

IJCAI Tutorial

Introduction to Parallelism

Marc Snir



Outline

- Introduction
- Current and future parallel architectures
- Software to Support Parallelism
- Metrics
- What will go wrong



INTRODUCTION



General Comments

- Feel free to ask questions and any time
- Feel free to provide feedback to help improve lectures
- Enjoy

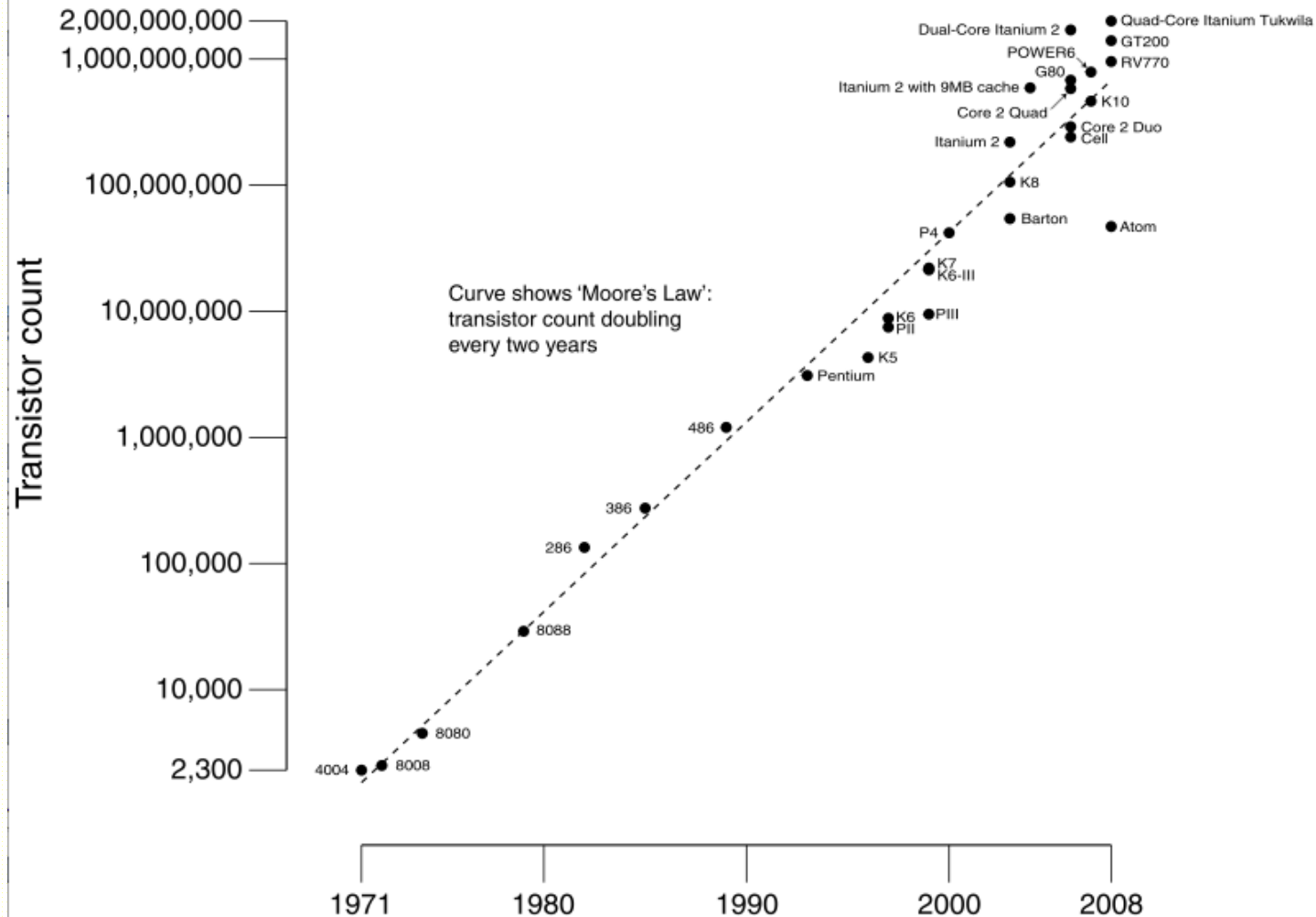


HARDWARE



Moore's Law

Number of transistors on a chip doubles every ~2 years

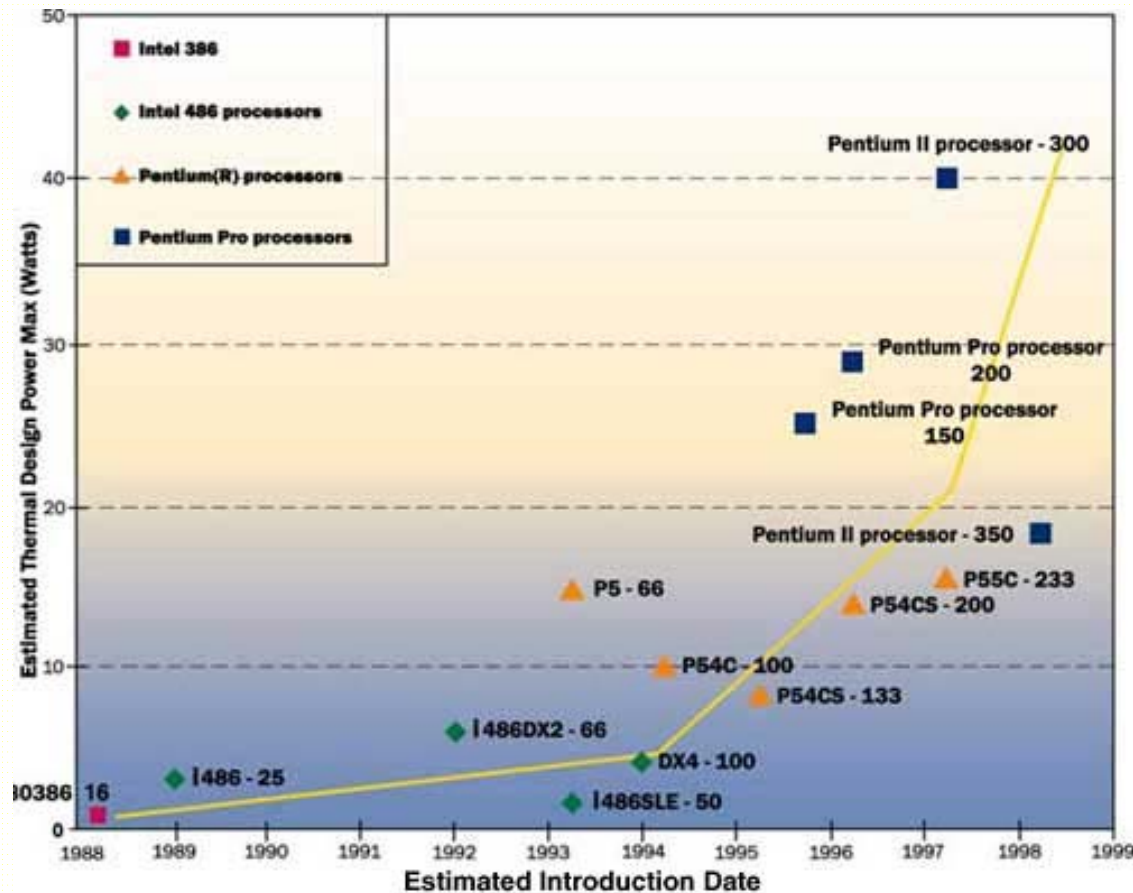
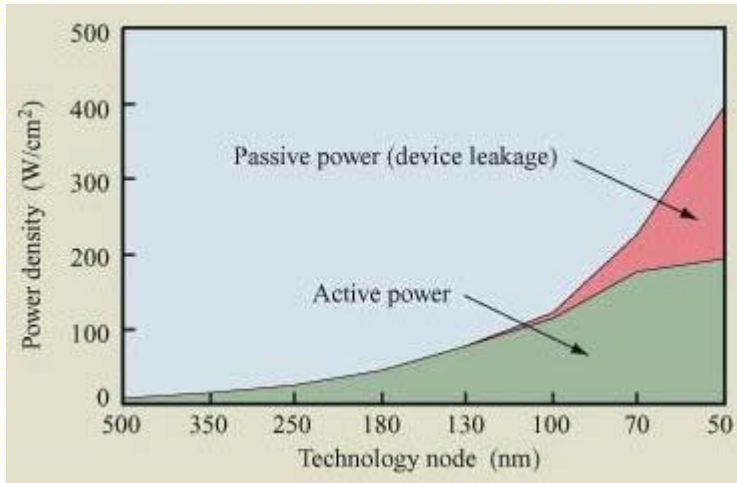


How Do We Leverage More, Smaller Transistors?

- Increase clock frequency
 - Cannot do it anymore – consumes too much energy
- Increase number of instructions per cycle (IPC)
 - Cannot do it anymore – memory wall
- *Cannot speed up anymore single sequential thread – need parallelism*



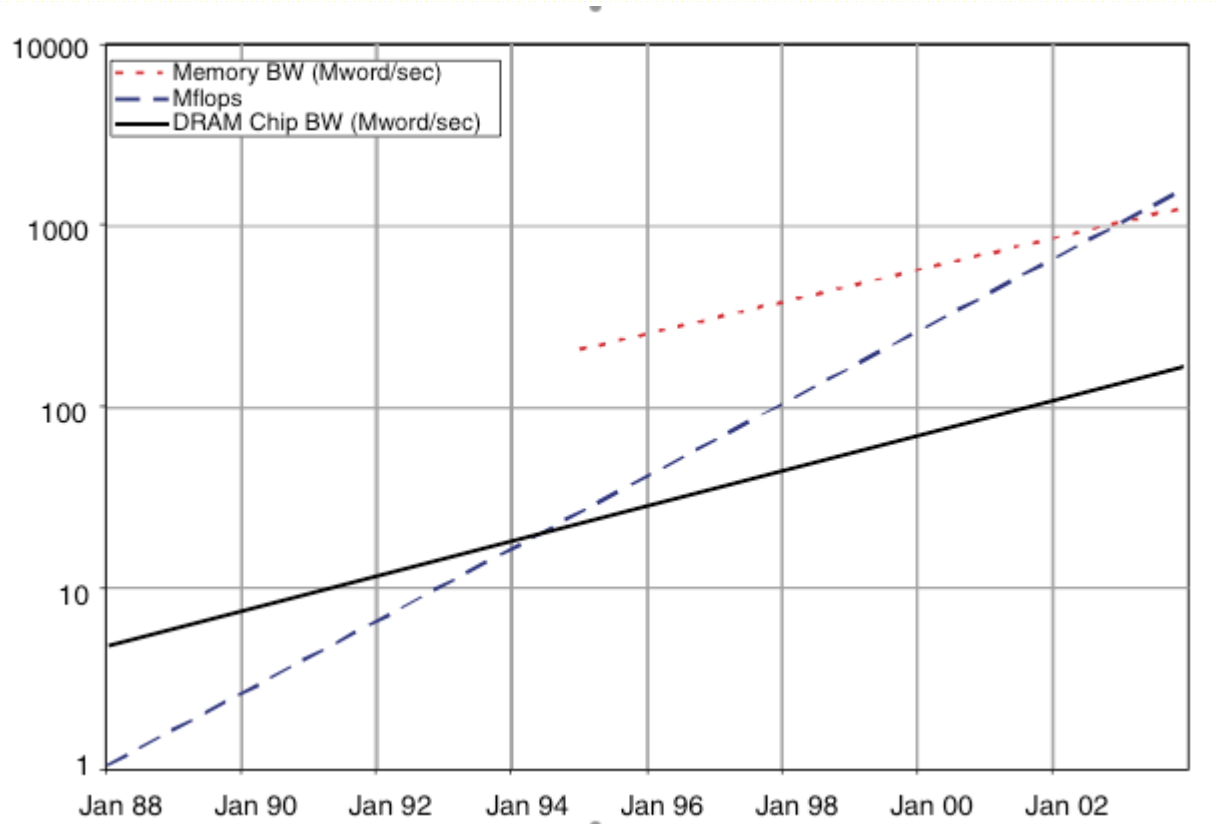
Power Per Chip



We are now at 130W

The Memory Wall (1)

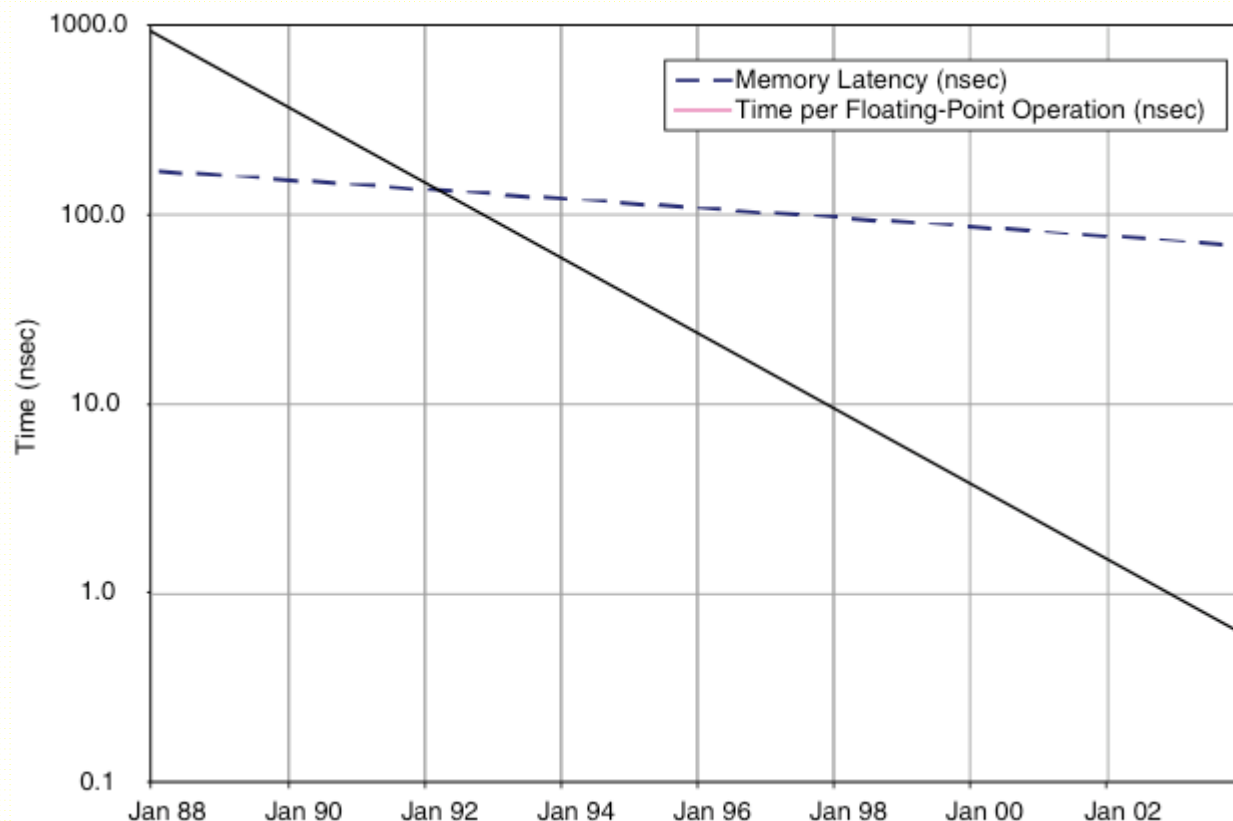
- Memory bandwidth increases more slowly than CPU performance
 - 25% vs. 59% CAGR



~1 Byte per flop

The Memory Wall (2)

- Memory latency decreases much more slowly than CPU performance increases
 - 5.5% vs. 59% CAGR



1 cache miss =
100's instructions

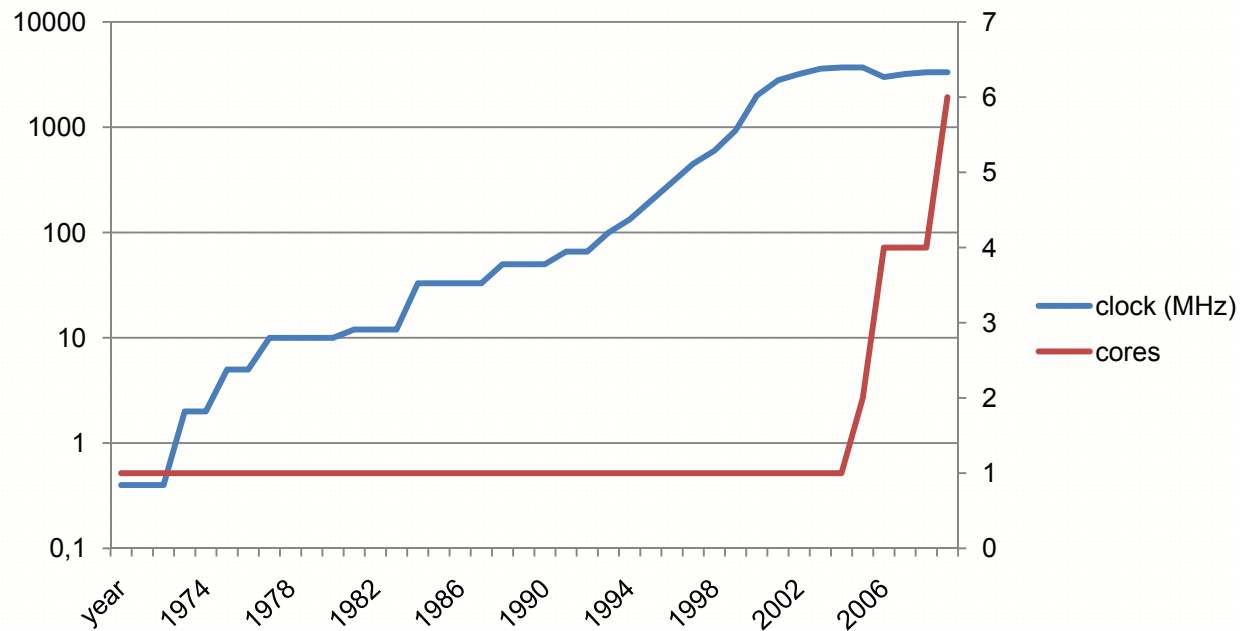
**Performance
is memory
bound most
of the time**

Parallel Hardware

- Simultaneous Multi-Threading (SMT)
 - multiple instruction streams executing on one core (multiple register files sharing same instruction units)
 - better use of core resources
 - Hides memory latency
 - slower execution, per stream
- Multicore
 - multiple independent processors per chip; shared memory
 - shared memory bandwidth is possible bottleneck
- Vector instructions
 - E.g., Intel SSE (processing 128 bits in one instruction)
- Accelerators
 - E.g., GPUs

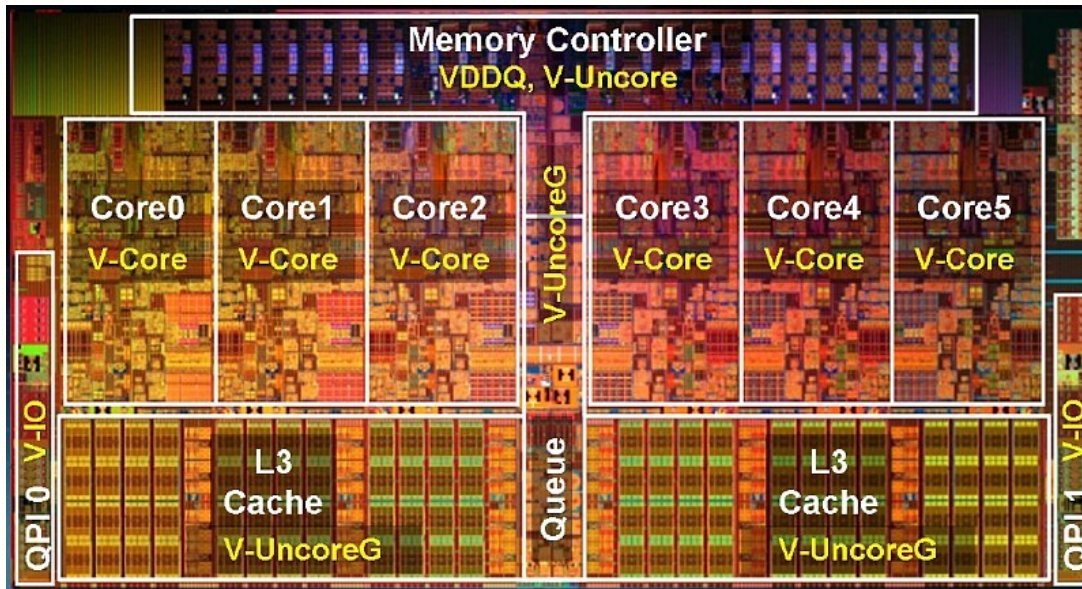


Intel – Peak Clock and Number of Cores



By year of introduction

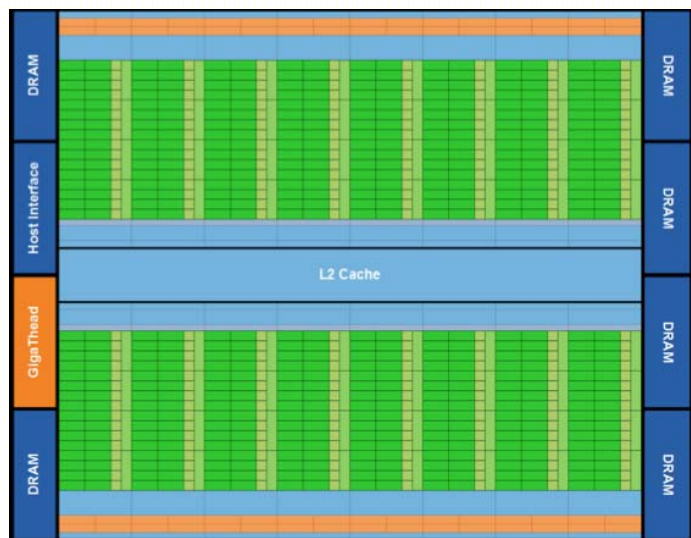
Example – Intel Nehalem/Westmere



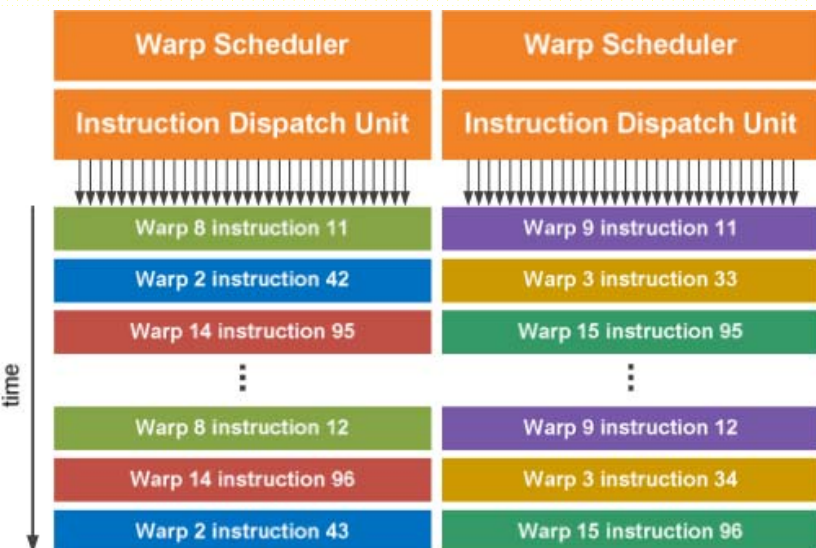
- 4-10 cores
- 1/2 threads per core
- vector unit per core
- Three cache levels:
 - Private L1(32K Data, 32K Instruction),
 - Private L2 (512 KB)
 - Shared L3 (16-30 MB)
- 32 nm technology
- Can be assembled in quad chip SMPs

~50 Gflop/s peak!

GPU – More, Simpler Cores



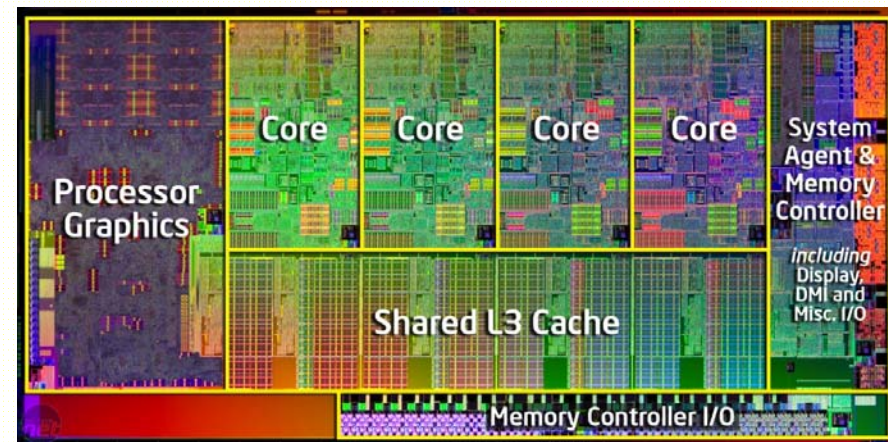
- nVIDIA Fermi – 512 cores
– > 1 Tflop/s peak!
- Warp = 32 threads execute same instruction (or 2 different instructions)
- Need to interleave multiple warps per group of cores to prevent pipeline bubbles (concurrent multithreading)
- Memory performance is lower if accesses are spread
- GPU is connected to main memory via (slow) I/O bus



Future Evolution

- Number of cores per chip double every 2 years
 - > 256 cores by end of decade ??
- Cores may be heterogeneous
 - fat and thin cores
 - on chip GPU-like accelerators
 - Special function units (e.g., encryption)
- *Not all cores can run at the same time*
- Denser, smarter memory subsystems
 - 3D stacks
 - Intelligent memory controllers

Intel Sandy Bridge



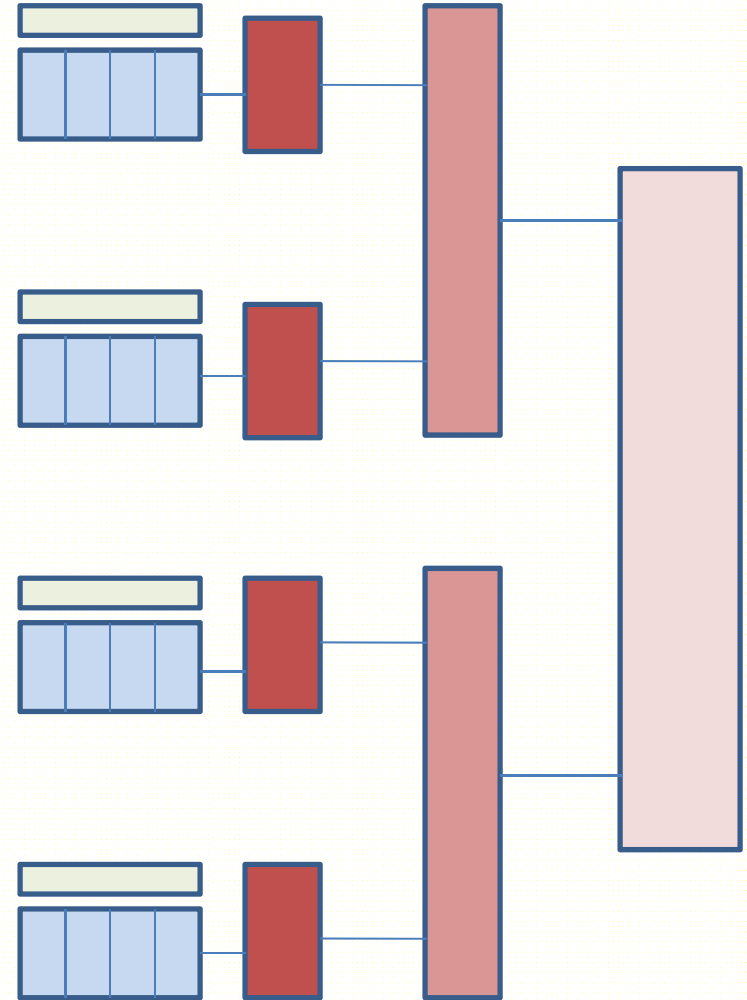
Clusters – Going Beyond Shared Memory

- Hard (expensive) to build large shared-memory systems
 - Hard to run reliably
- *Cluster*: Multiple connected computers that are managed as one system
- *Cloud*: very large cluster where (virtual) resources are allocated on demand
- *Supercomputer*: very large cluster with interconnect optimized for low-latency/high-bandwidth communication



Hardware Performance Bottlenecks

- SMT -- more threads, with lesser performance: is this a worthwhile tradeoff?
- Vector units: is the code vectorized?
- *Cores: Do we suffer from cache misses?*
- Multicore: Do we keep all cores busy?
- Memory: Do cores interfere with each other in shared memory accesses?
- Is the performance bottleneck a shared resource (e.g., memory) or a replicated resource (e.g., cores, private cache) ?



Bottlenecks for Clusters

- Internode communication
 - How long it takes to send/receive a (zero byte) message (*latency*)?
 - What is the network bandwidth at the node interface (*injection bandwidth*)?
 - What happens if all nodes communicate simultaneously (*bisection bandwidth*)?
- Are all nodes computing all the time (*jitter*)?
- And how about I/O?
 - Local file system, cluster file system
 - Distributed tables (row oriented, column oriented)

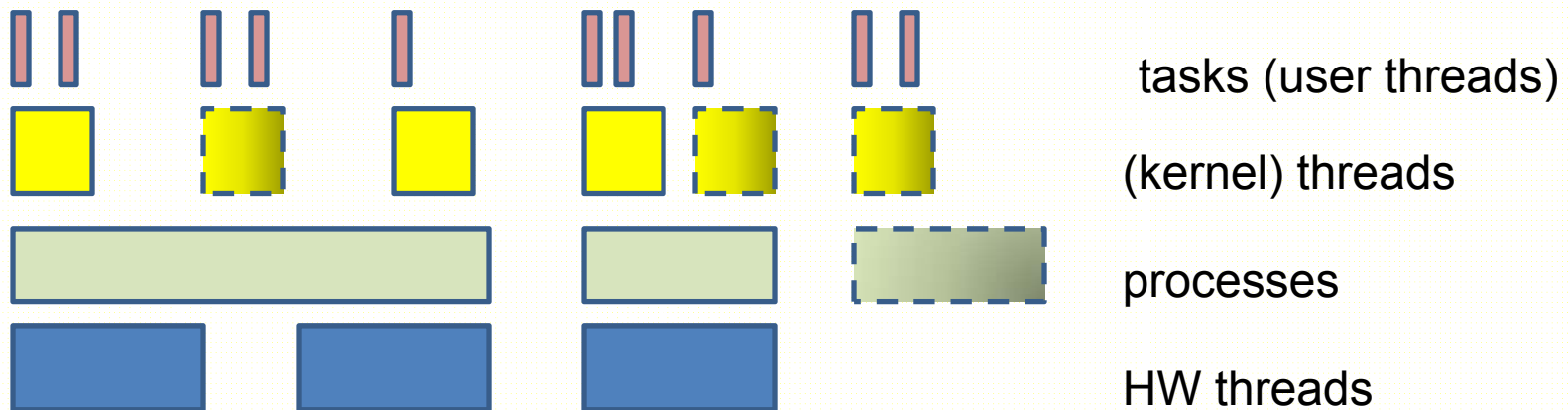


THE SOFTWARE STACK



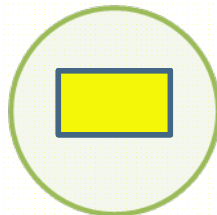
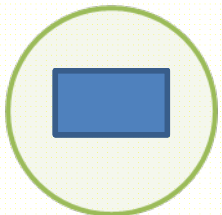
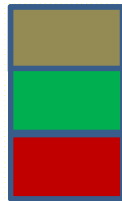
Software Objects

- **Process:**
 - address space, file descriptors, queues, pipes, etc.
 - code
 - one or more kernel threads
- **Kernel thread:**
 - Stack, instruction counter
 - *Scheduled by OS (in kernel mode) – preemptable*
 - *Heavier*
- **User thread (aka task):**
 - Stack, instruction counter
 - *Scheduled by run-time (in user mode) – non preemptable*
 - *Lighter*



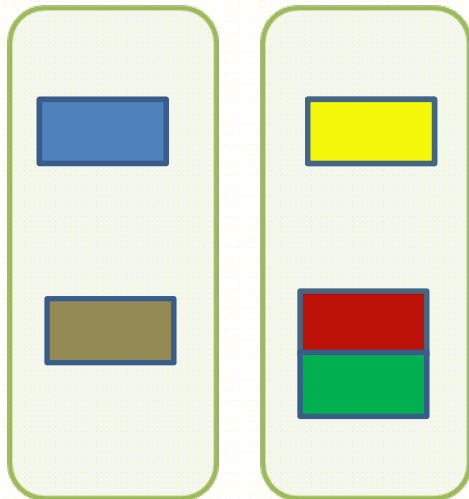
Scheduling Policies for Tasks (1)

- Central queue
 - a thread appends a newly created task to the queue
 - an idle thread pick a runnable task from the queue
 - possible optimizations: allocate small tasks in chunks; use stack, rather than queue (better temporal locality)



Scheduling Policies for Tasks (2)

- Work Stealing
 - a thread appends a newly created task to local queue
 - an idle thread picks a task from its local queue
 - if local queue is empty, it steals a task from another queue



- Less communication than with central queue
- Better locality

Shared-Memory Programming (1)

Programming with Threads: Your language of choice + thread library

- Spawn a thread and specify function to execute
- Wait for a thread to complete
- Synchronize threads
 - Mutual exclusion: locks aka mutexes (lock/unlock)
 - Producer/consumer: condition variables (signal/wait)
- *All variables are shared and accessible by all threads*



Shared-Memory Programming (2)

Programming with threads/tasks:

- Languages with explicit parallelism or parallel frameworks: OpenMP, Java, C#, TBB...
- Simpler synchronization: e.g., atomic sections or atomic updates
- Concurrent objects: concurrent hash tables, concurrent vectors, etc.
- Parallel iterators / parallel loops
- Private variables



OpenMP Example: Counting Occurrences

...

```
int count(int n, int A[n], int val) {  
    int cnt;  
    #pragma omp for reduction(+:cnt)  
    for (i=0; i<n; i++) {  
        cnt += (val == A[i]);  
    }  
    return cnt;  
}
```

Distributed Memory Parallel Programming

Message passing libraries – MPI

- Parallel job = fixed number of processes
- Processes communicate by sending/receiving messages
- Mostly used for compute intensive, in memory problems
- Mostly used on clusters with specialized interconnects (Infiniband, IBM/Cray proprietary networks, etc.)
 - User-space communication



MPI Example: Counting Occurrences

```
\* Broadcast val (from process 0) *\nMPI_Bcast(&val, 1, MPI_INT, 0,\n          MPI_COMM_WORLD);\n\n\* Local count in local array *\nfor(i=0; i<myn; i++)\n    lcnt += (A[i] == val);\n\n\* Sum local counts (to process 0) *\nMPI_Reduce(&lcnt, &gcnt, 1, MPI_INT,\n           MPI_SUM, 0, MPI_COMM_WORLD);
```

Frameworks for Data Intensive Computing

- Large amounts of data – petabytes
- Mostly linear (or $n\log n$) algorithms – e.g. lists of frequent words, reverse directories, sort, etc.
- Coarse grain, out of core
- Cheap, unreliable hardware, in huge quantities
 - Local disks only
- Bottlenecks are
 - disk read/write rate – reduce number of passes
 - network bisection bandwidth – bring work to data, not data to work
 - fault recovery -- reexecute
- CPU is for free



MapReduce Programming Model

- Input: (key, value) pairs of strings
- User provides two functions:
 - *map* (k_1, v_1) $\rightarrow$ *list*(k_2, v_2)
 - *reduce* ($k_2, \text{list}(v_2)$) $\rightarrow$ *list*(v_2)
- System:
 - Applies *map* to all input pairs
 - Sort intermediate result by *key* (and collect together values with same key)
 - Apply *reduce* to all values associated to each key
 - Output result, sorted by key value
- If *map* and *reduce* have no side-effects then execution is deterministic – equivalent to a sequential execution

Dean & Ghemawat, OSDI 2004



Example: Word Count

```
map(String key, String value):  
  // a document is associated with each key  
  for each word w in value:  
    EmitIntermediate(w, *1*);
```

```
reduce(String key, iterator values):  
  int sum = 0;  
  for each v in values:  
    sum += ParseInt(v);  
  Emit(AsString(sum));
```

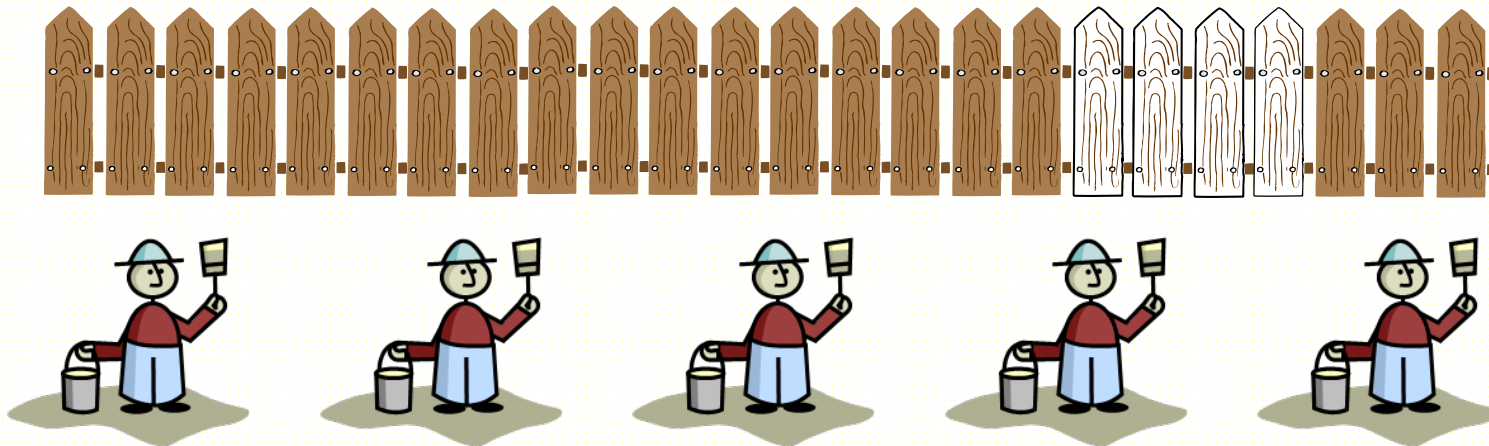


METRICS



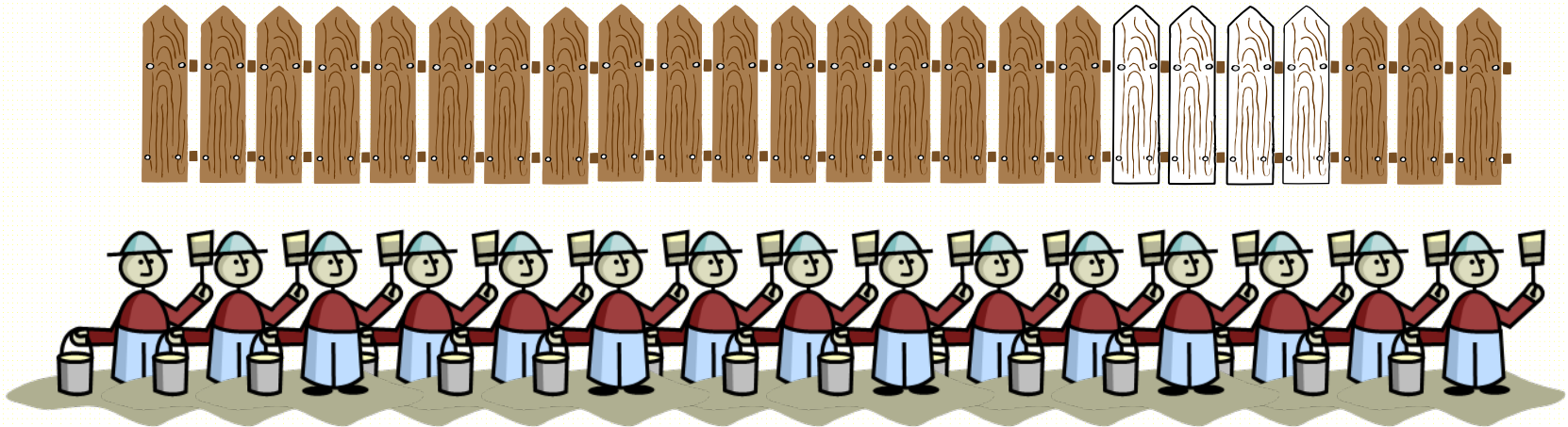
Sometimes Parallelism is Easy

- Painting a fence:
 - $\text{Time} = (\text{picket_painting_time}) * (\# \text{ pickets}) / \# \text{ painters}$
 - Perfect parallelism



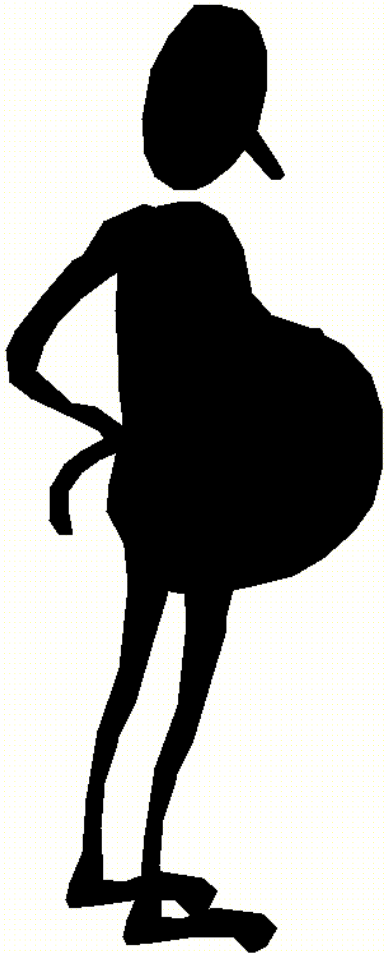
Up To a Limit

- Task granularity cannot be too small



— Too many painters spoil the fence

Sometimes Parallelism Does Not Help



- How many babies do 9 women in one month?

Some Definitions

T_P = Compute time with P HW threads

T_1 – sequential compute time

T_∞ -- compute time with no limitations on #threads = critical path length

$$T_P \geq T_\infty; T_P \geq T_1/P$$

- *Efficient algorithm*: $T_P \sim T_1/P$
 - $T_\infty \ll T_1/P; P \ll T_1/T_\infty$
 - Cannot use efficiently more HW threads than the “average width” of the computation
- Example -- *Amdahl Law*: fraction α of the computation is sequential, $(1-\alpha)$ fully parallel
 - $T_\infty = \alpha T_1$; can use efficiently $\sim 1/\alpha$ processors
 - E.g., 10% of code is sequential -> should not use more than ~ 10 HW threads



Speedup

- Measure of how much faster the computation executes versus the best serial code
 - Serial time divided by parallel time
- Example: *Painting a picket fence*
 - 30 minutes of preparation (serial)
 - One minute to paint a single picket
 - 30 minutes of cleanup (serial)
- Thus, 300 pickets takes 360 minutes (serial time)



Computing Speedup



Number of painters	Time	Speedup
1	$30 + 300 + 30 = 360$	1.0X
2	$30 + 150 + 30 = 210$	1.7X
10	$30 + 30 + 30 = 90$	4.0X
100	$30 + 3 + 30 = 63$	5.7X
Infinite	$30 + 0 + 30 = 60$	6.0X

Amdahl's Law

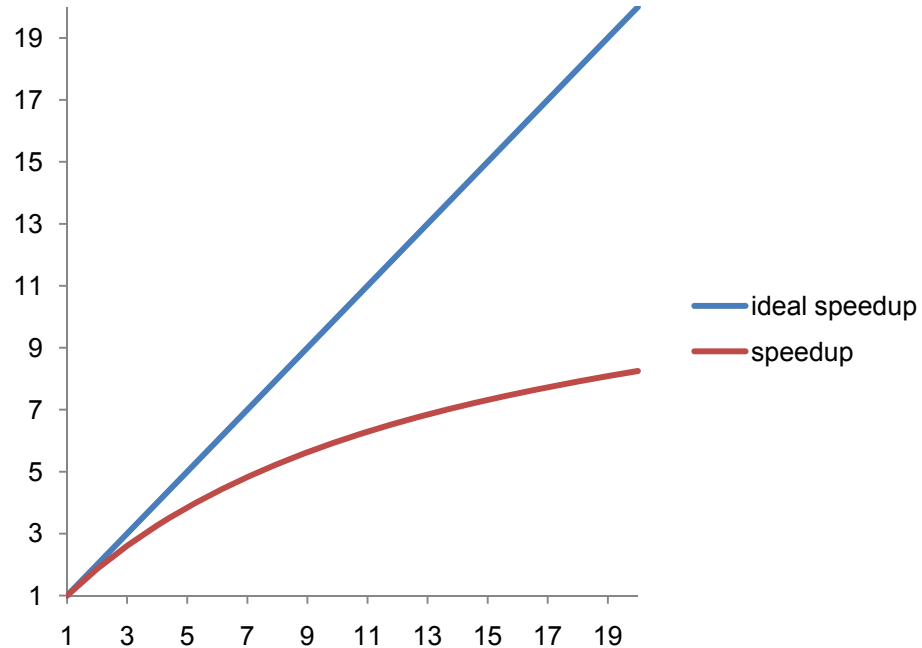
Potential speedup is restricted by serial portion

- $\text{Speedup} = T_p / T_1$
- $\text{Speedup} \leq P$ (P workers reduce time by at most a factor of P)
- $\text{Speedup} \leq T_\infty / T_1$ (Amdahl's law)



Speedup

- T_1/T_P



- Speedup usually is sub-linear and has a plateau
 - how could one have superlinear speedup?

Efficiency



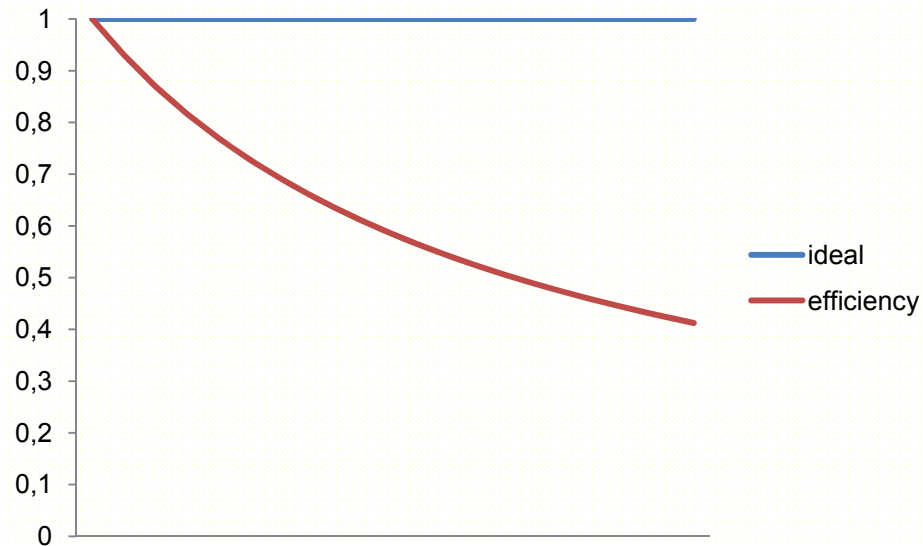
- Measure of how effectively computation resources (threads) are kept busy
 - Speedup divided by number of HW thread

Number of painters	Time	Speedup	Efficiency
1	360	1.0X	100 %
2	$30 + 150 + 30 = 210$	1.7X	85 %
10	$30 + 30 + 30 = 90$	4.0X	40 %
100	$30 + 3 + 30 = 63$	5.7X	5.7 %
<i>Infinite</i>	$30 + 0 + 30 = 60$	6.0X	0 %



Efficiency

- $T_1/(PT_P)$



– Efficiency is <1 and decreasing, usually

WHAT WILL GO WRONG



Pitfalls of Parallel Computing -- Correctness

- Code is nondeterministic – hard to debug
 - *Avoid non-ordering synchronization*
- Code has races (shared memory): a variable is updated by one thread and accessed by another at same time
 - Update may not be atomic
 - Memory accesses may not be in the order you expected

Thread 0

```
flag = 1;  
x = 5;
```

Thread 1

```
if (flag) y=x  
else y=5;
```

- *Use synchronization to avoid races*
 - Lock flag
- Improper synchronization & deadlock
 - *KISS*

Pitfalls of Parallel Computing -- Performance

- Idle resources
 - Because the computation does not have enough “width” – need new algorithm
 - Because resource allocation is not good – need better load balancing
 - (shared memory) Because locks are used too freely – need finer locks or lock-free algorithms
- Too much communication
 - Sharing
 - (shared memory) False sharing
 - Too frequent task migration (need less load balancing)
- Too small granularity
 - Shared memory: >>1K instructions
 - Map-Reduce: >> 1M instructions



Summary

- Discussed where parallel HW is and why, and where it goes
- Described the software stack(s) that supports parallelism
- Introduced standard terminology
- Spoke of pitfalls





Parallel Data Mining

Part II

IJCAI 2011 Tutorial

Anne Laurent

Alexandre Termier

Parallel Data Mining on Multicores

- What is it ?

Design or use of **parallel** data mining algorithms capable of (efficiently) exploiting **multicore** processors

- Where ?
 - Your laptop
 - Your student's desktop
 - You lab's server
 - You university's cluster of multicores

Parallel Data Mining on Multicores

- Why ?
 - To reduce run time of DM program
- Why is reducing run time important ?
 - Answer deferred...

What is Data Mining about ?

- Analyze automatically huge quantities of data
- Answer **reasonably** quickly (*depend on application*)
- **Discover usefull knowledge inside data**
- Problem:
 - Users don't know exactly what they are looking for
 - DM algos gives lots of results / need parameter tuning
- → Usually, DM is an **interactive process**

How can you make many trials and errors with a slow algorithm ?

Need Parallel Data Mining to...

- Have faster algorithms
 - enhance interactive experience
 - better quality of results (with help of user)
- If no user and fixed time:
 - More processing power → more complex analysis, potentially better results
- Better online stream mining algorithms
 - Ex: social networks: streams of evolving graph data

Parallel Data Mining today

User perspective

- Weka ? Not designed for multicore
 - Weka-parallel project:
 - parallelism for cross-validation only
 - targets clusters
- KNIME ? Partly
 - Classification with decision trees
 - Pervasive DataRush plugin
- Research prototypes
 - SIPINA: classification with decision trees
 - PLCM [Negrevergne et al., 10]: parallel closed itemset mining
 - PGLCM [Do et al, 10]: parallel closed gradual pattern mining
 - ParaMiner: *discussed later*
 - ...
- Some softwares, but parallel data mining not yet mainstream

Knime example

Applications Raccourcis Système

Dialog - 2:6 - Decision Tree Learner

File

Options Memory Policy

Class column **S class**

Quality measure **Gini index**

Pruning method **No pruning**

Min number records per node **2**

Number records to store for view **10,000**

☐ Average split point

☐ Binary nominal splits

Max #nominal **10**

Number threads **2**

OK Apply Cancel

00%

project

Interactive Table

Node 2

k-Means

Node 4

Color Manager

Node 3

Scatter Matrix

Node 5

Decision Tree Learner

Node 6

Decision Tree Predictor

Node 7

Node Description

Decision Tree Learner

This node induces a classification decision tree in main memory. The target attribute must be nominal. The other attributes used for decision making can be either nominal or numerical.

Server Workflo...

Workflow Server: **publicse**

Status: not **Connect**

Outline

Console

KNIME Console

Database

Data Manipulation

Data Views

Statistics

Mining

Meta

Misc

Time Series

[[Résolu] Logici...

fr sipina mult...

knime 2.3.1

KNIME

tanagra - logicie...

Images

Parallel Data Mining Problems

- Major tasks in Data Mining:
 - Classification
 - Clustering
 - Pattern Mining
- For each task:
 - Recall its goal
 - Show existing parallel solutions
 - Research challenges with parallelism

Classification

- Goal:
 - **Input:** Set of records labelled with a class (**training set**)
 - **Input:** Set of unlabelled records
 - **Output:** **Class assignment** for the unlabelled records
- Examples:
 - Spam filter
 - Loan decision
 - Legitimacy of credit card transactions
 - Classification of protein secondary structures
 - ...

Major classification techniques

- Decision trees
 - Split the dataset with tests on attributes
 - Leaves give the predicted class
- k -Nearest Neighbors (k -NN)
 - Need a distance metric
 - Each unlabelled record receives the **majority class** of its k nearest neighbors
- SVM
 - Constructs hyperplane(s) separating examples of each class
 - The region of the space where an unlabelled record falls gives its class

Parallel k -NN

- *Embarassingly parallel problem !*

Foreach data point d

Parallelize this loop
Easy to do with OpenMP

Neighbors = []

For each training point t

This loop can also be
parallelized

Dist = distance(d, t)

If |neighbors| < k

then insert(t , neighbors)

Else if (max(neighbors) > dist)

Insert(t , neighbors)

Remove farthest neighbor

In high dimensions, distance
computation can also benefit
from parallelism

Parallel decision trees

- Approach of SIPINA
- Use parallelism to accelerate growth of decision tree
- Function `split(candidate attributes)`: selected attribute

`Max = -infinite`

`For each candidate attribute`

Parallelism opportunity

`Relevance = quality of split(attribute)`

`If (relevance > max) then`

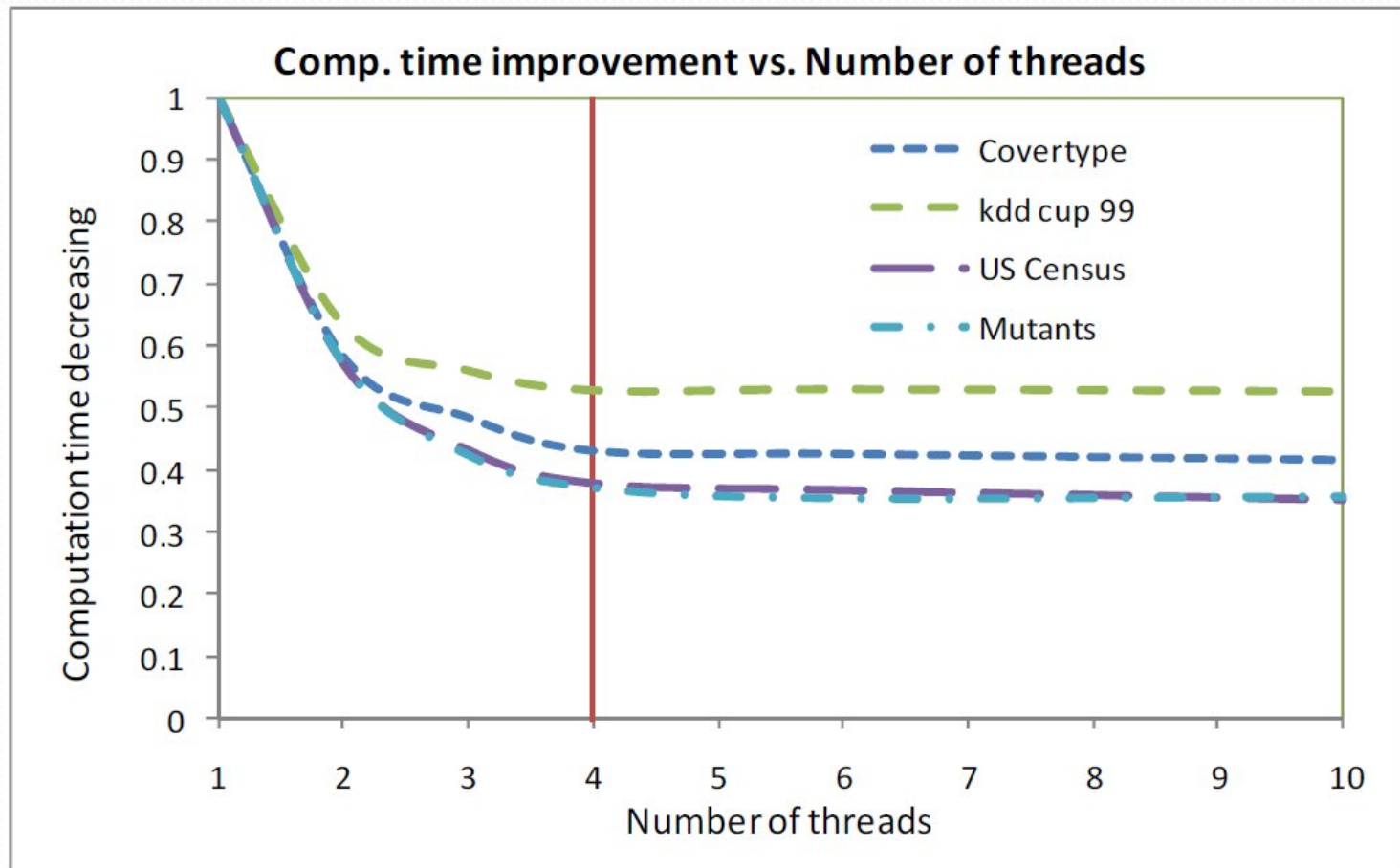
`Max = relevance`

`Selected attribute = attribute`

Some synchronization
may be necessary

`Return selected attribute`

Sipina times (4-cores machine)



Clustering

- Goal:
 - Find groups of objects such that the objects in a group will be similar to one another and different from the objects of other groups
- Examples:
 - Group related text documents / web pages
 - Group patients in a medical study
 - Group genes/proteins having similar functionality
 - ...

Major clustering techniques

- k -means
 - Iteratively find k clusters
 - Based on centroids of clusters of points
- Hierarchical clustering
 - Iteratively agglomerate closest points/clusters
- Density-based clustering
 - Density = number of points within a specified radius

Parallel *k*-means

Function *k*-means

Input: $X = \{x_1, \dots, x_n\}$ (Data to be clustered)

k (Number of clusters)

Output: $C = \{c_1, \dots, c_k\}$ (Cluster centroids)

$m: X \rightarrow C$ (Cluster assignments)

Initialize C (e. g. random selection from X)

While C has changed

For each x_i in X ←

Parallelize here !

$m(x_i) = \operatorname{argmin}_j \text{distance}(x_i, c_j)$

End

For each c_j in C ←

Parallelize here !

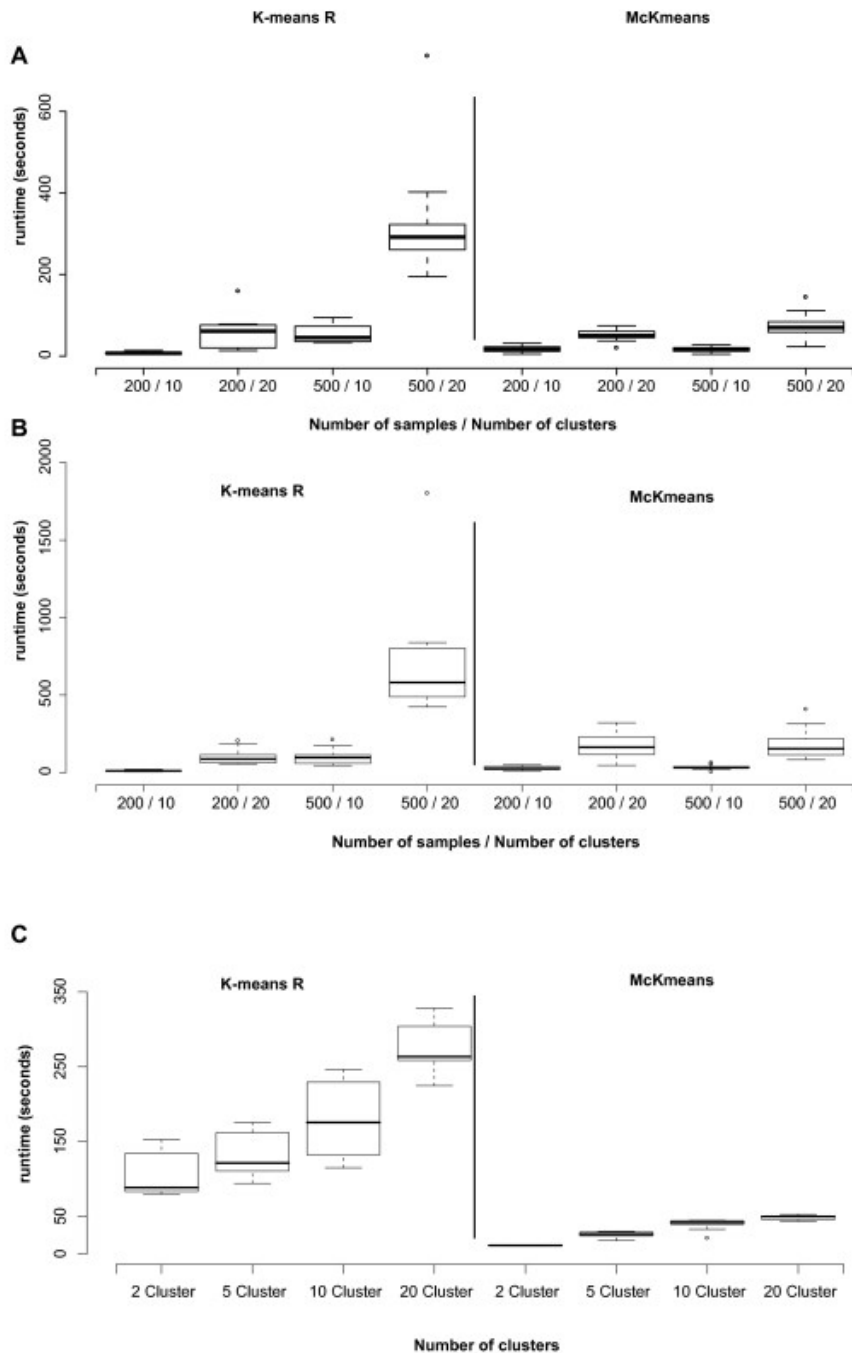
$c_j = \text{centroid}(\{x_i \mid m(x_i) = j\})$

End

End

McKMeans [Kraus, Kestler 2010]

- Algorithm presented before
- By bioinformaticians
- For a biological problem :
 - Cluster gene expression microarray type data
- Use Software Transactional Memory
- Implementation in **Clojure**
 - runs on the Java Virtual Machine



- Experiments on real data
- Dual quad-core Intel Xeon 3.2 GHz and 32 GB RAM.
- **A**: artificial microarray data set with 50000 genes, 200/500 arrays, and 10/20 clusters
- **B**: same as A with 100000 genes
- **C**: real data [Smirnov et al.], 22277 genes, 465 cell lines
- R implementation: single core

Pattern Mining

- Goal:
Extract (frequently occurring) patterns from huge volumes of data.
- Applications:
 - Market basket analysis
 - Chemistry, especially drug design
 - Bioinformatics
 - Log analysis
 - ...
- **Most computationally expensive data mining problem**
 - Combinatorial explosion in search space size

Some famous algorithms

- Apriori (frequent itemsets)
 - Pioneer algorithm
 - Breadth first
 - Pedagogical interest only, *NEVER USE IT IN PRACTICE !*
- FP-Growth (frequent itemsets)
 - Clever handling of database in FP-trees
 - Depth first search
- LCM (closed frequent itemsets)
 - Fastest itemset miner
 - Depth first
- gSpan (frequent subgraphs)
 - Depth first

Parallel pattern mining

- Pattern mining has the most need for parallelism
 - Run times can span days even on medium sized datasets
- Parallel pattern mining algorithms are the most difficult to write
- Why is it difficult ? How to solve it anyway ?
 - Benjamin's talk
 - Shirish's talk
 - *Your future research ?*



Benjamin's talk

What about GPUs ?

- Some algorithms implemented
 - Apriori (x 10 speedup)
 - k -means (x 40 speedup)
 - k -nn (x 120 speedup)
- Algorithms with regular computations better candidates for using GPU
 - *Research: how to make regular irregular algorithms ?*
 - Ex: for Apriori, heavy use of bitmaps
- Slow data transferts between main memory and GPU
 - Avoid too many transferts
 - Keep as much computation as possible in the GPU
 - Bulk transferts, even at the price of idle threads

Conclusion

- Multicore processors are here to stay
- Data mining algorithms need them for performance
- Some data mining algorithms can easily be parallelized
- Many others need more involvement
 - →Interesting research !
 - Understand parallelism
 - Better understand memory system
 - Better understand data mining problem at hand
 - Especially: think about the structure of your search space

Perspectives

- Multicore/GPU data mining still in its infancy
 - Many opportunities for young researchers !
- Propose efficient parallelizations of existing algorithms
 - Parallel data mining framework ?
- Improve parallel scalability of algorithms
 - High-level modification of algorithms
 - Low-level: better use of cache hierarchies
- Natively parallel algorithms ?

Bibliography (partial)

- Introduction to data mining, Tan, Steinbach, Kumar, 2004
- A survey of parallel algorithms for classification, V. Satuluri, 2007
- Porting decision tree algorithms to multicores using FastFlow, M. Aldinucci, S. Ruggieri, M. Torquati, PKDD 2010
- A highly efficient multi-core algorithm for clustering extremely large datasets, J.M. Kraus, H.A. Kestler, 2010, BMC Bioinformatics
- <http://eric.univ-lyon2.fr/~ricco/sipina.html>
- Mapping Data Mining Algorithms on a GPU Architecture: A Study, A. Gainaru, E. Slusanschi, S. Trausan-Matu, ISMIS 2011
- Frequent Itemset Mining on Graphics Processors, W. Fang, M. Lu, X. Xiao, B. He, Q. Luo, DaMoN 2009

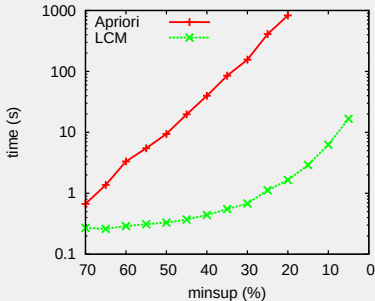
How to design high end parallel pattern mining algorithms

Benjamin Négrevergne

The case of itemset mining

■ Apriori (1993) VS LCM (2004)

- ▶ **Apriori** :
R. Agrawal,
implem :
C. Borgelt
- ▶ **LCM** : LCM2.5
T. Uno
- ▶ dataset : connect

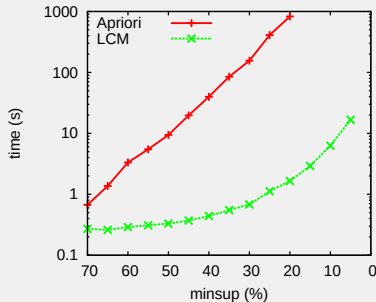


The case of itemset mining

■ Apriori (1993) VS LCM (2004)

- ▶ **Apriori** :
R. Agrawal,
implem :
C. Borgelt
- ▶ **LCM** : LCM2.5
T. Uno
- ▶ dataset : connect

⇒ LCM : 3 orders of magnitude faster



Recipe to build an efficient pattern mining algorithm

- Equivalence classes (Eclat) [Zaki et al., 1997]

Recipe to build an efficient pattern mining algorithm

- ▶ Equivalence classes (Eclat) [Zaki et al., 1997]
- ▶ Closed pattern mining (A-Closed) [Pasquier et al., 1999]

Recipe to build an efficient pattern mining algorithm

- ▶ Equivalence classes (Eclat) [Zaki et al., 1997]
- ▶ Closed pattern mining (A-Closed) [Pasquier et al., 1999]
- ▶ Database reduction (FP-Growth) [Han et al., 2000]

Recipe to build an efficient pattern mining algorithm

- ▶ Equivalence classes (Eclat) [Zaki et al., 1997]
- ▶ Closed pattern mining (A-Closed) [Pasquier et al., 1999]
- ▶ Database reduction (FP-Growth) [Han et al., 2000]

- ▶ Parallelism ?

Recipe to build an efficient pattern mining algorithm

- ▶ Equivalence classes (Eclat) [Zaki et al., 1997]
- ▶ Closed pattern mining (A-Closed) [Pasquier et al., 1999]
- ▶ Database reduction (FP-Growth) [Han et al., 2000]

- ▶ Parallelism ?

Although these are **fundamental improvements** to obtain efficiency very few algorithms combine these improvements into a parallel algorithm.

⇒ **Why ?**

Recipe to build an efficient pattern mining algorithm

- ▶ Equivalence classes (Eclat) [Zaki et al., 1997]
- ▶ Closed pattern mining (A-Closed) [Pasquier et al., 1999]
- ▶ Database reduction (FP-Growth) [Han et al., 2000]

- ▶ Parallelism ?

Although these are **fundamental improvements** to obtain efficiency very few algorithms combine these improvements into a parallel algorithm.

⇒ **Why ?**

State of the art improvements don't play nice with parallelism.

⇒ **What can we do about it ?**

Exploring the search space

Conquering the dataset

Structured search space

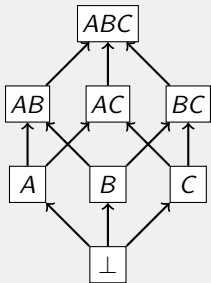
Any pattern mining problem deals with a **structured** search space :
small patterns $\subset$ bigger patterns

Structured search space

Any pattern mining problem deals with a **structured** search space :
small patterns $\subset$ bigger patterns

patterns + inclusion relation $\Rightarrow$ DAG

Itemset Mining

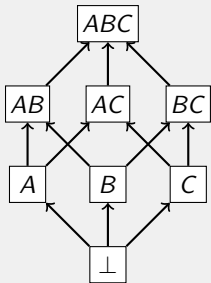


Structured search space

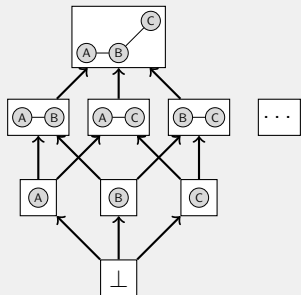
Any pattern mining problem deals with a **structured** search space :
small patterns $\subset$ bigger patterns

patterns + inclusion relation $\Rightarrow$ DAG

Itemset Mining



Graph Mining

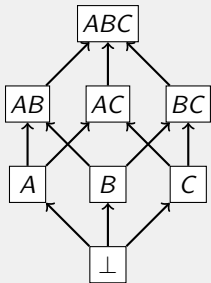


Structured search space

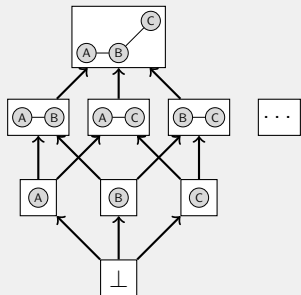
Any pattern mining problem deals with a **structured** search space :
small patterns $\subset$ bigger patterns

patterns + inclusion relation $\Rightarrow$ DAG

Itemset Mining



Graph Mining



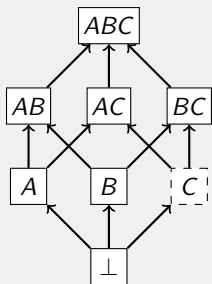
Properties are monotonous along paths in the DAG. eg. frequency

Structured search space

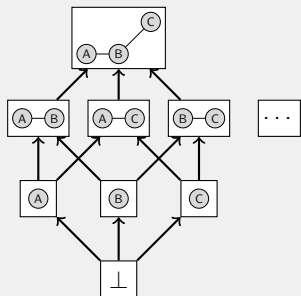
Any pattern mining problem deals with a **structured** search space :
small patterns $\subset$ bigger patterns

patterns + inclusion relation $\Rightarrow$ DAG

Itemset Mining



Graph Mining



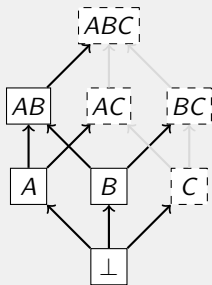
Properties are monotonous along paths in the DAG. eg. frequency

Structured search space

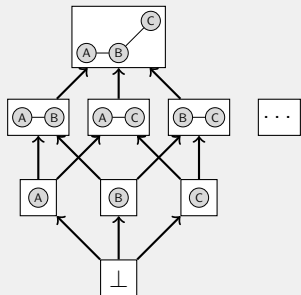
Any pattern mining problem deals with a **structured** search space :
small patterns $\subset$ bigger patterns

patterns + inclusion relation $\Rightarrow$ DAG

Itemset Mining



Graph Mining



Properties are monotonous along paths in the DAG. eg. frequency

Enumeration strategies are required to explore the DAG

- ▶ Without omitting patterns or generating duplicates
- ▶ Without testing too many candidate patterns.

■ Breadth First Search

- ▶ Require communications and synchronization
- ▶ Require storage for large amount of patterns

■ Depth First Search

- ▶ Prone to irregular computations

How to explore the DAG in parallel

■ DFS ?

- ▶ Natively Divide & Conquer
- ▶ Recursive algorithms are widely supported by parallel environments (Charm++, Cilk, IntelTBB, ...)
- ▶ Branch wise division matches dataset division
- ▶ Fastest algorithms are DFS (LCM, DCI, gSpan, ...)

■ Equivalence class

Defined by [Zaki, 1997] (itemsets) such that :

Each pattern has a **unique parent** in its equivalence class.

■ Equivalence class

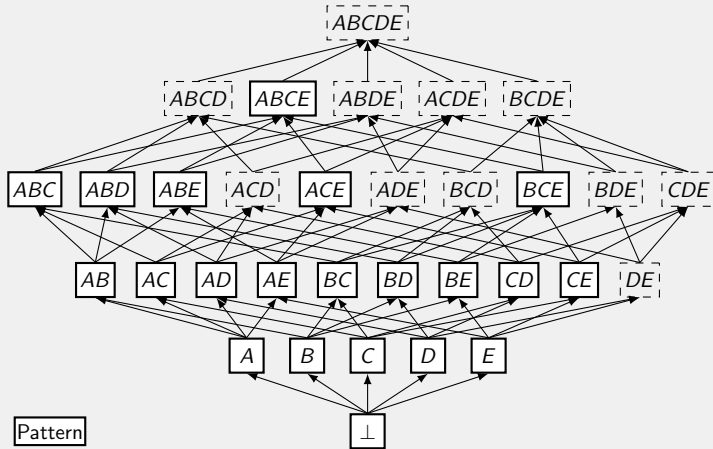
Defined by [Zaki, 1997] (itemsets) such that :

Each pattern has a **unique parent** in its equivalence class.

- ▶ Itemset mining : prefix classes

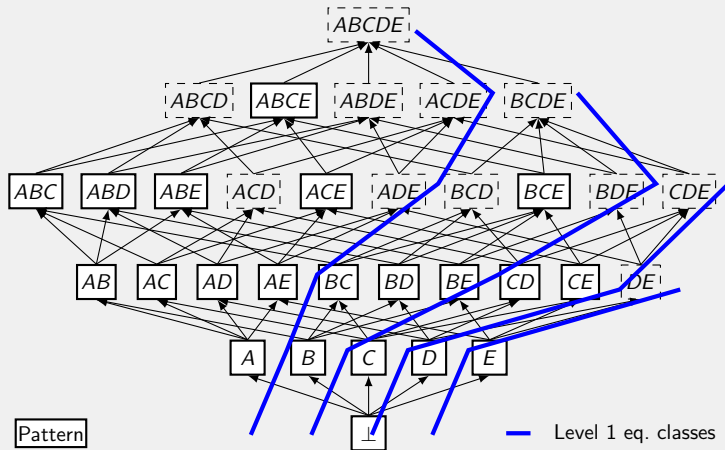
Enumeration Tree

Prefix based equivalence class in itemset mining



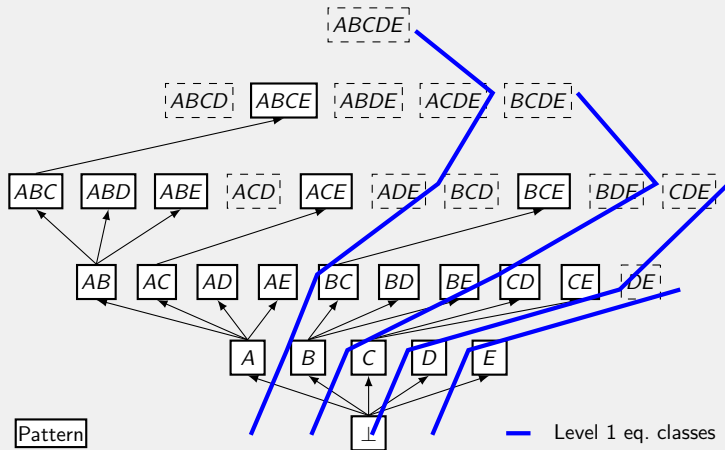
Enumeration Tree

Prefix based equivalence class in itemset mining



Enumeration Tree

Prefix based equivalence class in itemset mining



■ Equivalence class

Defined by [Zaki, 1997] (itemsets) such that :

Each pattern has a **unique parent** in its equivalence class.

- ▶ Itemset mining : prefix classes

■ Equivalence class

Defined by [Zaki, 1997] (itemsets) such that :

Each pattern has a **unique parent** in its equivalence class.

- ▶ Itemset mining : prefix classes
- ▶ Tree mining : right most expansion (TreeMiner [Zaki, 2005])

■ Equivalence class

Defined by [Zaki, 1997] (itemsets) such that :

Each pattern has a **unique parent** in its equivalence class.

- ▶ Itemset mining : prefix classes
- ▶ Tree mining : right most expansion (TreeMiner [Zaki, 2005])
- ▶ Graph mining : DFS code (gSpan Yan and Han, 2002)

■ Equivalence class

Defined by [Zaki, 1997] (itemsets) such that :

Each pattern has a **unique parent** in its equivalence class.

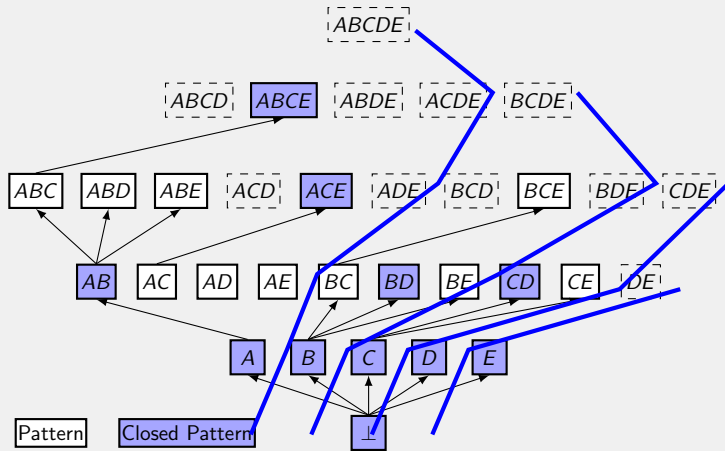
- ▶ Itemset mining : prefix classes
- ▶ Tree mining : right most expansion (TreeMiner [Zaki, 2005])
- ▶ Graph mining : DFS code (gSpan Yan and Han, 2002)

Spawn a task at each node of the spanning tree

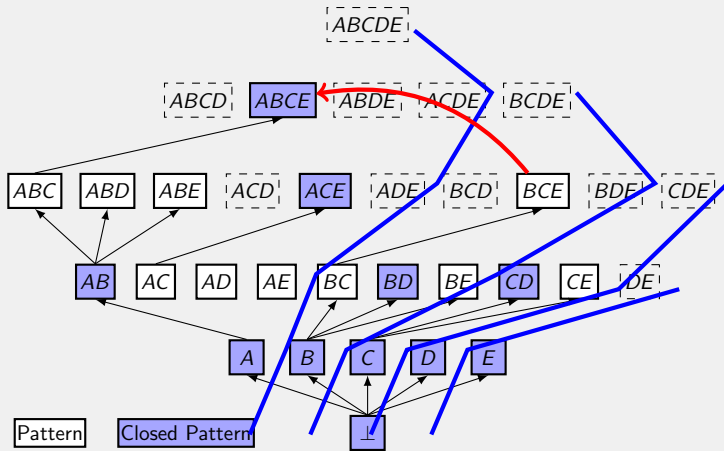
⇒ no communication/synchronization required

- ▶ ParEclat [Zaki, 1997]

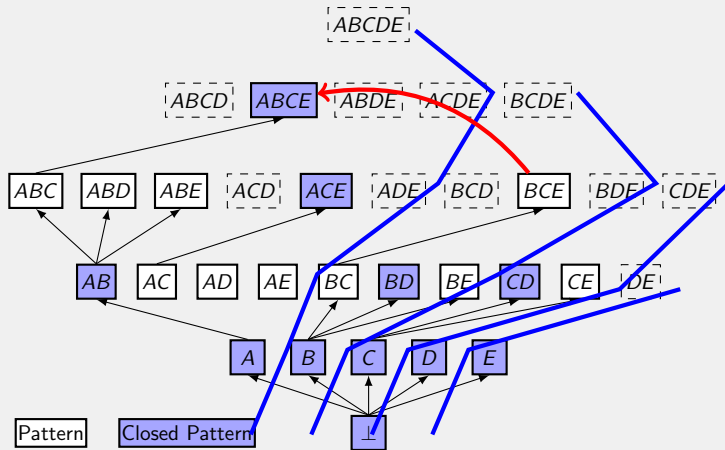
Enumerating Closed Itemsets



Enumerating Closed Itemsets



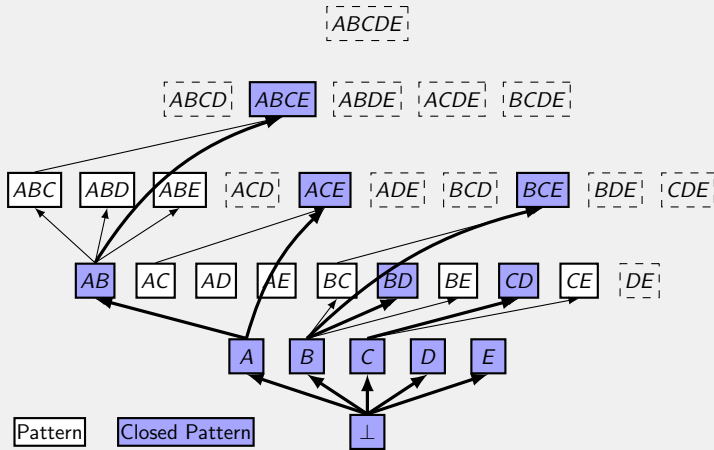
Enumerating Closed Itemsets



Problem :

Closed pattern mining breaks with naive DFS enumeration strategy

Enumerating Closed Itemsets



Problem :

$|\text{patterns}| \gg |\text{closed patterns}|$

■ Discovering a tree spanning the closed patterns

Each thread : « Should I explore this path ? »

■ Discovering a tree spanning the closed patterns

Each thread : « Should I explore this path ? »

Flawed approach : Storing the set of closed patterns into the memory

- ▶ Non determinism or heavy synchronization
- ▶ Implies communication among the branches
- ▶ No I/O / Computation overlapping
- ▶ Exponential space complexity

■ Discovering a tree spanning the closed patterns

Each thread : « Should I explore this path ? »

Flawed approach : Storing the set of closed patterns into the memory

- ▶ Non determinism or heavy synchronization
- ▶ Implies communication among the branches
- ▶ No I/O / Computation overlapping
- ▶ Exponential space complexity

⇒ Inefficient and inadequate to parallel exploration

Parallel exploration of the search space?

- ▶ Specify a tree spanning the closed patterns
- ▶ Parallelize the exploration

Parallel exploration of the search space?

- ▶ Specify a tree spanning the closed patterns
- ▶ Parallelize the exploration

A Generic Approach ? :

⇒ ParaMiner is a parallel closed pattern mining algorithm build on a **generic polyspace enumeration strategy**.

So far it is able to mine

- ▶ frequent itemsets, connected relational graphs, periodic patterns, gradual patterns ...

Parallel exploration of the search space?

- ▶ Specify a tree spanning the closed patterns
- ▶ Parallelize the exploration

A Generic Approach ? :

⇒ ParaMiner is a parallel closed pattern mining algorithm build on a **generic polyspace enumeration strategy**.

So far it is able to mine

- ▶ frequent itemsets, connected relational graphs, periodic patterns, gradual patterns ...

Very to irregular computations ⇒ Load unbalance.

Exploring the search space

Conquering the dataset

Dataset reduction

Testing the patterns is a costly task that requires access to a large dataset.

Most of the dataset is **irrelevant** or **redundant** to a given pattern

Dataset reduction

Testing the patterns is a costly task that requires access to a large dataset.

Most of the dataset is **irrelevant** or **redundant** to a given pattern

Given a closed pattern P :

- ▶ Build $\mathcal{DS}_P$, the **reduced dataset** relevant to P and its branch
- ▶ Test the super-patterns of $\mathcal{DS}_P$

$\mathcal{DS}_\perp, ||\mathcal{DS}_\perp|| = 15$

A B

A C

A B C D F

A B C D

B C

$\mathcal{DS}_{BC}$

?

Dataset reduction

Testing the patterns is a costly task that requires access to a large dataset.

Most of the dataset is **irrelevant** or **redundant** to a given pattern

Given a closed pattern P :

- ▶ Build $\mathcal{DS}_P$, the **reduced dataset** relevant to P and its branch
- ▶ Test the super-patterns of $\mathcal{DS}_P$

$\mathcal{DS}_\perp, ||\mathcal{DS}_\perp|| = 15$

A B
A C
A B C D F
A B C D
B C



$\mathcal{DS}_{BC}$

*Remove transaction
not including BC*

Dataset reduction

Testing the patterns is a costly task that requires access to a large dataset.

Most of the dataset is **irrelevant** or **redundant** to a given pattern

Given a closed pattern P :

- ▶ Build $\mathcal{DS}_P$, the **reduced dataset** relevant to P and its branch
- ▶ Test the super-patterns of $\mathcal{DS}_P$

$\mathcal{DS}_\perp, ||\mathcal{DS}_\perp|| = 15$

A B
A C
A B C D F
A B C D
B C



$\mathcal{DS}_{BC}$

Remove items
infrequent with BC

Dataset reduction

Testing the patterns is a costly task that requires access to a large dataset.

Most of the dataset is **irrelevant** or **redundant** to a given pattern

Given a closed pattern P :

- ▶ Build $\mathcal{DS}_P$, the **reduced dataset** relevant to P and its branch
- ▶ Test the super-patterns of $\mathcal{DS}_P$

$\mathcal{DS}_\perp, ||\mathcal{DS}_\perp|| = 15$

A B
A C
A B C D F
A B C D
B C



$\mathcal{DS}_{BC}$

*Remove non-closed
items from different
equivalence classes*

Dataset reduction

Testing the patterns is a costly task that requires access to a large dataset.

Most of the dataset is **irrelevant** or **redundant** to a given pattern

Given a closed pattern P :

- ▶ Build $\mathcal{DS}_P$, the **reduced dataset** relevant to P and its branch
- ▶ Test the super-patterns of $\mathcal{DS}_P$

$\mathcal{DS}_\perp, ||\mathcal{DS}_\perp|| = 15$

A B

A C

A B C **D** F

A B C **D**

B C



$\mathcal{DS}_{BC}$

Remove BC itself

Dataset reduction

Testing the patterns is a costly task that requires access to a large dataset.

Most of the dataset is **irrelevant** or **redundant** to a given pattern

Given a closed pattern P :

- ▶ Build $\mathcal{DS}_P$, the **reduced dataset** relevant to P and its branch
- ▶ Test the super-patterns of $\mathcal{DS}_P$

$\mathcal{DS}_\perp, ||\mathcal{DS}_\perp|| = 15$

A B

A C

A B C D F (weight=2)

A B C D

B C



$\mathcal{DS}_{BC}$

Remove redundant transaction (add weighted transaction)

Dataset reduction

Testing the patterns is a costly task that requires access to a large dataset.

Most of the dataset is **irrelevant** or **redundant** to a given pattern

Given a closed pattern P :

- ▶ Build $\mathcal{DS}_P$, the **reduced dataset** relevant to P and its branch
- ▶ Test the super-patterns of $\mathcal{DS}_P$

$\mathcal{DS}_\perp, ||\mathcal{DS}_\perp|| = 15$

A B

A C

A B C D F (weight=2)

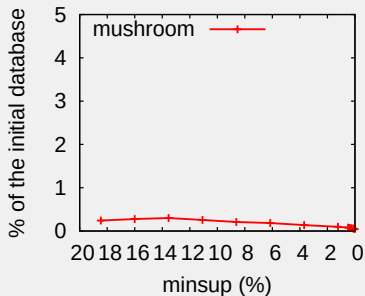
A B C D

B C

$\Rightarrow$

$\mathcal{DS}_{BC}$

D (weight=2)



Pattern testing is performed in a reduced dataset that is in average **99.5% smaller** than the original dataset.

■ Drawback :

Large number of dataset creation/deletion.

Issues related to dataset creation/deletion

- ▶ Large number of memory allocations

⇒ contention ?

Issues related to dataset creation/deletion

- ▶ Large number of memory allocations

⇒ contention ?

- ▶ Poor cache locality

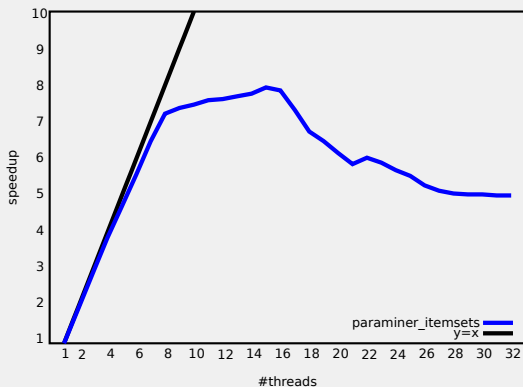
- ▶ Higher $\frac{\text{memory instructions}}{\text{other instructions}}$ ratio

- ▶ Bus traffic increase

⇒ contention !

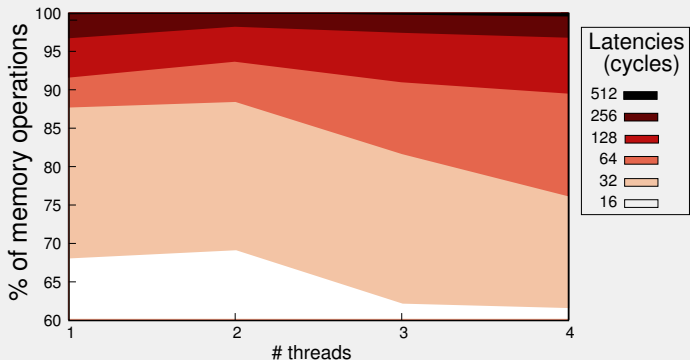
Impact (2)

Itemset mining (connect dataset) : speedup VS #cores



Impact (2)

Itemset mining (connect dataset) : speedup VS #cores



Problem

Memory operations are delayed $\Rightarrow$ poor #instruction/s

How to reduce the memory footprint

■ By hacking the algorithm

- ▶ Involve the programmer (cf. S. Tatikonda's talk)

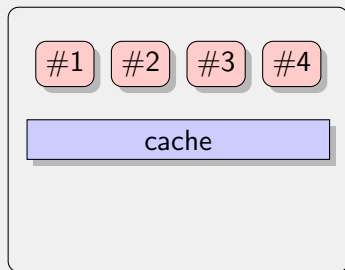
How to reduce the memory footprint

■ By hacking the algorithm

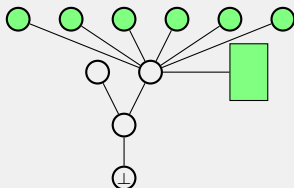
■ By exploiting tasks memory affinities

close collaboration : architecture / application

architecture



application

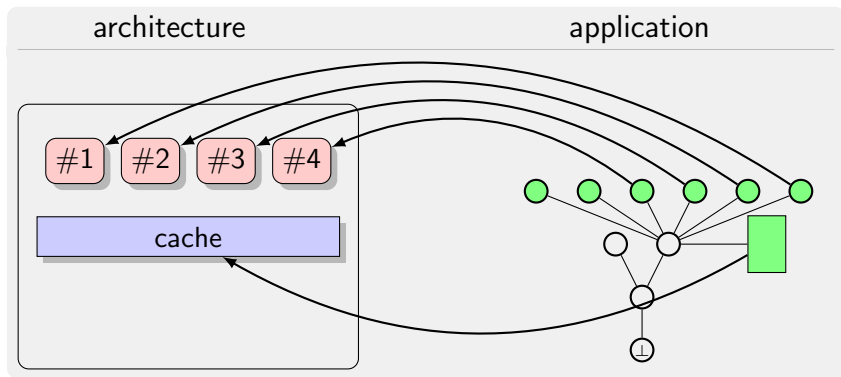


How to reduce the memory footprint

■ By hacking the algorithm

■ By exploiting tasks memory affinities

close collaboration : architecture / application



Pattern mining on modern architectures

Pattern mining algorithms are **Memory consuming**

Modern multicore architectures are :

- ▶ Hierarchical : cores / processors / NUMA nodes
- ▶ Dissimilar and unknown at compile time

Pattern mining on modern architectures

Pattern mining algorithms are **Memory consuming**

Modern multicore architectures are :

- ▶ Hierarchical : cores / processors / NUMA nodes
- ▶ Dissimilar and unknown at compile time

We need an **machine-adaptive** approach to drive the parallel exploration of the search space.

Pattern mining on modern architectures

Pattern mining algorithms are **Memory consuming**

Modern multicore architectures are :

- ▶ Hierarchical : cores / processors / NUMA nodes
- ▶ Dissimilar and unknown at compile time

We need an **machine-adaptive** approach to drive the parallel exploration of the search space.

Melinda : a work sharing tool to evaluate parallel exploration strategies

- ▶ PLCM[Negrevergne et al., 2010]
- ▶ PGLCM[Trong Dinh et al., 2010]
- ▶ ParaMiner

⇒ Best strategy 10% perf. increase

Naive parallelization of high end pattern mining algorithms :
Performs poorly because,

- ▶ Communication and synchronization
- ▶ Irregular computations
- ▶ High memory / bus bandwidth consumption

Most of these problem can be addressed in a had-oc way.

Naive parallelization of high end pattern mining algorithms :
Performs poorly because,

- ▶ Communication and synchronization
- ▶ Irregular computations
- ▶ High memory / bus bandwidth consumption

Most of these problem can be addressed in a had-oc way.

On going work PARAMINER : A generic and parallel algorithm for pattern mining.

Naive parallelization of high end pattern mining algorithms :
Performs poorly because,

- ▶ Communication and synchronization
- ▶ Irregular computations
- ▶ High memory / bus bandwidth consumption

Most of these problem can be addressed in a had-oc way.

On going work PARAMINER : A generic and parallel algorithm for pattern mining.

A **unified** approach for pattern mining ?

Thanks

...

Efficient Frequent Subtree Mining on Multicores

Shirish Tatikonda

IBM Almaden Research Center, USA

Presented By

This work has been supported by a research award from Google and the following NSF grants: CAREER-IIS-034662; RI-CNS-0403342; CCF-0702587; AND IIS-0917070.

Outline

❑ Introduction

- Problem definition
- Challenges

❑ Proposed Approach

- Data transformations
- Memory optimizations
 - Sequence-based subtree matching
 - Computation chunking
- Parallel algorithms
 - Adaptive load balancing

❑ Conclusions

Types of Subtrees

□ Induced subtree

- Preserves parent-child relationships



□ Embedded subtree

- Preserves ancestor-descendant relationships
- Every induced subtree is an embedded subtree

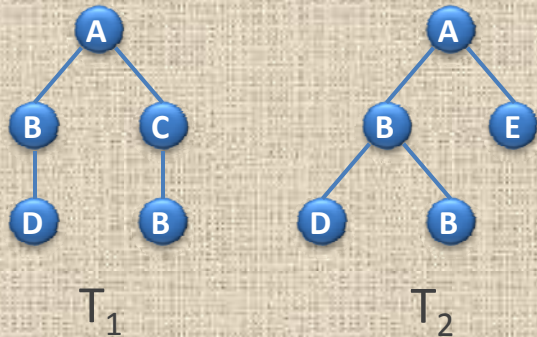


Problem Definition

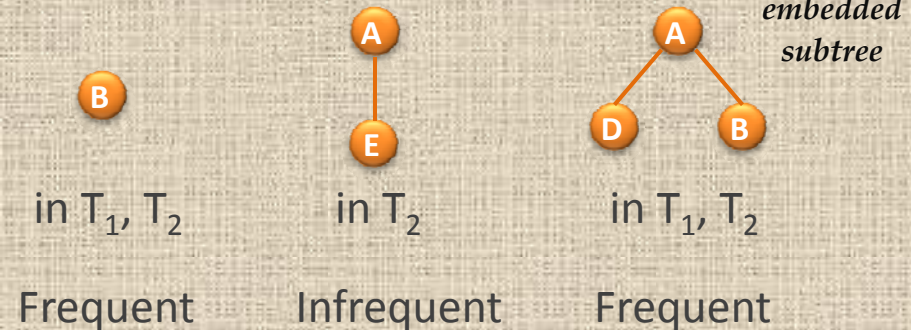
A subtree S is frequent if $\text{support}(S) \geq \theta$

$$\text{Support}(S) = \frac{\text{freq}(S)}{|D|} \begin{array}{l} \longrightarrow \text{Number of trees in which } S \text{ appears as a subtree} \\ \longrightarrow \text{Number of trees in the database} \end{array}$$

Example database



Sample patterns ($\theta = 2$)



Given a database of trees and θ , enumerate *all frequent embedded* subtrees

Applications

❑ Bioinformatics

- Classification and function prediction of new biological molecules

❑ Software engineering

- Identification of sub-routines to be optimized
- Finding abnormal function calling & communication patterns

❑ XML Indexing

- Designing effective caching policies

❑ Web analytics

- Website reorganization and personalization

❑ Computational linguistics

- Automatic language parser generation

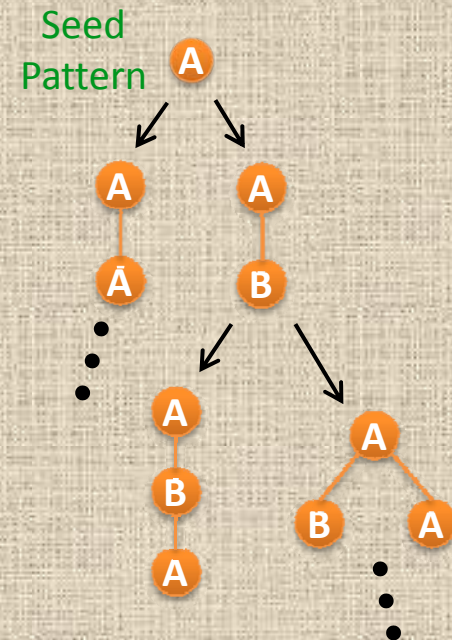
Overview

❑ Two primary steps

- Candidate subtree generation
 - Generate all possible candidate subtrees
 - *Challenge*: search space traversal
- Support counting
 - Evaluate each candidate for their frequency
 - *Challenge*: subtree isomorphism

❑ A recursive *pattern-growth* approach

- Start with a seed pattern (a single node)
- Repeatedly grow the pattern by adding nodes (pattern extension)
 - This step corresponds to search space traversal
- Evaluate the frequency of the generated pattern



```
1. Pattern_mine(P)
2. loop
3.   sup = find_frequency(P)
4.   if sup ≥  $\theta$ 
5.     P' = grow P with a new node
6.     Pattern_mine(P')
```

How can we do it efficiently?

Mining on Multicores – Challenges

- Candidate generation

- Exponential search space

} Sequence-based Approach

- Irregular in nature

- Hard to estimate the access pattern
- Multicores typically have shared caches
- Must maintain **good data locality**

- Memory intensive

(due to large meta-data structures)

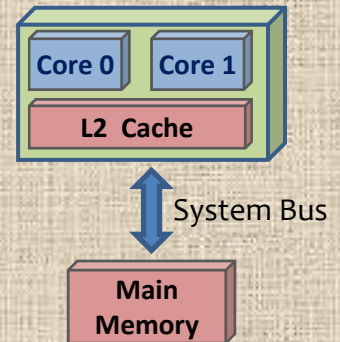
- Latency to main memory
- Pressure on memory bandwidth
- Must maintain **small memory footprints** and working sets

} Memory Optimizations

- Workload estimation is difficult

- Data and parameter dependent tasks
- Must maintain **good load balance** among cores

} Load-balancing Strategies



Outline

□ Introduction

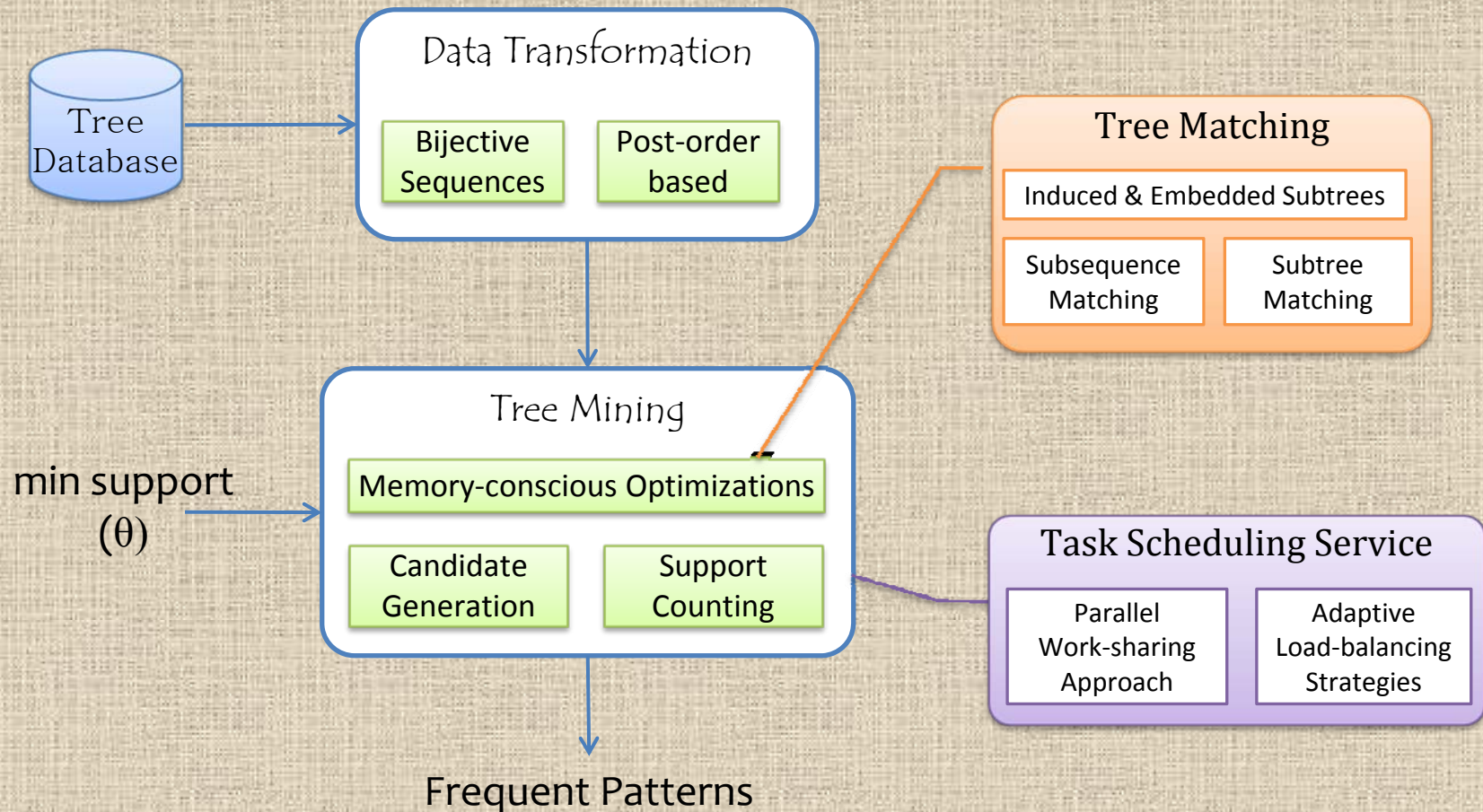
- Problem definition
- Challenges

□ Proposed Approach

- **Data transformations**
- Memory optimizations
 - Sequence-based subtree matching
 - Computation chunking
- Parallel algorithms
 - Adaptive load balancing

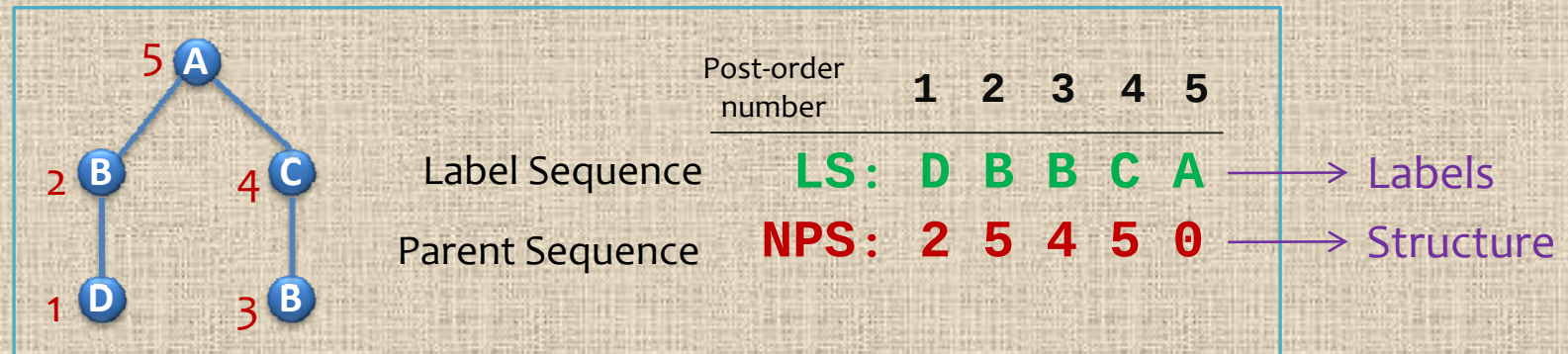
□ Conclusions

Approach



Data Transformation

- ❑ Trees are represented as sequences
- ❑ Prufer sequences
 - Traverse the tree in post-order
 - LS – Node labels in post-order
 - NPS – Parent numbers in post-order



- + Non-redundant & complementing sequences
- + Linear time construction
- + Simple array based data structures

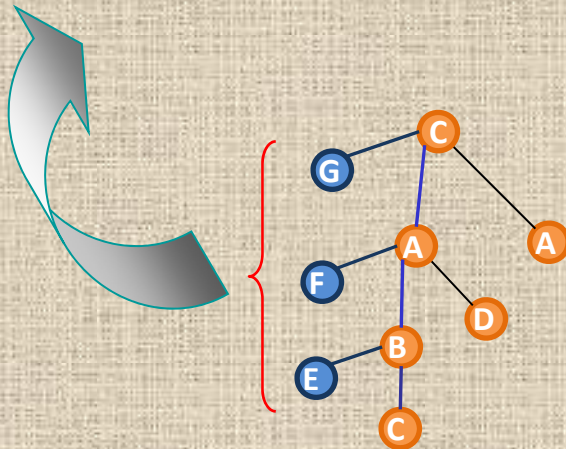
Sequence-based non-redundant candidate generation

Sequence-based Candidate Generation

- ❑ Traverse the search space of subtrees by operating entirely on sequences

Label Sequence

G F E C B D A A C

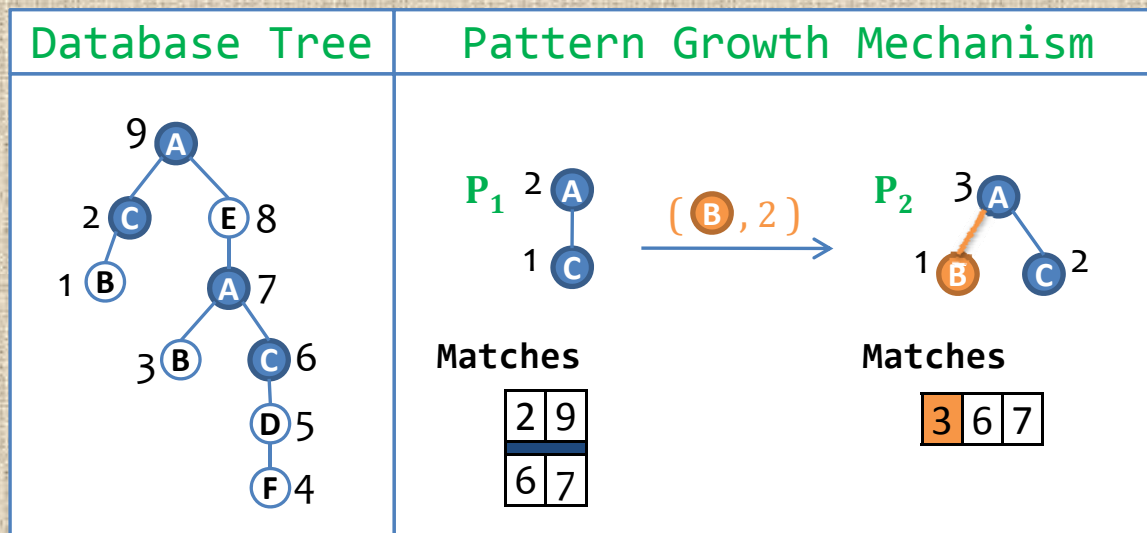


Growth on the left most path
corresponds to
prefixes of Prüfer Sequence

Generate new patterns by adding nodes on the left most path

Support Counting – Two Methods

- 1) Execute a subtree matching algorithm (for every candidate)
- 2) Employ meta-data structures (*state-of-the-art*)
 - Known as *embedding lists*
 - Store all matches of a pattern
 - Trade space for improved execution time

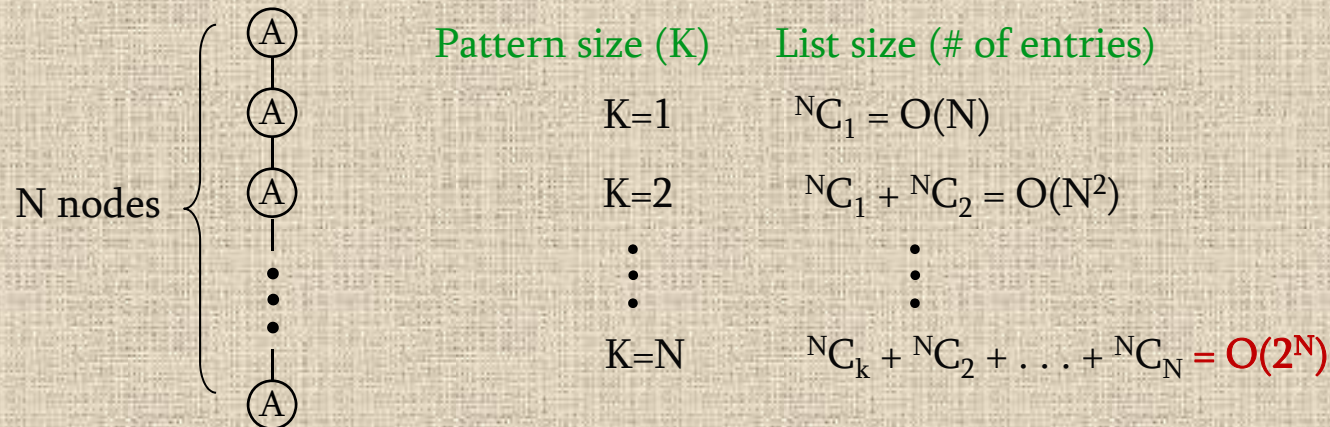


Matches of larger patterns can be found by searching the neighborhood of smaller patterns

Problem with Embedding Lists

- ❑ Store all occurrences of the pattern that is being mined
- ❑ Can become very large
 - In a worst case, lists can be *exponential in tree size*
- ❑ Results in huge memory footprints

Worst case Scenario



How can we improve the memory performance ?

Outline

□ Introduction

- Problem definition
- Challenges

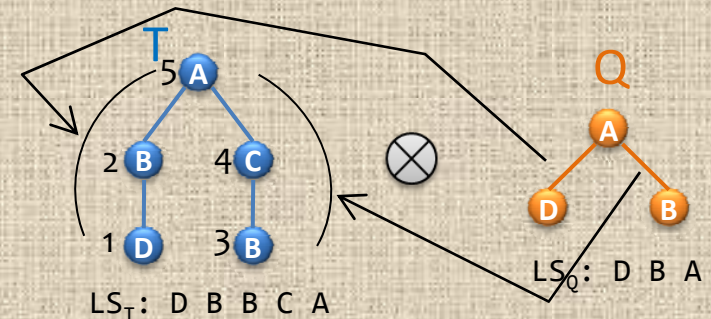
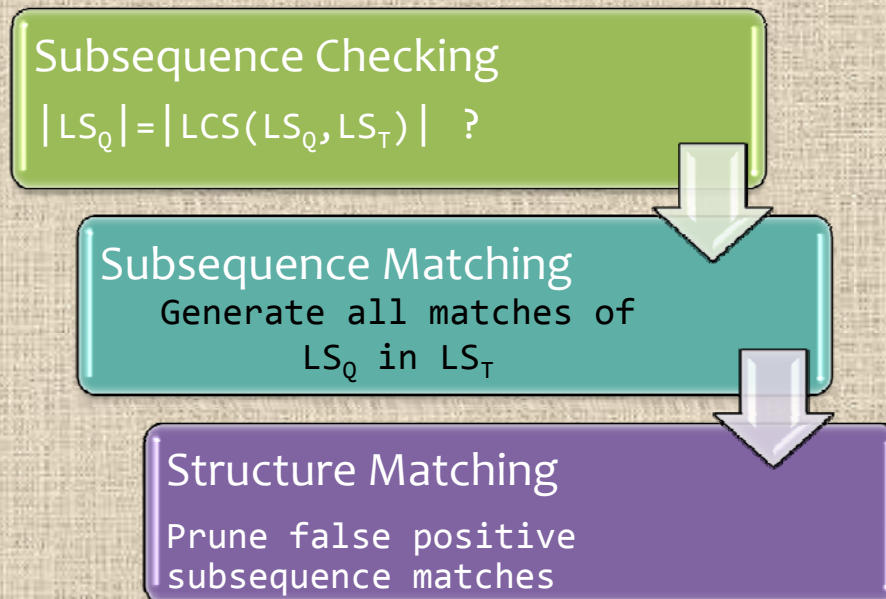
□ Proposed Approach

- Data transformations
- **Memory optimizations**
 - Sequence-based subtree matching
 - Computation chunking
- Parallel algorithms
 - Adaptive load balancing

□ Conclusions

Memory Optimizations – Subtree Matching

- ❑ Avoid large meta-data structures by constructing the lists *on-the-fly*
- ❑ Subtree matching is transformed into subsequence matching
 - If Q is a *sub tree* of T then LS_Q is a *subsequence* of LS_T
 - Verified by computing the length of Longest Common Subsequence (LCS)



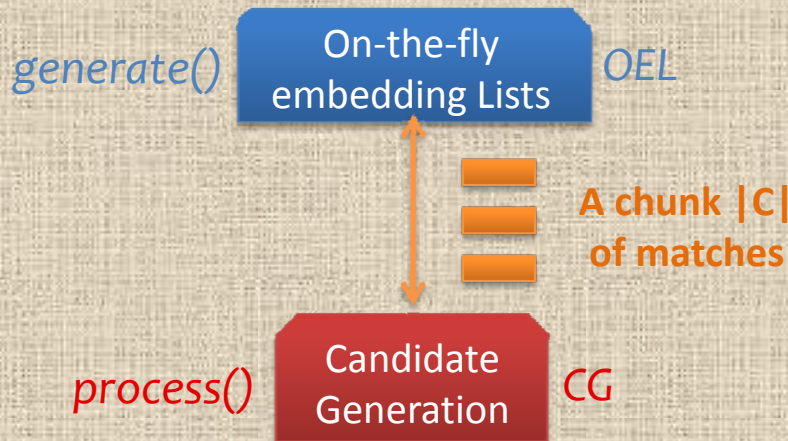
Subsequence matches
(1, 2, 5) (1, 3, 5)

Structure matches
(1, 3, 5)

Memory Optimizations – Chunking

□ Localize the computation in candidate generation

- *Integrate* match generation (in line 3) with match processing (in line 5: pattern growth)



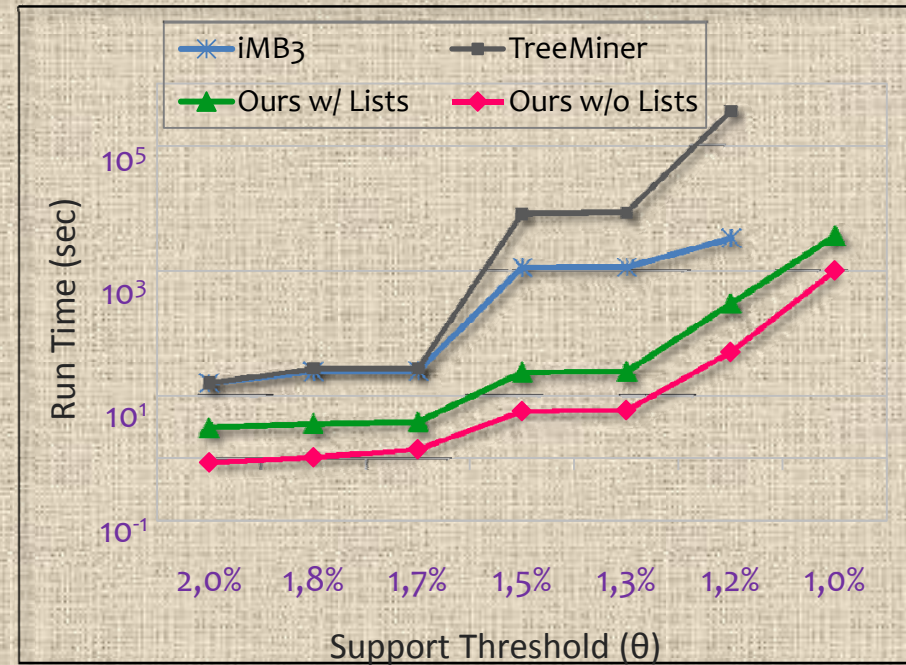
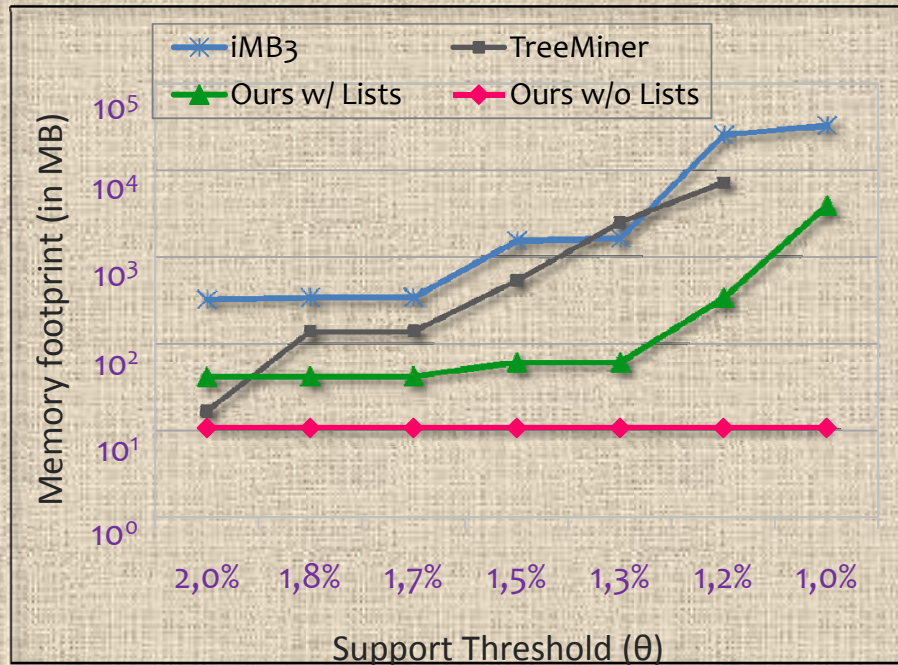
Pattern—growth approach

```
1. Pattern_mine(P)
2. loop
3.   sup = find_frequency (P)
4.   if sup ≥  $\theta$ 
5.     P' = grow P with a new node
6.     Pattern_mine(P')
```

At any time,
only |c| matches
are kept in memory

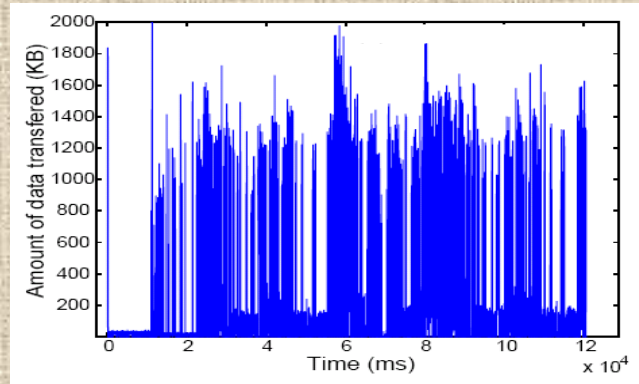
Performance – Memory Usage and Run Time

On cslogs data set (web analytics) ~ 60,000 trees

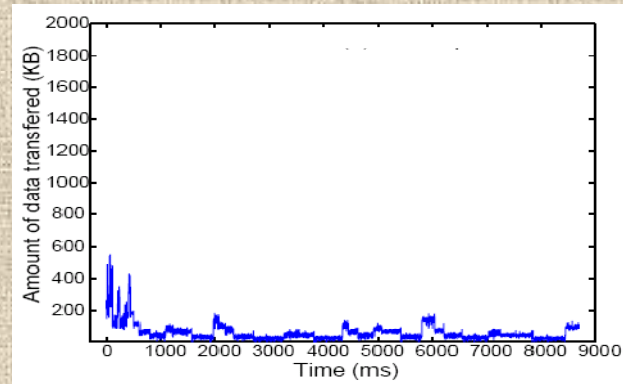


- 1) Memory footprint is *constant* irrespective θ
- 2) Improved locality results in *smaller* run time
- 3) *Two-Three orders* of magnitude improvements in both run time and memory performance

Performance – Memory Bandwidth



Persistent list maintenance
(TreeMiner¹)



No explicit lists
(Ours)

- TreeMiner (list maintenance) aggravates the bandwidth pressure
 - More than **1.2 GB/sec** per core → can not scale with large number of cores
- Memory optimizations make *regular and small-sized accesses*
 - Well below **200 MB/sec** per core (third graph)

Experimentation Method:

- Total execution time is divided into 1 msec slices
- Data transferred over memory bus in each slice $\approx$ L3 line size * L3 cache misses
- Cache misses are measured using PAPI performance toolkit

Outline

□ Introduction

- Problem definition
- Challenges

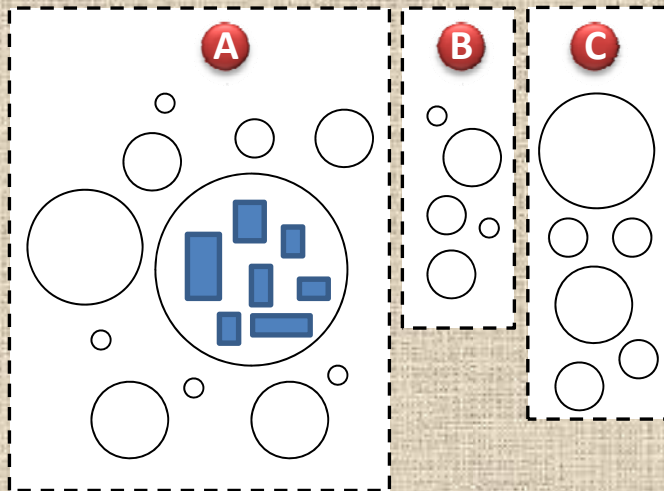
□ Proposed Approach

- Data transformations
- Memory optimizations
 - Sequence-based subtree matching
 - Computation chunking
- **Parallel algorithms**
 - Adaptive load balancing

□ Conclusions

Parallel Algorithms – Challenge

Search Space



- Search space is partitioned into equivalence classes ()
- Each equivalence class contains many patterns ()
- Processing each pattern involves many trees ()

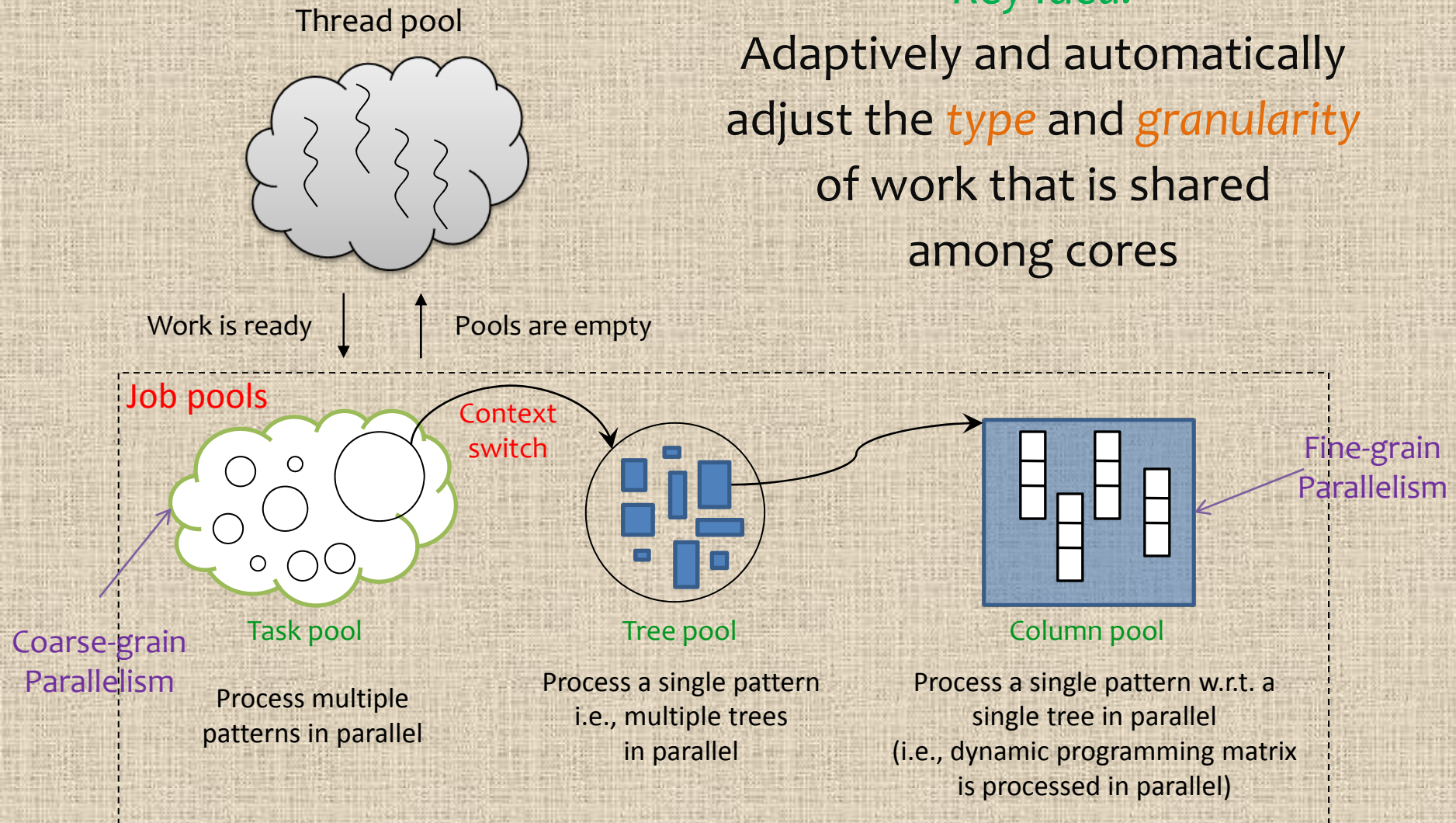
- Workload skew is present at every level
 - One equivalence class may contain more patterns than the other
 - Processing one pattern may be more expensive than the other
 - One tree may be bigger than the other

Challenge: load balancing in the presence of skew

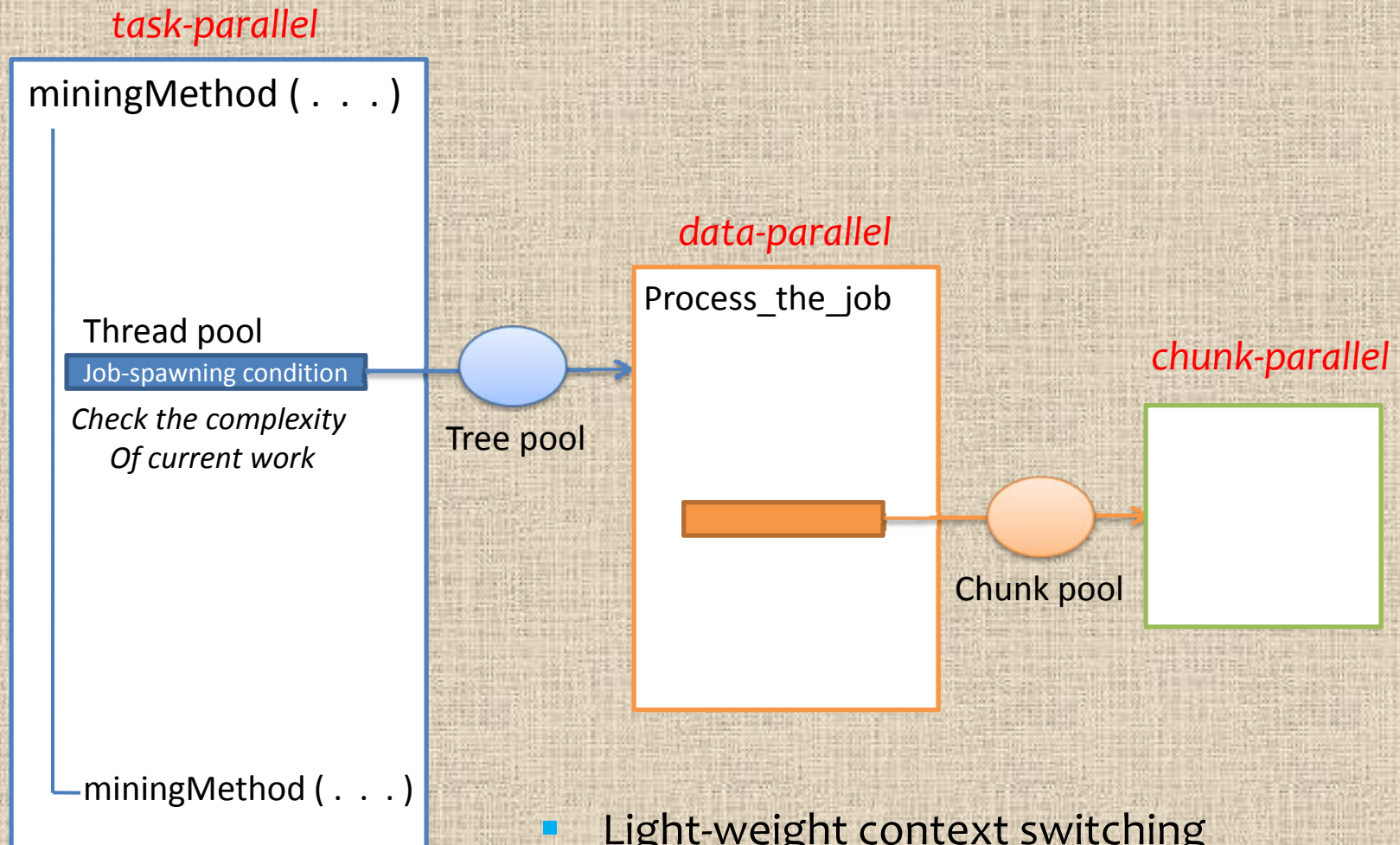
Adaptive Design

Key Idea:

Adaptively and automatically adjust the *type* and *granularity* of work that is shared among cores



Implementation of Parallel Algorithm

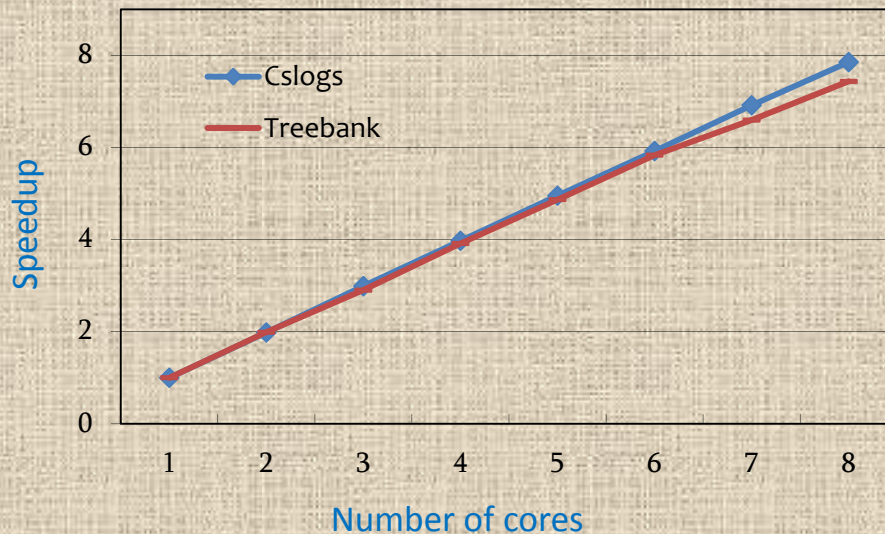


- Light-weight context switching
- General-purpose scheduling service

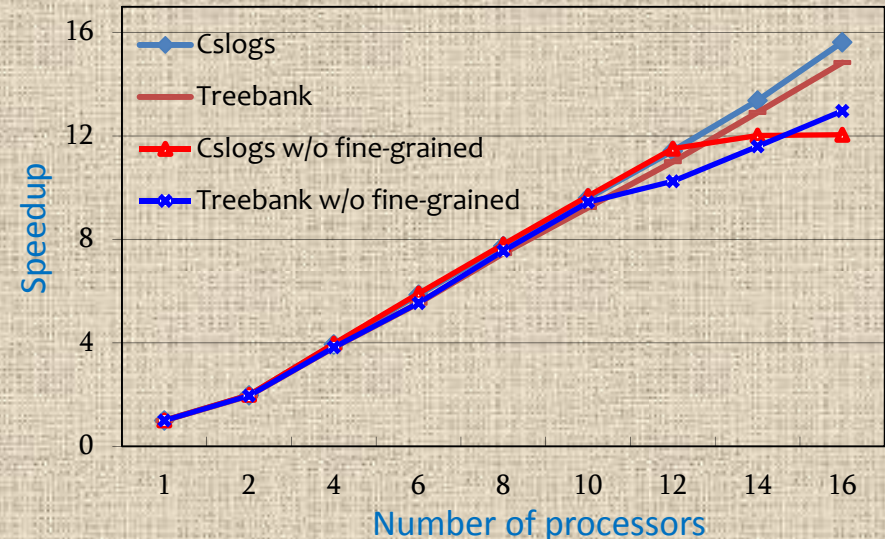
Performance – Parallel Efficiency

On two data sets: **Cslogs** (web analytics) and **treebank** (computational linguistics)

On a dual quad core system



On a SGI Altix - SMP system



- 1) Near-linear speedups
- 2) Need for *fine-grain* parallelism
 - Without which the speedups saturate
- 3) Memory optimizations are critical
 - Without them, the algorithms are not scalable (speedup of 1.7 on 8 processors)

Impact of Fine-grained Parallelism

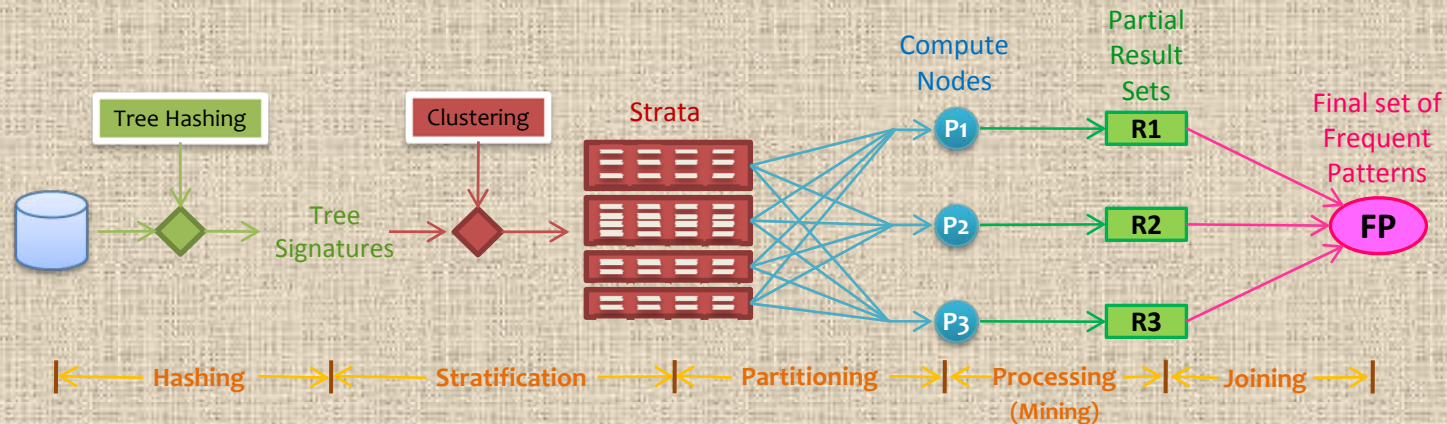
Number of cores					
Granularity	1	2	4	8	16
Task	0	4	7	26	48
Data	0	2	2	10	11
Chunk	0	0	0	9	19

On Treebank dataset @ $\theta = 56\%$

Parallelism at high level of granularity is explored first

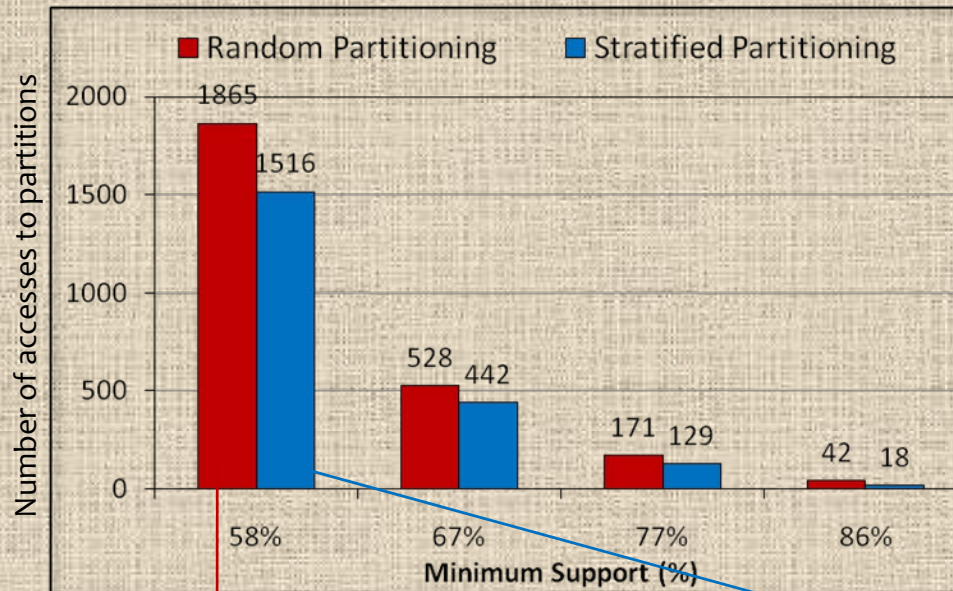
Need for fine-grain parallelism grows as we increase the number of cores

Hash-based Distributed Subtree Mining



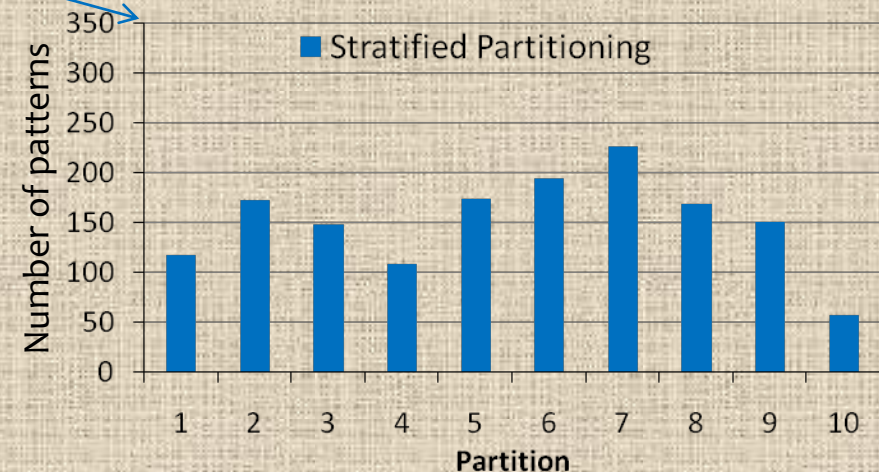
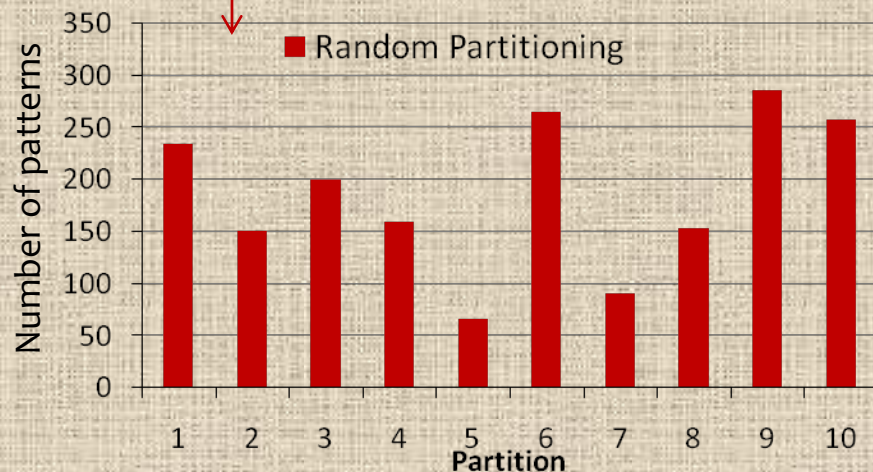
- ❑ Placement enables scalable algorithm deployment
 - Localizes *computation* & minimizes *communication*
 - Ensures *load balance* & provides *fault tolerance*
- ❑ Partitioning schemes affect the *joining* phase
 - False positive patterns must be re-evaluated
 - Cost Metric = # of times partitions need to be accessed while *joining*
- ❑ Stratified sampling mechanisms

Distributed Mining – Evaluation



Stratified strategy

- Distribution of each partition is similar to the data set
- Small number of false positives
- Small number of accesses to partitions
- Variance in number of patterns generated from each partition is small



Take Home Message

- ❑ Trading off memory for redundant computations
 - Benefits
 - Reduces memory footprints & working sets
 - Reduces *bandwidth pressure*
 - Challenges addressed for Tree Mining
 - On the fly embedding lists and computation chunking
 - Reduction in bandwidth pressure (up to 10 times)
 - Overall reduction in memory footprint & sequential time (up to three orders of magnitude)

- ❑ Moldable task partitioning and adaptive scheduling
 - Benefits
 - Better CPU utilization
 - Exploits *fine-grain parallelism*
 - Challenges addressed for Tree Mining
 - Sensing the problem (load imbalance) and leveraging adaptive scheduling strategy
 - Realizing linear speedup over best sequential implementation

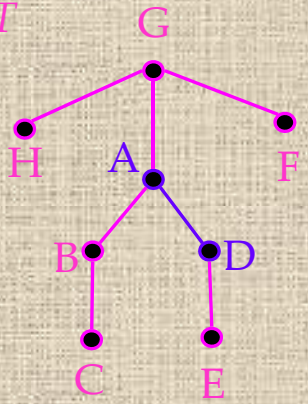
Thank You!

contact: statiko@us.ibm.com

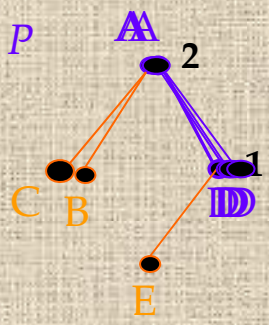
Backup Slides – Tree Mining

Backup Slide

DB Tree, T



Pattern, P



NPS	8	3	6	5	6	8	8	-
LS	H	C	B	E	D	A	F	G
PON	1	2	3	4	5	6	7	8

(E, 1)

(B, 2)

(C, 2)

- ❑ Use *embedding lists* to store matches of P in T
- ❑ Extending P involves only $\{1, 2, 3, 4\}$
- ❑ Connectivity Check
 - See if any of the *ancestor* is part of the embedding
 - If yes, $(label, ancestor PON)$ is an extension point

Complete & non-redundant candidate generation



Tree Mining – Related Work

Type of subtrees	Algorithms
<i>Ordered and Induced</i>	FreqT [19], AMOIT [103]
<i>Ordered and Embedded</i>	Treeminer [222], MB3 [184], iMB3 [185], Chopper and XSpanner [199], PrefixTreeESpan [238], Trips and Tides [191]
<i>Unordered and Induced</i>	uNot [20], uFreqT [150], PathJoin [210], CMTreeMiner [56], HybridTreeMiner [54]
<i>Unordered and Embedded</i>	Treefinder [193], Sleuth [223], Wti-Miner [81]
<i>Free Trees</i>	FreeTreeMiner [171], Gaston [151], Chi <i>et al</i> [55], HybridTreeMiner [54], CFFTree [235]
<i>Closed and Maximal</i>	Treefinder [193], PathJoin [210], CMTreeMiner [55], Dryade [194], DryadeParent [192], CFFTree [235]

Summary – Tree Pattern Mining

- ❑ Leverage modern architectures
- ❑ Rethinking the tradeoff between time and space
 - Trade space for redundant computations
 - Reduces memory footprints and working sets
 - Alleviates *bandwidth pressure*
- ❑ *Adaptive* task partitioning and scheduling
 - Better CPU utilization
 - Tasks must be easy to generate and share
- ❑ Exploiting *fine-grain* parallelism
 - Critical for efficient performance on multicores
- ❑ Hash-based distributed data placement framework
 - Sampling from semi-structured data