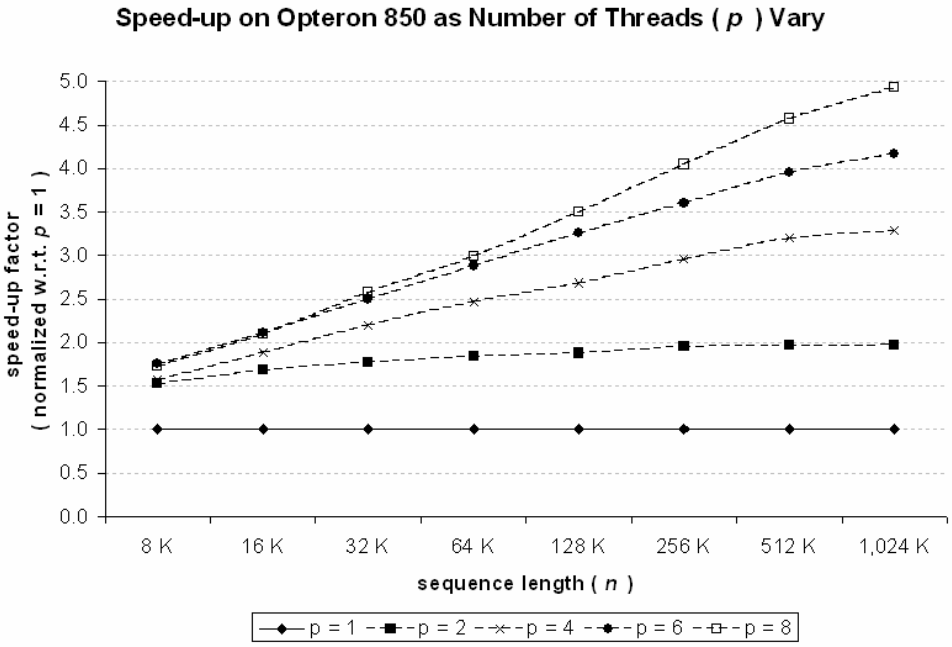
Pair-wise Sequence Alignment: Algorithms Compared

Algorithm	Comments	Time	Space	Cache Misses
PA-CO	Our cache-oblivious algorithm	$O(n^2)$	$O(n)$	$O\left(\frac{n^2}{BM}\right)$
PA-LS	Our implementation of linear-space variant of Gotoh's algorithm	$O(n^2)$	$O(n)$	$O\left(\frac{n^2}{B}\right)$
PA-FASTA	Linear-space variant of Gotoh's algorithm available in *fasta2* package	$O(n^2)$	$O(n)$	$O\left(\frac{n^2}{B}\right)$

Pair-wise Sequence Alignment: Random Sequences

Model	# Processors	Processor Speed	L1 Cache (*B*)	L2 Cache (*B*)	RAM
AMD Opteron 250	2	2.4 GHz	64 KB (64 B)	1 MB (64 B)	4 GB
AMD Opteron 850	8	2.2 GHz	64 KB (64 B)	1 MB (64 B)	32 GB





Experimental Results

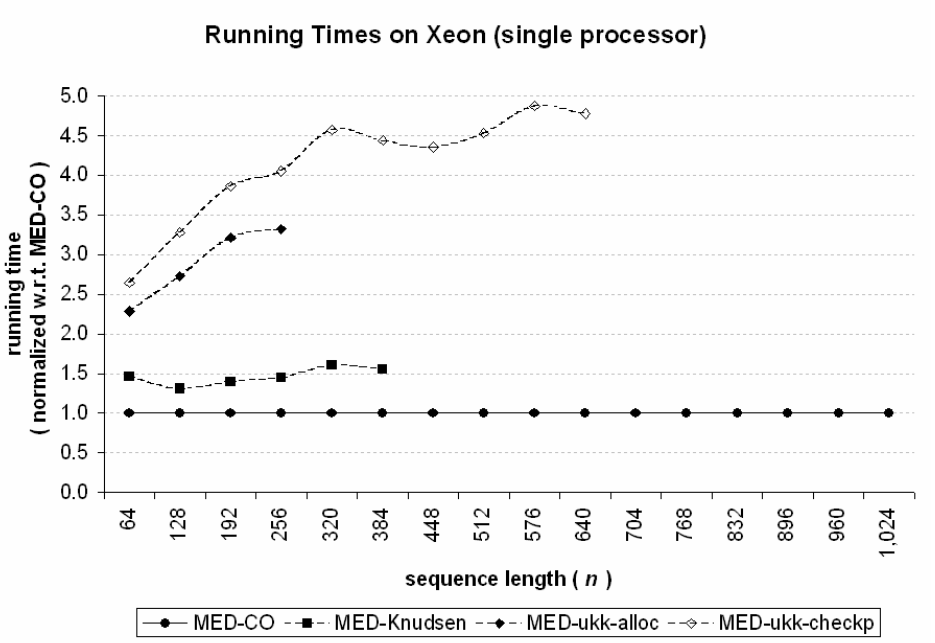
2. Median (Chowdhury, Hai-Son Le)

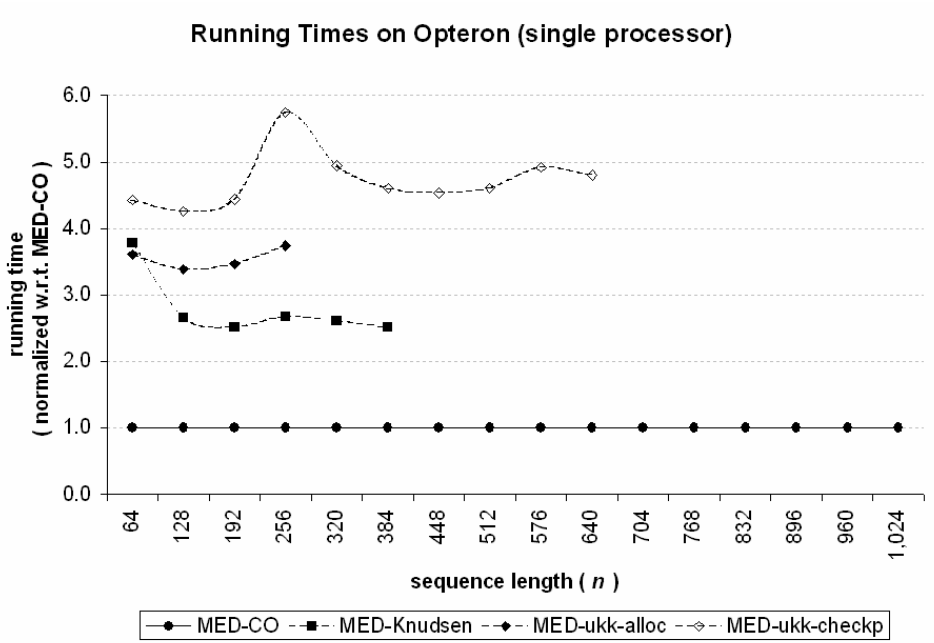
Median: Algorithms Compared

Algorithm	Comments	Time	Space	Cache Misses
MED-CO	Our cache-oblivious algorithm	$O(n^3)$	$O(n^2)$	$O\left(\frac{n^3}{B\sqrt{M}}\right)$
MED-Knudsen	Knudsen's implementation of his algorithm	$O(n^3)$	$O(n^3)$	$O\left(\frac{n^3}{B}\right)$
MED-ukk.alloc	Powell's implementation of an $O(d^3)$-space algorithm (d = 3-way edit dist)	$O(n + d^3)$	$O(n + d^3)$	$O\left(\frac{d^3}{B}\right)$
MED-ukk.checkp	Powell's implementation of an $O(d^2)$-space algorithm (d = 3-way edit dist)	$O(n \log d + d^3)$	$O(n + d^2)$	$O\left(\frac{d^3}{B}\right)$

Median: Random Sequences

Model	Processor Speed	L1 Cache (*B*)	L2 Cache (*B*)	RAM
Intel P4 Xeon	3.06 GHz	8 KB (64 B)	512 KB (64 B)	4 GB
AMD Opteron 250	2.4 GHz	64 KB (64 B)	1 MB (64 B)	4 GB





Median: `AWPM-19-like' Protein Sequences (LEA_10)

Model	# Processors	Processor Speed	L1 Cache (*B*)	L2 Cache (*B*)	RAM
AMD Opteron 850	8	2.2 GHz	64 KB (64 B)	1 MB (64 B)	32 GB

Triplet	Sequence Lengths	Alignment Cost	MED-ukk.checkp [1 proc]	MED-CO [1 proc]	MED-CO [8 procs]
1.	405 438 414	479	2,508 (12.54)	626 (3.13)	200 (1.00)
2.	405 522 546	506	2,707 (10.29)	950 (3.61)	263 (1.00)
3.	525 414 546	516	2,937 (12.09)	907 (3.73)	243 (1.00)
4.	513 504 438	542	3,543 (14.06)	961 (3.81)	252 (1.00)
5.	438 522 594	585	4,424 (12.75)	1,191 (3.43)	347 (1.00)

Cache-oblivious Buffer Heap and Dijkstra's SSSP Algorithm (Priority queue with decrease-keys)

(with Chowdhury, Lingling Tong, David Lan Roche, Mo Chen)

Cache-Oblivious Priority Queue with *Decrease-Key*

Our Result: Cache-Oblivious Buffer Heap

The following operations are supported:

- ***Delete-Min( )*:**

  Extracts an element with minimum key from queue.

- ***Decrease-Key*(x, k_x): (weak *Decrease-Key*)**

  If x already exists in the queue, replaces key k'_x of x with $\min(k_x, k'_x)$, otherwise inserts x with key k_x into the queue.

- ***Delete*(x):**

  Deletes the element x from the queue.

A new element x with key k_x can be inserted into queue by *Decrease-Key*(x, k_x).

Priority Queue with *Decrease-Key*

Supports the following operations:

- *Insert*(x, k_x):

  Inserts a new element x with key k_x to the queue.

- *Delete-Min*( ):

  Retrieves and deletes an element with minimum key from queue.

- *Decrease-Key*(x, k_x):

  Replaces key k'_x of x with $\min(k_x, k'_x)$.

- *Delete*(x):

  Deletes the element x from the queue if exists.

Cache-Oblivious Buffer Heap

Buffer Heap **is the first cache-oblivious priority queue supporting *Decrease-Keys*.**

		Amortized I/O Bounds	
	Priority Queue	***Delete-Min / Delete***	***Decrease-Key***
Cache-oblivious	Buffer Heap [C & R, SPAA'04] (independently [Brodal et al., SWAT'04])	$O\left(\frac{1}{B}\log_2 \frac{N}{M}\right)$	
Cache-aware	Tournament Tree [Kumar & Schwabe, SPDP'96]		
Internal Memory	Binary Heap (worst-case)	$O(\log_2 N)$	
	Fibonacci Heap [Fredman & Tarjan, JACM'87]	$O(\log_2 N)$	$O(1)$

Cache-Oblivious Buffer Heap: Structure

Consists of $r = 1 + \lceil \log_2 N \rceil$ levels, where N = total number of elements.

For $0 \le i \le r - 1$, level i contains two buffers:

- *element buffer* B_i
  contains elements of the form (x, k_x), where x is the element id, and k_x is its key
- *update buffer* U_i
  contains updates (*Delete*, *Decrease-Key* and *Sink*), each augmented with a time-stamp.

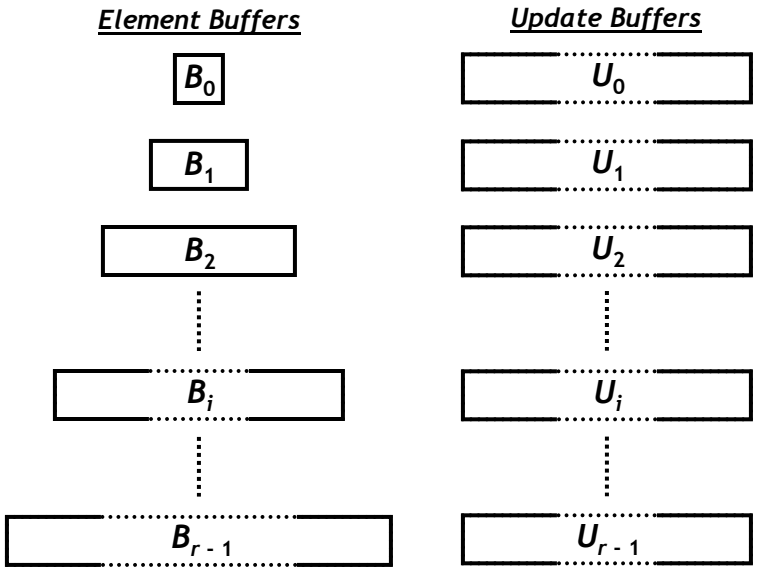


Fig: The Buffer Heap

Cache-Oblivious Buffer Heap: Invariants

Invariant 1: $|B_i| \leq 2^i$

Invariant 2:

(a) No key in B_i is larger than any key in B_{i+1}

(b) For each element x in B_i, all updates yet to be applied on x reside in $U_0, U_1, \ldots, U_i$

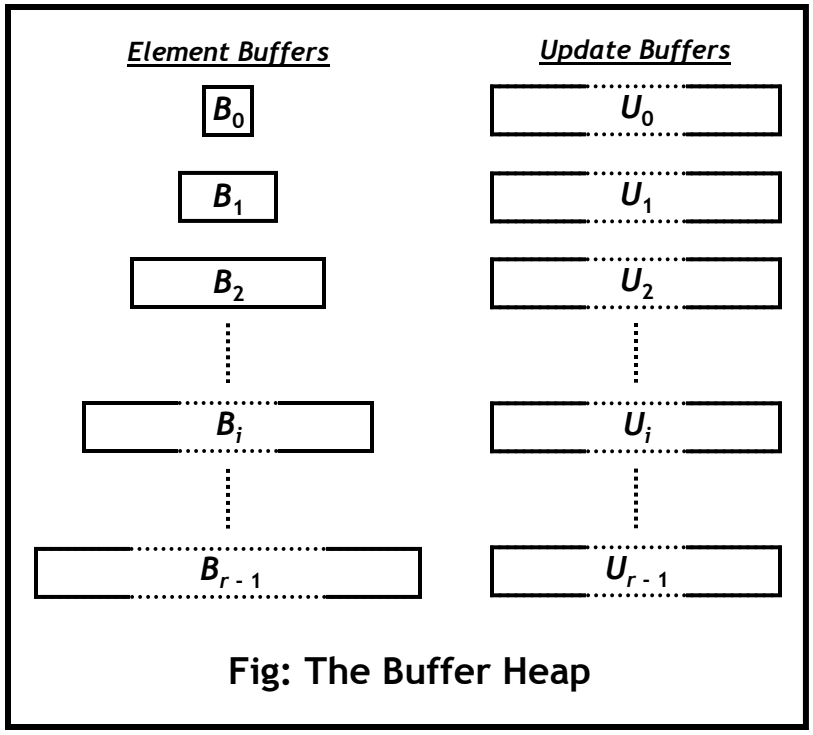


Fig: The Buffer Heap

Invariant 3:

(a) Each B_i is kept sorted by element id

(b) Each U_i (except U_0) is kept (coarsely) sorted by element id and time-stamp

Cache-Oblivious Buffer Heap: Operations

The following operations are supported:

- *Delete-Min( )*:

  Extracts an element with minimum key from queue.

- *Decrease-Key*(x, k_x): (weak *Decrease-Key*)

  If x already exists in the queue, replaces key k'_x of x with $\min(k_x, k'_x)$, otherwise inserts x with key k_x into the queue.

- *Delete*(x):

  Deletes the element x from the queue.

A new element x with key k_x can be inserted into queue by *Decrease-Key*(x, k_x).

Cache-Oblivious Buffer Heap: Operations

Decrease-Key(x, k_x) :

Insert the operation into U_0 augmented with current time-stamp.

Delete(x) :

Insert the operation into U_0 augmented with current time-stamp.

Delete-Min( ) :

Two phases:

- Descending Phase (Apply Updates)
- Ascending Phase (Redistribute Elements)

Cache-Oblivious Buffer Heap: I/O Complexity

Potential Function: $\Phi(H) = \frac{1}{B}\left(3r|U_0| + \sum_{i=1}^{r-1}(2r-i)|U_i| + \sum_{i=0}^{r-1}(i+1)|B_i| \right)$

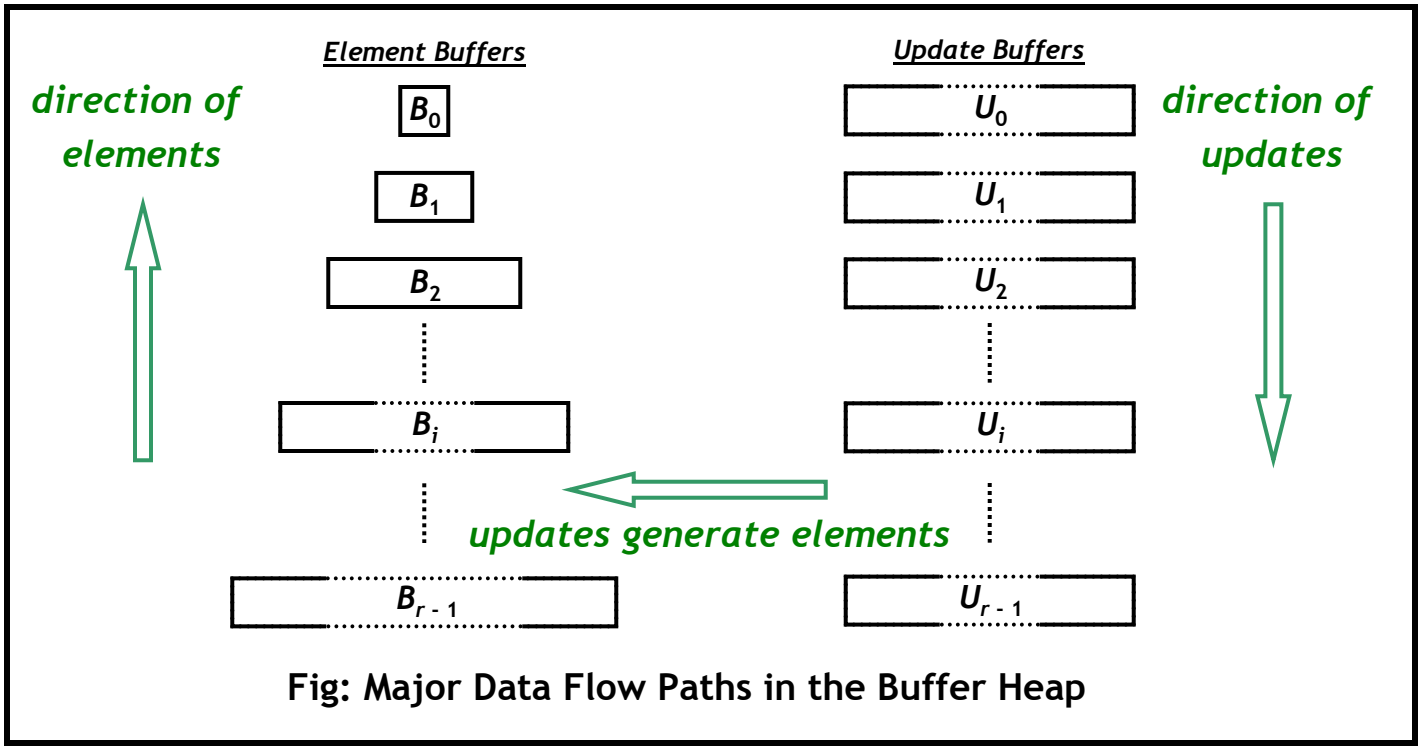


Fig: Major Data Flow Paths in the Buffer Heap

Lemma: A *Buffer Heap* on N elements supports *Delete*, *Delete-Min* and *Decrease-Key* operations cache-obliviously in $O\left(\frac{1}{B}\log_2 N\right)$ amortized I/Os each using $O(N)$ space.

Buffer Heap Summary

- Amortized I/Os per operation: $O\left(\frac{1}{B}\log_2 N\right)$

- Buffer heap achieves improved I/O bound while maintaining the traditional $O(\log N)$ running time (amortized)

- Since the top $\log_2 M$ levels of the buffer heap always resides in internal-memory, the amortized I/Os per operation reduces to

$$O\left(\frac{1}{B}\left(\log_2 N - \log_2 M\right)\right) = O\left(\frac{1}{B}\log_2 \frac{N}{M}\right)$$

Experimental Results

(Rezaul Chowdhury, Lingling Tong, also David Lan Roche)

Priority Queue Operations: Out-of-core using STXXL

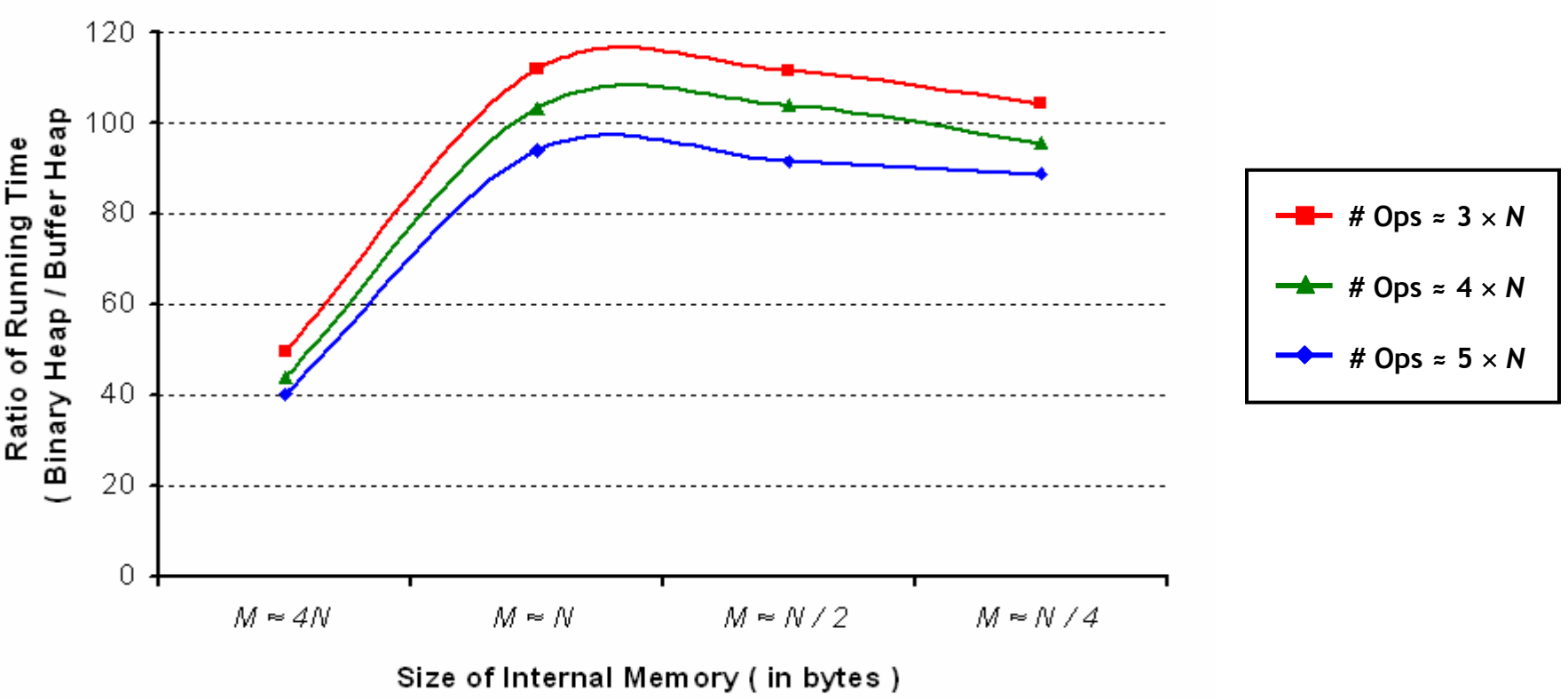
Processor	Speed	Local Hard Disk
Intel Xeon	3 GHz	73 GB, 10K RPM, ~5ms avg. seek time, 107 MB/s max xfer rate

Ratios of Running Times of Binary Heap to Buffer Heap

(Lingling Tong & David Lan Roche)

***N* = 2 million = # *Delete-Min*, *B* = 64 KB**

***M* varies**



SSSP (Dijkstra's Algorithm)

Graph Type	Cache-Aware Results	Cache-Oblivious Results
Undirected	$O\left(V + \frac{E}{B}\log_2\frac{V}{B}\right)$ (Kumar & Schwabe, SPDP'96)	$O\left(V + \frac{E}{B}\log_2\frac{V}{M}\right)$ (new, C & R, SPAA'04)
	$O\left(V + \frac{VE}{BM} + sort(E)\right)$ (Chiang et al., SODA'95)	
	$O\left(\sqrt{\frac{VE}{B}\log_2\rho} + sort(V+E)\log_2\log_2\frac{VB}{E}\right)$ (Meyer & Zeh, ESA'03)	
	$O\left(V + \frac{E}{B}\log_2\frac{V}{M}\right)$ (new)	
Directed	$O\left(\left(V + \frac{E}{B}\right)\cdot\log_2\frac{V}{B}\right)$ (Kumar & Schwabe, SPDP'96)	$O\left(\left(V + \frac{E}{B}\right)\cdot\log_2\frac{V}{B}\right)$ (new, C & R, SPAA'04)
	$O\left(V + \frac{VE}{BM}\log_2\frac{V}{B}\right)$ (Chiang et al., SODA'95)	

I/O bounds for a graph with V nodes, E edges and non-negative edge-weights.

- Bound for best traditional algorithm is $O(V \log V + E)$
- Our results give good performance for moderately dense graphs.

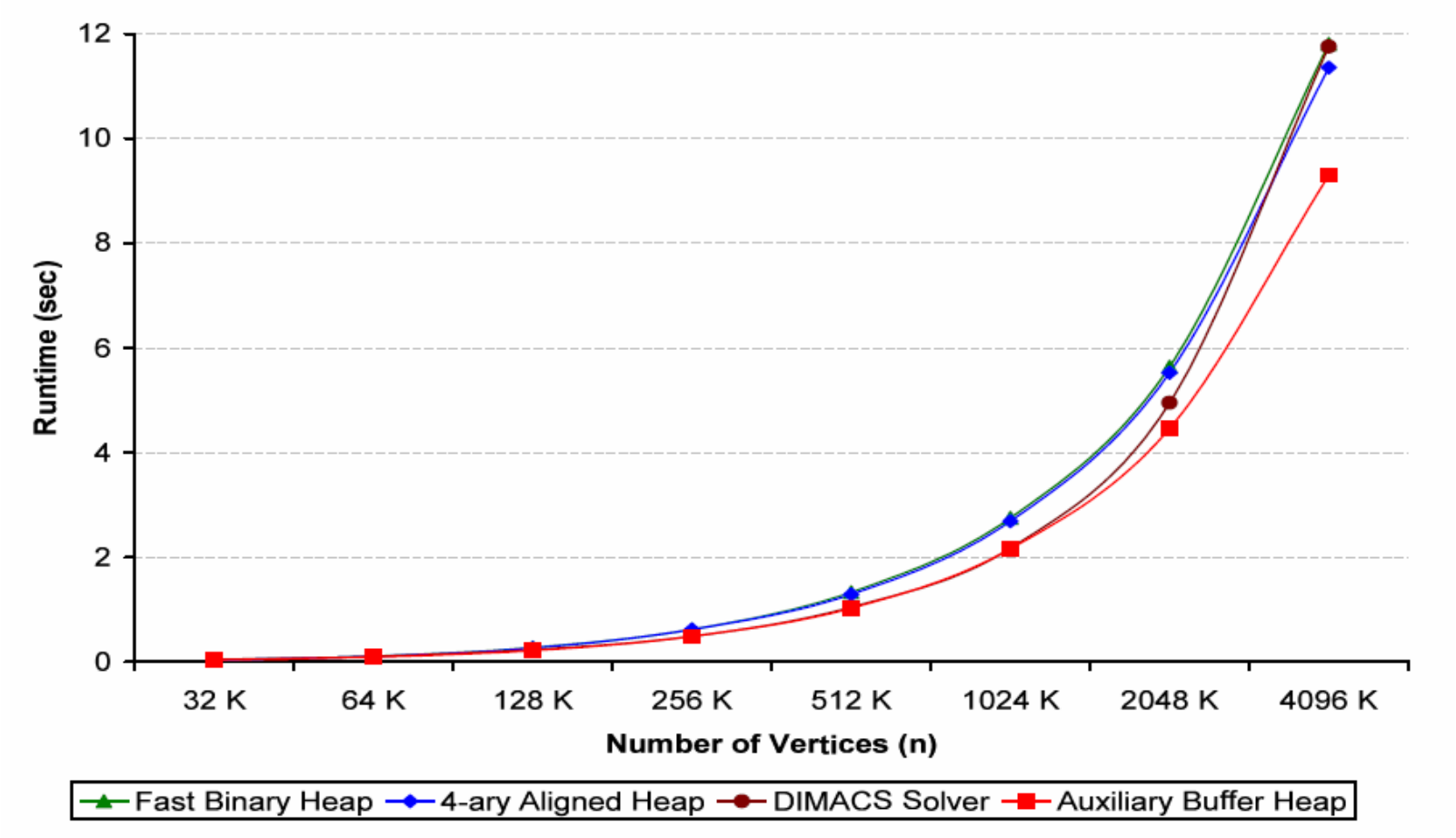
Experimental Results

(Chowdhury, David Lan Roche, also Mo Chen)

SSSP: In-core Running Times

Architecture	Processor Speed	L1 Cache (*B*)	L2 Cache (*B*)	RAM
Intel P4 Xeon	3.06 GHz	8 KB (64 B)	512 KB (64 B)	4 GB

SSSP on $G_{n,m}$ with $n = 8m$ and Random Integer Edge-Weights



Summary

- Presented efficient cache-oblivious algorithms from three different problem domains
- Simple portable code with very good performance
- Simple and effective parallelism (for DP and I-GEP/C-GEP)

- Current trends in computer architecture and in massive datasets would indicate that cache-efficient algorithms will become increasingly important in the future

The Cache - Oblivious Gaussian Elimination Paradigm (GEP)

(Chowdhury & Ramachandran [SODA'06, SPAA'07])

Gaussian Elimination Paradigm: Triply-nested Loops

Gaussian Elimination without Pivoting

1. *for* $k \leftarrow 1$ *to* $n - 2$ *do*
2. *for* $i \leftarrow k + 1$ *to* $n - 1$ *do*
3. *for* $j \leftarrow k + 1$ *to* n *do*
4. $$c[i, j] \leftarrow c[i, j] - \frac{c[i, k]}{c[k, k]} \times c[k, j]$$

Floyd-Warshall's All-Pairs Shortest Path

1. *for* $k \leftarrow 1$ *to* n *do*
2. *for* $i \leftarrow 1$ *to* n *do*
3. *for* $j \leftarrow 1$ *to* n *do*
4. $$c[i, j] \leftarrow \min(c[i, j],\ c[i, k] + c[k, j])$$

The Gaussian Elimination Paradigm (GEP)

- $c[\,1 \ldots n,\ 1 \ldots n\,]$ is an $n \times n$ matrix with entries chosen from an arbitrary set S
- $f : S \times S \times S \times S \to S$ is an arbitrary function
- $\langle i, j, k \rangle$ is an *update* of the form:

$$c[\,i, j\,] \leftarrow f\,(\,c[\,i, j\,], c[\,i, k\,], c[\,k, j\,], c[\,k, k\,]\,)$$

- Σ_G is an arbitrary set of updates

GEP Computation

```
Algorithm G( c, n, f, Σ_G )
1. for k ← 1 to n do
2.     for i ← 1 to n do
3.         for j ← 1 to n do
4.             if ⟨i, j, k⟩ ∈ Σ_G then
5.                 c[ i, j ] ← f(c[ i, j ], c[ i, k ], c[ k, j ], c[ k, k ])
```

Gaussian Elimination Paradigm (GEP): Time and I/O Bounds

GEP Computation

```
Algorithm G( c, n, f, Σ_G )
1. for k ← 1 to n do
2.     for i ← 1 to n do
3.         for j ← 1 to n do
4.             if ⟨i, j, k⟩ ∈ Σ_G then
5.                 c[ i, j ] ← f( c[ i, j ], c[ i, k ], c[ k, j ], c[ k, k ] )
```

Assumption: The following can be performed in $O(1)$ time with no cache misses -

- testing $\langle i, j, k \rangle \in \Sigma_G$ in line 4
- evaluating $f(\cdot, \cdot, \cdot, \cdot)$ in line 5

Running Time: $\Theta\left(n^3\right)$

I/O Complexity: $\mathrm{O}\left(\frac{n^3}{B}\right)$

I-GEP

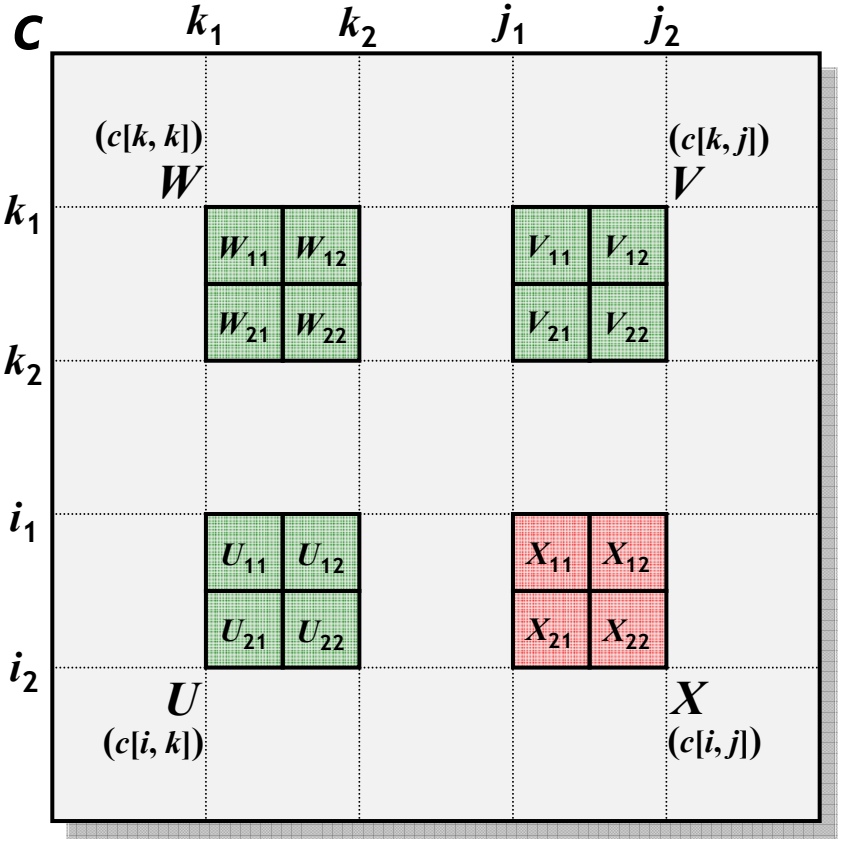
We present a recursive algorithm called *I-GEP* which

- solves *GEP* for several important special cases of f and $\sum_G$
  - Gaussian elimination / LU decomposition w/o pivoting
  - path computation over closed semirings (including Floyd-Warshall)
  - matrix multiplication
- runs in $\Theta(n^3)$ time,
- is *in-place*,
- is *cache-oblivious* incurring only $O\left(\frac{n^3}{B\sqrt{M}}\right)$ cache misses.

I-GEP

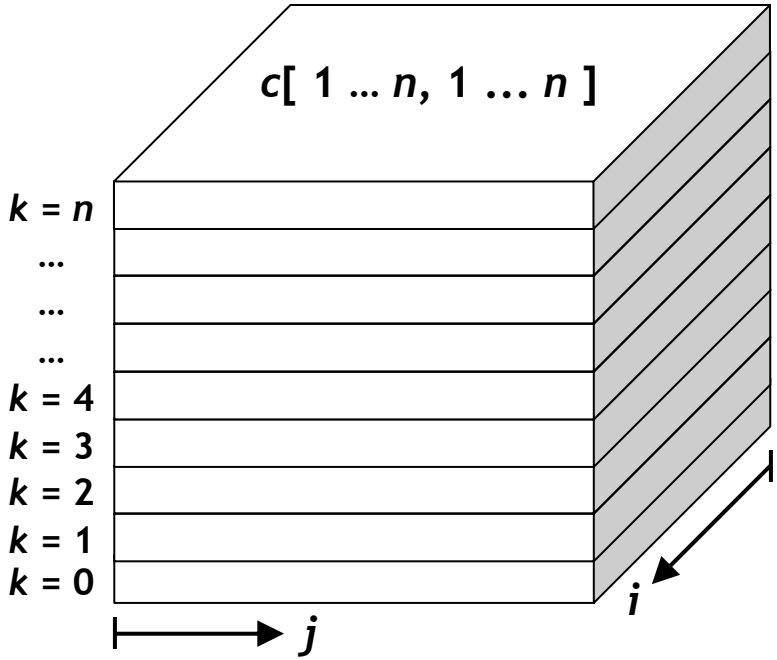
Algorithm $F(X, U, V, W)$ { initial call: $F(c, c, c, c)$ }

1. *if* $T_{XUV} \cap \Sigma_G = \varnothing$ *then return* { T_{XUV} = { updates on X using $(i, k) \in U$ and $(k, j) \in V$ } }
2. *if* $X = 1 \times 1$ matrix *then* $X \leftarrow f(X, U, V, W)$
3. *else*
4. $F(X_{11}, U_{11}, V_{11}, W_{11})$; $F(X_{12}, U_{11}, V_{12}, W_{11})$; $F(X_{21}, U_{21}, V_{11}, W_{11})$; $F(X_{22}, U_{21}, V_{12}, W_{11})$
5. $F(X_{22}, U_{22}, V_{22}, W_{22})$; $F(X_{21}, U_{22}, V_{21}, W_{22})$; $F(X_{12}, U_{12}, V_{22}, W_{22})$; $F(X_{11}, U_{12}, V_{21}, W_{22})$



GEP

```
Algorithm G( c, n, f, Σ_G )
1.  for k ← 1 to n do
2.      for i ← 1 to n do
3.          for j ← 1 to n do
4.              if ⟨i, j, k⟩ ∈ Σ_G then
5.                  c[ i, j ] ← f( c[ i, j ], c[ i, k ], c[ k, j ], c[ k, k ] )
```

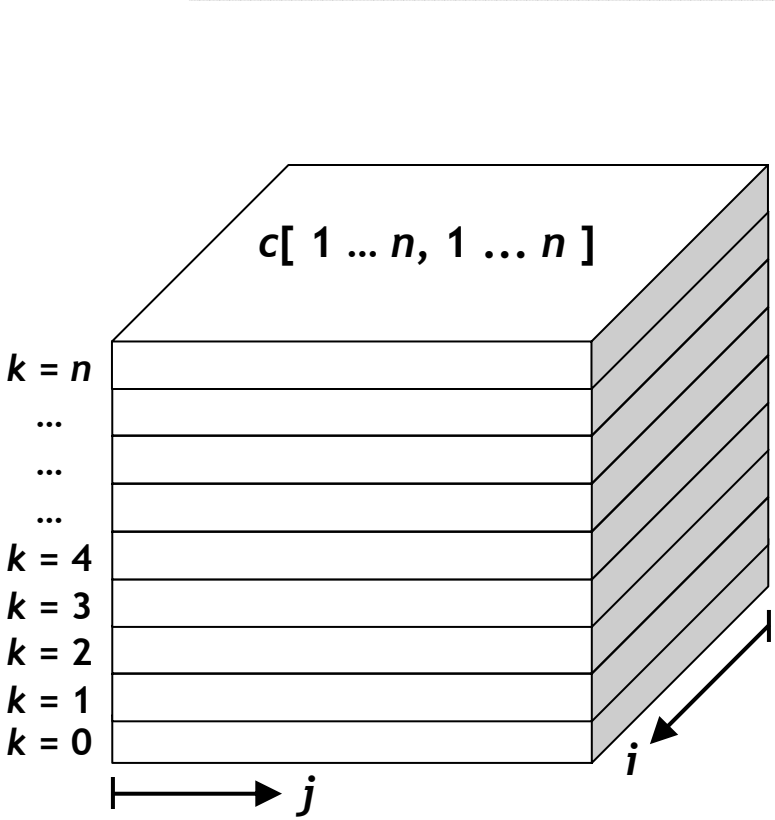


GEP Execution

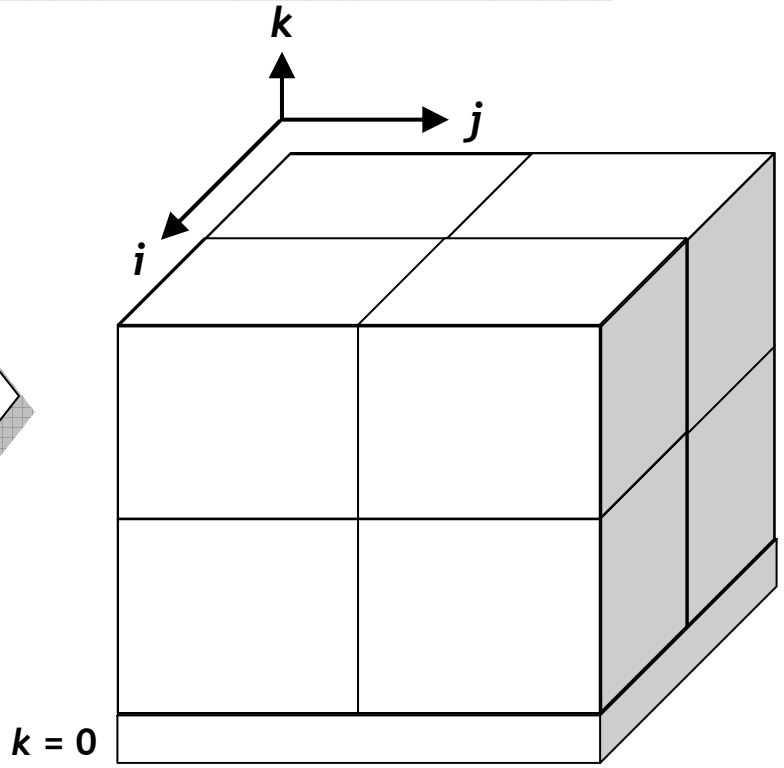
I-GEP

Algorithm $F(X, U, V, W)$ { initial call: $F(c, c, c, c)$ }

1. *if* $T_{XUV} \cap \Sigma_G = \varnothing$ *then return* { T_{XUV} = { updates on X using $(i, k) \in U$ and $(k, j) \in V$ } }
2. *if* $X = 1 \times 1$ matrix *then* $X \leftarrow f(X, U, V, W)$
3. *else*
4. $F(X_{11}, U_{11}, V_{11}, W_{11})$; $F(X_{12}, U_{11}, V_{12}, W_{11})$; $F(X_{21}, U_{21}, V_{11}, W_{11})$; $F(X_{22}, U_{21}, V_{12}, W_{11})$
5. $F(X_{22}, U_{22}, V_{22}, W_{22})$; $F(X_{21}, U_{22}, V_{21}, W_{22})$; $F(X_{12}, U_{12}, V_{22}, W_{22})$; $F(X_{11}, U_{12}, V_{21}, W_{22})$



GEP Execution



I-GEP Execution

I-GEP: I/O Complexity

Algorithm $F(X, U, V, W)$ { initial call: $F(c, c, c, c)$ }

1. *if* $T_{XUV} \cap \Sigma_G = \varnothing$ *then return*
2. *if* $X = 1 \times 1$ matrix *then* $X \leftarrow f(X, U, V, W)$
3. *else*
4. $F(X_{11}, U_{11}, V_{11}, W_{11})$; $F(X_{12}, U_{11}, V_{12}, W_{11})$; $F(X_{21}, U_{21}, V_{11}, W_{11})$; $F(X_{22}, U_{21}, V_{12}, W_{11})$
5. $F(X_{22}, U_{22}, V_{22}, W_{22})$; $F(X_{21}, U_{22}, V_{21}, W_{22})$; $F(X_{12}, U_{12}, V_{22}, W_{22})$; $F(X_{11}, U_{12}, V_{21}, W_{22})$

Number of I/O operations performed by *F* on submatrices of size $n \times n$ each:

$$I(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right), & \text{if } n^2 \le \alpha M \\ 8I\left(\frac{n}{2}\right) + O(1), & \text{otherwise.} \end{cases}$$

Solving, $I(n) = O\left(\frac{n^3}{B\sqrt{M}}\right)$ **(assuming a *tall cache*, i.e.,** $M = \Omega(B^2)$ **)**

Optimal for general *GEP*

I-GEP vs Other Methods

Cache-oblivious algorithms for several problems solved by I-GEP (except the *gap problem*) have already been obtained by different sets of authors.

- Matrix multiplication [Frigo et al., FOCS'99]
- LU decomposition w/o pivoting [Blumofe et al., SPAA'96; Toledo, 1999]
- Floyd-Warshall's APSP [Park et al., 2005]
- Simple DP [Cherng & Ladner, 2005]

But *I-GEP*

- gives cache-oblivious algorithms for all of these problems,
- matches the I/O bound of the best known solution for the problem,
- can be implemented as a compile-time optimization.

I-GEP and C-GEP

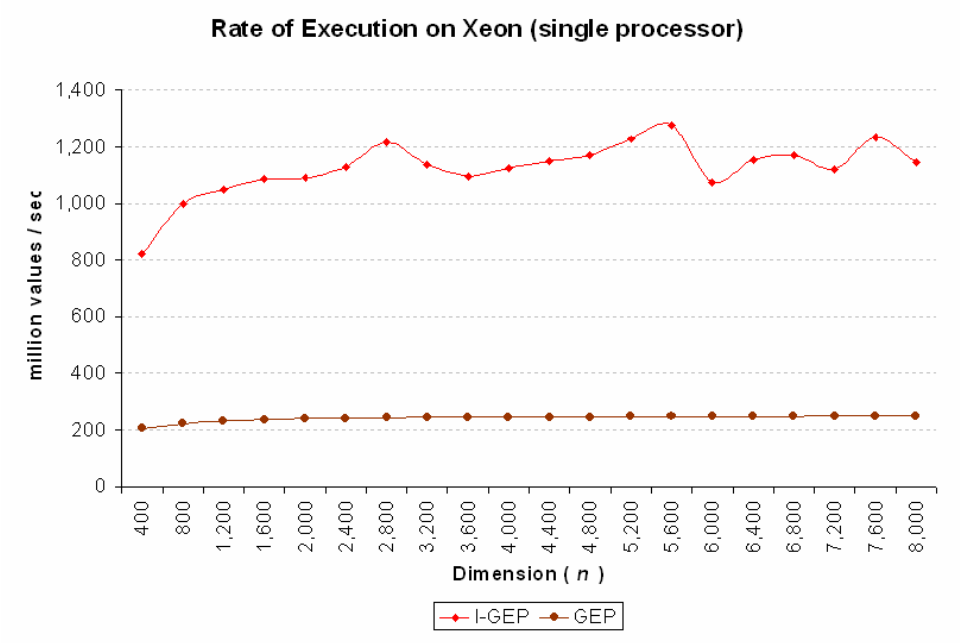
Problem	Time	Space	I/O Complexity
I-GEP **(solves most important special cases of GEP)** - Gaussian elimination / LU decomposition w/o pivoting - Floyd-Warshall's APSP, transitive closure - matrix multiplication [C & R, SODA'06]	$\Theta(n^3)$	n^2 (in-place)	$O\left(\frac{n^3}{B\sqrt{M}}\right)$ **traditional** $O\left(\frac{n^3}{B}\right)$
C-GEP **(solves GEP in its full generality)** [C & R, SPAA'07]		n^2 ($n^2 + n$ extra space)	

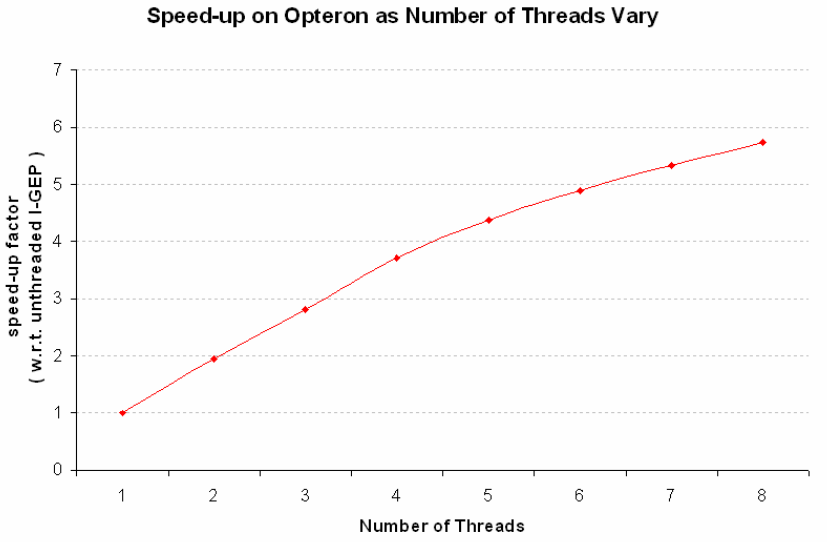
Experimental Results

1. Floyd-Warshall All-Pairs Shortest Paths in Graphs

Floyd-Warshall's APSP

Model	# Processors	Processor Speed	L1 Cache (*B*)	L2 Cache (*B*)	RAM
Intel P4 Xeon	2	3.06 GHz	8 KB (64 B)	512 KB (64 B)	4 GB
AMD Opteron 850	8	2.2 GHz	64 KB (64 B)	1 MB (64 B)	4 GB





Additional Slides

Cache-Oblivious Buffer Heap: Structure

Consists of $r = 1 + \lceil \log_2 N \rceil$ levels, where N = total number of elements.

For $0 \le i \le r - 1$, level i contains two buffers:

- *element buffer* B_i
  contains elements of the form (x, k_x), where x is the element id, and k_x is its key
- *update buffer* U_i
  contains updates (*Delete*, *Decrease-Key* and *Sink*), each augmented with a time-stamp.

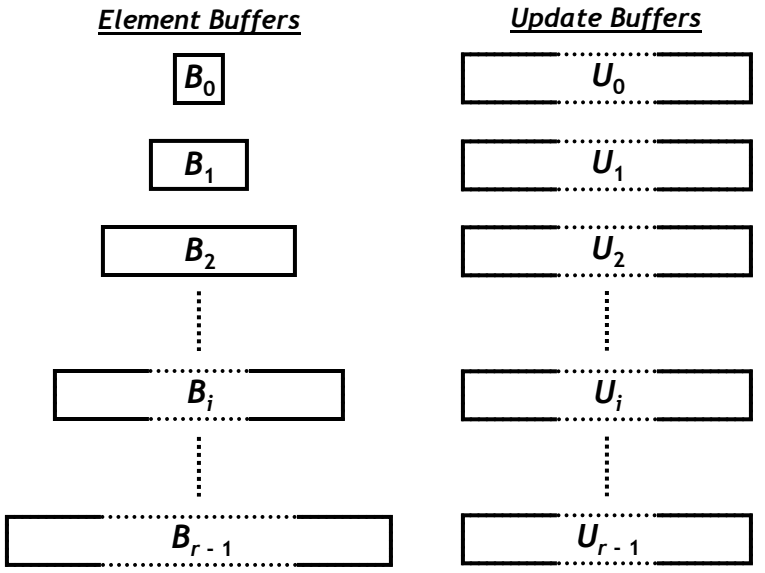


Fig: The Buffer Heap

Cache-Oblivious Buffer Heap: Invariants

Invariant 1: $|B_i| \leq 2^i$

Invariant 2:

(a) No key in B_i is larger than any key in B_{i+1}

(b) For each element x in B_i, all updates yet to be applied on x reside in $U_0, U_1, \ldots, U_i$

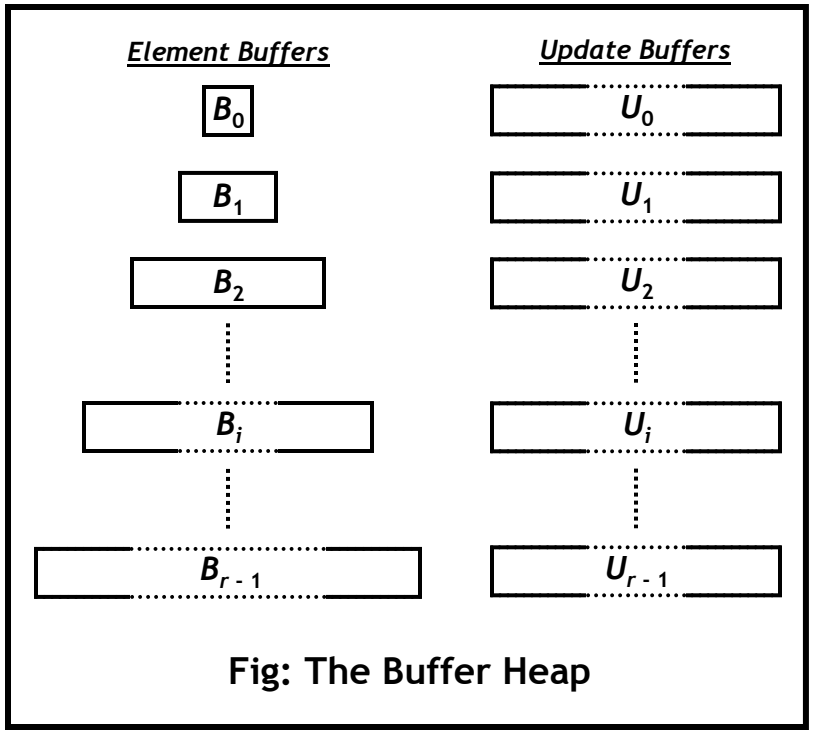


Fig: The Buffer Heap

Invariant 3:

(a) Each B_i is kept sorted by element id

(b) Each U_i (except U_0) is kept (coarsely) sorted by element id and time-stamp

Cache-Oblivious Buffer Heap: Operations

The following operations are supported:

- *Delete-Min( )*:

  Extracts an element with minimum key from queue.

- *Decrease-Key*(x, k_x): (weak *Decrease-Key*)

  If x already exists in the queue, replaces key k'_x of x with $\min(k_x, k'_x)$, otherwise inserts x with key k_x into the queue.

- *Delete*(x):

  Deletes the element x from the queue.

A new element x with key k_x can be inserted into queue by *Decrease-Key*(x, k_x).

Cache-Oblivious Buffer Heap: Operations

Decrease-Key(x, k_x) :

Insert the operation into U_0 augmented with current time-stamp.

Delete(x) :

Insert the operation into U_0 augmented with current time-stamp.

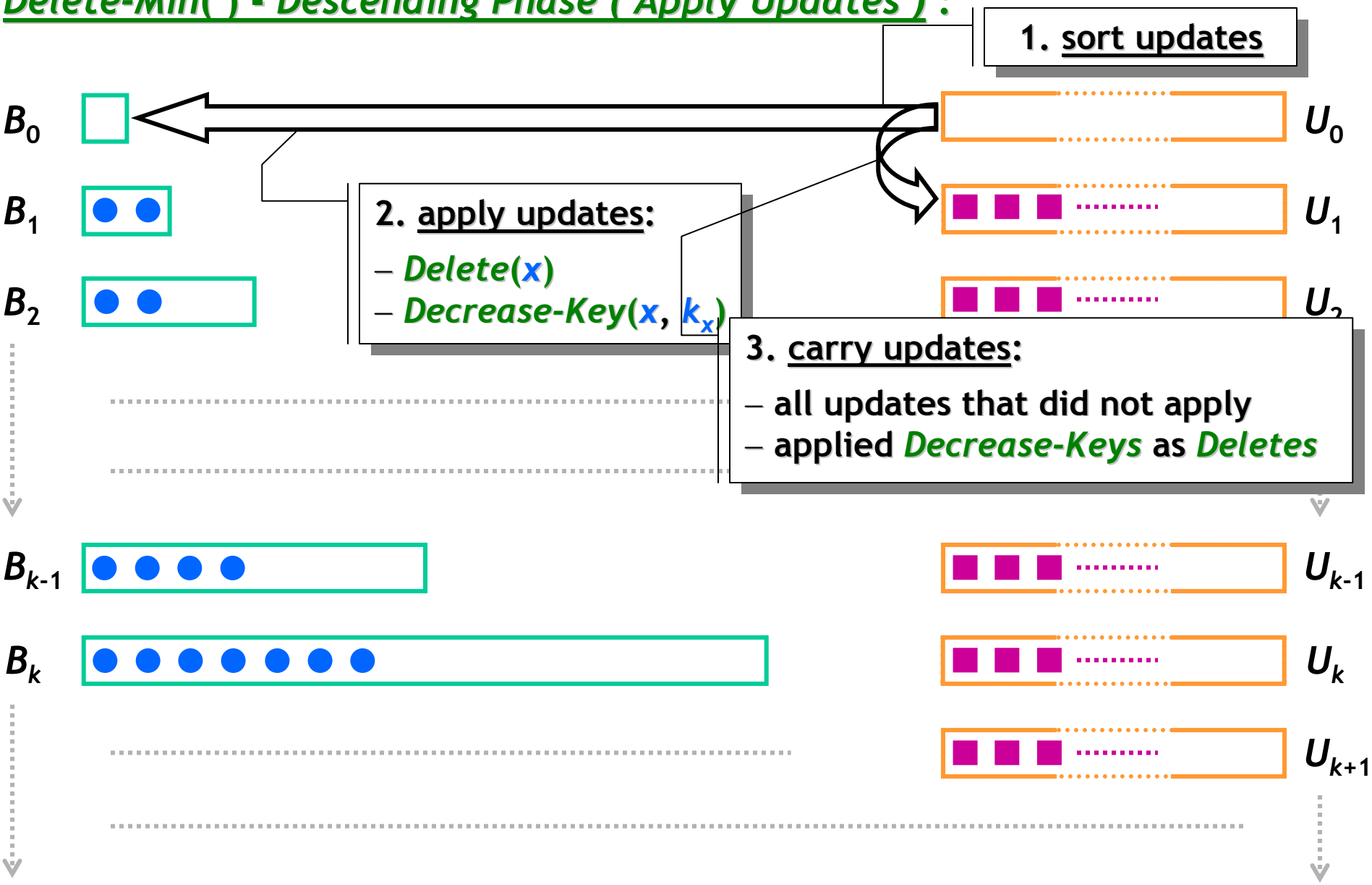
Delete-Min( ) :

Two phases:

- Descending Phase (Apply Updates)
- Ascending Phase (Redistribute Elements)

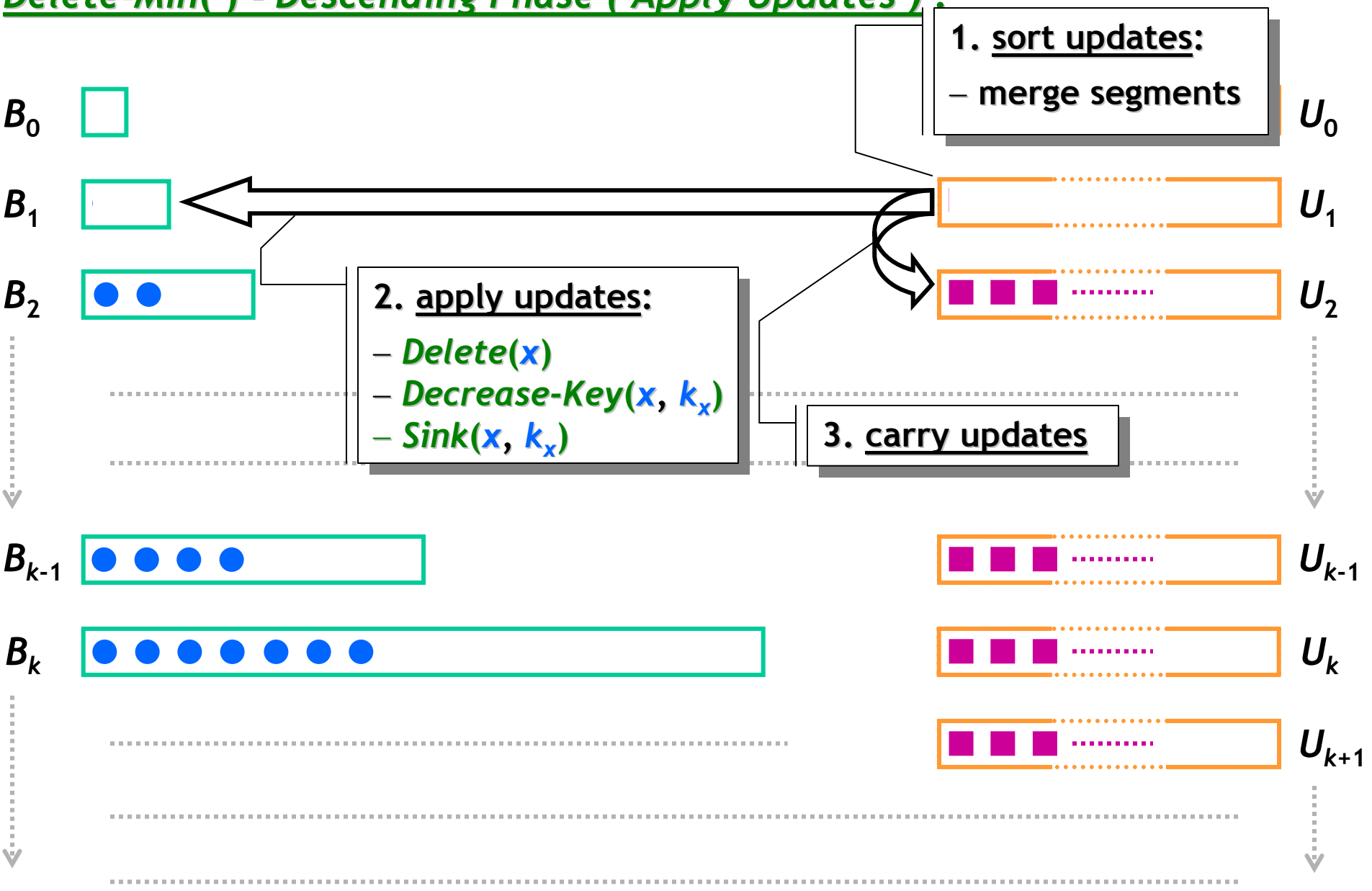
Cache-Oblivious Buffer Heap: Delete-Min

Delete-Min( ) - Descending Phase (Apply Updates) :



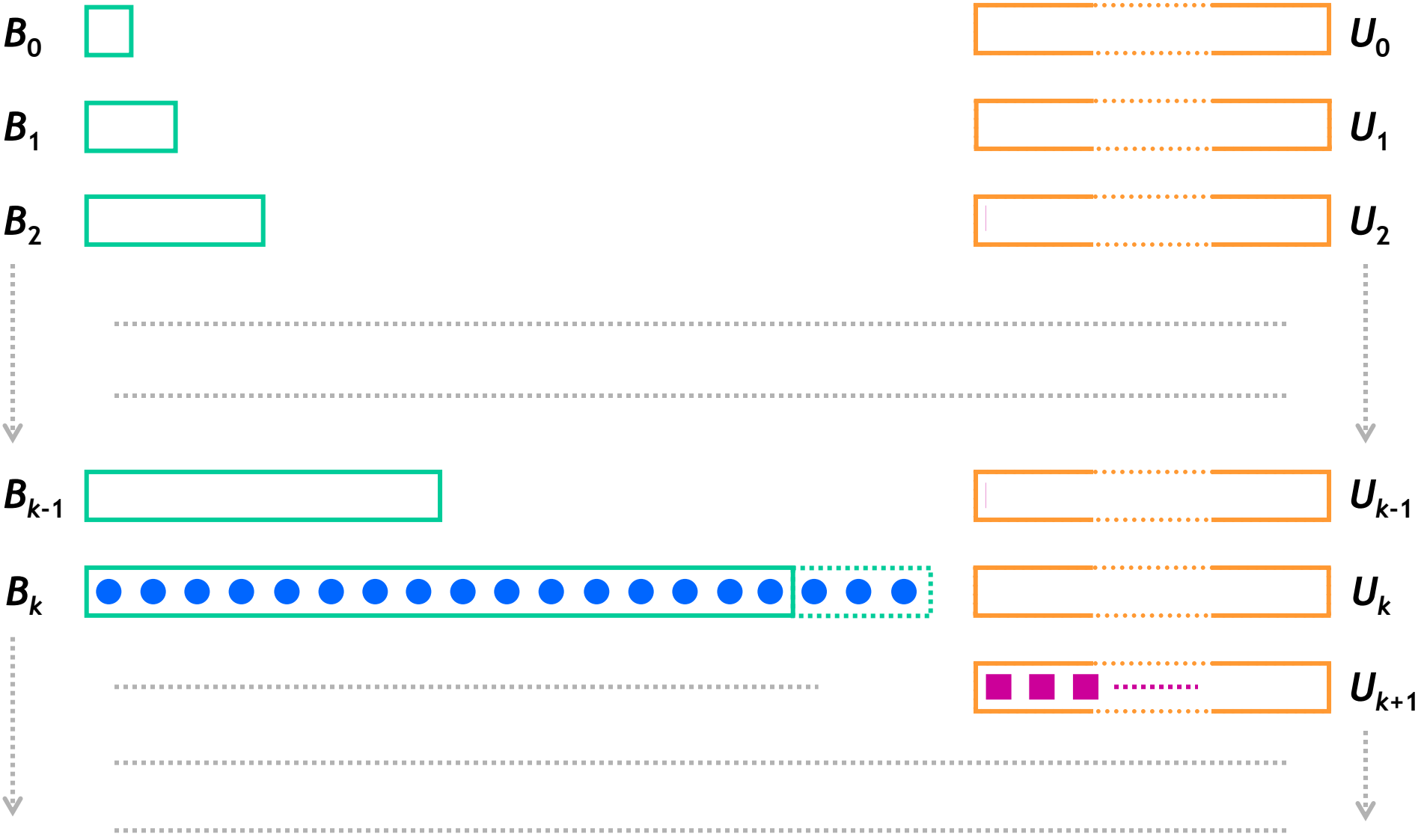
Cache-Oblivious Buffer Heap: Delete-Min

Delete-Min( ) - Descending Phase (Apply Updates) :



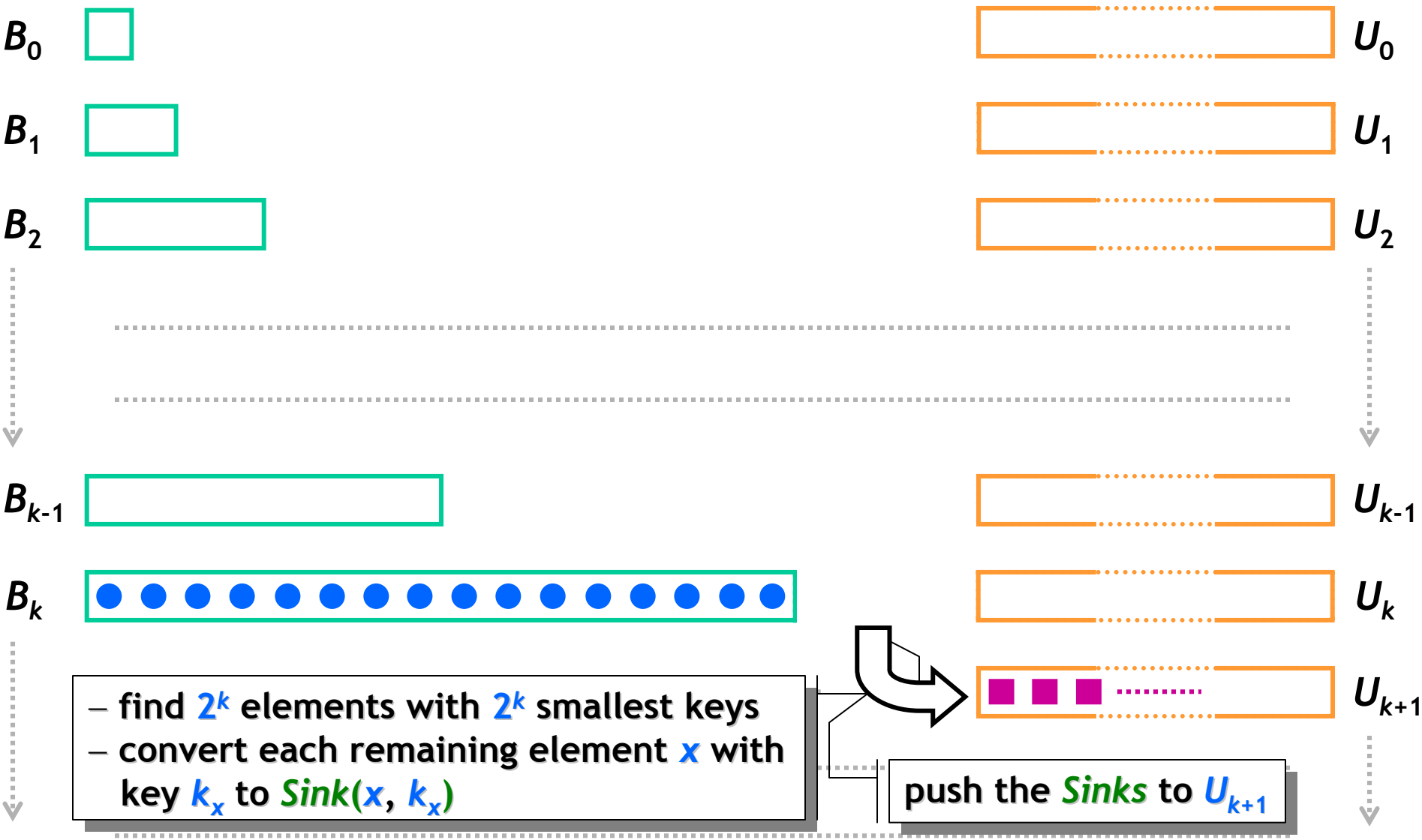
Cache-Oblivious Buffer Heap: Delete-Min

Delete-Min( ) - Descending Phase (Apply Updates) :



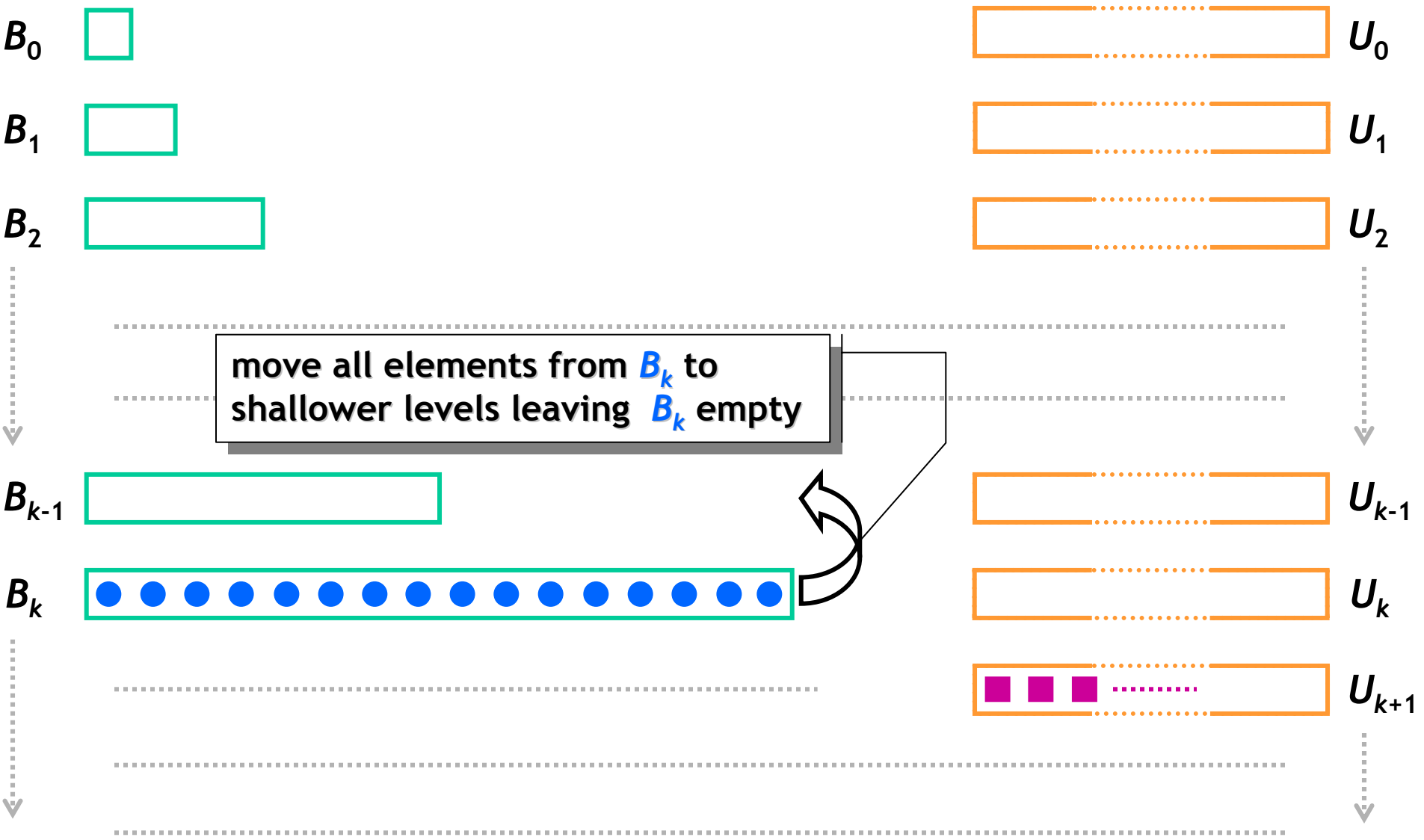
Cache-Oblivious Buffer Heap: Delete-Min

Delete-Min( ) - Ascending Phase (Redistribute Elements) :



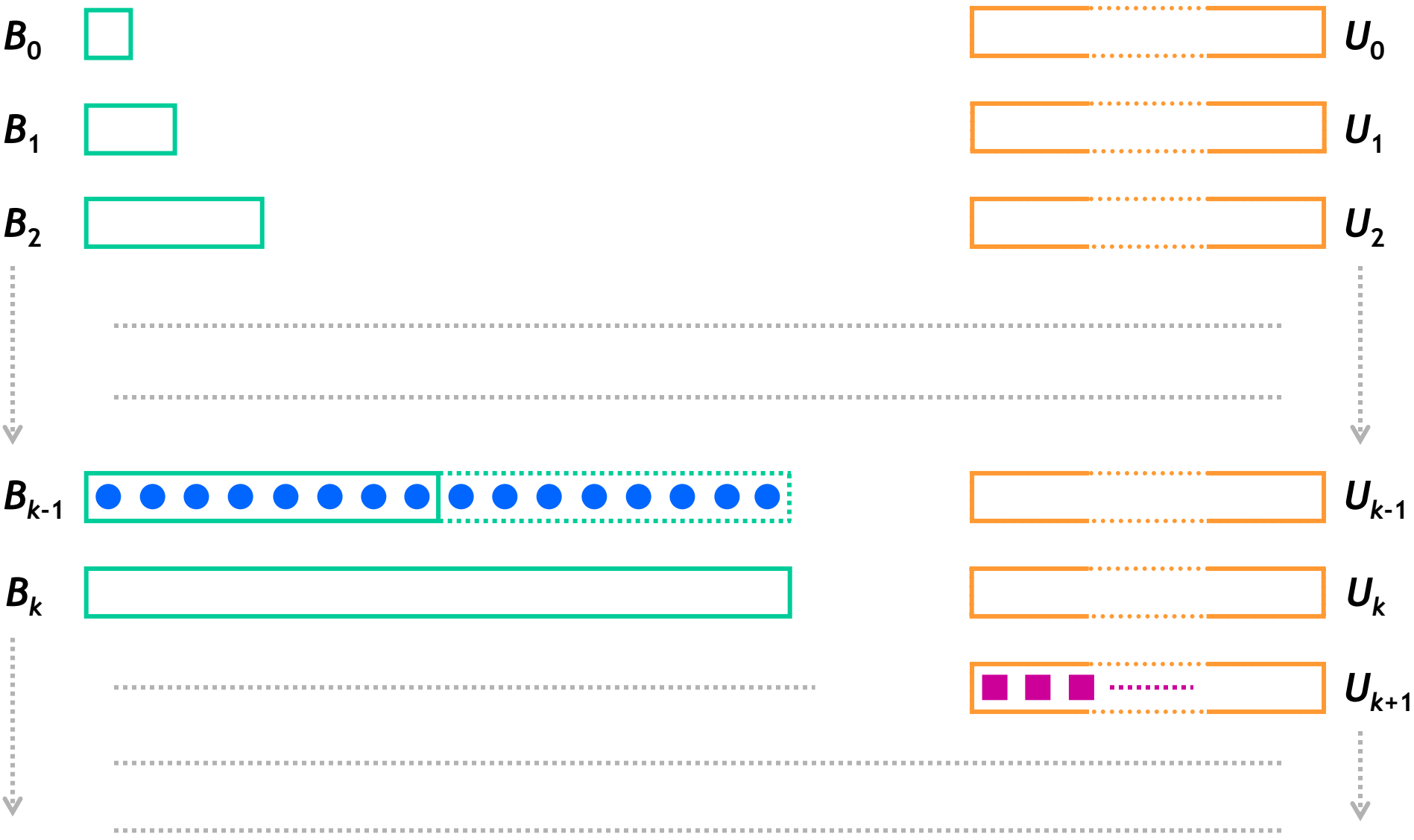
Cache-Oblivious Buffer Heap: Delete-Min

Delete-Min( ) - Ascending Phase (Redistribute Elements) :



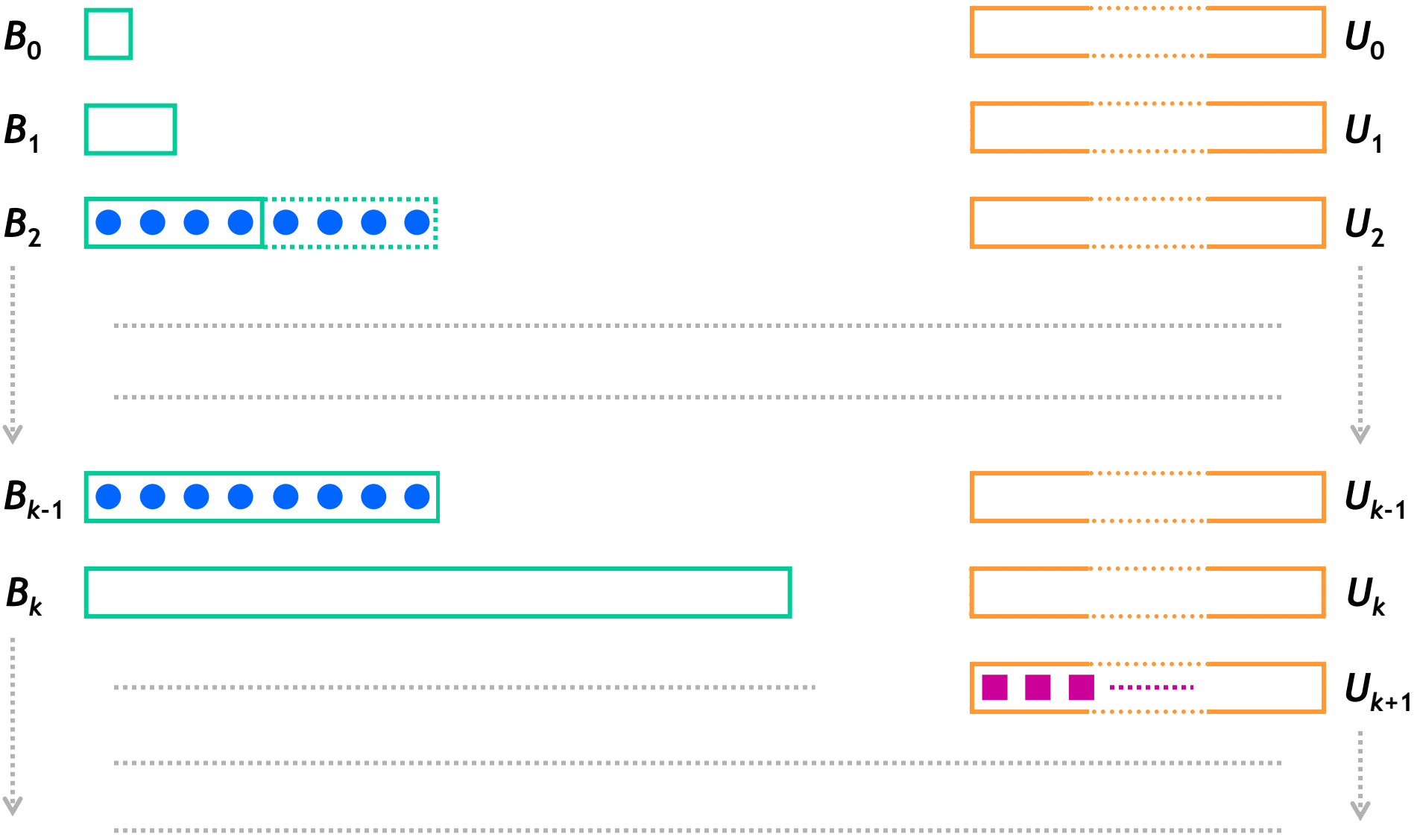
Cache-Oblivious Buffer Heap: Delete-Min

Delete-Min( ) - Ascending Phase (Redistribute Elements) :



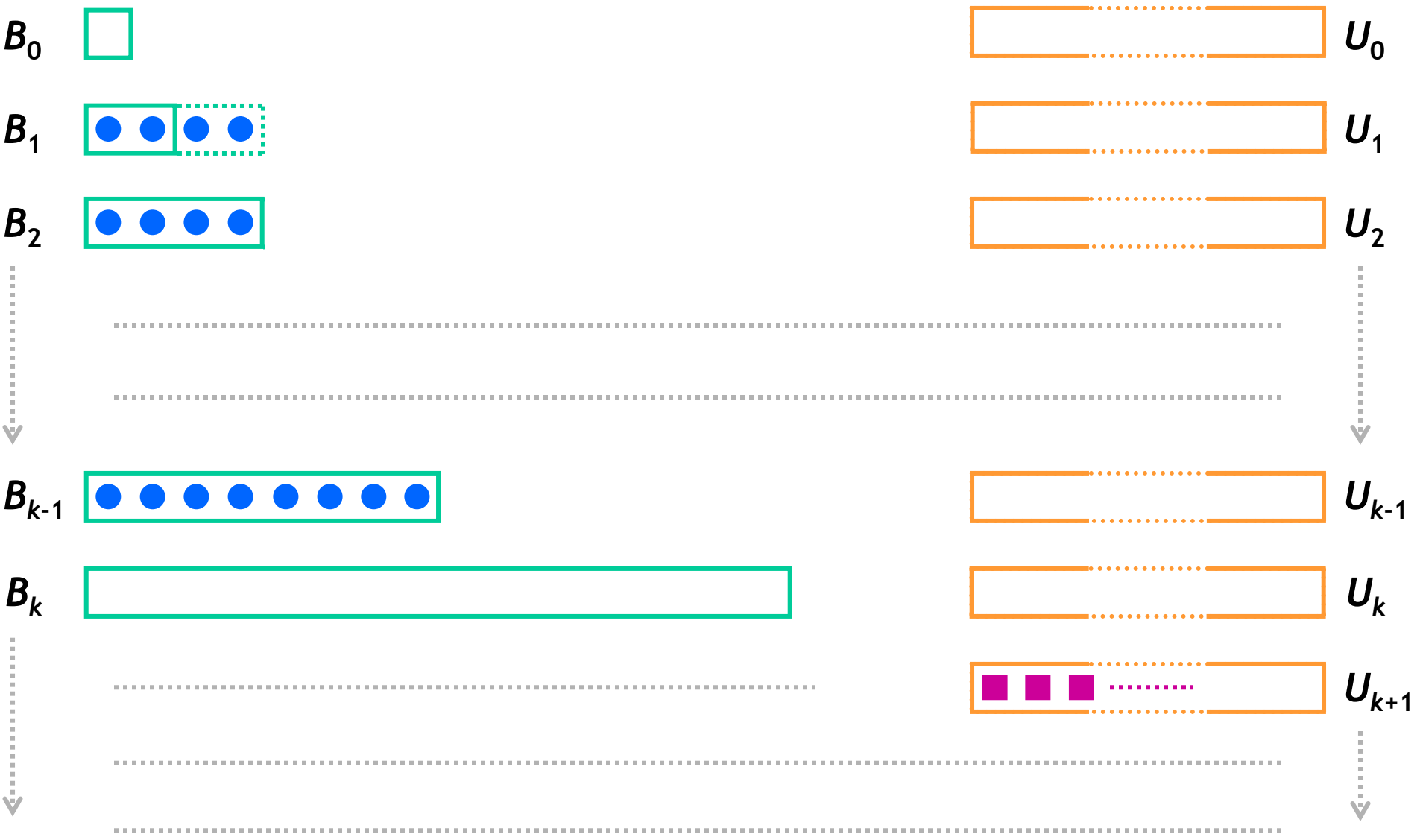
Cache-Oblivious Buffer Heap: Delete-Min

Delete-Min( ) - Ascending Phase (Redistribute Elements) :



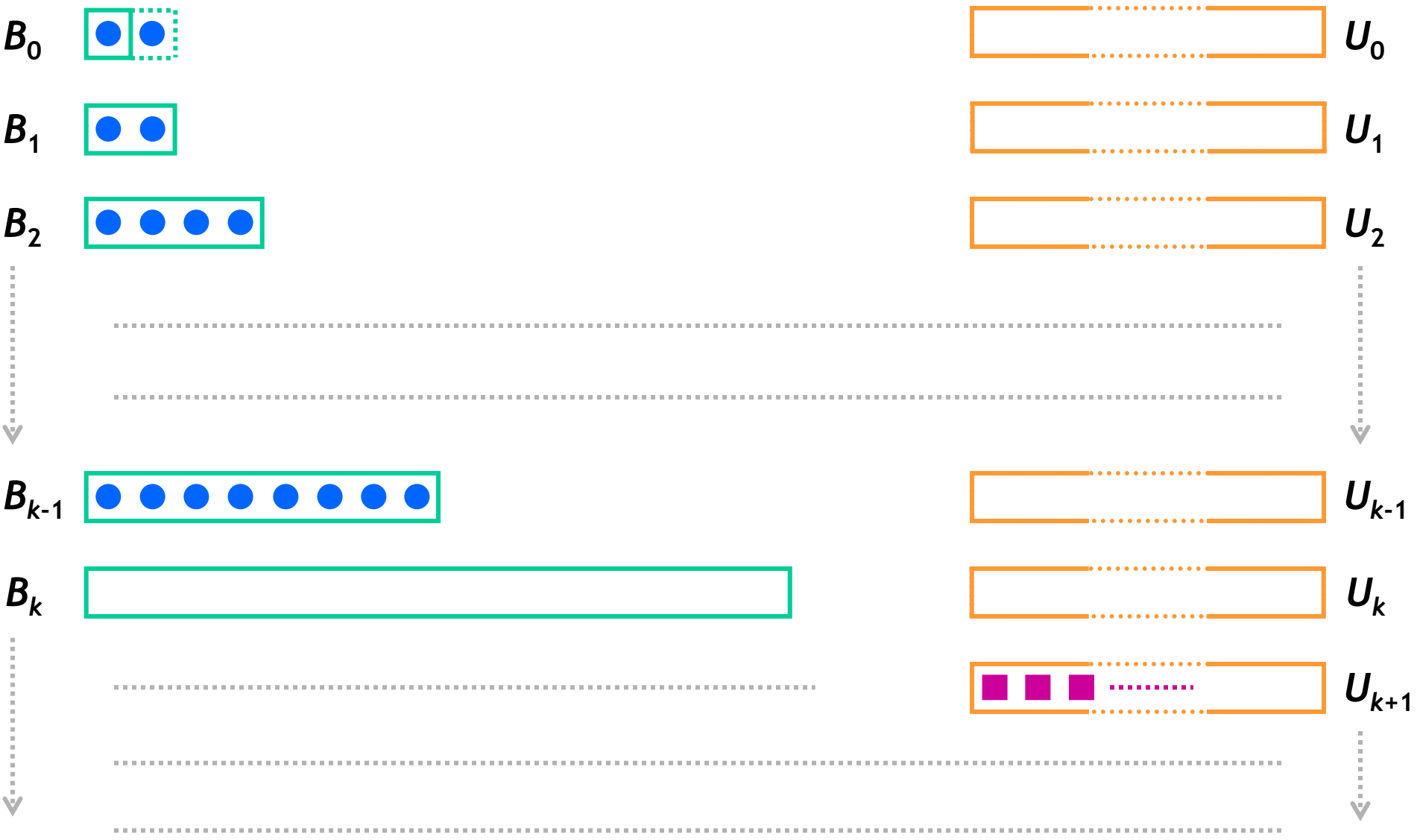
Cache-Oblivious Buffer Heap: Delete-Min

Delete-Min( ) - Ascending Phase (Redistribute Elements) :



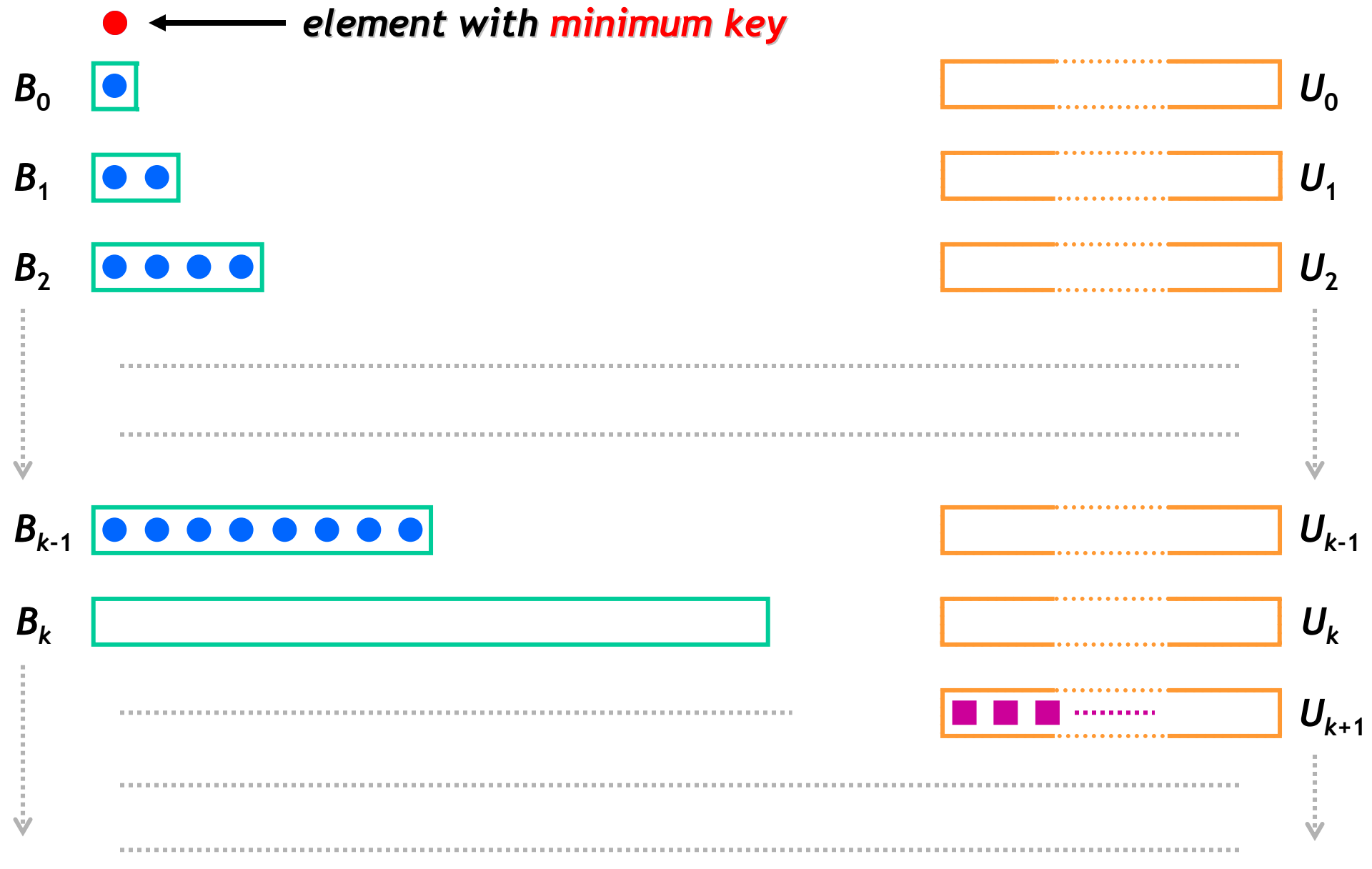
Cache-Oblivious Buffer Heap: Delete-Min

Delete-Min( ) - Ascending Phase (Redistribute Elements) :



Cache-Oblivious Buffer Heap: Delete-Min

Delete-Min( ) - Ascending Phase (Redistribute Elements) :



Cache-Oblivious Buffer Heap: I/O Complexity

Lemma: A *BH* supports *Delete*, *Delete-Min*, and *Decrease-Key* operations in $O((1 / B) \log_2(N / M))$ amortized I/Os each assuming a tall cache.

Proof: We use *Potential Method*. (In equation below $r = \log N$.)

- Each *Decrease-Key* inserted into U_0 will be treated as a pair of operations: ⟨*Decrease-Key*, *Dummy*⟩.
- If *H* is the current state of *BH*, we define *potential* of *H* as:

$$\Phi(H) = \frac{1}{B}\left(3r|U_0| + \sum_{i=1}^{r-1}(2r - i)|U_i| + \sum_{i=0}^{r-1}(i+1)|B_i|\right)$$

Cache-Oblivious Buffer Heap: I/O Complexity

Potential Function: $\Phi(H) = \frac{1}{B}\left(3r|U_0| + \sum_{i=1}^{r-1}(2r-i)|U_i| + \sum_{i=0}^{r-1}(i+1)|B_i| \right)$

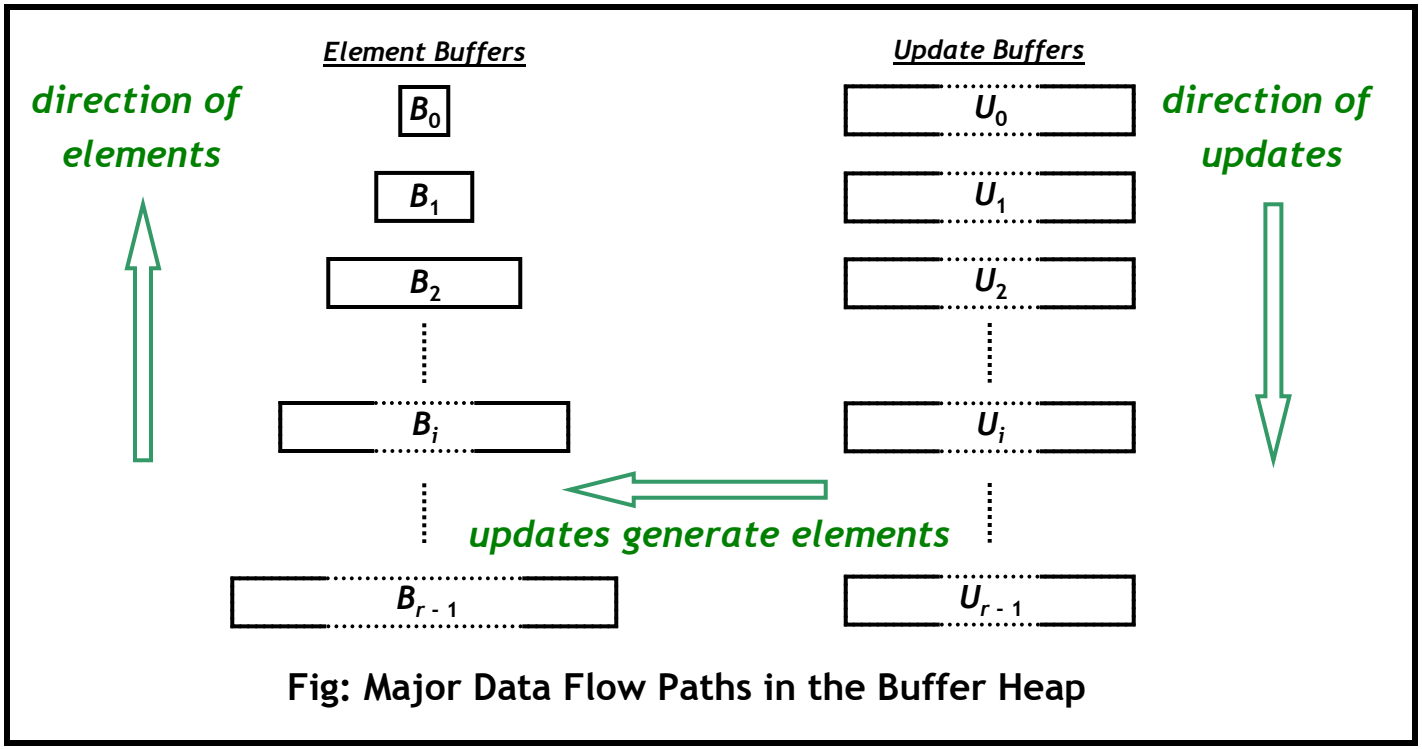


Fig: Major Data Flow Paths in the Buffer Heap

Lemma: A *Buffer Heap* on N elements supports *Delete*, *Delete-Min* and *Decrease-Key* operations cache-obliviously in $O\left(\frac{1}{B}\log_2 N\right)$ amortized I/Os each using $O(N)$ space.

Cache-Oblivious Buffer Heap: Invariants

Invariant 1: $|B_i| \leq 2^i$

Invariant 2:

(a) No key in B_i is larger than any key in B_{i+1}

(b) For each element x in B_i, all updates yet to be applied on x reside in $U_0, U_1, \ldots, U_i$

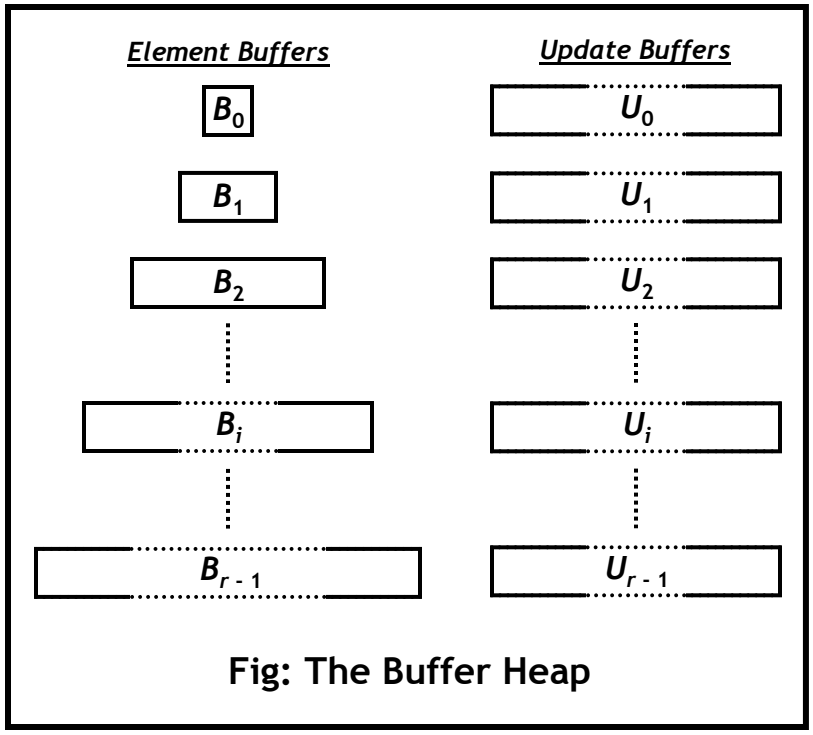


Fig: The Buffer Heap

Invariant 3:

(a) Each B_i is kept sorted by element id

(b) Each U_i (except U_0) is kept (coarsely) sorted by element id and time-stamp

Buffer Heap Summary

- Amortized I/Os per operation: $O\left(\frac{1}{B}\log_2 N\right)$
- Amortized 'running time' per operation: $O(\log N)$
- Buffer heap achieves improved I/O bound while maintaining the traditional $O(\log N)$ running time (amortized)
- Since the top $\log_2 M$ levels of the buffer heap always resides in internal-memory, the amortized I/Os per operation reduces to

$$O\left(\frac{1}{B}\left(\log_2 N - \log_2 M\right)\right) = O\left(\frac{1}{B}\log_2 \frac{N}{M}\right)$$

The Cache-Oblivious Model: Some Known Results

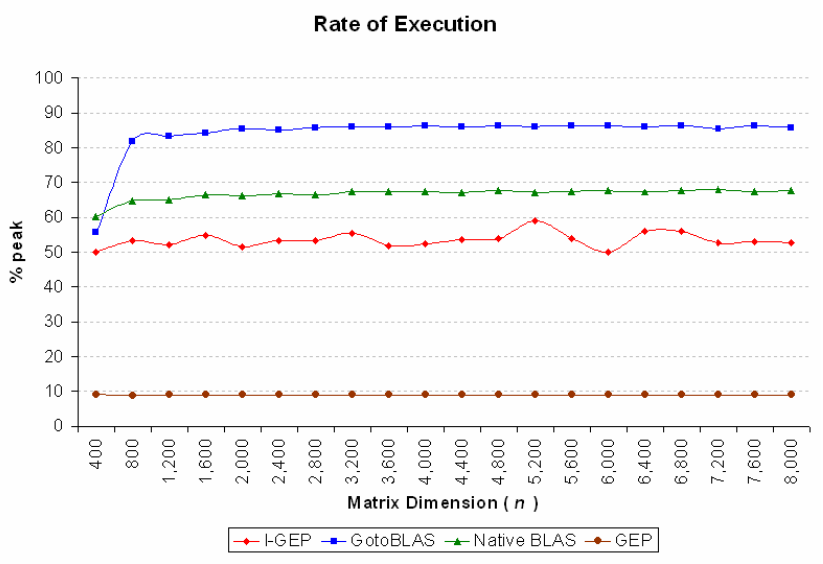
Problem	Cache-Aware Results	Cache-Oblivious Results
Array Scanning (*scan(N)*)	$O\left(\frac{N}{B}\right)$	$O\left(\frac{N}{B}\right)$
Sorting (*sort(N)*)	$O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$	$O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$
Selection	$O(scan(N))$	$O(scan(N))$
Priority Queue [Am] (*Insert*, *Weak Delete*, *Delete-Min*)	$O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$	$O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$
B-Trees [Am] (*Insert*, *Delete*)	$O\left(\log_B\frac{N}{B}\right)$	$O\left(\log_B\frac{N}{B}\right)$
List Ranking	$O(sort(N))$	$O(sort(N))$
Directed BFS/DFS	$O\left(\left(V+\frac{E}{B}\right)\cdot\log_2\frac{V}{B}\right)$	$O\left(\left(V+\frac{E}{B}\right)\cdot\log_2\frac{V}{B}\right)$
Undirected BFS	$O(V+sort(E))$	$O(V+sort(E))$
Minimum Spanning Forest	$O\left(\min\left(sort(E)\log_2\log_2 V,\ V+sort(E)\right)\right)$	$O\left(\min\left(sort(E)\log_2\log_2\frac{VB}{E},\ V+sort(E)\right)\right)$

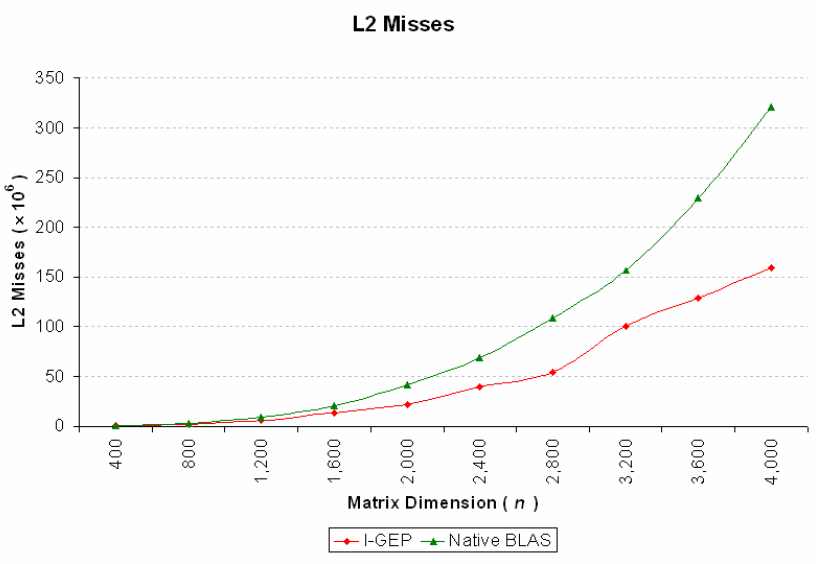
Table 1: N = # elements. $V = |V[G]|$, $E = |E[G]|$, Am = Amortized.

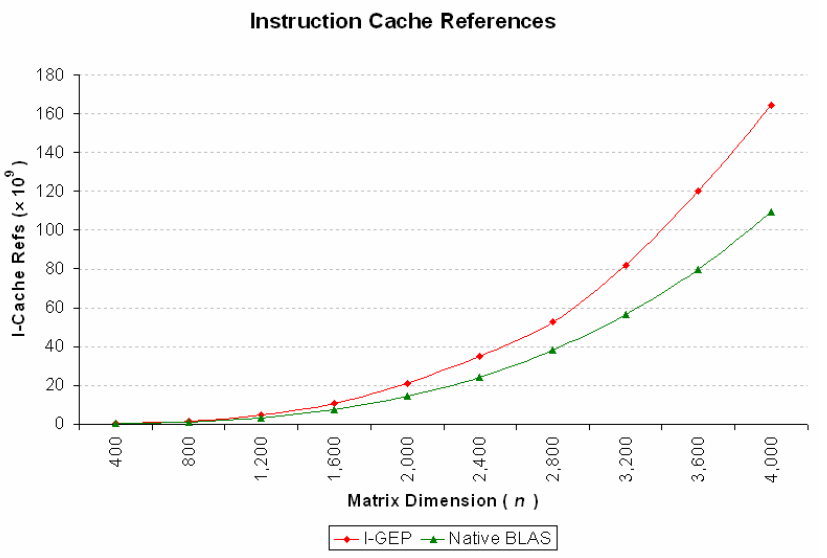
Some of these results require a *tall cache*: $M=\Omega\left(B^{1+\varepsilon}\right)$.

Comparison to BLAS: MM on Xeon (single proc)

Architecture	Processor Speed	L1 Cache (*B*)	L2 Cache (*B*)	RAM
Intel Xeon	3.06 GHz	8 KB (64 B)	512 KB (64 B)	4 GB

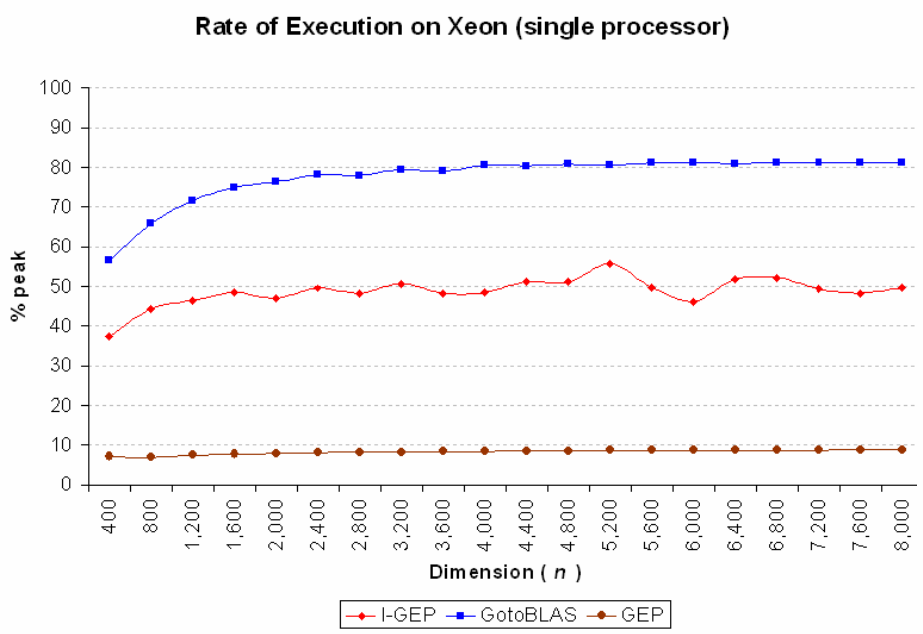


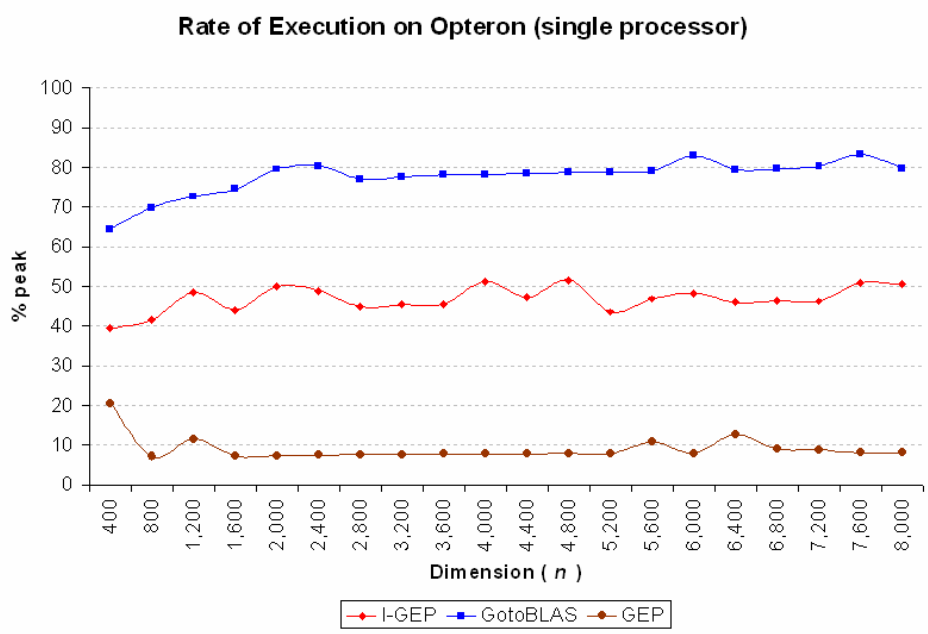




Comparison to BLAS: Gaussian Elimination w/o Pivoting

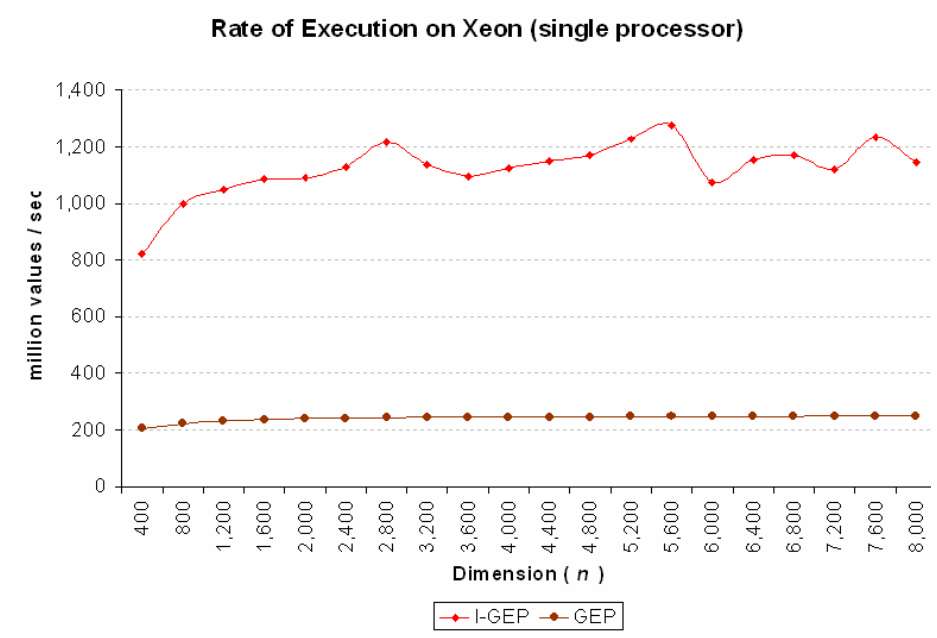
Architecture	Processor Speed	L1 Cache (*B*)	L2 Cache (*B*)	RAM
Intel Xeon	3.06 GHz	8 KB (64 B)	512 KB (64 B)	4 GB
AMD Opteron	2.4 GHz	64 KB (64 B)	1 MB (64 B)	4 GB

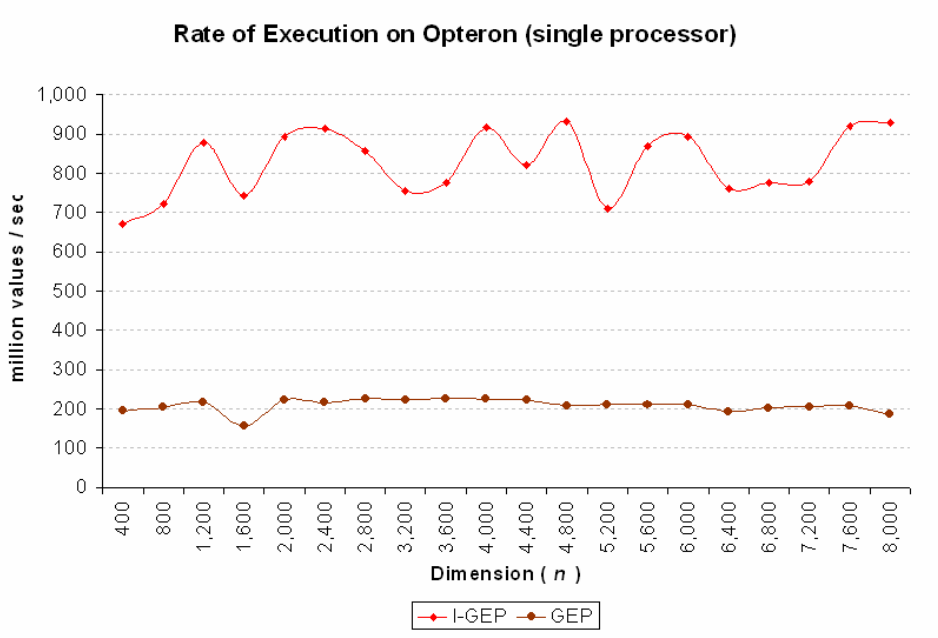




Floyd-Warshall's APSP (single proc)

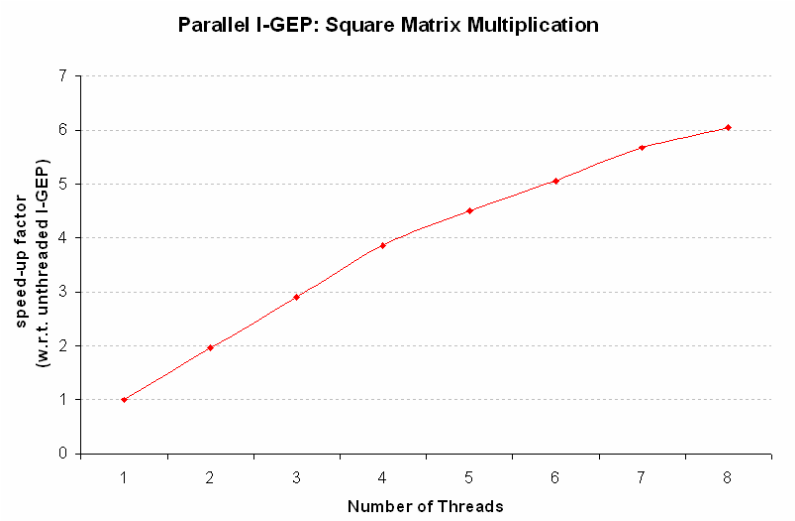
Model	Processor Speed	L1 Cache (*B*)	L2 Cache (*B*)	RAM
Intel P4 Xeon	3.06 GHz	8 KB (64 B)	512 KB (64 B)	4 GB
AMD Opteron 250	2.4 GHz	64 KB (64 B)	1 MB (64 B)	4 GB

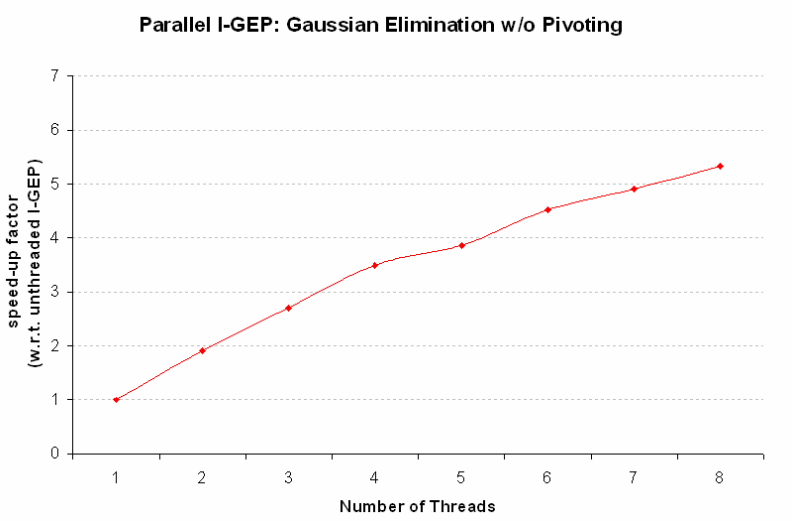


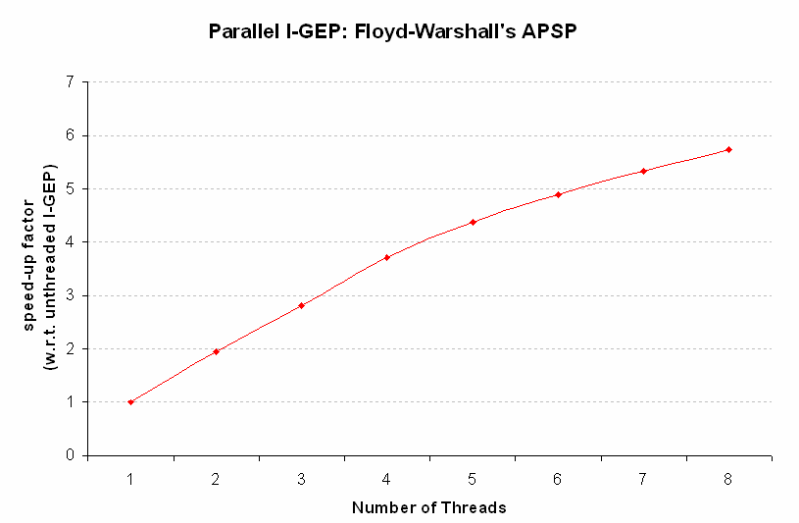


Parallel I-GEP: Speed-Up Factors

Model	# Processors	Processor Speed	L1 Cache (*B*)	L2 Cache (*B*)	RAM
AMD Opteron 850	8	2.2 GHz	64 KB (64 B)	1 MB (64 B)	32 GB

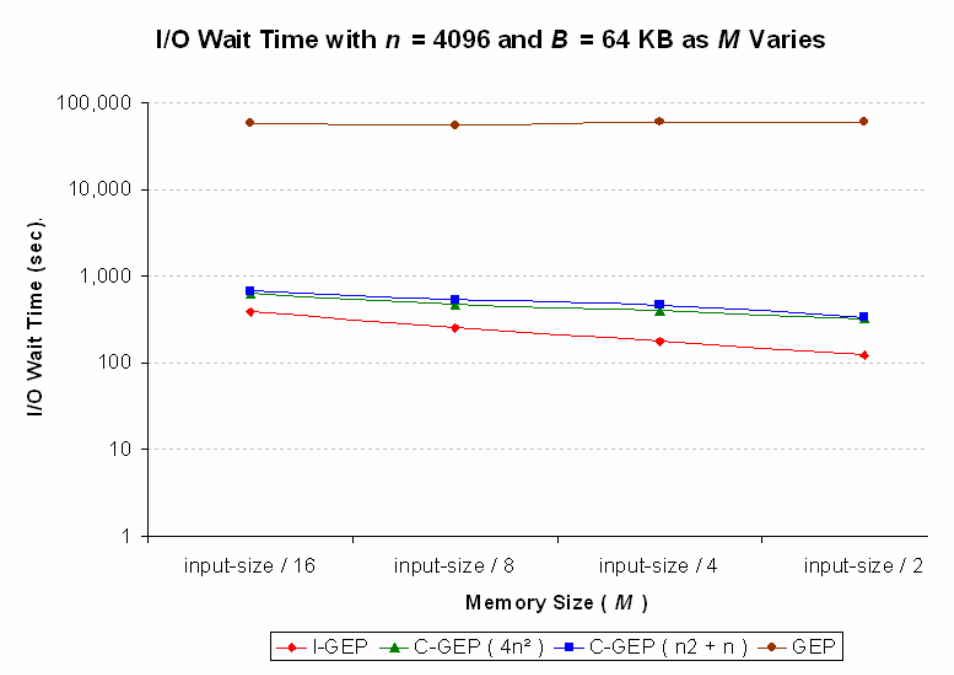


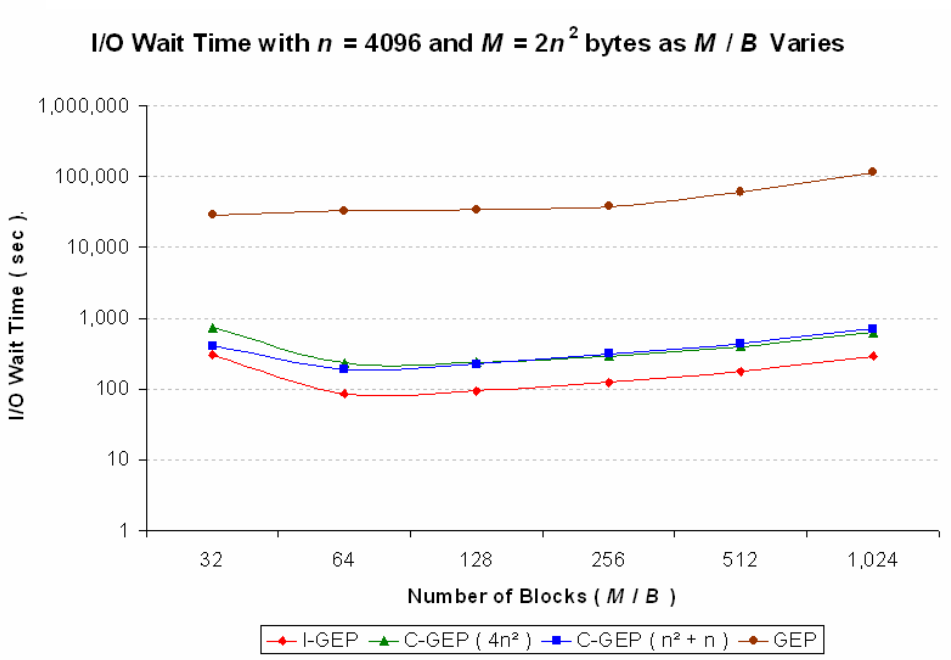




Out-of-Core: I/O Wait Times Using STXXL

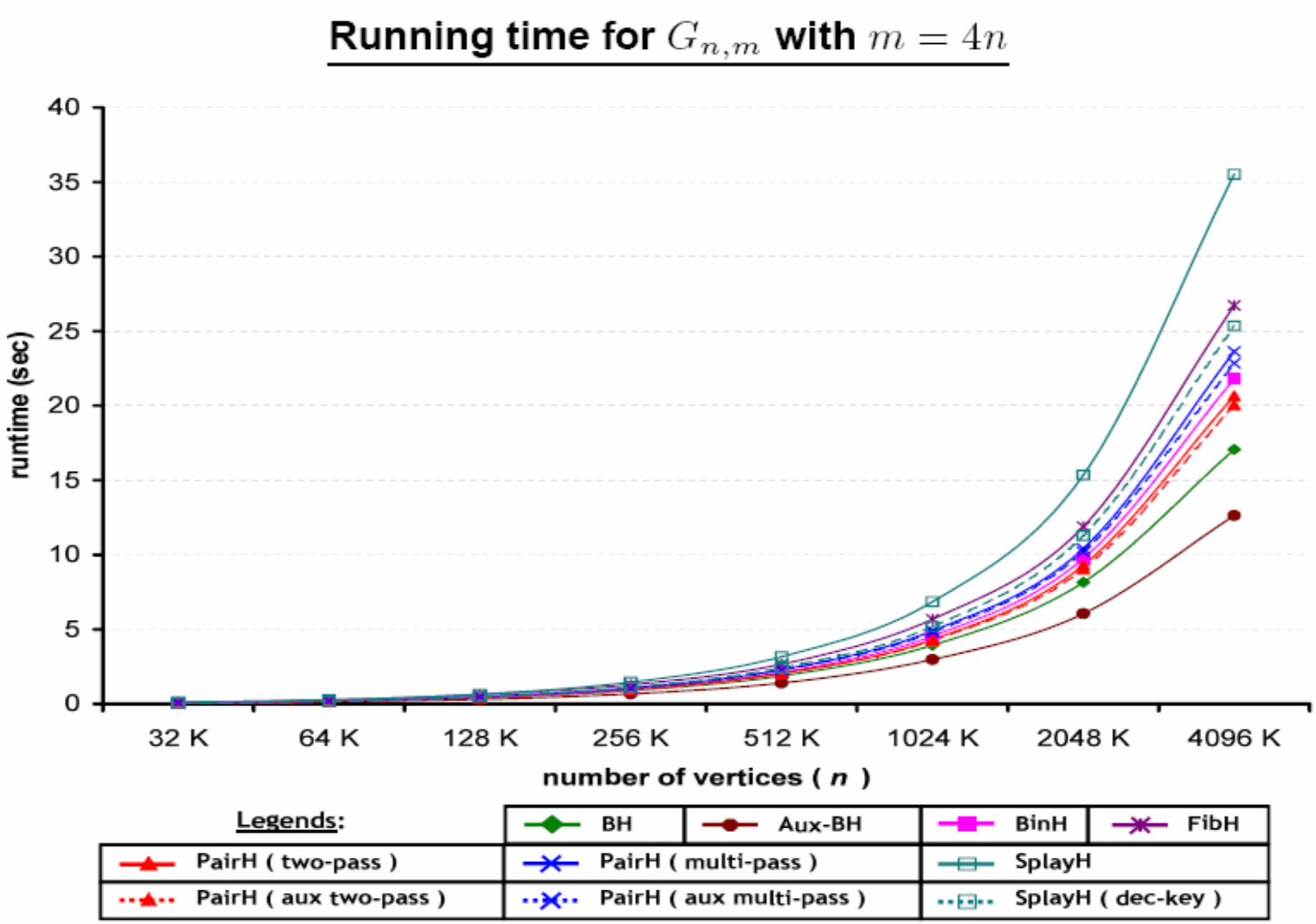
Processor	Speed	RAM	Local Hard Disk
Intel P4 Xeon	3 GHz	4 GB	73 GB, 10K RPM, 8 MB buffer, ~5ms avg. seek time, 107 MB/s max xfer rate





SSSP: In-core Running Times

Architecture	Processor Speed	L1 Cache (*B*)	L2 Cache (*B*)	RAM
Intel P4 Xeon	3.06 GHz	8 KB (64 B)	512 KB (64 B)	4 GB

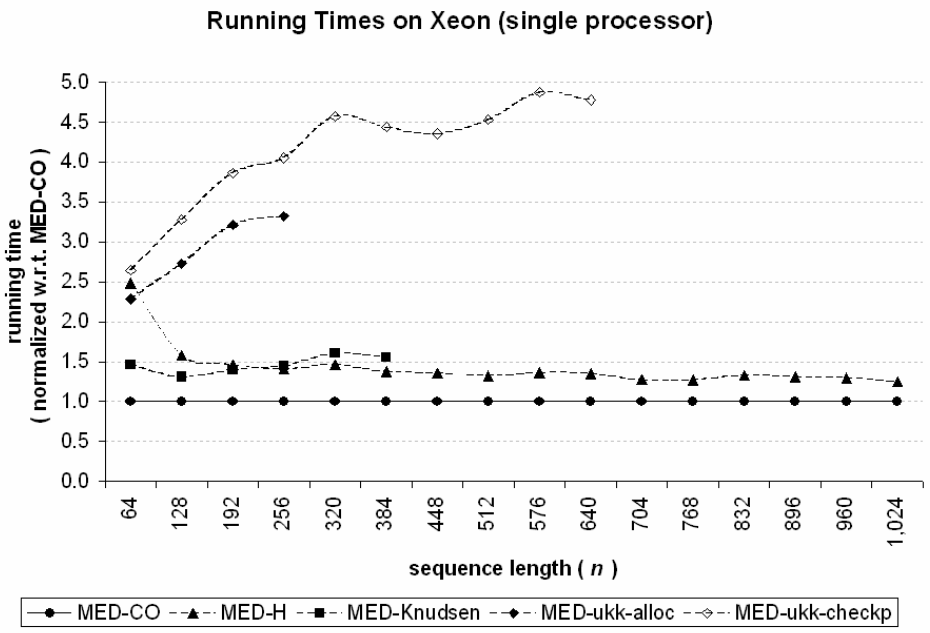


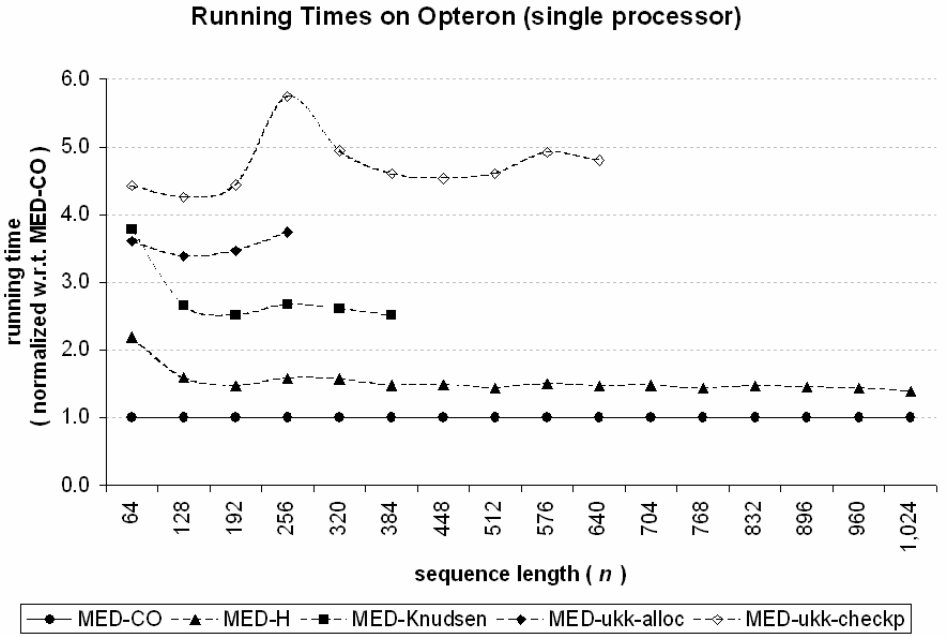
Median: Algorithms Compared

Algorithm	Comments	Time	Space	Cache Misses
MED-CO	Our cache-oblivious algorithm	$O(n^3)$	$O(n^2)$	$O\left(\frac{n^3}{B\sqrt{M}}\right)$
MED-Knudsen	Knudsen's implementation of his algorithm	$O(n^3)$	$O(n^3)$	$O\left(\frac{n^3}{B}\right)$
MED-H	Our implementation of MED-Knudsen using Hirschberg's technique	$O(n^3)$	$O(n^2)$	$O\left(\frac{n^3}{B}\right)$
MED-ukk.alloc	Powell's implementation of an $O(d^3)$-space algorithm (d = 3-way edit dist)	$O(n + d^3)$	$O(n + d^3)$	$O\left(\frac{d^3}{B}\right)$
MED-ukk.checkp	Powell's implementation of an $O(d^2)$-space algorithm (d = 3-way edit dist)	$O(n \log d + d^3)$	$O(n + d^2)$	$O\left(\frac{d^3}{B}\right)$

Median: Random Sequences

Model	Processor Speed	L1 Cache (*B*)	L2 Cache (*B*)	RAM
Intel P4 Xeon	3.06 GHz	8 KB (64 B)	512 KB (64 B)	4 GB
AMD Opteron 250	2.4 GHz	64 KB (64 B)	1 MB (64 B)	4 GB





Pair-wise Sequence Alignment: Random Sequences

Model	# Processors	Processor Speed	L1 Cache (*B*)	L2 Cache (*B*)	RAM
AMD Opteron 250	2	2.4 GHz	64 KB (64 B)	1 MB (64 B)	4 GB